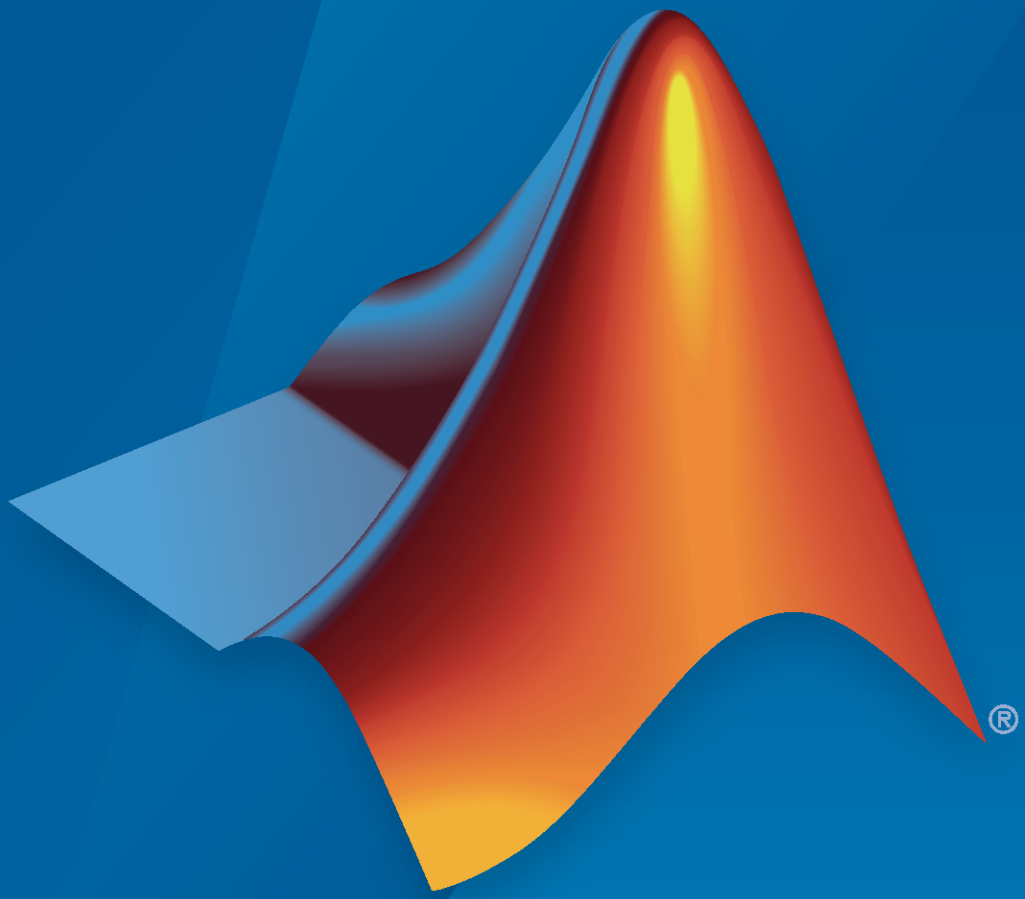
Automated Driving Toolbox™

User's Guide



MATLAB® & SIMULINK®

R2020b

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

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

Phone: 508-647-7000

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

Automated Driving Toolbox™ User's Guide

© COPYRIGHT 2017–2020 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

March 2017	Online only	New for Version 1.0 (Release 2017a)
September 2017	Online only	Revised for Version 1.1 (Release 2017b)
March 2018	Online only	Revised for Version 1.2 (Release 2018a)
September 2018	Online only	Revised for Version 1.3 (Release 2018b)
March 2019	Online only	Revised for Version 2.0 (Release 2019a)
September 2019	Online only	Revised for Version 3.0 (Release 2019b)
March 2020	Online only	Revised for Version 3.1 (Release 2020a)
September 2020	Online only	Revised for Version 3.2 (Release 2020b)

Contents

1 Sensor Configuration and Coordinate System Transformations

2 Ground Truth Labeling and Verification

3 Tracking and Sensor Fusion

4 Planning, Mapping, and Control

5 Cuboid Driving Scenario Simulation

6 3D Simulation – User's Guide

7 Featured Examples

Sensor Configuration and Coordinate System Transformations

Coordinate Systems in Automated Driving Toolbox

Automated Driving Toolbox uses these coordinate systems:

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

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

World Coordinate System

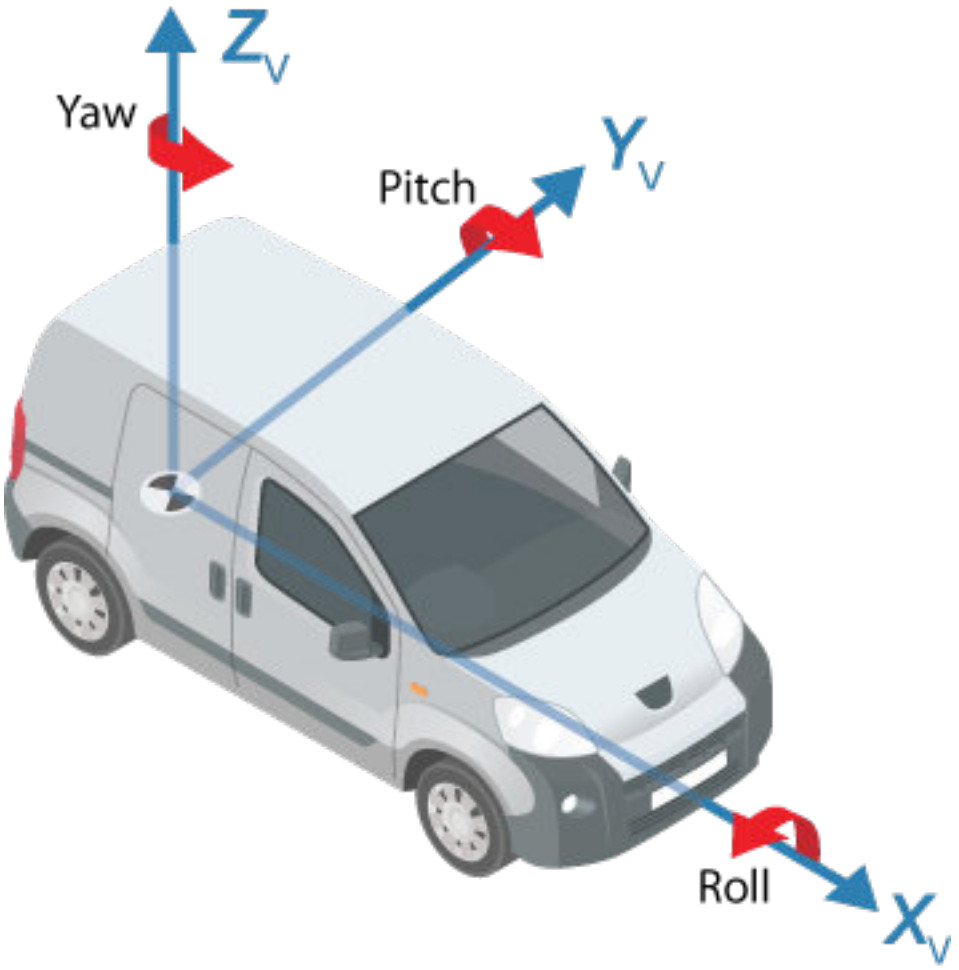
All vehicles, sensors, and their related coordinate systems are placed in the world coordinate system. A world coordinate system is important in global path planning, localization, mapping, and driving scenario simulation. Automated Driving Toolbox uses the right-handed Cartesian world coordinate system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

Vehicle Coordinate System

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

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

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

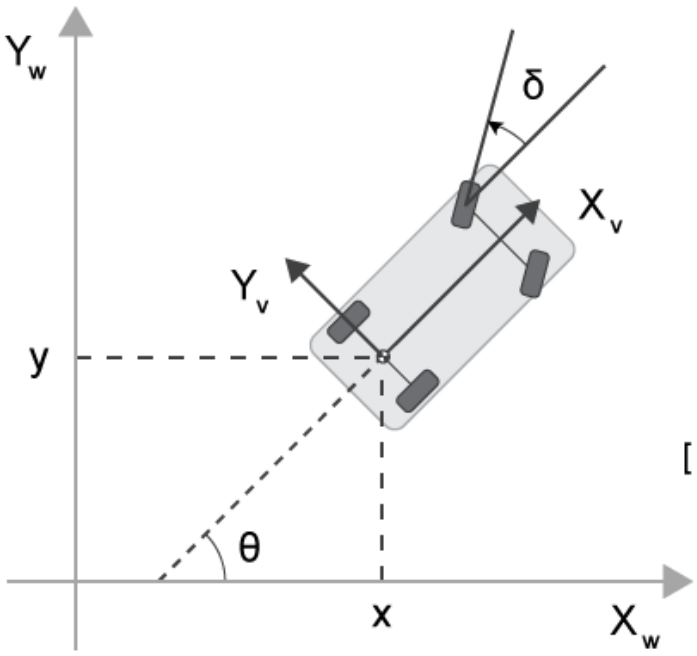


In most Automated Driving Toolbox functionality, such as cuboid driving scenario simulations and visual perception algorithms, the origin of the vehicle coordinate system is on the ground, below the midpoint of the rear axle. In 3D driving scenario simulations, the origin is on ground, below the longitudinal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

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

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

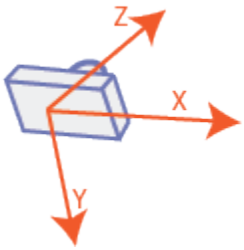
For global path planning, localization, mapping, and driving scenario simulation, the state of the vehicle can be described using the pose of the vehicle. The steering angle of the vehicle is positive in the counterclockwise direction.



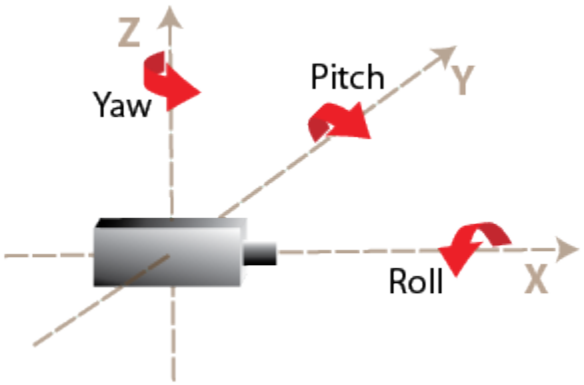
X_v, Y_v Vehicle Coordinate System
X_w, Y_w World Coordinate System
$[x, y, \theta]$ Vehicle Pose
δ Steering Angle

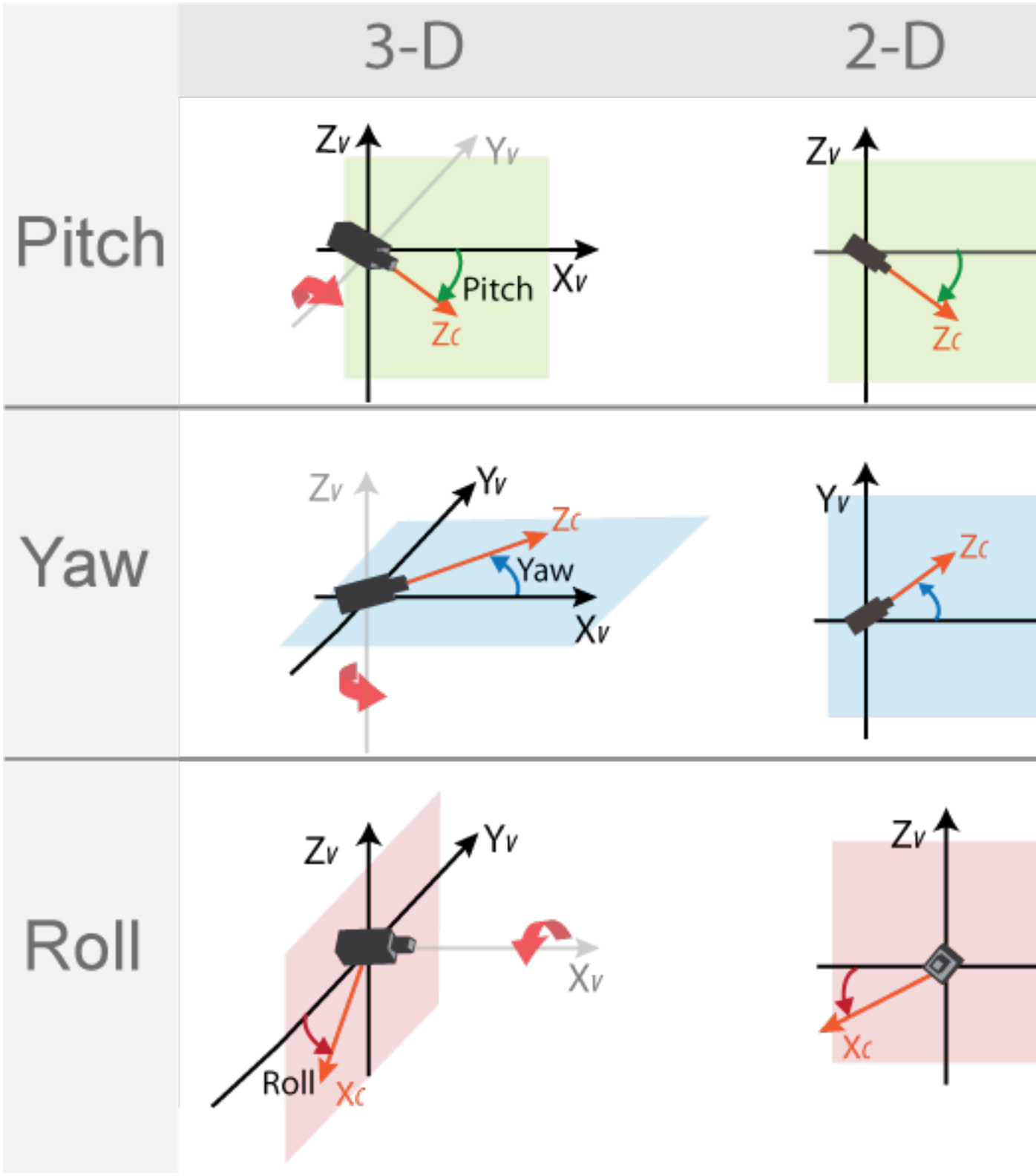
Sensor Coordinate System

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



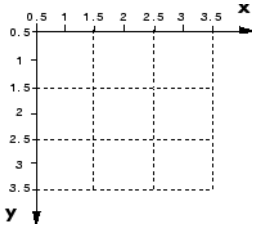
The yaw, pitch, and roll angles of sensors follow an ISO convention. These angles have positive clockwise directions when looking in the positive direction of the *Z*-, *Y*-, and *X*-axes, respectively.


Z
Yaw
Pitch
Y
Roll
X


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

Spatial Coordinate System

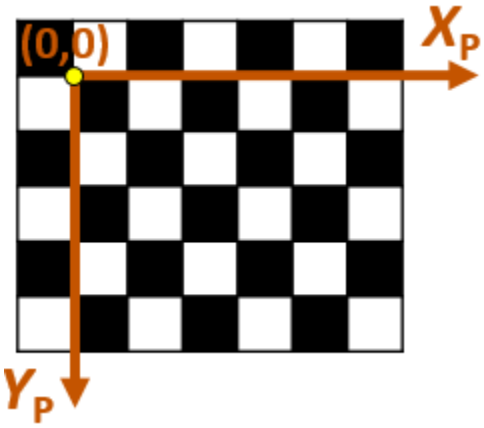
Spatial coordinates enable you to specify a location in an image with greater granularity than pixel coordinates. In the pixel coordinate system, a pixel is treated as a discrete unit, uniquely identified by an integer row and column pair, such as (`3,4`). In the spatial coordinate system, locations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



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

Pattern Coordinate System

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



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

See Also

More About

- “Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox” on page 6-10

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

Calibrate a Monocular Camera

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

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

Estimate Intrinsic Parameters

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

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

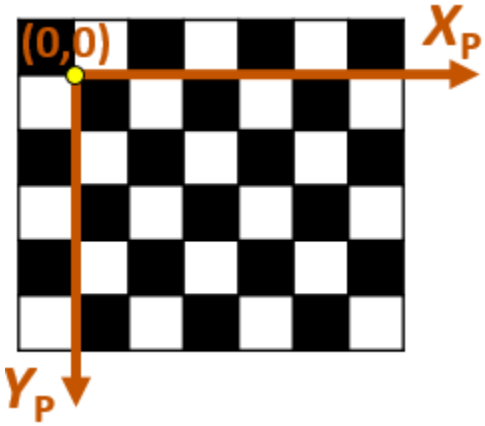
Alternatively, to better visualize the results, use the **Camera Calibrator** app. For information on setting up the camera, preparing the checkerboard pattern, and calibration techniques, see “Single Camera Calibrator App” (Computer Vision Toolbox).

Place Checkerboard for Extrinsic Parameter Estimation

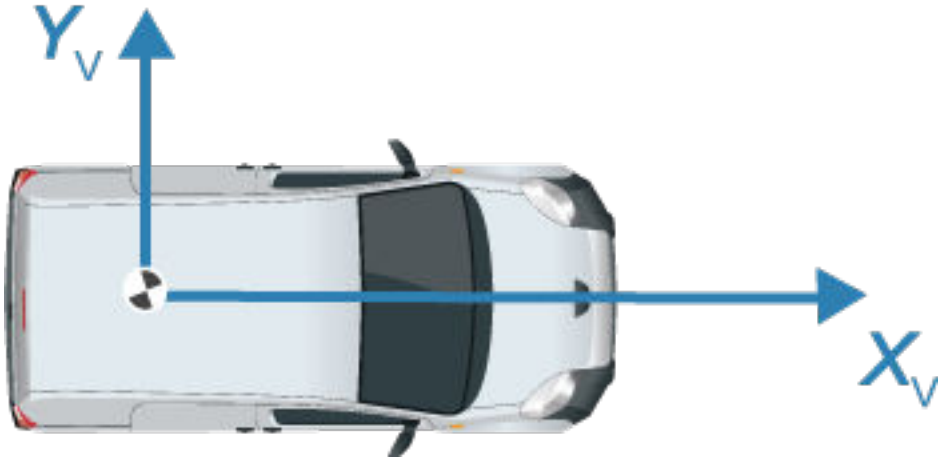
For a monocular camera mounted on a vehicle, the *extrinsic parameters* define the mounting position of that camera. These parameters include the rotation angles of the camera with respect to the vehicle coordinate system, and the height of the camera above the ground.

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

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



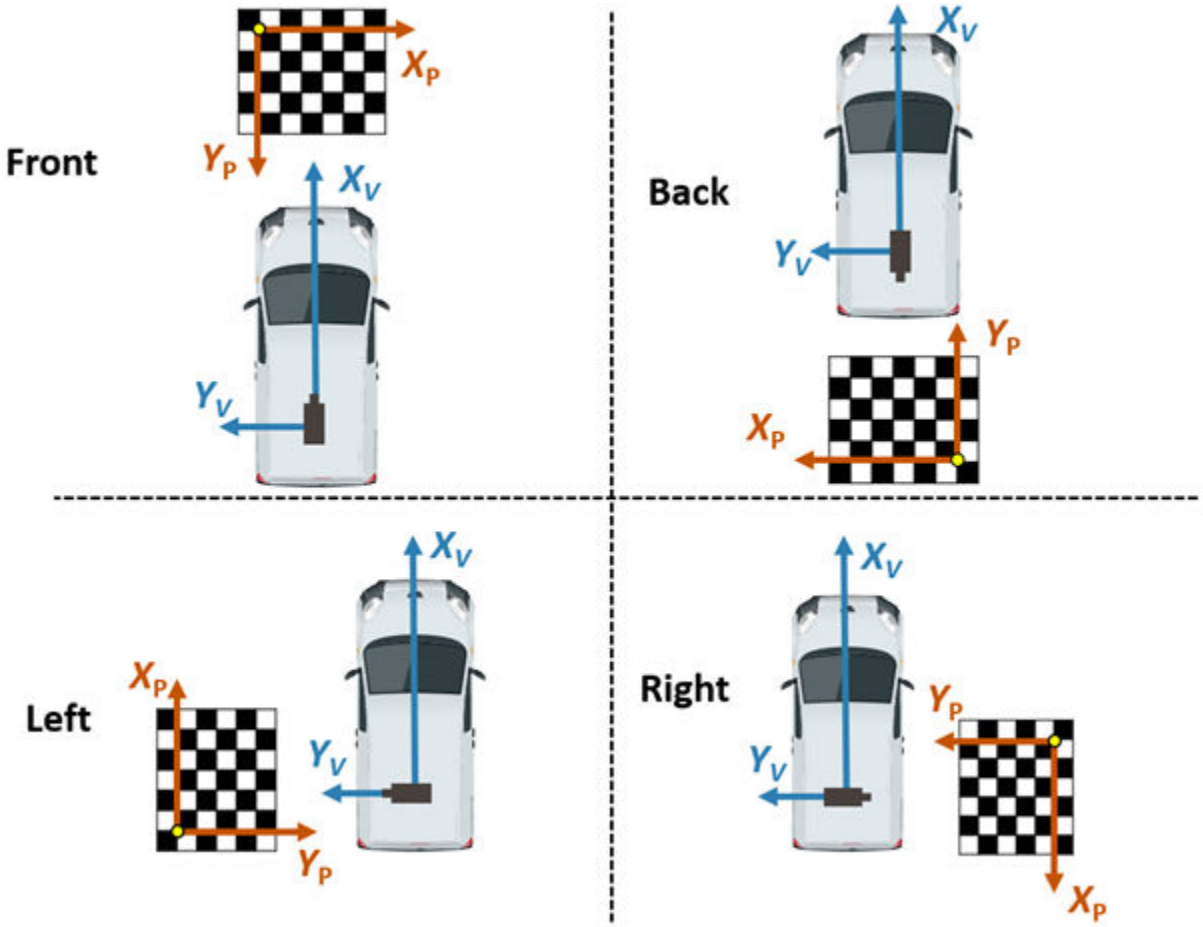
When placing the checkerboard pattern in relation to the vehicle, the X_P- and Y_P-axes must align with the X_V- and Y_V-axes of the vehicle. In the vehicle coordinate system, the X_V-axis points forward from the vehicle and the Y_V-axis points to the left, as viewed when facing forward. The origin is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



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

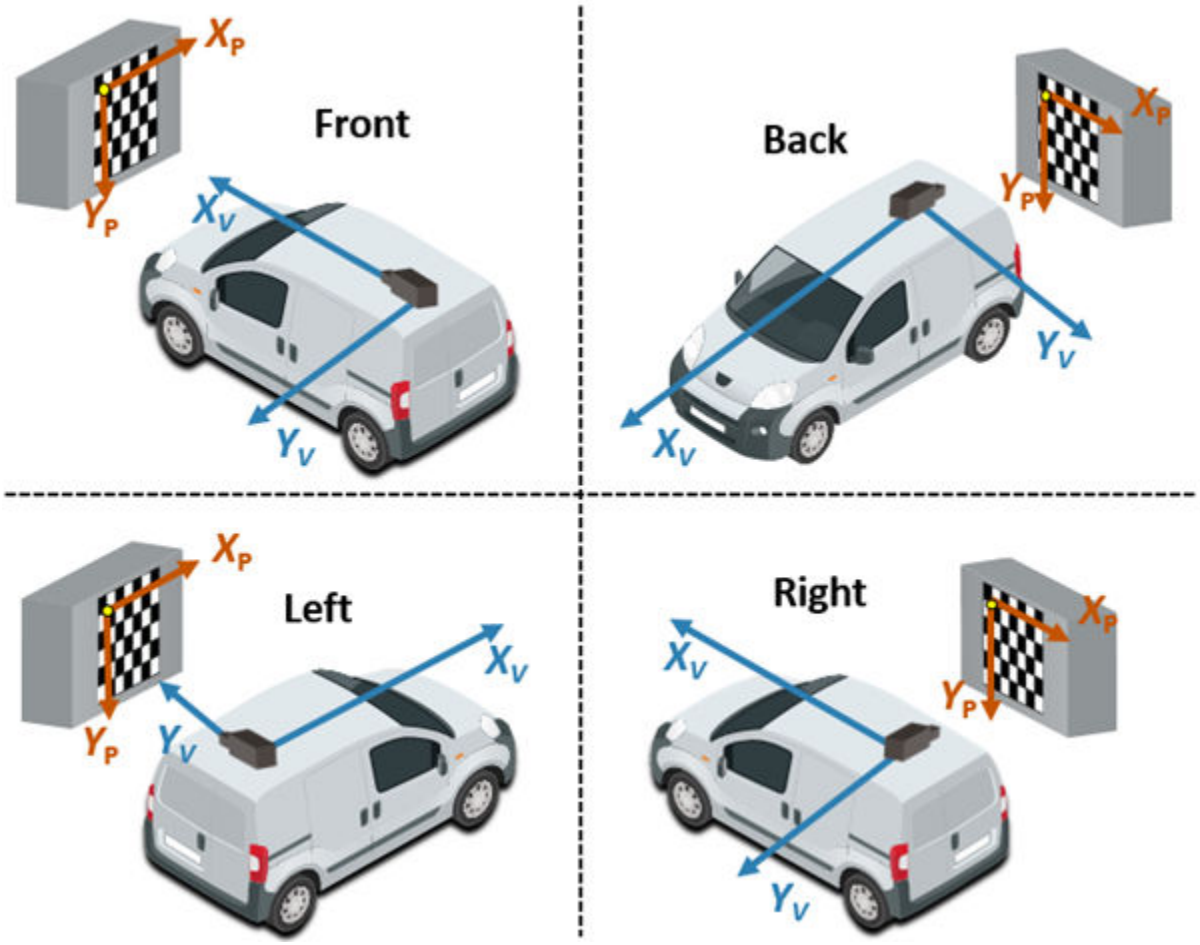
Horizontal Orientation

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



Vertical Orientation

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



Estimate Extrinsic Parameters

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

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

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

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

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

Configure Camera Using Intrinsic and Extrinsic Parameters

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

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

See Also

Apps
Camera Calibrator

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

Objects
`monoCamera`

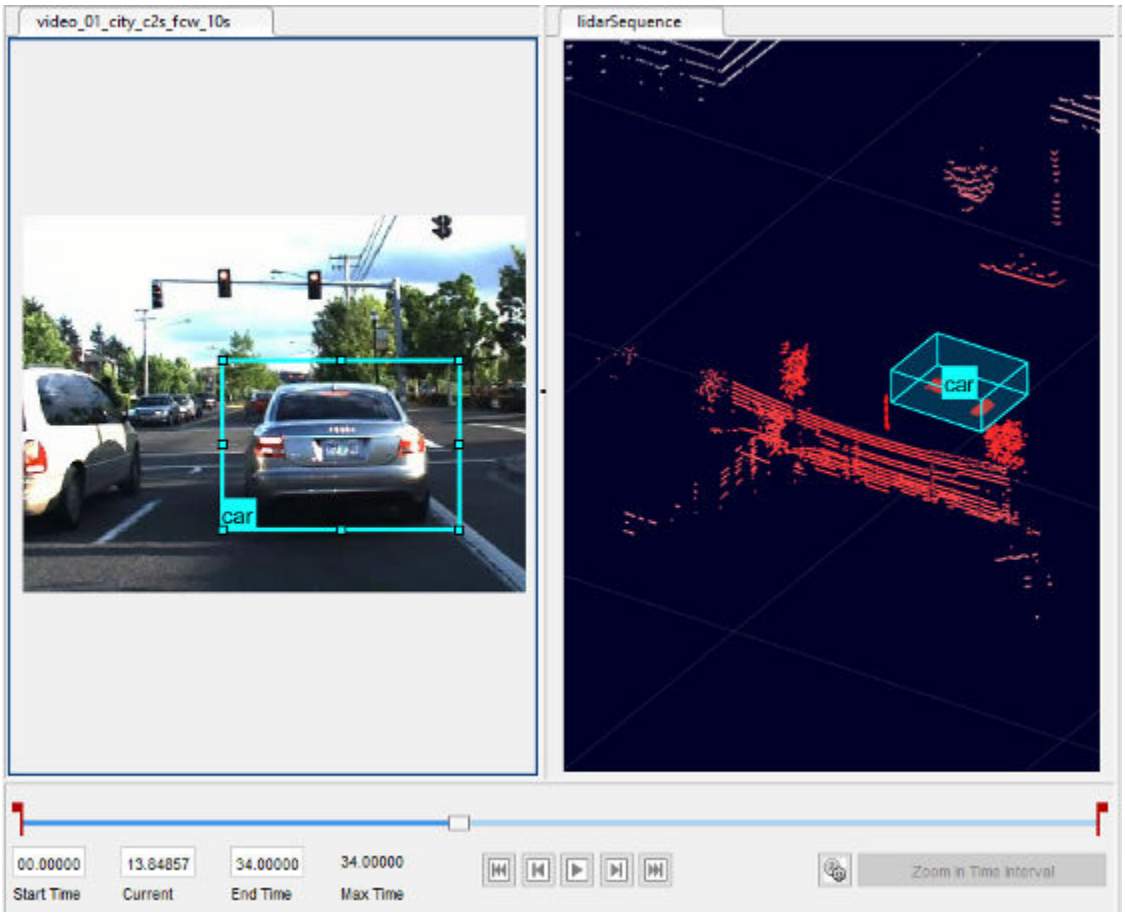
More About

- “Coordinate Systems in Automated Driving Toolbox” on page 1-2
- “Configure Monocular Fisheye Camera” on page 7-3
- “Single Camera Calibrator App” (Computer Vision Toolbox)
- “Fisheye Calibration Basics” (Computer Vision Toolbox)

Ground Truth Labeling and Verification

Get Started with the Ground Truth Labeler

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



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

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

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

See Also

More About

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

Load Ground Truth Signals to Label

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

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

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

Load Timestamps

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

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

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

Open Ground Truth Labeler App

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

```
groundTruthLabeler
```

The app opens to an empty session.

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

Load Signals from Data Sources

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

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

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

Load Video

Load the video into the app.

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

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

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

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

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

Load Point Cloud Sequence

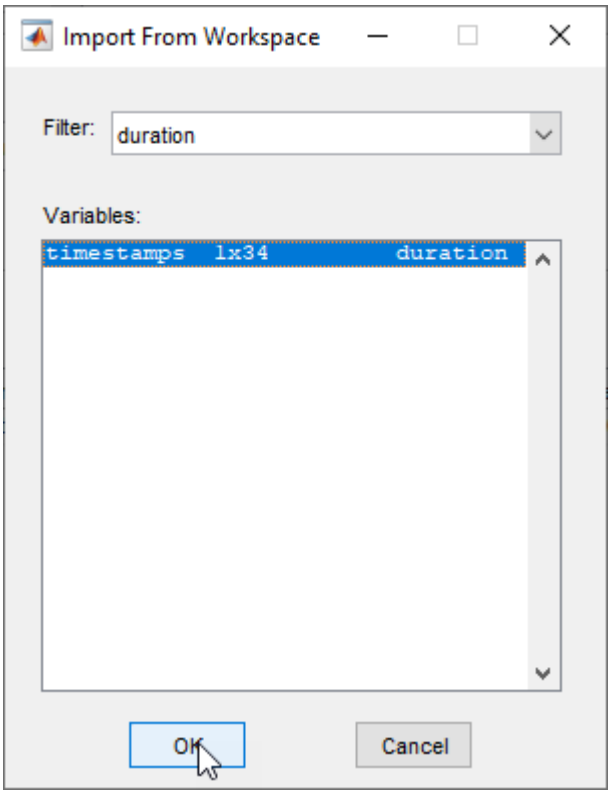
Load the point cloud sequence into the app.

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

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

   ```
   <matlabroot>\toolbox\driving\drivingdata\lidarSequence
   ```

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



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

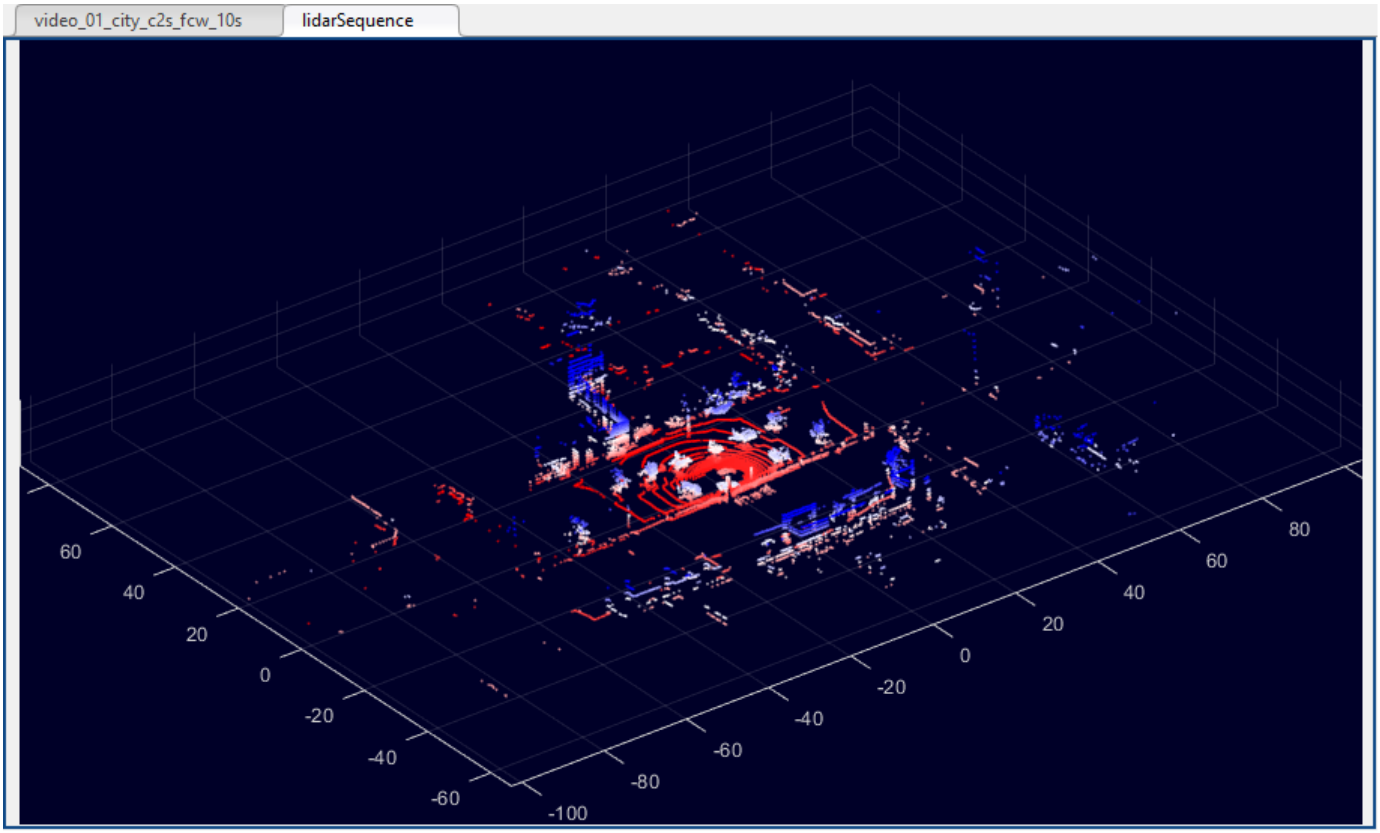
Verify Information About Loaded Signals

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

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

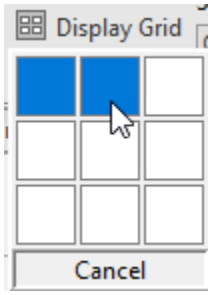
For the point cloud sequence, if you left **Timestamps** set to `Use Default`, then the **Time Range** value for the sequence ranges from 0 to 33 seconds. This range is based on the 34 PCD files in the folder. By default, the app sets the timestamps of a point cloud sequence to a `duration` vector from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

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

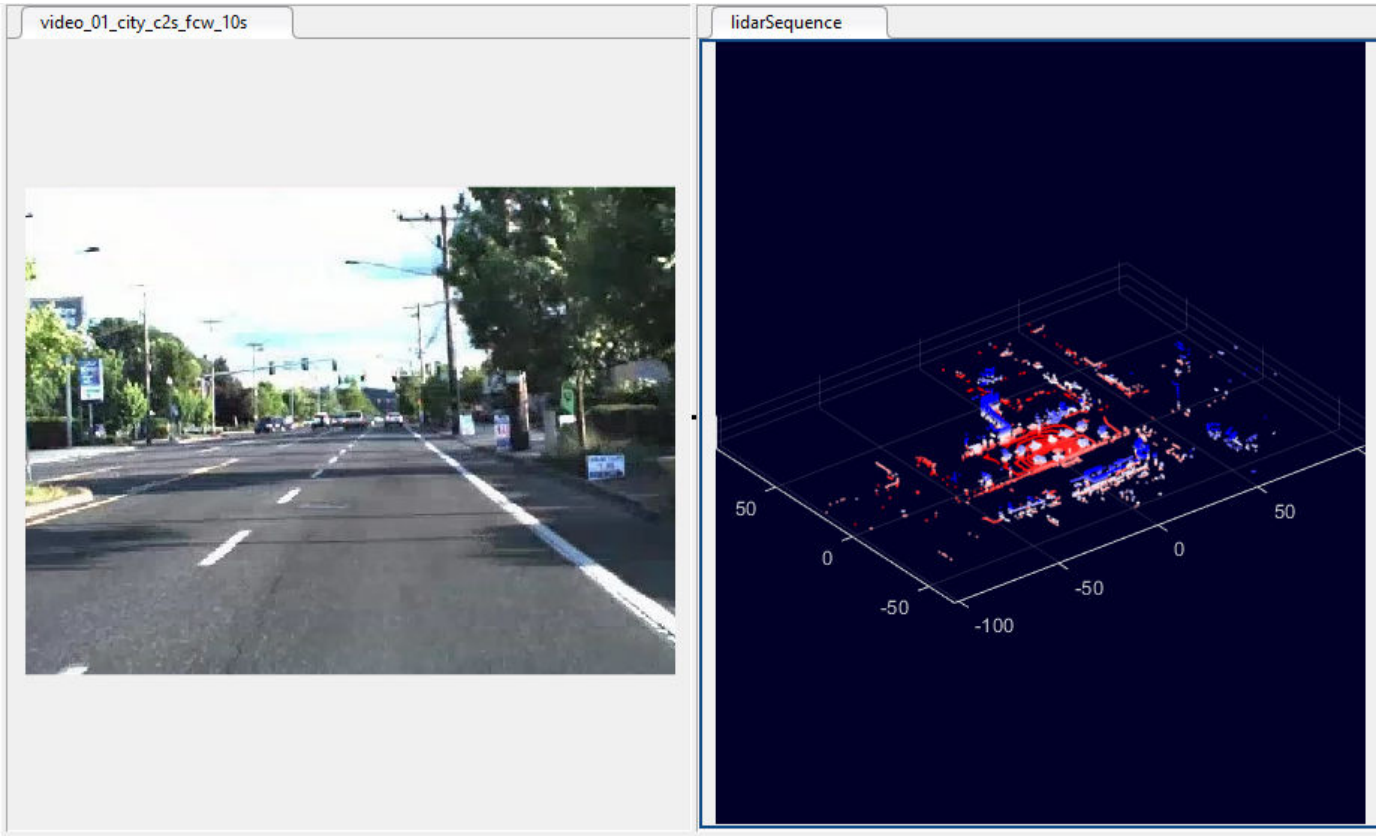


Configure Signal Display

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



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



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

By default, the app plays all frames from the signal with the highest frame rate. This signal is called the master signal. For all other signals, the app displays the frame that is time-aligned with the currently displaying frame of the master signal. To configure which signal is the master signal, use the options in the Playback Control Settings dialog box. To open this dialog box, below the slider, click the clock settings button . For more details about using these options to control the display of signal frames, see “Control Playback of Signal Frames for Labeling” on page 2-35.

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

See Also

More About

- “Sources vs. Signals in Ground Truth Labeling” on page 2-28
- “Control Playback of Signal Frames for Labeling” on page 2-35

Label Ground Truth for Multiple Signals

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

Create Label Definitions

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

Create ROI Label

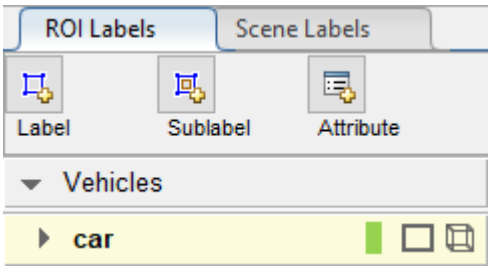
An ROI label is a label that corresponds to a region of interest (ROI) in a signal frame. You can define these ROI label types.

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

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

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

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



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

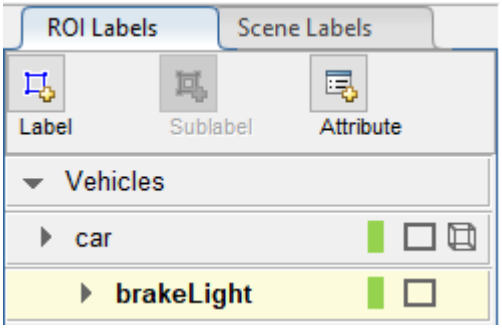
Create ROI Sublabel

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

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

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

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



Create ROI Attribute

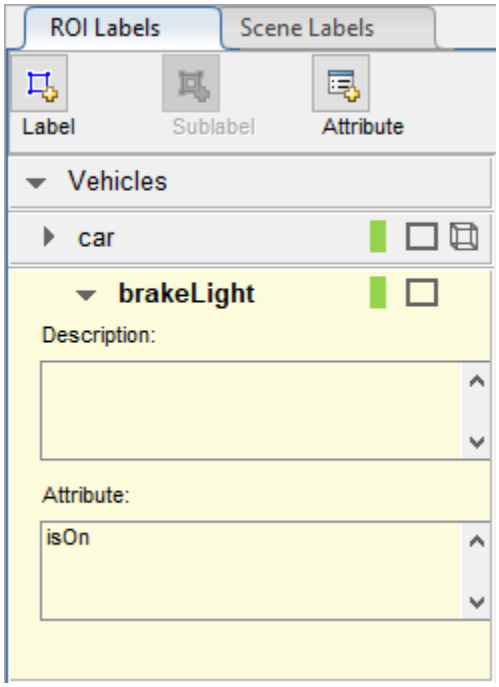
An ROI attribute specifies additional information about an ROI label or sublabel. For example, in a driving scene, attributes can include the type or color of a vehicle. You can define ROI attributes of these types.

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

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

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

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

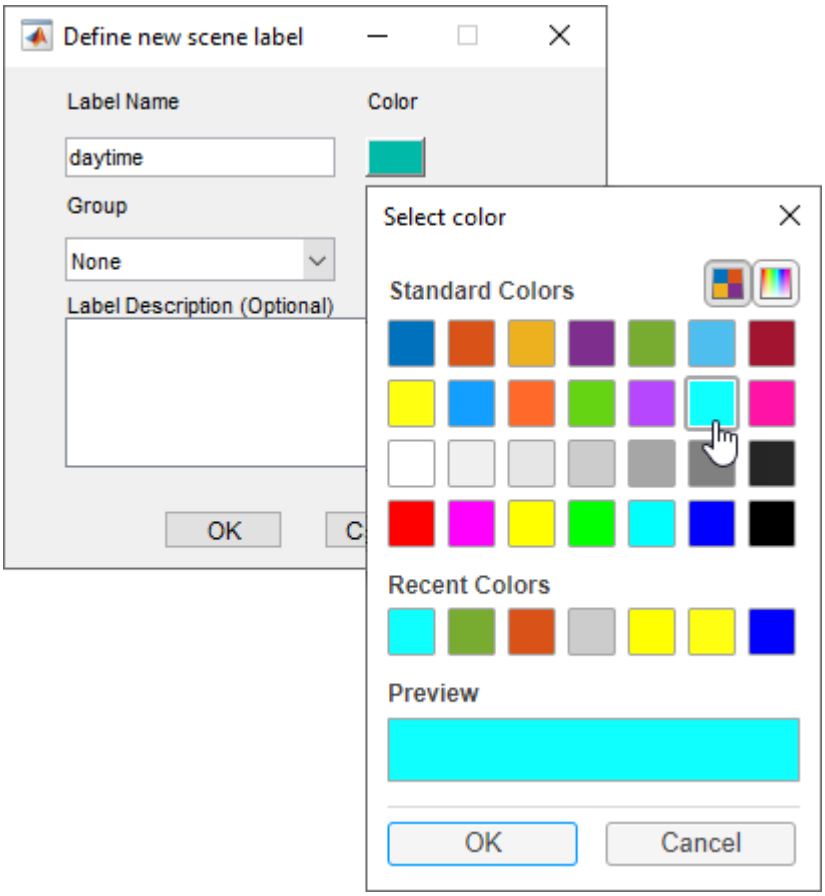


Create Scene Label

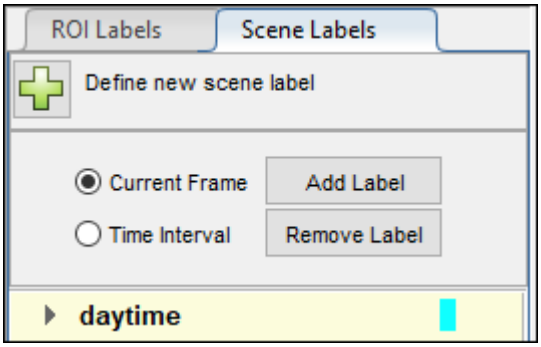
A scene label defines additional information across all signals in a scene. Use scene labels to describe conditions, such as lighting and weather, or events, such as lane changes.

Create a scene label to apply to the signal frames.

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



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



Verify Label Definitions

Verify that your label definitions have this setup.

1 The **ROI Labels** tab contains a **Vehicles** group with a **car** label of type `Rectangle/Cuboid`.
2 The **car** label contains a sublabel named **brakeLight**.
3 The **brakeLight** sublabel contains an attribute named **isOn**.
4 The **Scene Labels** tab contains a light blue scene label named **daytime**.

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

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

Label Video Using Automation

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

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

   a. In the **Current** box, type `8` and press **Enter**.

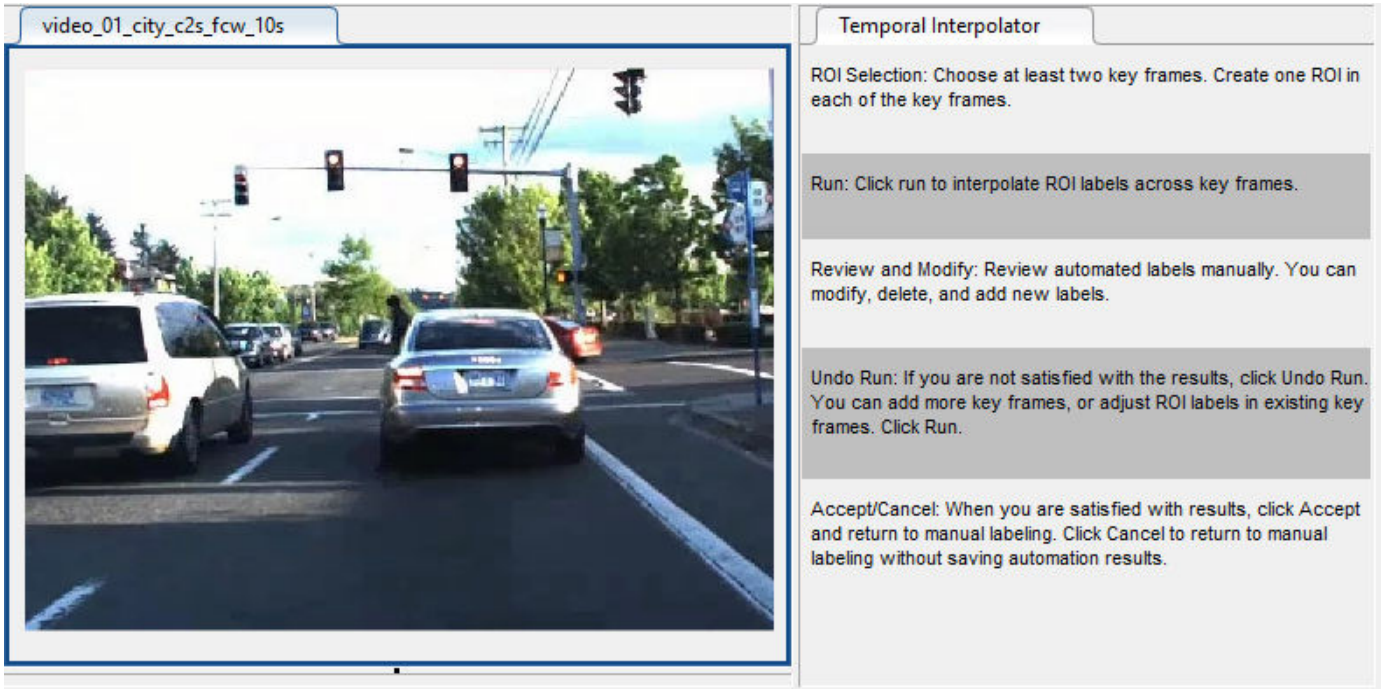
   b. In the **Start Time** box, type `8` so that the slider is at the start of the time interval.

   c. In the **End Time** box, type `10`.

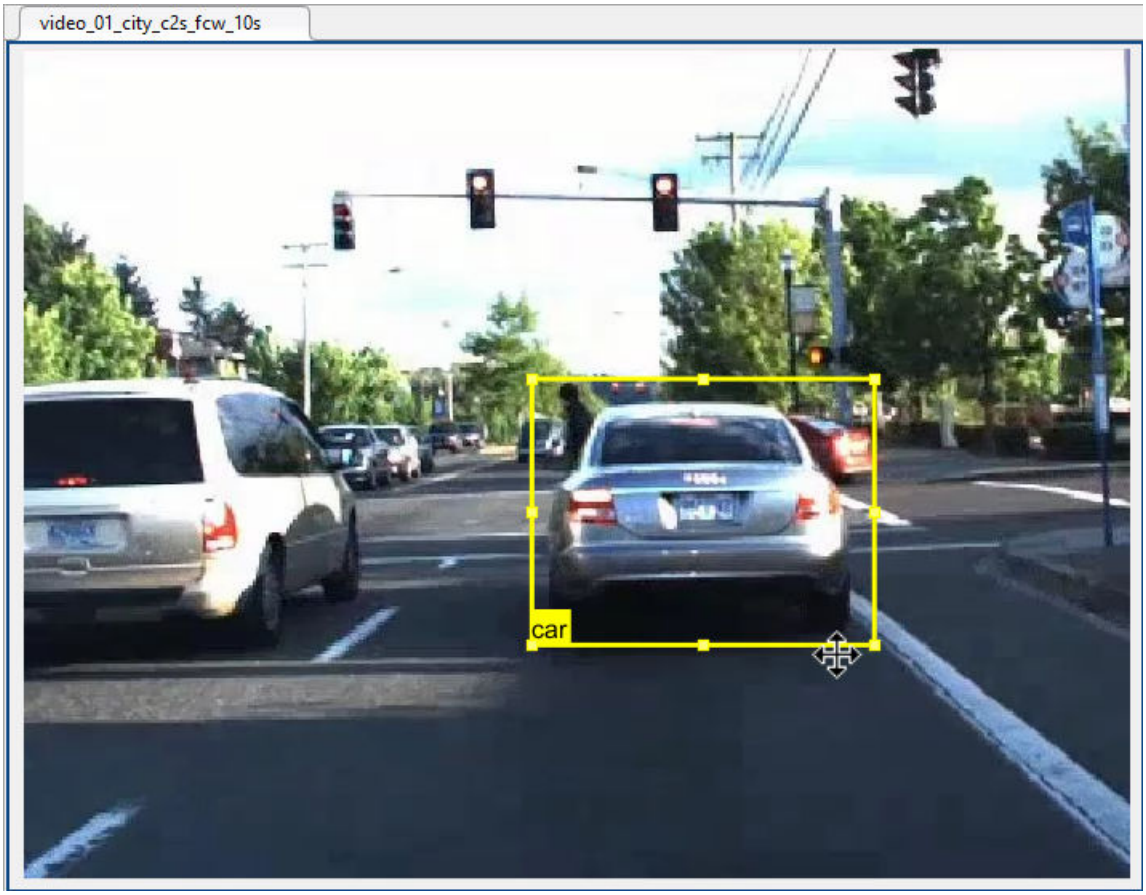
   The range slider and text boxes are set to this 8–10 second interval. The red flags indicate the start and end of the interval.

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

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



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



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

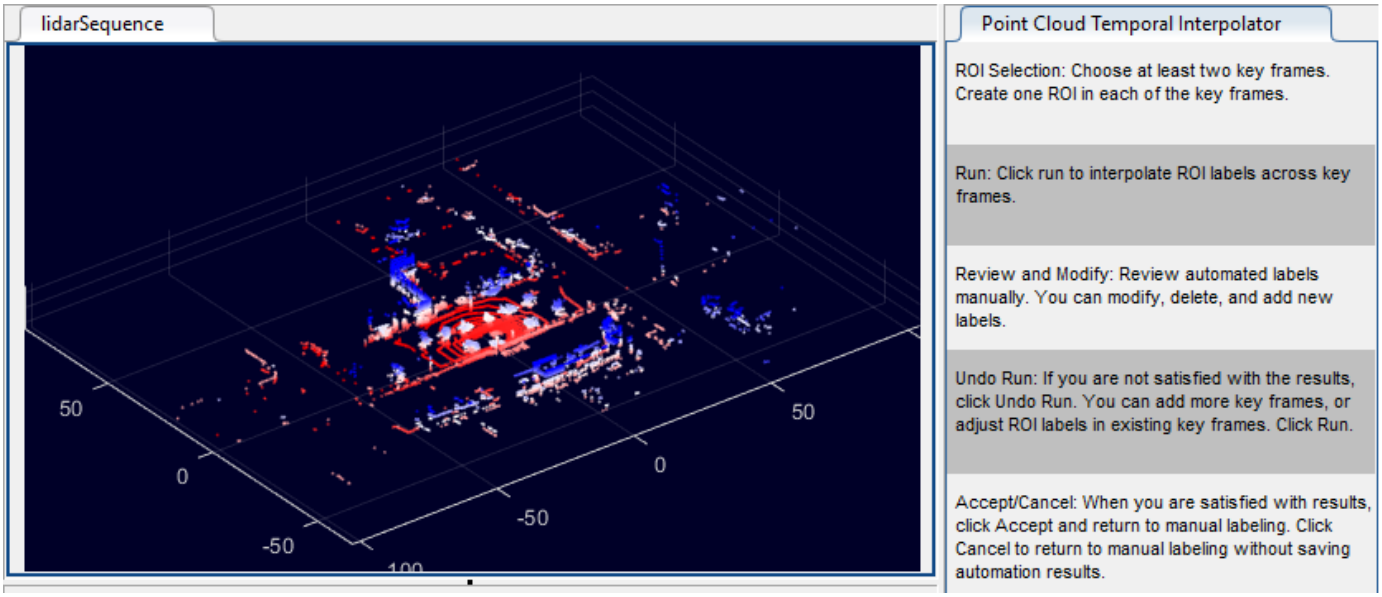
7. Drag the slider to the last frame and draw a **car** label around the same car in this frame. Optionally, to improve automation results, label the car in intermediate frames.
8. Click **Run**. The automation algorithm applies the **car** label to the intermediate frames. Drag the slider to view the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

Label Point Cloud Sequence Using Automation

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

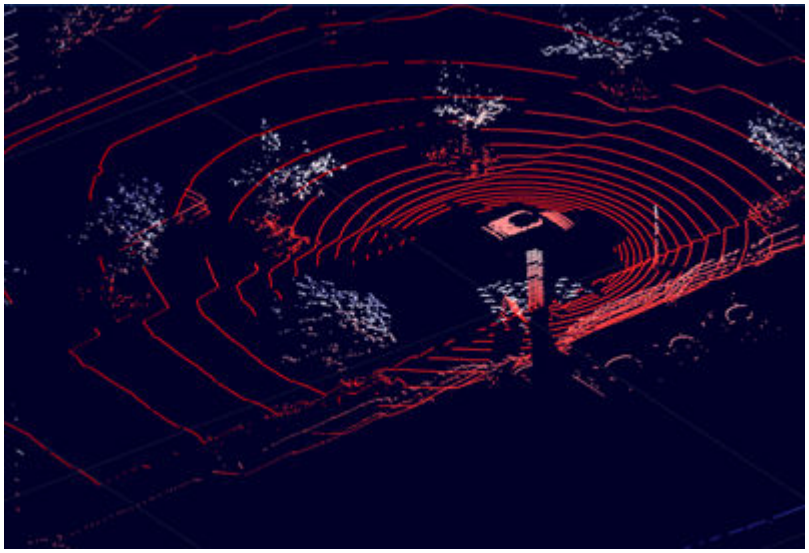
1. At the bottom of the app, verify that the time range is still set to 8 to 10 seconds.
2. In the labeling window, click the point cloud sequence to select it.
3. In the **ROI Labels** tab, click the **car** label definition.
4. On the **Label** tab of the app toolstrip, select **Select Algorithm**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

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

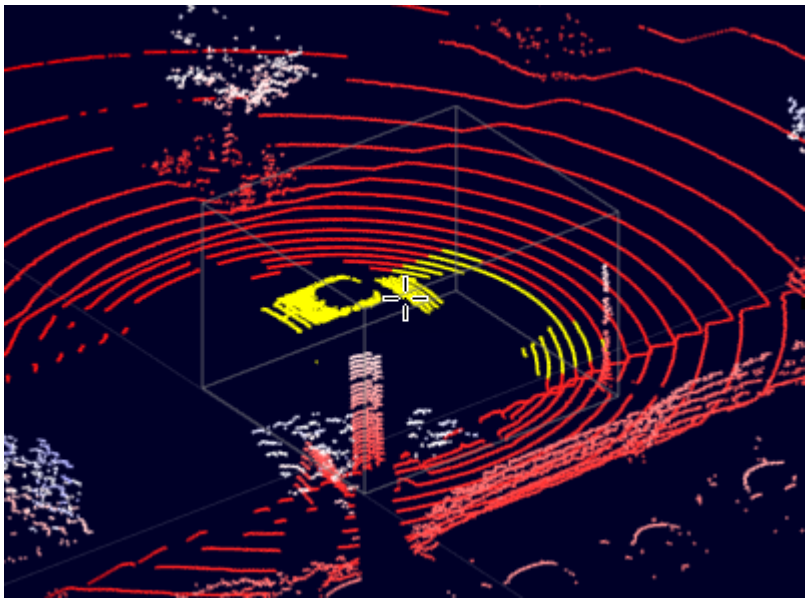


6 At the start of the time interval, draw a **car** label around the ego vehicle.

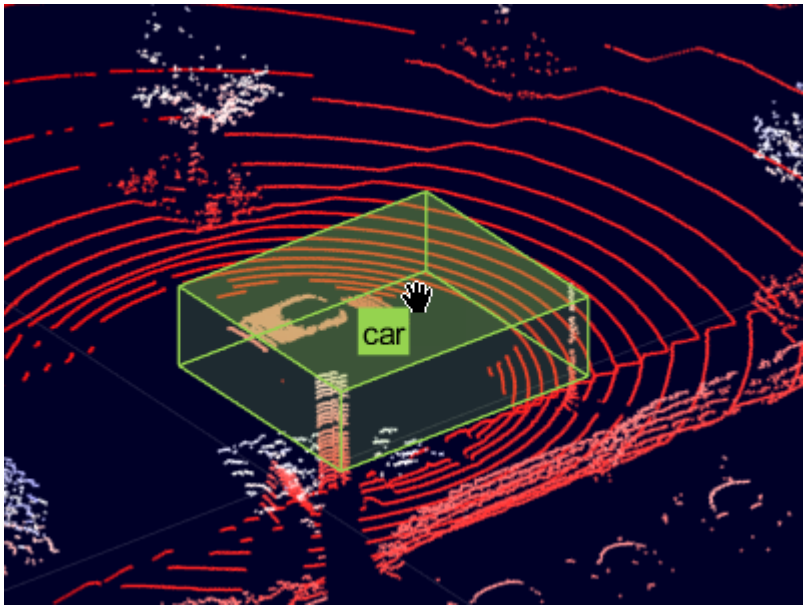
a Zoom in on the car, using either the scroll wheel or the Zoom In button at the top-right corner of the frame. You can also use the Pan button to center the car in the frame.



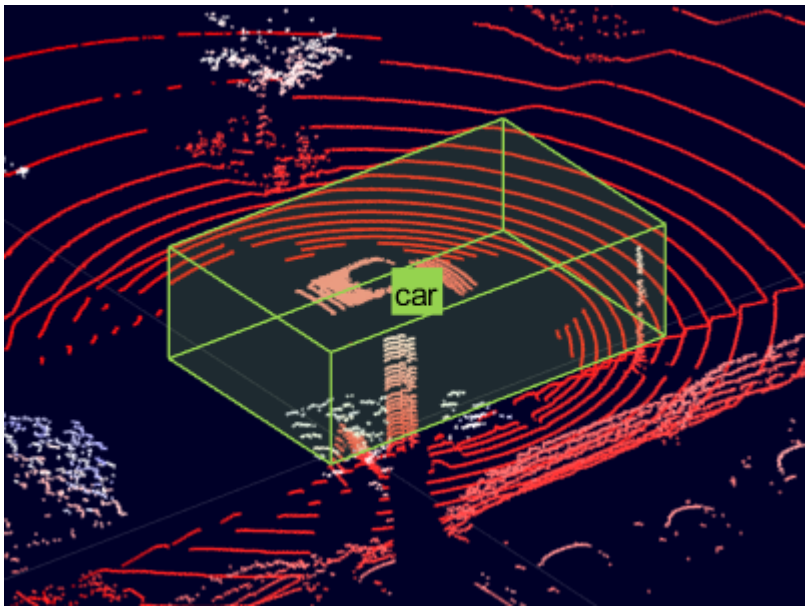
b On the **ROI Labels** tab in the left pane, click the **car** label. Drag the gray preview cuboid until it highlights the ego vehicle.



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



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



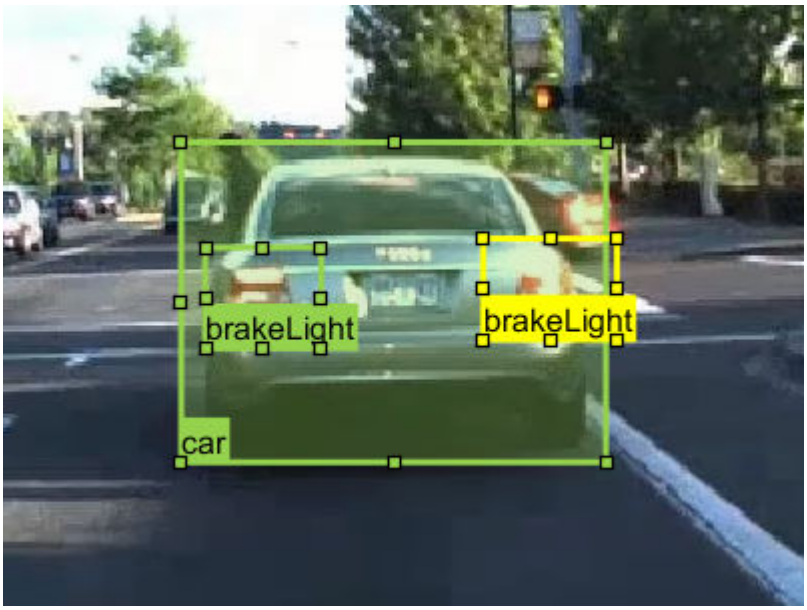
For additional tips and techniques for labeling point clouds, see “Label Lidar Point Clouds for Object Detection” on page 2-39.

7. Click the cuboid and press **Ctrl+C** to copy it. Then, drag the slider to the last frame and paste (**Ctrl+V**) the cuboid into the new frame at the same position. Optionally, to improve automation results, manually adjust the position of the copied label.
8. Click **Run**. The automation algorithm applies the **car** label to the intermediate frames. Drag the slider to view the results. If necessary, manually adjust the labels to improve their accuracy.
9. When you are satisfied with the results, click **Accept** to close the session and apply the labels to this portion of the point cloud sequence.

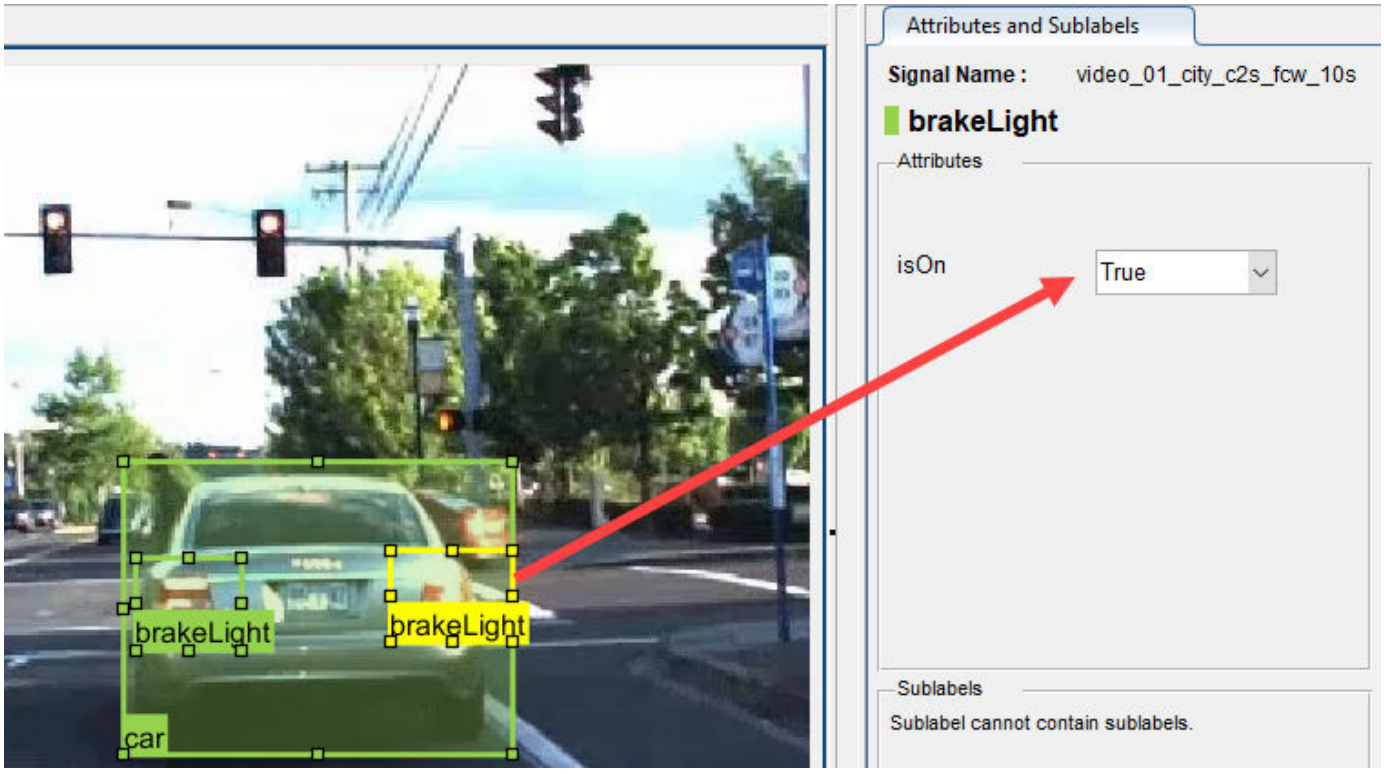
Label with Sublabels and Attributes Manually

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

1. At the bottom of the app, verify that the time range is still set to 8 to 10 seconds. If necessary, drag the slider to the first frame of the time range.
2. In the **ROI Labels** tab, click the **brakeLight** sublabel definition to select it.
3. Hide the point cloud sequence. On the **Label** tab of the app toolstrip, under **Show/Hide Signals**, clear the check mark for the lidar point cloud sequence. Hiding a signal only hides the display. The app maintains the labels for hidden signals, and you can still export them.
4. Expand the video signal to fill the entire labeling window.
5. In the video frame, select the drawn **car** label. The label turns yellow. You must select the **car** label (parent ROI) before you can add a sublabel to it.
6. Draw **brakeLight** sublabels for the car. Optionally, set **Show ROI Labels** to `Always` so that you can confirm the association between the **car** label and its sublabels.



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

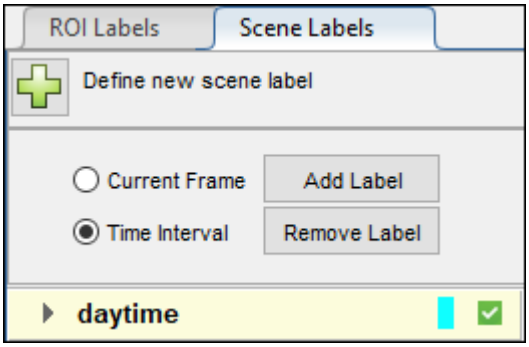


For more details about working with sublabels and attributes, see “Use Sublabels and Attributes to Label Ground Truth Data” (Computer Vision Toolbox).

Label Scene Manually

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

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



View Label Summary

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

Save App Session

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

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

See Also

More About

- “Label Lidar Point Clouds for Object Detection” on page 2-39
- “Label Pixels for Semantic Segmentation” (Computer Vision Toolbox)
- “Keyboard Shortcuts and Mouse Actions for Ground Truth Labeler” on page 2-30
- “View Summary of Ground Truth Labels” (Computer Vision Toolbox)

Export and Explore Ground Truth Labels for Multiple Signals

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

Setup

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

```
groundTruthLabeler groundTruthLabelingSession.mat
```

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

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

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

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

```
gTruth
```

```
gTruth =

  groundTruthMultisignal with properties:

         DataSource: [1x2 vision.labeler.loading.MultiSignalSource]
    LabelDefinitions: [3x7 table]
        ROILabelData: [1x1 vision.labeler.labeldata.ROILabelData]
      SceneLabelData: [1x1 vision.labeler.labeldata.SceneLabelData]
```

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

Data Sources

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

```
gTruth.DataSource
```

```
ans =
```

```
  1x2 heterogeneous MultiSignalSource (VideoSource, PointCloudSequenceSource) array with properti

    SourceName
    SourceParams
    SignalName
    SignalType
    Timestamp
    NumSignals
```

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

```
gTruth.DataSource.SignalName
```

```
ans =

    "video_01_city_c2s_fcw_10s"


ans =

    "lidarSequence"
```

Label Definitions

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

```
gTruth.LabelDefinitions
```

```
ans =

  3x7 table

       Name        SignalType    LabelType       Group        Description     LabelColor      Hie
    ___________    __________    _________    _____________    ___________    _____________    ___

    {'car'    }    Image         Rectangle    {'Vehicles'}    {0x0 char}     {1x3 double}    {1x
    {'car'    }    PointCloud    Cuboid       {'Vehicles'}    {0x0 char}     {1x3 double}    {1x
    {'daytime'}    Time          Scene        {'None'    }    {0x0 char}     {1x3 double}    {0x
```

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

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

```
ans =
```

```
  struct with fields:

     brakeLight: [1x1 struct]
           Type: Rectangle
    Description: ''
```

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

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

```
ans =

  struct with fields:

           Type: Rectangle
    Description: ''
     LabelColor: [0.5862 0.8276 0.3103]
           isOn: [1x1 struct]
```

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

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

```
ans =

  struct with fields:

    DefaultValue: []
     Description: ''
```

ROI Label Data

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

```
gTruth.ROILabelData
```

```
ans =

  ROILabelData with properties:

    video_01_city_c2s_fcw_10s: [204x1 timetable]
                lidarSequence: [34x1 timetable]
```

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

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

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

```
videoLabels =

  10x1 timetable

      Time          car
    ________    ____________

    8 sec       {1x1 struct}
    8.05 sec    {1x1 struct}
    8.1 sec     {1x1 struct}
    8.15 sec    {1x1 struct}
    8.2 sec     {1x1 struct}
    8.25 sec    {1x1 struct}
    8.3 sec     {1x1 struct}
    8.35 sec    {1x1 struct}
    8.4 sec     {1x1 struct}
    8.45 sec    {1x1 struct}


lidarLabels =

  2x1 timetable

       Time           car
    __________    ____________

    8.0495 sec    {1x1 struct}
    8.3497 sec    {1x1 struct}
```

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

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

```
ans =

  struct with fields:

      Position: [296 203 203 144]
    brakeLight: [1x2 struct]
```

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

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

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

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

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

```
ans =

  1x2 struct array with fields:

    Position
    isOn
```

View the bounding box positions for the sublabels.

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

```
ans =

   304   245    50    46


ans =

   435   243    54    51
```

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

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

```
ans =

  logical

   1


ans =

  logical

   1
```

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

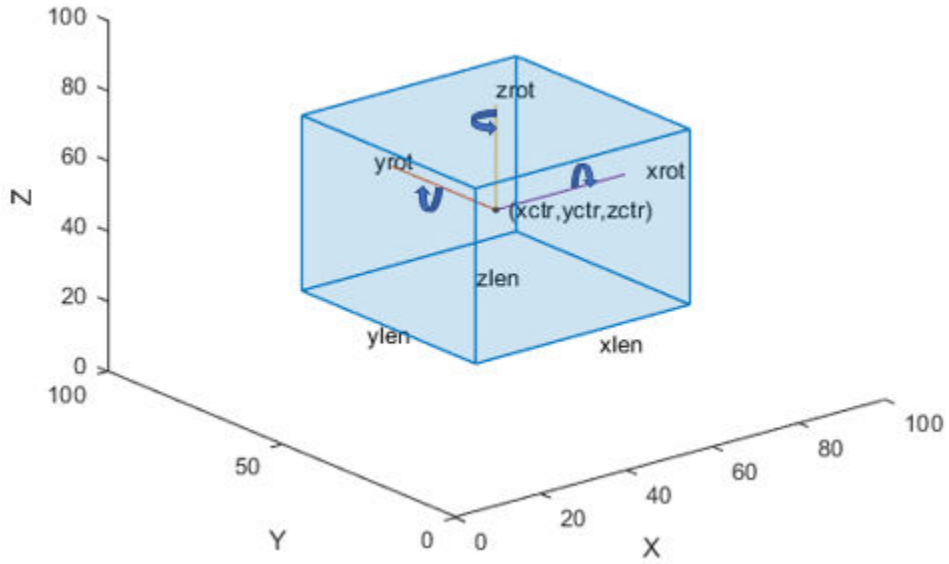
```
lidarLabels.car{1}
```

```
ans =

  struct with fields:

      Position: [-1.1559 -0.7944 1.2012 12.6196 5.9278 3.0010 0 0 0]
    brakeLight: []
```

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

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

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



Scene Label Data

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

```
gTruth.SceneLabelData
```

```
ans =

  SceneLabelData with properties:
```

```
    daytime: [0 sec    10.15 sec]
```

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

Use Ground Truth Labels

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

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

See Also

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

More About

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

Sources vs. Signals in Ground Truth Labeling

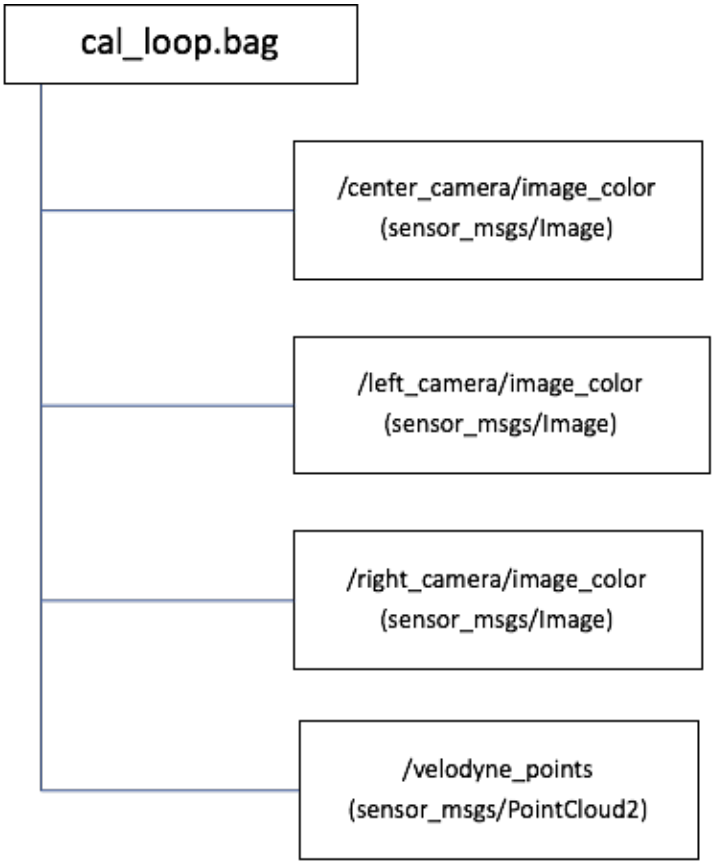
In the **Ground Truth Labeler** app, a source is the file or folder containing the data that you want to load. A signal is the data from that source that you want to label. A source can contain one or more signals.

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

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

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

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



See Also

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

More About

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

Keyboard Shortcuts and Mouse Actions for Ground Truth Labeler

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

Label Definitions

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

Frame Navigation and Time Interval Settings

Navigate between frames and change the time interval of the signal. These controls are located in the bottom pane of the app.

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

Labeling Window

Perform labeling actions, such as adding, moving, and deleting regions of interest (ROIs). The ROIs can be pixel labels or non-pixel ROI labels that include line, rectangle, cuboid, and projected cuboid.

Task	Action
Undo labeling action	**Ctrl+Z**
Redo labeling action	**Ctrl+Y**
Select all non-pixel ROIs	**Ctrl+A**

Task	Action
Select specific non-pixel ROIs	Hold **Ctrl** and click the ROIs you want to select
Cut selected non-pixel ROIs	**Ctrl+X**
Copy selected non-pixel ROIs to clipboard	**Ctrl+C**
Paste copied non-pixel 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**
Switch between selected non-pixel ROI labels. You can switch between labels only of the same type. For example, if you select a rectangle ROI, you can switch only between other rectangle ROIs.	**Tab** or **Shift+Tab**
Move a drawn non-pixel ROI label	Hold **Ctrl** and press the up, down, left or right arrows
Resize a rectangle ROI uniformly across all dimensions	**Ctrl+Plus (+)** or **Ctrl+Minus (-)**
Delete selected non-pixel ROIs	**Delete**
Copy all pixel ROIs	**Ctrl+Shift+C**
Cut all pixel ROIs	**Ctrl+Shift+X**
Paste copied or cut pixel ROIs	**Ctrl+Shift+V**
Delete all pixel ROIs	**Ctrl+Shift+Delete**
Fill all or all remaining pixels	**Shift+click**

Cuboid Resizing and Moving

Draw cuboids to label lidar point clouds. For examples on how to use these shortcuts to label lidar point clouds efficiently, see “Label Lidar Point Clouds for Object Detection” on page 2-39.

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

Task	Action
Resize a cuboid uniformly across all dimensions before applying it to the point cloud	Hold **A** and move the scroll wheel up to increase size or down to decrease size
Resize a cuboid along only the *x*-dimension before applying it to the point cloud	Hold **X** and move the scroll wheel up to increase size or down to decrease size

Task	Action
Resize a cuboid along only the *y*-dimension before applying it to the point cloud	Hold **Y** and move the scroll wheel up to increase size or down to decrease size
Resize a cuboid along only the *z*-dimension before applying it to the point cloud	Hold **Z** and move the scroll wheel up to increase size or down to decrease size
Resize a cuboid after applying it to the point cloud	Click and drag one of the cuboid faces
Move a cuboid	Hold **Shift** and click and drag one of the cuboid faces The cuboid is translated along the dimension of the selected face.
Move multiple cuboids simultaneously	Follow these steps: **1** Hold **Ctrl** and click the cuboids that you want to move. **2** Hold **Shift** and click and drag a face of one of the selected cuboids. The cuboids are translated along the dimension of the selected face.

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, Panning, and Rotating

Task	Action
Zoom in or out of an image frame	Move the scroll wheel up to zoom in or down to zoom out If the frame is in pan mode, then zooming is not supported. To enable zooming, in the upper-right corner of the frame, either click the Pan button to disable panning or click one of the zoom buttons.
Zoom in on specific section of an image frame	In the upper-right corner of the frame, click the Zoom In button and then click and drag within the frame to draw a box around the section that you want to zoom in on Zooming in on a specific section of a point cloud is not supported.
Pan across an image frame	Press the up, down, left, or right arrows
Zoom in on or out of a point cloud frame	In the top-left corner of the display, click the Zoom In or Zoom Out button. Then, move the scroll wheel up (zoom in) or down (zoom out). Alternatively, move the cursor up or right (zoom in) or down or left (zoom out). Zooming in and out is supported in all modes (pan, zoom, and rotate).
Pan across a point cloud frame	Hold **Shift** and press the up, down, left, or right arrows
Rotate a point cloud frame	Hold **R** and click and drag the point cloud frame **Note** Only yaw rotation is allowed.

App Sessions

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

See Also

Ground Truth Labeler

More About

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

Control Playback of Signal Frames for Labeling

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

Signal Frames

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

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

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

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

Master Signal

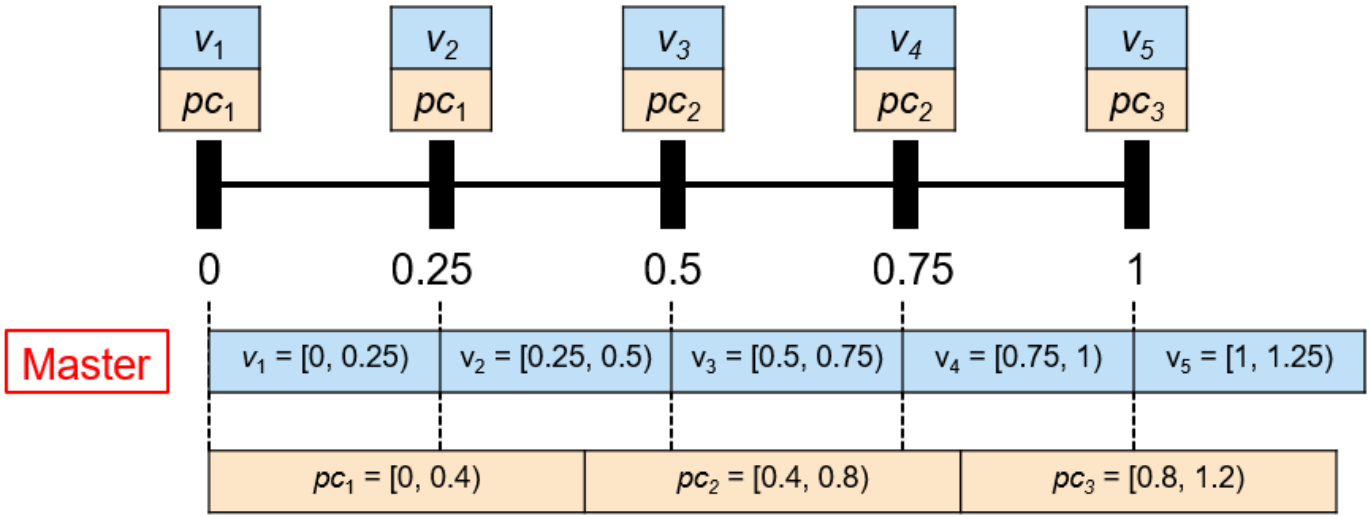
When you load multiple signals into a new app session, by default, the app designates the signal with the highest frame rate as the master signal. When you play back signals or navigate between frames, the app displays all frames from the master signal.

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

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

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

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



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

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

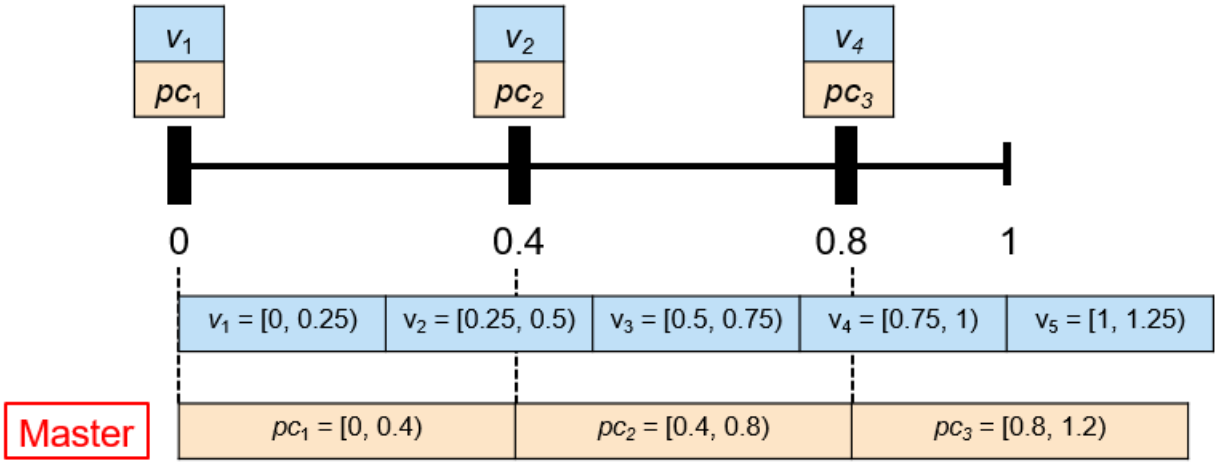
The app repeats the point cloud frame because the next point cloud frame, pc_2, does not start until 0.4 seconds. To display this frame, you must either set the **Current Time** parameter to 0.4 seconds or click the Next Frame button again to navigate to a time of 0.5 seconds.

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

Change Master Signal

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

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



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

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

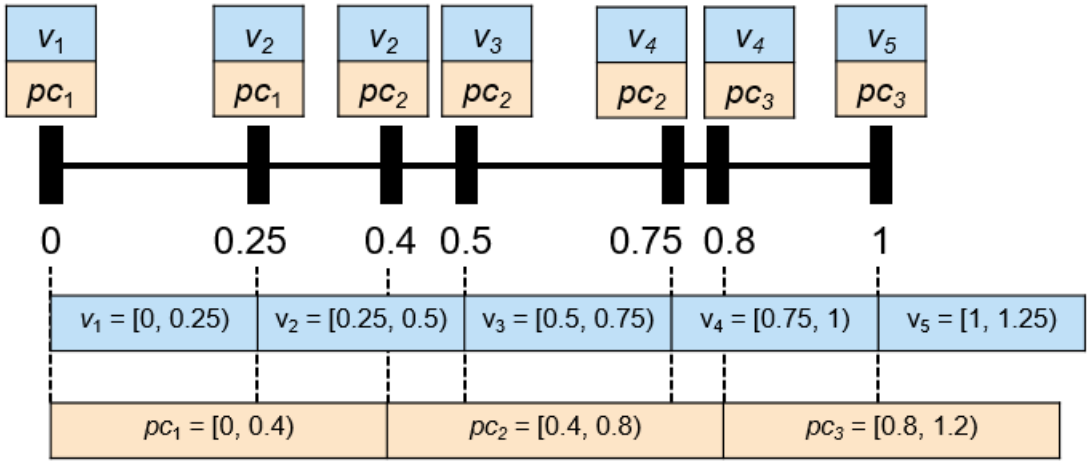
Changing the master signal after you begin labeling can affect existing scene labels. For example, suppose you apply a scene label to the entire time range of the master signal. If you change the master signal, the time range changes. If the new master signal has a longer duration, then the scene label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

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

Display All Timestamps

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

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



Specify Timestamps

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

Frame Display and Automation

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

See Also

`duration` | `groundTruthMultisignal`

More About

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

Label Lidar Point Clouds for Object Detection

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

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

Set Up Lidar Point Cloud Labeling

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

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

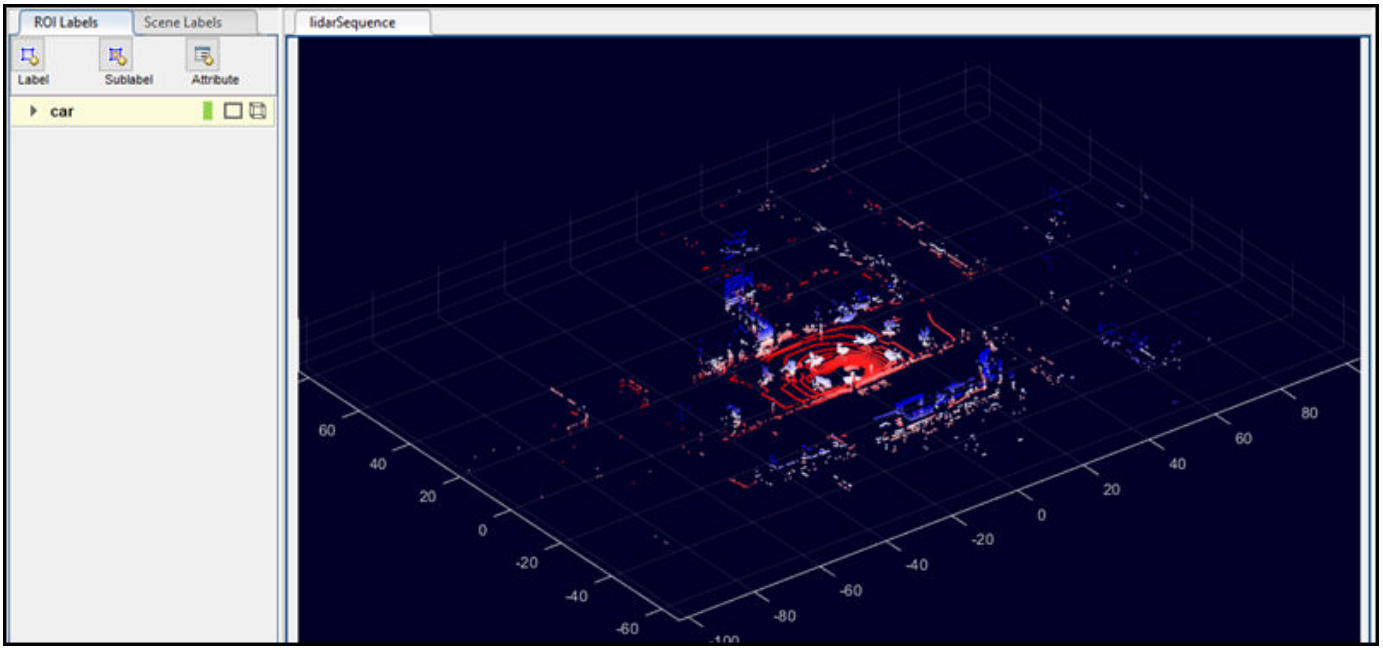
   ```
   groundTruthLabeler
   ```

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

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

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

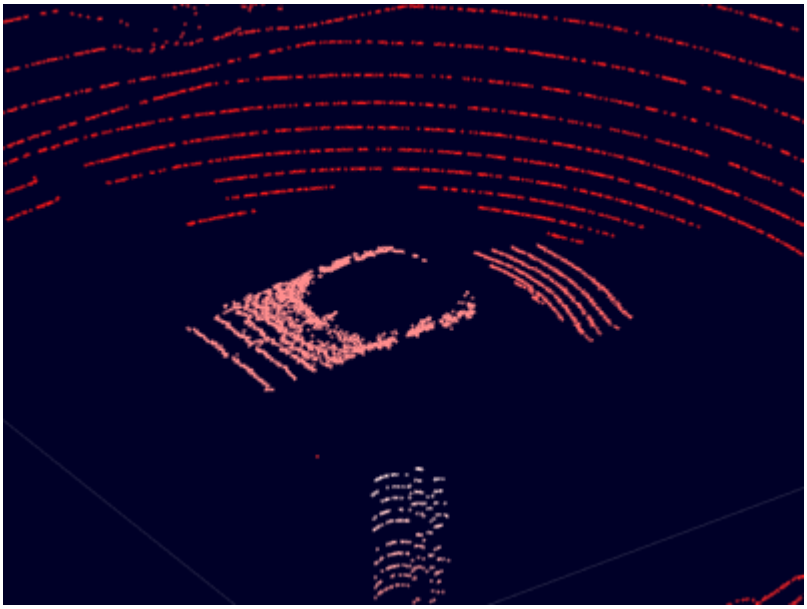
This figure shows the **Ground Truth Labeler** app setup after following these steps.



Zoom, Pan, and Rotate Frame

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

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

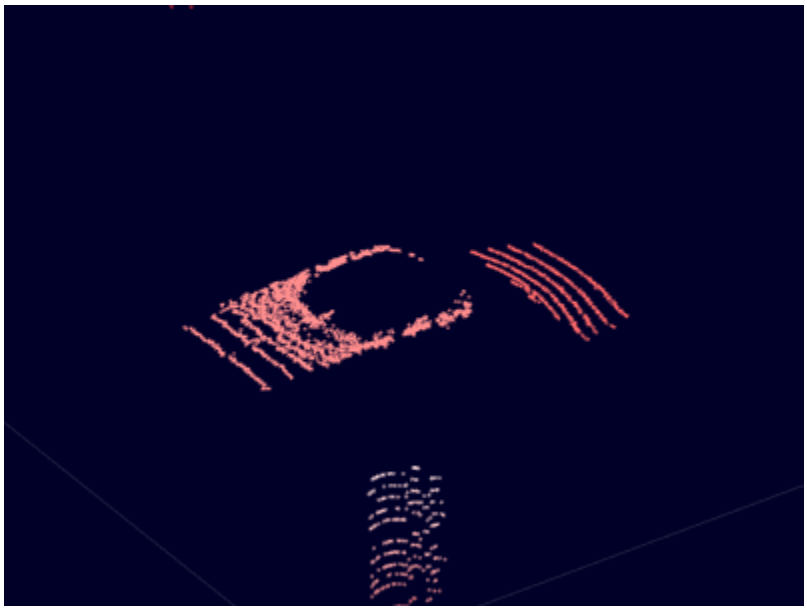


Optionally, you can use the Pan button or Rotate 3D button to help you view more of the ego vehicle points. To view additional options for viewing or rotating the point cloud, click the Rotate 3D button and then right-click the point cloud frame. The options provided are the same options provided with the `pcshow` function.

Hide Ground

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

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



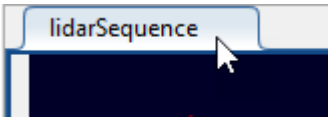
This option only hides the ground from the display. It does not remove ground data from the point cloud. If you label a section of the point cloud containing hidden ground points, when you export the ground truth labels, those ground points are a part of that label.

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

Label Cuboid

Label the ego vehicle by using a cuboid label.

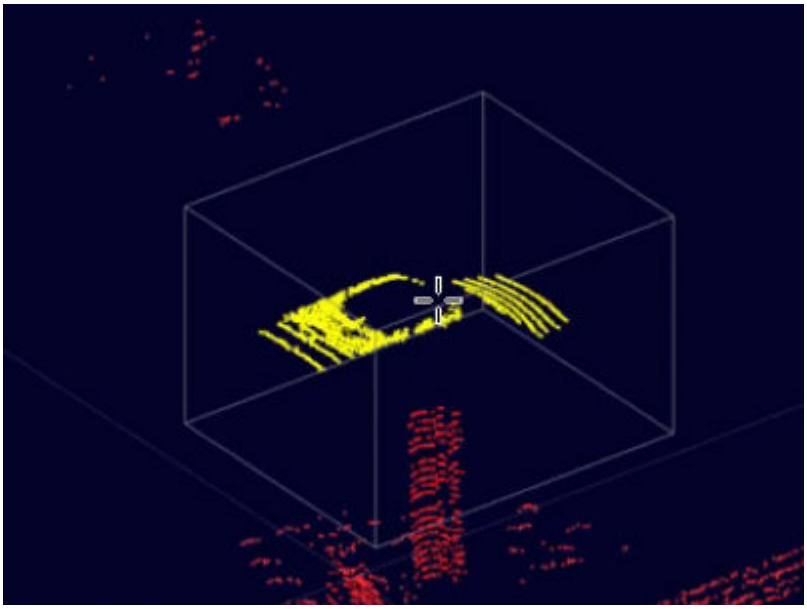
1. In the **ROI Labels** pane on the left, click the **car** label.
2. Select the lidar point sequence frame by clicking the **lidarSequence** tab.



Note To enable the labeling keyboard shortcuts, you must first select the signal frame.

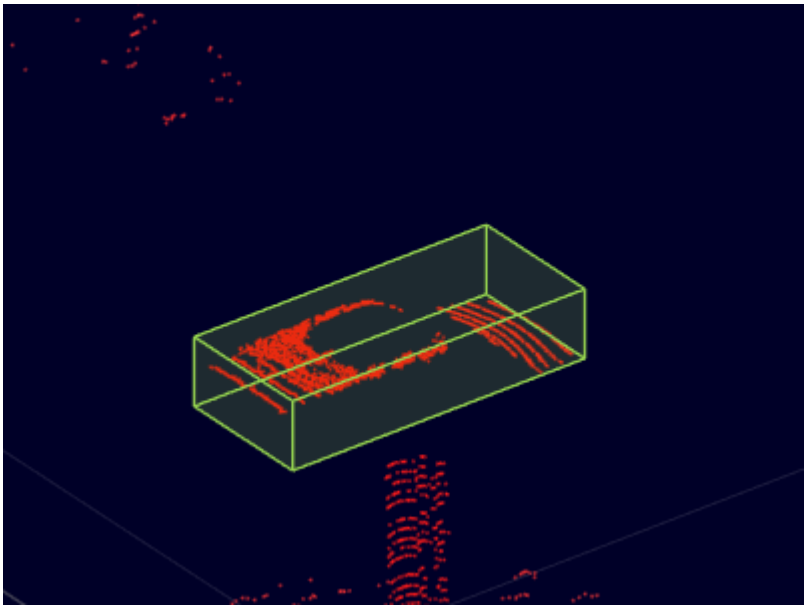
3. Move the pointer over the ego vehicle until the gray preview cuboid encloses the ego vehicle points. The points enclosed in the preview cuboid highlight in yellow.

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



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

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

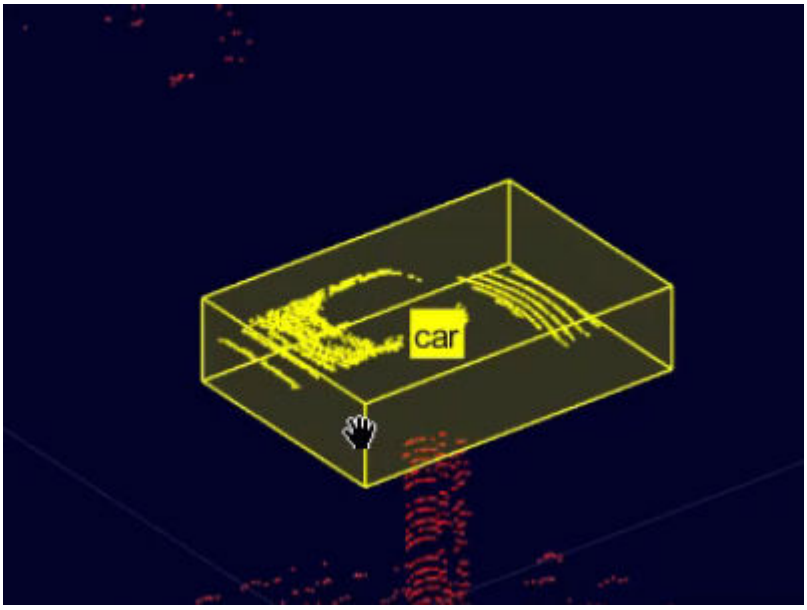


For more control over the labeling of point clouds, on the app toolstrip, click **Snap to Cluster**. When you label with this option selected, the cuboid snaps to the nearest point cloud cluster by using the `segmentLidarData` function. To configure point cloud clustering, click **Cluster Settings** and adjust the options in the dialog box. To view point cloud clusters as you navigate between frames, select **View Clusters** in this dialog box. During playback of a signal, the visualization of point cloud clusters is disabled.

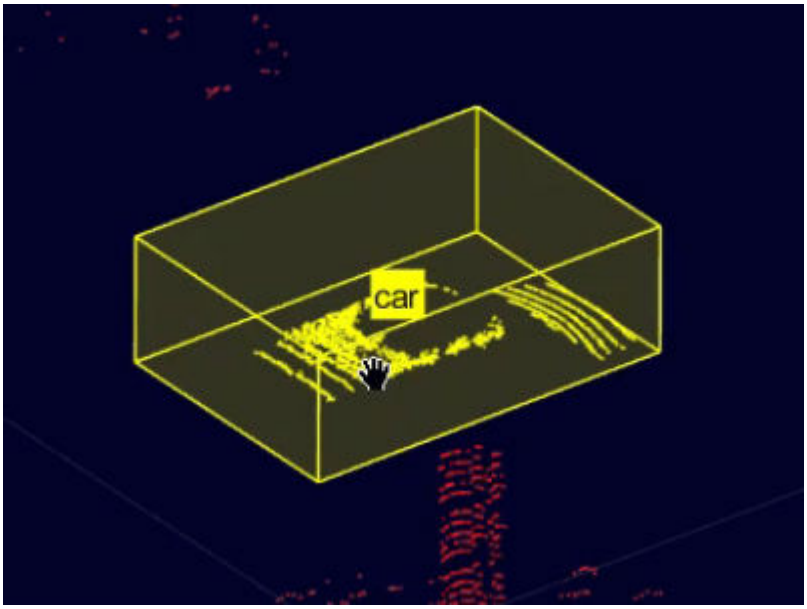
Modify Cuboid Label

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

1. To enable the point cloud labeling keyboard shortcuts, verify that the **lidarSequence** tab is selected.
2. In the signal frame, click the drawn cuboid label. Drag the faces to expand the cuboid.



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



Apply Cuboids to Multiple Frames

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

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

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

Configure Display

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

Change Colormap

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

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

Change Views

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

You can select from these views:

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

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

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

See Also

More About

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

Create Class for Loading Custom Ground Truth Data Sources

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

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

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

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

Custom Class Folder

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

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

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

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

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

Class Definition

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

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

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

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

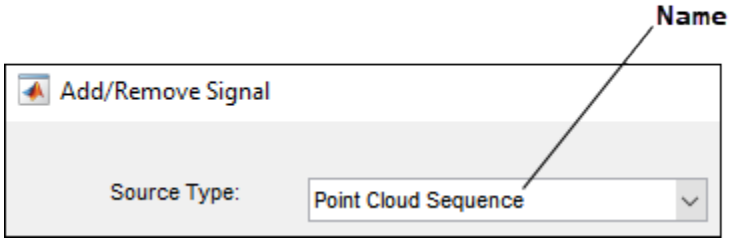
Class Properties

Data source classes must define these abstract, constant properties.

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

- `Description` — A string scalar describing the class

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



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

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

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

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

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

Method to Customize Load Panel

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

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

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

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

    computePositions(this);

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

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

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

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

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

    this.TimeStampsDropDown = uicontrol('Parent', this.Panel,...
```

```
            'Style', 'popupmenu',...
            'String', ["Use Default", "From Workspace"],...
            'Position', this.TimeStampsDropDownPos,...
            'Callback', @this.timeStampsDropDownCallback,...
            'Tag', 'timeStampSourceSelectList');

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

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

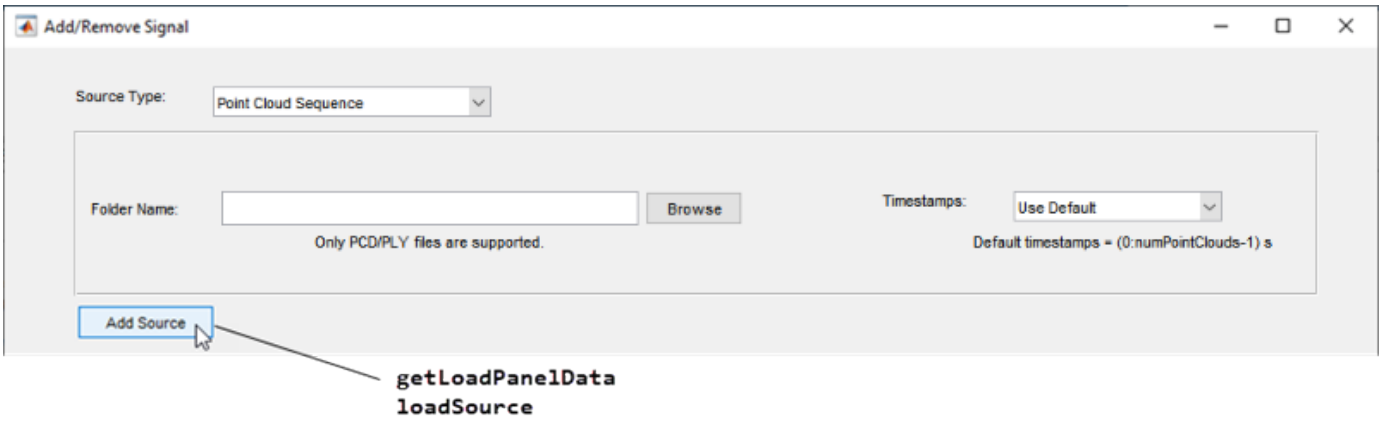
```
vision.labeler.loading.PointCloudSequenceSource.loadPanelChecker
```

Methods to Get Load Panel Data and Load Data Source

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

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

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



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

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

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

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

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

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

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

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

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

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

    % Populate timestamps

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

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

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

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

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

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

Method to Read Frames

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

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

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

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

Use Predefined Data Source Classes

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

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

See Also

Apps
Ground Truth Labeler

Classes
`vision.labeler.loading.MultiSignalSource`

Objects
`groundTruthMultisignal`

Tracking and Sensor Fusion

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

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

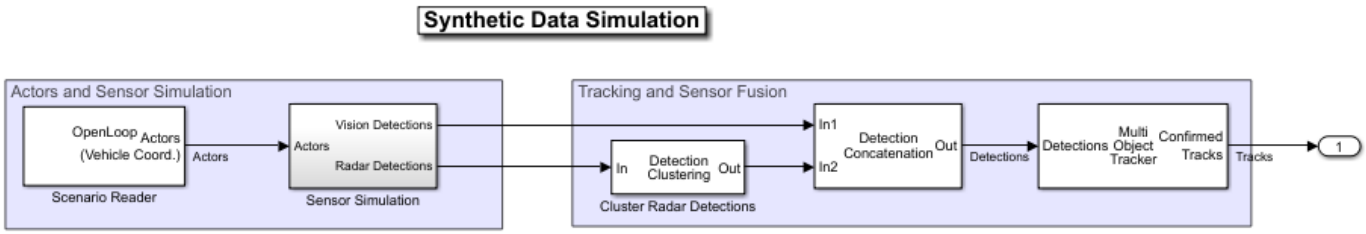
- Sensor coverages of vision, radar, and lidar sensors
- Sensor detections of actors and lane boundaries
- Tracks of moving objects in the scenario

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

Open Model and Scope

Open a model containing signals for sensor detections and tracks. This model is used in the "Sensor Fusion Using Synthetic Radar and Vision Data in Simulink" on page 7-205 example. Also add the file folder of the model to the MATLAB search path.

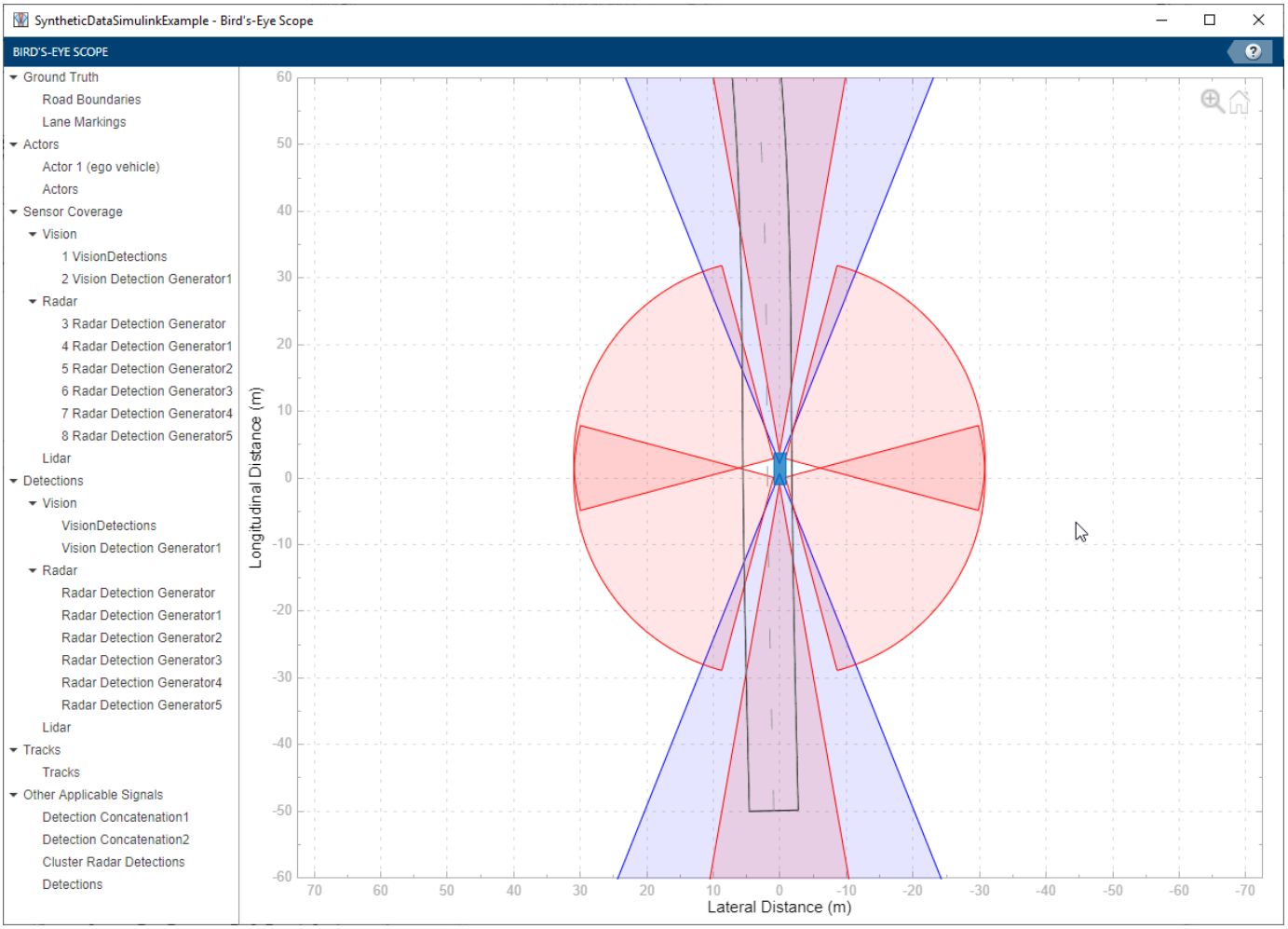
```
addpath(genpath(fullfile(matlabroot,'examples','driving')))
open_system('SyntheticDataSimulinkExample')
```



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

Find Signals

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



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

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

Signal Group	Description	Signal Sources

Actors	Actors in the scenario, including the ego vehicle
	
	You cannot modify this group or any of its signals or subgroups.
	
	To view actors that are located away from the ego vehicle, use the **World Coordinates View** window. See “Vehicle and World Coordinate Views”.
		• Scenario Reader block
		• Vision Detection Generator, Radar Detection Generator, and Lidar Point Cloud Generator blocks (for actor profile information only, such as the length, width, and height of actors)
		  • If actor profile information is not set or is inconsistent between blocks, the scope sets the actor profiles to the default actor profile values for each block.
		  • The profile of the ego vehicle is always set to the default profile for each block.

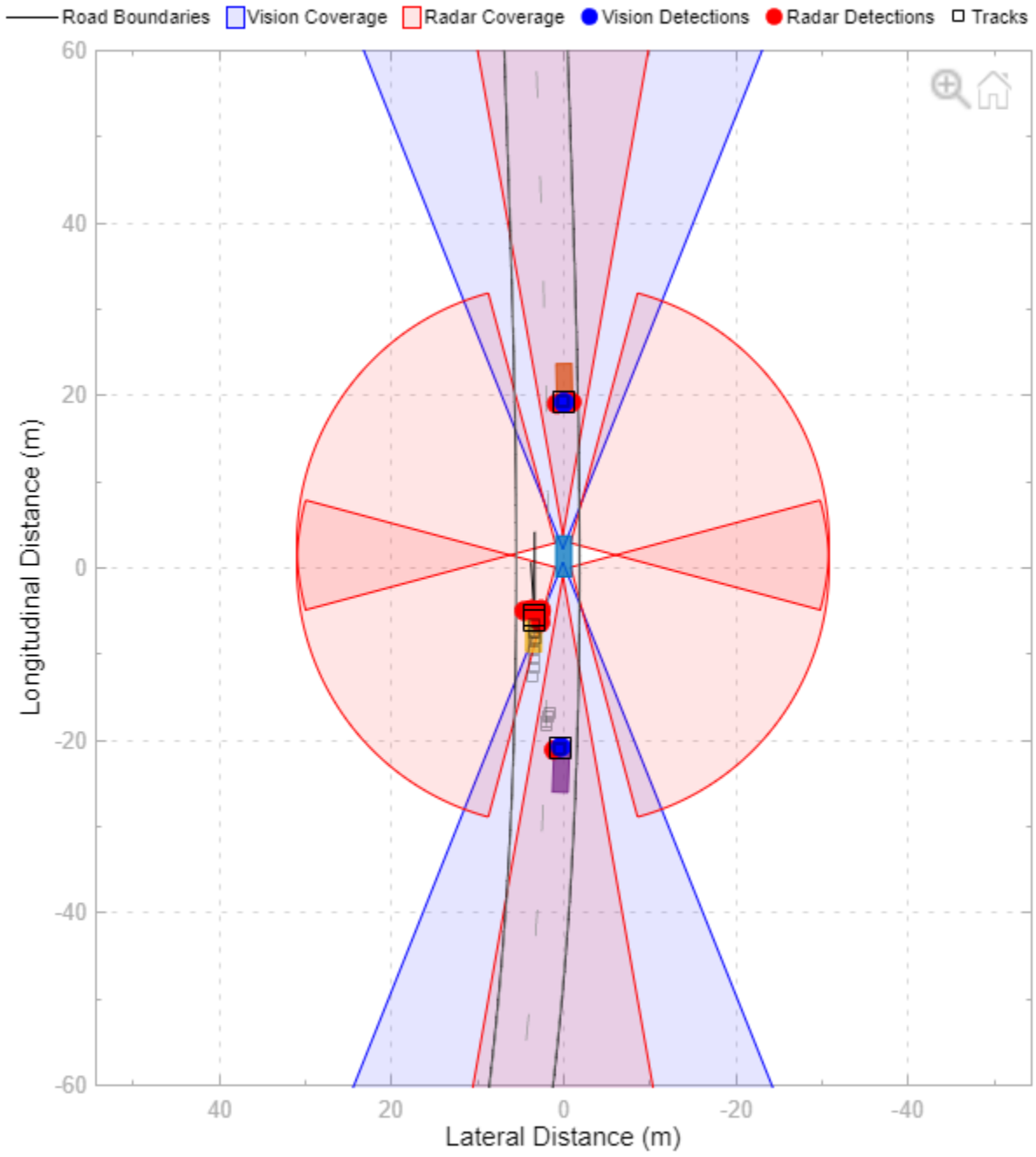
Sensor Coverage	Coverage areas of vision, radar, and lidar sensors, sorted into **Vision**, **Radar**, and **Lidar** subgroups You can modify signals in this group. You can rename or delete subgroups but not the top-level **Sensor Coverage** group. You can also add subgroups and move signals between subgroups. If you delete a subgroup, its signals move to the top-level **Sensor Coverage** group.	• Vision Detection Generator block • Simulation 3D Vision Detection Generator • Radar Detection Generator block • Simulation 3D Probabilistic Radar block • Lidar Point Cloud Generator block • Simulation 3D Lidar block

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

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

Run Simulation

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



During simulation, you can perform these actions:

- Inspect detections, tracks, sensor coverage areas, and ego vehicle behavior. The default view displays the simulation in vehicle coordinates and is centered on the ego vehicle. To view the wider area around the ego vehicle, or to view other parts of the scenario, on the scope toolstrip, click **World Coordinates**. The **World Coordinates View** window displays the entire scenario, with the ego vehicle circled. This circle is not a sensor coverage area. To return to the default display of either window, move your pointer over the window, and in the upper-right corner, click

the home button  that appears. For more details on these views, see “Vehicle and World Coordinate Views”.

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

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

Organize Signal Groups (Optional)

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

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

Update Model and Rerun Simulation

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

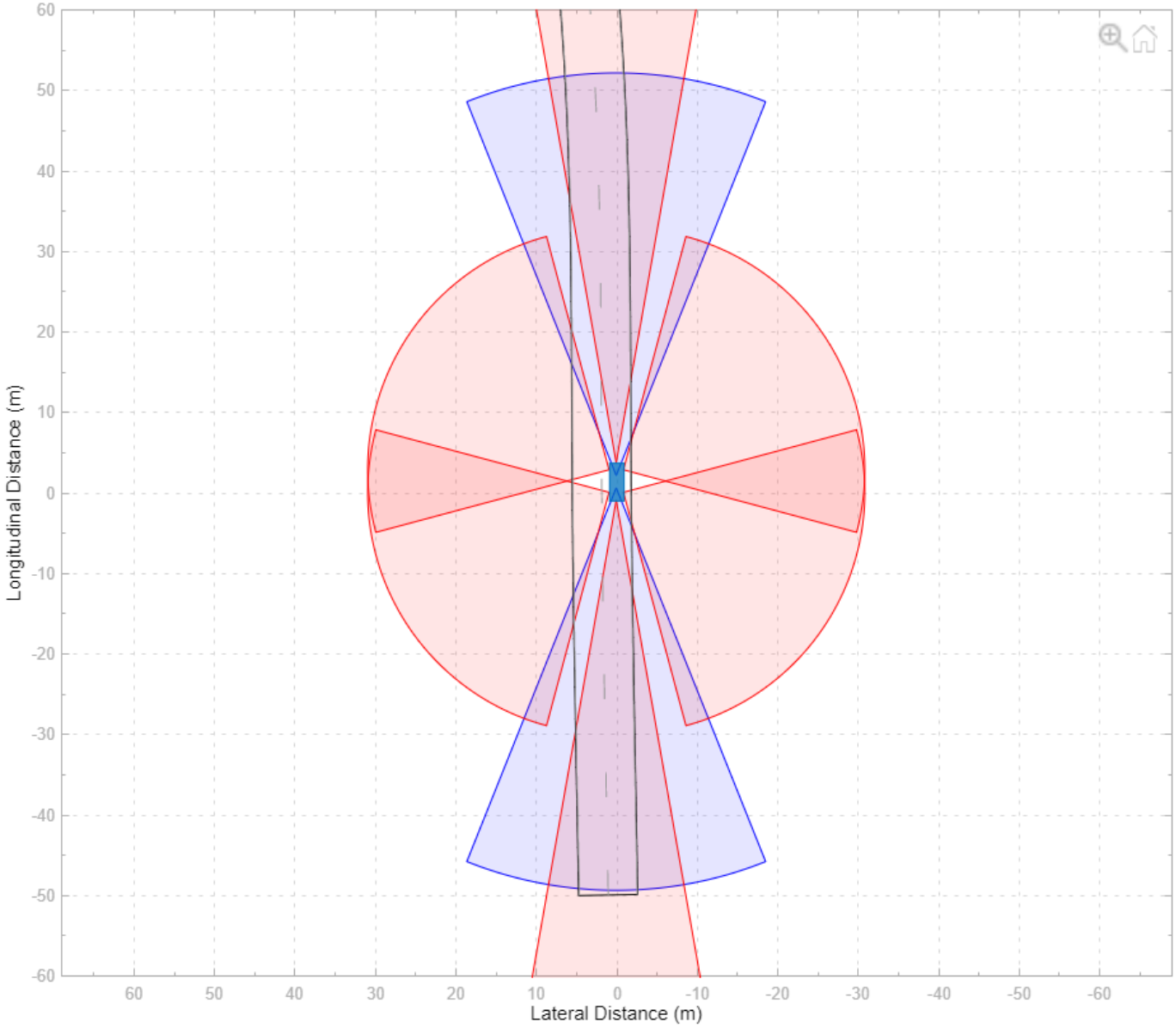
When you modify block parameters, you can rerun the simulation without having to find signals again. If you add or remove blocks, ports, or signal lines, then you must click **Find Signals** again before rerunning the simulation.

Save and Close Model

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

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

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



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

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

See Also

Apps
Bird's-Eye Scope

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

Related Examples

- “Visualize Sensor Data from Unreal Engine Simulation Environment” on page 6-36
- “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” on page 7-205
- “Lateral Control Tutorial” on page 7-589
- “Autonomous Emergency Braking with Sensor Fusion” on page 7-214
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-121
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-127

Linear Kalman Filters

In this section...

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

State Equations

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

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

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

If you define the state as

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

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

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

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

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

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

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

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

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

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

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

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

The state equation can be generalized to

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

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

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

Measurement Models

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

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

$$z_k = H_k x_k + w_k$$

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

Linear Kalman Filter Equations

Without noise, the dynamic equations are

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

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

$$z_k = H_k x_k$$

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

Filter Loop

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

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

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

Propagate the covariance matrix as well.

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

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

Then predict the measurement at the updated time.

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

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

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

Then the Kalman gain is

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

and is derived from using an optimality condition.

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

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

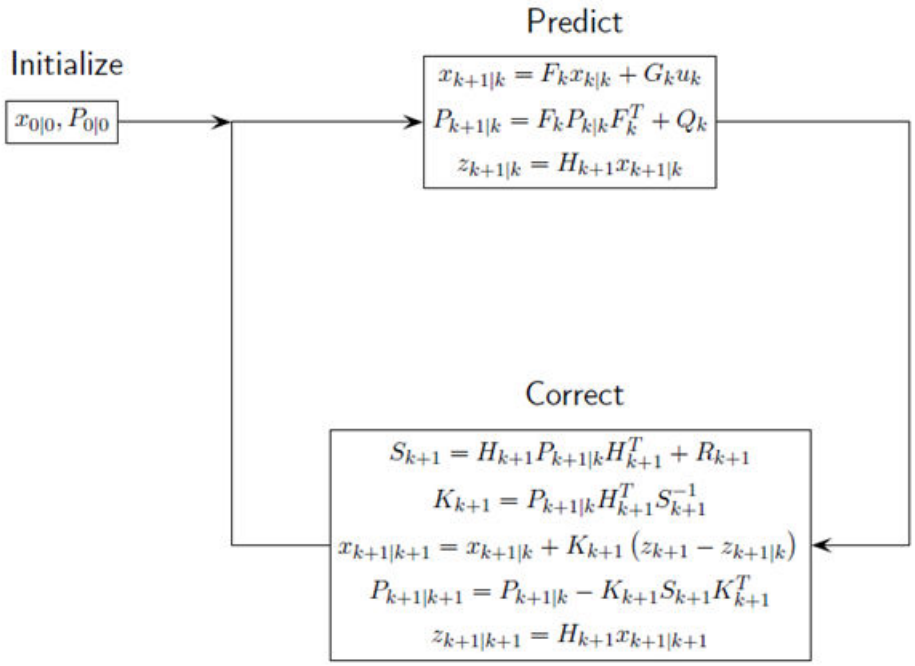
where K_{k+1} is the Kalman gain. The corrected state is often called the *a posteriori* estimate of the state because it is derived after the measurement is included.

Correct the state covariance matrix

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

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

This figure summarizes the Kalman loop operations.



Constant Velocity Model

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

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

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

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

Constant Acceleration Model

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

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

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

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

See Also

Objects
`trackingKF`

Extended Kalman Filters

In this section...

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

State Update Model

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

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

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

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

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

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

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

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

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

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

Measurement Model

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

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

The Jacobian of the measurement with respect to the state is

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

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

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

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

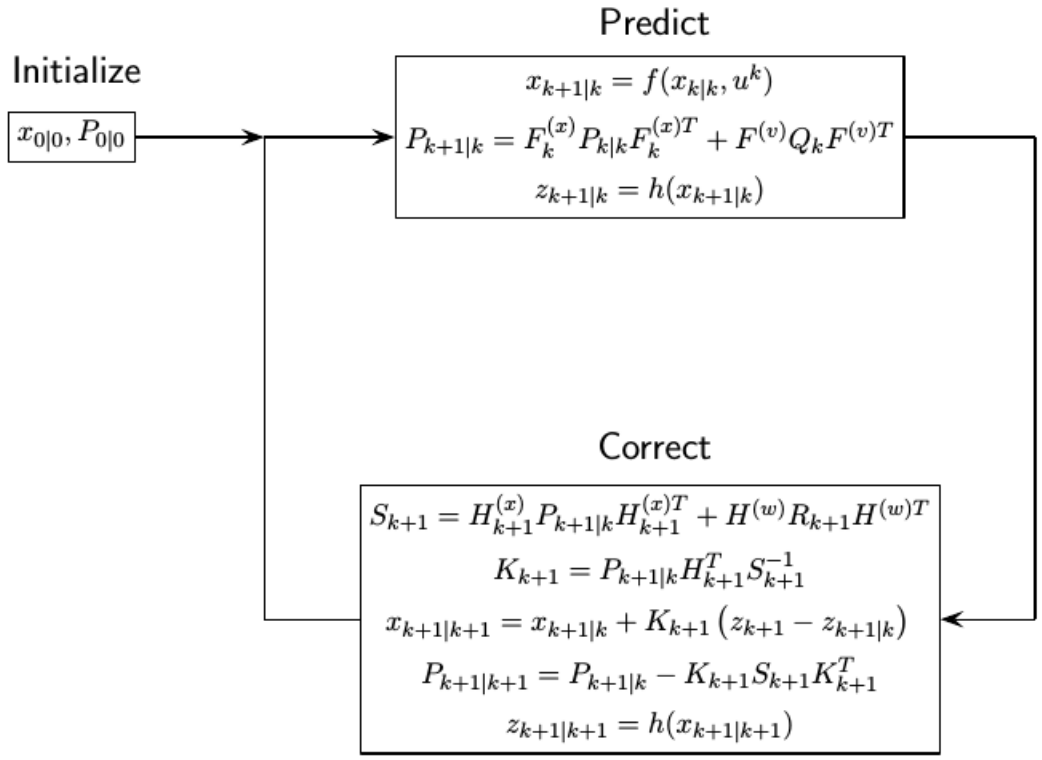
$$z_k = h(x_k, t) + v_k$$

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

Extended Kalman Filter Loop

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

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



Predefined Extended Kalman Filter Functions

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

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

See Also

Objects
`trackingEKF`

Planning, Mapping, and Control

Display Data on OpenStreetMap Basemap

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

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

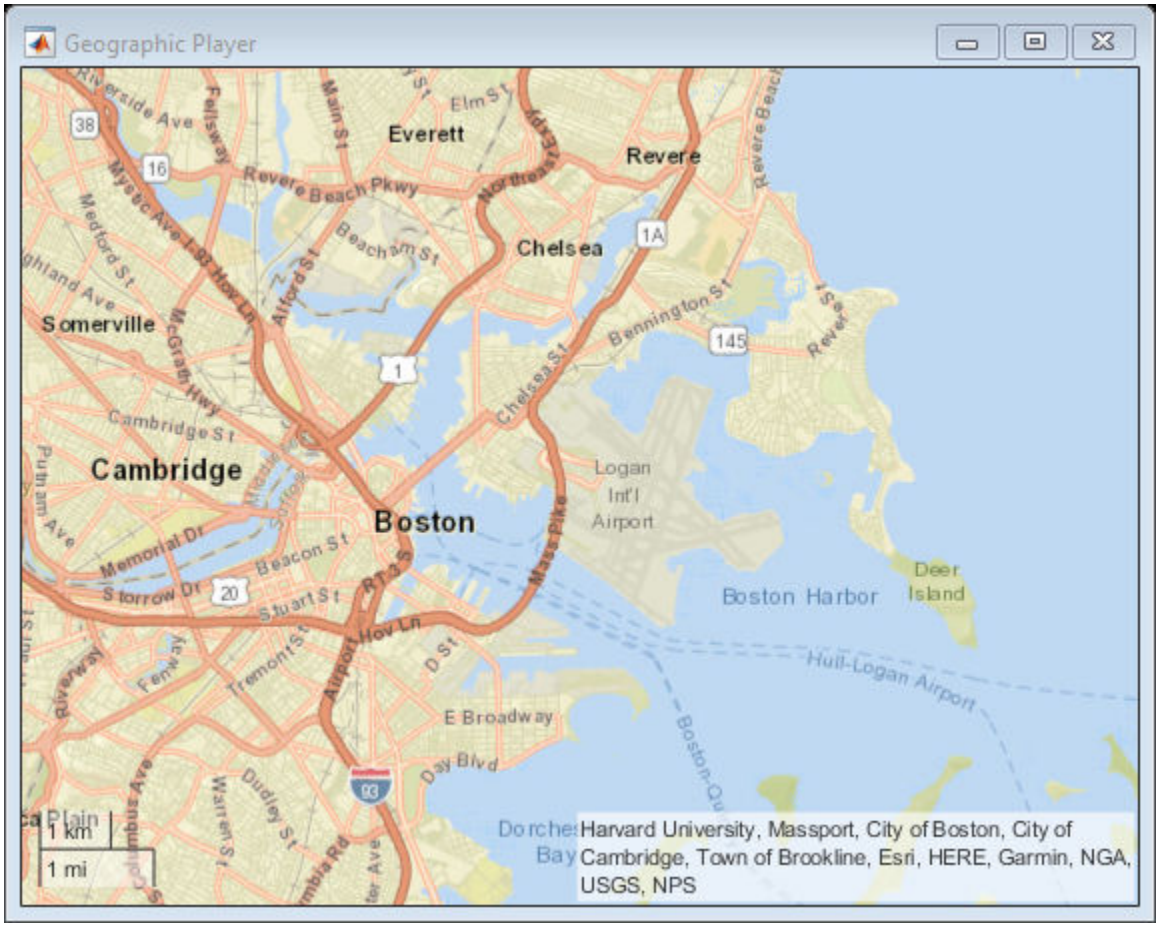
```
name = 'openstreetmap';
url = 'https://a.tile.openstreetmap.org/${z}/${x}/${y}.png';
copyright = char(uint8(169));
attribution = copyright + "OpenStreetMap contributors";
addCustomBasemap(name,url,'Attribution',attribution)
```

Load a sequence of latitude and longitude coordinates.

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

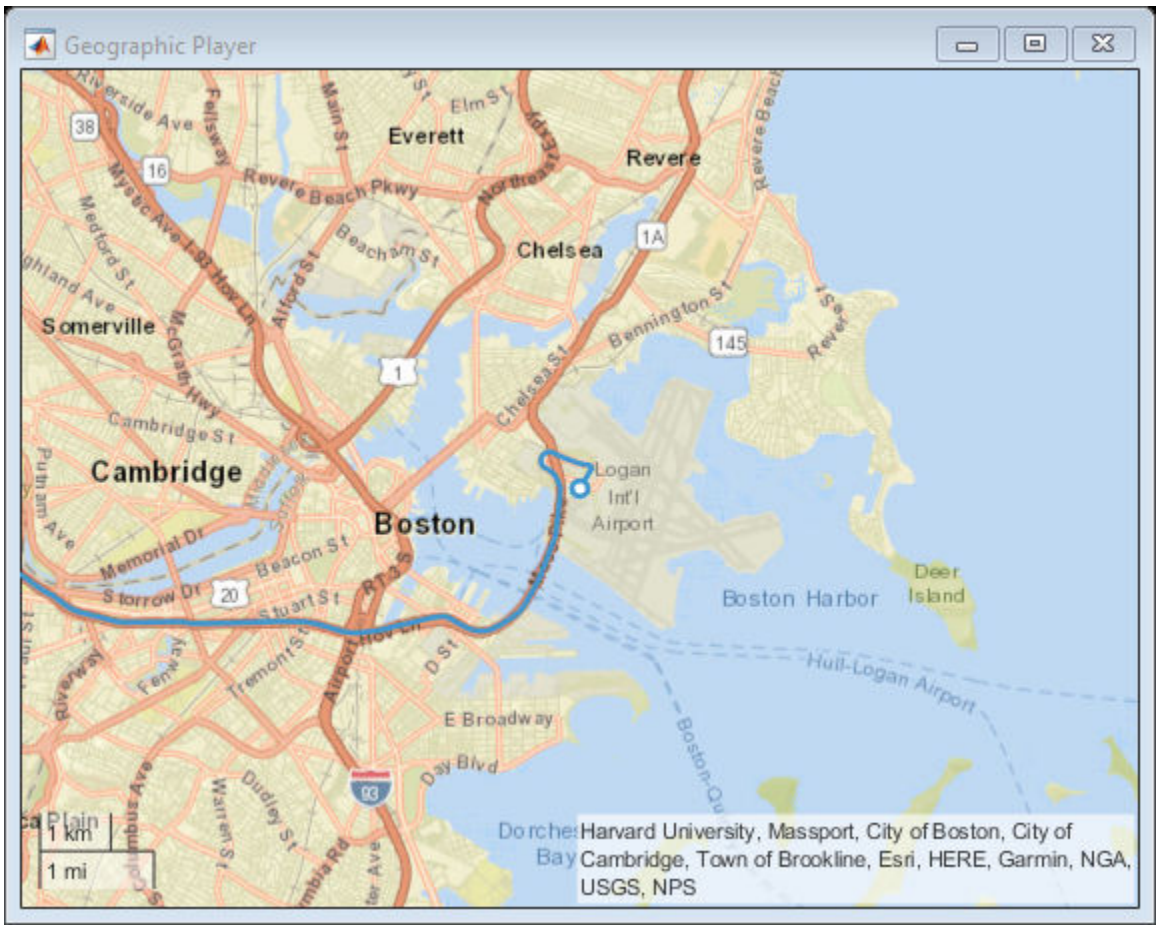
Create a geographic player. Center the geographic player on the first position of the driving route and set the zoom level to 12.

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



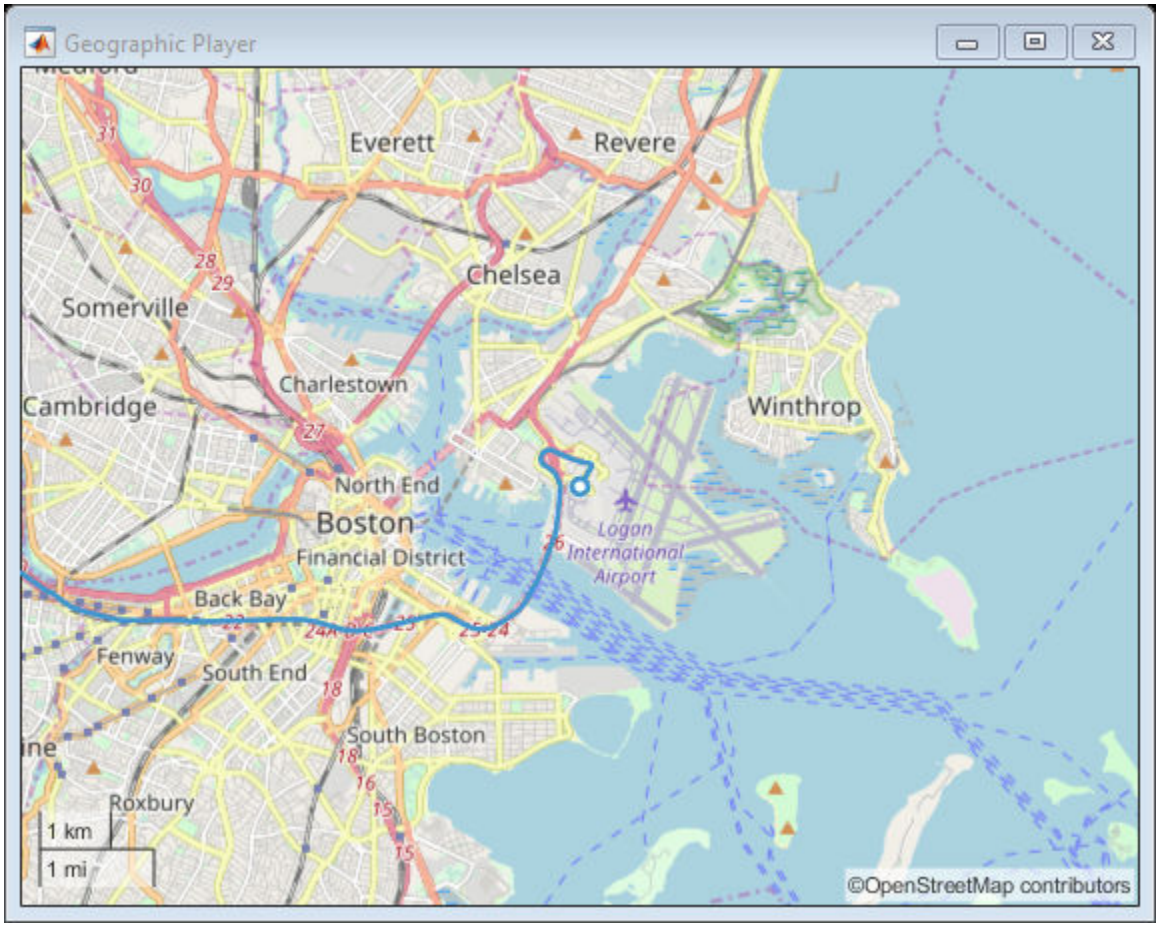
Display the full route.

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



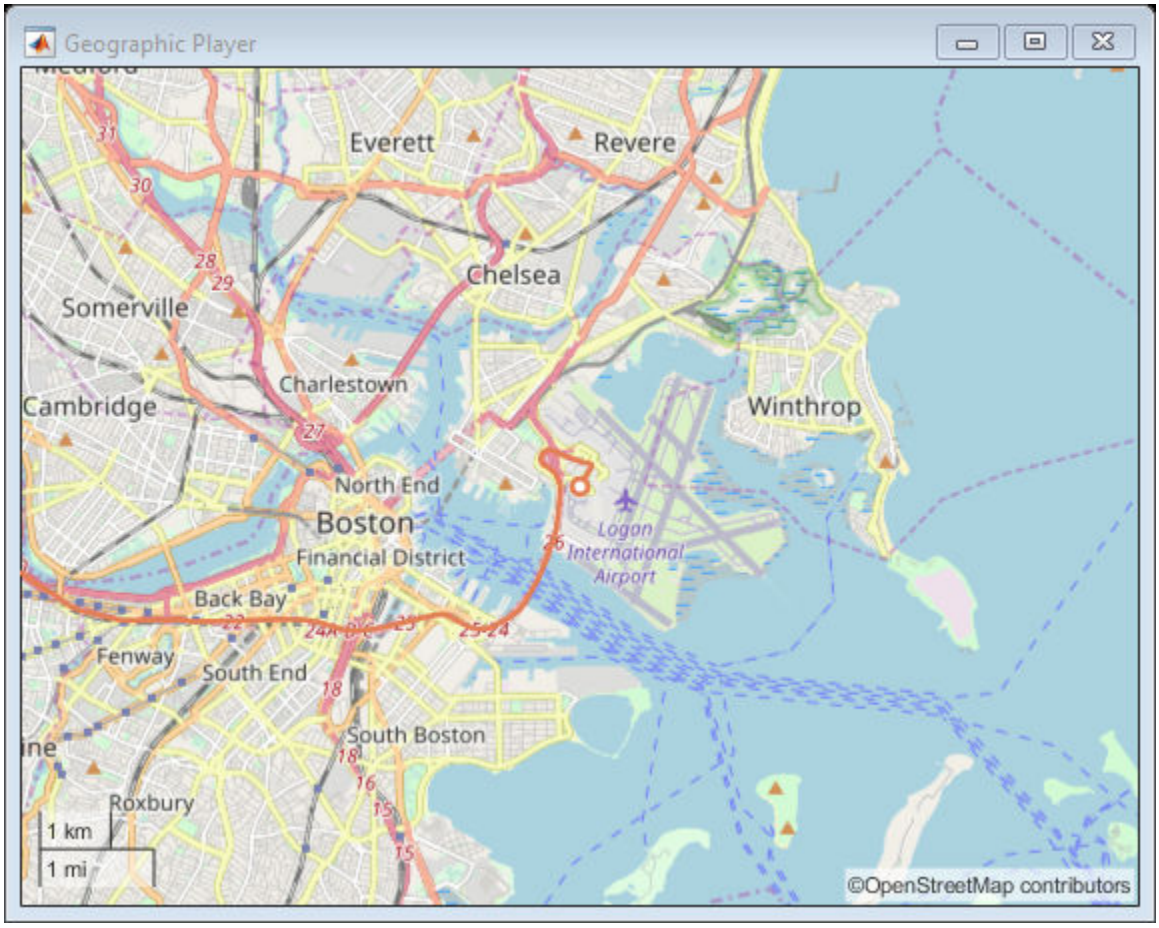
By default, the geographic player uses the World Street Map basemap (`'streets'`) provided by Esri®. Update the geographic player to use the added OpenStreetMap basemap instead.

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



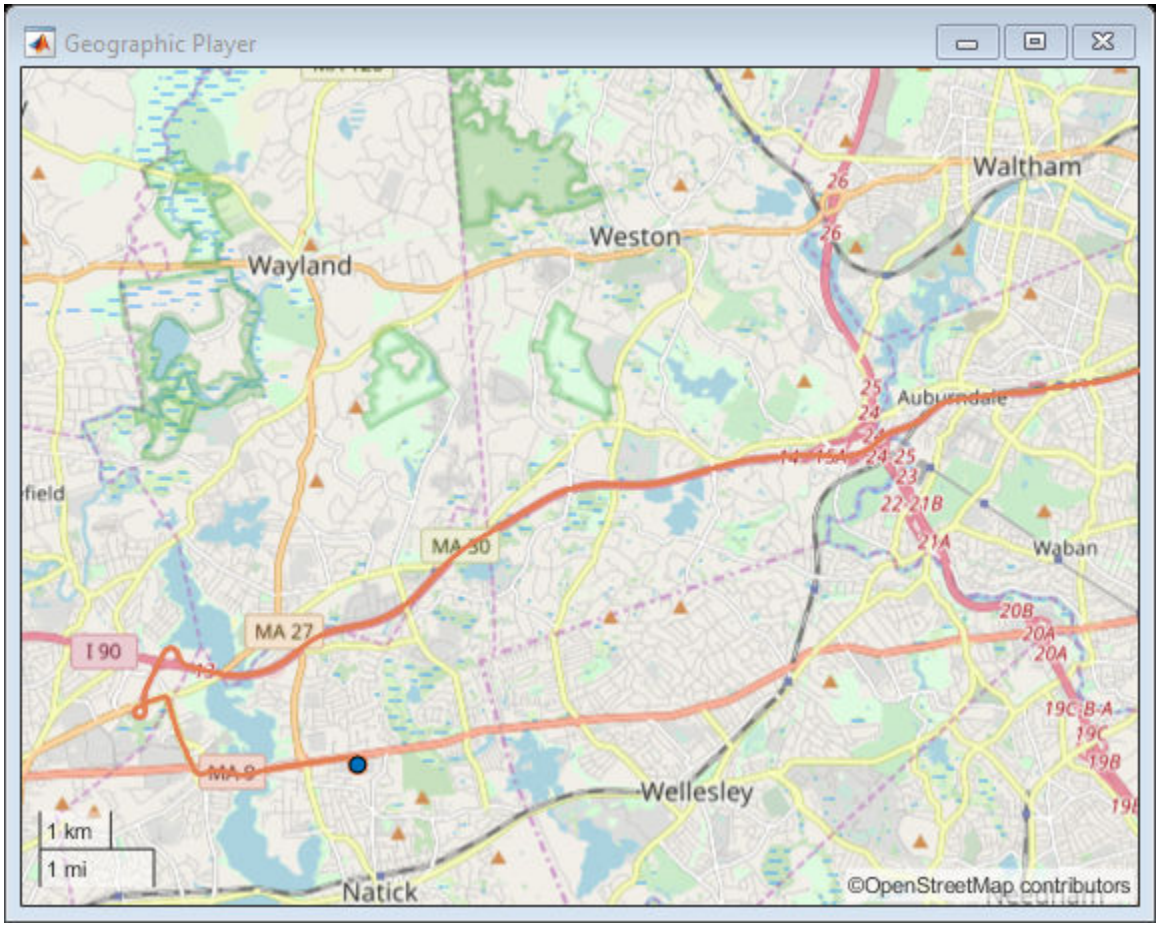
Display the route again.

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



Display the positions of the vehicle in a sequence.

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



See Also

`addCustomBasemap` | `geoplayer` | `plotPosition` | `plotRoute` | `removeCustomBasemap`

Read and Visualize HERE HD Live Map Data

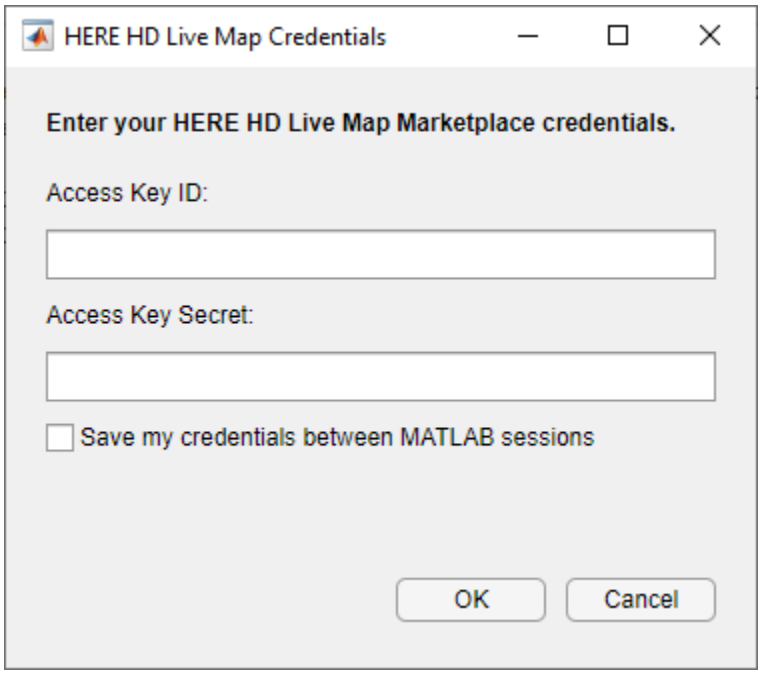
HERE HD Live Map[1] (HERE HDLM), developed by HERE Technologies, is a cloud-based web service that enables you to access highly accurate, continuously updated map data. The data is composed of tiled map layers containing information such as the topology and geometry of roads and lanes, road-level attributes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

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

Enter Credentials

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

```
hereHDLMCredentials setup
```



Enter a valid **Access Key ID** and **Access Key Secret**, and click **OK**. The credentials are 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.

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

Configure Reader to Search Specific Catalog

In the HERE HDLM web service, map data is stored in a set of databases called catalogs. Each catalog roughly corresponds to a different geographic region, such as North America or Western Europe. By creating a `hereHDLMConfiguration` object, you can configure a HERE HDLM reader to search for map data from only a specific catalog. You can also optionally specify the version of the catalog that you want to search. These configurations can speed up the performance of the reader, because the reader does not search unnecessary catalogs for map data.

For example, create a configuration for the catalog that roughly corresponds to the North America region.

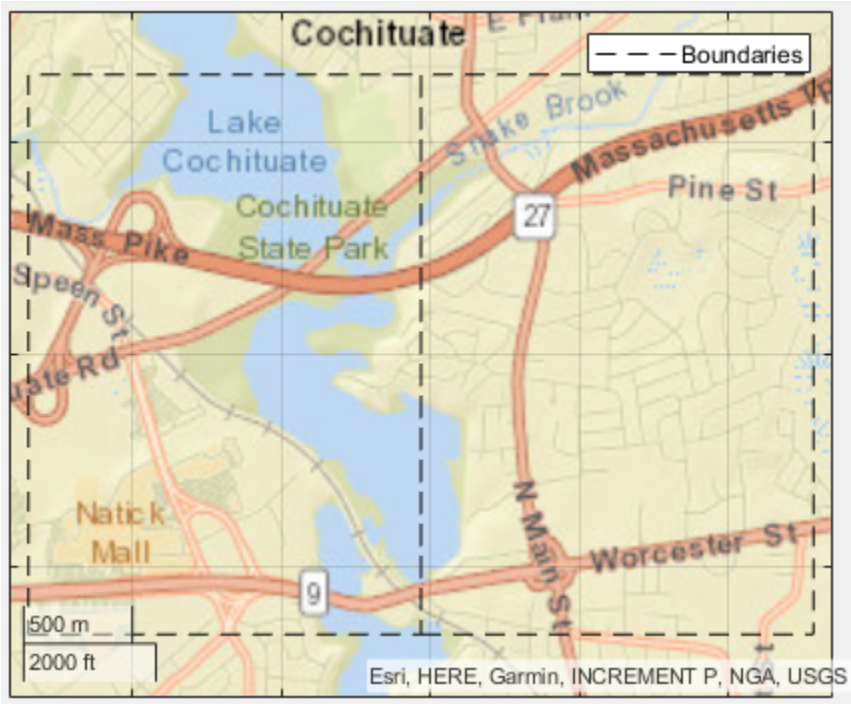
```
config = hereHDLMConfiguration('hrn:here:data::olp-here-had:here-hdlm-protobuf-na-2');
```

Readers created with this configuration search for map data from only the specified catalog.

Configuring a HERE HDLM reader is optional. If you do not specify a configuration, the reader defaults to searching for map tiles across all catalogs. The reader returns map data from the latest version of the catalog in which those tiles were found.

Create Reader for Specific Map Tiles

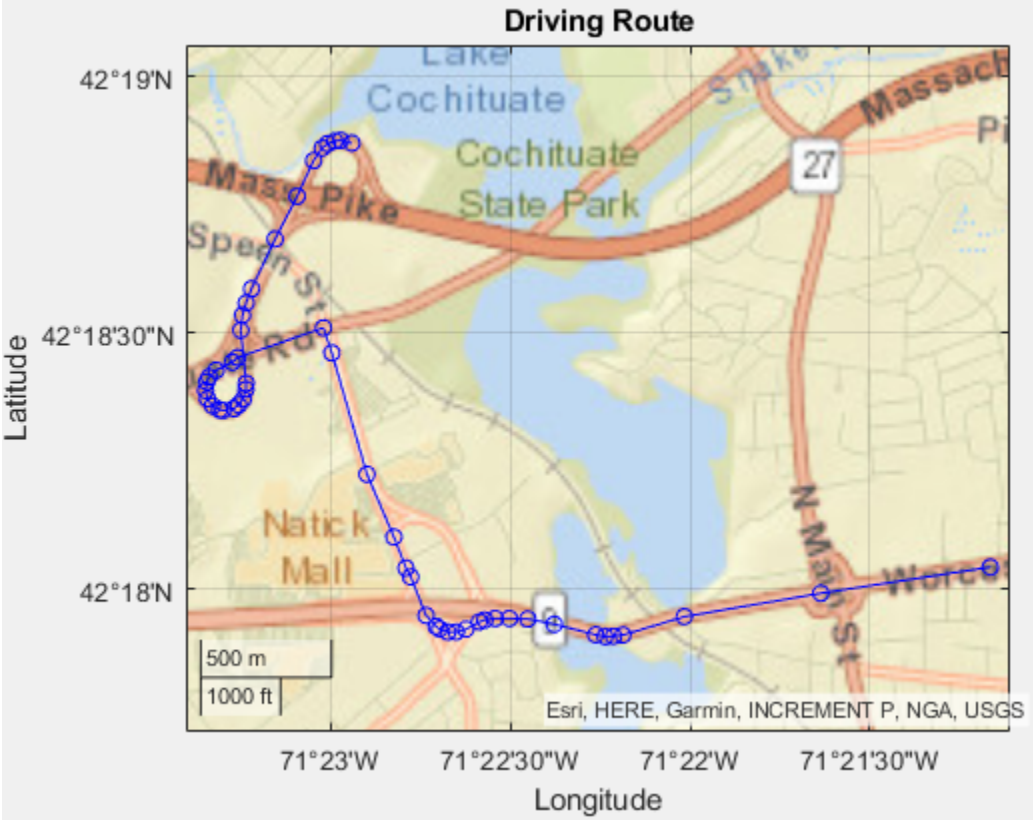
The `hereHDLMReader` object reads HERE HDLM data from a selection of map tiles. By default, these map tiles are set to a zoom level of 14, which corresponds to a rectangular area of about 5–10 square kilometers.



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

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

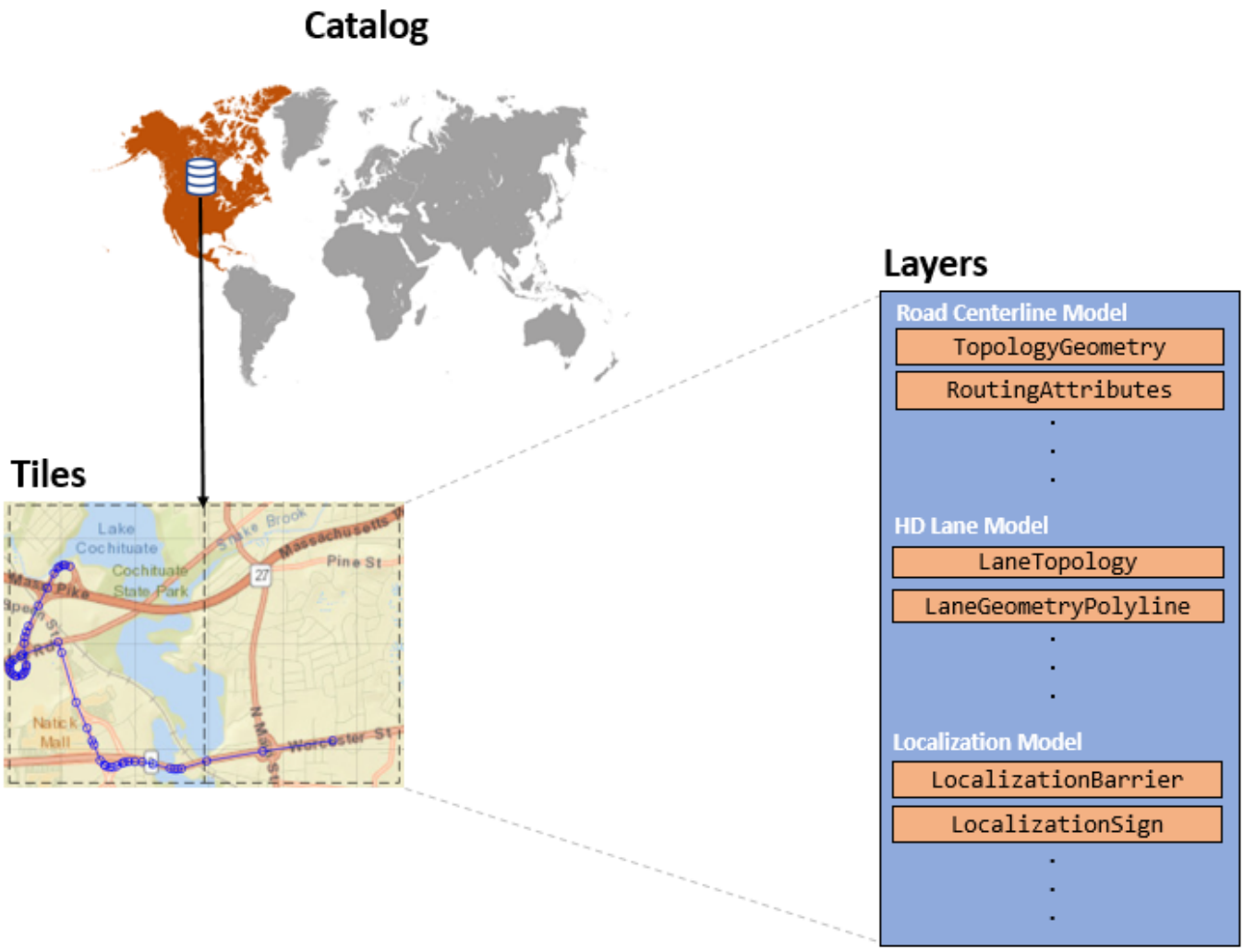
```
route = load('geoSequenceNatickMA.mat');
lat = route.latitude;
lon = route.longitude;

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



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

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

This reader enables you to read map data for the tiles that this 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, HD Lane Model, and HD Localization Model.



For more details on the layers in these models, see “HERE HD Live Map Layers” on page 4-15.

Read Map Layer Data

The `read` function reads data for the selected map tiles. The map data is returned as a series of layer objects. Read data from the layer containing the topology geometry of the road.

```
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 tile that you selected using the input `hereHDLMReader` object. The IDs of these map tiles are stored in the `TileIds` property of the reader. Inspect the properties of the map layer object for the first map tile. Your catalog version and map data might differ from what is shown here.

```
topology(1)
```

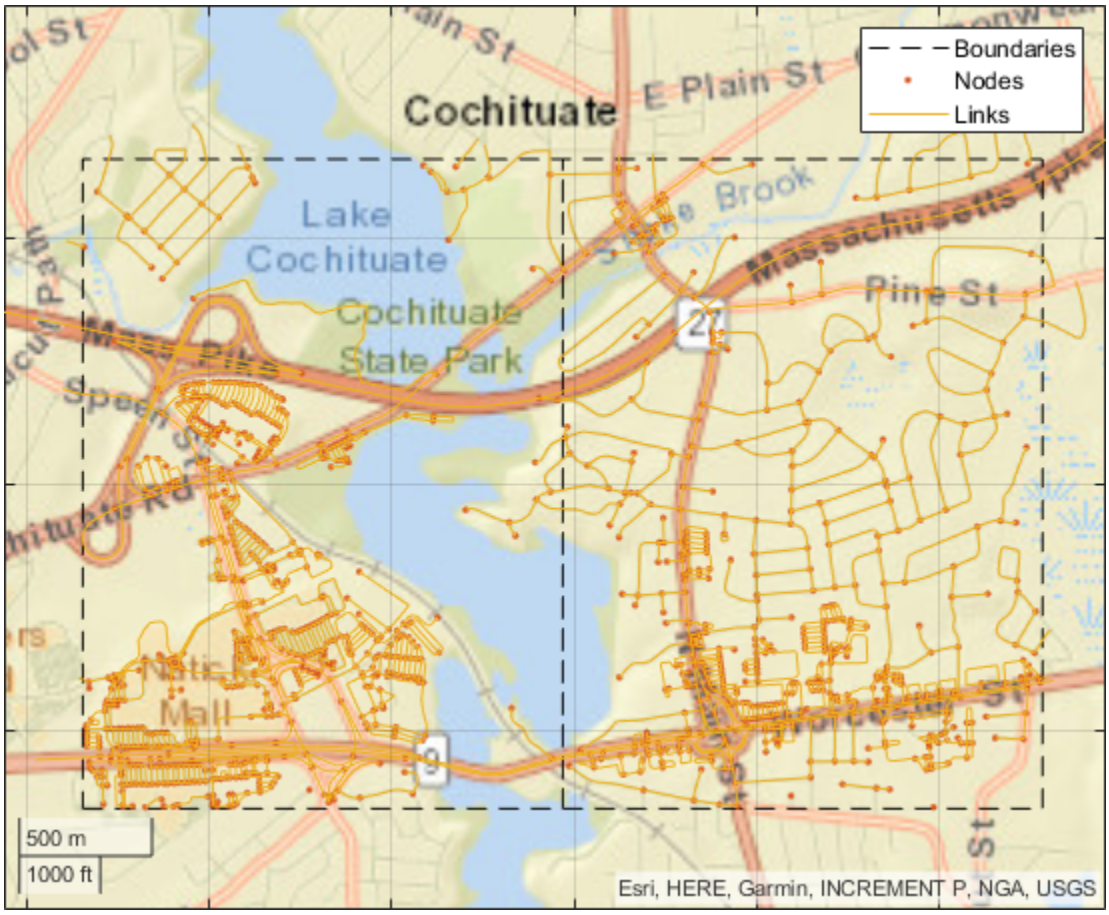
```
ans =

  TopologyGeometry with properties:

   Data:
                   HereTileId: 321884279
          IntersectingLinkRefs: [42×1 struct]
           LinksStartingInTile: [905×1 struct]
                   NodesInTile: [635×1 struct]
    TileCenterHere2dCoordinate: [42.3083 -71.3782]

   Metadata:
                      Catalog: 'hrn:here:data::olp-here-had:here-hdlm-protobuf-na-2'
                CatalogVersion: 3321
```

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 more details about the layer objects, see the `layerData` output argument description on the `read` function reference page.

Visualize Map Layer Data

To visualize the data of map 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 outside the boundaries of the specified map tiles, 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

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

More About

- “HERE HD Live Map Layers” on page 4-15
- “Use HERE HD Live Map Data to Verify Lane Configurations” on page 7-524
- “Import HERE HD Live Map Roads into Driving Scenario” on page 5-93

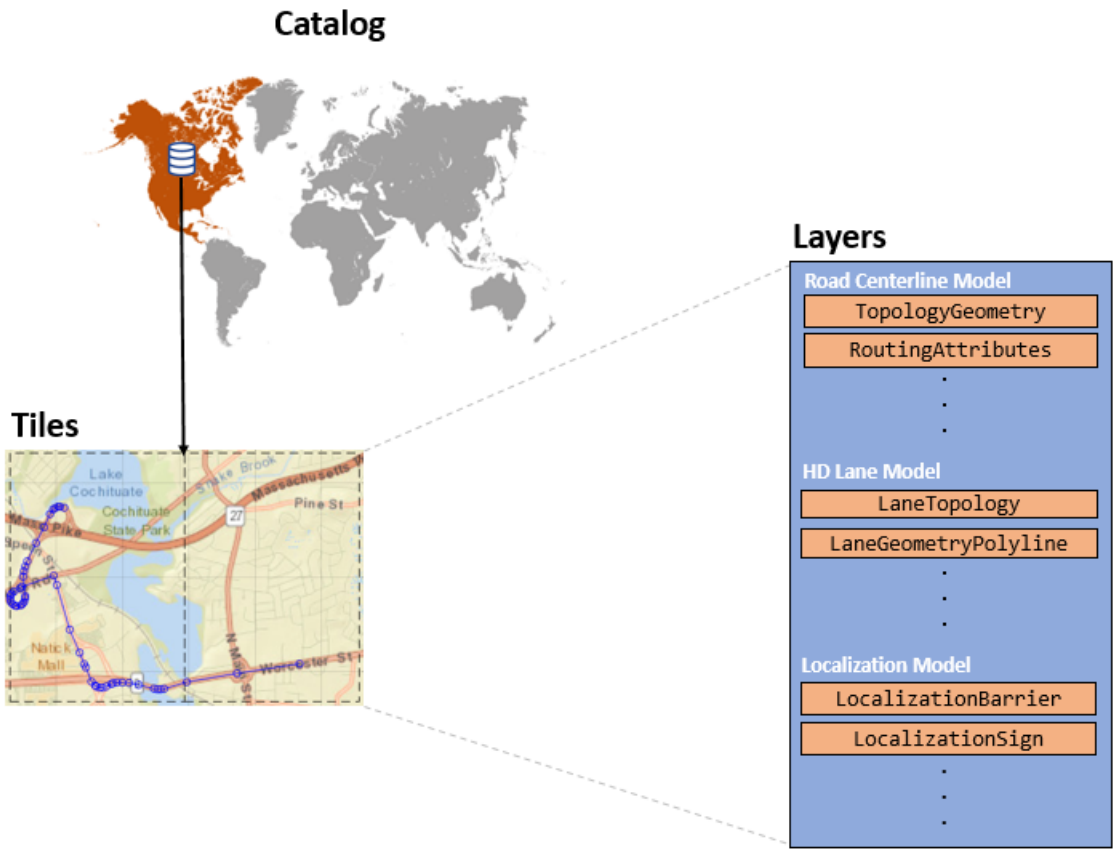
HERE HD Live Map Layers

HERE HD Live Map[2] (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 and lane-level attributes, and the barriers, signs, and poles found along roads. 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 “Read and Visualize HERE HD Live Map Data” on page 4-7.

The layers are grouped into these models:

- “Road Centerline Model” on page 4-16 — Provides road topology, shape geometry, and other road-level attributes
- “HD Lane Model” on page 4-17 — Contains lane topology, highly accurate geometry, and lane-level attributes
- “HD Localization Model” on page 4-18 — Includes multiple features, such as road signs, to support localization strategies



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

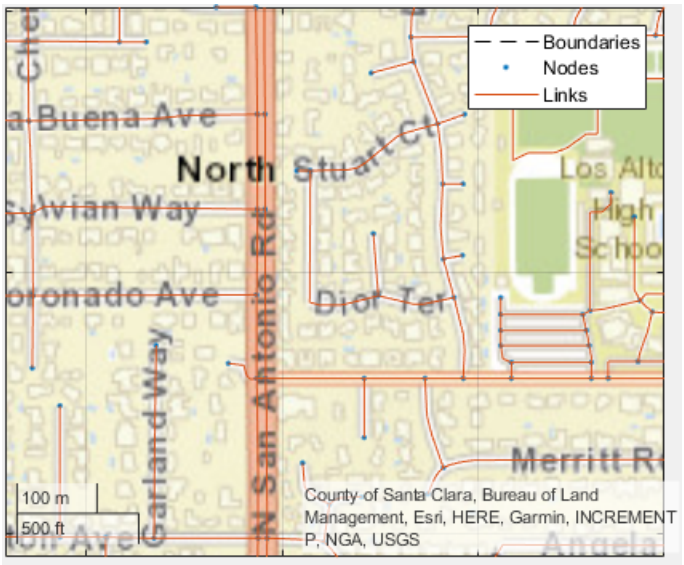
2. You need to enter into a separate agreement with HERE in order to gain access to the HDLM services and to get the required credentials (access_key_id and access_key_secret) for using the HERE Service.

function returns the layer data as an object. For more information about these layer objects, see the `read` function reference page.

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.

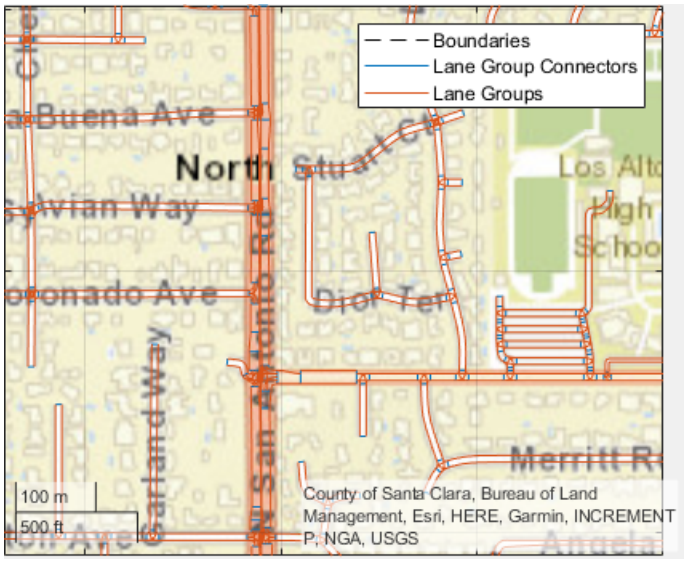
Road Centerline Model Layers	Description
`TopologyGeometry`	Topology and 2-D line geometry of the road. This layer also contains definitions of the links (streets) and nodes (intersections and dead-ends) in the map tile.
`RoutingAttributes`	Road attributes related to navigation and conditions. These attributes are mapped parametrically to the 2-D polyline geometry in the topology layer.
`RoutingLaneAttributes`	Core navigation lane attributes and conditions, such as the number of lanes in a road. These values are mapped parametrically to 2-D polylines along the road links.

Road Centerline Model Layers	Description
`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.

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.
`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, barriers, and poles, that helps autonomous vehicles accurately locate where they are within a road network.

This table shows the map layers of the HD Localization Model that a `hereHDLMReader` object can read. The reader does not support visualization of this layer data.

HD Localization Model Layers	Description
`LocalizationBarrier`	Positions, dimensions, and attributes of barriers such as guardrails and Jersey barriers found along roads
`LocalizationPole`	Positions, dimensions, and attributes of traffic signal poles and other poles found along or hanging over roads
`LocalizationSign`	Positions, dimensions, and attributes of traffic-sign faces found along roads

See Also

`hereHDLMReader` | `plot` | `read`

More About

- “Read and Visualize HERE HD Live Map Data” on page 4-7
- “Use HERE HD Live Map Data to Verify Lane Configurations” on page 7-524

Rotations, Orientations, and Quaternions for Automated Driving

A quaternion is a four-part hypercomplex number used to describe three-dimensional rotations and orientations. Quaternions have applications in many fields, including aerospace, computer graphics, and virtual reality. In automated driving, sensors such as inertial measurement units (IMUs) report orientation readings as quaternions. To use this data for localization, you can capture it using a `quaternion` object, perform mathematical operations on it, or convert it to other rotation formats, such as Euler angles and rotation matrices.

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

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

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

Quaternion Format

A quaternion number is represented in this form:

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

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

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

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

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

Quaternion Creation

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

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

```
q =

  quaternion

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

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

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

```
qArray =

  2x2 quaternion array


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

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

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

```
qRandom =

  4x1 quaternion array


     0.17446 +  0.59506i -  0.73295j +  0.27976k
     0.21908 -  0.89875i -    0.298j +  0.23548k
      0.6375 +  0.49338i -  0.24049j +  0.54068k
     0.69704 - 0.060589i +  0.68679j -  0.19695k
```

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

```
qRandom(3)
```

```
ans =

  quaternion


     0.6375 + 0.49338i - 0.24049j + 0.54068k
```

Reshape the quaternion array.

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

```
ans =

  2x2 quaternion array


     0.17446 +  0.59506i -  0.73295j +  0.27976k       0.6375 +  0.49338i -  0.24049j +  0.54068
     0.21908 -  0.89875i -    0.298j +  0.23548k     0.69704 - 0.060589i +  0.68679j -  0.19695
```

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

```
[qRandom; q]
```

```
ans =

  5x1 quaternion array


      0.17446 +  0.59506i -  0.73295j +  0.27976k
      0.21908 -  0.89875i -    0.298j +  0.23548k
       0.6375 +  0.49338i -  0.24049j +  0.54068k
      0.69704 - 0.060589i +  0.68679j -  0.19695k
            1 +        2i +        3j +        4k
```

Quaternion Math

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

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

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

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

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

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

```
q1 + q2
```

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

Subtraction of quaternions works similar to addition of quaternions.

```
q1 - q2
```

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

Because the complex elements of quaternions must satisfy the equation

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

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

```
q1 * q2
```

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

```
q2 * q1
```

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

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

```
q1 ./ q2
```

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

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

```
q1 .\ q2
```

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

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

```
conj(q1)
```

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

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

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

Normalize a quaternion.

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

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

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

```
norm(qNormalized)
```

```
ans = 1.0000
```

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

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

ans = 3×3

   -0.6667    0.6667    0.3333
    0.1333   -0.3333    0.9333
    0.7333    0.6667    0.1333


inv(rotmat(qNormalized,'point'))

ans = 3×3

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

Extract Quaternions from Transformation Matrix

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

Postmultiply Format

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

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

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

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

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

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

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

Premultiply Format

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

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

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

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

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

Convert from Postmultiply to Premultiply Format

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

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

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

tform = rigid3d(rot,trans)
```

```
tform =
  rigid3d with properties:

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

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

```
rotPost = tform.Rotation
```

```
rotPost = 3×3

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

```
rotPre = rotPost.'
```

```
rotPre = 3×3

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

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

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

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

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

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

```
rotatedPointQuaternion = 1×3

   -0.1340    2.2321    3.0000
```

```
rotatedPointRotationMatrix = point * rotPost
```

```
rotatedPointRotationMatrix = 1×3

   -0.1340    2.2321    3.0000
```

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

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

```
ans = 3×3

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

See Also

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

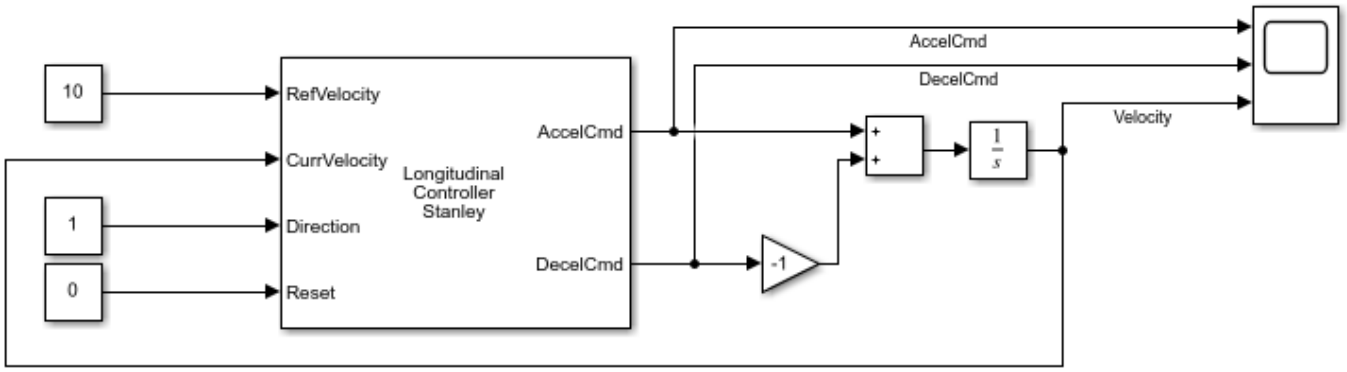
More About

- “Build a Map from Lidar Data” on page 7-539
- “Build a Map from Lidar Data Using SLAM” on page 7-559

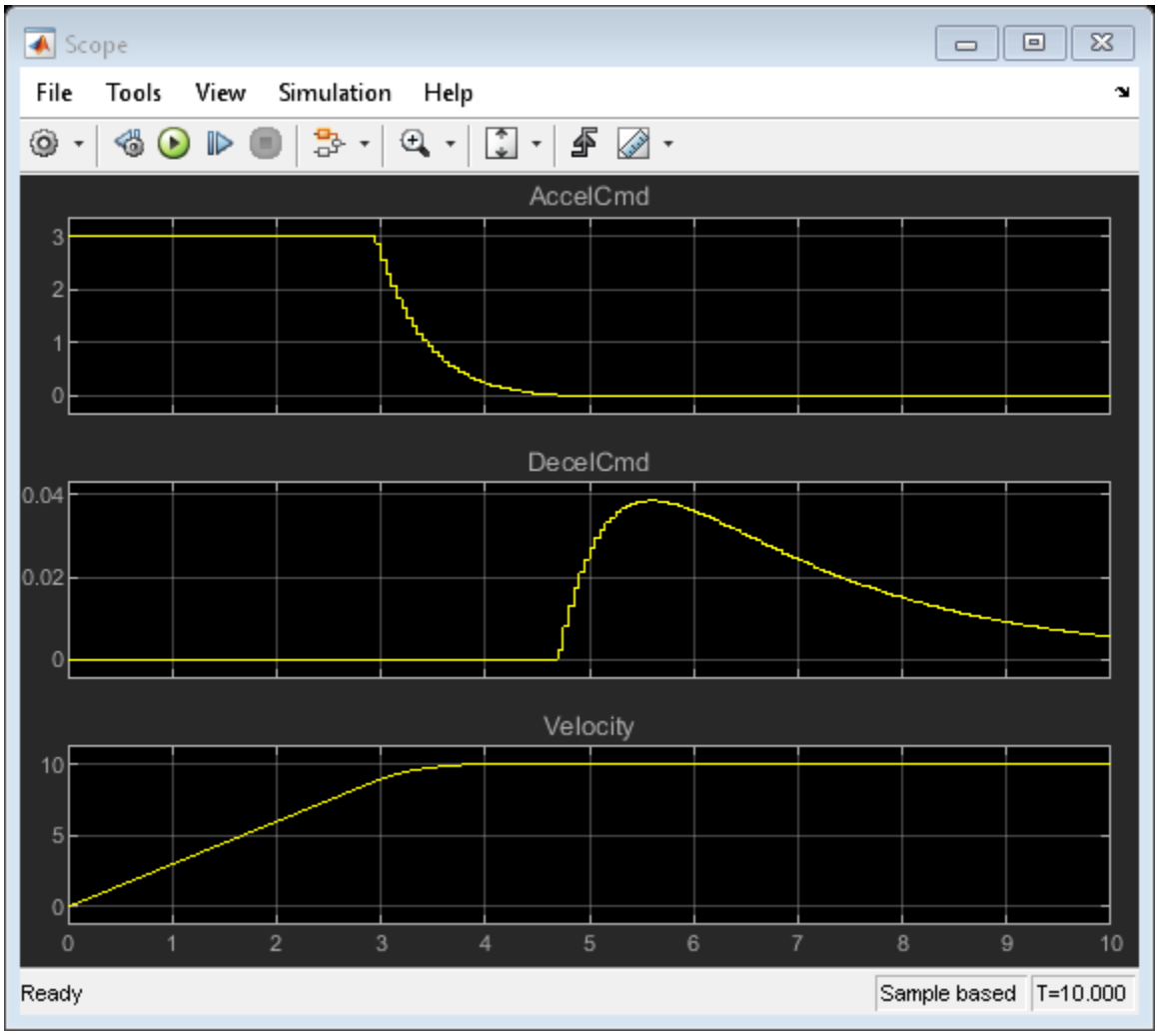
Control Vehicle Velocity

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

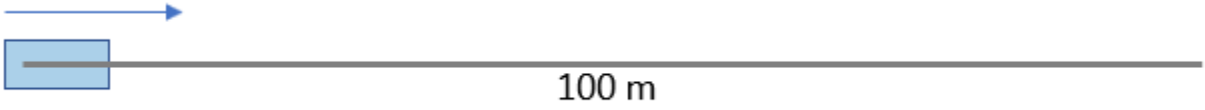
Lateral Controller Stanley | Longitudinal Controller Stanley

More About

- “Automated Parking Valet in Simulink” on page 7-493

Velocity Profile of Straight Path

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



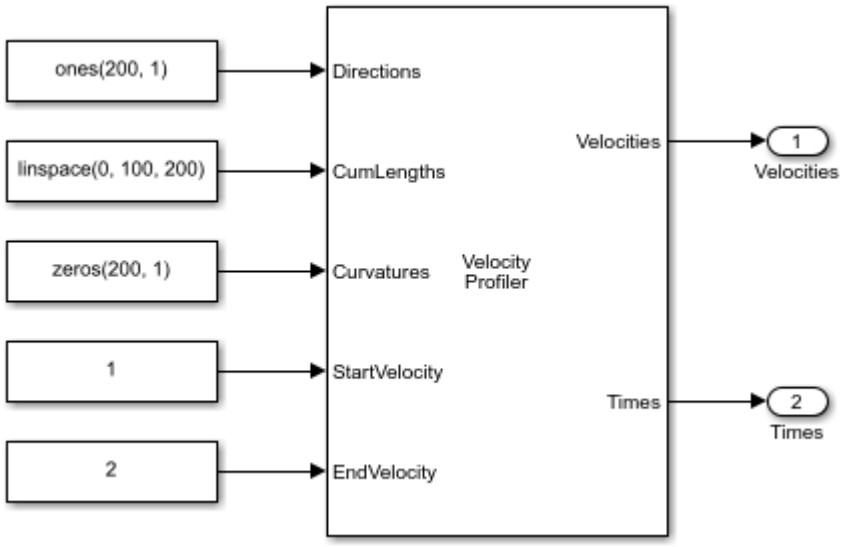
The Velocity Profiler block generates velocity profiles based on the speed, acceleration, and jerk constraints that you specify using parameters. You can use the generated velocity profile as the input reference velocities of a vehicle controller.

This model is for illustrative purposes and does not show how to use the Velocity Profiler block in a complete automated driving model. To see how to use this block in such a model, see the “Automated Parking Valet in Simulink” on page 7-493 example.

Open and Inspect Model

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

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



The first three inputs specify information about the driving path.

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

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

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

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

Generate Velocity Profile

Simulate the model to generate the velocity profile.

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

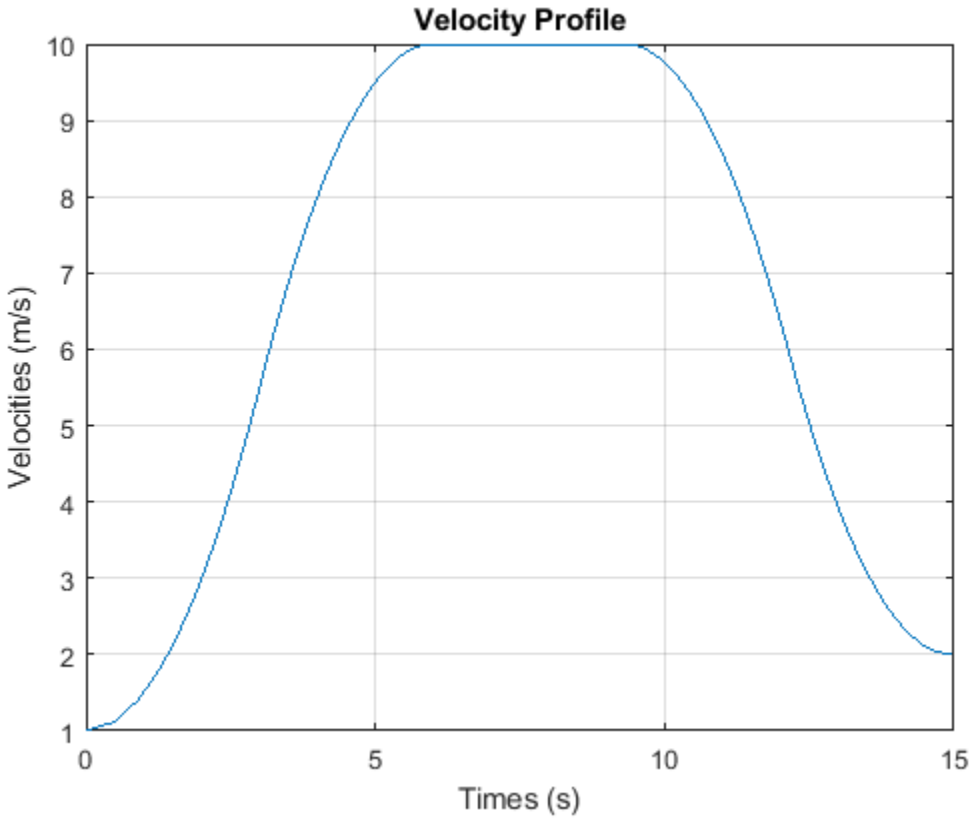
The output velocity profile is a sequence of velocities along the path that meet the speed, acceleration, and jerk constraints specified in the parameters of the Velocity Profiler block.

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

Visualize Velocity Profile

Use the simulation output to plot the velocity profile.

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

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

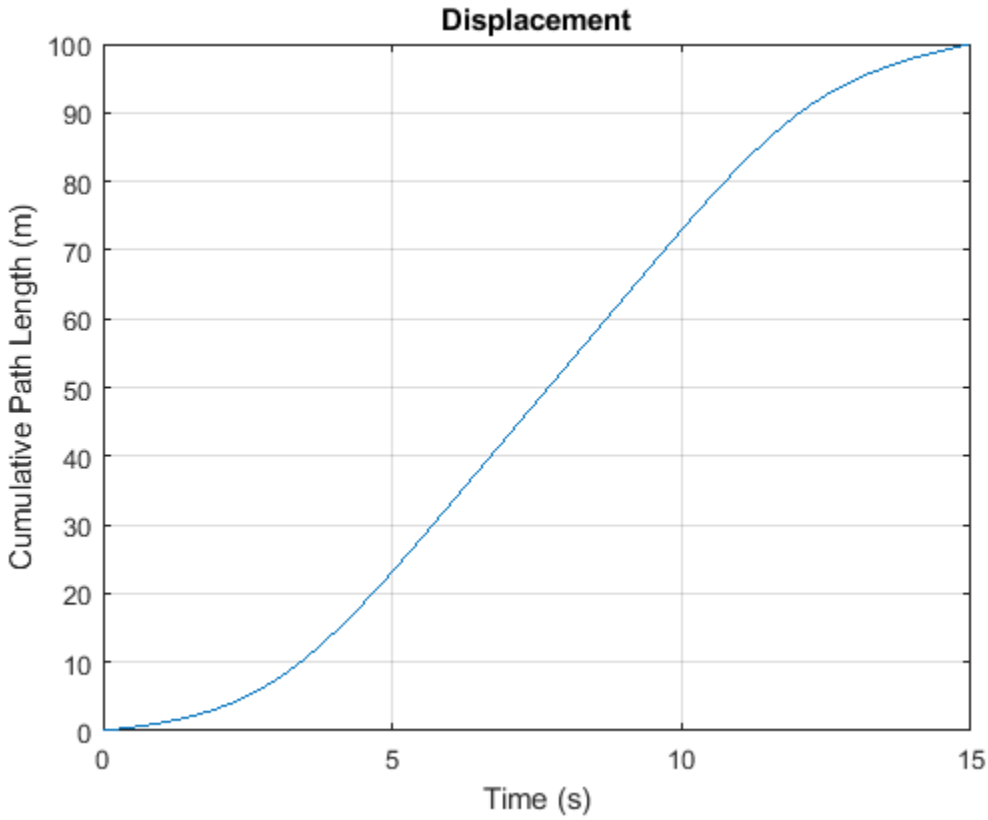


A vehicle that follows this velocity profile:

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

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

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



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

See Also

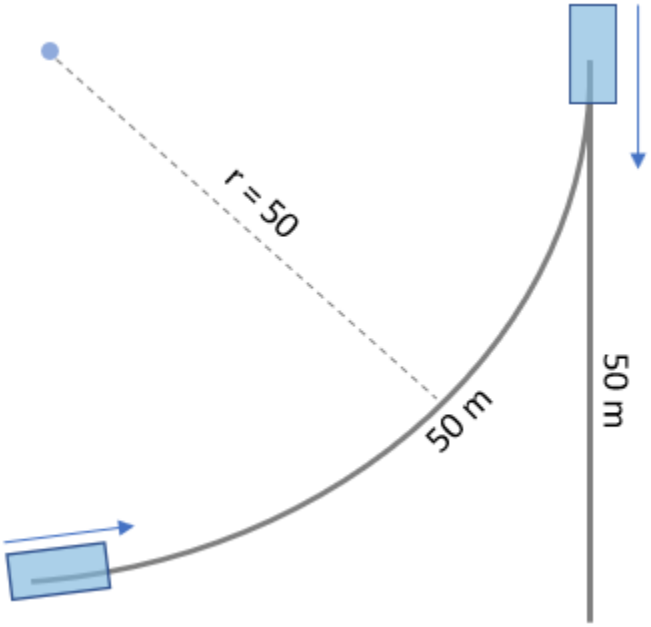
Path Smoother Spline | Velocity Profiler

More About

- “Velocity Profile of Path with Curve and Direction Change” on page 4-32
- “Automated Parking Valet in Simulink” on page 7-493

Velocity Profile of Path with Curve and Direction Change

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



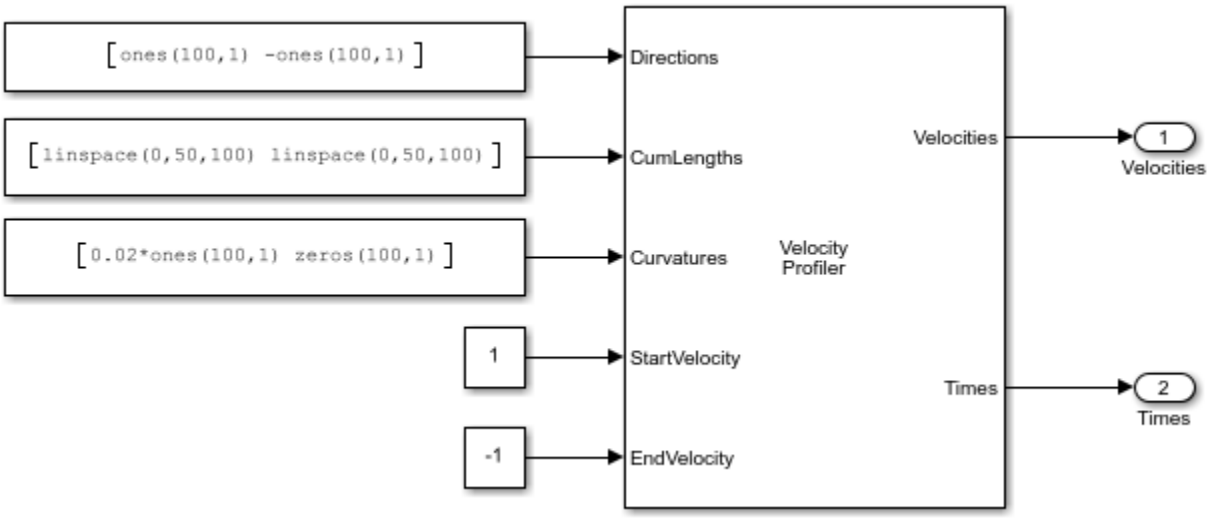
The Velocity Profiler block generates velocity profiles based on the speed, acceleration, and jerk constraints that you specify using parameters. You can use the generated velocity profile as the input reference velocities of a vehicle controller.

This model is for illustrative purposes and does not show how to use the Velocity Profiler block in a complete automated driving model. To see how to use this block in such a model, see the “Automated Parking Valet in Simulink” on page 7-493 example.

Open and Inspect Model

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

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



The first three inputs specify information about the driving path.

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

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

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

Generate Velocity Profile

Simulate the model to generate the velocity profile.

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

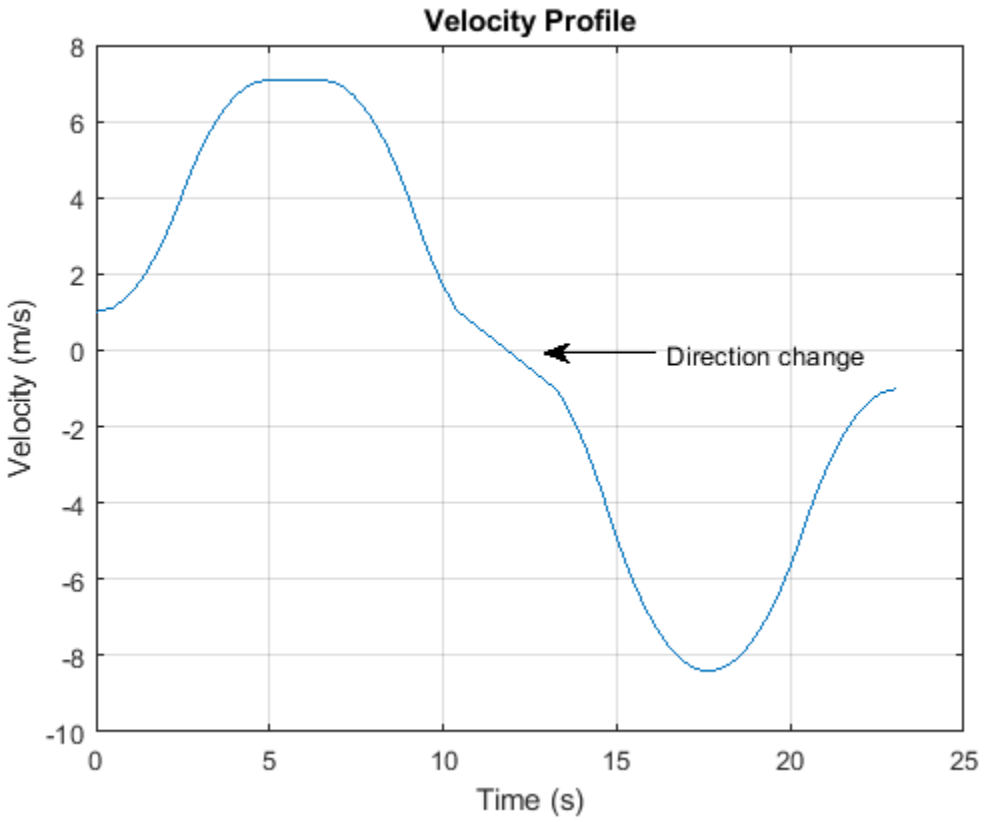
The output velocity profile is a sequence of velocities along the path that meet the speed, acceleration, and jerk constraints specified in the parameters of the Velocity Profiler block.

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

Visualize Velocity Profile

Use the simulation output to plot the velocity profile.

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

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



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

A vehicle that follows this velocity profile:

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

5 Decelerates until it reaches its ending velocity

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

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

See Also

Path Smoother Spline | Velocity Profiler

More About

- “Velocity Profile of Straight Path” on page 4-28
- “Automated Parking Valet in Simulink” on page 7-493

Cuboid Driving Scenario Simulation

Create Driving Scenario Interactively and Generate Synthetic Sensor Data

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

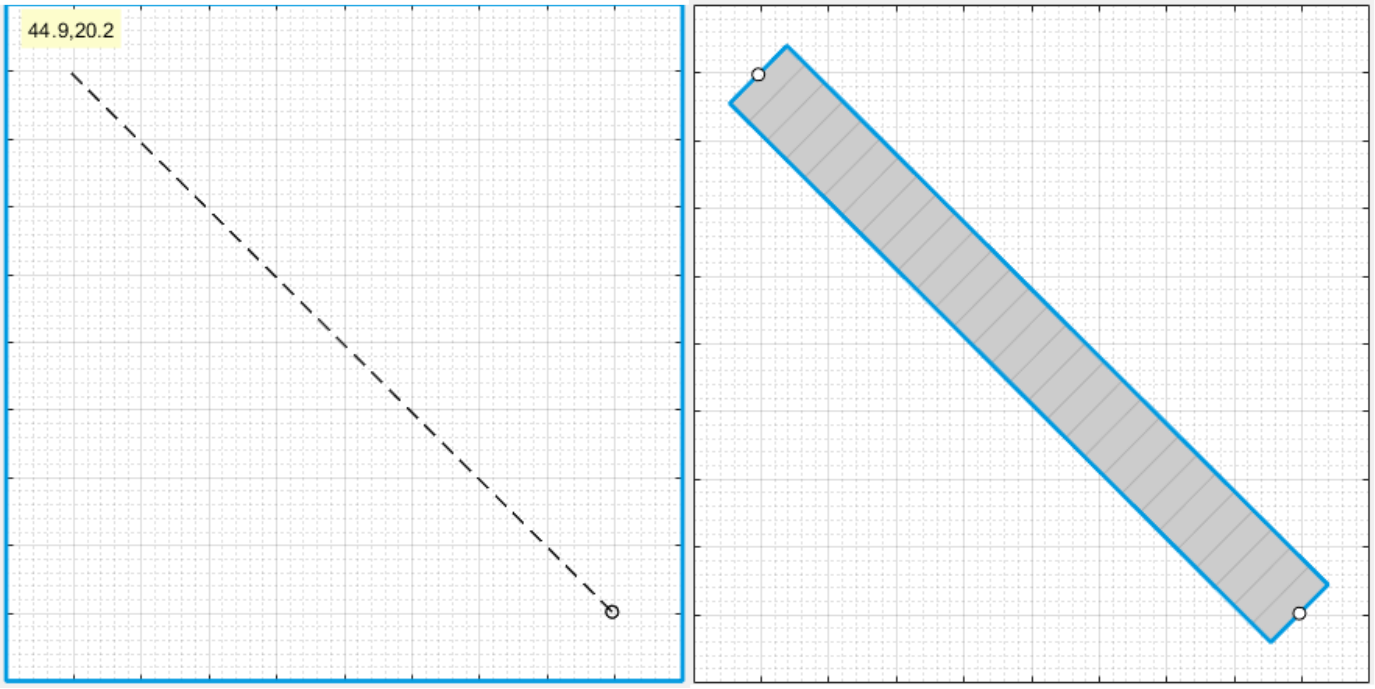
This example shows the entire workflow for creating a scenario and generating synthetic sensor data. Alternatively, you can generate sensor data from prebuilt scenarios. See “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-19.

Create Driving Scenario

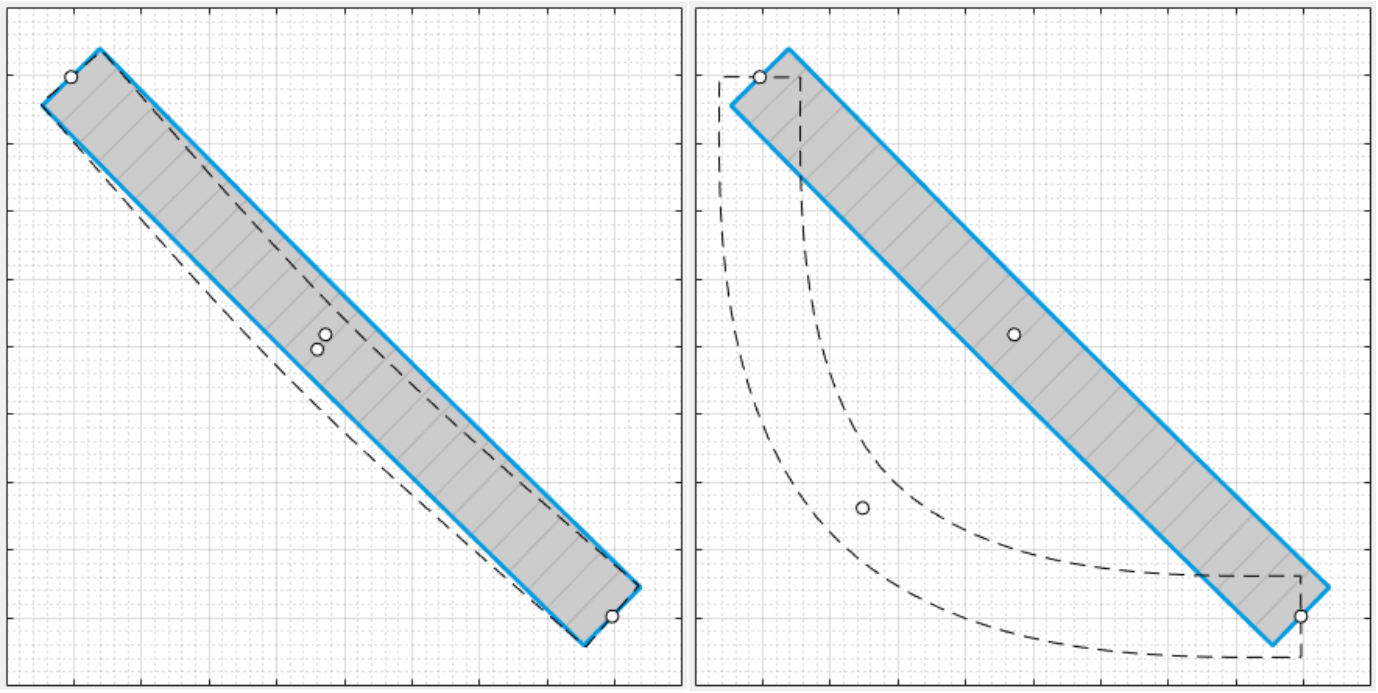
To open the app, at the MATLAB command prompt, enter `drivingScenarioDesigner`.

Add a Road

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



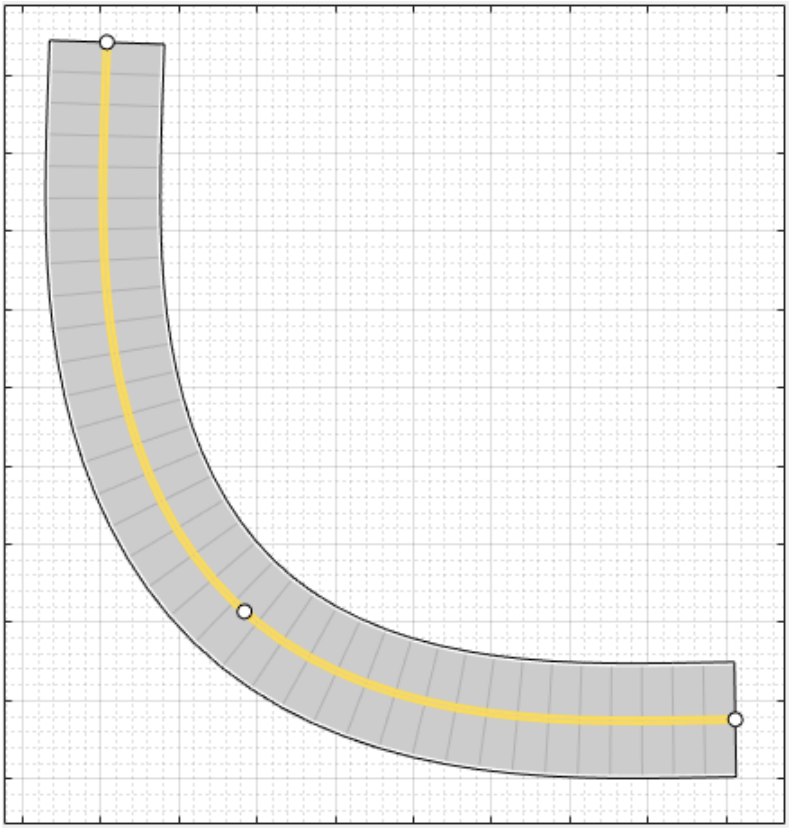
To make the road curve, add a road center around which to curve it. Right-click the middle of the road and select **Add Road Center**. Then drag the added road center to one of the empty corners of the canvas.



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

Add Lanes

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



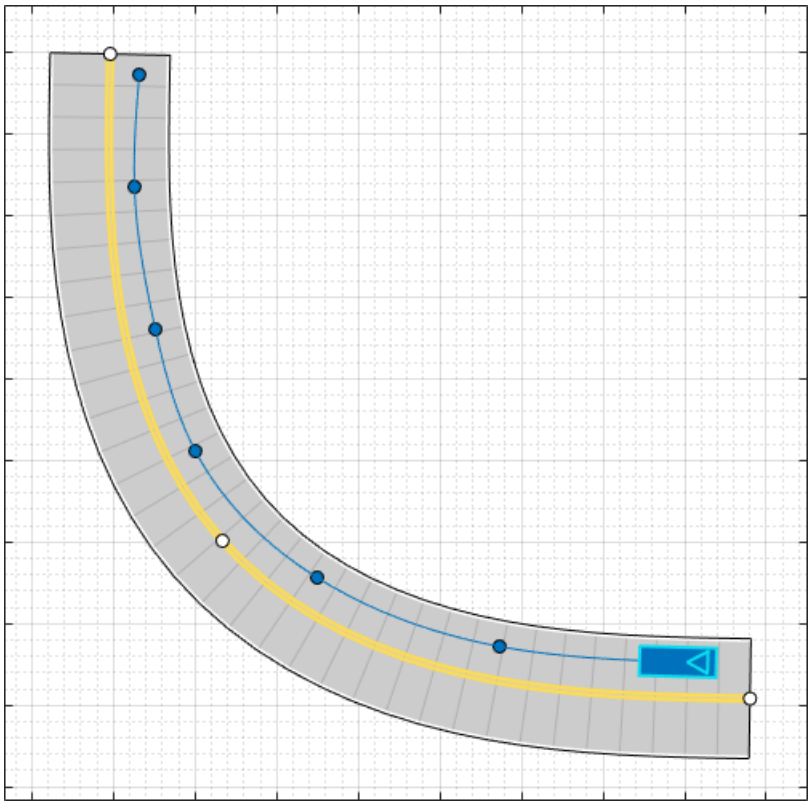
The white, solid lanes markings on either edge of the road indicate the road shoulder. The yellow, double-solid lane marking in the center indicates that the road is a two-way road. To inspect or modify these lanes, from the **Lane Marking** list, select one of the lanes and modify the lane parameters.

Add Vehicles

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

Add Ego Vehicle

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

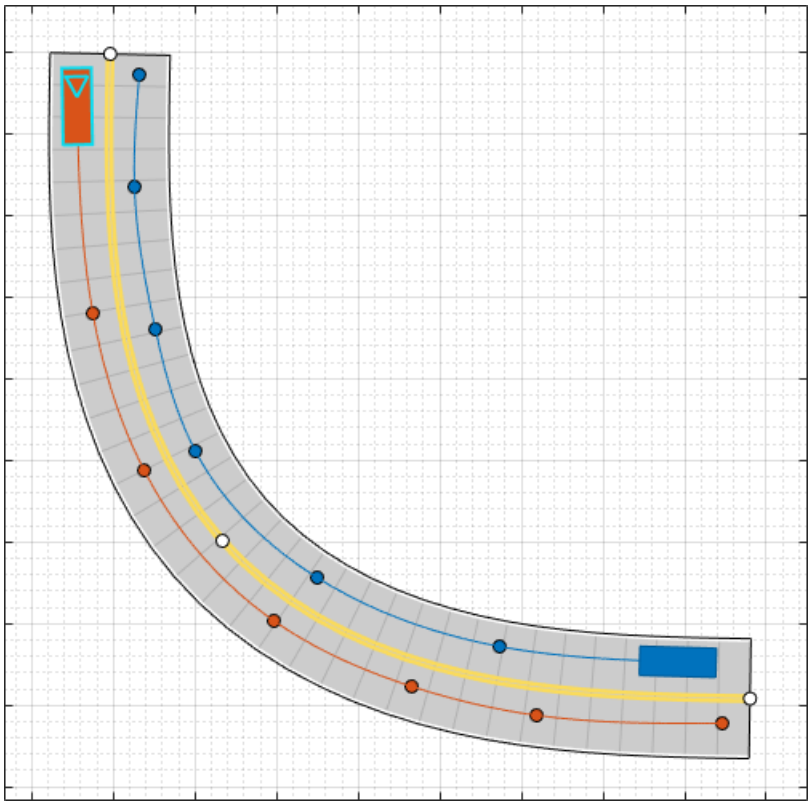


The triangle indicates the pose of the vehicle, with the origin located at the center of the rear axle of the vehicle.

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, set the velocity between waypoints in the **v (m/s)** column of the **Waypoints, Speeds, Wait Times, and Yaw** table.

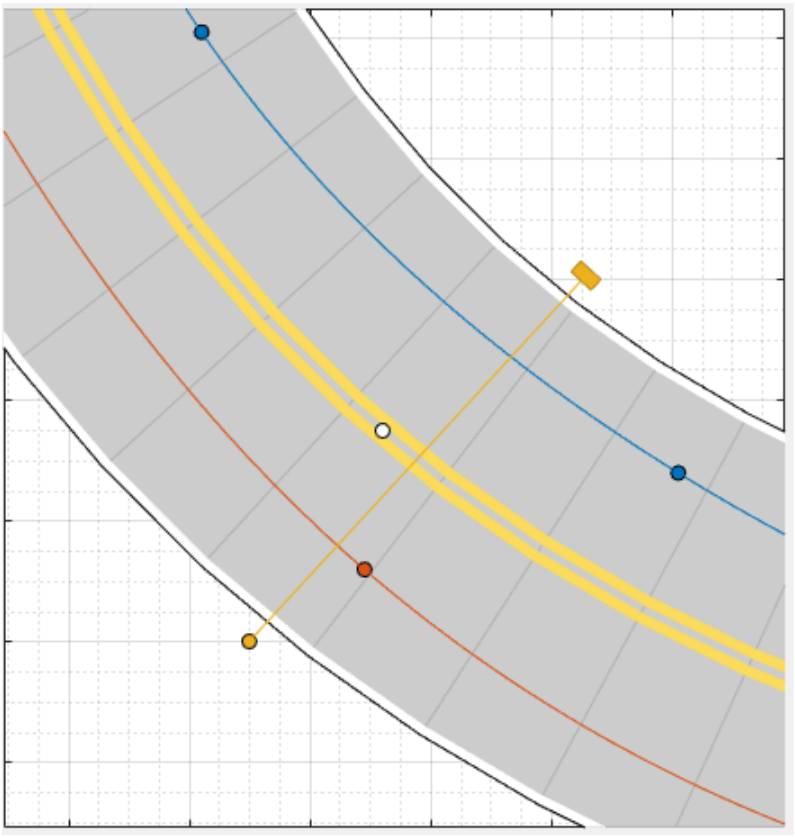
Add Second Car

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



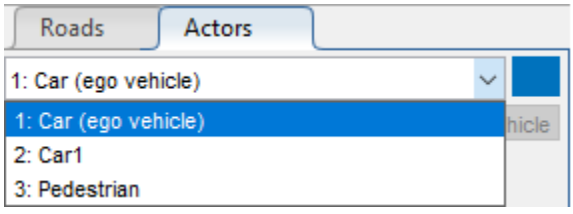
Add a Pedestrian

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



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

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



For example, if the cars are colliding with the pedestrian, in the **v (m/s)** column of the **Waypoints, Speeds, Wait Times, and Yaw** table, adjust the speeds of the cars or the pedestrian. Alternatively, in the **wait (s)** column of the table, set a wait time for the cars at the waypoint before the pedestrian crosses the street.

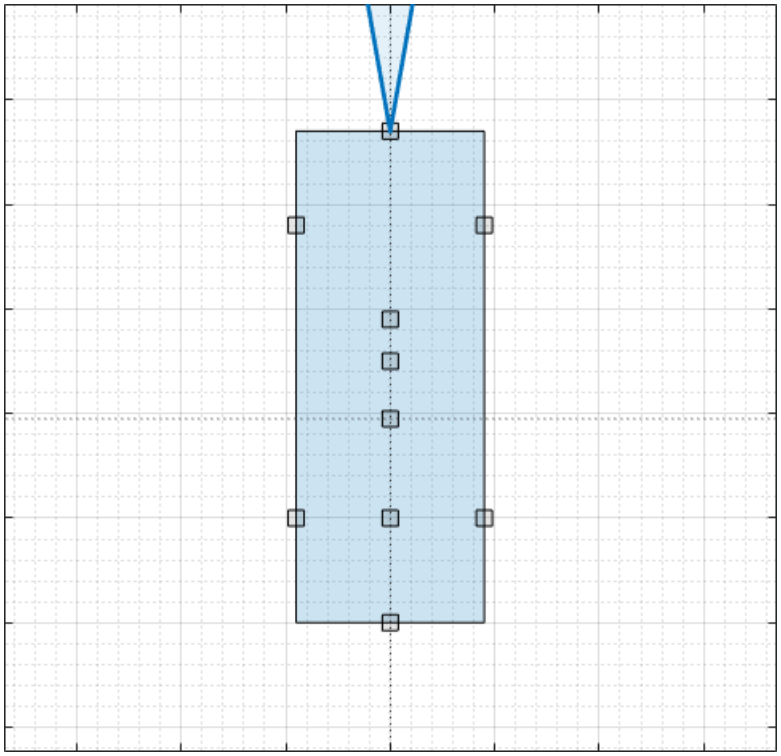
By default, the simulation ends when the first actor completes its trajectory. To end the simulation only after all actors complete their trajectories, on the app toolstrip, first click **Settings**. Then, set **Stop Condition** to `Last actor stops`.

Add Sensors

Add camera, radar, and lidar sensors to the ego vehicle. Use these sensors to generate detections and point cloud data from the scenario.

Add Camera

On the app toolstrip, click **Add Camera**. The sensor canvas shows standard locations at which to place sensors. Click the frontmost predefined sensor location to add a camera sensor to the front bumper of the ego vehicle.



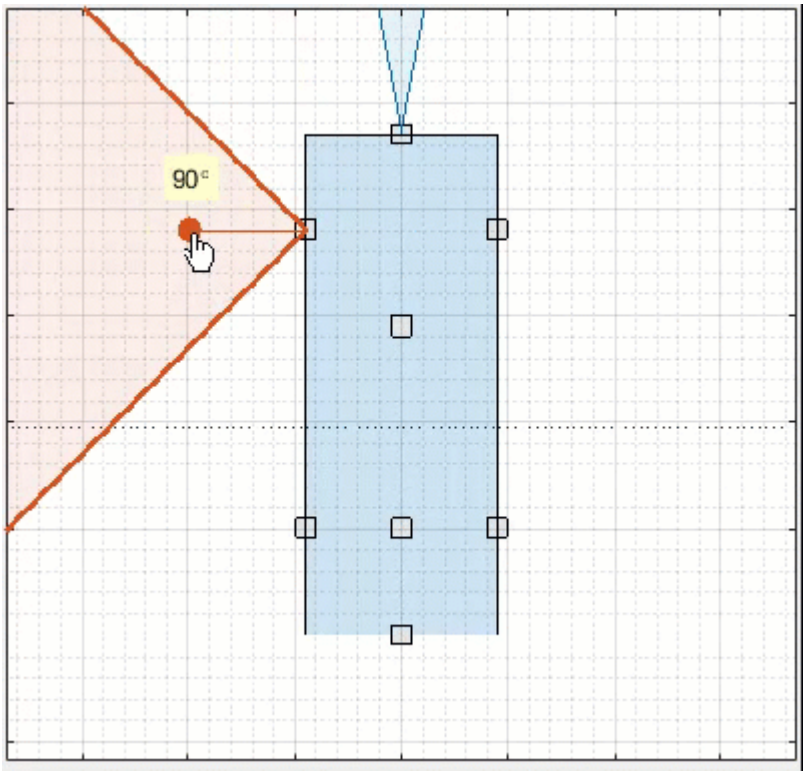
To place sensors more precisely, you can disable snapping options. In the bottom-left corner of the sensor canvas, click the Configure the Sensor Canvas button .

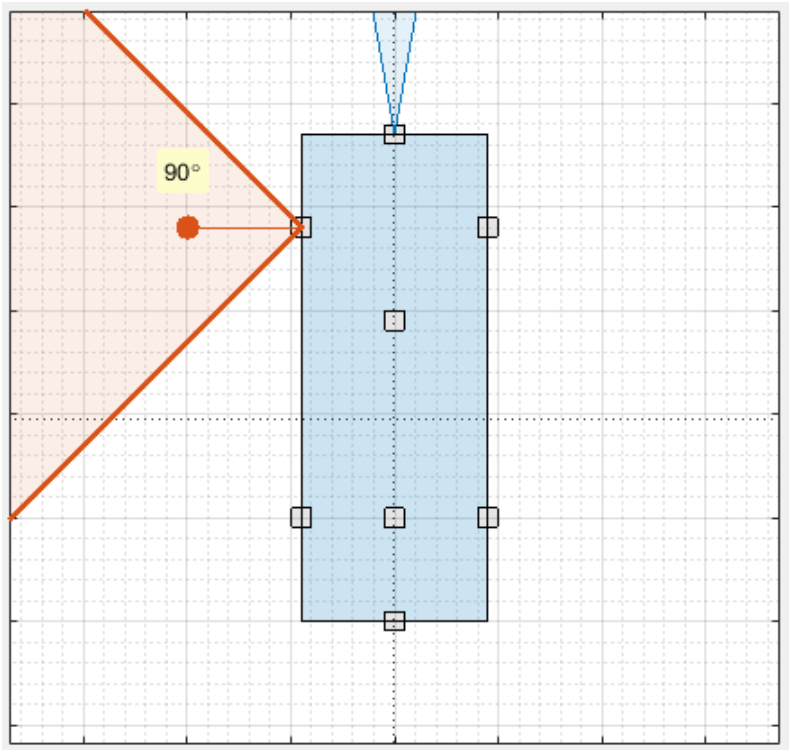
By default, the camera detects only actors and not lanes. To enable lane detections, on the **Sensors** tab in the left pane, expand the **Detection Parameters** section and set **Detection Type** to `Objects & Lanes`. Then expand the **Lane Settings** section and update the settings as needed.

Add Radars

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 sensors.

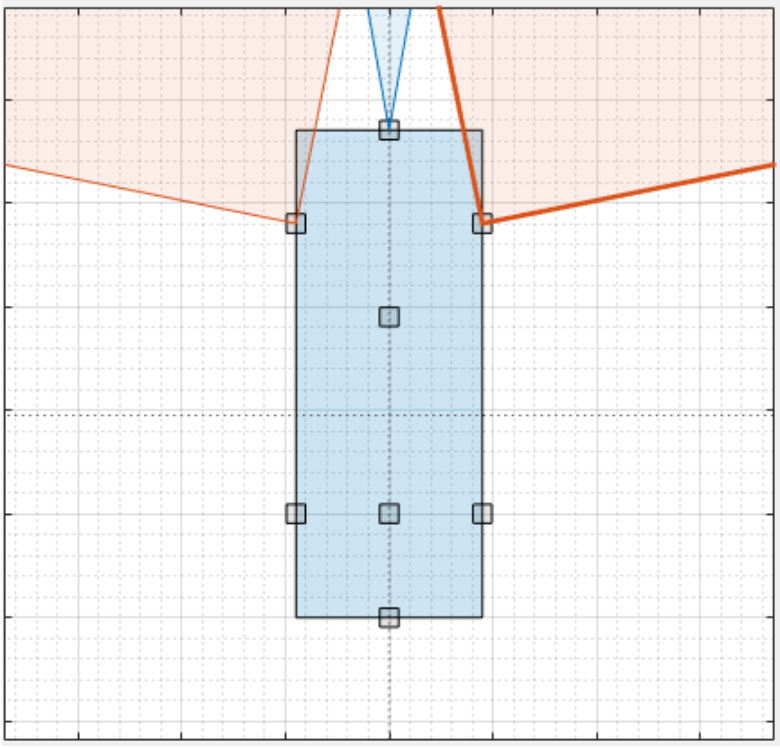
Tilt the radar sensor toward the front of the car. Move your cursor over the coverage area, then click and drag the angle marking.





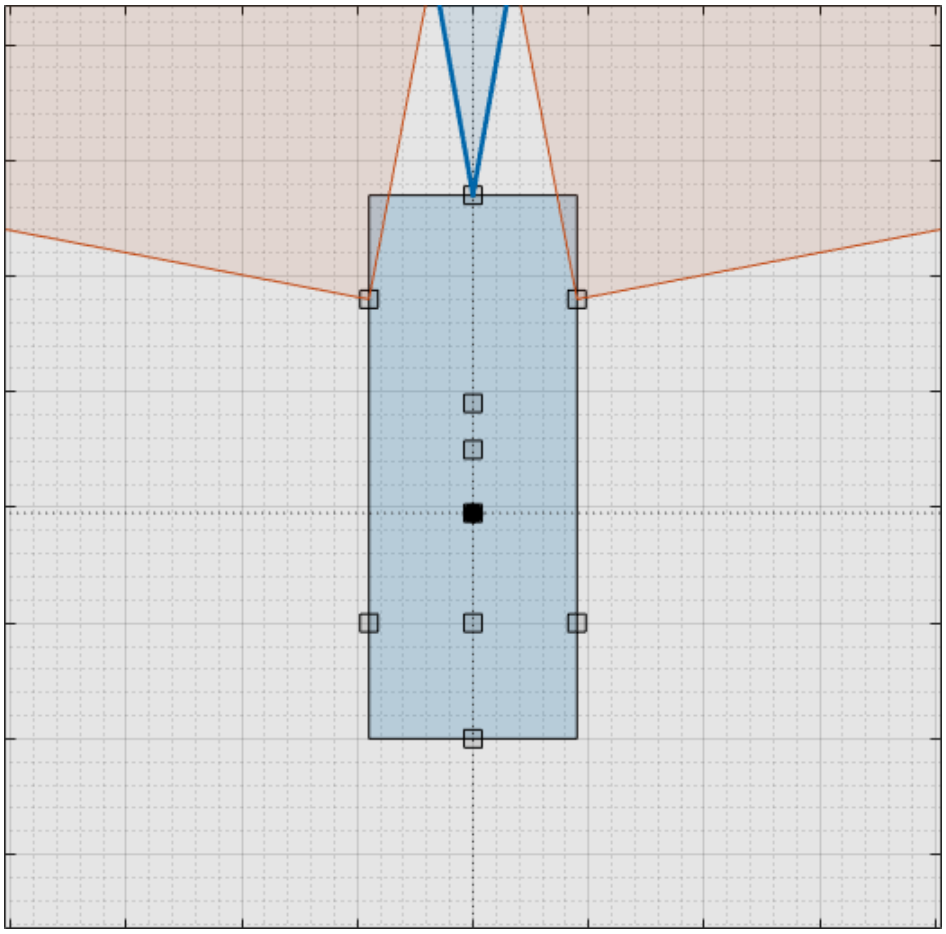
Add an identical radar sensor to the front-right wheel. Right-click the sensor on the front-left wheel and click **Copy**. Then right-click the predefined sensor location for the front-right wheel and click

Paste. The orientation of the copied sensor mirrors the orientation of the sensor on the opposite wheel.

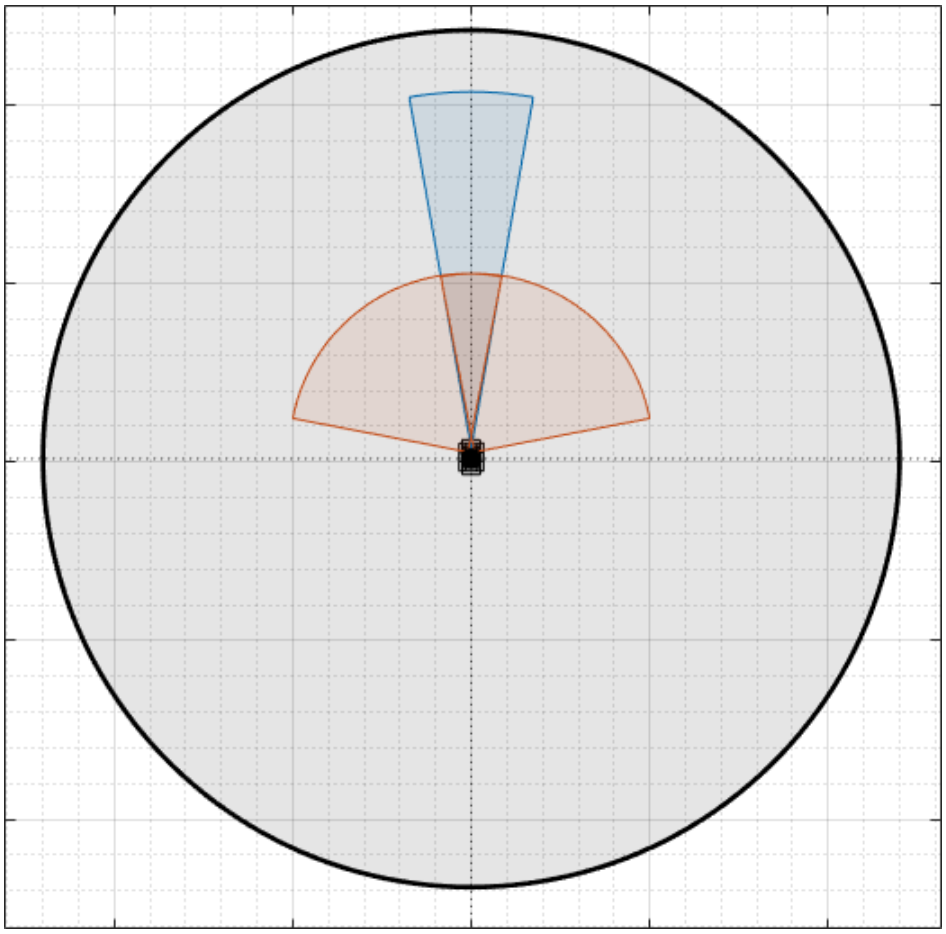


Add Lidar

Snap a lidar sensor to the center of the roof of the vehicle. Right-click the predefined sensor location for the roof center and select **Add Lidar**.

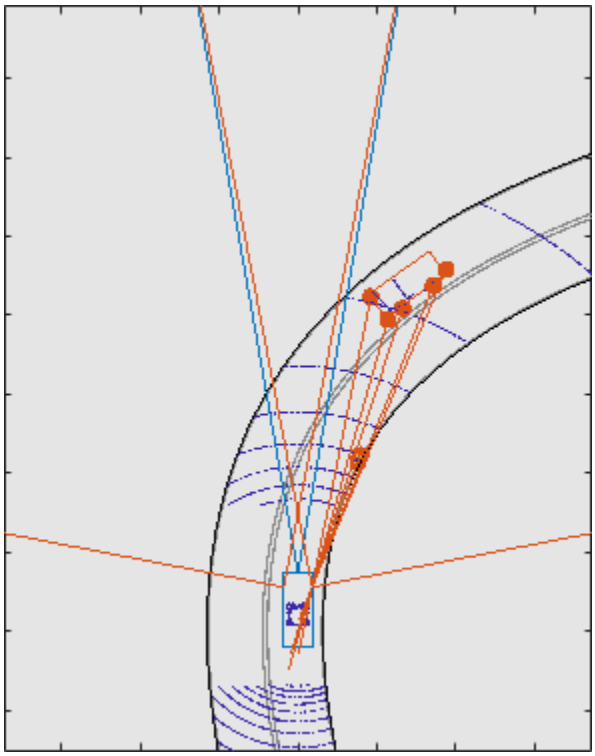


The lidar sensor appears in black. The gray surrounding the vehicle is the coverage area of the sensor. Zoom out to see the full view of the coverage areas for the different sensors.

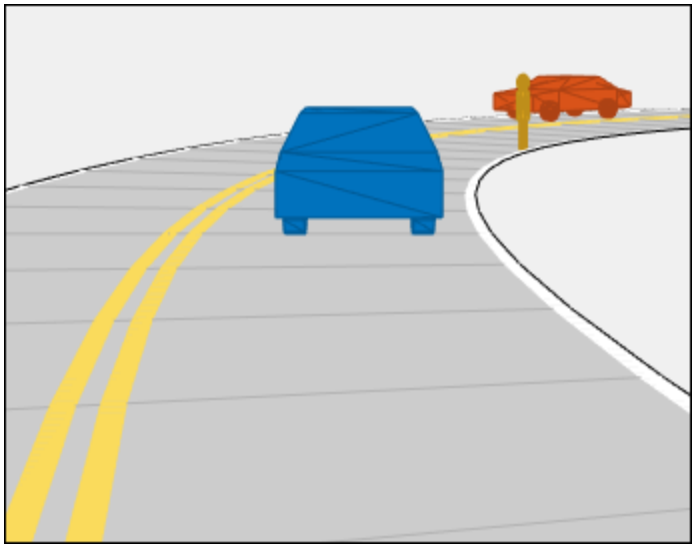


Generate Synthetic Sensor Data

To generate data from the sensors, click **Run**. As the scenario runs, The **Bird's-Eye Plot** displays the detections and point cloud data.



The **Ego-Centric View** displays the scenario from the perspective of the ego vehicle.



Because you specified a lidar sensor, both the **Ego-Centric View** and **Bird's-Eye Plot** display the mesh representations of actors instead of the cuboid representations. The lidar sensors use these more detailed representations of actors to generate point cloud data. The **Scenario Canvas** still displays only the cuboid representations. The radar and vision sensors base their detections on the cuboid representations.

To turn off actor meshes, certain types of detections, or other aspects of the displays, use the properties under **Display** on the app toolstrip.

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

Next, export the sensor detection:

- To export sensor data to the MATLAB workspace, on the app toolstrip, select **Export** > **Export Sensor Data**. Name the workspace variable and click **OK**. The app saves the sensor data as a structure containing sensor data such as the actor poses, object detections, and lane detections at each time step.
- To export a MATLAB function that generates the scenario and its sensor data, select **Export** > **Export MATLAB Function**. This function returns the sensor data as a structure, the scenario as a `drivingScenario` object, and the sensor models as `visionDetectionGenerator`, `radarDetectionGenerator`, and `lidarPointCloudGenerator` System objects. By modifying this function, you can create variations of the original scenario. For an example of this process, see “Create Driving Scenario Variations Programmatically” on page 5-107.

Save Scenario

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

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

```
drivingScenarioDesigner(scenarioFileName)
```

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

```
drivingScenarioDesigner(scenario)
```

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

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

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

See Also

Apps
Driving Scenario Designer

Blocks
Lidar Point Cloud Generator | Radar Detection Generator | Scenario Reader | Vision Detection Generator

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

More About

- “Keyboard Shortcuts and Mouse Actions for Driving Scenario Designer” on page 5-16
- “Create Reverse Motion Driving Scenarios Interactively” on page 5-69
- “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-19
- “Create Driving Scenario Variations Programmatically” on page 5-107
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-121
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-127

Keyboard Shortcuts and Mouse Actions for Driving Scenario Designer

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

Canvas Operations

These operations apply when you edit scenarios or sensors on the **Scenario Canvas** or **Sensor Canvas** panes, respectively.

Task	Action
Cut road, actor, or sensor	**Ctrl+X**
Copy road, actor, or sensor	**Ctrl+C**
Paste road, actor, or sensor	**Ctrl+V**
Delete road, actor, or sensor	**Delete**
Undo	**Ctrl+Z**
Redo	**Ctrl+Y**
Zoom in or out	Scroll wheel

Road Operations

These operations apply when you add or edit roads on the **Scenario Canvas** pane.

Task	Action
Move road one meter in any direction	Up, down, left, and right arrows
Commit a road to the canvas at the last-clicked road center	Press **Enter** or right-click in the canvas while creating the road
Commit a road to the canvas and create a road center at the current location	Double-click in the canvas while creating the road A new road center is committed at the point where you double-click.
Exit the road editing mode and remove any road centers added while editing	**Esc**
Add a road center to an existing road	Double-click the selected road at the point where you want to add the road center

Actor Operations

Actor Movement

These operations apply after you select an actor on the **Scenario Canvas** pane.

Task	Action
Move actor 1 meter in any direction	Up, down, left, and right arrows
Move actor 0.1 meter in any direction	Hold **Ctrl** and use the up, down, left, and right arrows

Actor Trajectories

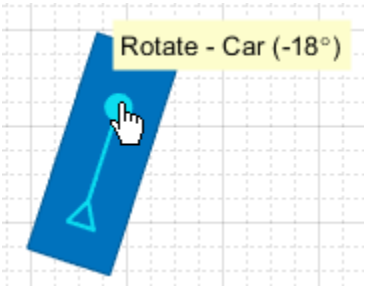
These operations apply after you select an actor on the **Scenario Canvas** pane and right-click the actor to add trajectory waypoints.

Task	Action
Commit a trajectory to the canvas at the last-clicked waypoint	Press **Enter** or right-click while creating the trajectory
Commit an actor trajectory to the canvas and create a waypoint at the current location	Double-click in the canvas while creating the trajectory A new waypoint is committed at the point where you double-click.
Exit the trajectory editing mode and remove any waypoints added while editing	**Esc**
Add a waypoint to an existing trajectory	Double-click the selected actor at the point where you want to add the waypoint
Add forward motion waypoints to a trajectory	Press **Ctrl+F** and add new waypoints
Add reverse motion waypoints to a trajectory	Press **Ctrl+R** and add new waypoints to trajectory

Actor Rotation

These operations apply to actors on the **Scenario Canvas** pane that do not already have specified trajectories. To modify existing trajectories for a selected actor, interactively move actor waypoints. Alternatively, on the **Actors** pane, edit the yaw values in the **Waypoints, Speeds, Wait Times, and Yaw** table.

To interactively rotate an actor that does not already have a trajectory, move your pointer over the actor and select the actor rotation widget.



If you do not see this widget, try zooming in.

Alternatively, click in the pane to select the actor you want to rotate and use these keyboard shortcuts.

Task	Action
Rotate actor 1 degree clockwise	Hold **Alt** and press the right arrow key
Rotate actor 1 degree counterclockwise	Hold **Alt** and press the left arrow key
Rotate actor 15 degrees clockwise	Hold **Alt+Ctrl** and press the right arrow key
Rotate actor 15 degrees counterclockwise	Hold **Alt+Ctrl** and press the left arrow key
Set actor rotation to 0 degrees	Hold **Alt** and press the up arrow key
Set actor rotation to 180 degrees	Hold **Alt** and press the down arrow key

Sensor Operations

These operations apply after you select a sensor on the **Sensor Canvas** pane.

Task	Action
Undo a sensor rotation while still rotating it.	**Esc**

File Operations

Task	Action
Open scenario file	**Ctrl+O**
Save scenario file	**Ctrl+S**

See Also

Driving Scenario Designer

Prebuilt Driving Scenarios in Driving Scenario Designer

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

Choose a Prebuilt Scenario

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

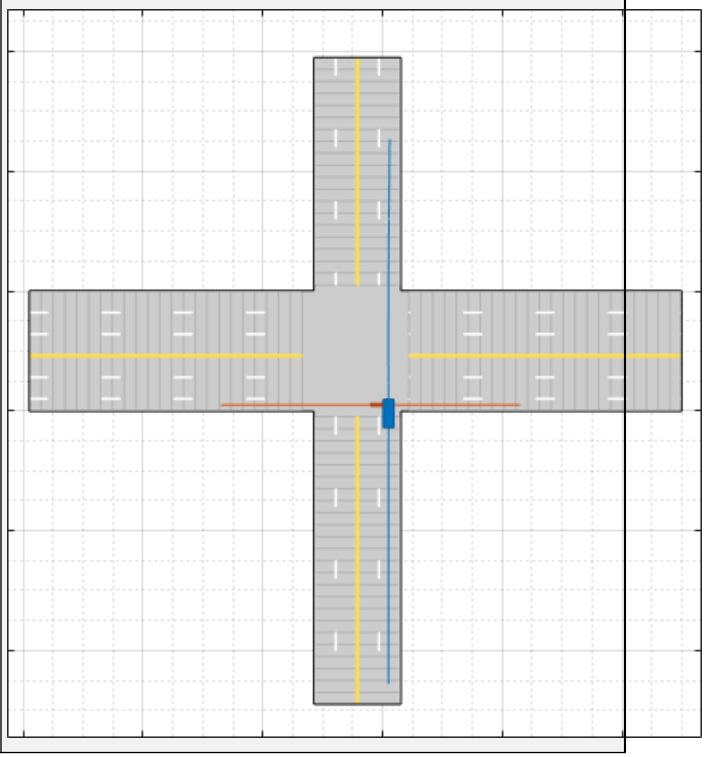
In the app, the prebuilt scenarios are stored as MAT-files and organized into folders. To open a prebuilt scenario file, from the app toolstrip, select **Open > Prebuilt Scenario**. Then select a prebuilt scenario from one of the folders.

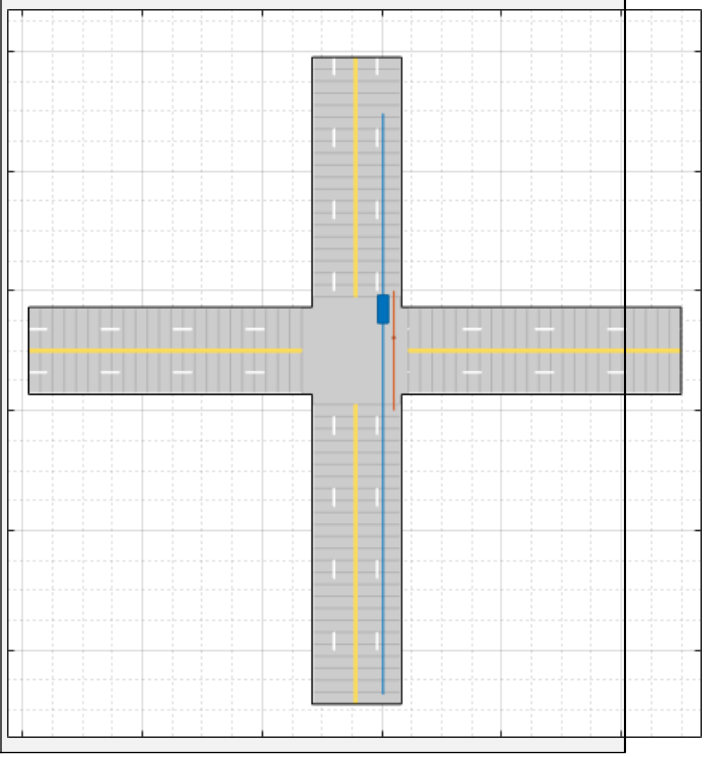
Euro NCAP

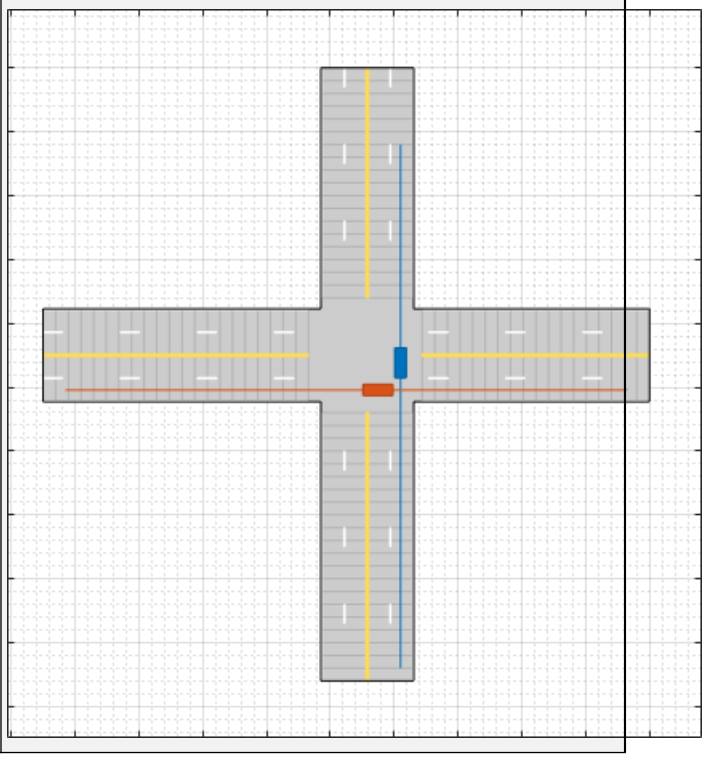
These scenarios represent Euro NCAP test protocols. The app includes scenarios for testing autonomous emergency braking, emergency lane keeping, and lane keep assist systems. For more details, see “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-41.

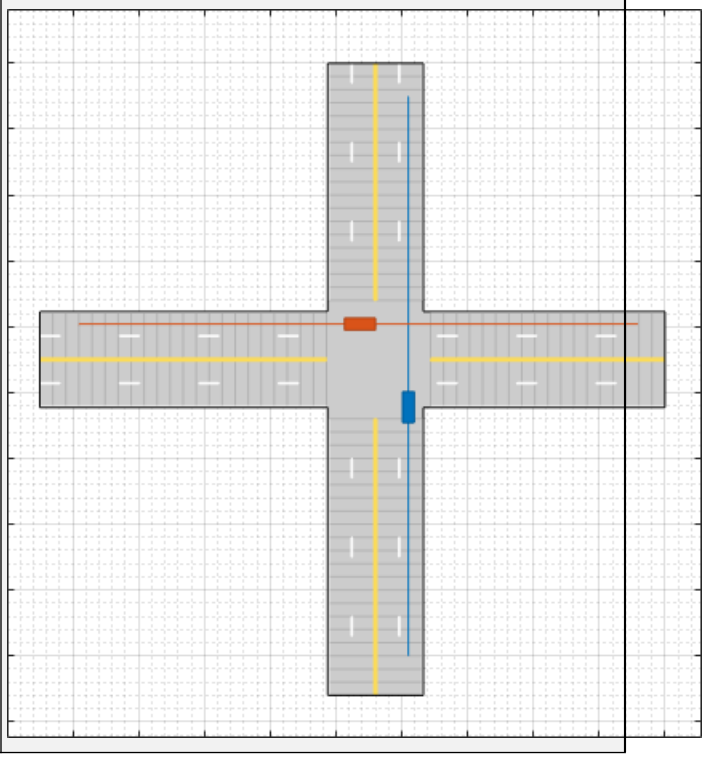
Intersections

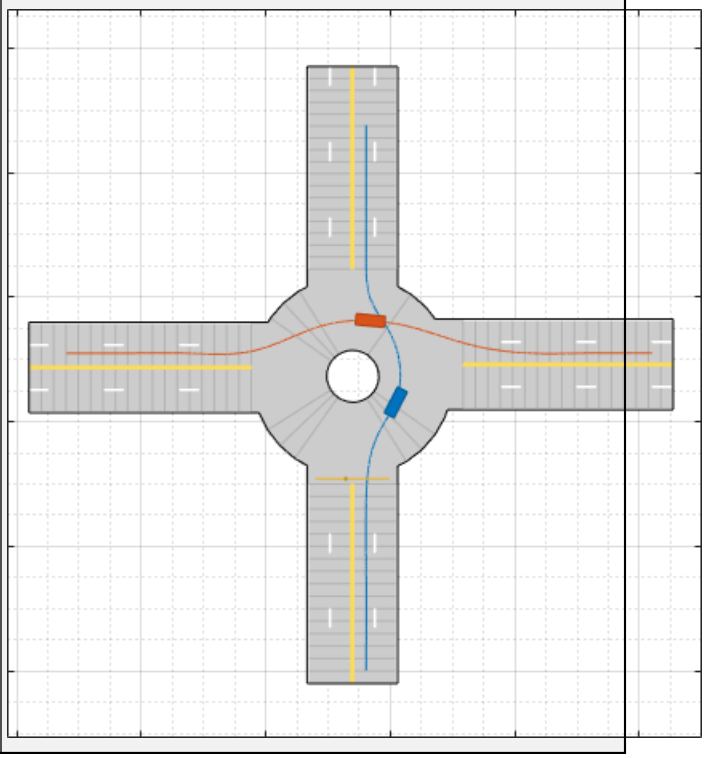
These scenarios involve common traffic patterns at four-way intersections and roundabouts.

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

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

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

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

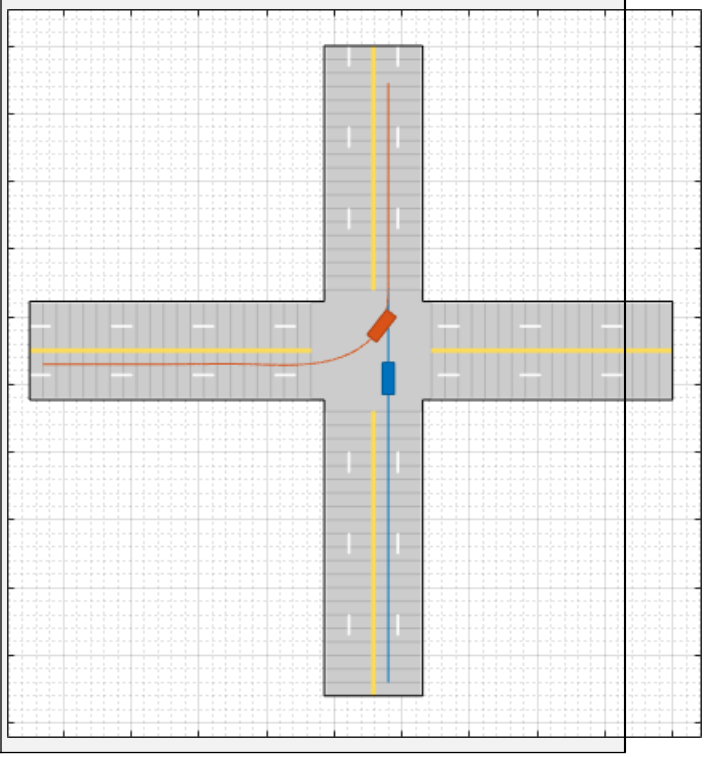
File Name	Description
`Roundabout.mat`	The ego vehicle travels north and crosses the path of a pedestrian while entering a roundabout. The ego vehicle then crosses the path of another vehicle as both vehicles drive through the roundabout. 

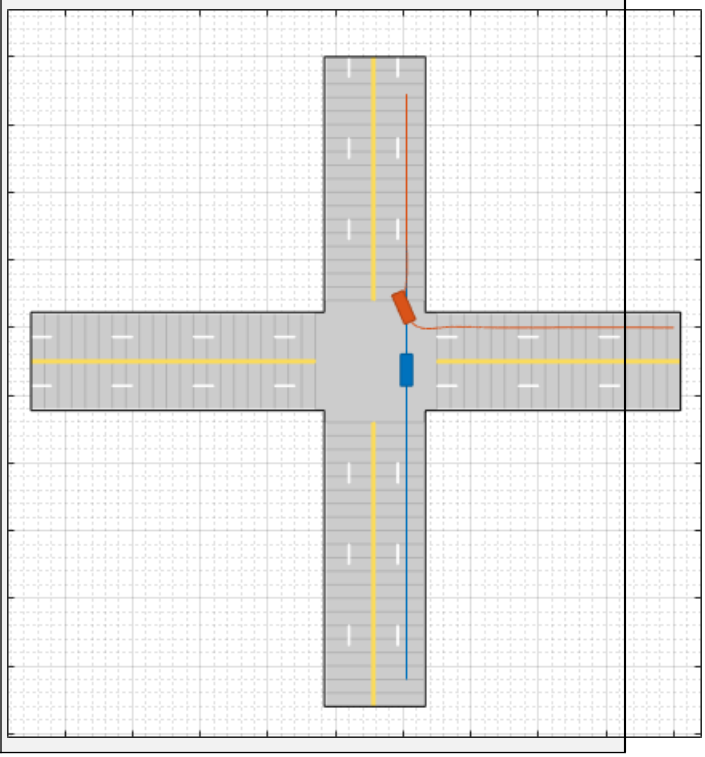
Simulation 3D

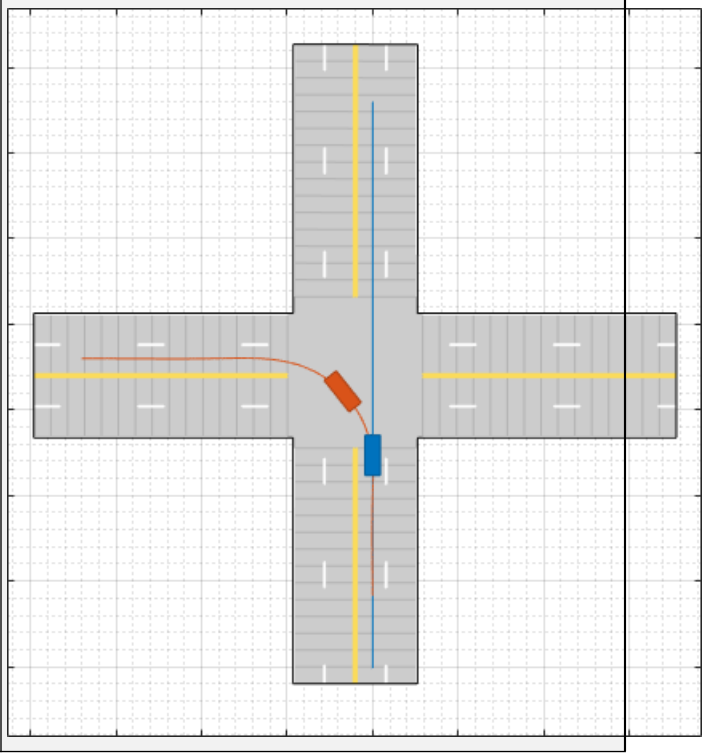
These scenarios are cuboid versions of several of the prebuilt scenes available in the 3D simulation environment. You can add vehicles and trajectories to these scenarios. Then, you can include these vehicles and trajectories in your Simulink model to simulate them in the 3D environment. This environment is rendered using the Unreal Engine from Epic Games. For more details on these scenarios, see “Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer” on page 5-62.

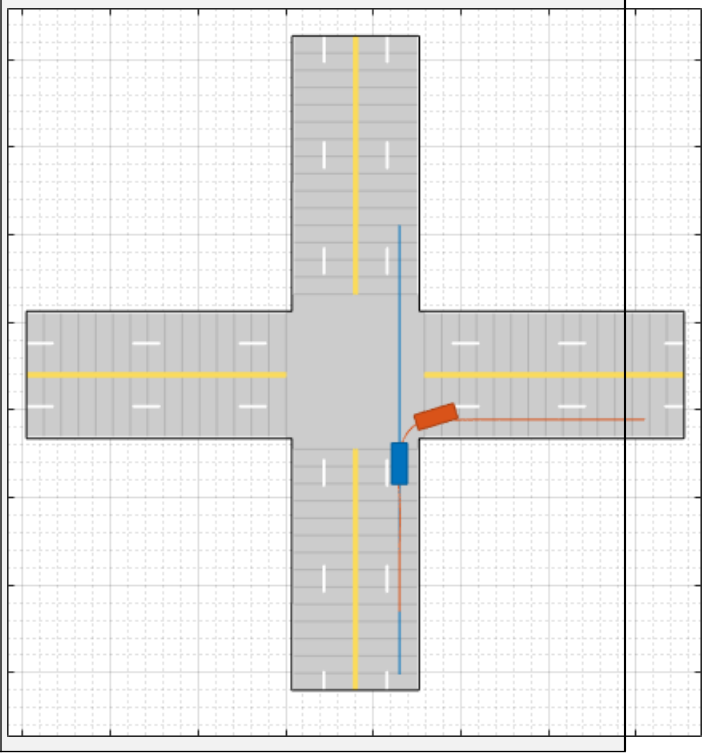
Turns

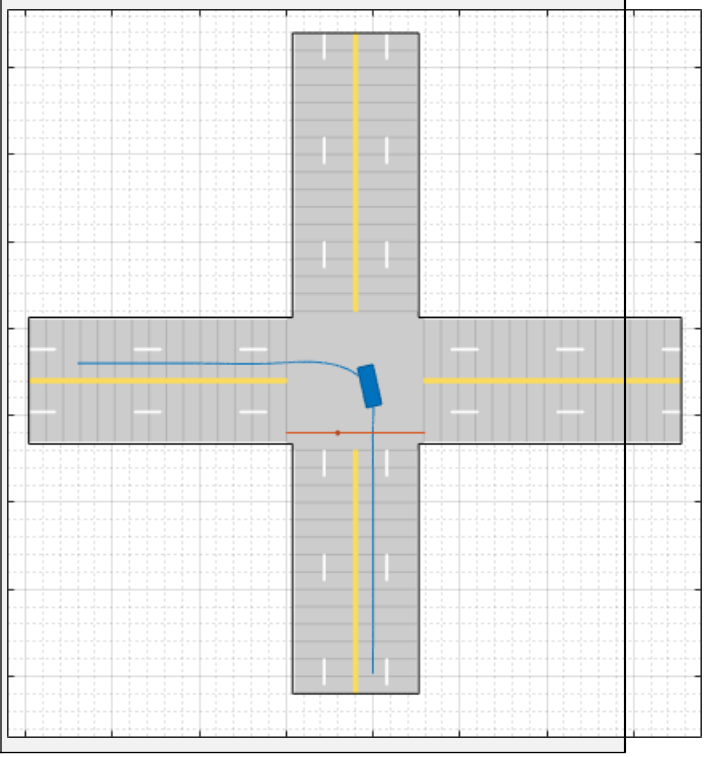
These scenarios involve turns at four-way intersections.

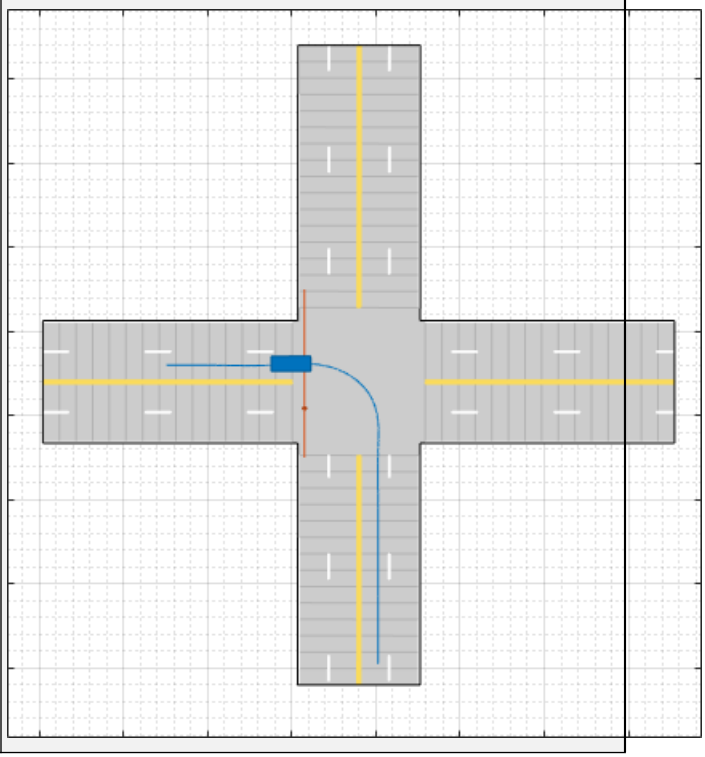
File Name	Description
`EgoVehicleGoesStraight_VehicleFromLeftTurnsLeft.mat`	The ego vehicle travels north and goes straight through an intersection. A vehicle coming from the left side of the intersection turns left and ends up in front of the ego vehicle. 

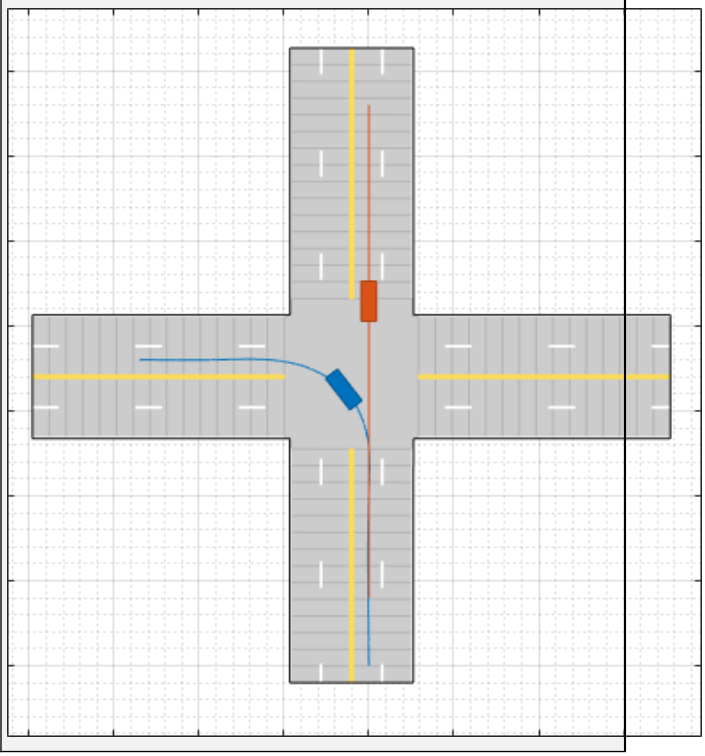
File Name	Description
`EgoVehicleGoesStraight_VehicleFromRightTurnsRight.mat`	The ego vehicle travels north and goes straight through an intersection. A vehicle coming from the right side of the intersection turns right and ends up in front of the ego vehicle. 

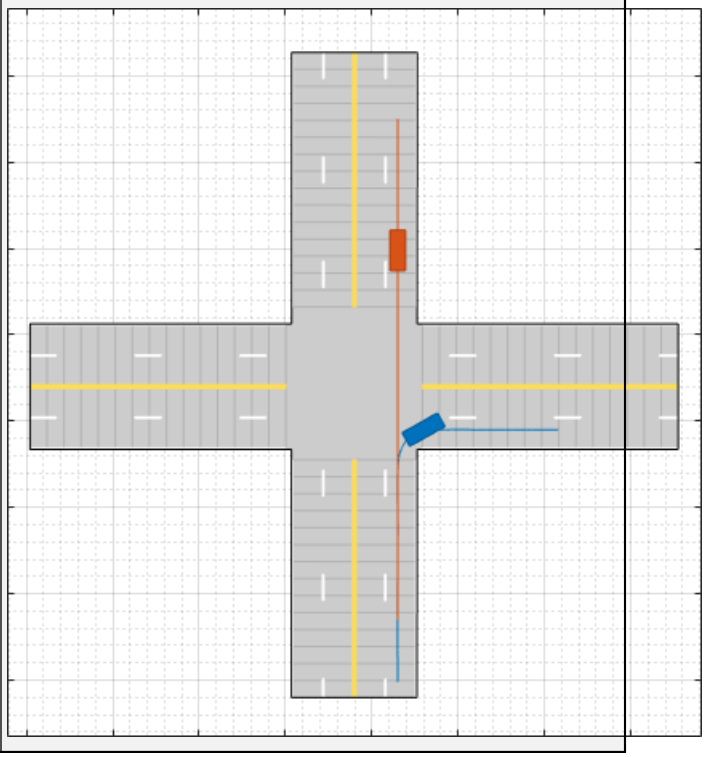
File Name	Description
`EgoVehicleGoesStraight_VehicleInFrontTurnsLeft.mat`	The ego vehicle travels north and goes straight through an intersection. A vehicle in front of the ego vehicle turns left at the intersection. 

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

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

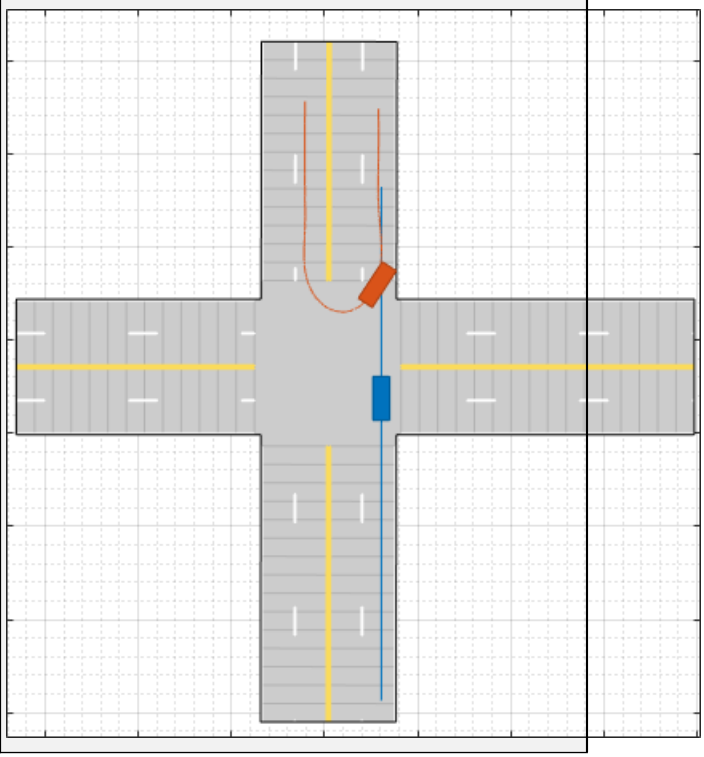
File Name	Description
`EgoVehicleTurnsLeft_PedestrianInOppLaneGoesStraight.mat`	The ego vehicle travels north and turns left at an intersection. A pedestrian in the opposite lane goes straight through the intersection. The ego vehicle completes its turn before the pedestrian finishes crossing the intersection. 

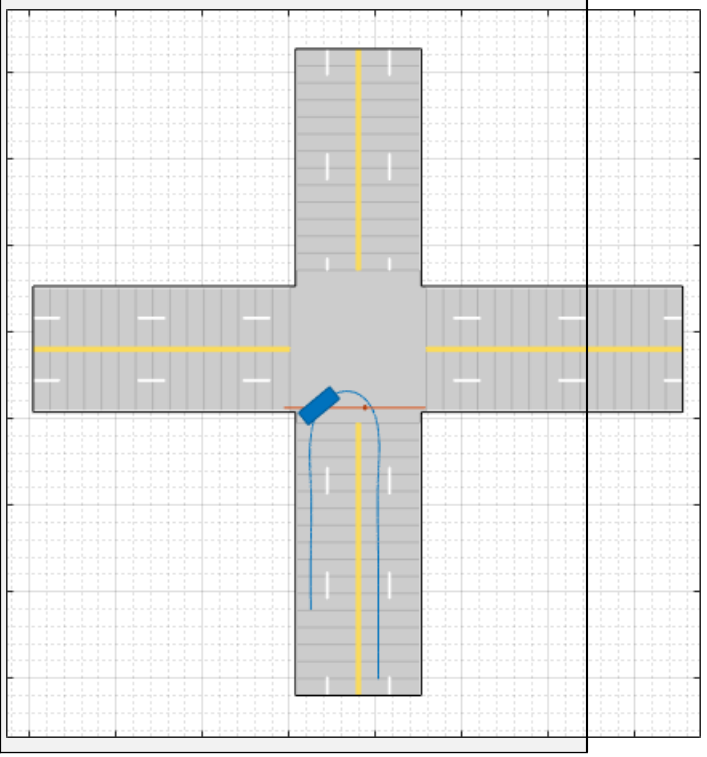
File Name	Description
`EgoVehicleTurnsLeft_VehicleInFrontGoesStraight.mat`	The ego vehicle travels north and turns left at an intersection. A vehicle in front of the ego vehicle goes straight through the intersection. 

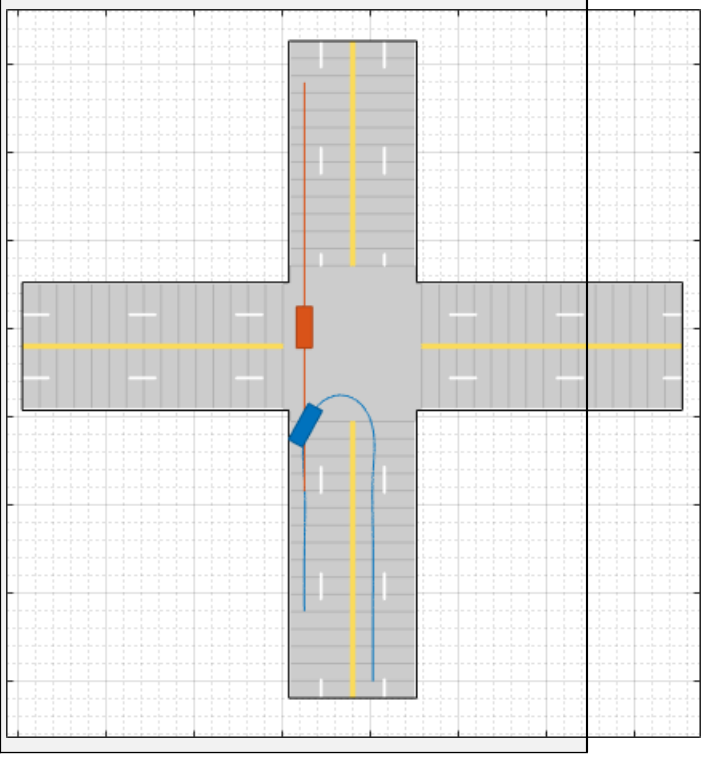
File Name	Description
`EgoVehicleTurnsRight_VehicleInFrontGoesStraight.mat`	The ego vehicle travels north and turns right at an intersection. A vehicle in front of the ego vehicle goes straight through the intersection. 

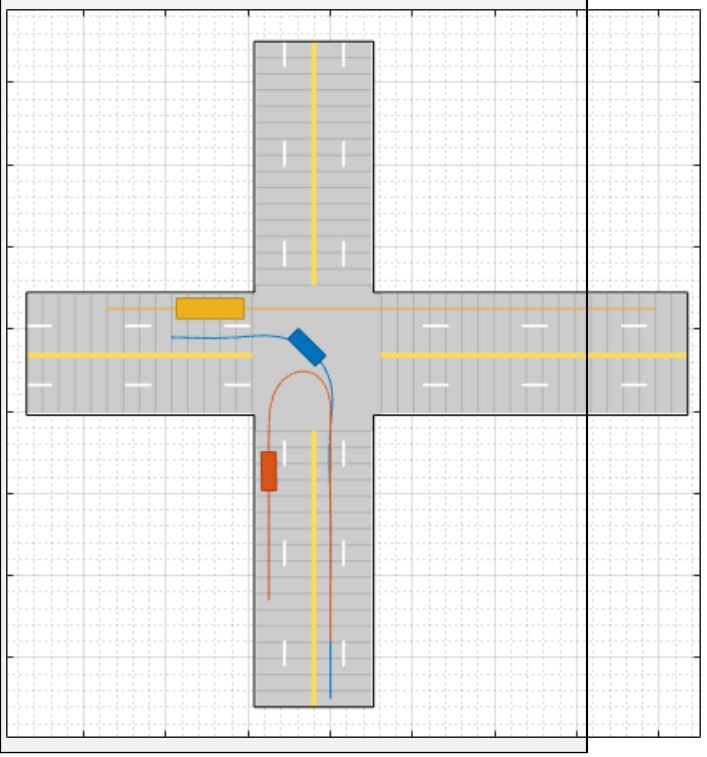
U-Turns

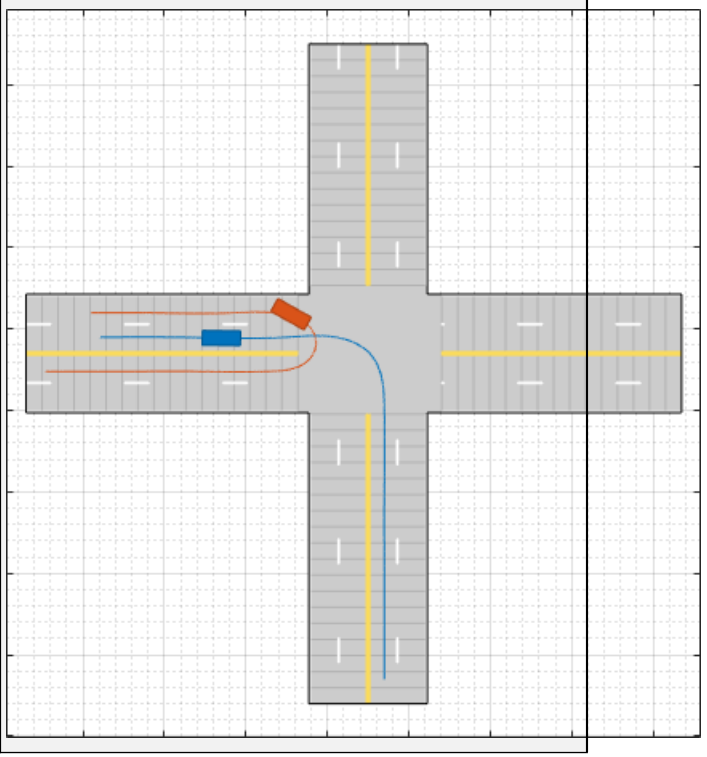
These scenarios involve U-turns at four-way intersections.

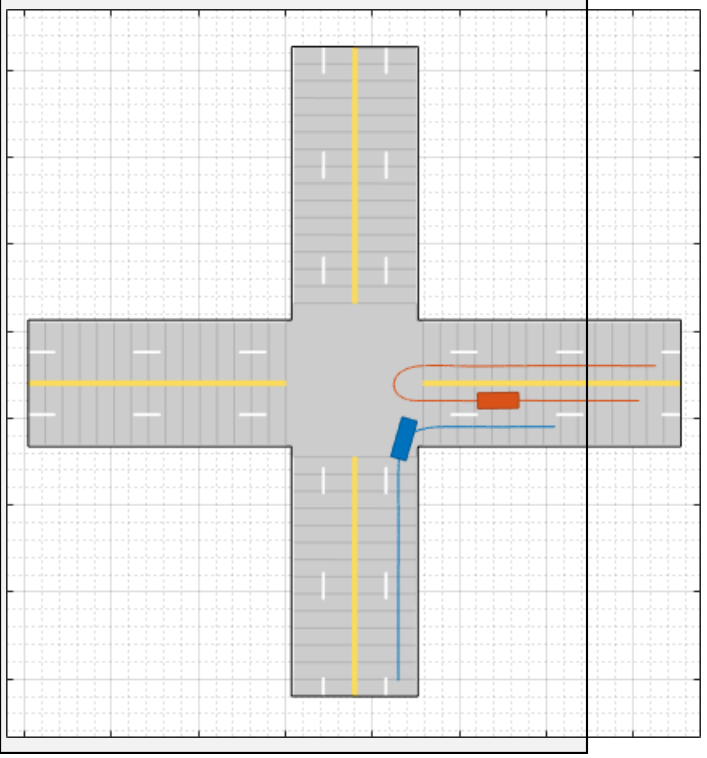
File Name	Description
`EgoVehicleGoesStraight_VehicleInOppLaneMakesUTurn.mat`	The ego vehicle travels north and goes straight through an intersection. A vehicle in the opposite lane makes a U-turn. The ego vehicle ends up behind the vehicle. 

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

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

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

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

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

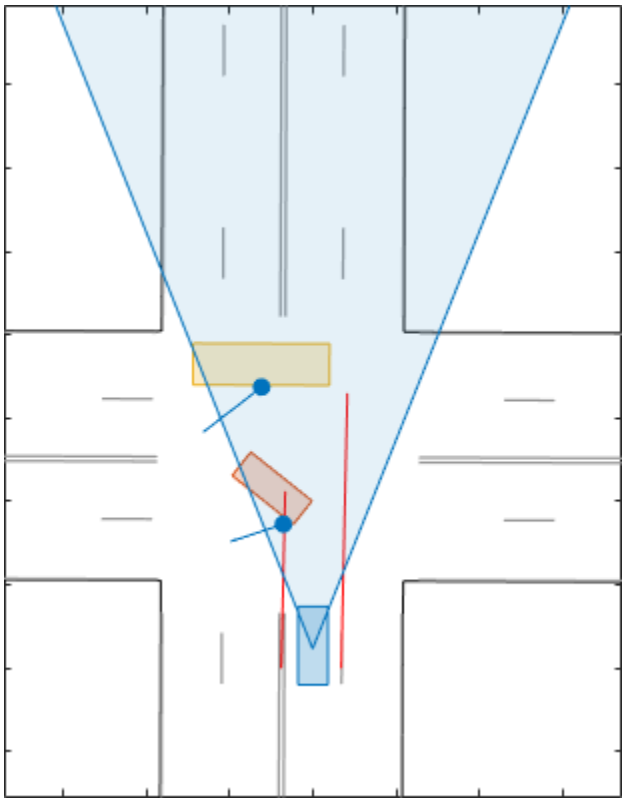
Modify Scenario

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

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

Generate Synthetic Sensor Data

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



Export the sensor data.

- To export sensor data to the MATLAB workspace, on the app toolstrip, select **Export** > **Export Sensor Data**. Name the workspace variable and click **OK**. The app saves the sensor data as a structure containing sensor data such as the actor poses, object detections, and lane detections at each time step.
- To export a MATLAB function that generates the scenario and its sensor data, select **Export** > **Export MATLAB Function**. This function returns the sensor data as a structure, the scenario as a `drivingScenario` object, and the sensor models as `visionDetectionGenerator`, `radarDetectionGenerator`, and `lidarPointCloudGenerator` System objects. By modifying this function, you can create variations of the original scenario. For an example of this process, see “Create Driving Scenario Variations Programmatically” on page 5-107.

Save Scenario

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

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

```
drivingScenarioDesigner(scenarioFileName)
```

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

```
drivingScenarioDesigner(scenario)
```

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

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

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

See Also

Apps
Driving Scenario Designer

Blocks
Lidar Point Cloud Generator | Radar Detection Generator | Vision Detection Generator

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

More About

- “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2
- “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-41
- “Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer” on page 5-62
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-121
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-127

Euro NCAP Driving Scenarios in Driving Scenario Designer

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

Choose a Euro NCAP Scenario

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

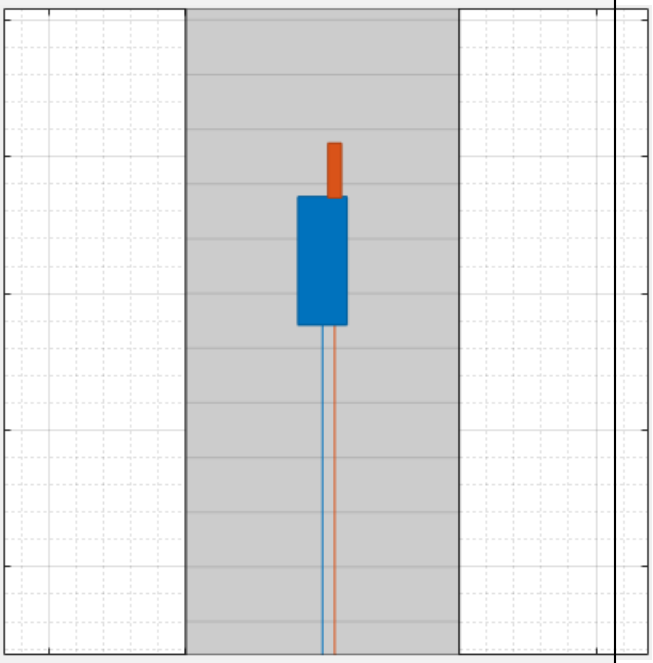
In the app, the Euro NCAP scenarios are stored as MAT-files and organized into folders. To open a Euro NCAP file, on the app toolstrip, select **Open** > **Prebuilt Scenario**. The `PrebuiltScenarios` folder opens, which includes subfolders for all prebuilt scenarios available in the app (see also “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-19).

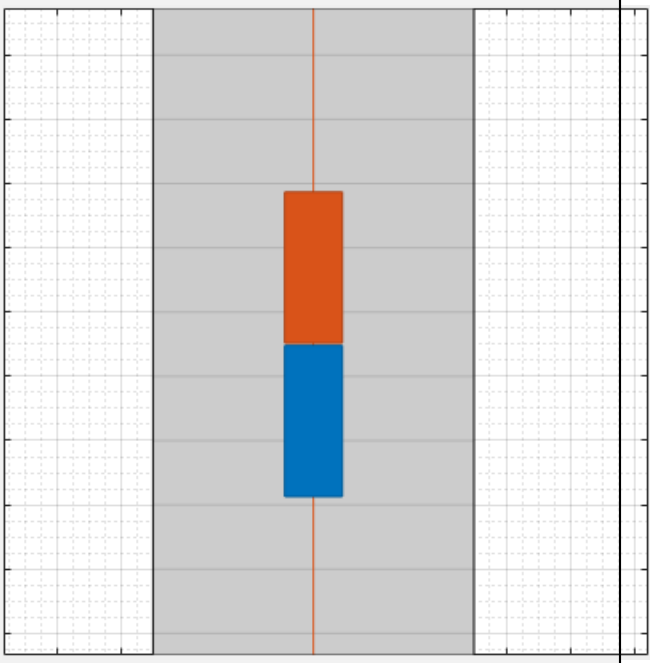
Double-click the **EuroNCAP** folder, and then choose a Euro NCAP scenario from one of these subfolders.

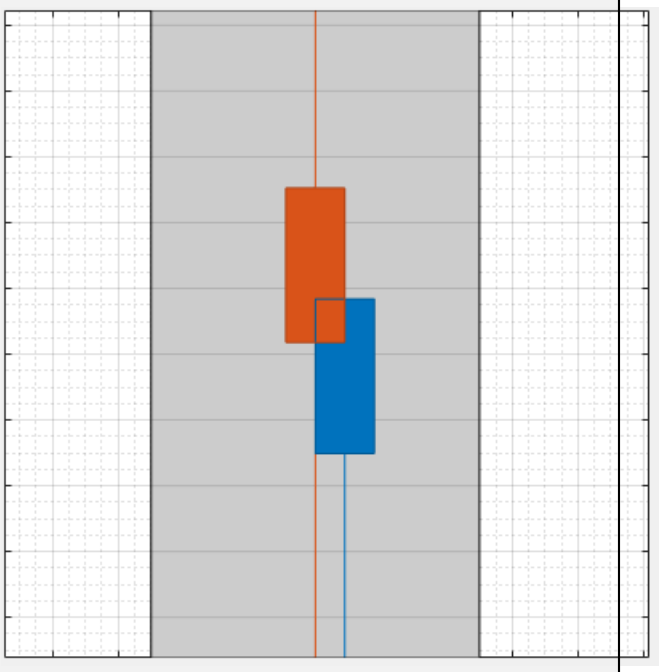
- “Autonomous Emergency Braking” on page 5-41
- “Emergency Lane Keeping” on page 5-49
- “Lane Keep Assist” on page 5-53

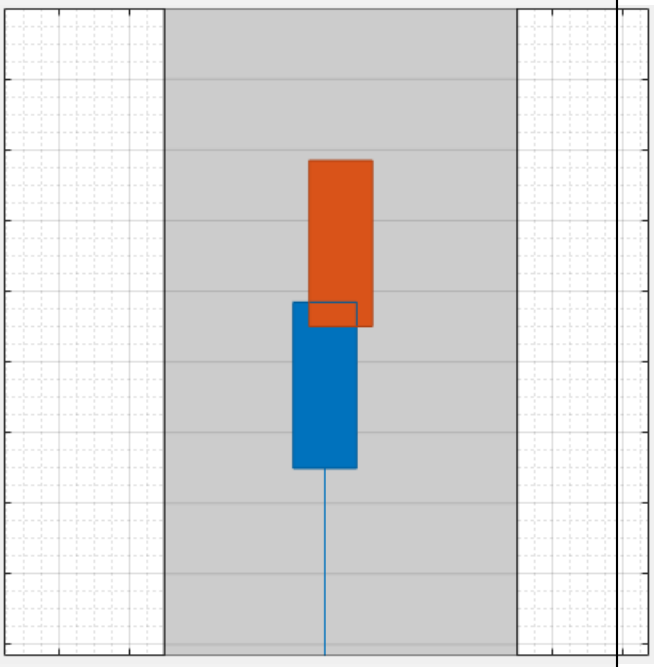
Autonomous Emergency Braking

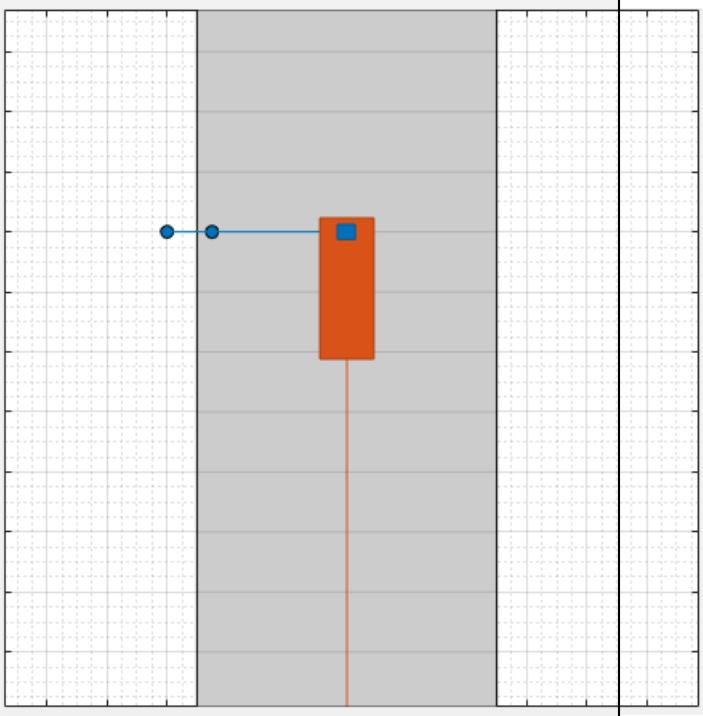
These scenarios are designed to test autonomous emergency braking (AEB) systems. AEB systems warn drivers of impending collisions and automatically apply brakes to prevent collisions or reduce the impact of collisions. Some AEB systems prepare the vehicle and restraint systems for impact. The table describes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

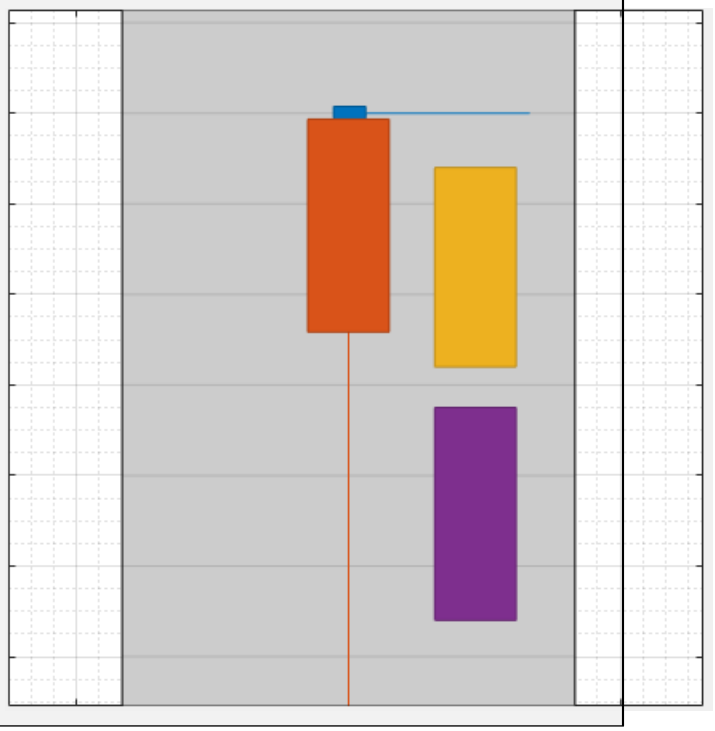
File Name	Description
AEB_Bicyclist_Longitudinal_25width.mat	The ego vehicle collides with the bicyclist that is in front of it. Before the collision, the bicyclist and ego vehicle are traveling in the same direction along the longitudinal axis. At collision time, the bicycle is 25% of the way across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

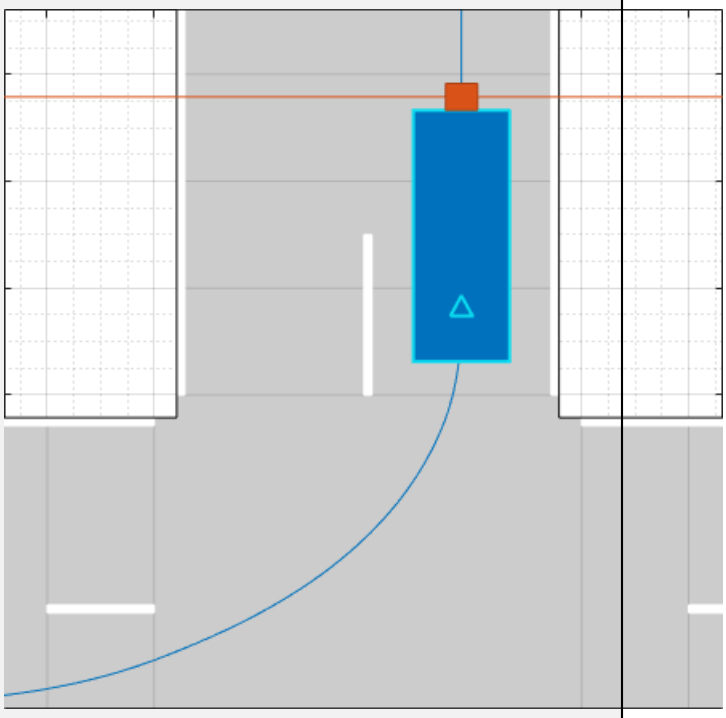
File Name	Description
`AEB_CCRb_2_initialGap_12m.mat`	A car-to-car rear braking (CCRb) scenario, where the ego vehicle rear-ends a braking vehicle. The braking vehicle begins to decelerate at 2 m/s^2. The initial gap between the ego vehicle and the braking vehicle is 12 m.  Additional scenarios vary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

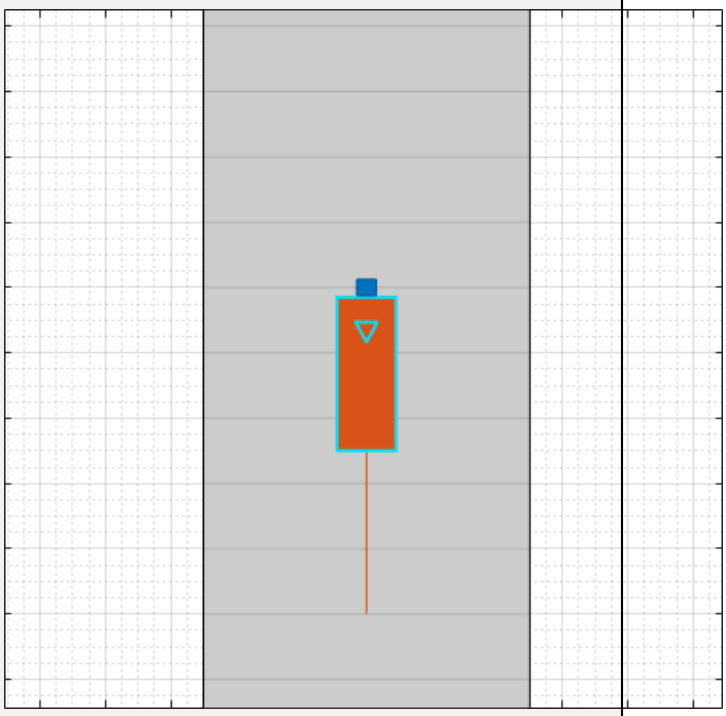
File Name	Description
AEB_CCRm_50overlap.mat	A car-to-car rear moving (CCRm) scenario, where the ego vehicle rear-ends a moving vehicle. At collision time, the ego vehicle overlaps with 50% of the width of the moving vehicle.  Additional scenarios vary the amount of overlap and the location of the overlap.

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.  Additional scenarios vary the amount of overlap and the location of the overlap.

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.  Additional scenarios vary the location of the pedestrian at collision time.

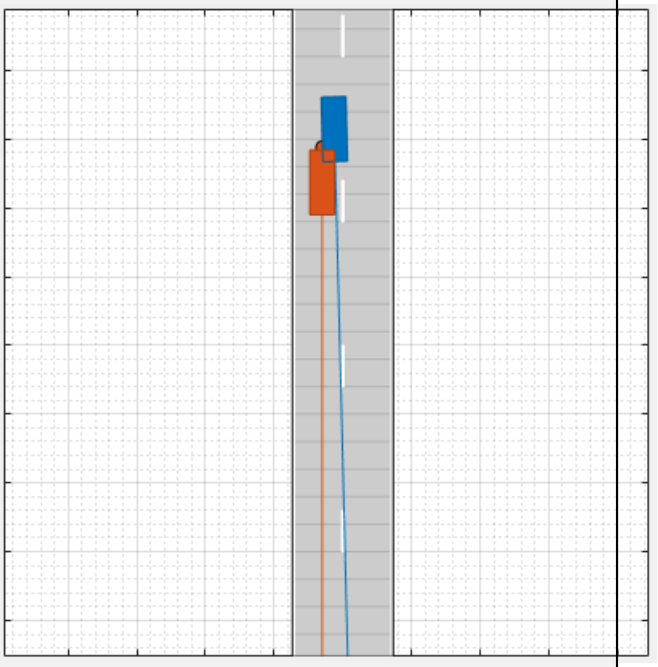
File Name	Description
`AEB_PedestrianChild_Nearside_50width.mat`	The ego vehicle collides with a pedestrian who is traveling from the right side of the road, which Euro NCAP test protocols refer to as the near side. These protocols assume that vehicles travel on the right side of the road. Therefore,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

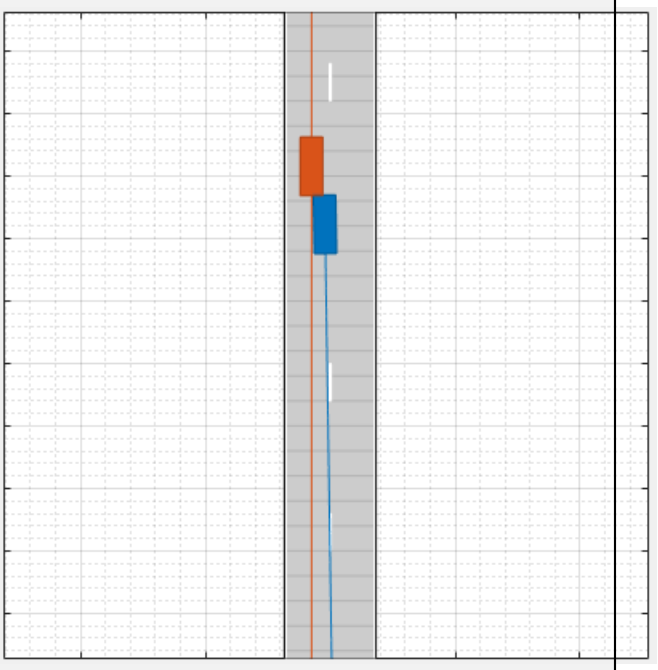
File Name	Description
`AEB_PedestrianTurning_Farside_50width.mat`	The ego vehicle turns at an intersection and collides with a pedestrian who is traveling parallel with the left side, or far side, of the vehicle at the start of the simulation. At collision time, the pedestrian is 50% of the way across the width of the ego vehicle.  In an additional scenario, the pedestrian is on the other side of the intersection and travels parallel with the right side, or near side, of the vehicle at the start of the simulation.

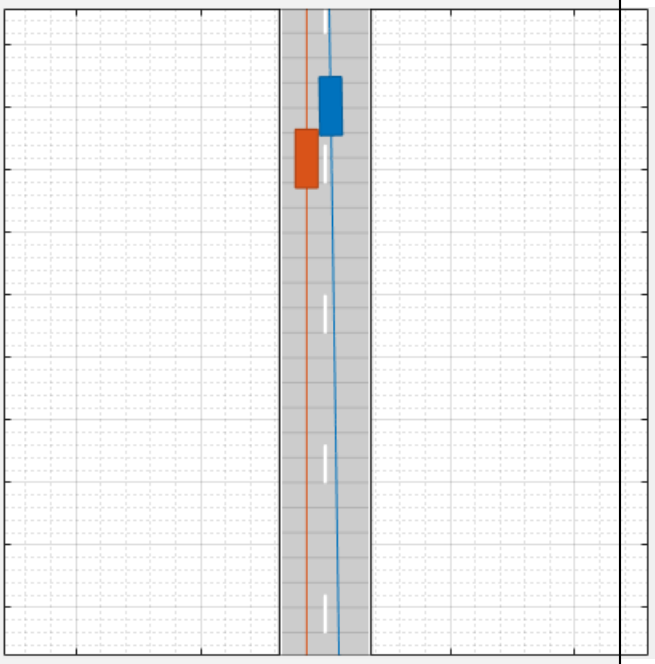
File Name	Description
`Reverse_AEB_Pedestrian_Stationary_50width.mat`	The ego vehicle travels in reverse and collides with a stationary pedestrian. At collision time, the pedestrian is 50% of the way across the width of the ego vehicle.  In an additional scenario, before the collision, the pedestrian travels from the right side, or near side, of the forward frame of reference of the vehicle.

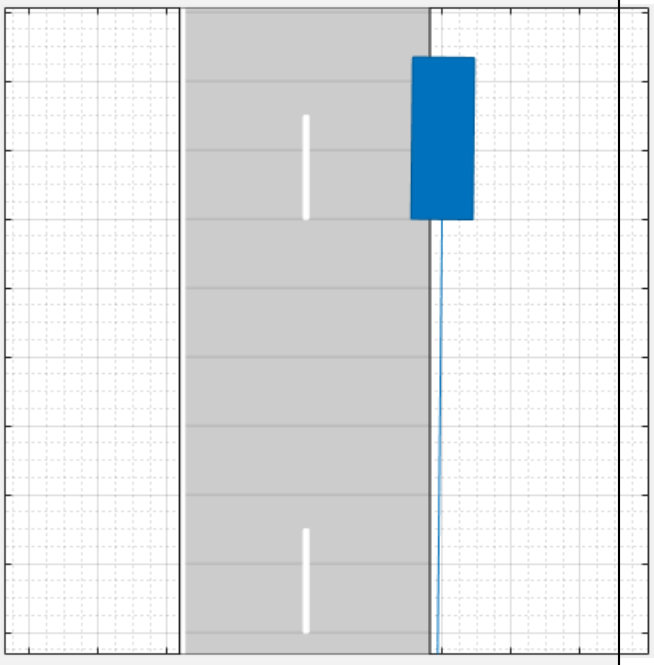
Emergency Lane Keeping

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 describes a subset of the ELK scenarios.

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.  Additional scenarios vary the lateral velocity and whether the lane change was intentional or unintentional.

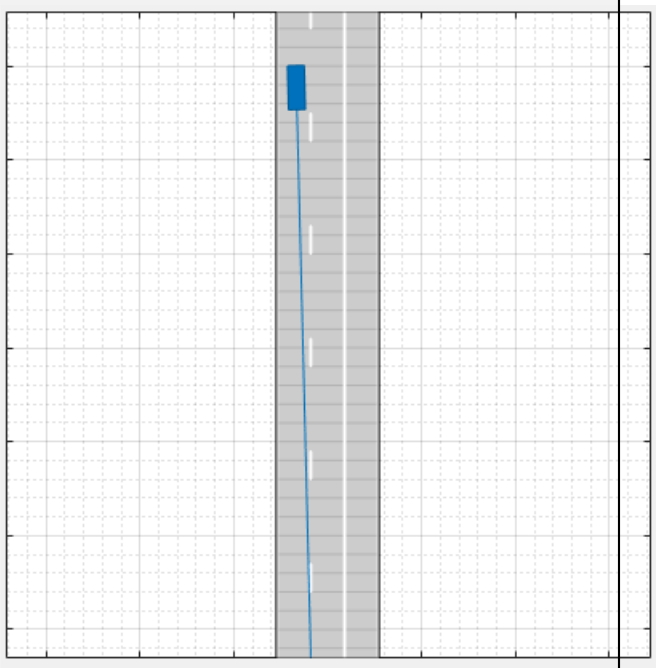
File Name	Description
`ELK_OncomingVeh_Vlat_0.3.mat`	The ego vehicle unintentionally changes lanes and collides with an oncoming vehicle that is in the other lane. The ego vehicle travels at a lateral velocity of 0.3 m/s.  Additional scenarios vary the lateral velocity.

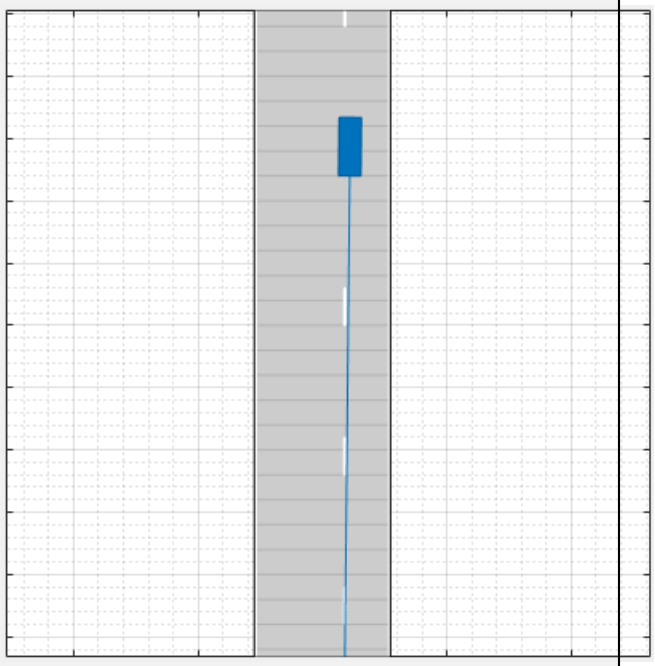
File Name	Description
`ELK_OvertakingVeh_Unintent_Vlat_0.3.mat`	The ego vehicle unintentionally changes lanes, overtakes a vehicle in the other lane, and collides with that vehicle. The ego vehicle travels at a lateral velocity of 0.3 m/s.  Additional scenarios vary the lateral velocity and whether the lane change was intentional or unintentional.

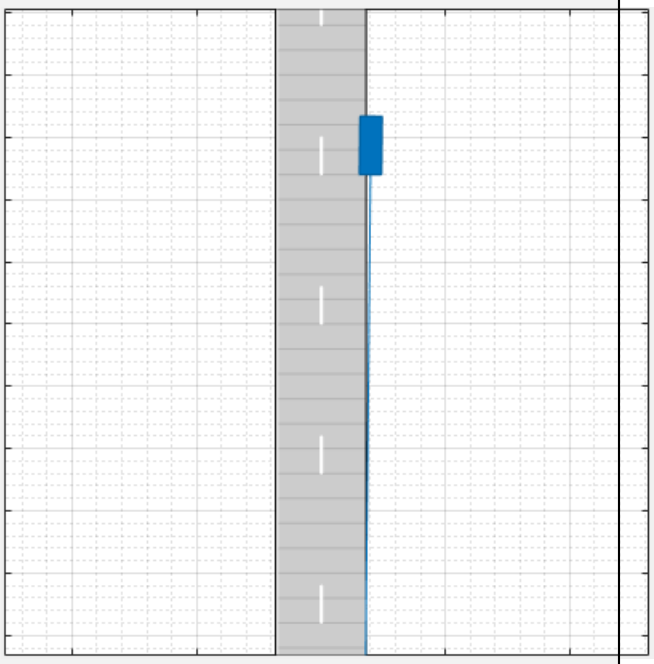
File Name	Description
`ELK_RoadEdge_NoBndry_Vlat_0.2.mat`	The ego vehicle unintentionally changes lanes and ends up on the road edge. The road edge has no lane boundary markings. The ego vehicle travels at a lateral velocity of 0.2 m/s.  Additional scenarios vary the lateral velocity and whether the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

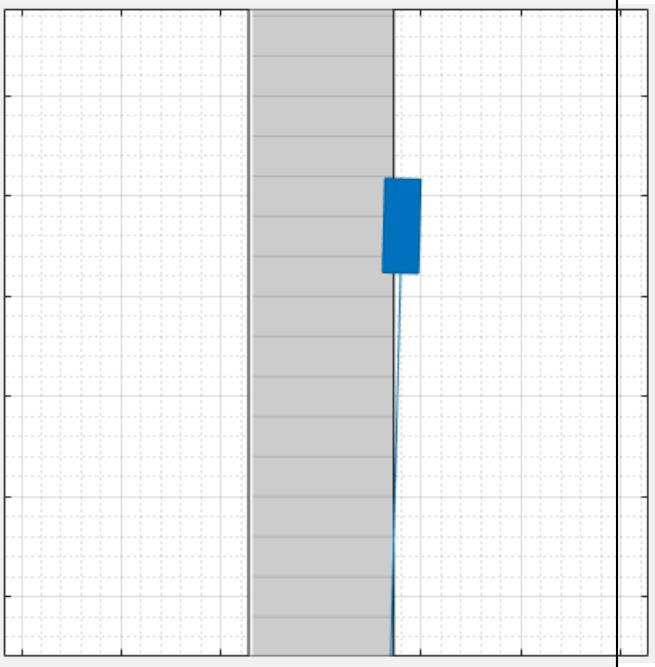
Lane Keep Assist

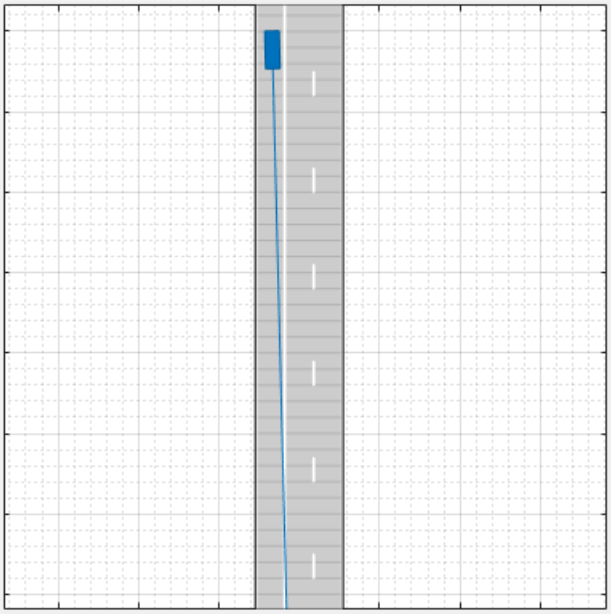
These scenarios are designed to test lane keep assist (LKA) systems. LKA systems detect unintentional lane departures and automatically adjust the steering angle of the vehicle to stay within the lane boundaries. The table lists a subset of the LKA scenarios.

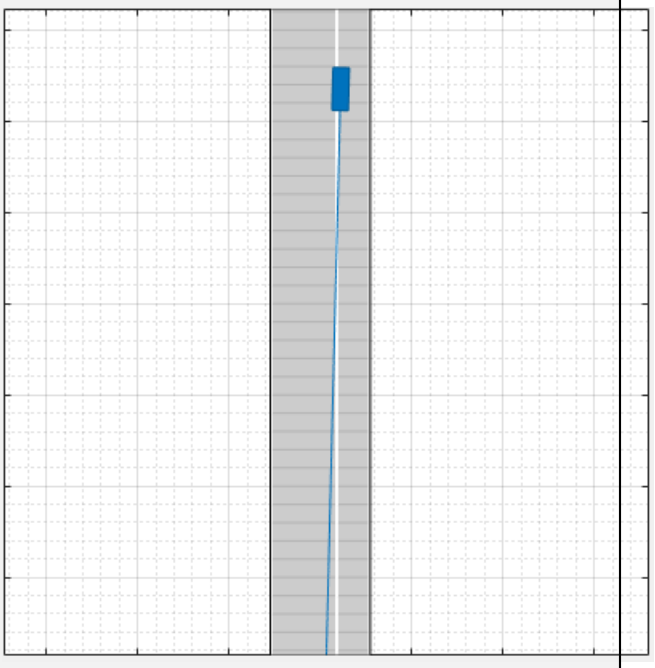
File Names	Description
`LKA_DashedLine_Solid_Left_Vlat_0.5.mat`	The ego vehicle unintentionally departs from a lane that is dashed on the left and solid on the right. The car departs the lane from the left (dashed) side, traveling at a lateral velocity of 0.5 m/s.  Additional scenarios vary the lateral velocity and whether the dashed lane that the vehicle crosses over is on the left or right.

File Names	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.  Additional scenarios vary the lateral velocity and whether the dashed lane marking that the vehicle crosses over is on the left or right.

File Names	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.  Additional scenarios vary the lateral velocity.

File Names	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.  Additional scenarios vary the lateral velocity.

File Names	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.  Additional scenarios vary the lateral velocity and whether the solid lane marking that the vehicle crosses over is on the left or right.

File Names	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.  Additional scenarios vary the lateral velocity and whether the solid lane marking that the vehicle crosses over is on the left or right.

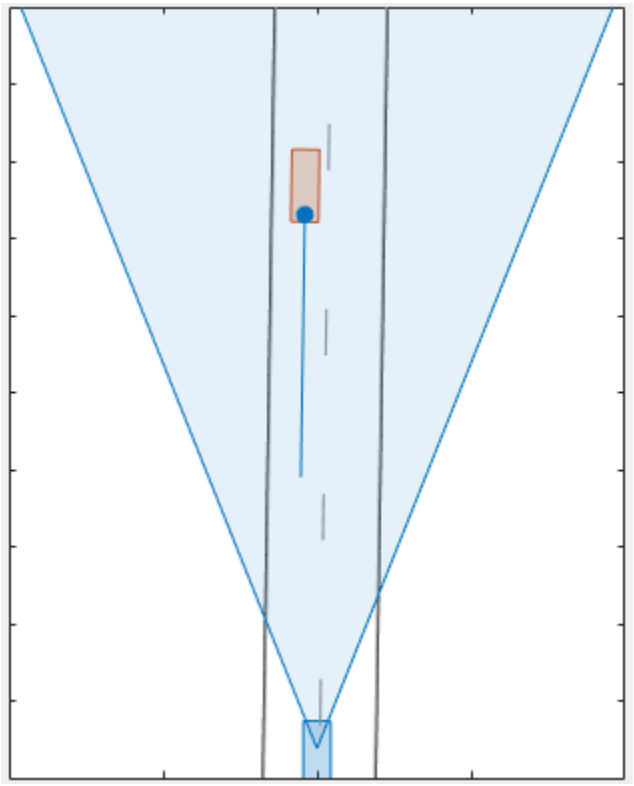
Modify Scenario

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

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

Generate Synthetic Detections

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



Export the detections.

- To export sensor data to the MATLAB workspace, on the app toolstrip, select **Export** > **Export Sensor Data**. Name the workspace variable and click **OK**. The app saves the sensor data as a structure containing sensor data such as the actor poses, object detections, and lane detections at each time step.
- To export a MATLAB function that generates the scenario and its sensor data, select **Export** > **Export MATLAB Function**. This function returns the sensor data as a structure, the scenario as a `drivingScenario` object, and the sensor models as `visionDetectionGenerator`, `radarDetectionGenerator`, and `lidarPointCloudGenerator` System objects. By modifying this function, you can create variations of the original scenario. For an example of this process, see “Create Driving Scenario Variations Programmatically” on page 5-107.

Save Scenario

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

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

```
drivingScenarioDesigner(scenarioFileName)
```

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

```
drivingScenarioDesigner(scenario)
```

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

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

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

References

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

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

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

See Also

Apps
Driving Scenario Designer

Blocks
Lidar Point Cloud Generator | Radar Detection Generator | Scenario Reader | Vision Detection Generator

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

More About

- “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2
- “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-19
- “Create Driving Scenario Variations Programmatically” on page 5-107
- “Autonomous Emergency Braking with Sensor Fusion” on page 7-214
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-121
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-127

External Websites

- Euro NCAP Safety Assist Protocols

Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer

The **Driving Scenario Designer** app provides prebuilt scenarios that recreate scenes from the 3D simulation environment within the cuboid simulation environment. In these cuboid versions of the scenes, you can add vehicles represented using simple box shapes and specify their trajectories. Then, you can simulate these vehicles and trajectories in your Simulink model by using the higher fidelity 3D simulation versions of the scenes. The 3D environment renders these scenes using the Unreal Engine from Epic Games. For more details about the environment, see “Unreal Engine Simulation for Automated Driving” on page 6-2.

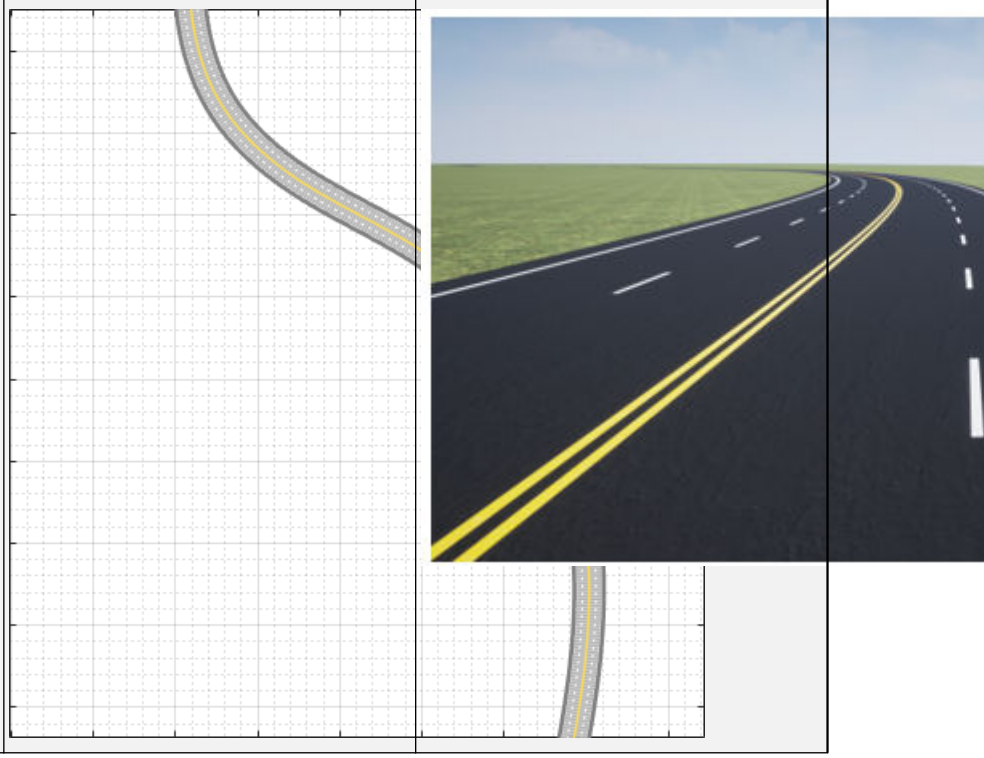
Choose 3D Simulation Scenario

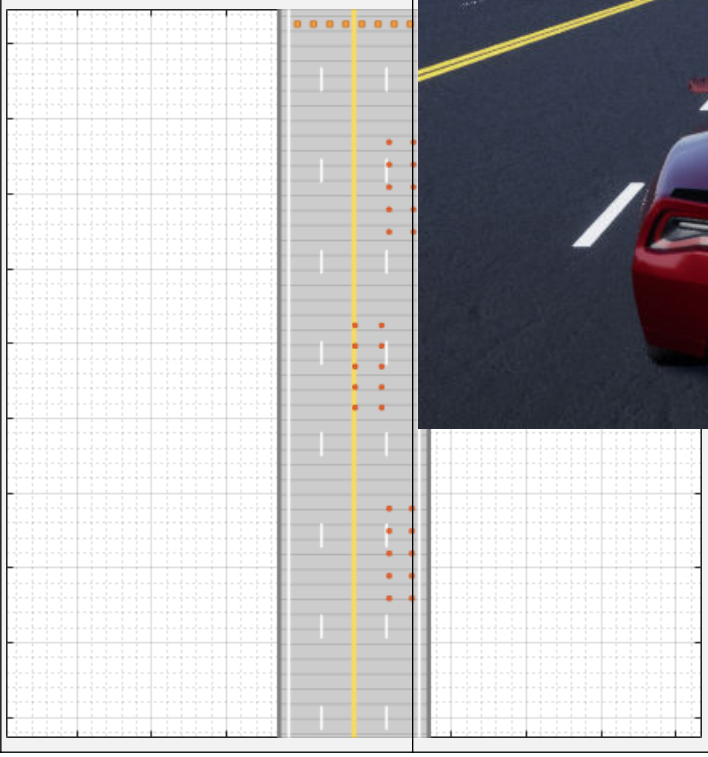
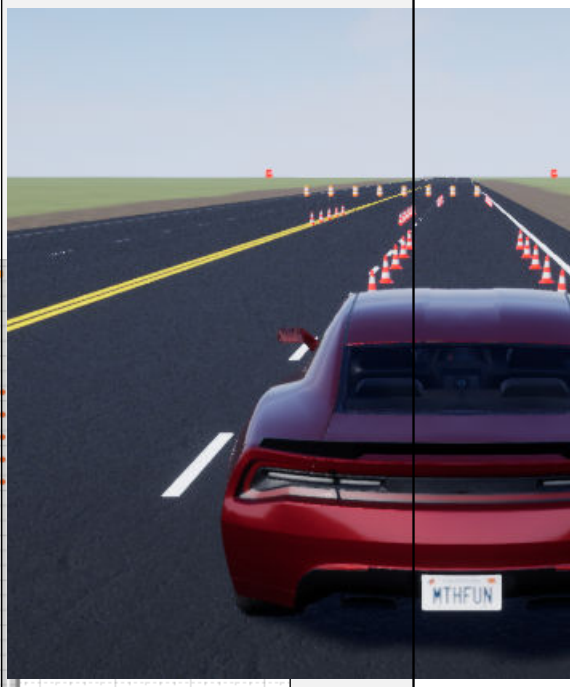
Open the **Driving Scenario Designer** app. At the MATLAB command prompt, enter `drivingScenarioDesigner`.

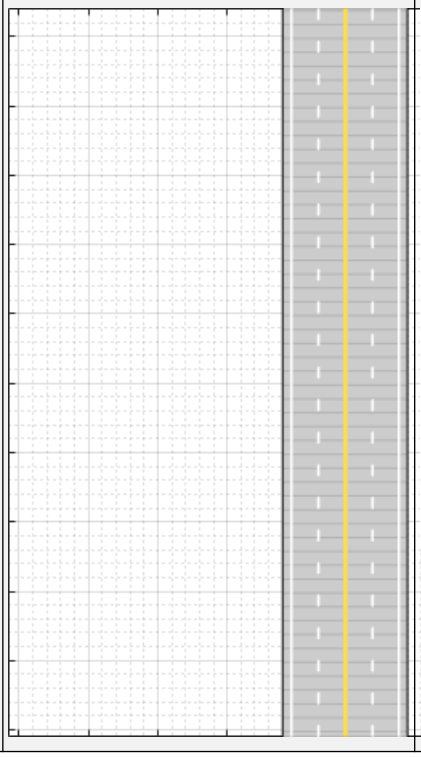
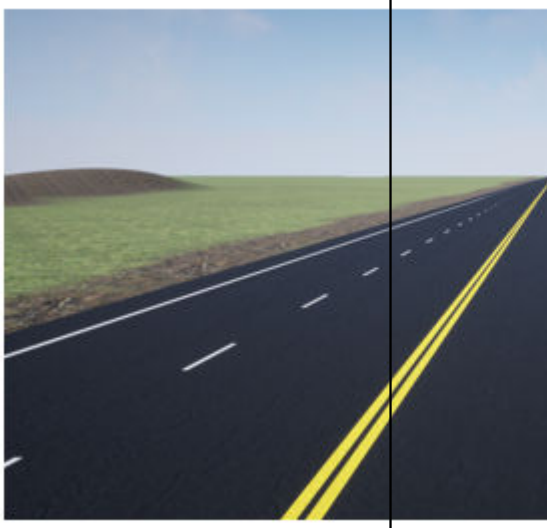
The app stores the 3D simulation scenarios as MAT-files called scenario files. To open a scenario file, first select **Open** > **Prebuilt Scenario** on the app toolstrip. The **PrebuiltScenarios** folder that opens includes subfolders for all prebuilt scenarios available in the app.

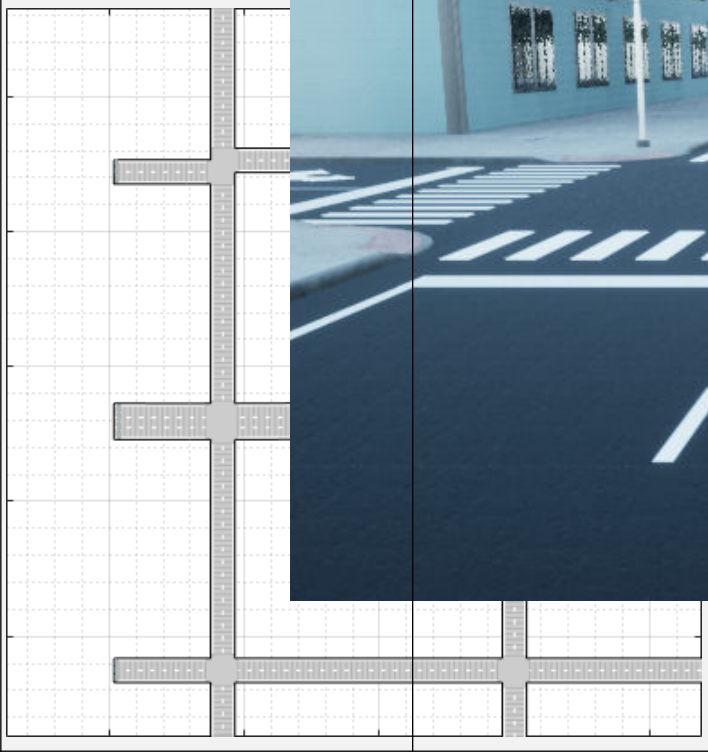
Double-click the **Simulation3D** folder, and then choose one of the scenarios described in this table.

File Name of Cuboid Scenario	Description	Corresponding 3D Scene
`CurvedRoad.mat`	Curved, looped road	**Curved Road**

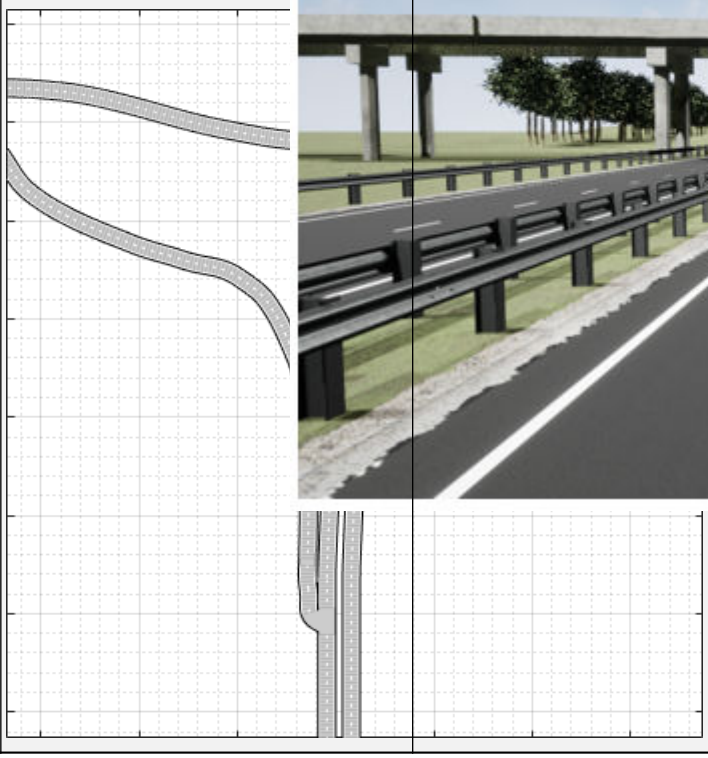


File Name of Cuboid Scenario	Description	Corresponding 3D Scene
`DoubleLaneChange.mat`	Straight road with traffic cones and traffic barrels that are set up for executing a double lane change The cuboid version does not include the traffic signs or traffic light that are in the corresponding 3D scene. 	**Double Lane Change** 

File Name of Cuboid Scenario	Description	Corresponding 3D Scene
`StraightRoad.mat`	Straight road segment 	**Straight Road** 

File Name of Cuboid Scenario	Description	Corresponding 3D Scene
`USCityBlock.mat`	City block with intersections and barriers The cuboid version doe include the traffic light in the corresponding 3I It also does not include crosswalk or pedestrian markings at intersectio objects inside the city b such as buildings, trees hydrants. 	**US City Block** 

File Name of Cuboid Scenario	Description	Corresponding 3D Scene
USHighway.mat	Highway with traffic cones and barriers The cuboid version doe include the traffic signs guard rails that are in t corresponding 3D scen	**US Highway**



Note The **Driving Scenario Designer** app does not include cuboid versions of these scenes:

- **Large Parking Lot**
- **Open Surface**
- **Parking Lot**
- **Virtual Mcity**

To generate vehicle trajectories for these unsupported scenes or for custom scenes, use the process described in the “Select Waypoints for Unreal Engine Simulation” on page 7-626 example.

Modify Scenario

With the scenario loaded, you can now add vehicles to the scenario, set their trajectories, and designate one of the vehicles as the ego vehicle. For an example that shows how to do complete these actions, see “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2.

If you plan to simulate these vehicles in the corresponding 3D scene, avoid modifying the road network or existing road objects, such as barriers and traffic cones. Otherwise, you can break parity between the two versions and simulation results might differ. To prevent such accidental changes to the road network, road interactions are disabled by default. If you want to modify the road network, in the bottom-left corner of the **Scenario Canvas** pane, click the Configure the Scenario Canvas button  . Then, select **Enable road interactions**.

You can add sensors to the ego vehicle, but you cannot recreate these sensors in the 3D environment. The environment has its own sensors in the form of Simulink blocks. For more details, see “Choose a Sensor for Unreal Engine Simulation” on page 6-16.

Save Scenario

Because these scenarios are read-only, to save your scenario file, you must save a copy of it to a new folder. On the app toolstrip, select **Save > Scenario File As**.

You can reopen the saved scenario file from the app. Alternatively, at the MATLAB command prompt, enter this command, where *scenarioFileName* represents the scenario file to open.

```
drivingScenarioDesigner(scenarioFileName)
```

Recreate Scenario in Simulink for 3D Environment

After you save the scenario file containing the vehicles and other actors that you added, you can recreate these vehicles in trajectories in Simulink. At a high level, follow this workflow:

1. Configure 3D scene — In a new model, add a Simulation 3D Scene Configuration block and specify the 3D scene that corresponds to your scenario file.
2. Read actor poses from scenario file — Add a Scenario Reader block and read the actor poses at each time step from your scenario file. These poses comprise the trajectories of the actors.
3. Transform actor poses — Output the actors, including the ego vehicle, from the Scenario Reader block. Use Vehicle To World and Cuboid To 3D Simulation blocks to convert the actor poses to the coordinate system used in the 3D environment.
4. Specify actor poses to vehicles — Add Simulation 3D Vehicle with Ground Following blocks that correspond to the vehicles in your model. Specify the converted actor poses as inputs to the vehicle blocks.
5. Add sensors and simulate — Add sensors, simulate in the 3D environment, and view sensor data using the **Bird's-Eye Scope**.

For an example that follows this workflow, see “Visualize Sensor Data from Unreal Engine Simulation Environment” on page 6-36.

See Also

Apps
Driving Scenario Designer

Blocks
Simulation 3D Scene Configuration | Cuboid To 3D Simulation | Scenario Reader | Vehicle To World

More About

- “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-19
- “Visualize Sensor Data from Unreal Engine Simulation Environment” on page 6-36

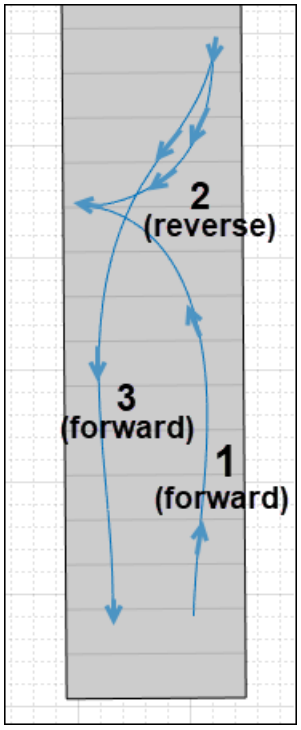
Create Reverse Motion Driving Scenarios Interactively

This example shows how to create a driving scenario in which a vehicle drives in reverse by using the **Driving Scenario Designer** app. In this example, you specify a vehicle that completes a three-point turn.

Three-Point Turn Scenario

A three-point turn is a basic driving maneuver for changing directions on a road. The three segments of a three-point turn consist of these motions:

1. Drive forward and turn toward the opposite side of the road.
2. Drive in reverse while turning back toward the original side of the road.
3. Drive forward toward the opposite side of the road to complete the change in direction.



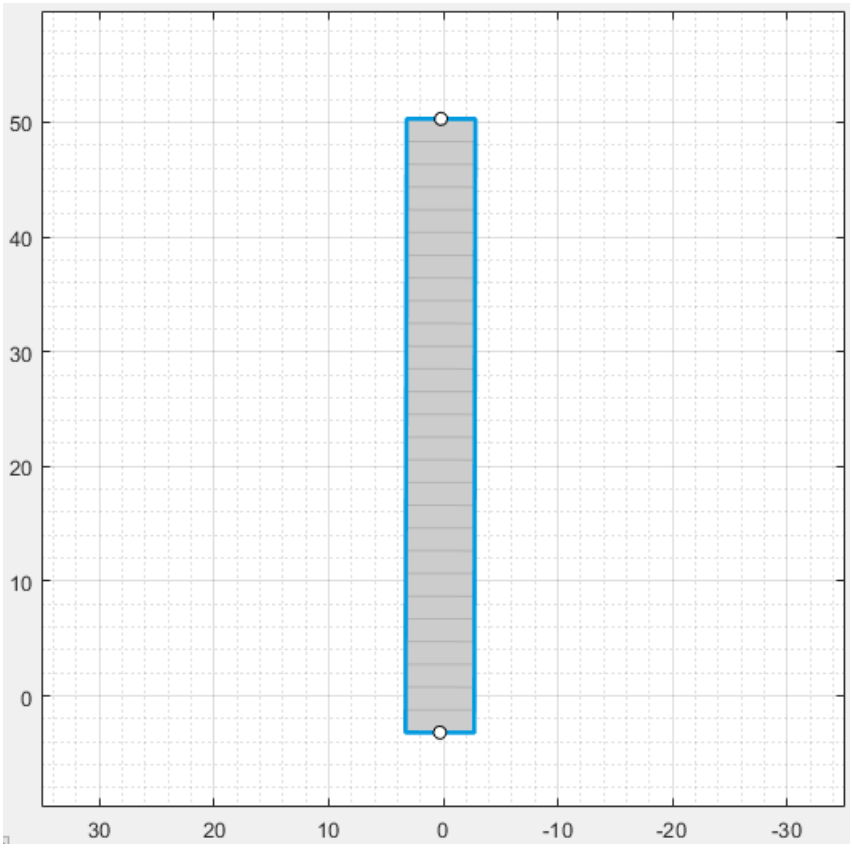
You can use reverse motions to design more complex scenarios for testing automated driving algorithms.

Add Road

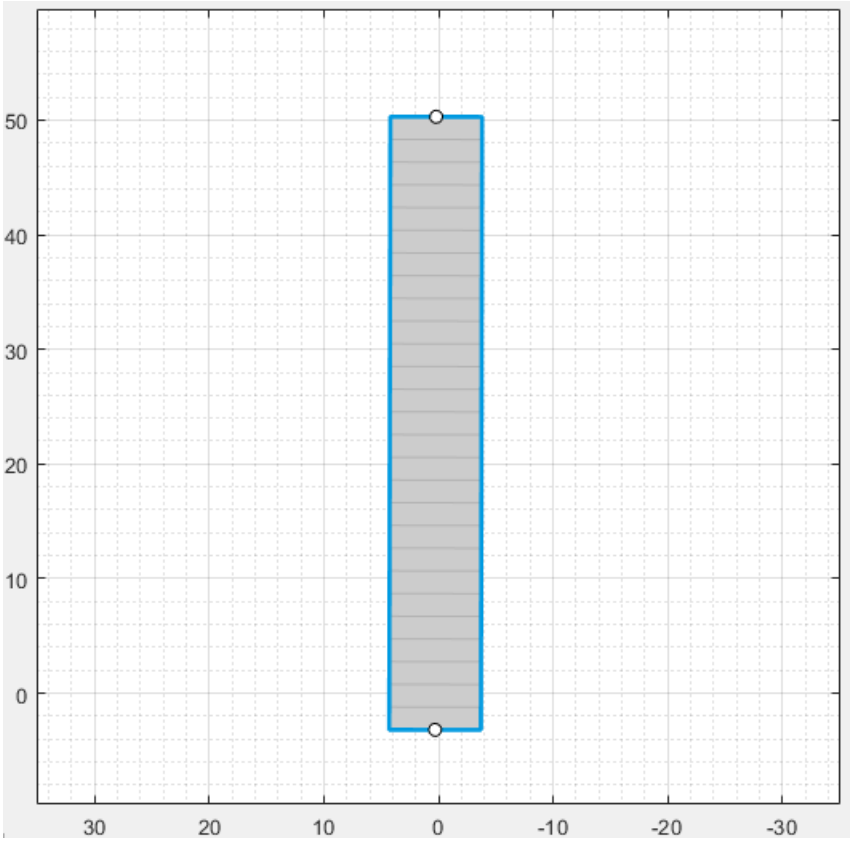
Open the **Driving Scenario Designer** app.

```
drivingScenarioDesigner
```

Add a straight road to the scenario. Right-click the **Scenario Canvas** pane and select **Add Road**. Extend the road toward the top of the canvas until it is about 50 meters long. Double-click to commit the road to the canvas.

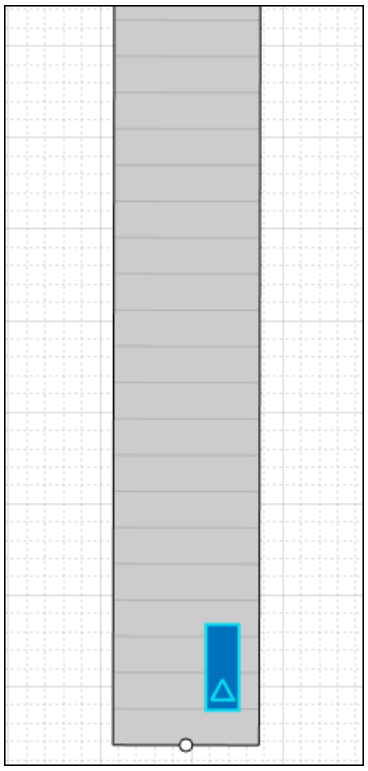


Expand the width of the road to leave enough room for the vehicle to complete the three-point turn. In the left pane, on the **Roads** tab, increase **Width (m)** from 6 to 8.



Add Vehicle

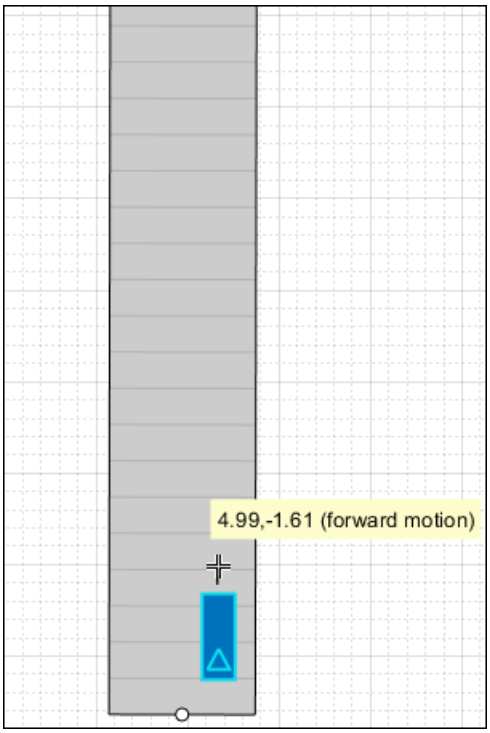
Add a vehicle to the road. Right-click the bottom right corner of the road and select **Add Car**. Zoom in on the vehicle and the first half of the road, which is where the vehicle will execute the three-point turn.



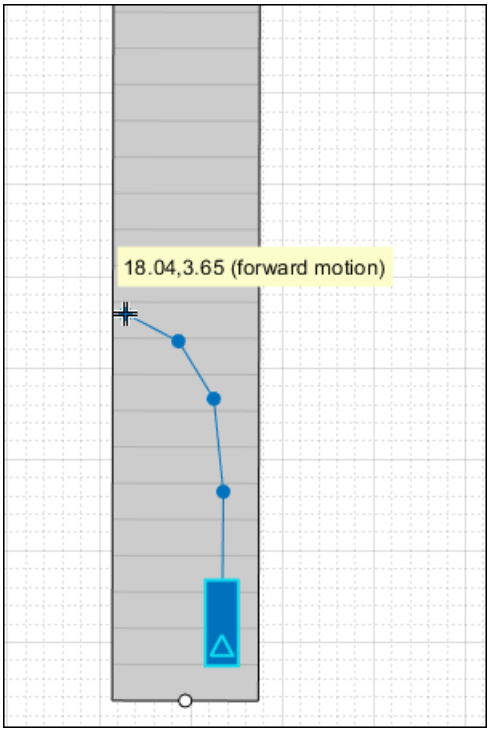
Add Trajectory

Specify a trajectory for the vehicle to complete a three-point turn.

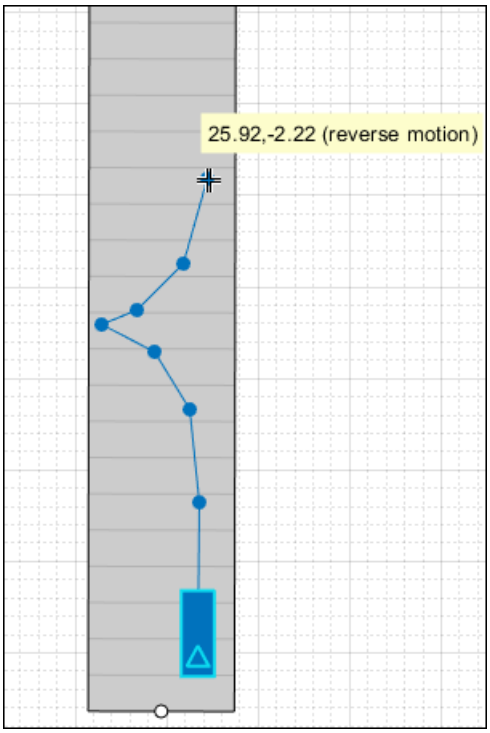
1 Right-click the vehicle and select **Add Forward Waypoints**. The pointer displays the (*x*,*y*) position on the canvas and the motion direction that the car will travel as you specify waypoints.



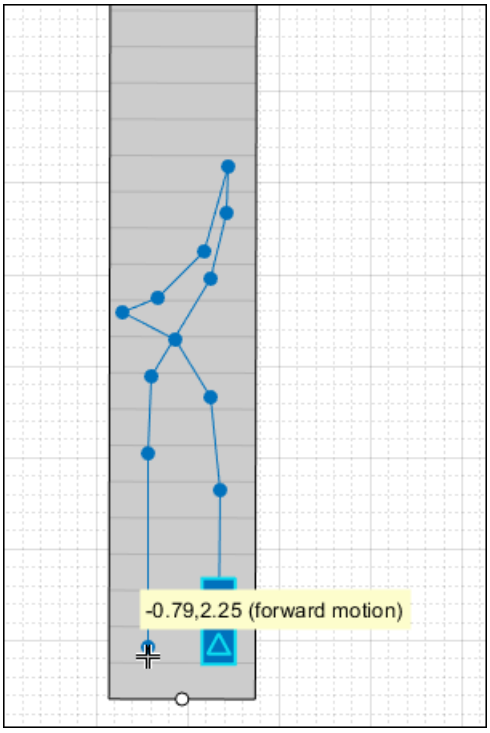
2 Specify the first segment of the three-point turn. Click to add waypoints that turn toward the left side of the road.



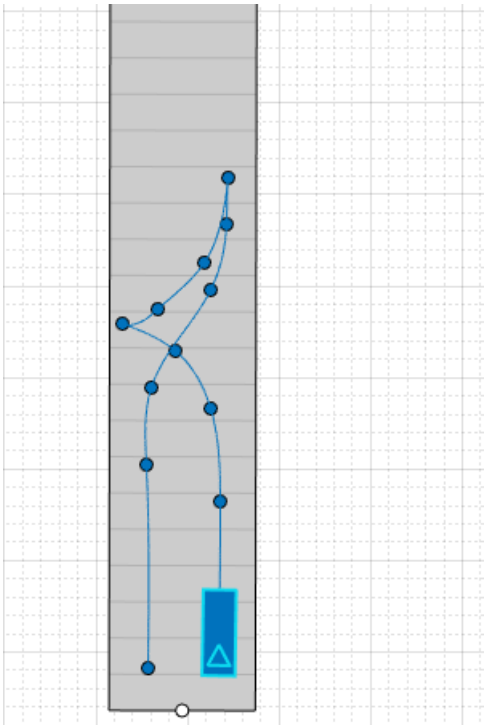
3 Specify the second segment of the three-point turn. Press **Ctrl+R** to switch to specifying reverse waypoints. Then, click to add waypoints that turn back toward the right side of the road.



4 Specify the third segment of the three-point turn. Press **Ctrl+F** to switch back to specifying forward waypoints. Then click to add waypoints that turn back toward the left side of the road, adjacent to the first specified waypoint.



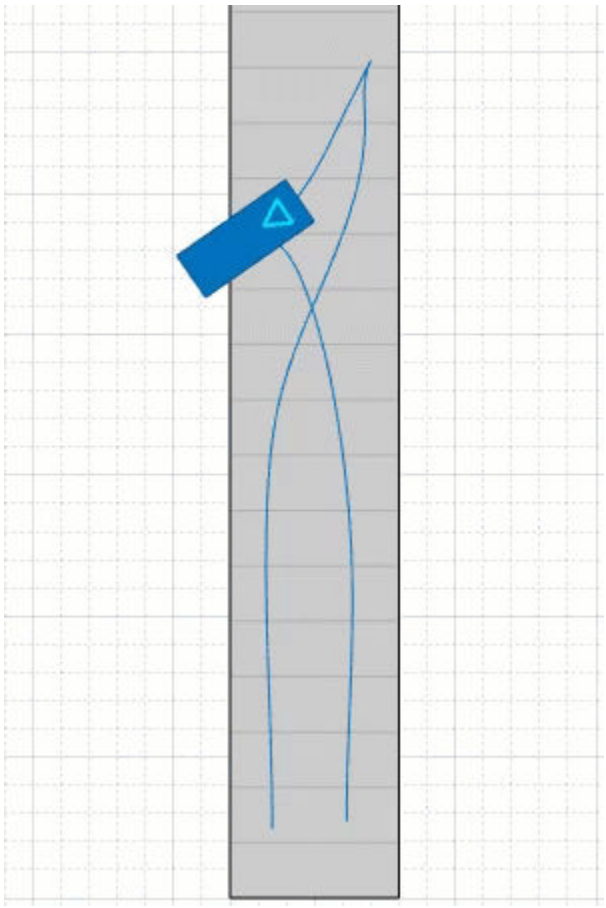
5 Press **Enter** to commit the waypoints to the canvas.



Run Simulation

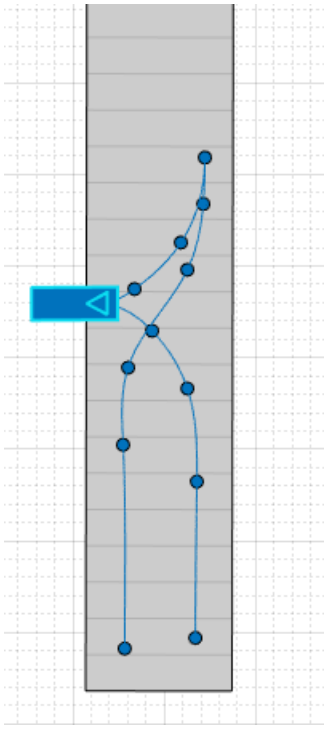
Run the simulation. To view the direction that the car is traveling, on the app toolstrip, select **Display > Show actor pose indicator during simulation**.

As the simulation runs, the vehicle briefly stops between each point in the three-point turn. When switching between forward and reverse motions in a trajectory, the app automatically sets the **v (m/s)** value at the waypoint where the switch occurs to `0`.

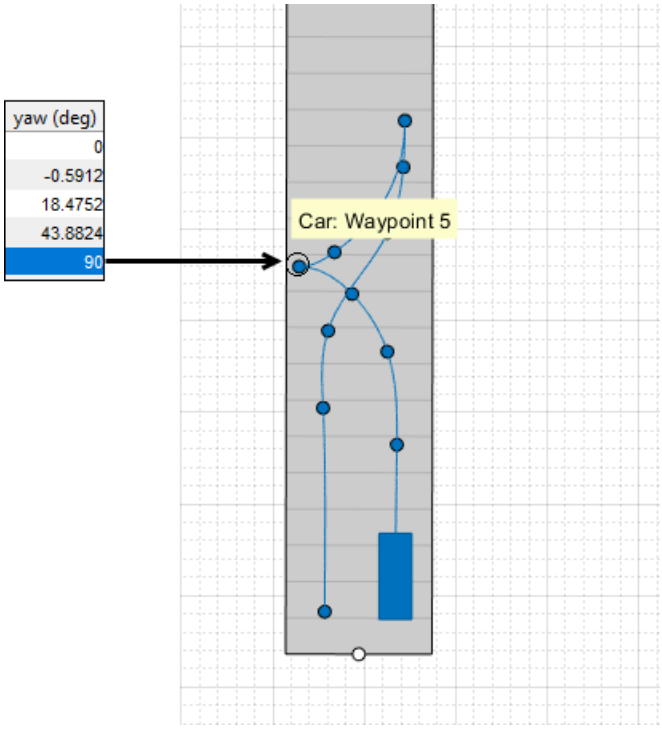


Adjust Trajectory Using Specified Yaw Values

To fine-tune the trajectory, set specific yaw orientation angles for the vehicle to reach at specific waypoints. For example, as the vehicle begins its reverse motion trajectory, suppose you want the vehicle to be at exactly a 90-degree angle from where it started.



First, determine the ID of the waypoint where the vehicle begins this reverse motion by moving your pointer over that waypoint. Then, in the **Waypoints, Speeds, Wait Times, and Yaw** table in the left pane, set the **yaw (deg)** value of the corresponding row to `90`. For example, if the vehicle begins its reverse motion at waypoint 5, update the fifth row of the **yaw (deg)** column.



During simulation, the vehicle is now turned exactly 90 degrees from where it began. To clear a previously set yaw value, right-click a waypoint and select **Restore Default Yaw**. You can also clear all set yaw values by right-clicking the vehicle and selecting **Restore Default Yaws**.

To work with prebuilt scenarios that use reverse motions and turns with specified yaw values, see the prebuilt autonomous emergency braking (AEB) scenarios described in “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-41.

See Also

Apps
Driving Scenario Designer

Functions
`trajectory`

More About

- “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-41
- “Create Actor and Vehicle Trajectories Programmatically” on page 7-442

Import OpenDRIVE Roads into Driving Scenario

OpenDRIVE [1] is an open file format that enables you to specify large and complex road networks. Using the **Driving Scenario Designer** app, you can import roads and lanes from an OpenDRIVE file into a driving scenario. You can then add actors and sensors to the scenario and generate synthetic lane and object detections for testing your driving algorithms developed in MATLAB. Alternatively, to test driving algorithms developed in Simulink, you can use a Scenario Reader block to read the road network and actors into a model.

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

Import OpenDRIVE File

Open the **Driving Scenario Designer** app. At the MATLAB command prompt, enter:

```
drivingScenarioDesigner
```

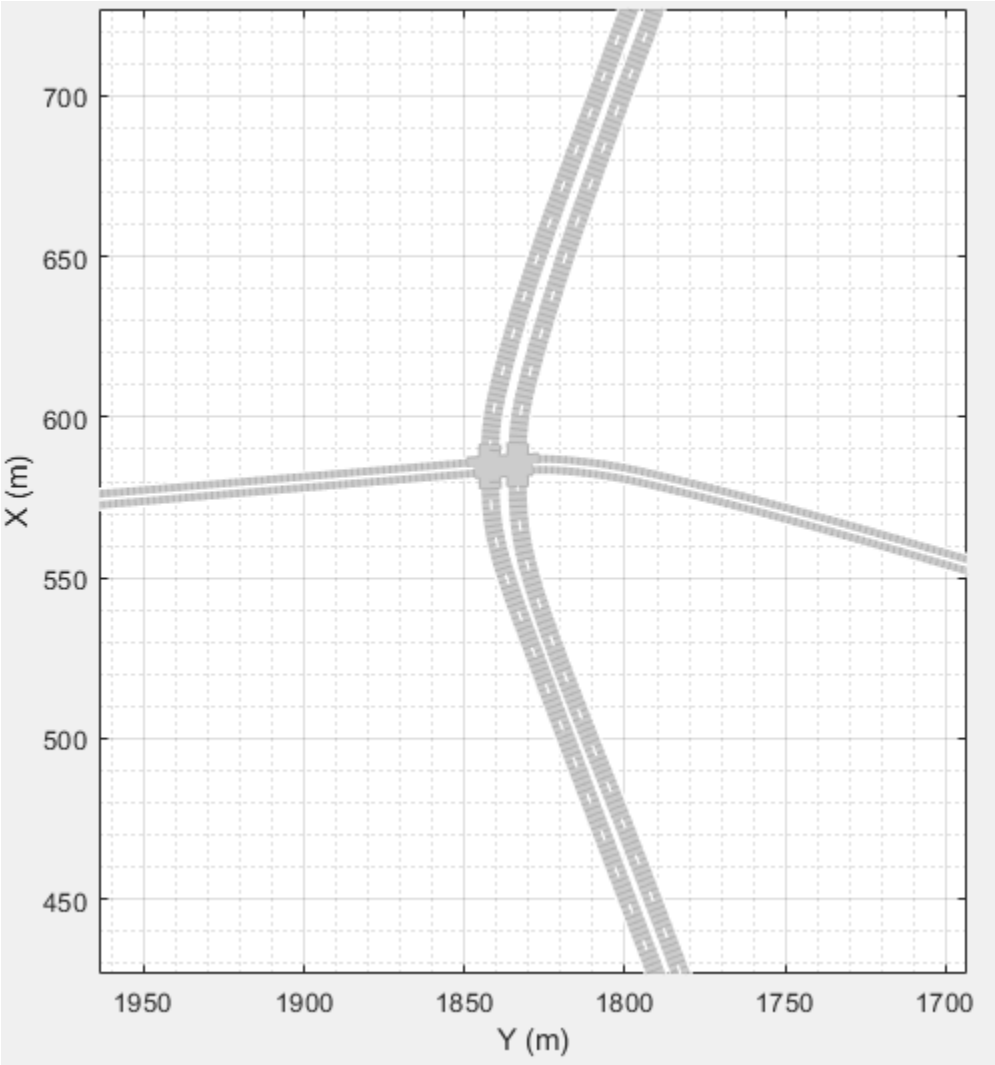
To import an OpenDRIVE file, on the app toolstrip, select **Import** > **OpenDRIVE Road Network**. The file you select must be a valid OpenDRIVE file of type `.xodr` or `.xml`. In addition, the file must conform with OpenDRIVE format specification version 1.4H.

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

```
matlabroot/examples/driving/data/intersection.xodr
```

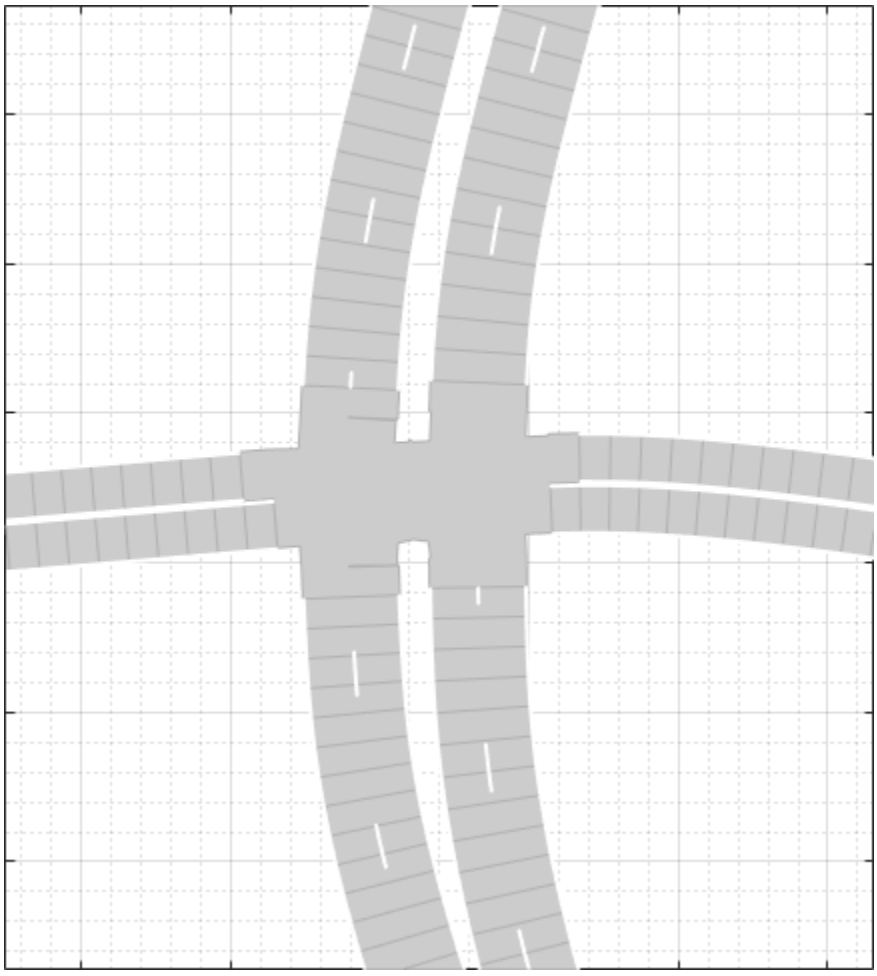
Because you cannot import an OpenDRIVE road network into an existing scenario file, the app prompts you to save your current driving scenario.

The **Scenario Canvas** of the app displays the imported road network. The roads in this network are thousands of meters long. Zoom in (press **Ctrl+Plus**) on the road to inspect it more closely.



Inspect Roads

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



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

OpenDRIVE Import Limitations

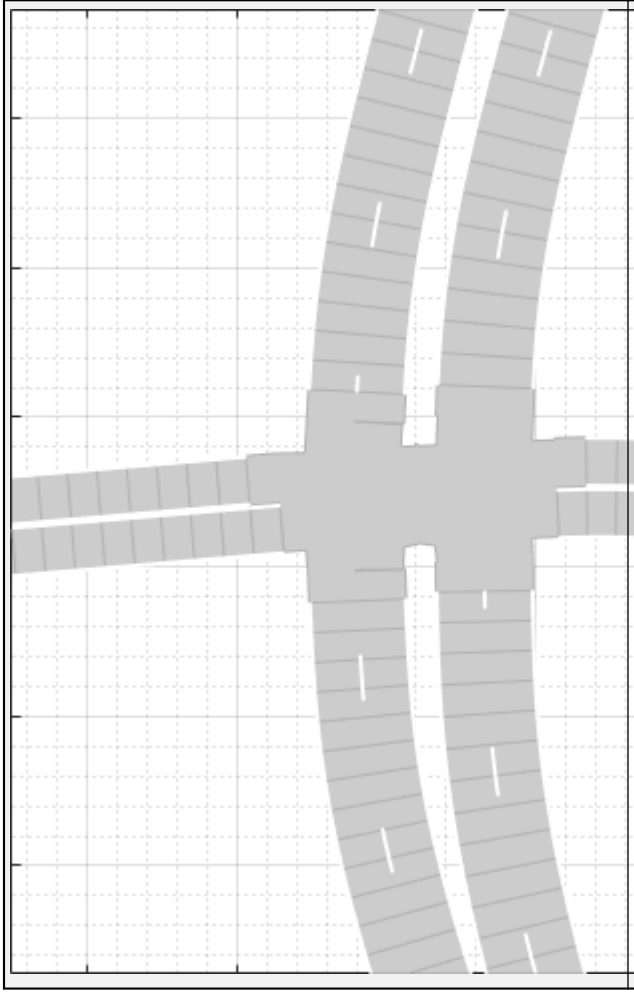
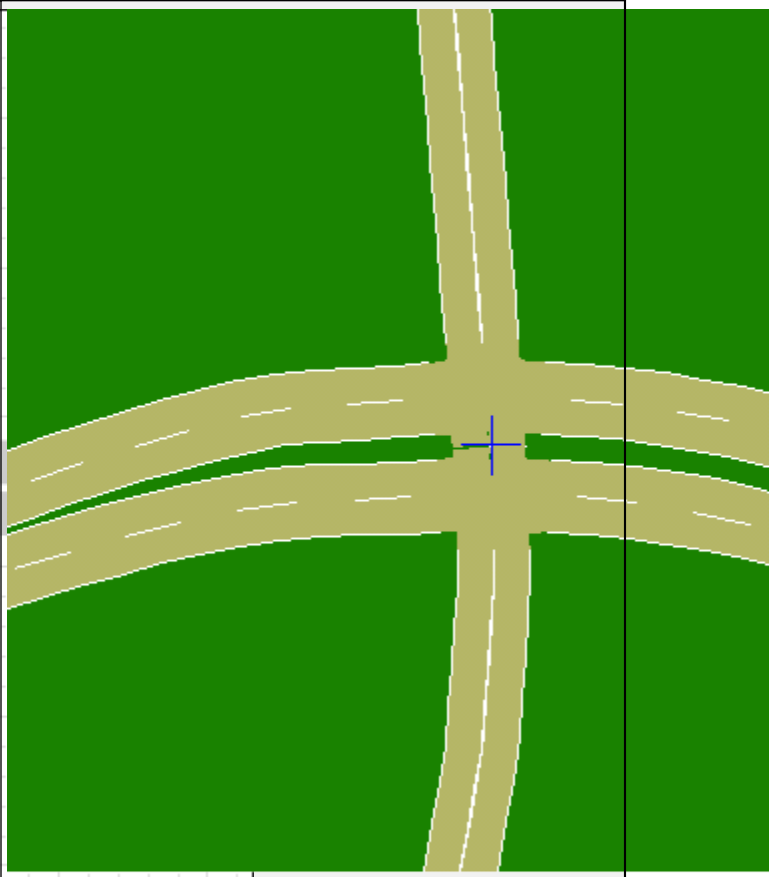
The **Driving Scenario Designer** app does not support all components of the OpenDRIVE specification.

- You can import only lanes, lane type information, and roads. The import of road objects and traffic signals is not supported.
- OpenDRIVE files containing large road networks can take up to several minutes to load. In addition, these road networks can cause slow interactions on the app canvas. Examples of large road networks include ones that model the roads of a city or ones with roads that are thousands of meters long.
- Lanes with variable widths are not supported. The width is set to the highest width found within that lane. For example, if a lane has a width that varies from 2 meters to 4 meters, the app sets the lane width to 4 meters throughout.
- Roads with lane type information specified as `driving`, `border`, `restricted`, `shoulder`, and `parking` are supported. Lanes with any other lane type information are imported as border lanes.
- Roads with multiple lane marking styles specified as `'Unmarked'`, `'Solid'`, `'DoubleSolid'`, `'Dashed'`, `'DoubleDashed'`, `'SolidDashed'`, and `'DashedSolid'` are supported.

- Lane marking styles `Bott Dots`, `Curbs`, and `Grass` are not supported. Lanes with these marking styles are imported as unmarked.

Road Orientation

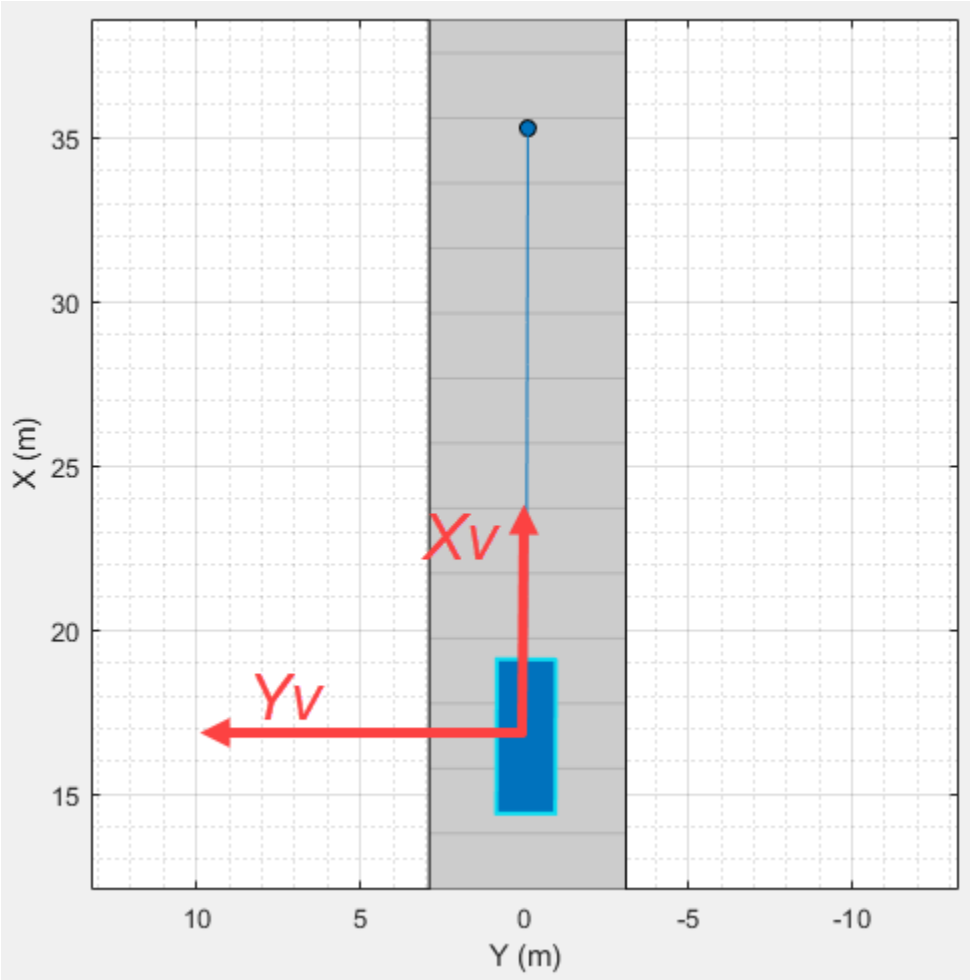
In the **Driving Scenario Designer** app, the orientation of roads can differ from the orientation of roads in other tools that display OpenDRIVE roads. The table shows this difference in orientation between the app and the OpenDRIVE ODR Viewer.

Driving Scenario Designer	OpenDRIVE ODR Viewer
	

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

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

- The X_V-axis (longitudinal axis) points forward from a vehicle in the scenario.
- The Y_V-axis (lateral axis) points to the left of the vehicle, as viewed when facing forward.



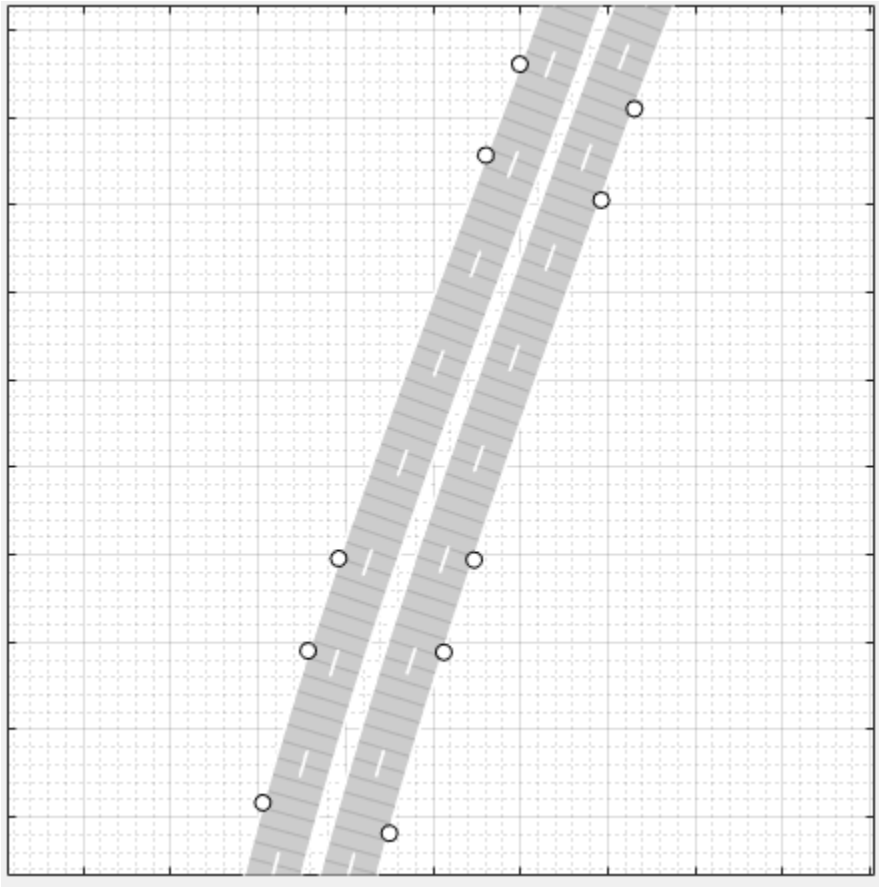
For more details about the coordinate systems, see “Coordinate Systems in Automated Driving Toolbox” on page 1-2.

Road Centers on Edges

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

Consider the divided highway in the imported OpenDRIVE file. First, enable road interactions so that you can see the road centers. In the bottom-left corner of the **Scenario Canvas**, click the Configure the Scenario Canvas button, and then select **Enable road interactions**.

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

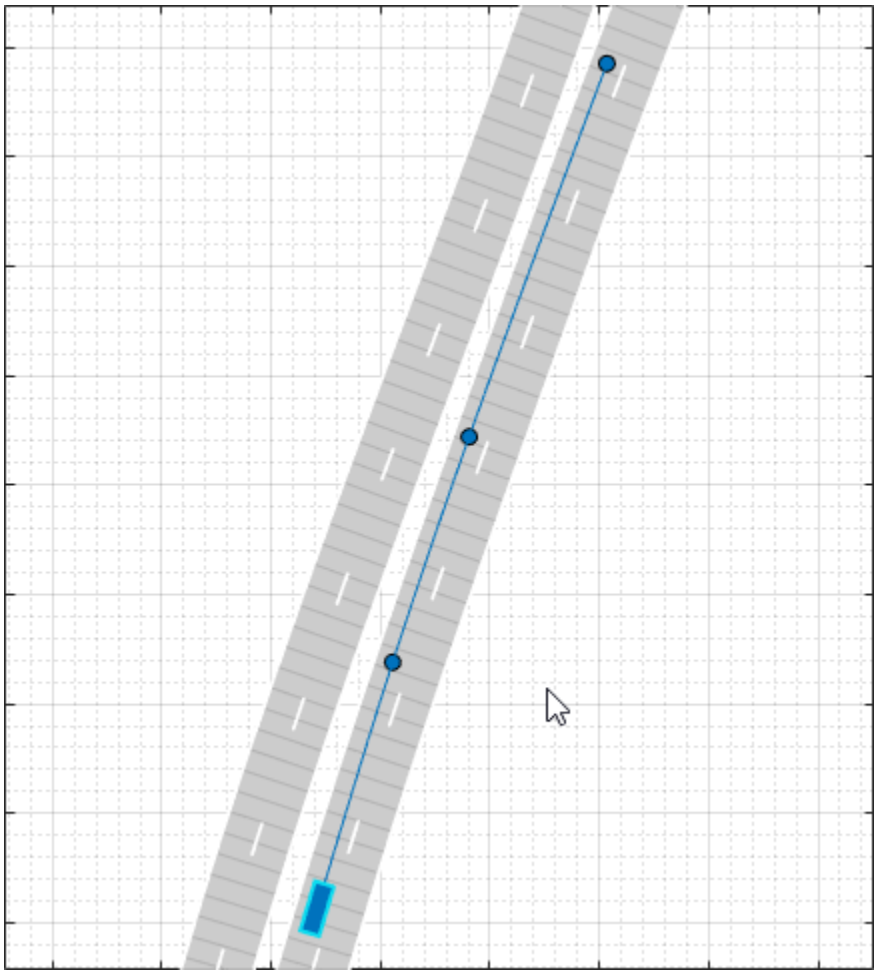


Add Actors and Sensors to Scenario

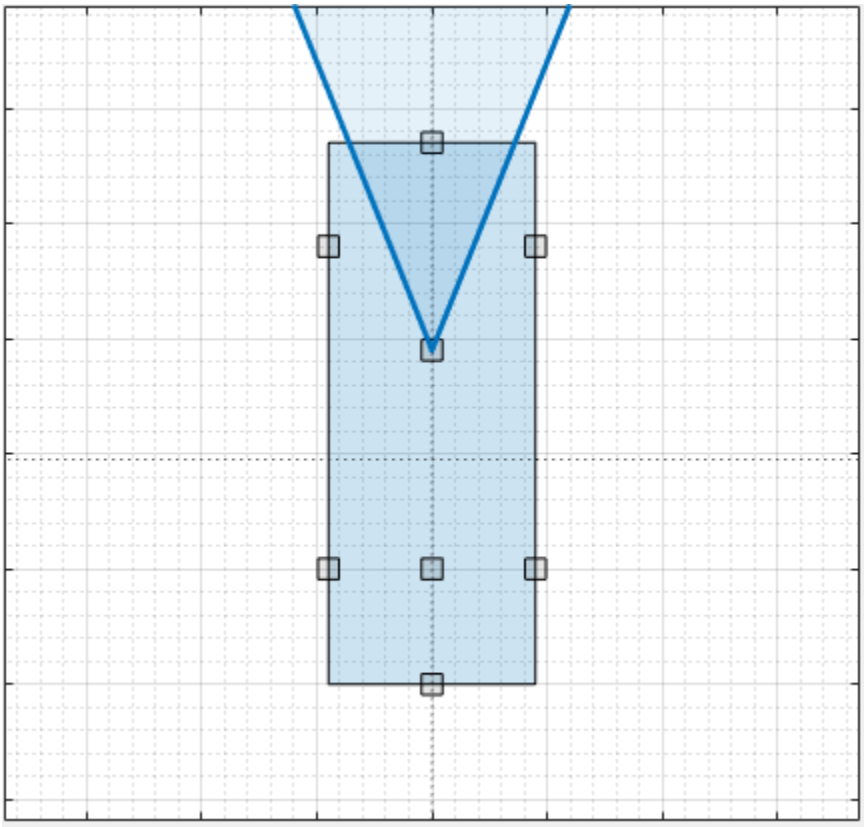
You can add actors and sensors to a scenario containing OpenDRIVE roads. However, you cannot add other roads to the scenario. If a scenario contains an OpenDRIVE road network, the **Add Road** button in the app toolstrip is disabled. In addition, you cannot import additional OpenDRIVE road networks into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

Before adding an actor and sensors, if you have road interactions enabled, consider disabling them to prevent you from accidentally dragging road centers and changing the road network. If road interactions are enabled, in the bottom-left corner of the **Scenario Canvas**, click the Configure the Scenario Canvas button  , and then clear **Disable road interactions**.

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



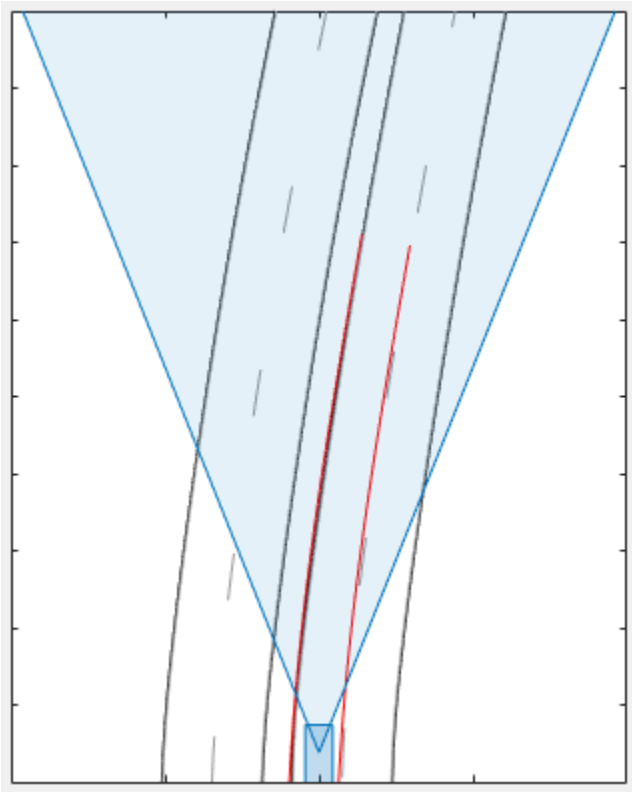
Add a camera sensor to the ego vehicle. On the app toolstrip, click **Add Camera**. Then, on the sensor canvas, add the camera to the predefined location representing the front window of the car.



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

Generate Synthetic Detections

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



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

The **Export** > **Export MATLAB Function** option is disabled. If a scenario contains OpenDRIVE roads, then you cannot export a MATLAB function that generates the scenario and its detections.

Save Scenario

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

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

```
drivingScenarioDesigner(scenarioFileName)
```

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

If you are developing a driving algorithm in Simulink, you can use a Scenario Reader block to read the roads and actors from the scenario file into your model. Scenario files containing large OpenDRIVE road networks can take up to several minutes to read into models.

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

Model. In this model, a Scenario Reader block reads the scenario and Radar Detection Generator and Vision Detection Generator blocks model the sensors.

References

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

See Also

Apps
Driving Scenario Designer

Blocks
Scenario Reader

Objects
`drivingScenario`

Functions
`roadNetwork`

More About

- “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2
- “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-19
- “Coordinate Systems in Automated Driving Toolbox” on page 1-2
- “Scenario Generation from Recorded Vehicle Data” on page 7-350
- “Export Driving Scenario to OpenDRIVE File” on page 5-89

External Websites

- ASAM OpenDRIVE

Export Driving Scenario to OpenDRIVE File

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 export the roads and lanes in a driving scenario to an OpenDRIVE file.

To programmatically export the roads, lanes, and junctions in a `drivingScenario` object to an OpenDRIVE file, use the `export` object function of the `drivingScenario` object.

The export file format conforms with OpenDRIVE format specification version 1.4H.

Load Scenario File

To open the **Driving Scenario Designer** app, enter this command at the MATLAB command prompt:

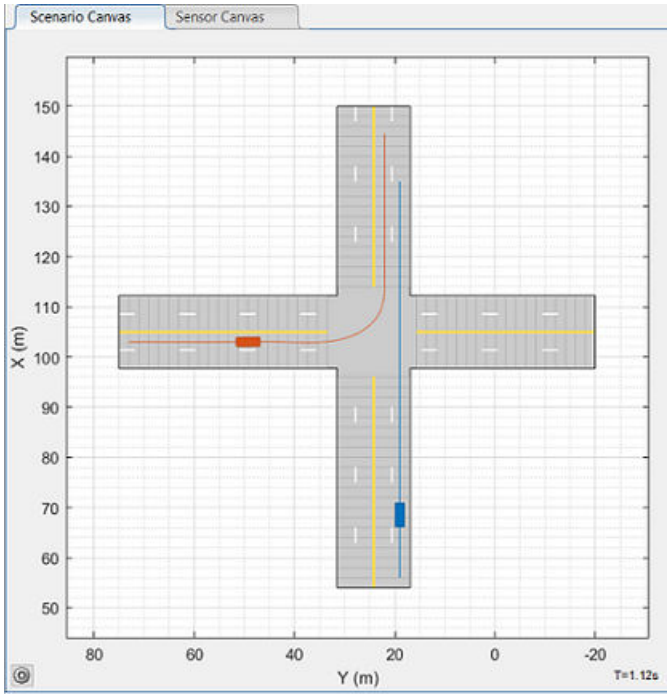
```
drivingScenarioDesigner
```

To load a scenario file, on the app toolstrip, select **Open** > **Scenario File**. The file you select must be a valid driving scenario session file with the `.mat` file extension.

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

```
matlabroot/examples/driving/data/StraightRoadScenario.mat
```

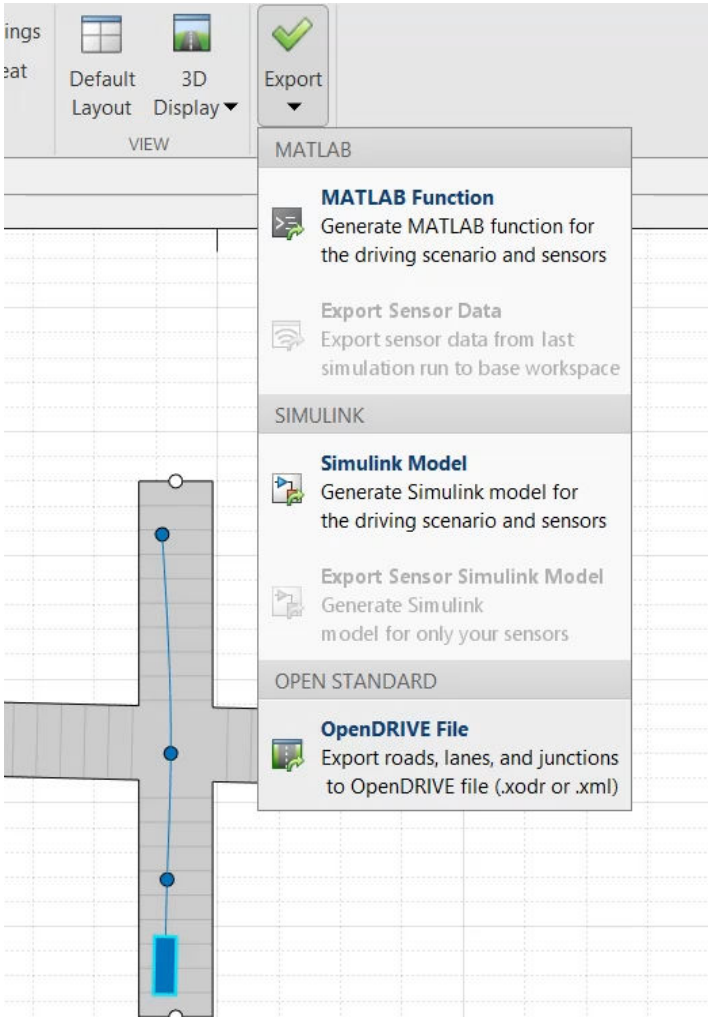
The **Scenario Canvas** tab displays the scenario.



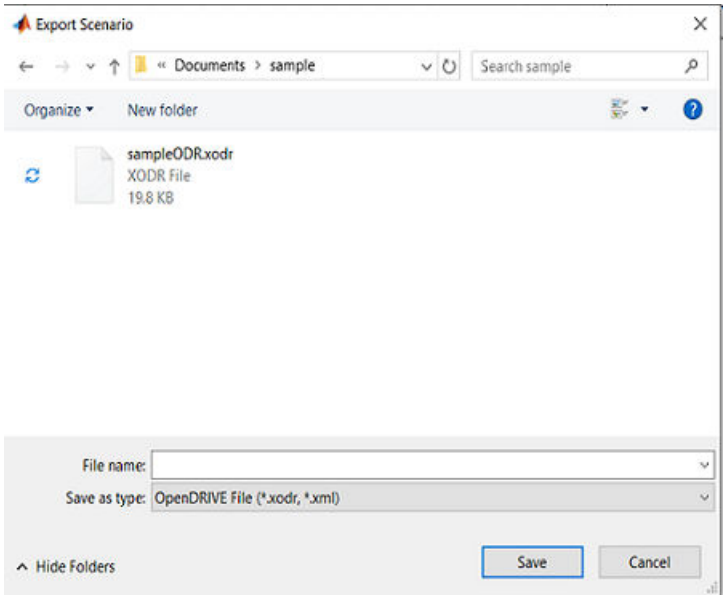
Note You can also create a scenario by using the **Driving Scenario Designer** app, and then export the scenario to an OpenDRIVE file. For information about how to create a scenario, see “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2.

Export to OpenDRIVE

To export the roads, lanes, and junctions in the scenario to an OpenDRIVE file, on the app toolstrip, select **Export** > **OpenDRIVE File**.

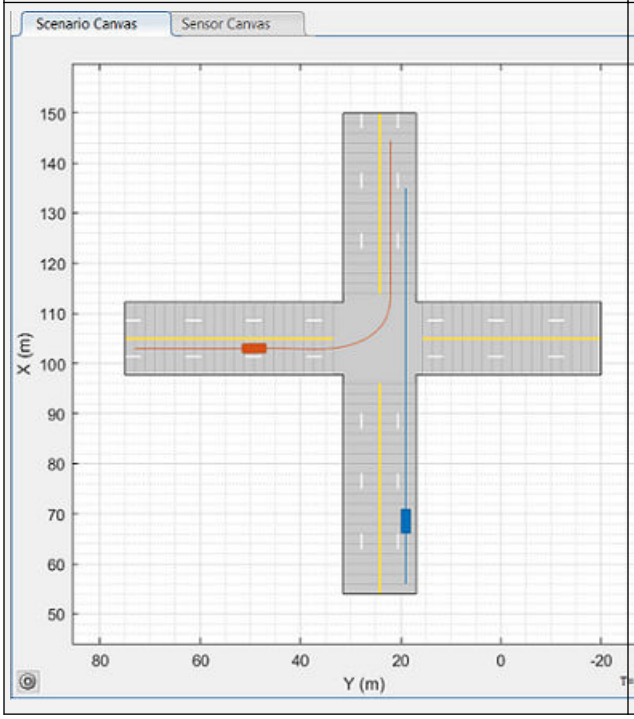
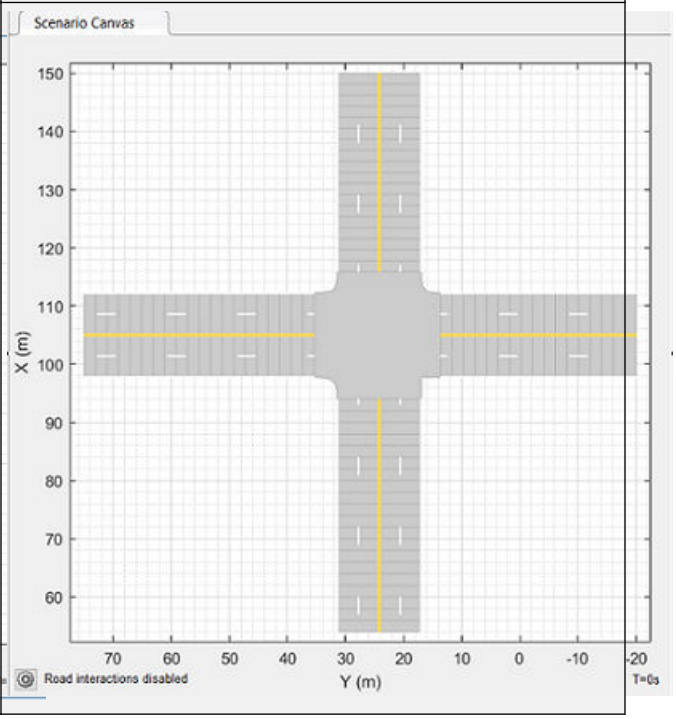


The app opens the **Export Scenario** window. In this window, specify a name for the output OpenDRIVE file, and choose a file extension. The OpenDRIVE files have either `.xodr` or `.xml` file extension. Once you have specified a file name, click **Save**. If the specified file already exists, the app overwrites the existing file.



Inspect Exported Scenario

To inspect the exported scenario using the **Driving Scenario Designer** app, on the app toolstrip, select **Import** > **OpenDRIVE File**. Select the exported OpenDRIVE file and click **Open**.

Actual Scenario	Exported Scenario
	

Notice that the exported road network does not have boundary lines and there are variations at the road junction. For more information about the variations, see `export`.

See Also

Apps
Driving Scenario Designer

Objects
`drivingScenario`

Functions
`export`

More About

- “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2
- “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-19
- “Coordinate Systems in Automated Driving Toolbox” on page 1-2
- “Import OpenDRIVE Roads into Driving Scenario” on page 5-79

External Websites

- opendrive.org

Import HERE HD Live Map Roads into Driving Scenario

HERE HD Live Map[3] (HERE HDLM), developed by HERE Technologies, is a cloud-based web service that enables you to access highly accurate, continuously updated map data. Using the **Driving Scenario Designer** app, you can import map data from the HERE HDLM service and use it to generate roads for your driving scenarios.

This example focuses on importing map data in the app. Alternatively, to import HERE HDLM roads into a `drivingScenario` object, use the `roadNetwork` function.

Set Up HERE HDLM Credentials

To access the HERE HDLM web service, you need to enter valid HERE credentials obtained from HERE Technologies. Set up these credentials by using the `hereHDLMCredentials` function. At the MATLAB command prompt, enter:

```
hereHDLMCredentials setup
```

In the HERE HD Live Map Credentials dialog box, enter a valid **Access Key ID** and **Access Key Secret**. To save your credentials for future MATLAB sessions on your machine, in the dialog box, select **Save my credentials between MATLAB sessions** and click **OK**. The credentials are now saved for the rest of your MATLAB session on your machine.

If you need to change your credentials, you can delete them and set up new ones by using the `hereHDLMCredentials` function.

Specify Geographic Coordinates

To select the roads you want to import, you need to specify a region of interest from which to obtain the road data. To define this region of interest, specify latitude and longitude coordinates that are near that road data. You can specify coordinates for a single point or a set of points, such as ones that make up a driving route.

Specify the coordinates from a driving route.

1. Load a sequence of latitude and longitude coordinates that make up a driving route. At the MATLAB command prompt, enter these commands:

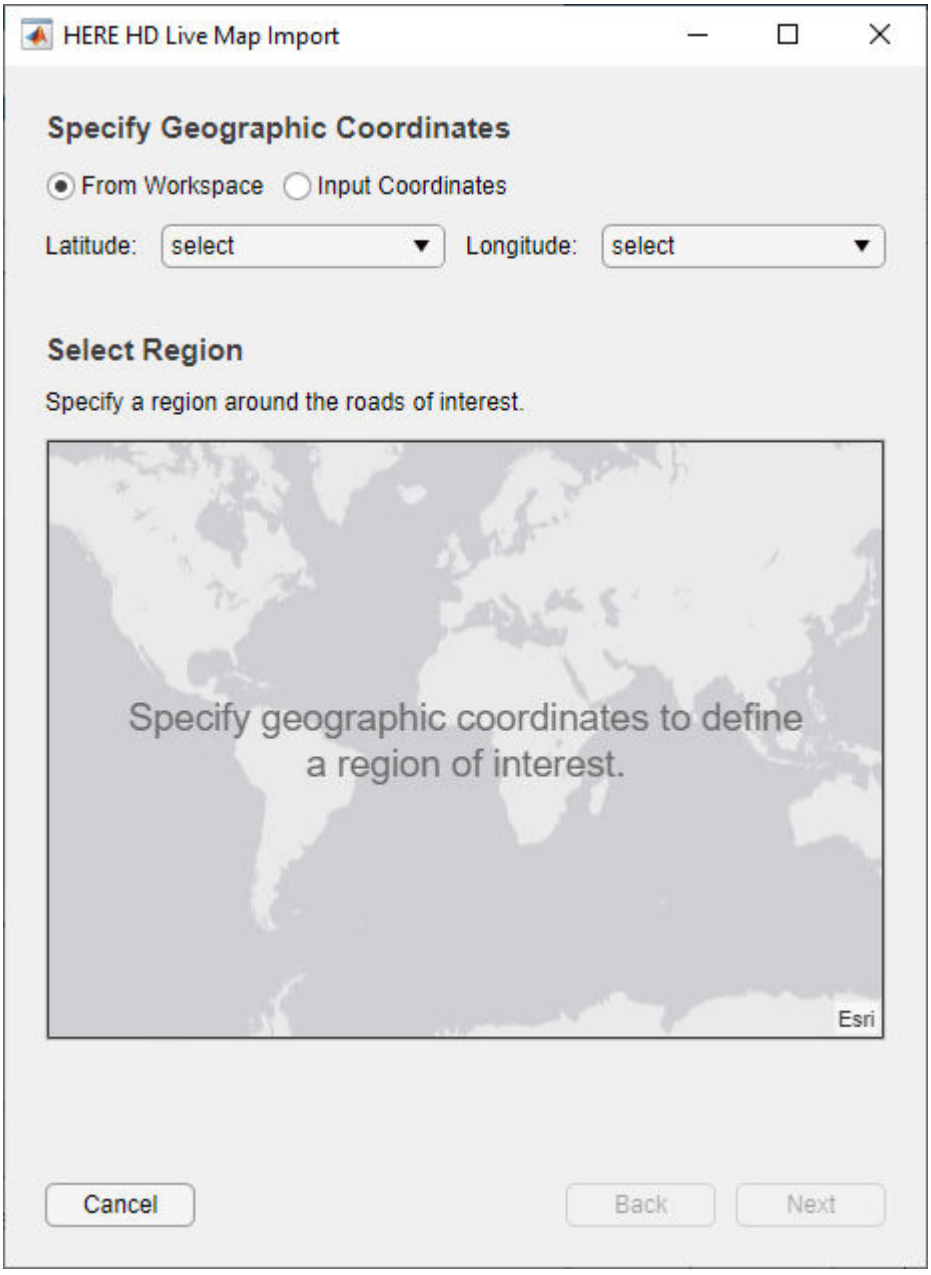
   ```
   data = load('geoSequence.mat');
   lat = data.latitude;
   lon = data.longitude;
   ```

2. Open the app.

   ```
   drivingScenarioDesigner
   ```

3. On the app toolstrip, select **Import > HERE HD Live Map**. If you previously entered or saved HERE credentials, then the dialog box opens directly to the page where you can specify geographic coordinates.

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 (access_key_id and access_key_secret) for using the HERE Service.



4 Leave **From Workspace** selected, and then select the variables for the route coordinates.

- Set the **Latitude** parameter to `lat`.
- Set the **Longitude** parameter to `lon`.

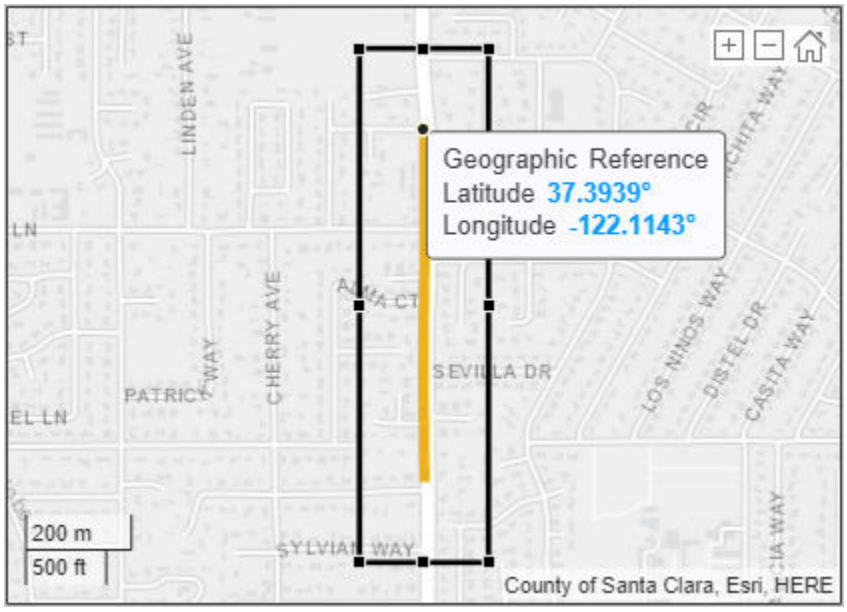
This table describes the complete list of options for specifying latitude and longitude coordinates.

Specify Geographic Coordinates Parameter Value	Description	Latitude Parameter Value	Longitude Parameter Value
From Workspace	Specify a set of latitude and longitude coordinates, such as from a driving route obtained through a GPS. These coordinates must be stored as variables in the MATLAB workspace.	Workspace variables containing vectors of decimal values in the range [–90, 90]. Units are in degrees. **Latitude** and **Longitude** must be the same size. After you select a **Latitude** variable, the **Longitude** list includes only variables of the same size as your **Latitude** selection.	Workspace variables containing vectors of decimal values in the range [–180, 180]. Units are in degrees. **Latitude** and **Longitude** must be the same size. After you select a **Longitude** variable, if you select a **Latitude** variable of a different size, the dialog box clears your **Longitude** selection.
Input Coordinates	Specify latitude and longitude coordinates for a single geographic point.	Decimal scalar in the range [–90, 90]. Units are in degrees.	Decimal scalar in the range [–180, 180]. Units are in degrees.

Select Region Containing Roads

After you specify the latitude and longitude coordinates, the **Select Region** section of the dialog box displays these coordinates in orange on a map. The geographic reference point, which is the first coordinate in the driving route, is also displayed. This point is the origin of the imported scenario. Click this point to show or hide the coordinate data.

The coordinates are connected in a line. A rectangular region of interest displays around the coordinates. In the next page of the dialog box, you select the roads to import based on which roads are at least partially within this region.

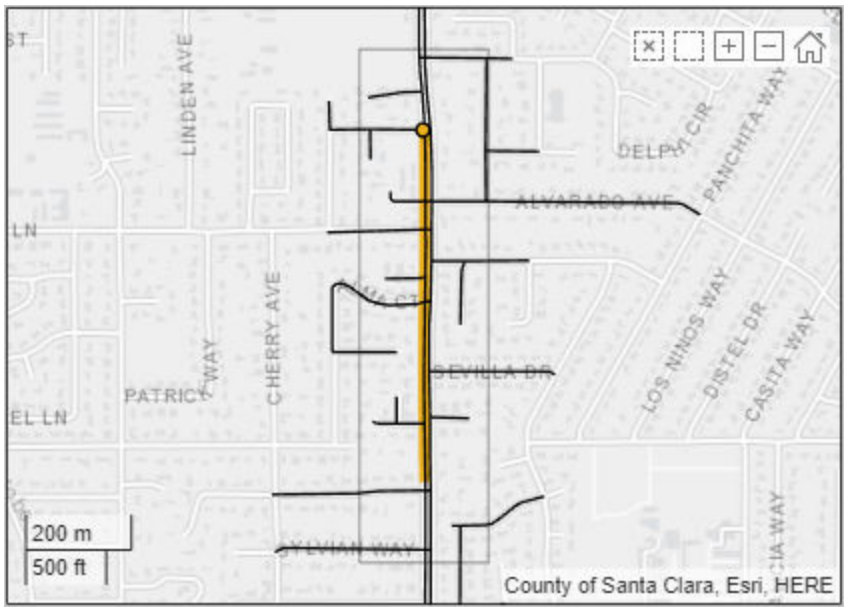


You can change the size of this region or move it around to select different roads. To zoom in and out of the region, use the buttons in the top-right corner of the map display.

With the coordinates still enclosed within the region, click **Next**.

Select Roads to Import

After you select a region, the **Select Roads** section of the dialog box displays selectable roads in black.



Using the selected region from the previous section, select the roads that are nearest to the driving route by clicking **Select Nearest Roads**. The selected roads are overlaid onto the driving route and appear in blue.

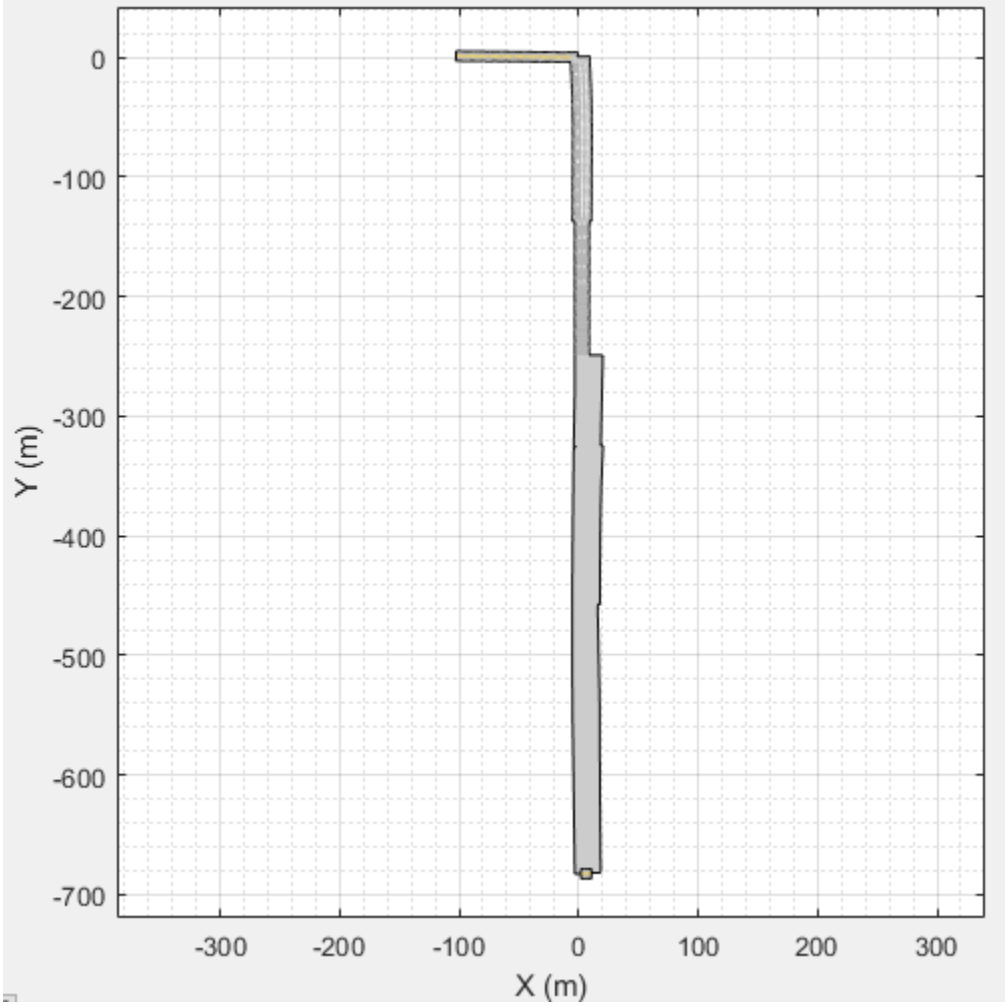
This table describes additional actions you can take for selecting roads from a region.

Goal	Action
Select individual roads from the region.	Click the individual roads to select them.
Select all roads from the region.	Click **Select All**.
Select all but a few roads from the region.	Click **Select All**, and then click the individual roads to deselect them.
Select roads from the region that are nearest to the specified coordinates.	Click **Select Nearest Roads**. Use this option when you have a sequence of nonsparse coordinates. If your coordinates are sparse or the underlying HERE HDLM data for those coordinates are sparse, then the app might not select the nearest roads.
Select a subset of roads from a region, such as all roads in the upper half of the region.	In the top-left corner of the map display, click the Select Roads button. Then, draw a rectangle around the roads to select. • To deselect a subset of roads from this selection, click the Deselect Roads button. Then, draw a rectangle around the roads to deselect. • To deselect all roads and start over, click **Deselect All**.

Note The number of roads you select has a direct effect on app performance. Select the fewest roads that you need to create your driving scenario.

Import Roads

With the roads nearest to the route still selected, click **Import**. The app imports the HERE HDLM roads and generates a road network.



To maintain the same alignment with the geographic map display, the *X*-axis of the **Scenario Canvas** is on the bottom and the *Y*-axis is on the left. In driving scenarios that are not imported from maps, the *X*-axis is on the left and the *Y*-axis is on the bottom. This alignment is consistent with the Automated Driving Toolbox world coordinate system.

The origin of the scenario corresponds to the geographic reference point and is the first point specified in the driving route. Even if you select roads from the end of a driving route, the origin is still anchored to this first point. If you specified a single geographic point by using the **Input Coordinates** option, then the origin is that point.

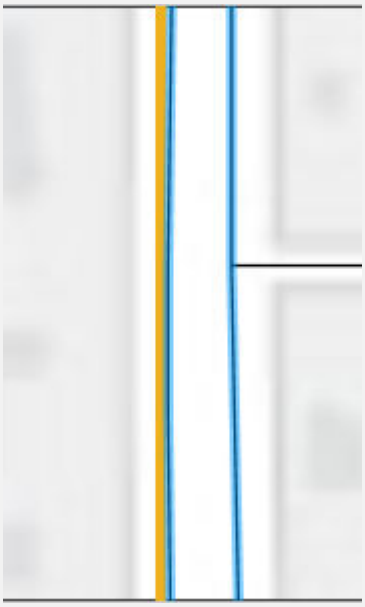
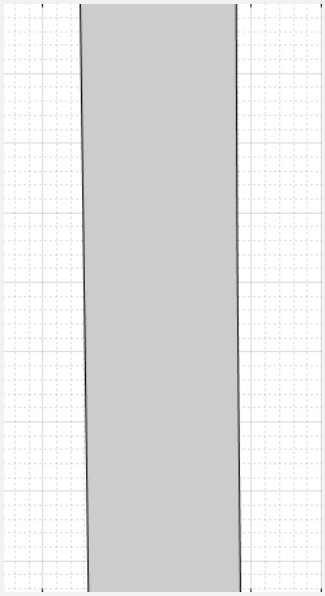
By default, road interactions are disabled. Disabled road interactions prevent you from accidentally modifying the network and reduces visual clutter by hiding the road centers. If you want to modify the roads, in the bottom-left corner of the **Scenario Canvas**, click the Configure the Scenario Canvas button  . Then, select **Enable road interactions**.

Note In some cases, the app is unable to import all selected roads. This issue can occur if the curvature of the road is too sharp for the app to render it properly. In these cases, the app pauses the import, and the dialog box highlights the nonimportable roads in red. To continue importing all other selected roads, click **Continue**.

Compare Imported Roads Against Map Data

The generated road network in the app has several differences from the actual HERE HDLM road network. For example, the actual HERE HDLM road network contains roads with varying widths and varying numbers of lanes. The **Driving Scenario Designer** app does not support these features. Instead, the app sets each road to have the maximum width and the maximum number of lanes found along its entire length. These changes increase the widths of the roads and causes roads to overlap and appear as one road. Sensors that detect lanes are unable to detect the covered lanes.

This table shows the difference between a portion of the HERE HDLM road network and the imported driving scenario.

HERE HDLM Road Network	Imported Driving Scenario
	

Road networks generated from imported HERE HDLM data can have several differences from the actual HERE HDLM road network. For more details on the unsupported HERE HDLM road and lane features, see the “Limitations” section of the **Driving Scenario Designer** app reference page.

Save Scenario

Save the scenario file. After you save the scenario, you cannot import additional HERE HDLM roads into it. Instead, you need to create a new scenario and import a new road network.

You can now add actors and sensors to the scenario, generate synthetic lane and object detections for testing your driving algorithms, or import the scenario into Simulink.

See Also

Apps
Driving Scenario Designer

Blocks
Scenario Reader

Objects
`drivingScenario`

Functions
`roadNetwork`

More About

- “Read and Visualize HERE HD Live Map Data” on page 4-7
- “Coordinate Systems in Automated Driving Toolbox” on page 1-2
- “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2
- “Import OpenStreetMap Data into Driving Scenario” on page 5-101

External Websites

- HERE Technologies

Import OpenStreetMap Data into Driving Scenario

OpenStreetMap is a free, open-source web map service that enables you to access crowdsourced map data. Using the **Driving Scenario Designer** app, you can import map data from OpenStreetMap and use it to generate roads for your driving scenarios.

This example focuses on importing map data in the app. Alternatively, to import OpenStreetMap roads into a `drivingScenario` object, use the `roadNetwork` function.

Select OpenStreetMap File

To import a road network, you must first select an OpenStreetMap file containing the road geometry for that network. To export these files from `openstreetmap.org`, specify a map location, manually adjust the region around this location, and export the road geometry for that region to an OpenStreetMap with extension `.osm`. Only roads whose whole lengths are within this specified region are exported. In this example, you select an OpenStreetMap file that was previously exported from this website.

1. Open the **Driving Scenario Designer** app.

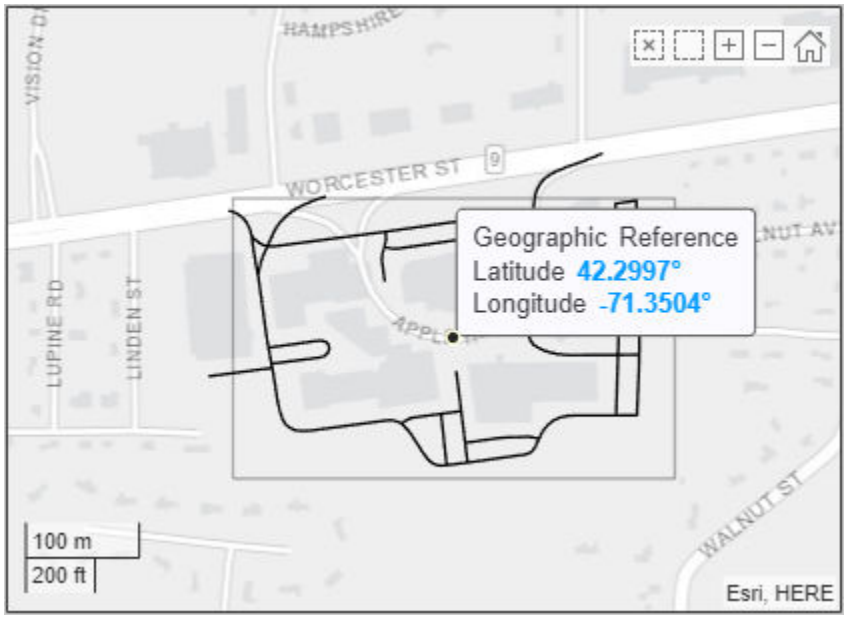
   ```
   drivingScenarioDesigner
   ```

2. On the app toolstrip, select **Import** > **OpenStreetMap**.
3. In the OpenStreetMap Import dialog box, browse for this file, where *`matlabroot`* is the root of your MATLAB folder:

   ```
   matlabroot/examples/driving/data/applehill.osm
   ```

   The file was downloaded from `https://www.openstreetmap.org`, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), `https://opendatacommons.org/licenses/odbl/`.

The **Select Roads** section of the dialog box displays the road network for the MathWorks® Apple Hill campus. The gray box represents the map region selected from `openstreetmap.org`. The center point of the gray box is the geographic reference point. Click this point to show or hide the coordinate data. When the roads are imported into that app, this point becomes the origin of the driving scenario.

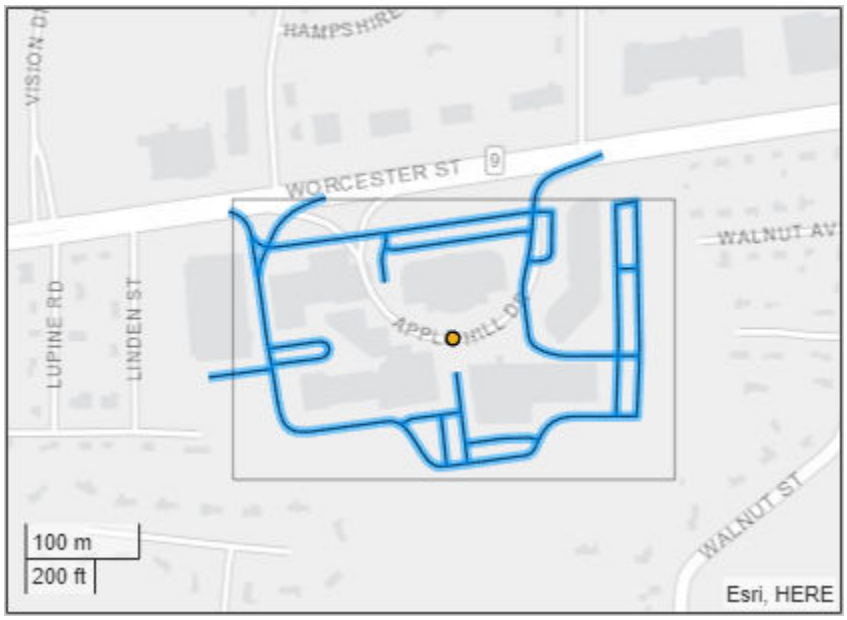


Select Roads to Import

In the **Select Roads** section of the dialog box, select the roads that you want to import into a driving scenario. The selectable roads are in black.

Note The number of roads you select has a direct effect on app performance. Select the fewest roads that you need to create your driving scenario.

Because this road network is small, click **Select All** to select all roads. The selected roads appear in blue.

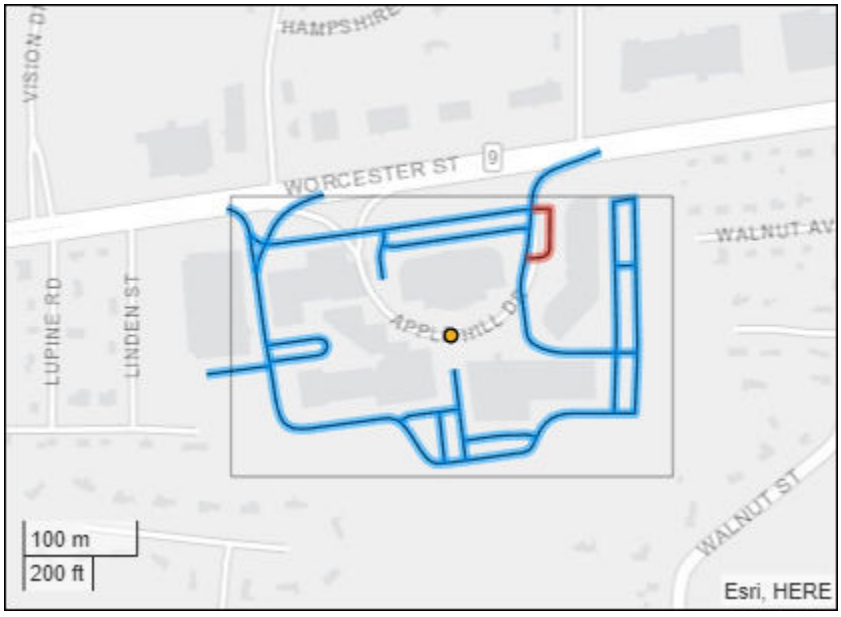


This table describes the actions you can take for selecting roads to import.

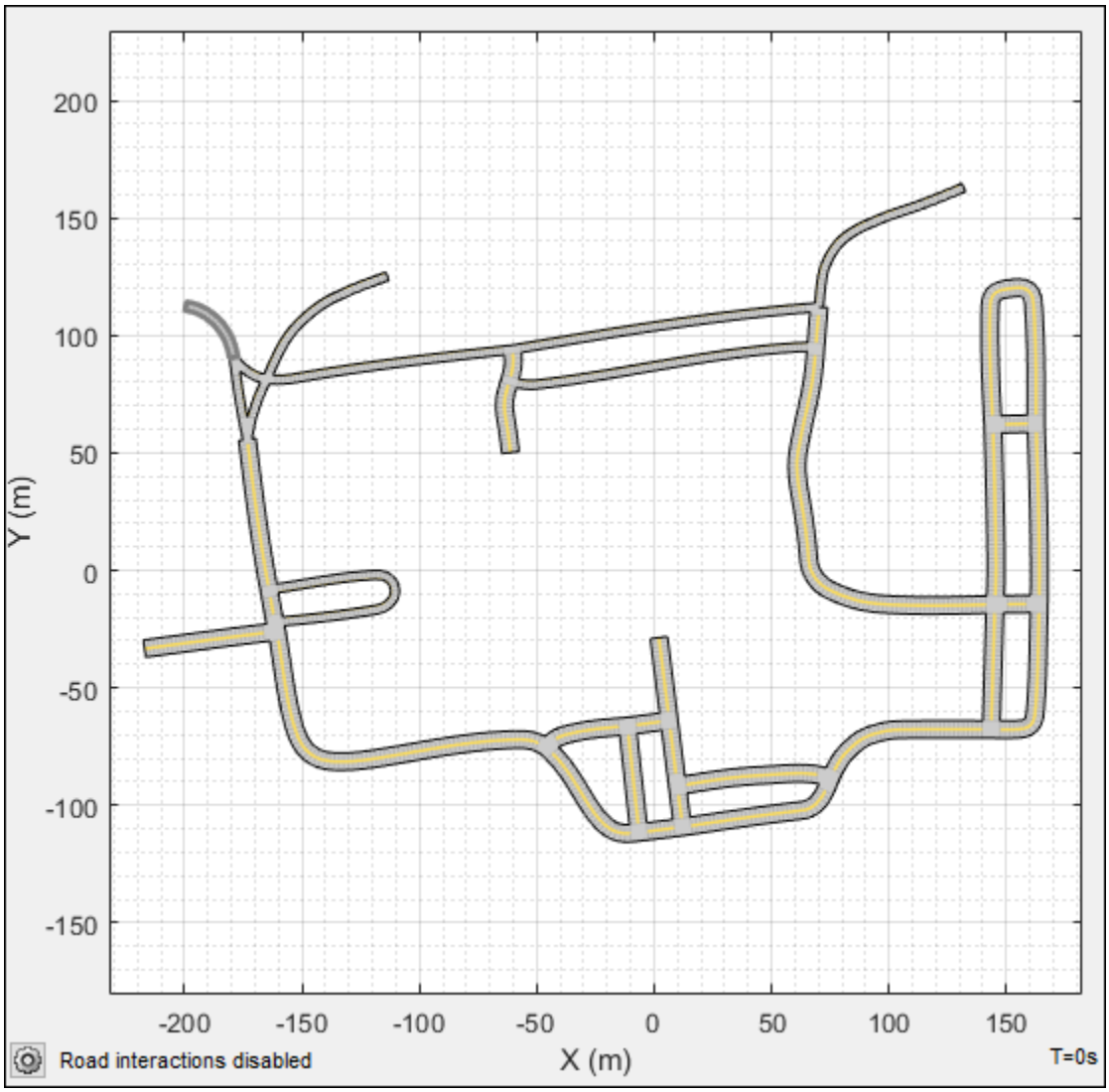
Goal	Action
Select individual roads from the region.	Click the individual roads to select them.
Select all roads from the region.	Click **Select All**.
Select all but a few roads from the region.	Click **Select All**, and then click the individual roads to deselect them.
Select a subset of roads from a region, such as all roads in the upper half of the region.	In the top-left corner of the map display, click the Select Roads button. Then, draw a rectangle around the roads to select. • To deselect a subset of roads from this selection, click the Deselect Roads button. Then, draw a rectangle around the roads to deselect. • To deselect all roads and start over, click **Deselect All**.

Import Roads

With all roads in the network still selected, click **Import**. The app pauses the import and highlights one of the roads in red. The app is unable to render the geometry of this road properly, so the road cannot be imported.



Click **Continue** to continue importing all other selected roads. The app imports the roads and generates a road network.

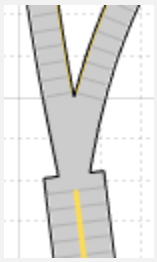


To maintain the same alignment with the geographic map display, the *X*-axis of the **Scenario Canvas** is on the bottom and the *Y*-axis is on the left. In driving scenarios that are not imported from maps, the *X*-axis is on the left and the *Y*-axis is on the bottom. This alignment is consistent with the Automated Driving Toolbox world coordinate system. The origin of the scenario corresponds to the geographic reference point.

By default, road interactions are disabled. Disabled road interactions prevent you from accidentally modifying the network and reduce visual clutter by hiding the road centers. If you want to modify the roads, in the bottom-left corner of the **Scenario Canvas**, click the Configure the Scenario Canvas button . Then, select **Enable road interactions**.

Compare Imported Roads Against Map Data

The generated road network in the app differs from the OpenStreetMap road network. For example, examine the difference in this portion of the road network.

OpenStreetMap Road Network	Imported Driving Scenario
	

The transition between roads in the imported scenario is more abrupt because the app does not support the gradual tapering of lanes as the number of lanes change. In addition, because the app does not import lane-level information from OpenStreetMap, the number of lanes in the generated road network is based only on the direction of travel specified in the road network, where:

- All one-way roads are imported as single-lane roads.
- All two-way roads are imported as two-lane roads.

These lanes all have the same width, which can lead to abrupt transitions such as in the example shown in the table.

For more details on the limitations of importing OpenStreetMap data, see the “Limitations” section of the **Driving Scenario Designer** app reference page.

Save Scenario

Save the scenario file. After you save the scenario, you cannot import additional OpenStreetMap roads into it. Instead, you must create a new scenario and import a new road network.

You can now add actors and sensors to the scenario, generate synthetic lane and object detections for testing your driving algorithms, or import the scenario into Simulink.

See Also

Apps
Driving Scenario Designer

Blocks
Scenario Reader

Objects
`drivingScenario`

Functions
`roadNetwork`

More About

- “Coordinate Systems in Automated Driving Toolbox” on page 1-2
- “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2

- “Import HERE HD Live Map Roads into Driving Scenario” on page 5-93

External Websites

- openstreetmap.org

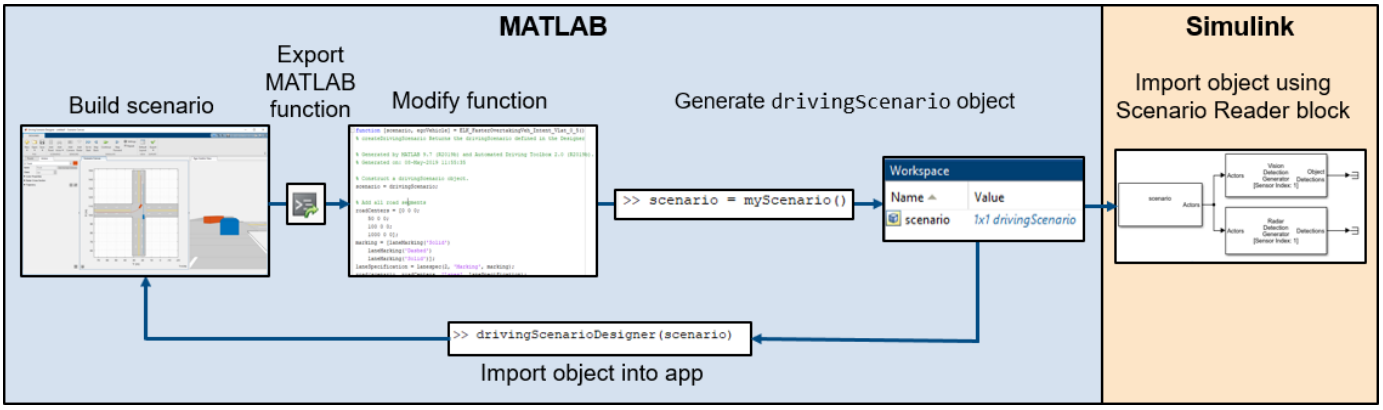
Create Driving Scenario Variations Programmatically

This example shows how to programmatically create variations of a driving scenario that was built using the Driving Scenario Designer app. Programmatically creating variations of a scenario enables you to rapidly test your driving algorithms under multiple conditions.

To create programmatic variations of a driving scenario, follow these steps:

1. Interactively build a driving scenario by using the Driving Scenario Designer app.
2. Export a MATLAB® function that generates the MATLAB code that is equivalent to this scenario.
3. In the MATLAB Editor, modify the exported function to create variations of the original scenario.
4. Call the function to generate a `drivingScenario` object that represents the scenario.
5. Import the scenario object into the app to simulate the modified scenario or generate additional scenarios. Alternatively, to simulate the modified scenario in Simulink®, import the object into a Simulink model by using a Scenario Reader block.

The diagram shows a visual representation of this workflow.



Build Scenario in App

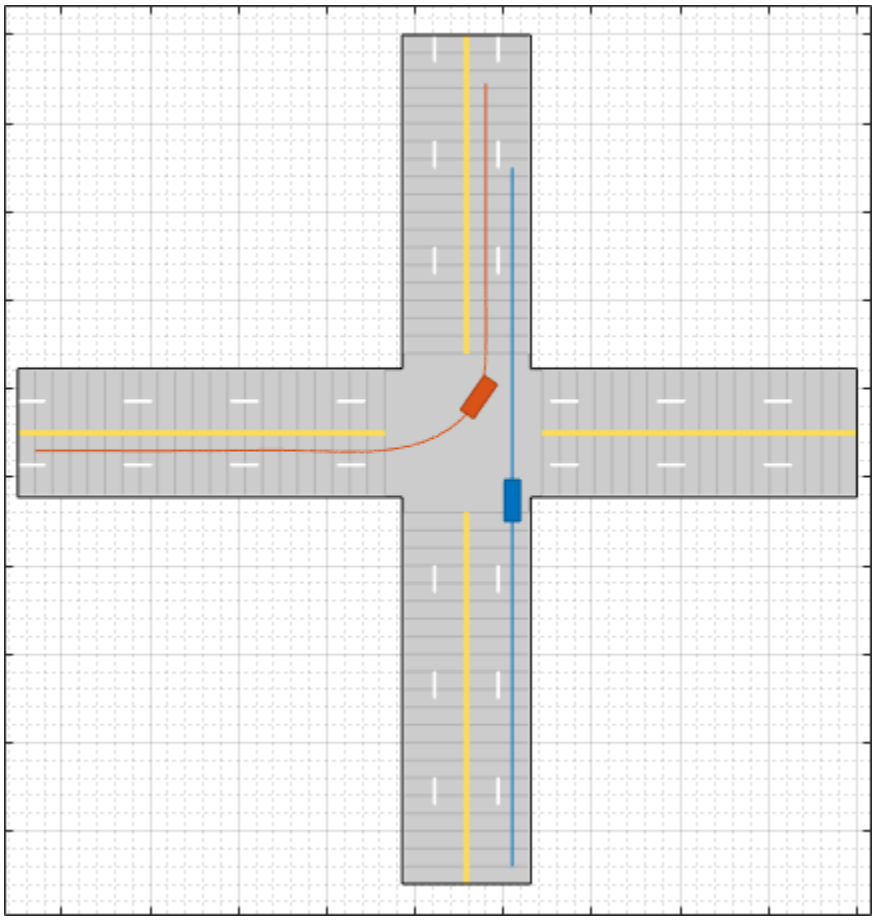
Use the Driving Scenario Designer to interactively build a driving scenario on which to test your algorithms. For more details on building scenarios, see “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2.

This example uses a driving scenario that is based on one of the prebuilt scenarios that you can load from the Driving Scenario Designer app.

Open the scenario file in the app.

```
drivingScenarioDesigner('LeftTurnScenarioNoSensors.mat')
```

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



For simplicity, this scenario does not include sensors mounted on the ego vehicle.

Export MATLAB Function of Scenario

After you view and simulate the scenario, you can export the scenario to the MATLAB command line. From the Driving Scenario Designer app toolstrip, select **Export > Export MATLAB Function**. The exported function contains the MATLAB code used to produce the scenario created in the app. Open the exported function.

```
open LeftTurnScenarioNoSensors.m
```

```
function [scenario, egoVehicle] = LeftTurnScenarioNoSensors()
% createDrivingScenario Returns the drivingScenario defined in the Designer
```

Calling this function returns these aspects of the driving scenario.

- `scenario` — Roads and actors of the scenarios, returned as a `drivingScenario` object.
- `egoVehicle` — Ego vehicle defined in the scenario, returned as a `Vehicle` object. For details, see the `vehicle` function.

If your scenario contains sensors, then the returned function includes additional code for generating the sensors. If you simulated the scenario containing those sensors, then the function can also generate the detections produced by those sensors.

Modify Function to Create Scenario Variations

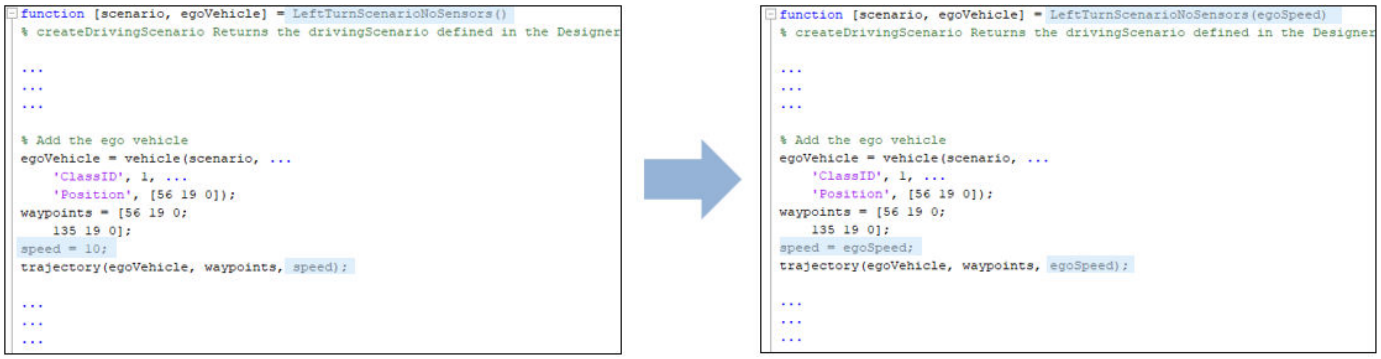
By modifying the code in the exported MATLAB function, you can generate multiple variations of a single scenario. One common variation is to test the ego vehicle at different speeds. In the exported MATLAB function, the speed of the ego vehicle is set to a constant value of 10 meters per second (`speed = 10`). To generate varying ego vehicle speeds, you can convert the speed variable into an input argument to the function. Open the script containing a modified version of the exported function.

```
open LeftTurnScenarioNoSensorsModified.m
```

In this modified function:

- `egoSpeed` is included as an input argument.
- `speed`, the constant variable, is deleted.
- To compute the ego vehicle trajectory, `egoSpeed` is used instead of `speed`.

This figure shows these script modifications.



To produce additional variations, consider:

- Modifying the road and lane parameters to view the effect on lane detections
- Modifying the trajectory or starting positions of the vehicles
- Modifying the dimensions of the vehicles

Call Function to Generate Programmatic Scenarios

Using the modified function, generate a variation of the scenario in which the ego vehicle travels at a constant speed of 20 meters per second.

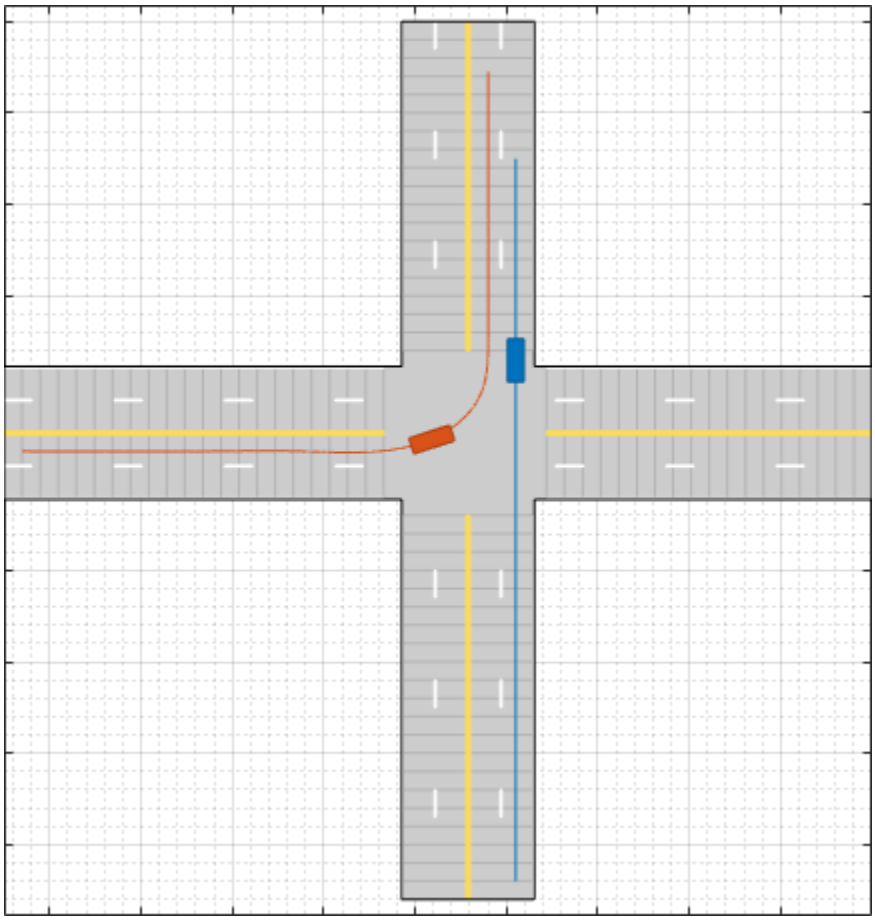
```
scenario = LeftTurnScenarioNoSensorsModified(20) % m/s
```

```
scenario =
  drivingScenario with properties:

        SampleTime: 0.0400
          StopTime: Inf
    SimulationTime: 0
         IsRunning: 1
            Actors: [1x2 driving.scenario.Vehicle]
```

Import Modified Scenario into App

To import the modified scenario with the modified vehicle into the app, use the `drivingScenarioDesigner` function. Specify the `drivingScenario` object as an input argument.

```
drivingScenarioDesigner(scenario)
```



Previously, the other vehicle passed through the intersection first. Now, with the speed of the ego vehicle increased from 10 to 20 meters per second, the ego vehicle passes through the intersection first.

When working with `drivingScenario` objects in the app, keep these points in mind.

- To try out different ego vehicle speeds, call the exported function again, and then import the new `drivingScenario` object using the `drivingScenarioDesigner` function. The app does not include a menu option for importing these objects.
- If your scenario includes sensors, you can reopen both the scenario and sensors by using this syntax: `drivingScenarioDesigner(scenario,sensors)`.
- If you make significant changes to the dimensions of an actor, be sure that the `ClassID` property of the actor corresponds to a **Class ID** value specified in the app. For example, in the app, cars have a **Class ID** of 1 and trucks have a **Class ID** of 2. If you programmatically change a car to have the dimensions of a truck, update the `ClassID` property of that vehicle from `1` (car) to `2` (truck).

Import Modified Scenario into Simulink

To import the modified scenario into a Simulink model, use a Scenario Reader block. This block reads the roads and actors from either a scenario file saved from the app or a `drivingScenario` variable saved to the MATLAB workspace or the model workspace. Add a Scenario Reader block to your model and set these parameters.

1. Set **Source of driving scenario** to `From workspace`.
2. Set **MATLAB or model workspace variable name** to the name of the `drivingScenario` variable in your workspace.

When working with `drivingScenario` objects in Simulink, keep these points in mind.

- When **Source of ego vehicle** is set to `Scenario`, the model uses the ego vehicle defined in your `drivingScenario` object. The block determines which actor is the ego vehicle based on the specified `ActorID` property of the actor. This actor must be a `Vehicle` object (see `vehicle`). To change the designated ego vehicle, update the **Ego vehicle ActorID** parameter.
- When connecting the output actor poses to sensor blocks, updates these blocks to obtain the actor profiles directly from the `drivingScenario` object. By default, the blocks use the same set of actor profiles for all actors, where the profiles are defined on the **Actor Profiles** tab of the blocks. To obtain the profiles from the object, on the **Actor Profiles** tab of each sensor block, set the **Select method to specify actor profiles** parameter to `MATLAB expression`. Then, set the **MATLAB expression for actor profiles** parameter to call the `actorProfiles` function on the object. For example: `actorProfiles(scenario)`.

See Also

Apps
Driving Scenario Designer

Blocks
Lidar Point Cloud Generator | Radar Detection Generator | Scenario Reader | Vision Detection Generator

Functions
`actorProfiles` | `vehicle`

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

More About

- “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2
- “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-19
- “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-41
- “Create Driving Scenario Programmatically” on page 7-423

Generate Sensor Detection Blocks Using Driving Scenario Designer

This example shows how to update the radar and camera sensors of a Simulink® model by using the Driving Scenario Designer app. The Driving Scenario Designer app enables you to generate multiple sensor configurations quickly and interactively. You can then use these generated sensor configurations in your existing Simulink models to test your driving algorithms more thoroughly.

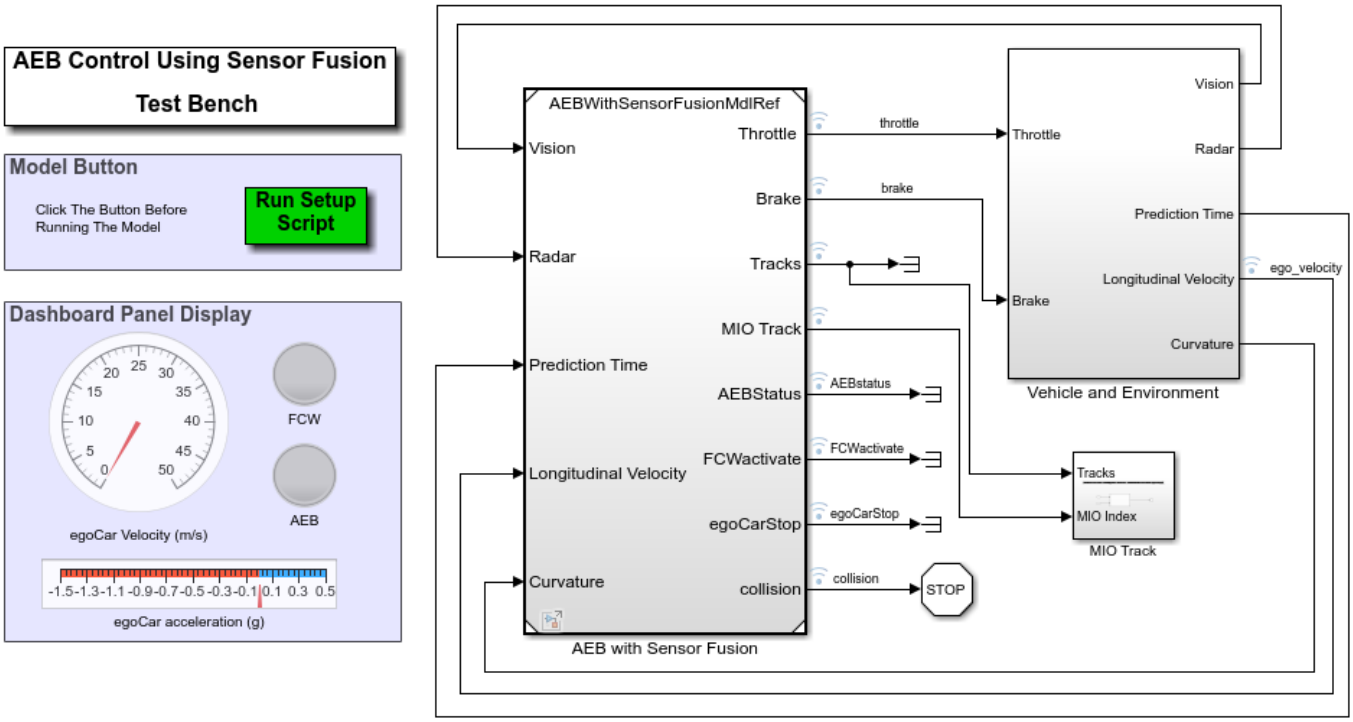
Before beginning this example, add the example file folder to the MATLAB® search path.

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

Inspect and Simulate Model

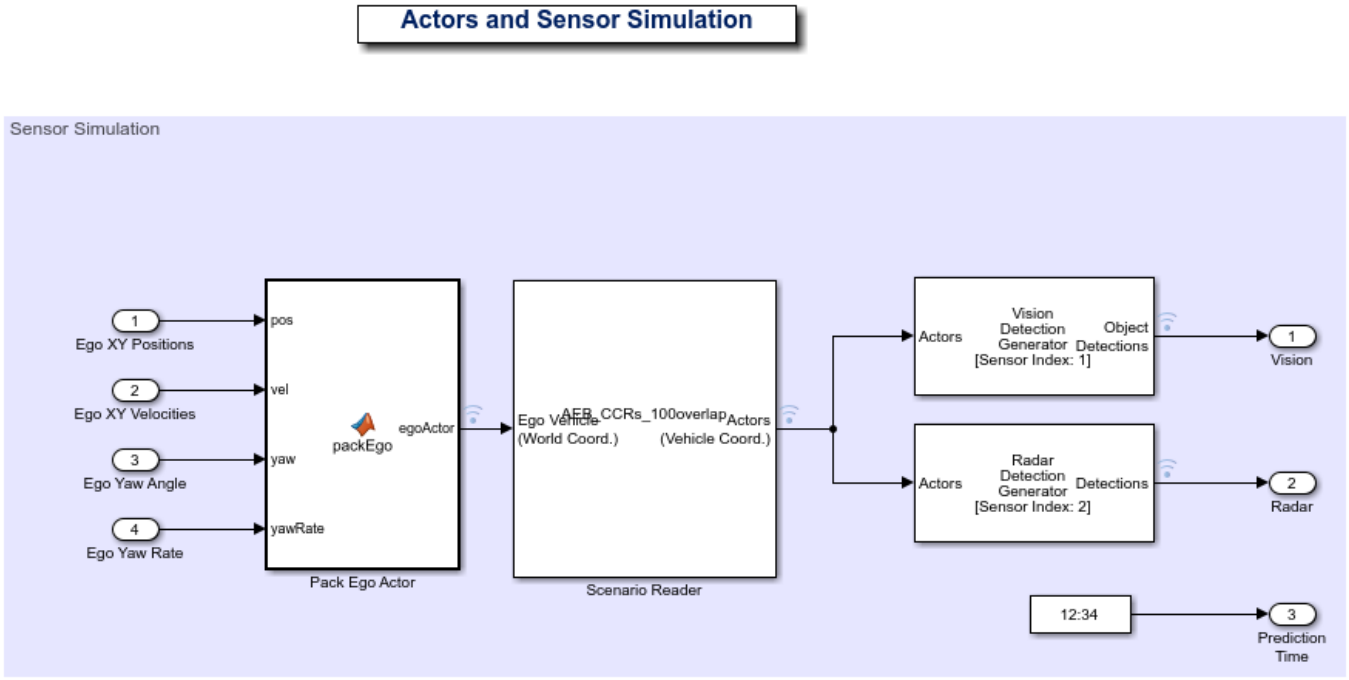
The model used in this example implements an autonomous emergency braking (AEB) sensor fusion algorithm. For more details about this model, see the "Autonomous Emergency Braking with Sensor Fusion" on page 7-214 example. Open this model.

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



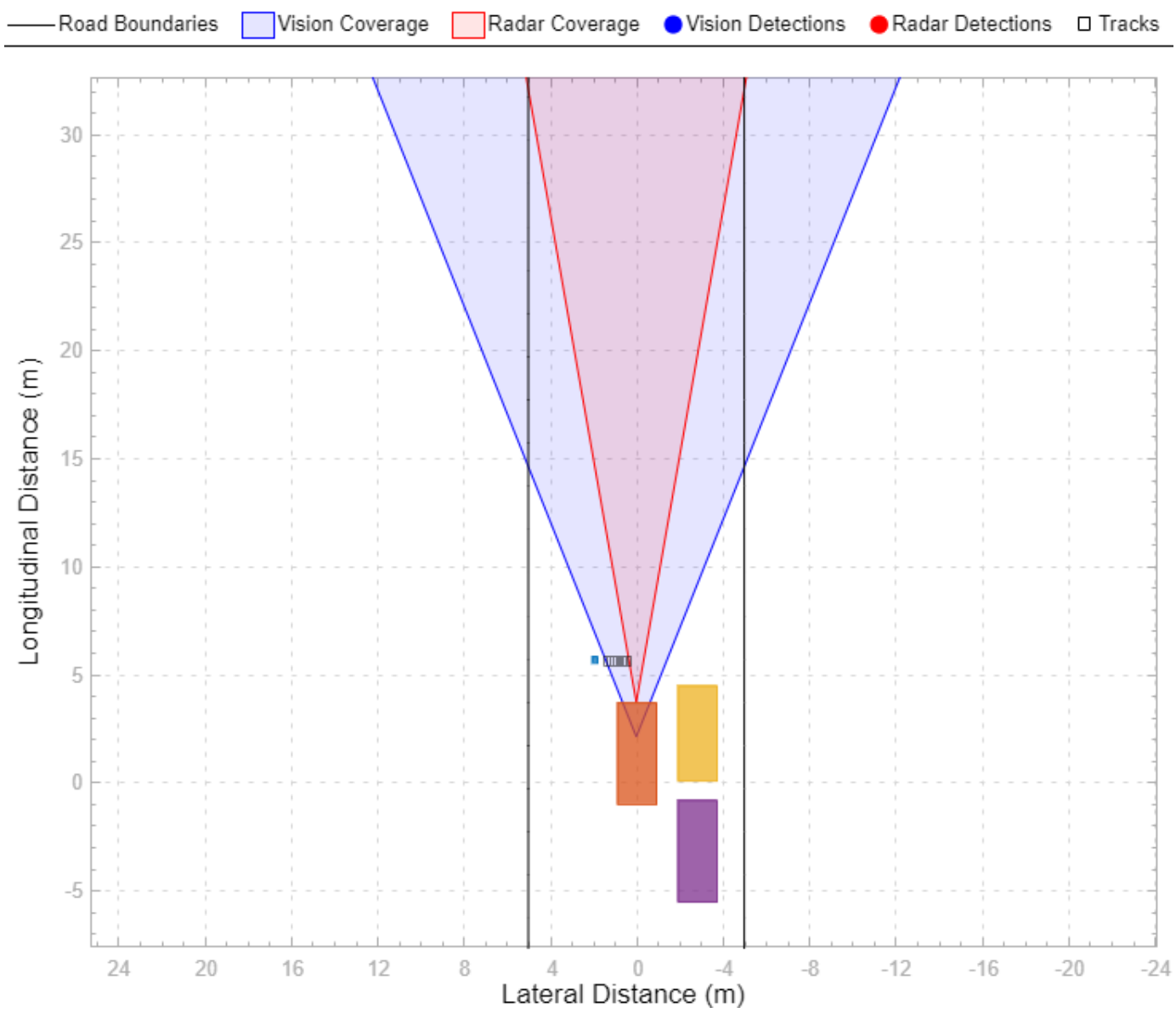
The driving scenario and sensor detection generators used to test the algorithm are located in the **Vehicle Environment > Actors and Sensor Simulation** subsystem. Open this subsystem.

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



A Scenario Reader block reads the actors and roads from the specified Driving Scenario Designer file. The block outputs the non-ego actors. These actors are then passed to Radar Detection Generator and Vision Detection Generator sensor blocks. During simulation, these blocks generate detections of the non-ego actors.

Simulate and visualize the scenario on the Bird's-Eye Scope. On the model toolstrip, under **Review Results**, click **Bird's-Eye Scope**. In the scope, click **Find Signals**, and then click **Run** to run the simulation. In this scenario, the AEB model causes the ego vehicle to brake in time to avoid a collision with a pedestrian child who is crossing the street.



During this example, you replace the existing sensors in this model with new sensors created in the Driving Scenario Designer app.

Load Scenario in App

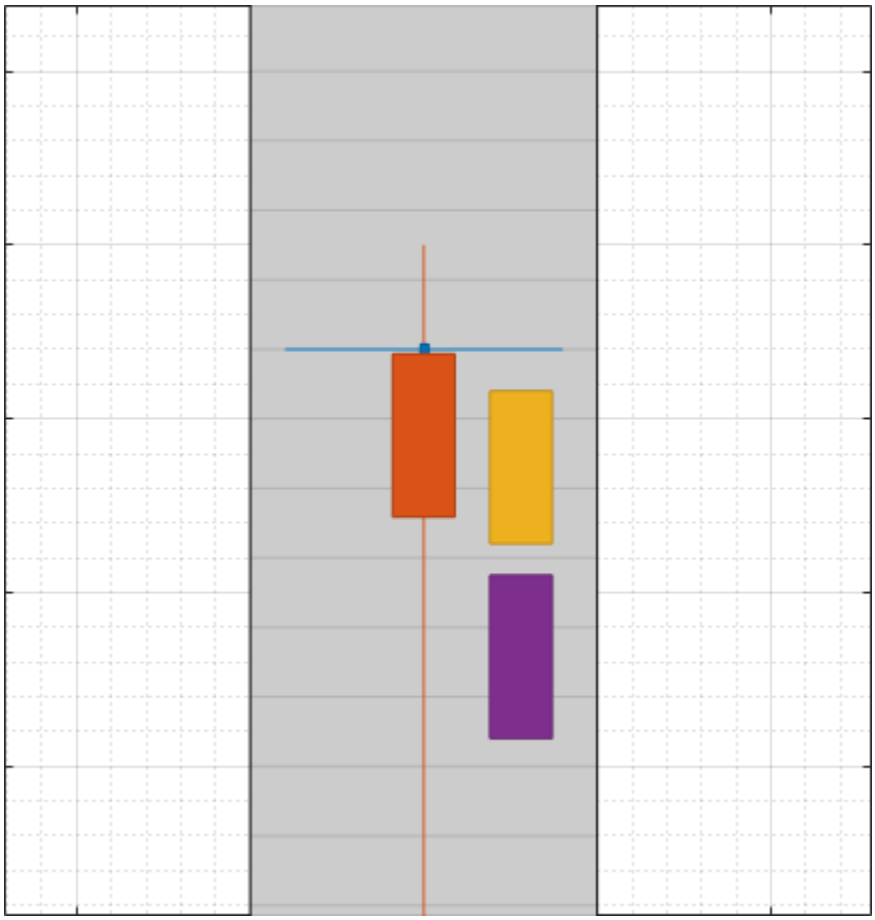
The model uses a driving scenario that is based on one of the prebuilt Euro NCAP test protocol scenarios. You can load these scenarios from the Driving Scenario Designer app. For more details on these scenarios, see “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-41.

Load the scenario file into the app.

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

To simulate the scenario in the app, click **Run**. In the app simulation, unlike in the model simulation, the ego vehicle collides with the pedestrian. The app uses a predefined ego vehicle trajectory,

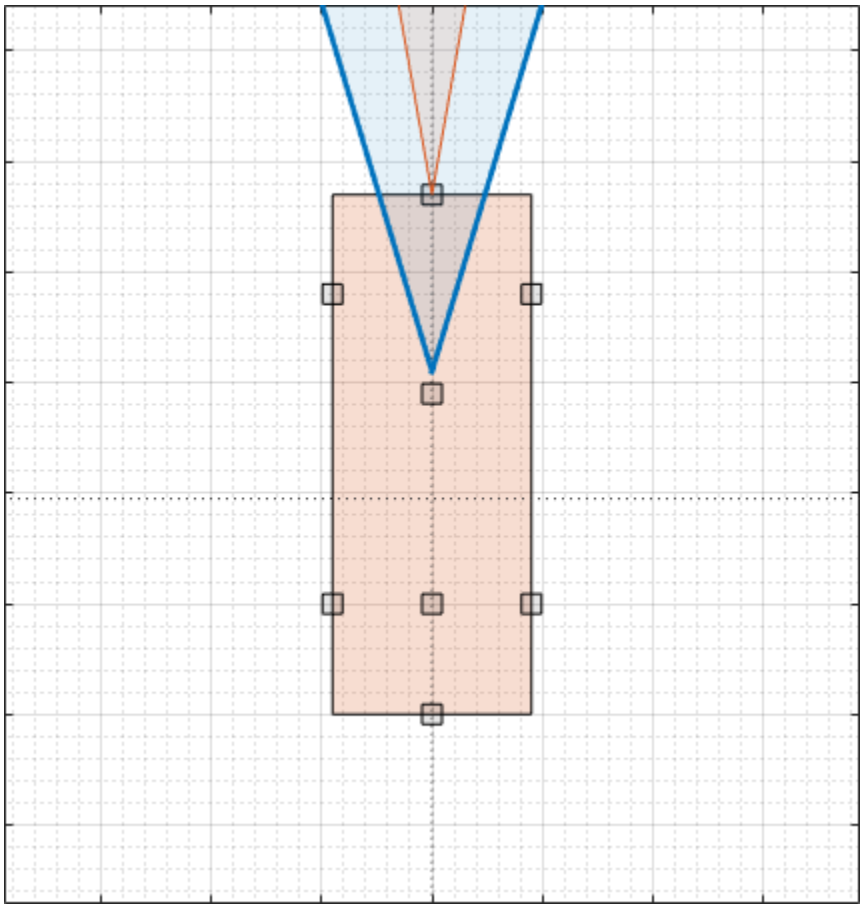
whereas the model uses the AEB algorithm to control the trajectory and cause the ego vehicle to brake.



Load Sensors

The loaded scenario file contains only the roads and actors in the scenario. A separate file contains the sensors. To load these sensors into the scenario, on the app toolstrip, select **Open > Sensors**. Open the `AEBSensor.mat` file located in the example folder. Alternatively, from your MATLAB root folder, navigate to and open this file: *`matlabroot`*`/examples/driving/AEBSensors.mat`.

A radar sensor is mounted to the front bumper of the ego vehicle. A camera sensor is mounted to the front window of the ego vehicle.



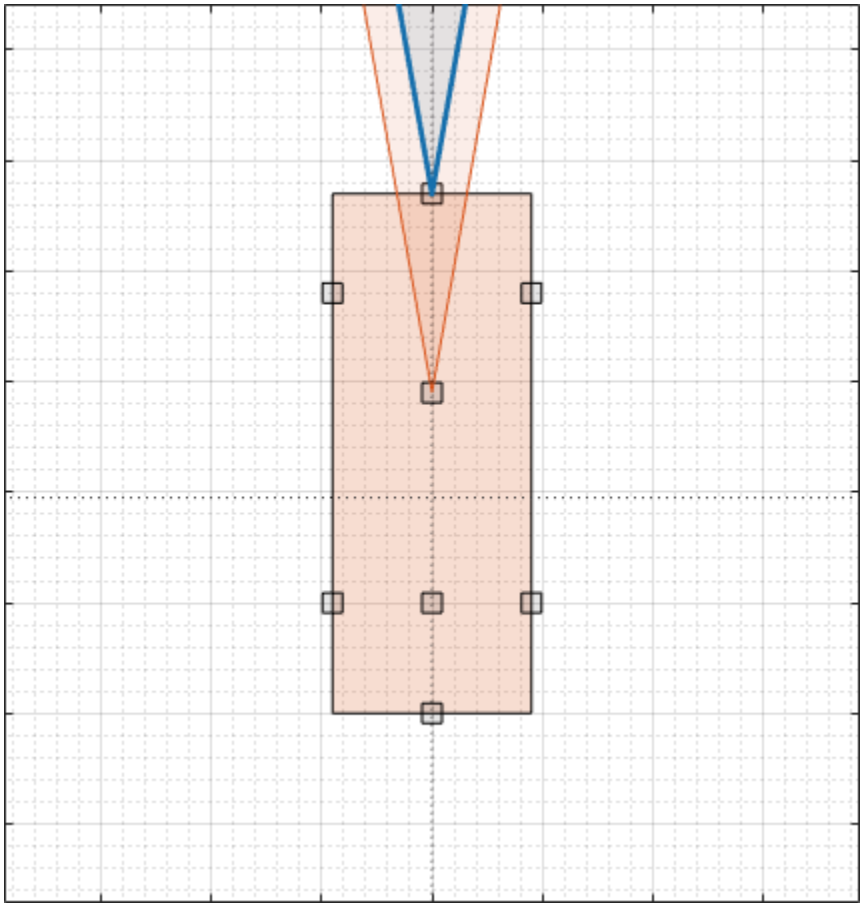
Update Sensors

Update the radar and camera sensors by changing their locations on the ego vehicles.

1. On the **Sensor Canvas**, click and drag the radar sensor to the predefined `Front Window` location.
2. Click and drag the camera sensor to the predefined `Front Bumper` location. At this predefined location, the app updates the camera from a short-range sensor to a long-range sensor.
3. Optionally, in the left pane, on the **Sensors** tab, try modifying the parameters of the camera and radar sensors. For example, you can change the detection probability or the accuracy and noise settings.
4. Save a copy of this new scenario and sensor configuration to a writeable location.

For more details on working with sensors in the app, see “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2.

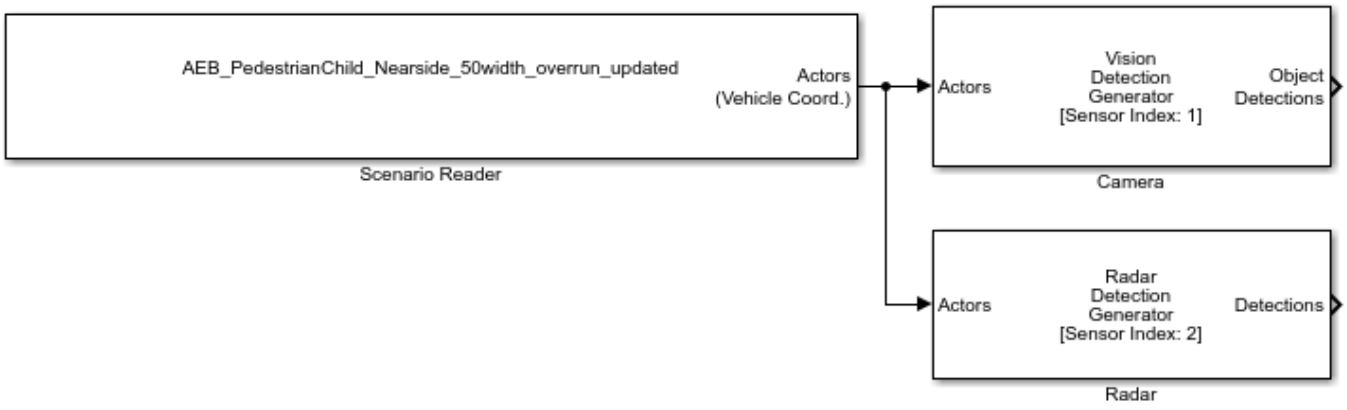
This image shows a sample updated sensor configuration.



Export Scenario and Sensors to Simulink

To generate Simulink blocks for the scenario and its sensors, on the app toolstrip, select **Export > Export Simulink Model**. This model shows sample blocks that were exported from the app.

```
open_system('AEBGeneratedScenarioAndSensors')
```



If you made no changes to the roads and actors in the scenario, then the Scenario Reader block reads the same road and actor data that was used in the AEB model. The Radar Detection Generator and Vision Detection Generator blocks model the radar and camera that you created in the app.

Copy Exported Scenario and Sensors into Existing Model

Replace the scenario and sensors in the AEB model with the newly generated scenario and sensors. Even if you did not modify the roads and actors and read data from the same scenario file, replacing the existing Scenario Reader block is still a best practice. Using this generated block keeps the bus names for scenario and sensors consistent as data passes between them.

To get started, in the AEB model, reopen the **Vehicle Environment > Actors and Sensor Simulation** subsystem.

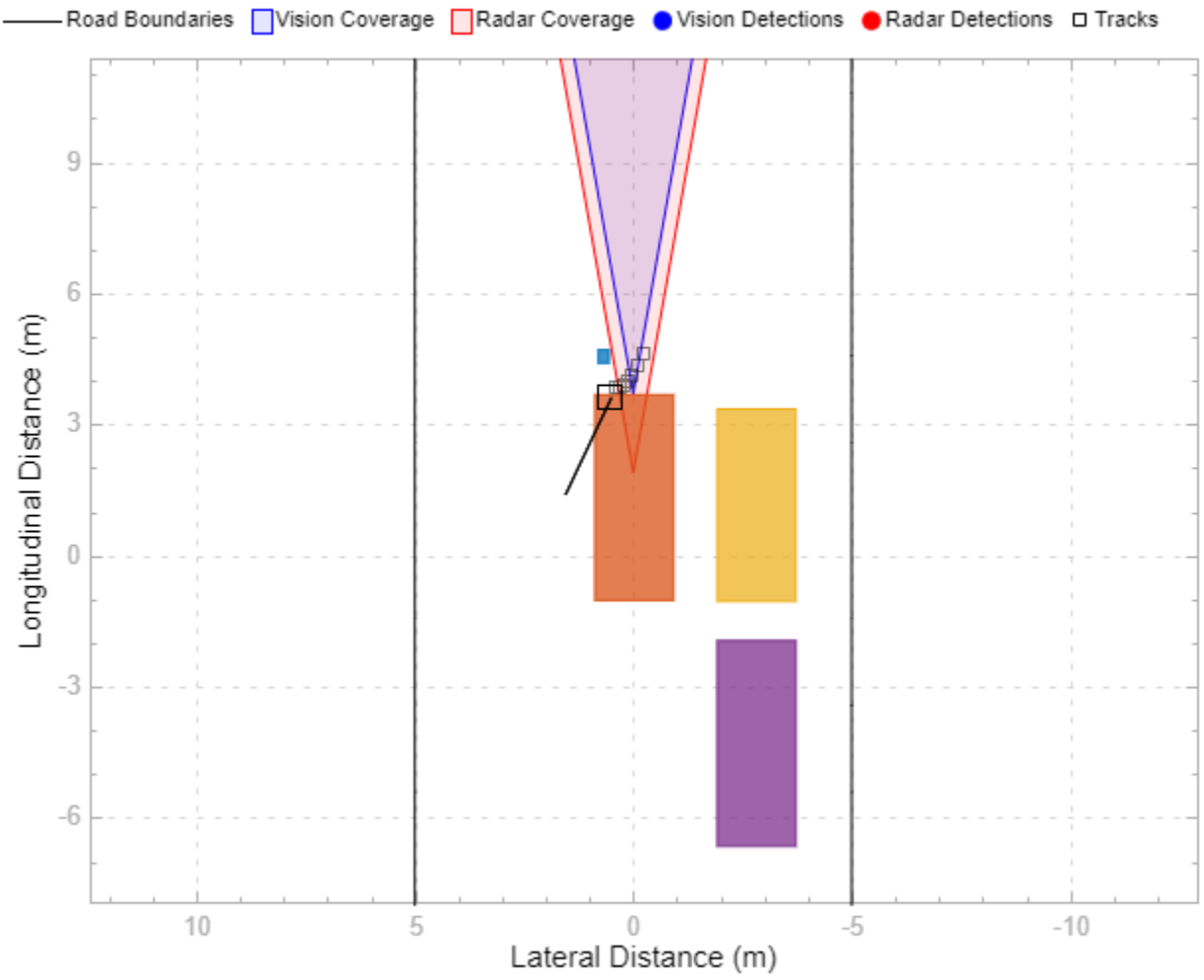
```
open_system('AEBTestBenchExample/Vehicle and Environment/Actors and Sensor Simulation')
```

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

1. Delete the existing Scenario Reader, Radar Detection Generator, and Vision Detection Generator blocks. Do not delete the signal lines that are input to the Scenario Reader block or output from the sensor blocks. Alternatively, disconnect these blocks without deleting them, and comment them out of the model. Using this option, you can compare the existing blocks to the new one and revert back if needed. Select each block. Then, on the **Block** tab, select **Comment Out**.
2. Copy the blocks from the generated model into the AEB model.
3. Open the copied-in Scenario Reader block and set the **Source of ego vehicle** parameter to `Input port`. Click **OK**. The AEB model defines the ego vehicle in the Pack Ego Actor block, which you connect to the **Ego Vehicle** port of the Scenario Reader block.
4. Connect the existing signal lines to the copied-in blocks. To clean up the layout of the model, on the **Format** tab of the model, select **Auto Arrange**.
5. Verify that the updated subsystem block diagram resembles the pre-existing block diagram. Then, save the model, or save a copy of the model to a writeable location.

Simulate Updated Model

To visualize the updated scenario simulation, reopen the Bird's-Eye Scope, click **Find Signals**, and then click **Run**. With this updated sensor configuration, the ego vehicle does not brake in time.



To try different sensor configurations, reload the scenario and sensors in the app, export new scenarios and sensors, and copy them into the AEB model.

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

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

See Also

Apps

Bird's-Eye Scope | Driving Scenario Designer

Blocks

Lidar Point Cloud Generator | Radar Detection Generator | Scenario Reader | Vision Detection Generator

More About

- “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2

- “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-41
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-121
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-127
- “Autonomous Emergency Braking with Sensor Fusion” on page 7-214

Test Open-Loop ADAS Algorithm Using Driving Scenario

This example shows how to test an open-loop ADAS (advanced driver assistance system) algorithm in Simulink®. In an open-loop ADAS algorithm, the ego vehicle behavior is predefined and does not change as the scenario advances during simulation.

To test the algorithm, you use a driving scenario that was saved from the Driving Scenario Designer app. In this example, you read in a scenario by using a Scenario Reader block, and then visualize the scenario and sensor detections on the Bird's-Eye Scope.

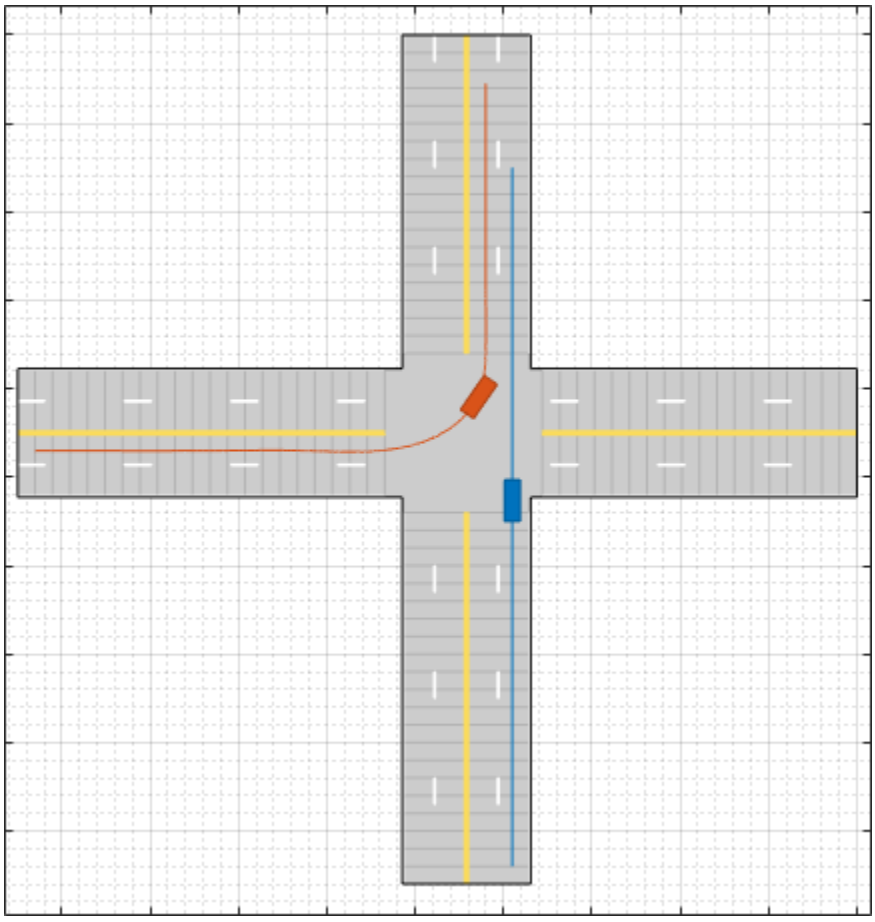
Inspect Driving Scenario

This example uses a driving scenario that is based on one of the prebuilt scenarios that you access through the Driving Scenario Designer app. For more details on these scenarios, see “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-19.

Open the scenario file in the app.

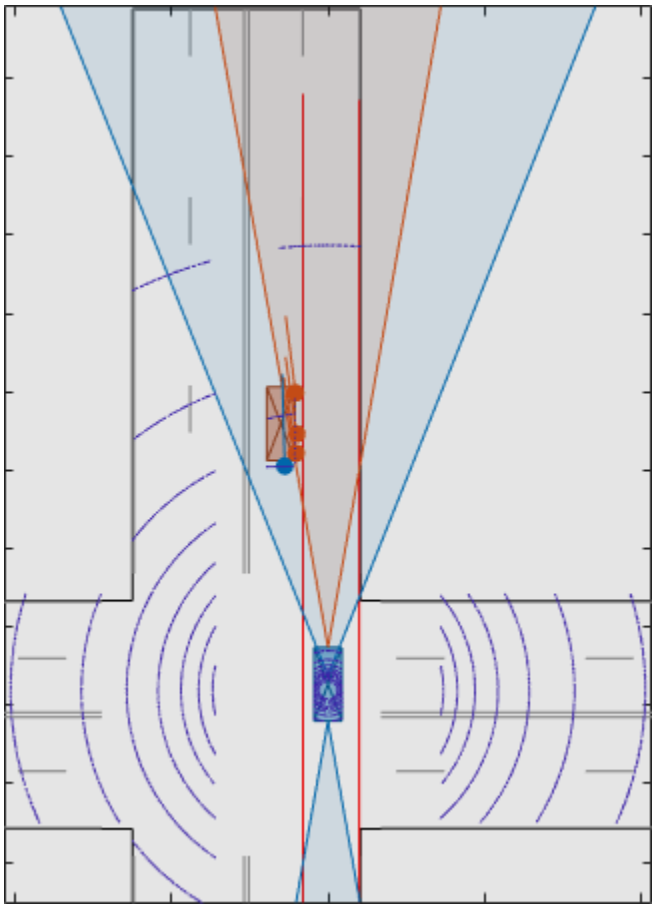
```
drivingScenarioDesigner('LeftTurnScenario.mat')
```

To simulate the scenario, click **Run**. In this scenario, 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.



The ego vehicle has these sensors:

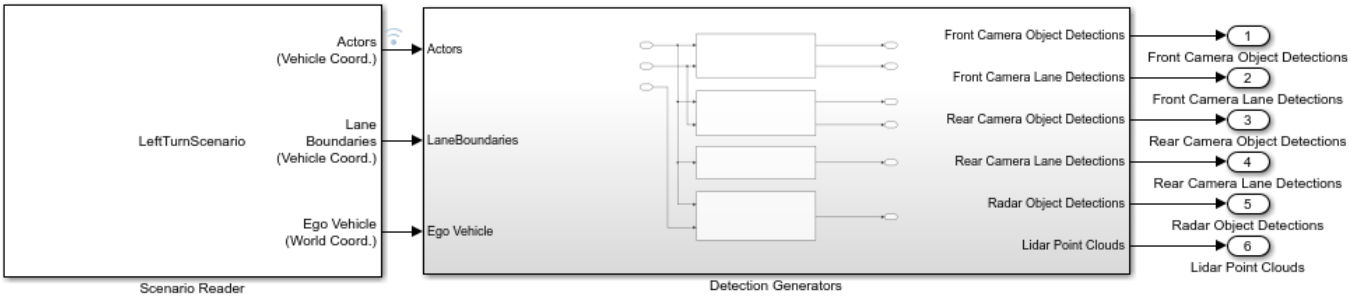
- A front-facing radar for generating object detections
- A front-facing camera and rear-facing camera for generating object and lane boundary detections
- A lidar on the center of its roof for generating point cloud data of the scenario



Inspect Model

The model in this example was generated from the app by selecting **Export > Export Simulink Model**. In the model, a Scenario Reader block reads the actors and roads from the scenario file and outputs the non-ego actors and lane boundaries. Open the model.

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

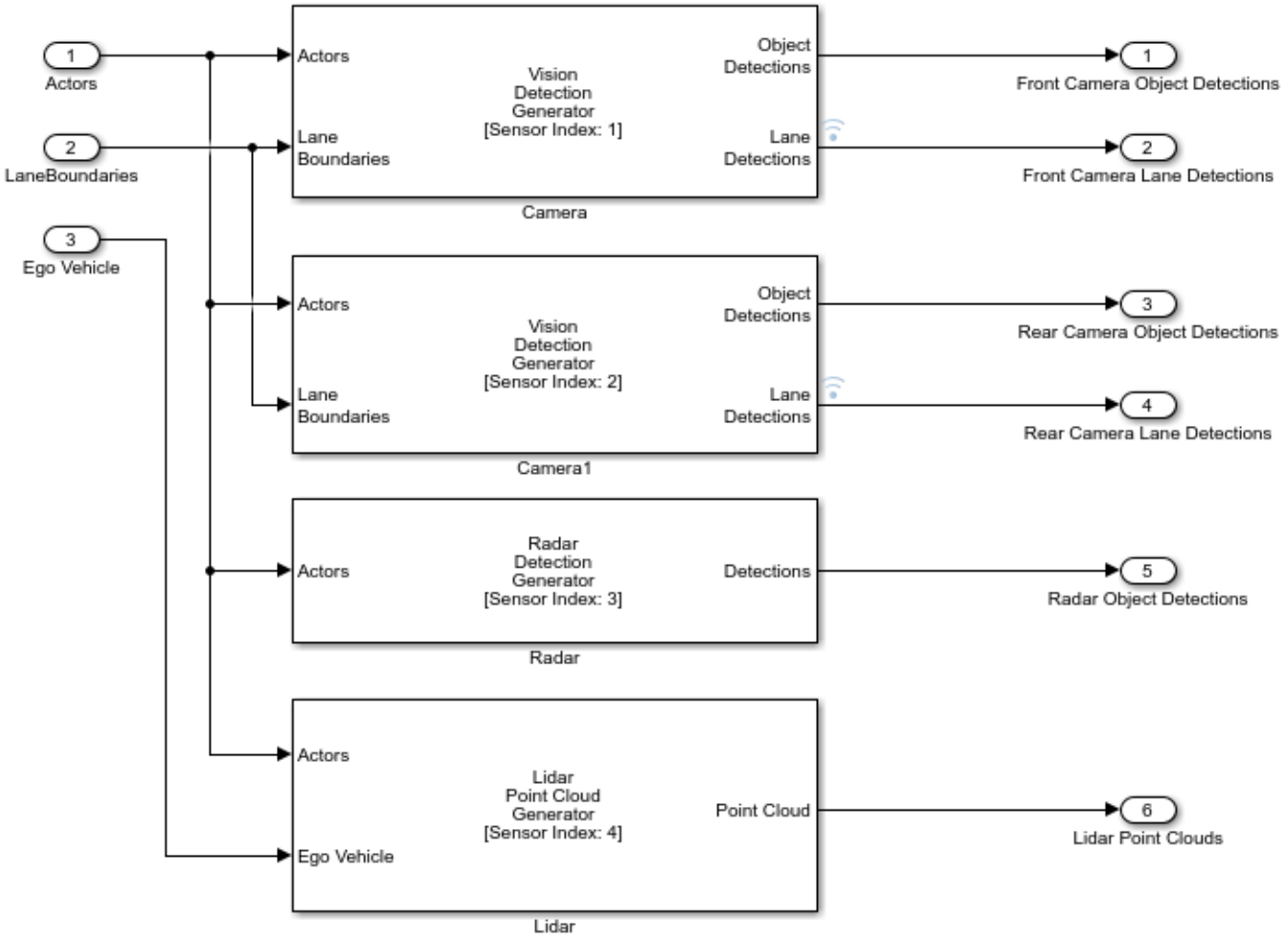


In the Scenario Reader block, the **Driving Scenario Designer file name** parameter specifies the name of the scenario file. You can specify a scenario file that is on the MATLAB search path, such as the scenario file used in this example, or the full path to a scenario file. Alternatively, you can specify a `drivingScenario` object by setting **Source of driving scenario** to `From workspace`, and then setting **MATLAB or model workspace variable name** to the name of a valid `drivingScenario` object workspace variable.

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

The actors, lane boundaries, and ego vehicle pose are passed to a subsystem containing the sensor blocks. Open the subsystem.

```
open_system('OpenLoopWithScenarios/Detection Generators')
```



The Radar Detection Generator, Vision Detection Generator, and Lidar Point Cloud Generator blocks produce synthetic detections from the scenario. You can fuse this sensor data to generate tracks, such as in the open-loop example “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” on page 7-205.

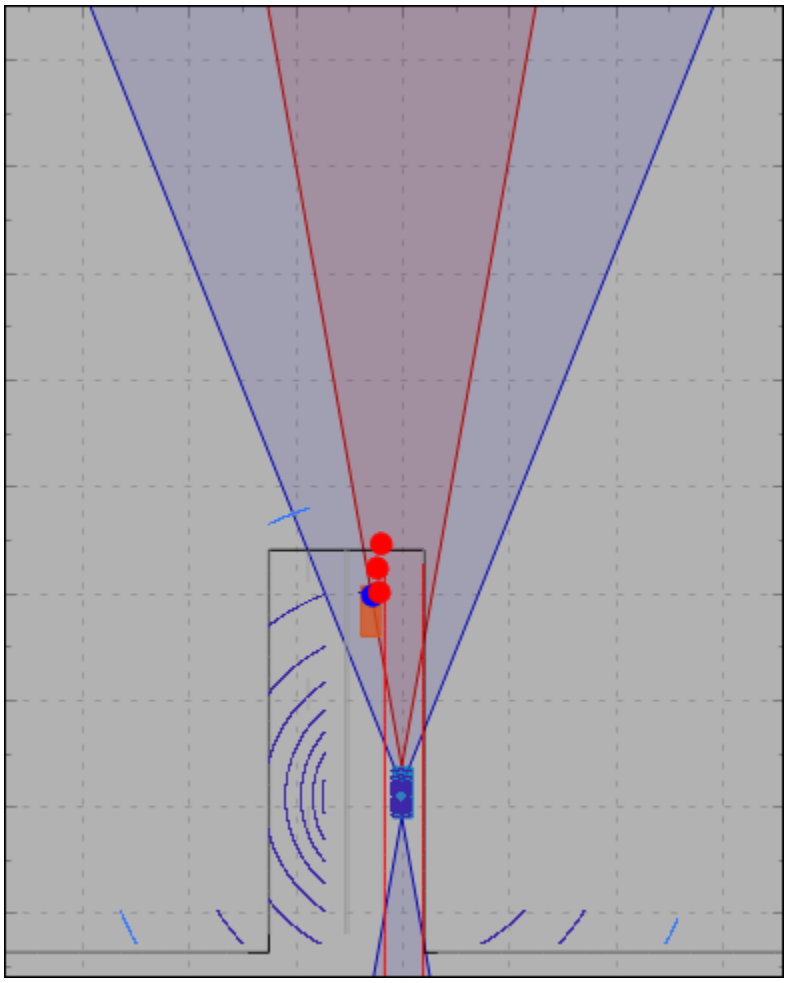
The outputs of the sensor blocks in this model are in vehicle coordinates, where:

- 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.

Because this model is open loop, the ego vehicle behavior does not change as the simulation advances. Therefore, the **Source of ego vehicle** parameter is set to `Scenario`, and the block reads the predefined ego vehicle pose and trajectory from the scenario file. For vehicle controllers and other closed-loop models, set the **Source of ego vehicle** parameter to `Input port`. With this option, you specify an ego vehicle that is defined in the model as an input to the Scenario Reader block. For an example, see “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-127.

Visualize Simulation

To visualize the scenario and sensor detections, use the Bird's-Eye Scope. On the Simulink toolstrip, under **Review Results**, click **Bird's-Eye Scope**. Then, in the scope, click **Find Signals** and run the simulation.



Update Simulation Settings

This model uses the default simulation stop time of 10 seconds. 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, on 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. On the Simulink toolstrip, select **Run > Simulation Pacing**. Select the **Enable pacing to slow down simulation** check box and decrease the simulation time to slightly less than 1

second per wall-clock second, such as 0.8 seconds. Then, rerun the simulation in the Bird's-Eye Scope.

See Also

Apps
Bird's-Eye Scope | Driving Scenario Designer

Blocks
Lidar Point Cloud Generator | Radar Detection Generator | Scenario Reader | Vision Detection Generator

More About

- “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” on page 7-205
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-127
- “Create Driving Scenario Variations Programmatically” on page 5-107
- “Generate Sensor Detection Blocks Using Driving Scenario Designer” on page 5-112

Test Closed-Loop ADAS Algorithm Using Driving Scenario

This model shows how to test a closed-loop ADAS (advanced driver assistance system) algorithm in Simulink®. In a closed-loop ADAS algorithm, the ego vehicle is controlled by changes in its scenario environment as the simulation advances.

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

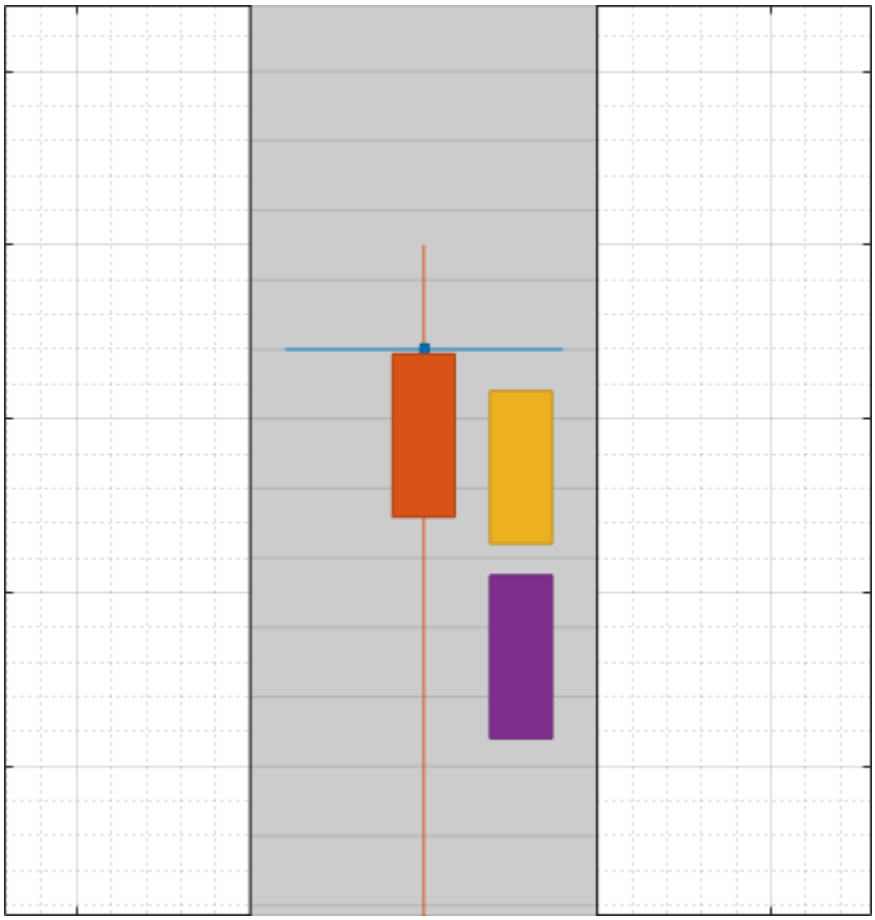
Inspect Driving Scenario

This example uses a driving scenario that is based on one of the prebuilt Euro NCAP test protocol scenarios that you can access through the Driving Scenario Designer app. For more details on these scenarios, see “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-41.

Open the scenario file in the app.

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

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

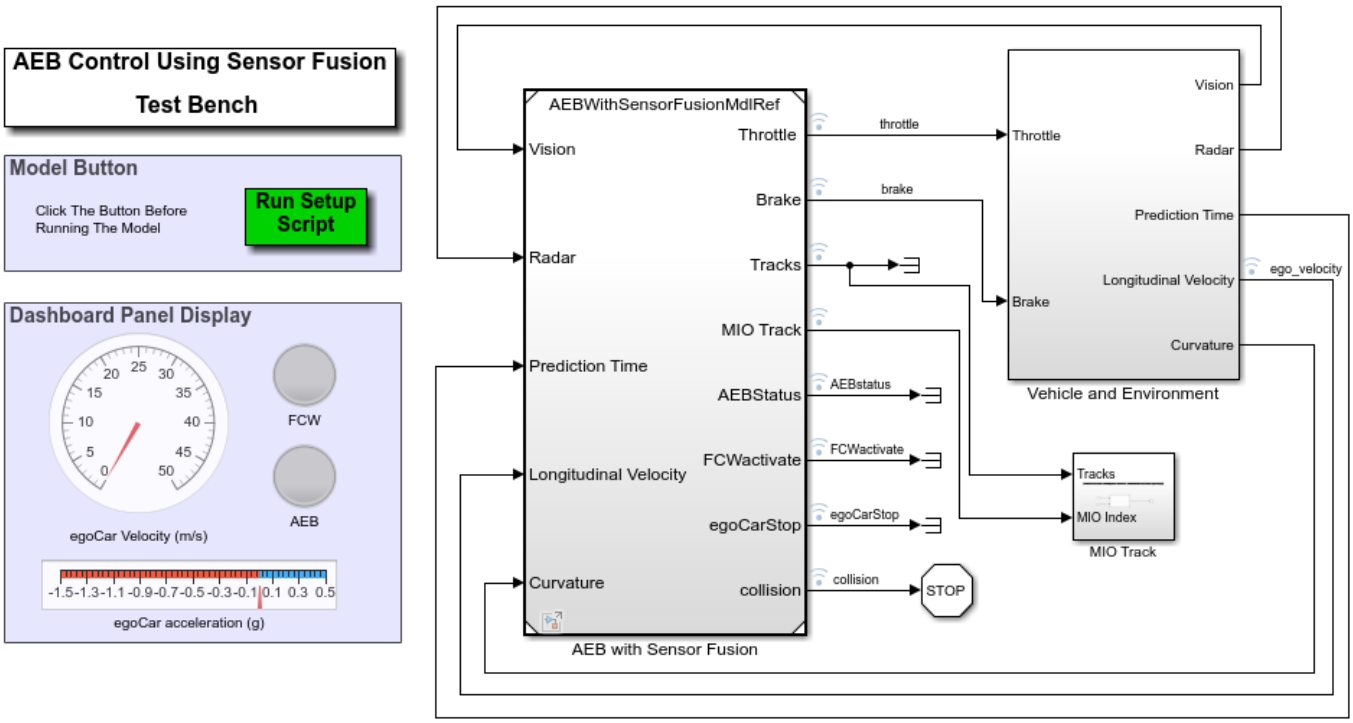


In the model used in this example, you use an AEB sensor fusion algorithm to detect the pedestrian child and test whether the ego vehicle brakes in time to avoid a collision.

Inspect Model

The model implements the AEB algorithm described in the "Autonomous Emergency Braking with Sensor Fusion" on page 7-214 example. Open the model.

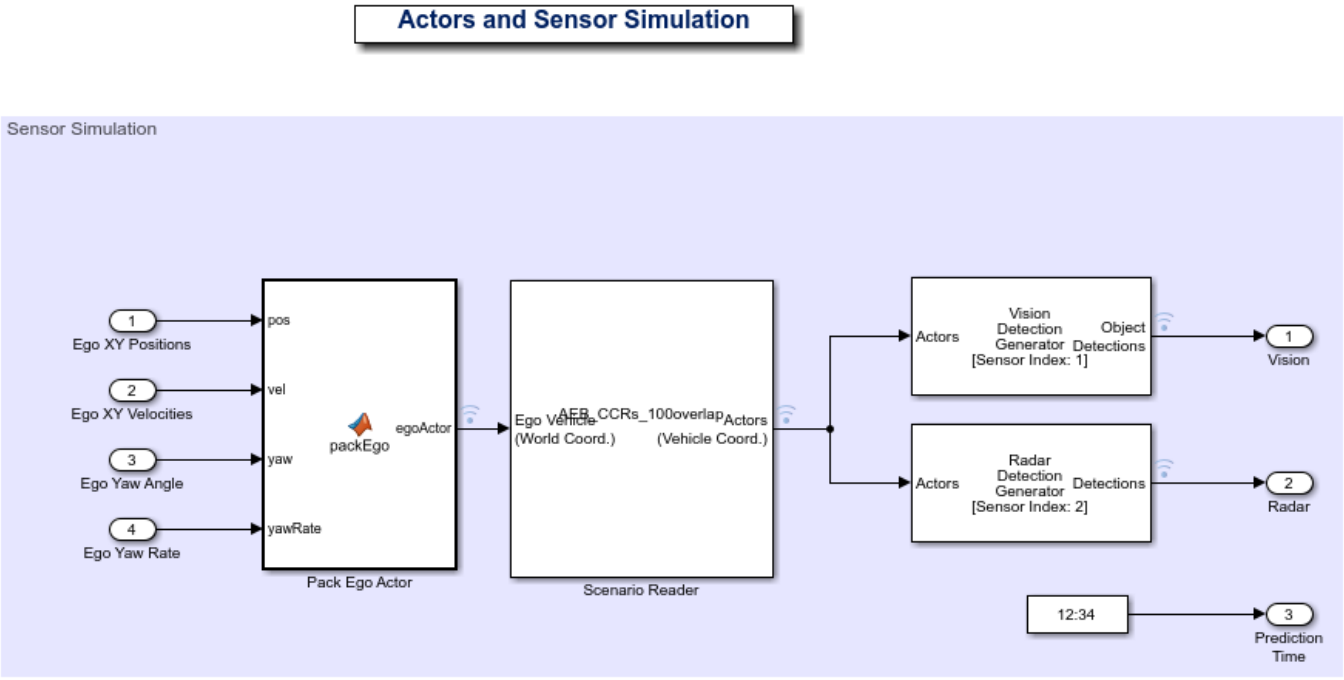
```
open_system('AEBTestBenchExample')
```



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

The Scenario Reader block is located in the **Vehicle Environment > Actors and Sensor Simulation** subsystem. Open this subsystem.

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



In the Scenario Reader block, the **Driving Scenario Designer file name** parameter specifies the name of the scenario file. You can specify a scenario file that is on the MATLAB search path, such as the scenario file used in this example, or the full path to a scenario file. Alternatively, you can specify a `drivingScenario` object by setting **Source of driving scenario** to `From workspace` and then setting **MATLAB or model workspace variable name** to the name of a valid `drivingScenario` object workspace variable. In closed-loop simulations, specifying the `drivingScenario` object is useful because it enables you finer control over specifying the initial position of the ego vehicle in your model.

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

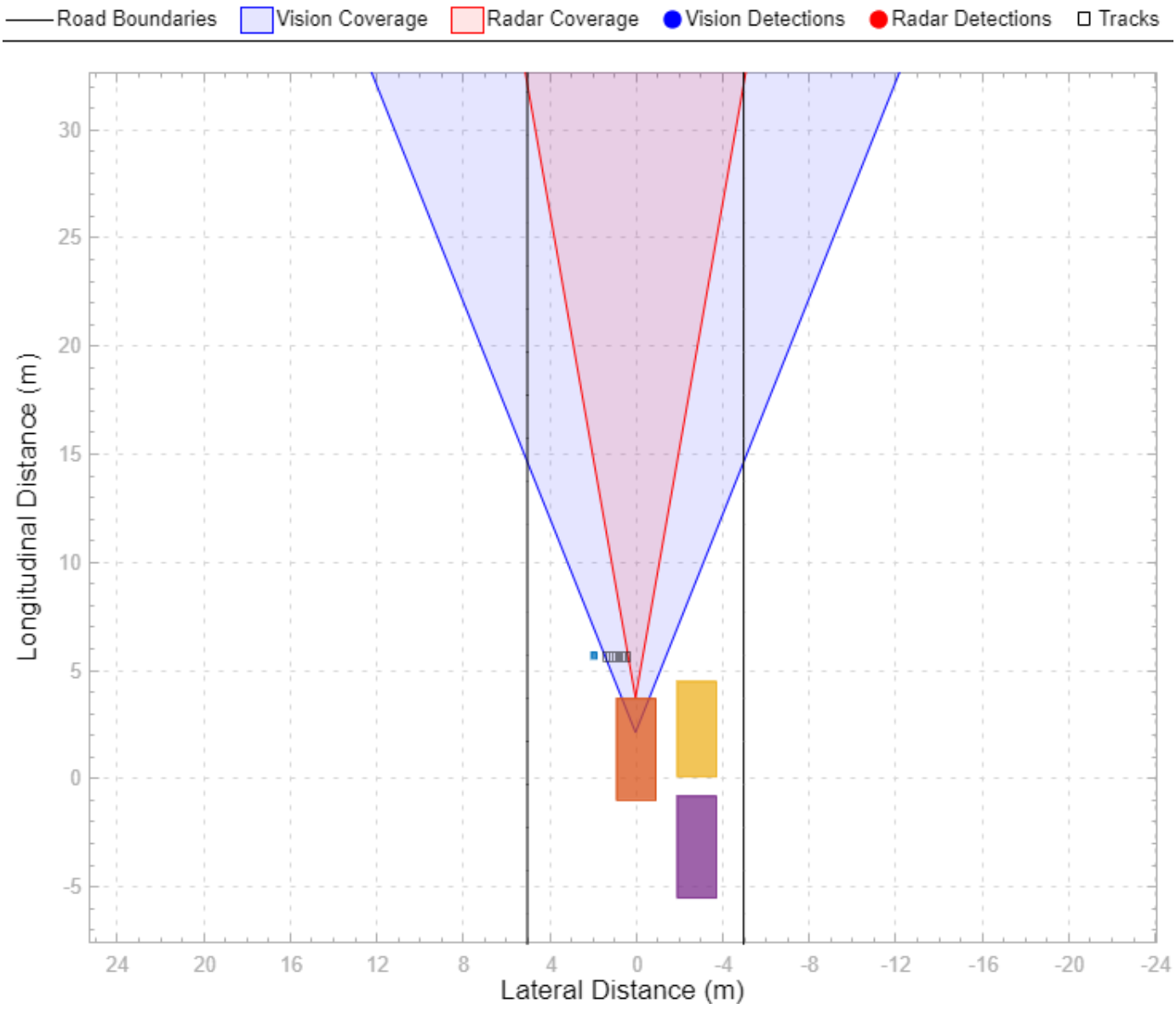
The actor poses are output in vehicle coordinates, where:

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

Although this scenario includes a predefined ego vehicle, the Scenario Reader block is configured to ignore this ego vehicle definition. Instead, the ego vehicle is defined in the model and specified as an input to the Scenario Reader block (the **Source of ego vehicle** parameter is set to `Input port`). As the simulation advances, the AEB algorithm determines the pose and trajectory of the ego vehicle. If you are developing an open-loop algorithm, where the ego vehicle is predefined in the driving scenario, set the **Source of ego vehicle** parameter to `Scenario`. For an example, see “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-121.

Visualize Simulation

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



See Also

Apps
Bird's-Eye Scope | Driving Scenario Designer

Blocks
Lidar Point Cloud Generator | Radar Detection Generator | Scenario Reader | Vision Detection Generator

More About

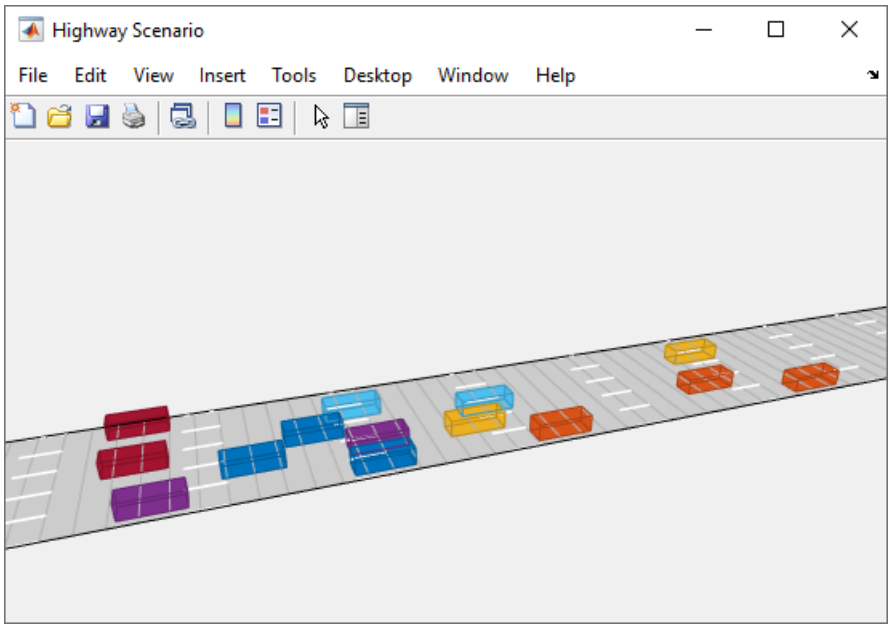
- “Autonomous Emergency Braking with Sensor Fusion” on page 7-214
- “Lateral Control Tutorial” on page 7-589
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-121
- “Create Driving Scenario Variations Programmatically” on page 5-107
- “Generate Sensor Detection Blocks Using Driving Scenario Designer” on page 5-112

Automate Control of Intelligent Vehicles by Using Stateflow Charts

This example shows how to model a highway scenario with intelligent vehicles that are controlled by the same decision logic. Each vehicle determines when to speed up, slow down, or change lanes based on the logic defined by a standalone Stateflow® chart. Because the driving conditions (including the relative position and speed of nearby vehicles) differ from vehicle to vehicle, separate chart objects in MATLAB® control the individual vehicles on the highway.

Open Driving Scenario

To start the example, run the script `sf_driver_demo.m`. The script displays a 3-D animation of a long highway and several vehicles. The view focuses on a single vehicle and its surroundings. As this vehicle moves along the highway, the standalone Stateflow chart `sf_driver` shows the decision logic that determines its actions.



Starting from a random position, each vehicle attempts to travel at a target speed. Because the target speeds are chosen at random, the vehicles can obstruct one another. In this situation, a vehicle will try to change lanes and resume its target speed.

The class file `HighwayScenario` defines a `drivingScenario` object that represents the 3-D environment that contains the highway and the vehicles on it. To control the motion of the vehicles, the `drivingScenario` object creates an array of Stateflow chart objects. Each chart object controls a different vehicle in the simulation.

Execute Decision Logic for Vehicles

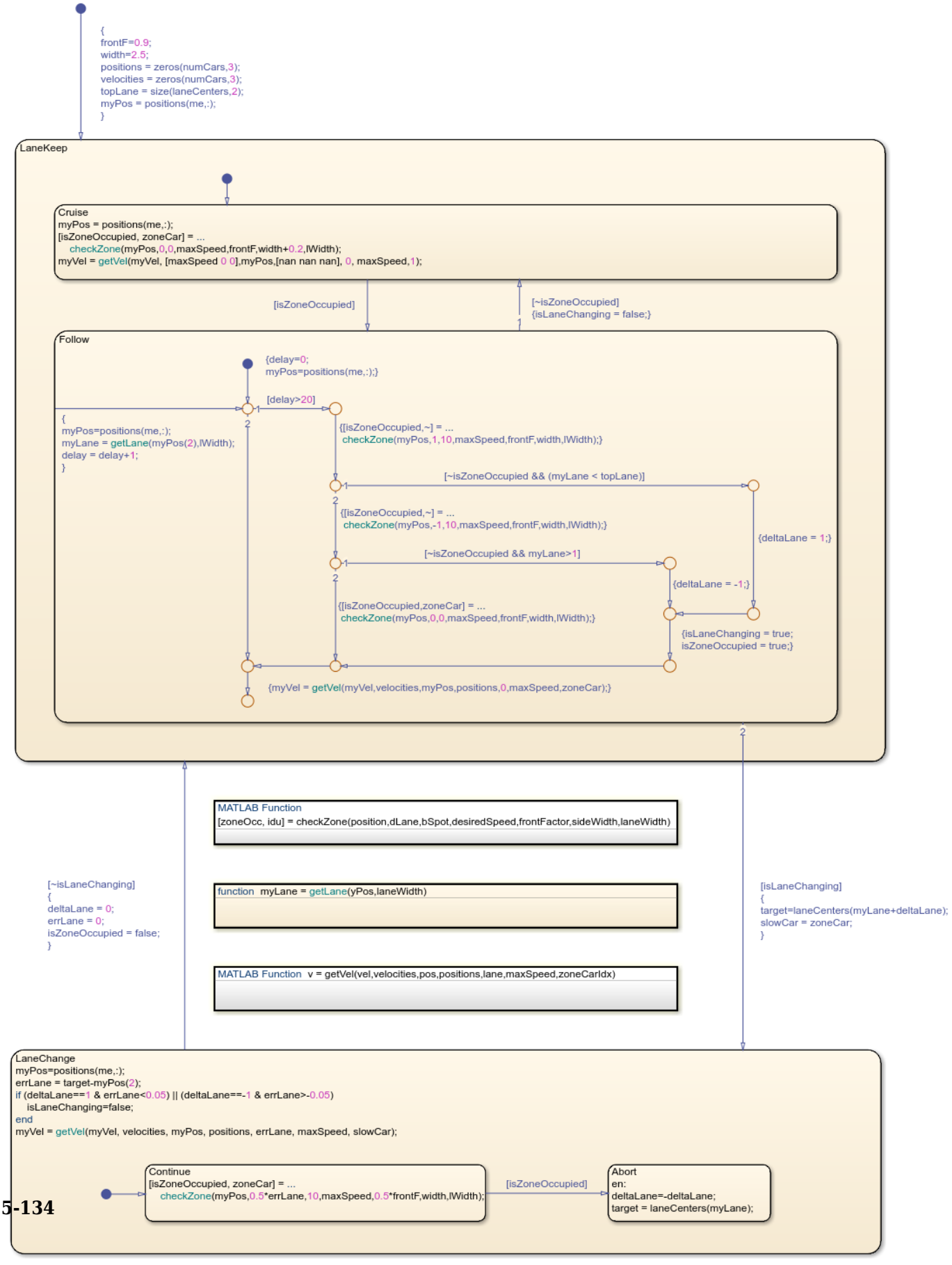
The Stateflow chart `sf_driver` consists of two top-level states, `LaneKeep` and `LaneChange`.

When the `LaneKeep` state is active, the corresponding vehicle stays in its lane of traffic. In this state, there are two possible substates:

- `Cruise` is active when the zone directly in front of the vehicle is empty and the vehicle can travel at its target speed.
- `Follow` becomes active when the zone directly in front of the vehicle is occupied and its target speed is faster than the speed of the vehicle in front. In this case, the vehicle is forced to slow down and attempt to change lanes.

When the `LaneChange` state is active, the corresponding vehicle attempts to change lanes. In this state, there are two possible substates:

- `Continue` is active when the zone next to the vehicle is empty and the vehicle can change lanes safely.
- `Abort` becomes active when the zone next to the vehicle is occupied. In this case, the vehicle is forced to remain in its lane.


{
frontF=0.9;
width=2.5;
positions = zeros(numCars,3);
velocities = zeros(numCars,3);
topLane = size(laneCenters,2);
myPos = positions(me,:);
}
LaneKeep
Cruise
myPos = positions(me,:);
[isZoneOccupied, zoneCar] = ...
checkZone(myPos,0,0,maxSpeed,frontF,width+0.2,lWidth);
myVel = getVel(myVel, [maxSpeed 0 0],myPos,[nan nan nan], 0, maxSpeed,1);
[isZoneOccupied]
[~isZoneOccupied]
{isLaneChanging = false;}
Follow
{delay=0;
myPos=positions(me,:);}
[delay>20]
{
myPos=positions(me,:);
myLane = getLane(myPos(2),lWidth);
delay = delay+1;
}
{[isZoneOccupied,~] = ...
checkZone(myPos,1,10,maxSpeed,frontF,width,lWidth);}
[~isZoneOccupied && (myLane < topLane)]
{[isZoneOccupied,~] = ...
checkZone(myPos,-1,10,maxSpeed,frontF,width,lWidth);}
{deltaLane = 1;}
[~isZoneOccupied && myLane>1]
{deltaLane = -1;}
{[isZoneOccupied,zoneCar] = ...
checkZone(myPos,0,0,maxSpeed,frontF,width,lWidth);}
{isLaneChanging = true;
isZoneOccupied = true;}
{myVel = getVel(myVel,velocities,myPos,positions,0,maxSpeed,zoneCar);}
MATLAB Function
[zoneOcc, idu] = checkZone(position,dLane,bSpot,desiredSpeed,frontFactor,sideWidth,laneWidth)
function myLane = getLane(yPos,laneWidth)
MATLAB Function v = getVel(vel,velocities,pos,positions,lane,maxSpeed,zoneCarIdx)
[~isLaneChanging]
{
deltaLane = 0;
errLane = 0;
isZoneOccupied = false;
}
[isLaneChanging]
{
target=laneCenters(myLane+deltaLane);
slowCar = zoneCar;
}
LaneChange
myPos=positions(me,:);
errLane = target-myPos(2);
if (deltaLane==1 & errLane<0.05) || (deltaLane==-1 & errLane>-0.05)
isLaneChanging=false;
end
myVel = getVel(myVel, velocities, myPos, positions, errLane, maxSpeed, slowCar);
Continue
[isZoneOccupied, zoneCar] = ...
checkZone(myPos,0.5*errLane,10,maxSpeed,0.5*frontF,width,lWidth);
[isZoneOccupied]
Abort
en:
deltaLane=-deltaLane;
target = laneCenters(myLane);

The transitions between the states `LaneKeep` and `LaneChange` are guarded by the value of `isLaneChanging`. In the `LaneKeep` state, the chart sets this local data to `true` when the substate `Follow` is active and there is enough room beside the vehicle to change lanes. In the `LaneChange` state, the chart sets this local data to `false` when the vehicle finishes changing lanes.

See Also

`drivingScenario`

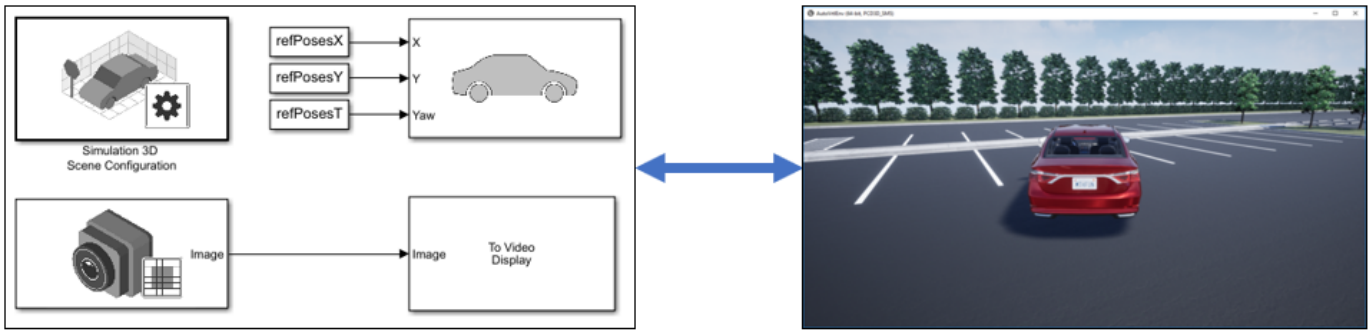
More About

- “Create Driving Scenario Programmatically” on page 7-423
- “Create Actor and Vehicle Trajectories Programmatically” on page 7-442
- “Define Road Layouts Programmatically” on page 7-453

3D Simulation – User's Guide

Unreal Engine Simulation for Automated Driving

Automated Driving Toolbox provides a co-simulation framework that models driving algorithms in Simulink and visualizes their performance in a virtual simulation environment. This environment uses the Unreal Engine from Epic Games.



Simulink blocks related to the simulation environment can be found in the **Automated Driving Toolbox > Simulation 3D** block library. These blocks provide the ability to:

- Configure prebuilt scenes in the simulation environment.
- Place and move vehicles within these scenes.
- Set up camera, radar, and lidar sensors on the vehicles.
- Simulate sensor outputs based on the environment around the vehicle.
- Obtain ground truth data for semantic segmentation and depth information.

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

For more details on the simulation environment, see “How Unreal Engine Simulation for Automated Driving Works” on page 6-8.

Unreal Engine Simulation Blocks

To access the **Automated Driving Toolbox > Simulation 3D** library, at the MATLAB command prompt, enter `drivingsim3d`.

Scenes

To configure a model to co-simulate with the simulation environment, add a Simulation 3D Scene Configuration block to the model. Using this block, you can choose from a set of prebuilt scenes where you can test and visualize your driving algorithms. The following image is from the Virtual Mcity scene.

The toolbox includes these scenes.

Scene	Description
Straight Road	Straight road segment
Curved Road	Curved, looped road
Parking Lot	Empty parking lot
Double Lane Change	Straight road with barrels and traffic signs that are set up for executing a double lane change maneuver
Open Surface	Flat, black pavement surface with no road objects
US City Block	City block with intersections, barriers, and traffic lights
US Highway	Highway with cones, barriers, traffic lights, and traffic signs
Large Parking Lot	Parking lot with parked cars, cones, curbs, and traffic signs
Virtual Mcity	City environment that represents the University of Michigan proving grounds (see Mcity Test Facility); includes cones, barriers, an animal, traffic lights, and traffic signs

If you have the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, then you can modify these scenes or create new ones. For more details, see “Customize Unreal Engine Scenes for Automated Driving” on page 6-44.

Vehicles

To define a virtual vehicle in a scene, add a Simulation 3D Vehicle with Ground Following block to your model. Using this block, you can control the movement of the vehicle by supplying the X, Y, and yaw values that define its position and orientation at each time step. The vehicle automatically moves along the ground.

You can also specify the color and type of vehicle. The toolbox includes these vehicle types:

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

Sensors

You can define virtual sensors and attach them at various positions on the vehicles. The toolbox includes these sensor modeling and configuration blocks.

Block	Description
Simulation 3D Camera	Camera model with lens. Includes parameters for image size, focal length, distortion, and skew.
Simulation 3D Fisheye Camera	Fisheye camera that can be described using the Scaramuzza camera model. Includes parameters for distortion center, image size, and mapping coefficients.
Simulation 3D Lidar	Scanning lidar sensor model. Includes parameters for detection range, resolution, and fields of view.
Simulation 3D Probabilistic Radar	Probabilistic radar model that returns a list of detections. Includes parameters for radar accuracy, radar bias, detection probability, and detection reporting. It does not simulate radar at an electromagnetic wave propagation level.
Simulation 3D Probabilistic Radar Configuration	Configures radar signatures for all actors detected by the Simulation 3D Probabilistic Radar blocks in a model.
Simulation 3D Vision Detection Generator	Camera model that returns a list of object and lane boundary detections. Includes parameters for modeling detection accuracy, measurement noise, and camera intrinsics.

For more details on choosing a sensor, see “Choose a Sensor for Unreal Engine Simulation” on page 6-16.

Algorithm Testing and Visualization

Automated Driving Toolbox simulation blocks provide the tools for testing and visualizing path planning, vehicle control, and perception algorithms.

Path Planning and Vehicle Control

You can use the Unreal Engine simulation environment to visualize the motion of a vehicle in a prebuilt scene. This environment provides you with a way to analyze the performance of path planning and vehicle control algorithms. After designing these algorithms in Simulink, you can use the `drivingsim3d` library to visualize vehicle motion in one of the prebuilt scenes.

For an example of path planning and vehicle control algorithm visualization, see “Visualize Automated Parking Valet Using Unreal Engine Simulation” on page 7-635.

Perception

Automated Driving Toolbox provides several blocks for detailed camera, radar, and lidar sensor modeling. By mounting these sensors on vehicles within the virtual environment, you can generate synthetic sensor data or sensor detections to test the performance of your sensor models against perception algorithms. For an example of generating radar detections, see “Simulate Vision and Radar Sensors in Unreal Engine Environment” on page 7-647.

You can also output and visualize ground truth data to validate depth estimation algorithms and train semantic segmentation networks. For an example, see “Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation” on page 6-31.

Closed-Loop Systems

After you design and test a perception system within the simulation environment, you can then use it to drive a control system that actually steers a vehicle. In this case, rather than manually set up a trajectory, the vehicle uses the perception system to drive itself. By combining perception and control into a closed-loop system in the 3D simulation environment, you can develop and test more complex algorithms, such as lane keeping assist and adaptive cruise control.

For an example of a closed-loop system in the Unreal Engine environment, see “Highway Lane Following” on page 7-653.

See Also

More About

- “Unreal Engine Simulation Environment Requirements and Limitations” on page 6-6
- “Simulate Simple Driving Scenario and Sensor in Unreal Engine Environment” on page 6-22
- “Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox” on page 6-10
- “Customize Unreal Engine Scenes for Automated Driving” on page 6-44

Unreal Engine Simulation Environment Requirements and Limitations

Automated Driving Toolbox provides an interface to a simulation environment that is visualized using the Unreal Engine from Epic Games. Version 4.23 of this visualization engine comes installed with Automated Driving Toolbox. When simulating in this environment, keep these requirements and limitations in mind.

Software Requirements

- Windows® 64-bit platform
- Visual Studio® 2017
- Microsoft® DirectX® — If this software is not already installed on your machine and you try to simulate in the 3D environment, Automated Driving Toolbox prompts you to install it. Once you install the software, you must restart the simulation.

In you are customizing scenes, verify that your Unreal Engine project is compatible with the Unreal Engine version supported by your MATLAB release.

MATLAB Release	Unreal Engine Version
R2019b	4.19
R2020a	4.23
R2020b	4.23

Note Mac and Linux® platforms are not supported.

Minimum Hardware Requirements

- Graphics card (GPU) — Virtual reality-ready with 8 GB of on-board RAM
- Processor (CPU) — 2.60 GHz
- Memory (RAM) — 12 GB

Limitations

The Unreal Engine simulation environment blocks do not support:

- Code generation
- Model reference
- Multiple instances of the Simulation 3D Scene Configuration block
- Multiple Unreal Engine instances in the same MATLAB session.
- Rapid accelerator mode

In addition, when using these blocks in a closed-loop simulation, all Unreal Engine simulation environment blocks must be in the same subsystem.

See Also

Simulation 3D Scene Configuration

More About

- “Unreal Engine Simulation for Automated Driving” on page 6-2
- “How Unreal Engine Simulation for Automated Driving Works” on page 6-8

External Websites

- Unreal Engine

How Unreal Engine Simulation for Automated Driving Works

Automated Driving Toolbox provides a co-simulation framework that you can use to model driving algorithms in Simulink and visualize their performance in a virtual simulation environment. This environment uses the Unreal Engine by Epic Games.

Understanding how this simulation environment works can help you troubleshoot issues and customize your models.

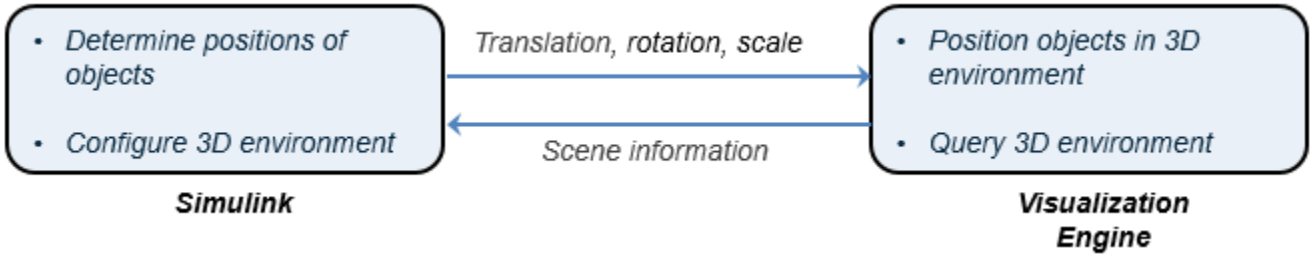
Communication with 3D Simulation Environment

When you use Automated Driving Toolbox to run your algorithms, Simulink co-simulates the algorithms in the visualization engine.

In the Simulink environment, Automated Driving Toolbox:

- Configures the visualization environment, specifically the ray tracing, scene capture from cameras, and initial object positions
- Determines the next position of the objects by using the simulation environment feedback

The diagram summarizes the communication between Simulink and the visualization engine.



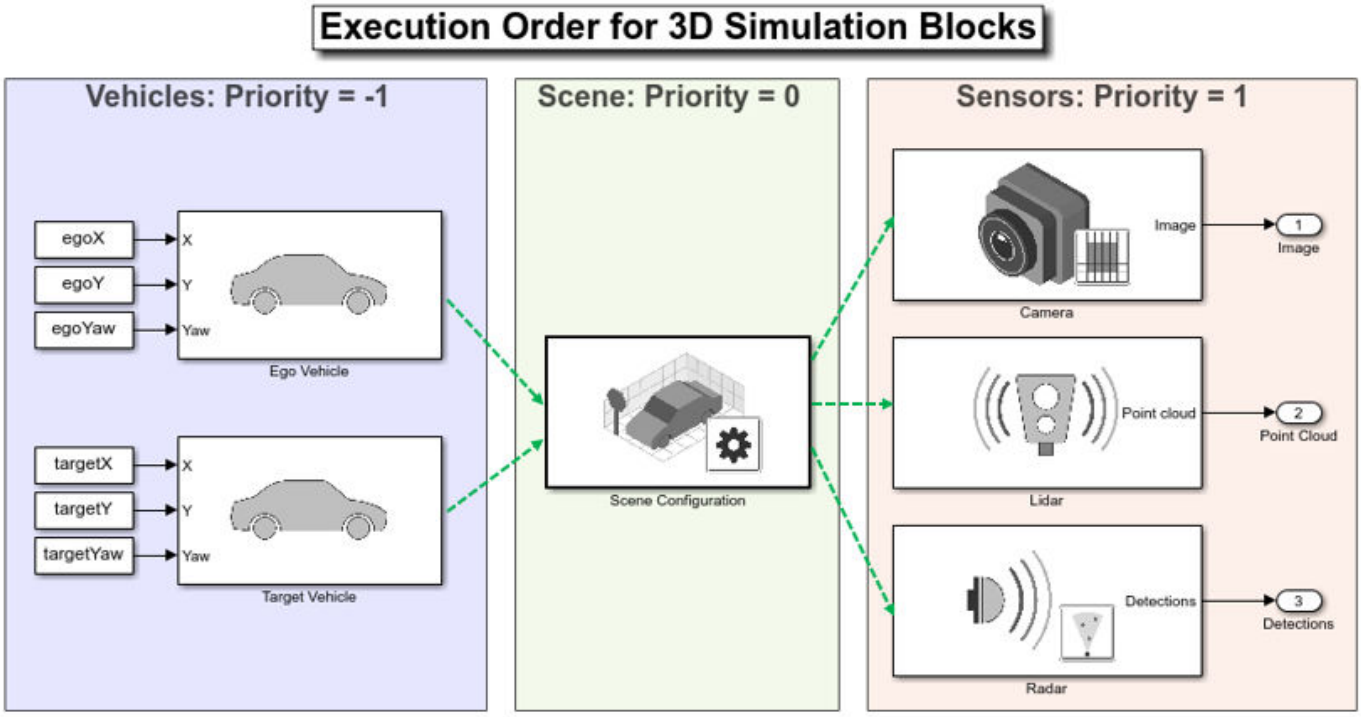
Block Execution Order

During simulation, the Unreal Engine simulation blocks follow a specific execution order:

1. The Simulation 3D Vehicle with Ground Following blocks initialize the vehicles and send their **X**, **Y**, and **Yaw** signal data to the Simulation 3D Scene Configuration block.
2. The Simulation 3D Scene Configuration block receives the vehicle data and sends it to the sensor blocks.
3. The sensor blocks receive the vehicle data and use it to accurately locate and visualize the vehicles.

The **Priority** property of the blocks controls this execution order. To access this property for any block, right-click the block, select **Properties**, and click the **General** tab. By default, Simulation 3D Vehicle with Ground Following blocks have a priority of `-1`, Simulation 3D Scene Configuration blocks have a priority of `0`, and sensor blocks have a priority of `1`.

The diagram shows this execution order.



If your sensors are not detecting vehicles in the scene, it is possible that the Unreal Engine simulation blocks are executing out of order. Try updating the execution order and simulating again. For more details on execution order, see “Control and Display Execution Order” (Simulink).

Also be sure that all 3D simulation blocks are located in the same subsystem. Even if the blocks have the correct **Priority** settings, if they are located in different subsystems, they still might execute out of order.

See Also

More About

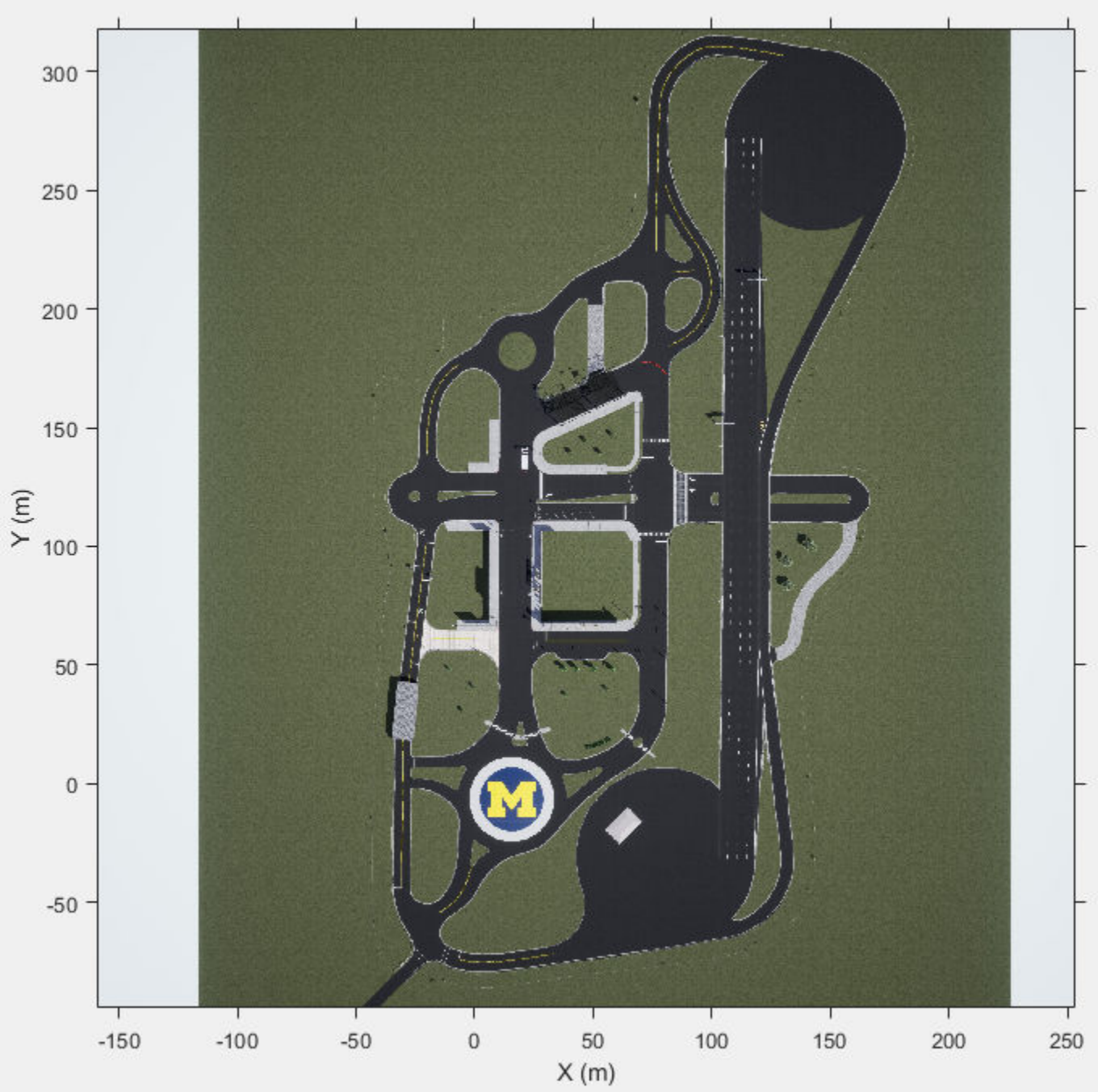
- “Unreal Engine Simulation for Automated Driving” on page 6-2
- “Unreal Engine Simulation Environment Requirements and Limitations” on page 6-6
- “Choose a Sensor for Unreal Engine Simulation” on page 6-16
- “Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox” on page 6-10

Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox

Automated Driving Toolbox enables you to simulate your driving algorithms in a virtual environment that uses the Unreal Engine from Epic Games. In general, the coordinate systems used in this environment follow the conventions described in “Coordinate Systems in Automated Driving Toolbox” on page 1-2. However, when simulating in this environment, it is important to be aware of the specific differences and implementation details of the coordinate systems.

World Coordinate System

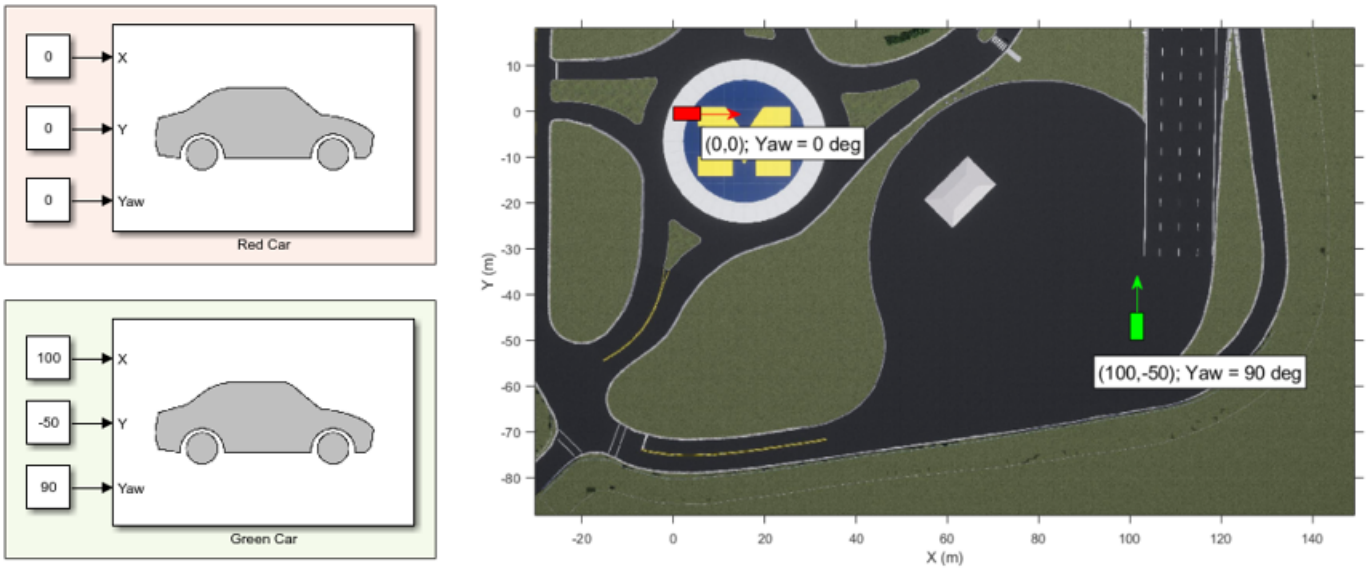
As with other Automated Driving Toolbox functionality, the simulation environment uses the right-handed Cartesian world coordinate system defined in ISO 8855. The following 2D top-view image of the **Virtual Mcity** scene shows the *X*- and *Y*-coordinates of the scene.



In this coordinate system, when looking in the positive direction of the *X*-axis, the positive *Y*-axis points left. The positive *Z*-axis points from the ground up. The yaw, pitch, and roll angles are clockwise-positive, when looking in the positive directions of the *Z*-, *Y*-, and *X*-axes, respectively. If you view a scene from a 2D top-down perspective, then the yaw angle is counterclockwise-positive, because you are viewing the scene in the negative direction of the *Z*-axis.

Placing Vehicles in a Scene

Vehicles are placed in the world coordinate system of the scenes. The figure shows how specifying the **X**, **Y**, and **Yaw** ports in the Simulation 3D Vehicle with Ground Following blocks determines their placement in a scene.



The elevation and banking angle of the ground determine the *Z*-axis, roll angle, and pitch angle of the vehicles.

Difference from Unreal Editor World Coordinates

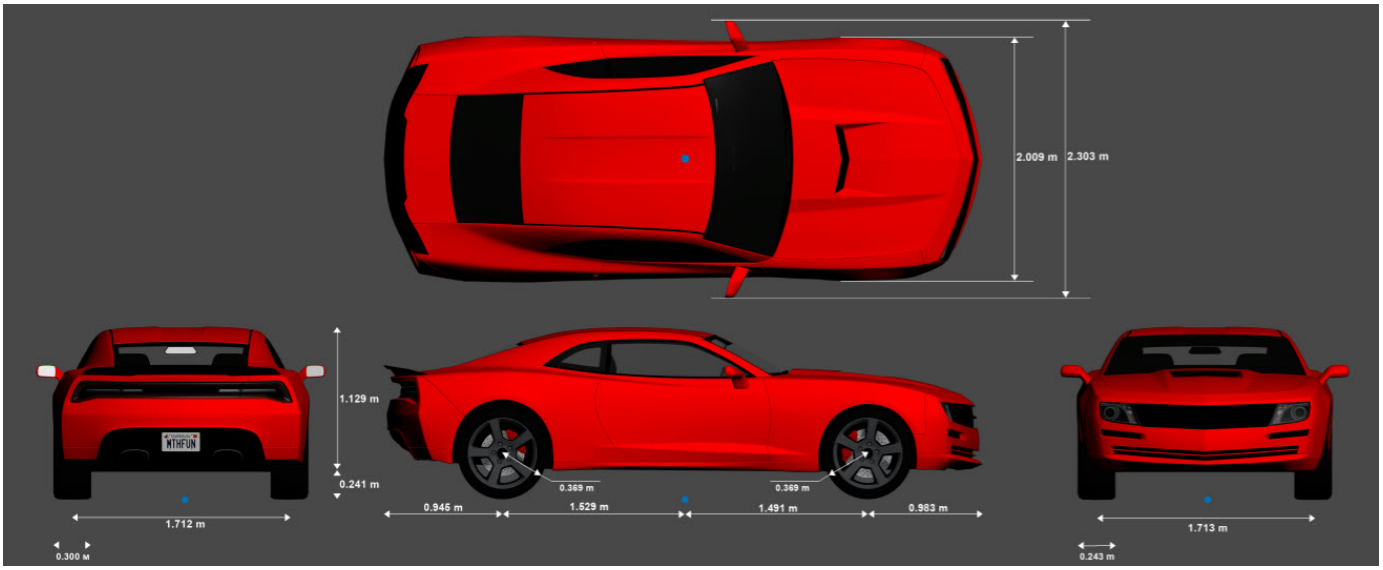
The Unreal® Editor uses a left-handed world Cartesian coordinate system in which the positive *Y*-axis points right. If you are converting from the Unreal Editor coordinate system to the coordinate system of the 3D environment, you must flip the sign of the *Y*-axis and pitch angle. The *X*-axis, *Z*-axis,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

Vehicle Coordinate System

The vehicle coordinate system is based on the world coordinate system. In this coordinate system:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle.
- The *Z*-axis points up from the ground.
- Roll, pitch, and yaw are clockwise-positive when looking in the forward direction of the *X*-, *Y*-, and *Z*-axes, respectively. As with the world coordinate system, when looking at a vehicle from the top down, then the yaw angle is counterclockwise-positive.

The vehicle origin is on the ground, at the geometric center of the vehicle. In this figure, the blue dot represents the vehicle origin.



Mounting Sensors on a Vehicle

When you add a sensor block, such as a Simulation 3D Camera block, to your model, you can mount the sensor to a predefined vehicle location, such as the front bumper of the root center. These mounting locations are in the vehicle coordinate system. When you specify an offset from these locations, you offset from the origin of the mounting location, not from the vehicle origin.

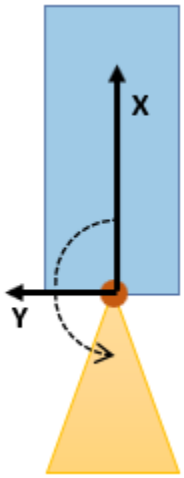
These equations define the vehicle coordinates for a sensor with location (X, Y, Z) and orientation ($Roll$, $Pitch$, Yaw):

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

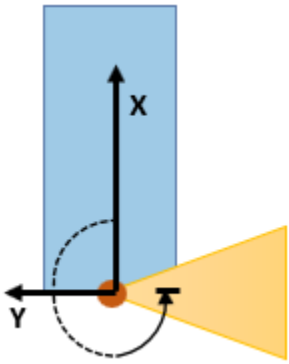
The "mount" variables refer to the predefined mounting locations relative to the vehicle origin. You define these mounting locations in the **Mounting location** parameter of the sensor block.

The "offset" variables refer to the amount of offset from these mounting locations. You define these offsets in the **Relative translation [X, Y, Z] (m)** and **Relative rotation [Roll, Pitch, Yaw] (deg)** parameters of the sensor block.

For example, consider a sensor mounted to the `Rear bumper` location. Relative to the vehicle origin, the sensor has an orientation of (0, 0, 180). In other words, when looking at the vehicle from the top down, the yaw angle of the sensor is rotated counterclockwise 180 degrees.

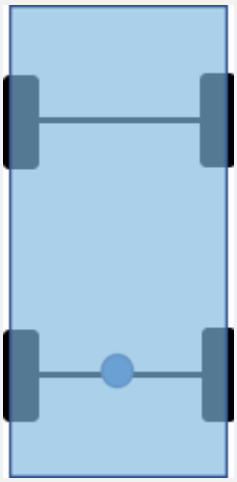


To point the sensor 90 degrees further to the right, you need to set the **Relative rotation [Roll, Pitch, Yaw] (deg)** parameter to `[0,0,90]`. In other words, the sensor is rotated 270 degrees counterclockwise relative to the vehicle origin, but it is rotated only 90 degrees counterclockwise relative to the origin of the predefined rear bumper location.



Difference from Cuboid Vehicle Origin

In the cuboid simulation environment, as described in “Cuboid Scenario Simulation”, the origin is on the ground, below the center of the rear axle of the vehicle.

Cuboid Vehicle Origin	3D Simulation Vehicle Origin
	

If you are converting sensor positions between coordinate systems, then you need to account for this difference in origin by using a Cuboid To 3D Simulation block. For an example model that uses this block, see “Highway Lane Following” on page 7-653.

Difference from Unreal Editor Vehicle Coordinates

The Unreal Editor uses a left-handed Cartesian vehicle coordinate system in which the positive *Y*-axis points right. If you are converting from the Unreal Editor coordinate system to the coordinate system of the Unreal Engine environment, you must flip the sign of the *Y*-axis and pitch angle. The *X*-axis, *Z*-axis, roll angle, and yaw angle are the same in both coordinate systems.

See Also

Cuboid To 3D Simulation | Simulation 3D Vehicle with Ground Following

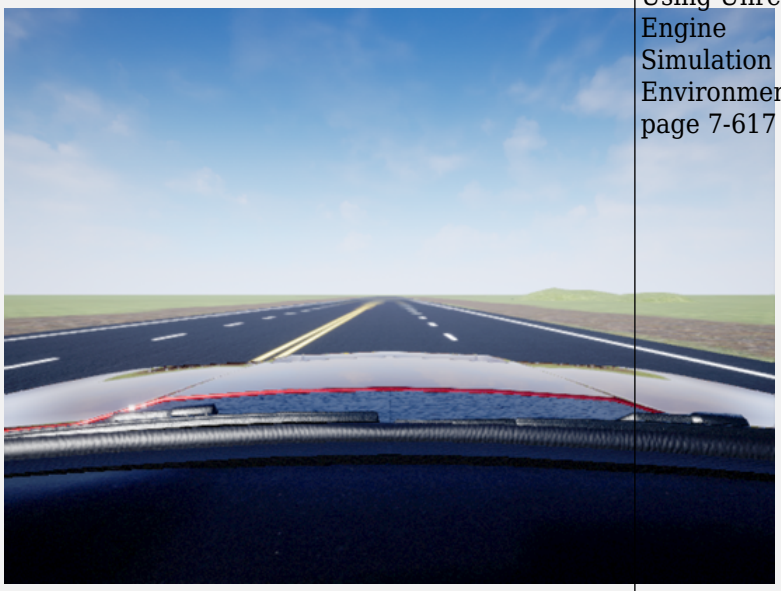
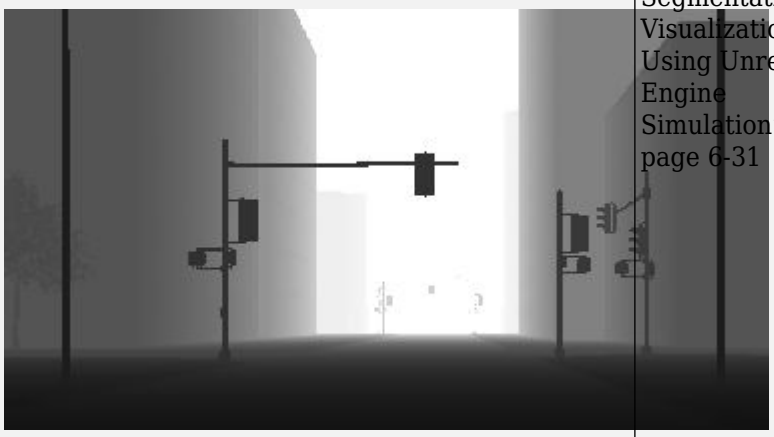
More About

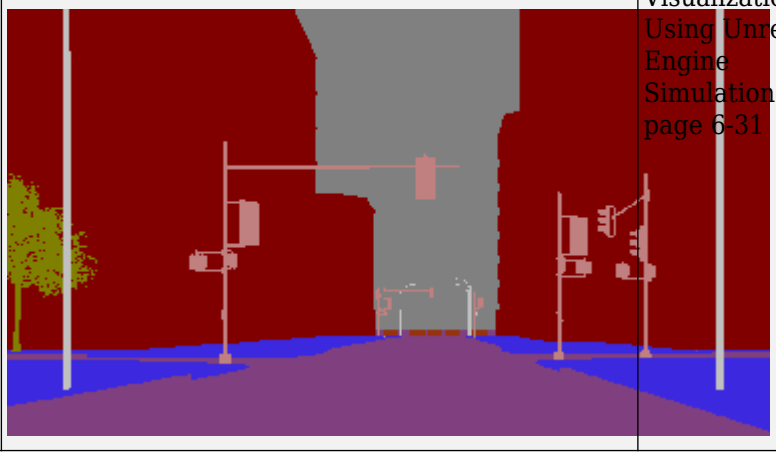
- “How Unreal Engine Simulation for Automated Driving Works” on page 6-8
- “Coordinate Systems in Automated Driving Toolbox” on page 1-2
- “Coordinate Systems in Vehicle Dynamics Blockset” (Vehicle Dynamics Blockset)

Choose a Sensor for Unreal Engine Simulation

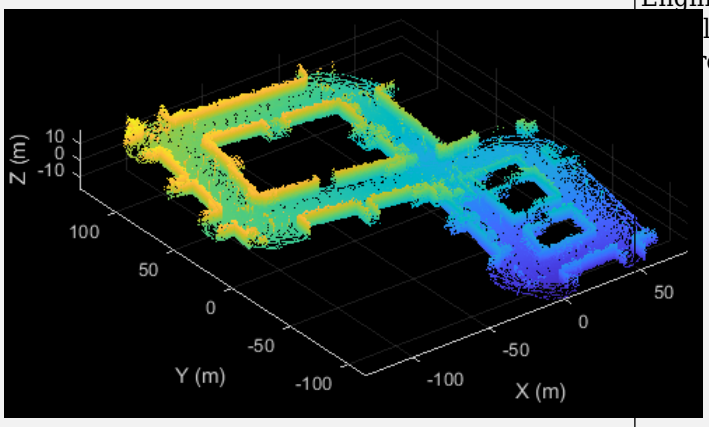
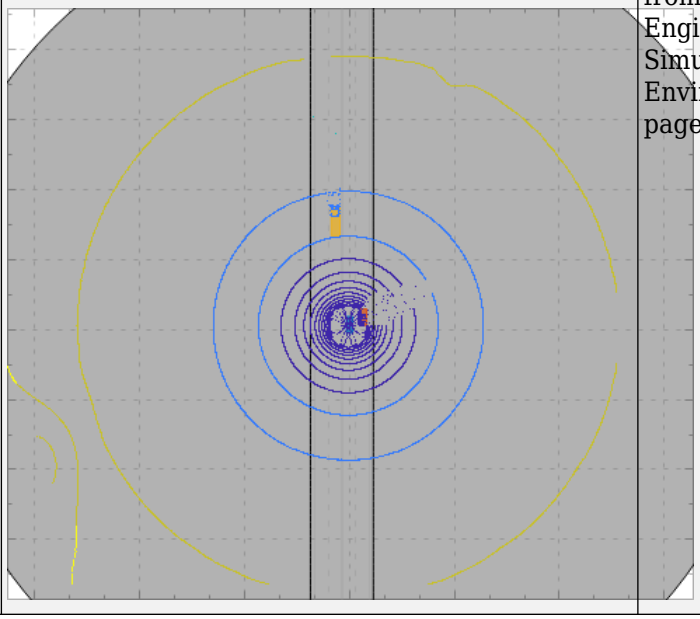
In Automated Driving Toolbox, you can obtain high-fidelity sensor data from a virtual environment. This environment is rendered using the Unreal Engine from Epic Games.

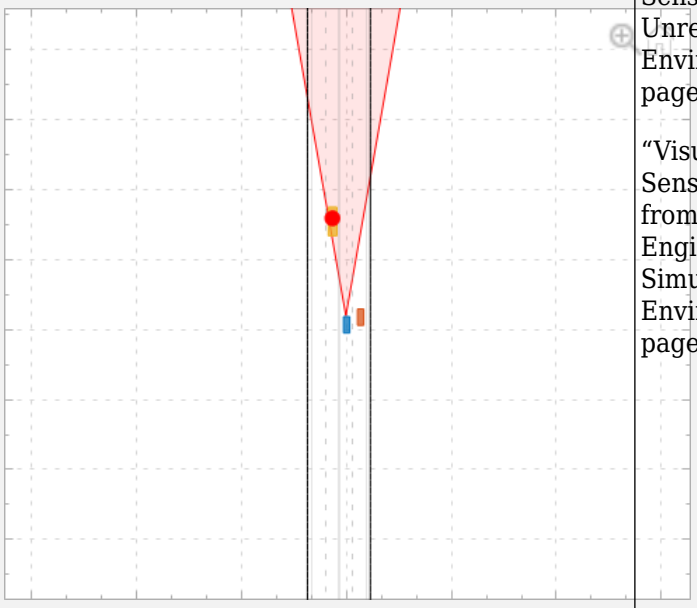
The table summarizes the sensor blocks that you can simulate in this environment.

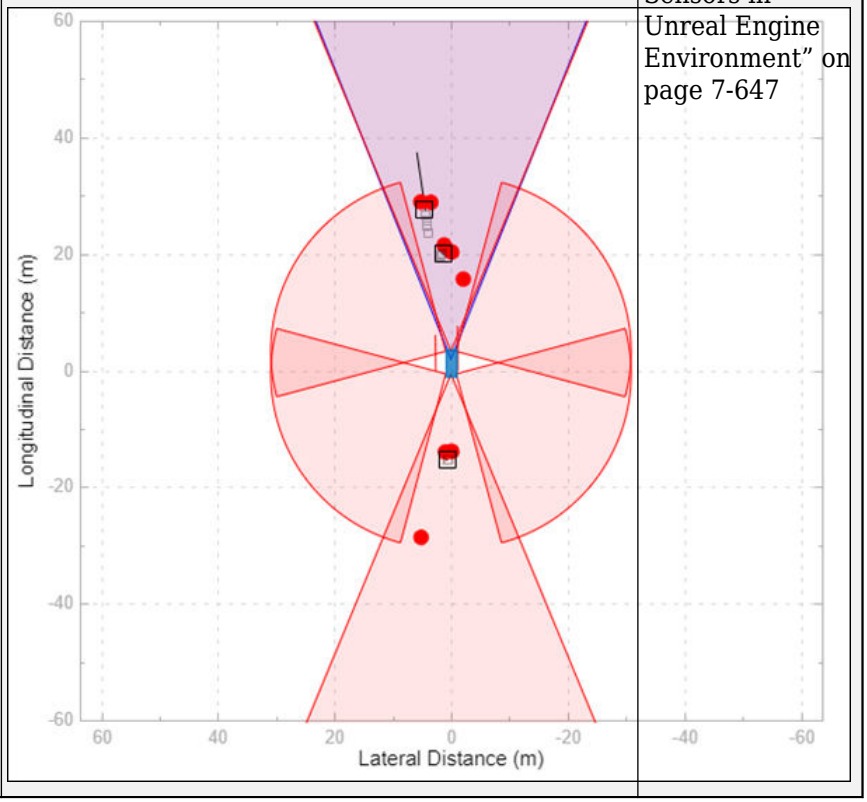
Sensor Block	Description	Visualization	Example
Simulation 3D Camera	• Camera with lens that is based on the ideal pinhole camera. See "What Is Camera Calibration?" (Computer Vision Toolbox) • Includes parameters for image size, focal length, distortion, and skew • Includes options to output ground truth for depth estimation and	Display camera images by using a Video Viewer or To Video Display block. Sample visualization: 	"Design Lane Marker Detector Using Unreal Engine Simulation Environment" on page 7-617
		Display depth maps by using a Video Viewer or To Video Display block. Sample visualization: 	"Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation" on page 6-31

Sensor Block	Description	Visualization	Example
	semantic segmentation	Display semantic segmentation maps by using a Video Viewer or To Video Display block. Sample visualization: 	"Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation" on page 6-31

Sensor Block	Description	Visualization	Example
Simulation 3D Fisheye Camera	• Fisheye camera that can be described using the Scaramuzza camera model. See "Fisheye Calibration Basics" (Computer Vision Toolbox) • Includes parameters for distortion center, image size, and mapping coefficients	Display camera images by using a Video Viewer or To Video Display block. Sample visualization: 	"Simulate Simple Driving Scenario and Sensor in Unreal Engine Environment" on page 6-22

Sensor Block	Description	Visualization	Example
Simulation 3D Lidar	• Scanning lidar sensor model • Includes parameters for detection range, resolution, and fields of view	Display point cloud data by using `pcplayer` within a MATLAB Function block. Sample visualization: 	“Design Lidar SLAM Algorithm Using Unreal Engine lation ronment” on 7-691
		Display lidar coverage areas and detections by using the **Bird's-Eye Scope**. Sample visualization: 	“Visualize Sensor Data from Unreal Engine Simulation Environment” on page 6-36

Sensor Block	Description	Visualization	Example
Simulation 3D Probabilistic Radar	• Probabilistic radar model that returns a list of detections • Includes parameters for radar accuracy, radar bias, detection probability, and detection reporting	Display radar coverage areas and detections by using the **Bird's-Eye Scope**. Sample visualization: 	"Simulate Vision and Radar Sensors in Unreal Engine Environment" on page 7-647 "Visualize Sensor Data from Unreal Engine Simulation Environment" on page 6-36

Sensor Block	Description	Visualization	Example
Simulation 3D Vision Detection Generator	• Camera model that returns a list of object and lane boundary detections • Includes parameters for detection accuracy, measurement noise, and camera intrinsics	Display vision coverage areas and detections by using the **Bird's-Eye Scope**. Sample visualization: 	"Simulate Vision and Radar Sensors in Unreal Engine Environment" on page 7-647

See Also

Blocks

Simulation 3D Probabilistic Radar Configuration | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following

More About

- "Unreal Engine Simulation for Automated Driving" on page 6-2

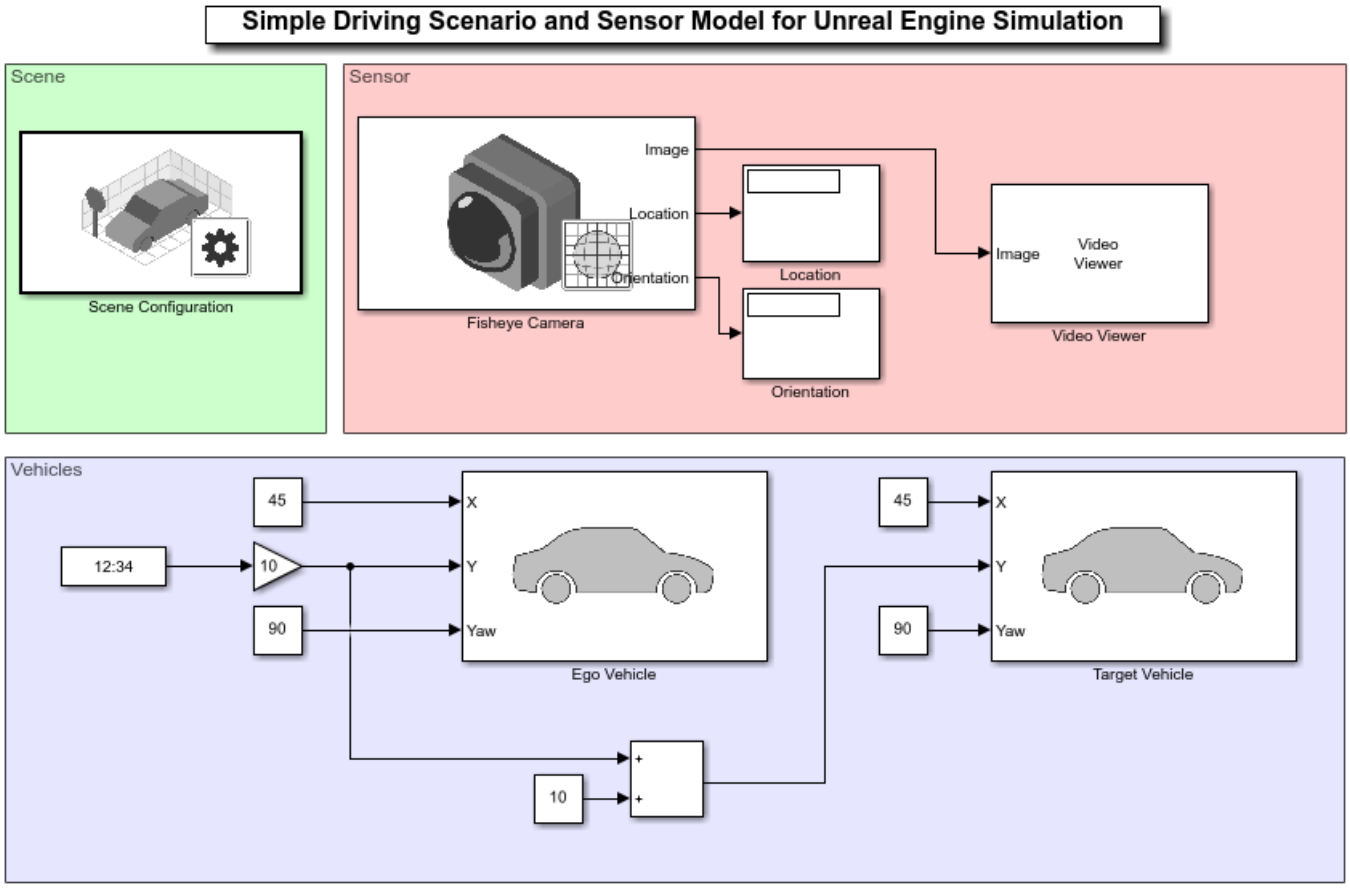
Simulate Simple Driving Scenario and Sensor in Unreal Engine Environment

Automated Driving Toolbox™ provides blocks for visualizing sensors in a simulation environment that uses the Unreal Engine® from Epic Games®. This model simulates a simple driving scenario in a prebuilt scene and captures data from the scene using a fisheye camera sensor. Use this model to learn the basics of configuring and simulating scenes, vehicles, and sensors. For more background on the Unreal Engine simulation environment, see “Unreal Engine Simulation for Automated Driving” on page 6-2.

Model Overview

The model consists of these main components:

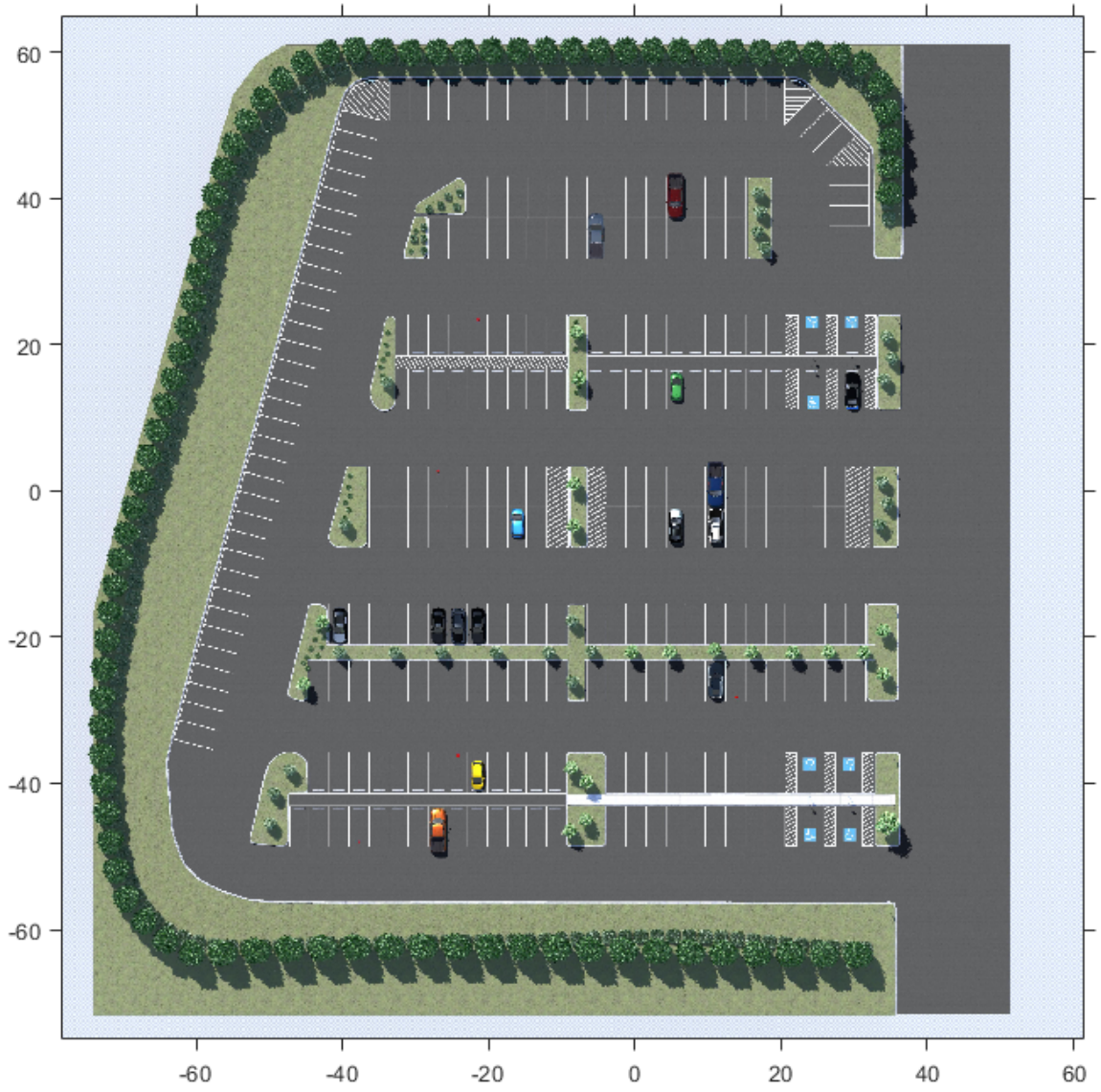
- Scene — A Simulation 3D Scene Configuration block configures the scene in which you simulate.
- Vehicles — Two Simulation 3D Vehicle with Ground Following blocks configure the vehicles within the scene and specify their trajectories.
- Sensor — A Simulation 3D Fisheye Camera configures the mounting position and parameters of the fisheye camera used to capture simulation data. A Video Viewer (Computer Vision Toolbox) block visualizes the simulation output of this sensor.



Inspect Scene

In the Simulation 3D Scene Configuration block, the **Scene name** parameter determines the scene where the simulation takes place. This model uses the Large Parking Lot scene, but you can choose among several prebuilt scenes. To explore a scene, you can open the 2D image corresponding to the 3D scene.

```
data = load('sim3d_SpatialReferences.mat');
spatialRef = data.spatialReference.LargeParkingLot;
figure; imshow('sim3d_LargeParkingLot.jpg',spatialRef)
set(gca,'YDir','normal')
```



To learn how to explore other scenes, see the corresponding scene reference pages.

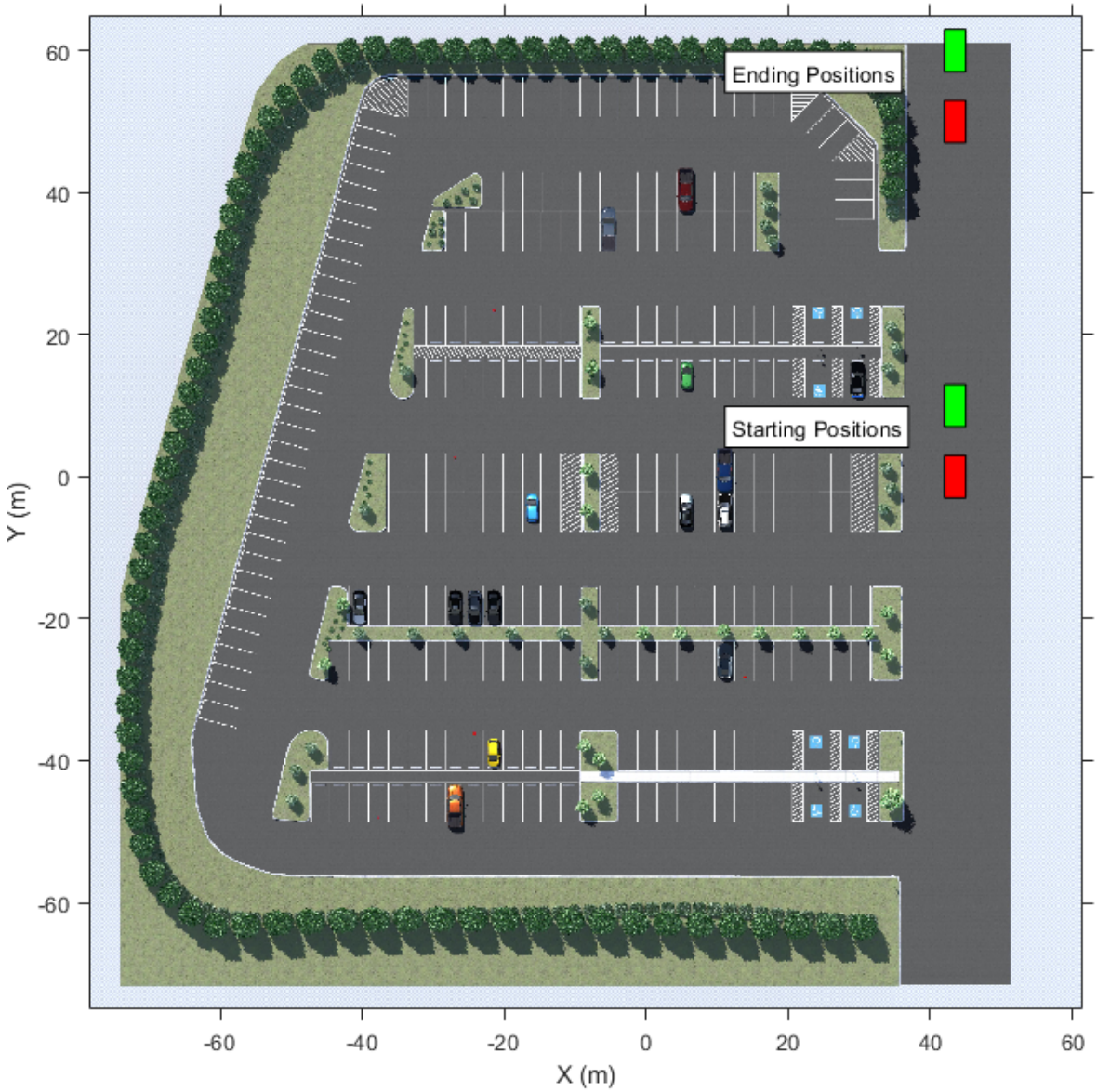
The **Scene view** parameter of this block determines the view from which the Unreal Engine window displays the scene. In this block, **Scene view** is set to `EgoVehicle`, which is the name of the ego vehicle (the vehicle with the sensor) in this scenario. During simulation, the Unreal Engine window displays the scene from behind the ego vehicle. You can also change the scene view to the other vehicle. To display the scene from the root of the scene (the scene origin), select `root`.

Inspect Vehicles

The Simulation 3D Vehicle with Ground Following blocks model the vehicles in the scenario.

- The Ego Vehicle block vehicle contains the fisheye camera sensor. This vehicle is modeled as a red hatchback.
- The Target Vehicle block is the vehicle from which the sensor captures data. This vehicle is modeled as a green SUV.

During simulation, both vehicles travel straight in the parking lot for 50 meters. The target vehicle is 10 meters directly in front of the ego vehicle.



The **X**, **Y**, and **Yaw** input ports control the trajectories of these vehicles. **X** and **Y** are in the world coordinates of the scene, which are in meters. **Yaw** is the orientation angle of the vehicle and is in degrees.

The ego vehicle travels from a position of (45,0) to (45,50), oriented 90 degrees counterclockwise from the origin. To model this position, the input port values are as follows:

- **X** is a constant value of `45`.
- **Y** is a multiple of the simulation time. A Digital Clock block outputs the simulation time every 0.1 second for 5 seconds, which is the stop time of the simulation. These simulation times are then multiplied by 10 to produce **Y** values of `[0 1 2 3 ... 50]`, or 1 meter for up to a total of 50 meters.
- **Yaw** is a constant value of `90`.

The target vehicle has the same **X** and **Yaw** values as the ego vehicle. The **Y** value of the target vehicle is always 10 meters more than the **Y** value of the ego vehicle.

In both vehicles, the **Initial position [X, Y, Z] (m)** and **Initial rotation [Roll, Pitch, Yaw] (deg)** parameters reflect the initial `[X, Y, Z]` and `[Yaw, Pitch, Roll]` values of the vehicles at the beginning of simulation.

To create more realistic trajectories, you can obtain waypoints from a scene interactively and specify these waypoints as inputs to the Simulation 3D Vehicle with Ground Following blocks. See “Select Waypoints for Unreal Engine Simulation” on page 7-626.

Inspect Sensor

The Simulation 3D Fisheye Camera block models the sensor used in the scenario. Open this block and inspect its parameters.

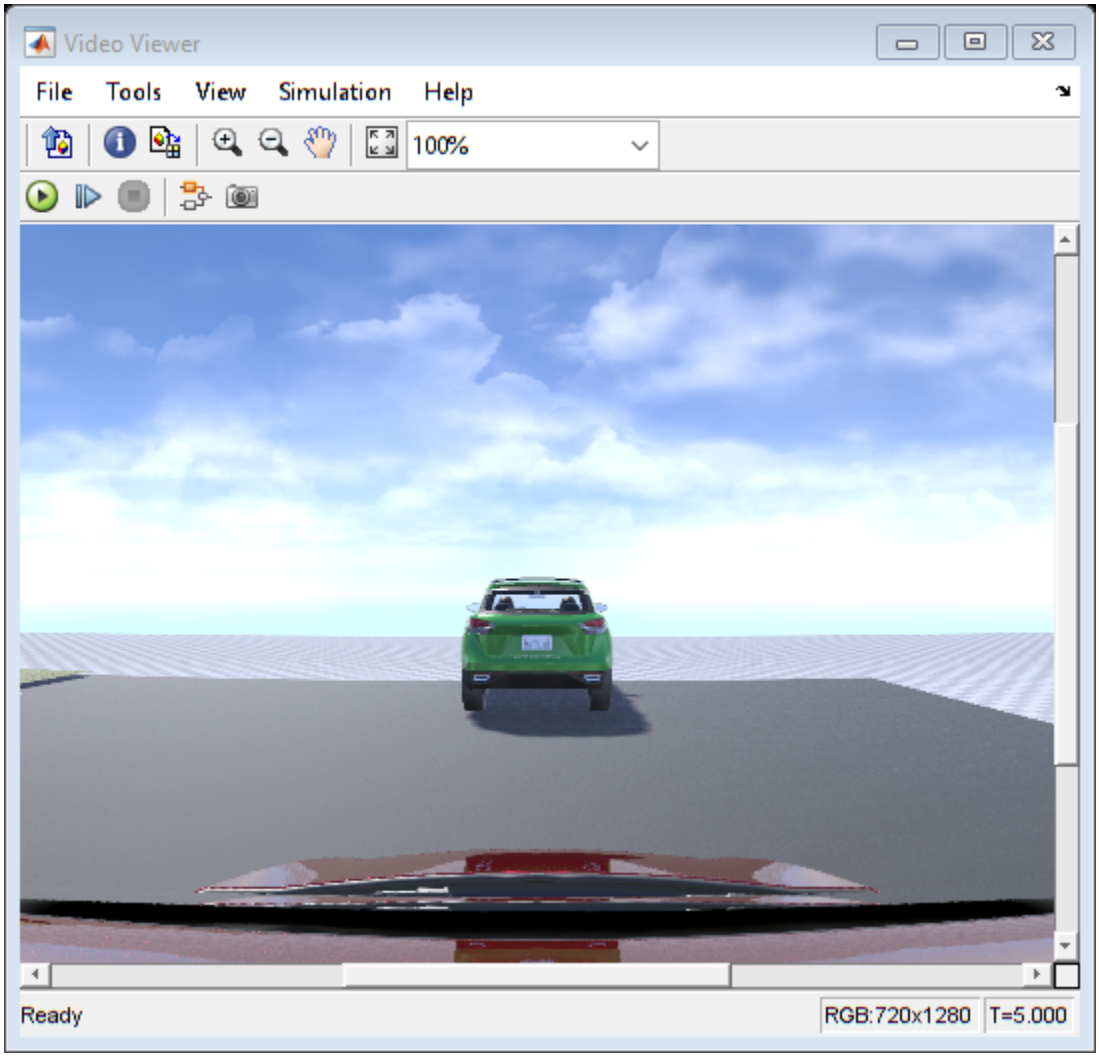
- The **Mounting** tab contains parameters that determine the mounting location of the sensor. The fisheye camera sensor is mounted to the center of the roof of the ego vehicle.
- The **Parameters** tab contains the intrinsic camera parameters of a fisheye camera. These parameters are set to their default values.
- The **Ground Truth** tab contains a parameter for outputting the location and orientation of the sensor in meters and radians. In this model, the block outputs these values so you can see how they change during simulation.

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

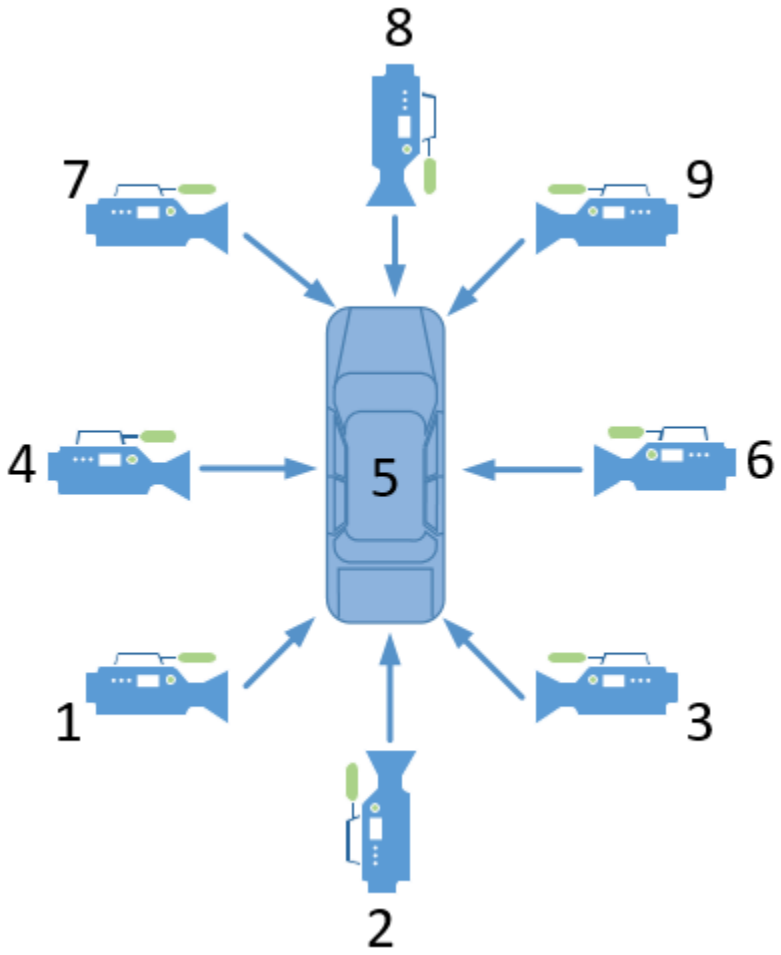
Simulate Model

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

The Video Viewer block shows the output of the fisheye camera.



To change the view of the scene during simulation, use the numbers 1–9 on the numeric keypad.



For a bird's-eye view of the scene, press 0.

After simulating the model, try modifying the intrinsic camera parameters and observe the effects on simulation. You can also change the type of sensor block. For example, try substituting the 3D Simulation Fisheye Camera with a 3D Simulation Camera block. For more details on the available sensor blocks, see “Choose a Sensor for Unreal Engine Simulation” on page 6-16.

See Also

Simulation 3D Camera | Simulation 3D Fisheye Camera | Simulation 3D Lidar | Simulation 3D Probabilistic Radar | Simulation 3D Vehicle with Ground Following | Simulation 3D Vision Detection Generator | Simulation 3D Scene Configuration

More About

- “Unreal Engine Simulation for Automated Driving” on page 6-2
- “Unreal Engine Simulation Environment Requirements and Limitations” on page 6-6
- “How Unreal Engine Simulation for Automated Driving Works” on page 6-8
- “Coordinate Systems in Automated Driving Toolbox” on page 1-2

- “Select Waypoints for Unreal Engine Simulation” on page 7-626
- “Design Lane Marker Detector Using Unreal Engine Simulation Environment” on page 7-617

Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation

This example shows how to visualize depth and semantic segmentation data captured from a camera sensor in a simulation environment. This environment is rendered using the Unreal Engine® from Epic Games®.

You can use depth visualizations to validate depth estimation algorithms for your sensors. You can use semantic segmentation visualizations to analyze the classification scheme used for generating synthetic semantic segmentation data from the Unreal Engine environment.

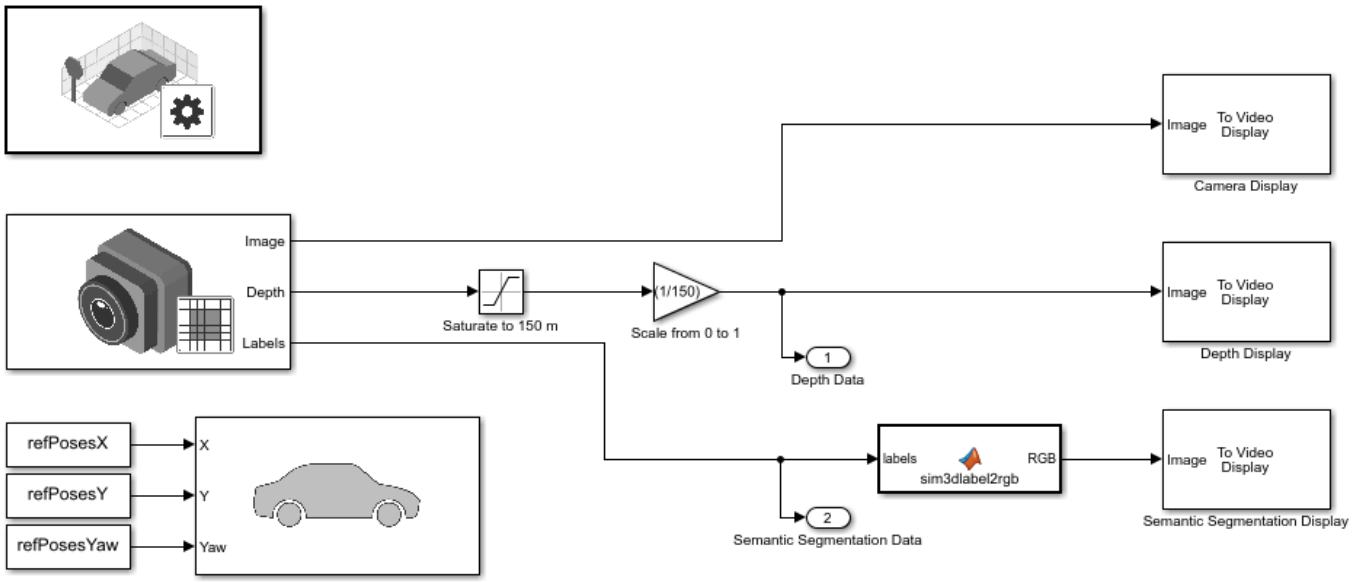
Model Setup

The model used in this example simulates a vehicle driving in a city scene.

- A Simulation 3D Scene Configuration block sets up simulation with the US City Block scene.
- A Simulation 3D Vehicle with Ground Following block specifies the driving route of the vehicle. The waypoint poses that make up this route were obtained using the technique described in the “Select Waypoints for Unreal Engine Simulation” on page 7-626 example.
- A Simulation 3D Camera block mounted to the rearview mirror of the vehicle captures data from the driving route. This block outputs the camera, depth, and semantic segmentation displays by using To Video Display (Computer Vision Toolbox) blocks.

Load the MAT-file containing the waypoint poses. Add timestamps to the poses and then open the model.

```
load smoothedPoses.mat;

refPosesX   = [linspace(0,20,1000)', smoothedPoses(:,1)];
refPosesY   = [linspace(0,20,1000)', smoothedPoses(:,2)];
refPosesYaw = [linspace(0,20,1000)', smoothedPoses(:,3)];

open_system('DepthSemanticSegmentation.slx')
```



Depth Visualization

A depth map is a grayscale representation of camera sensor output. These maps visualize camera images in grayscale, with brighter pixels indicating objects that are farther away from the sensor. You can use depth maps to validate depth estimation algorithms for your sensors.

The **Depth** port of the Simulation 3D Camera block outputs a depth map of values in the range of 0 to 1000 meters. In this model, for better visibility, a Saturation block saturates the depth output to a maximum of 150 meters. Then, a Gain block scales the depth map to the range [0, 1] so that the To Video Display block can visualize the depth map in grayscale.

Semantic Segmentation Visualization

Semantic segmentation describes the process of associating each pixel of an image with a class label, such as *road*, *building*, or *traffic sign*. In the 3D simulation environment, you generate synthetic semantic segmentation data according to a label classification scheme. You can them use these labels to train a neural network for automated driving applications, such as road segmentation. By visualizing the semantic segmentation data, you can verify your classification scheme.

The **Labels** port of the Simulation 3D Camera block outputs a set of labels for each pixel in the output camera image. Each label corresponds to an object class. For example, in the default classification scheme used by the block, `1` corresponds to buildings. A label of `0` refers to objects of an unknown class and appears as black. For a complete list of label IDs and their corresponding object descriptions, see the **Labels** port description on the Simulation 3D Camera block reference page.

The MATLAB Function block uses the `label2rgb (Image Processing Toolbox)` function to convert the labels to a matrix of RGB triplets for visualization. The colormap is based on the colors used in the CamVid dataset, as shown in the example “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox). The colors are mapped to the predefined label IDs used in the default 3D simulation scenes. The helper function `sim3dColormap` defines the colormap. Inspect these colormap values.

```
open sim3dColormap.m
```

```
function cmap = sim3dColormap
% Define colormap for object labels used in 3D simulation environment.

cmap = [
128 0 0     % Label 1: Building
0 0 0       % Label 2: Not used
72 0 90     % Label 3: Other
0 0 0       % Label 4: Not used
192 192 192 % Label 5: Pole
0 0 0       % Label 6: Not used
128 64 128  % Label 7: Roads
60 40 222   % Label 8: Sidewalk
128 128 0   % Label 9: Vegetation
64 0 128    % Label 10: Vehicle
0 0 0       % Label 11: Not used
```

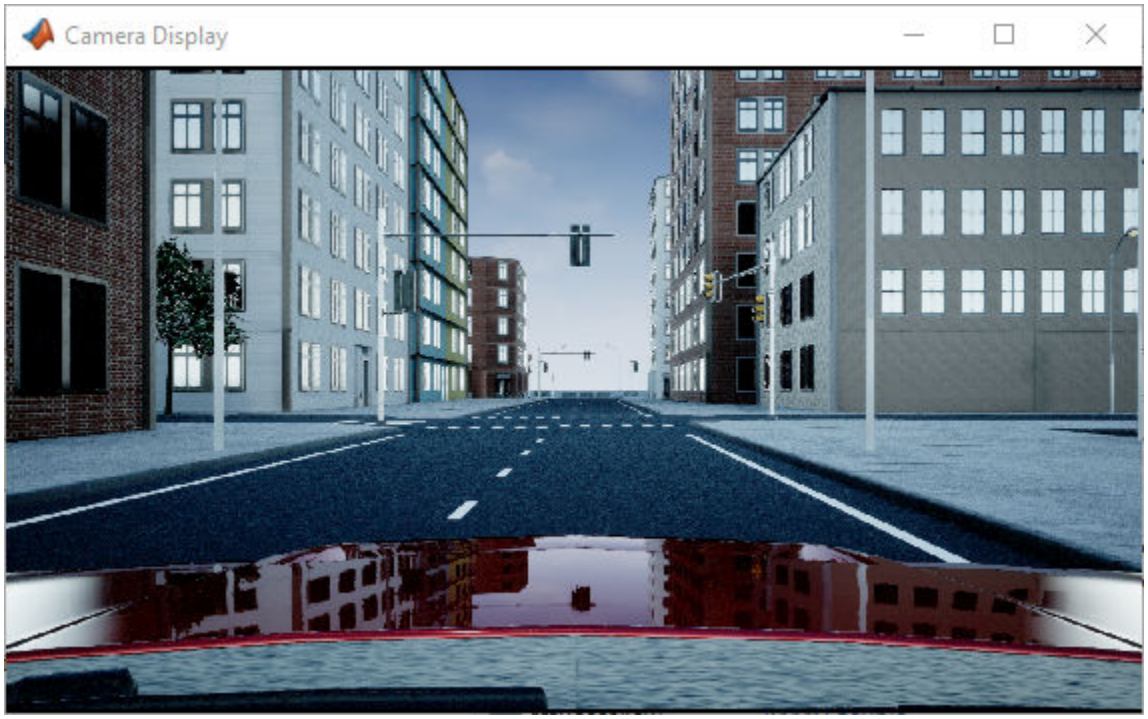
Model Simulation

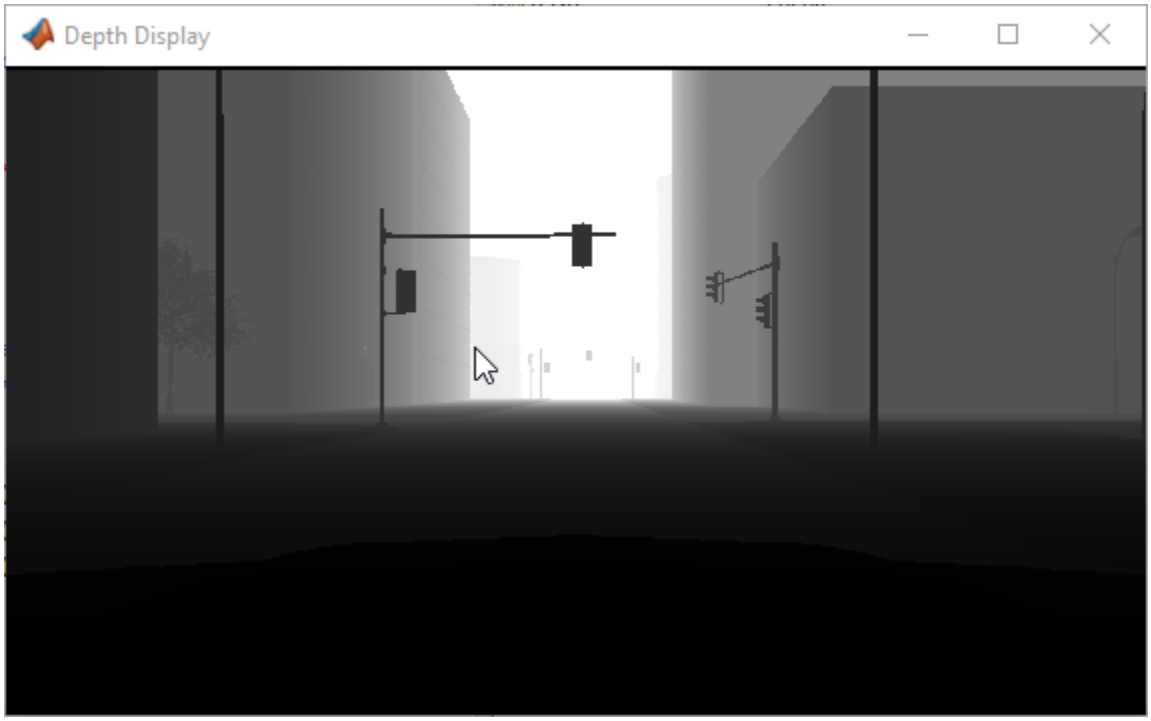
Run the model.

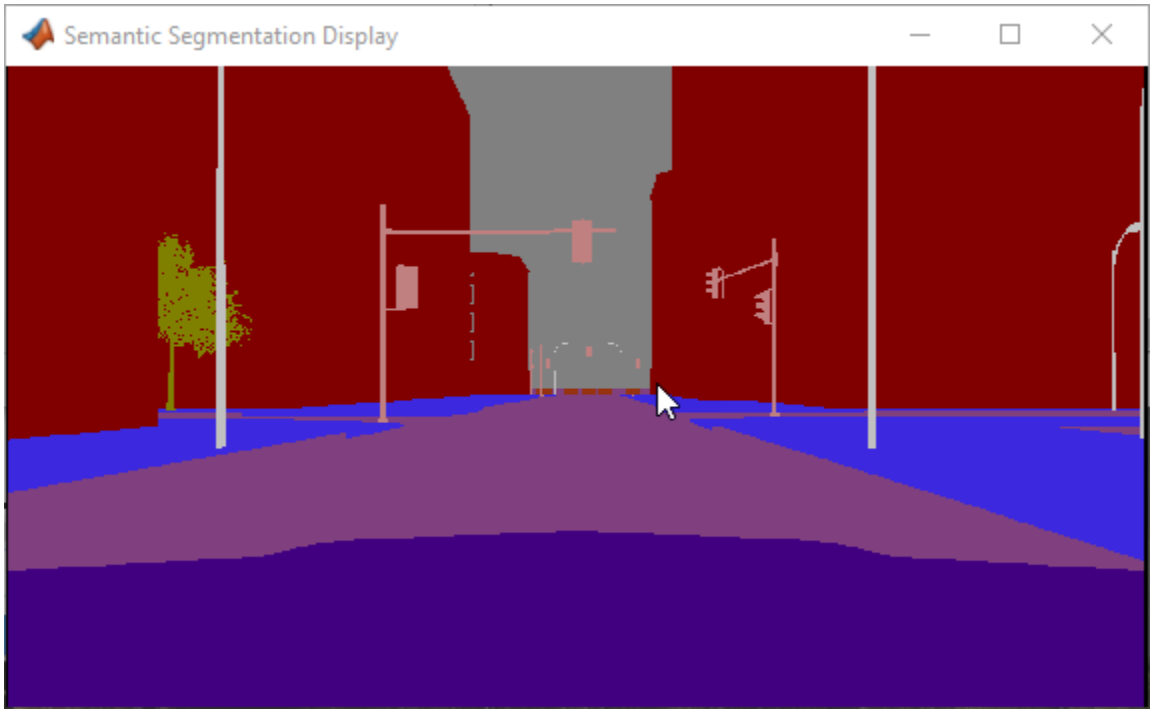
```
sim('DepthSemanticSegmentation.slx');
```

When the simulation begins, it can take a few seconds for the visualization engine to initialize, especially when you are running it for the first time. The `AutoVrtlEnv` window displays the scene from behind the ego vehicle. In this scene, the vehicles drives several blocks around the city. Because this example is mainly for illustrative purposes, the vehicle does not always follow the direction of traffic or the pattern of the changing traffic lights.

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


Camera Display


Depth Display



To change the visualization range of the output depth data, try updating the values in the Saturation and Gain blocks.

To change the semantic segmentation colors, try modifying the color values defined in the `sim3dColormap` function. Alternatively, in the `sim3dlabel2rgb` MATLAB Function block, try replacing the input colormap with your own colormap or a predefined colormap. See `colormap`.

See Also

Simulation 3D Camera | Simulation 3D Vehicle with Ground Following | Simulation 3D Scene Configuration

More About

- “Simulate Simple Driving Scenario and Sensor in Unreal Engine Environment” on page 6-22
- “Select Waypoints for Unreal Engine Simulation” on page 7-626
- “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)
- “Train Deep Learning Semantic Segmentation Network Using 3-D Simulation Data” (Deep Learning Toolbox)

Visualize Sensor Data from Unreal Engine Simulation Environment

This example shows how to visualize sensor coverages and detections obtained from high-fidelity radar and lidar sensors in a 3D simulation environment. In this example, you learn how to:

1. Configure Simulink® models to simulate within the 3D environment. This environment is rendered using the Unreal Engine® from Epic Games®.
2. Read ground truth data and vehicle trajectories from a scenario authored using the **Driving Scenario Designer** app, and then recreate this scenario in the Simulink model.
3. Add radar and lidar sensors to these models by using Simulation 3D Probabilistic Radar and Simulation 3D Lidar blocks.
4. Visualize the driving scenario and generated sensor data in the **Bird's-Eye Scope**.

You can use these visualizations and sensor data to test and improve your automated driving algorithms. You can also extend this example to fuse detections and visualize object tracking results, as shown in the “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” on page 7-205 example.

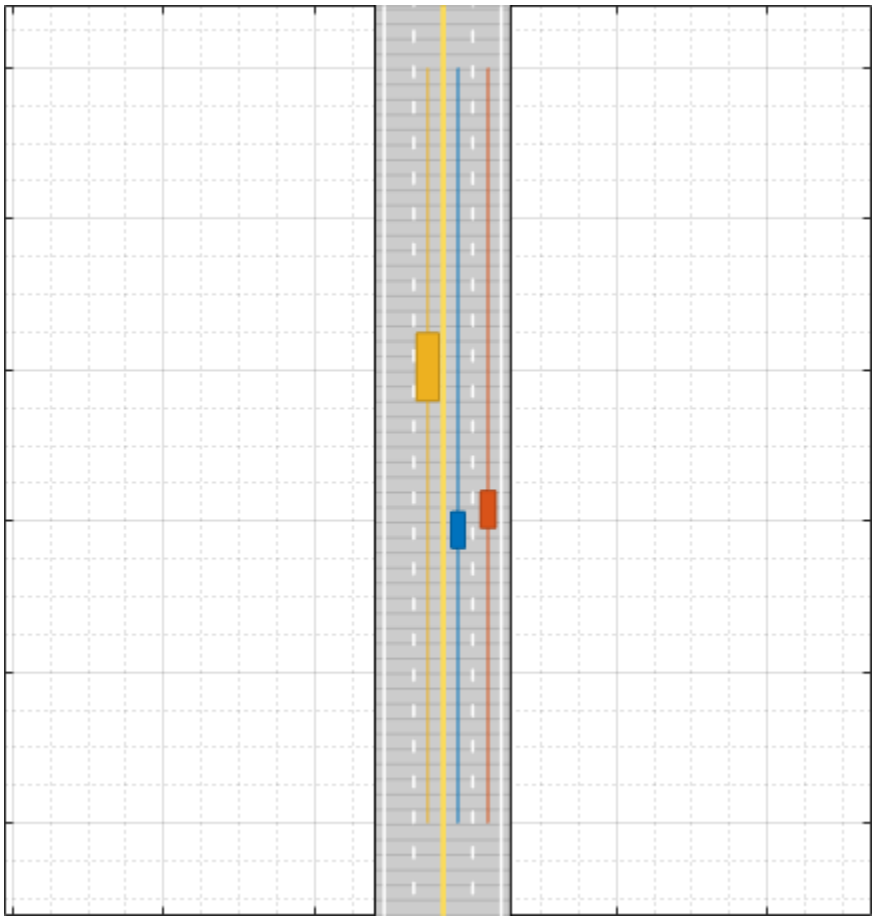
Inspect Cuboid Driving Scenario

In this example, the ground truth (roads, lanes, and actors) and vehicle trajectories come from a scenario that was authored in the **Driving Scenario Designer** app. In this app, vehicles and other actors are represented as simple box shapes called *cuboids*. For more details about authoring cuboid scenarios, see the “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2 example.

Open the cuboid driving scenario file in the app.

```
drivingScenarioDesigner('StraightRoadScenario.mat')
```

In the app, run the scenario simulation. In this scenario, the ego vehicle (a blue car) travels north along a straight road at a constant speed. In the adjacent lane, an orange car travels north at a slightly higher constant speed. In the opposite lane, a yellow truck drives south at a constant speed.



When authoring driving scenarios that you later recreate in the 3D simulation environment, you must use a road network identical to one from the default 3D scenes. Otherwise, in the recreated scenario, the positions of vehicles and sensors are inaccurate. This driving scenario uses a recreation of the Straight Road scene. To select a different cuboid version of a 3D scene, on the app toolstrip, select **Open > Prebuilt Scenario > Simulation3D** and choose from the available scenes. Not all 3D scenes have corresponding versions in the app.

- For a list of supported scenes and additional details about each scene, see “Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer” on page 5-62.
- To generate vehicle trajectories for scenes that are not available in the app, use the process described in the “Select Waypoints for Unreal Engine Simulation” on page 7-626 example instead.

The dimensions of vehicles in the cuboid scenarios must also match the dimensions of one of the predefined 3D simulation vehicle types. On the app toolstrip, under **3D Display**, the **Use 3D Simulation Actor Dimensions** selection sets each cuboid vehicle to have the dimensions of a 3D vehicle type. In this scenario, the vehicles have these 3D display types and corresponding vehicle dimensions.

- `Ego Vehicle` — Sedan vehicle dimensions
- `Vehicle in Adjacent Lane` — Muscle Car vehicle dimensions
- `Vehicle in Opposite Lane` — Box Truck vehicle dimensions

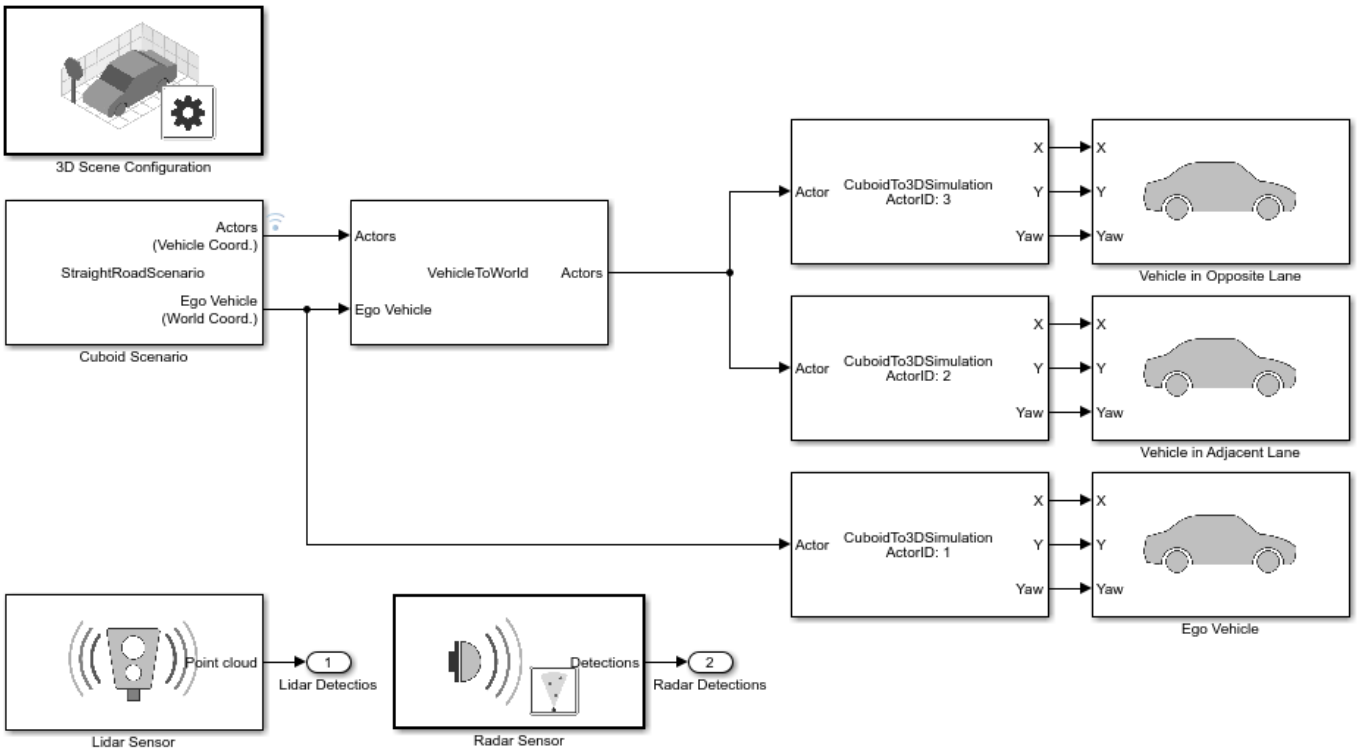
To change a vehicle to a different display type, on the **Actors** tab in the left pane of the app, update the **3D Display Type** parameter for that vehicle. To change the color of a vehicle, select the color patch next to the selected vehicle and choose a new color.

To preview how the vehicles display in the 3D environment, use the 3D display window available from the app. On the app toolstrip, select **3D Display > View Simulation in 3D Display** and rerun the simulation.

Open 3D Simulation Model

The model used in this example recreates the cuboid driving scenario. The model also defines high-fidelity sensors that generate synthetic detections from the environment. Open the model.

```
open_system('Visualize3DSimulationSensorCoveragesDetections')
```



Inspect Scene Configuration

The Simulation 3D Scene Configuration block configures the model to simulate in the 3D environment.

- The **Scene name** parameter is set to the default `Straight road` scene. This scene corresponds to the cuboid version defined in the app scenario file.
- The **Scene view** parameter is set to `Ego Vehicle`. During simulation, the 3D simulation window displays the scene from behind the ego vehicle.

The Scenario Reader block reads the ground truth data (road boundaries, lane markings, and actor poses) from the app scenario file. The **Bird's-Eye Scope** visualizes this ground truth data, not the ground truth data of the 3D simulation environment. To use the same scene for the cuboid and 3D simulation environments, the ground truth data for both environments must match. If you are creating a new scenario, you can generate a Scenario Reader block that reads data from your scenario file. First, open the scenario file in the **Driving Scenario Designer** app. Then, on the app toolstrip, select **Export > Export Simulink Model**. If you update the scenario, you do not need to generate a new Scenario Reader block.

The Simulation 3D Scene Configuration block and Scenario Reader block both have their **Sample time** parameter set to `0.1`. In addition, all other 3D simulation vehicle and sensor blocks inherit their sample time from the Simulation 3D Scene Configuration block. By setting a single sample time across the entire model, the **Bird's-Eye Scope** displays data from all blocks at a constant rate. If the ground truth and sensor data have different sample times, then the scope visualizes them at different time intervals. This process causes the ground truth and sensor data visualizations to flicker.

Inspect Vehicle Configuration

The Simulation 3D Vehicle with Ground Following blocks specify the appearances and trajectories of the vehicles in the 3D simulation environment. Each vehicle is a direct counterpart to one of the vehicles defined in the **Driving Scenario Designer** app scenario file.

In the 3D environment, vehicle positions are in world coordinates. However, the Scenario Reader block outputs the poses of non-ego actors in ego vehicle coordinates. A Vehicle To World block converts these non-ego actor poses into world coordinates. Because the ego vehicle is output in world coordinates, this conversion is not necessary for the ego vehicle. For more details about the vehicle and world coordinate systems, see “Coordinate Systems in Automated Driving Toolbox” on page 1-2.

Locations of vehicle origins differ between cuboid and 3D scenarios.

- In cuboid scenarios, the vehicle origin is on the ground, at the center of the rear axle.
- In 3D scenarios, the vehicle origin is on ground, at the geometric center of the vehicle.

The Cuboid To 3D Simulation blocks convert the cuboid origin positions to the 3D simulation origin positions. In the **ActorID used for conversion** parameters of these blocks, the specified `ActorID` of each vehicle determines which vehicle origin to convert. The Scenario Reader block outputs `ActorID` values in its **Actors** output port. In the **Driving Scenario Designer** app, you can find the corresponding `ActorID` values on the **Actors** tab, in the actor selection list. The `ActorID` for each vehicle is the value that precedes the colon.

Each Cuboid To 3D Simulation block outputs **X**, **Y**, and **Yaw** values that feed directly into their corresponding vehicle blocks. In the 3D simulation environment, the ground terrain of the 3D scene determines the *Z*-position (elevation), roll angle, and pitch angle of the vehicles.

In each Simulation 3D Vehicle with Ground Following block, the **Type** parameter corresponds to the **3D Display Type** selected for that vehicle in the app. In addition, the **Color** parameter corresponds to the vehicle color specified in the app. To maintain similar vehicle visualizations between the **Bird's-Eye Scope** and the 3D simulation window, the specified type and color must match. To change the color of a vehicle in the app, on the **Actors** tab, click the color patch to the right of the actor name in the actor selection list. Choose the color that most closely matches the colors available in the **Color** parameter of the Simulation 3D Vehicle with Ground Following block.

Inspect Sensor Configuration

The model includes two sensor blocks with default parameter settings. These blocks generate detections from the 3D simulation environment.

- The Simulation 3D Probabilistic Radar sensor block generates object detections based on a statistical model. This sensor is mounted to the front bumper of the ego vehicle.
- The Simulation 3D Lidar sensor block generates detections in the form of a point cloud. This sensor is mounted to the center of the roof of the ego vehicle.

Although you can specify sensors in the **Driving Scenario Designer** app and export them to Simulink, the exported blocks are not compatible with the 3D simulation environment. You must specify 3D simulation sensors in the model directly.

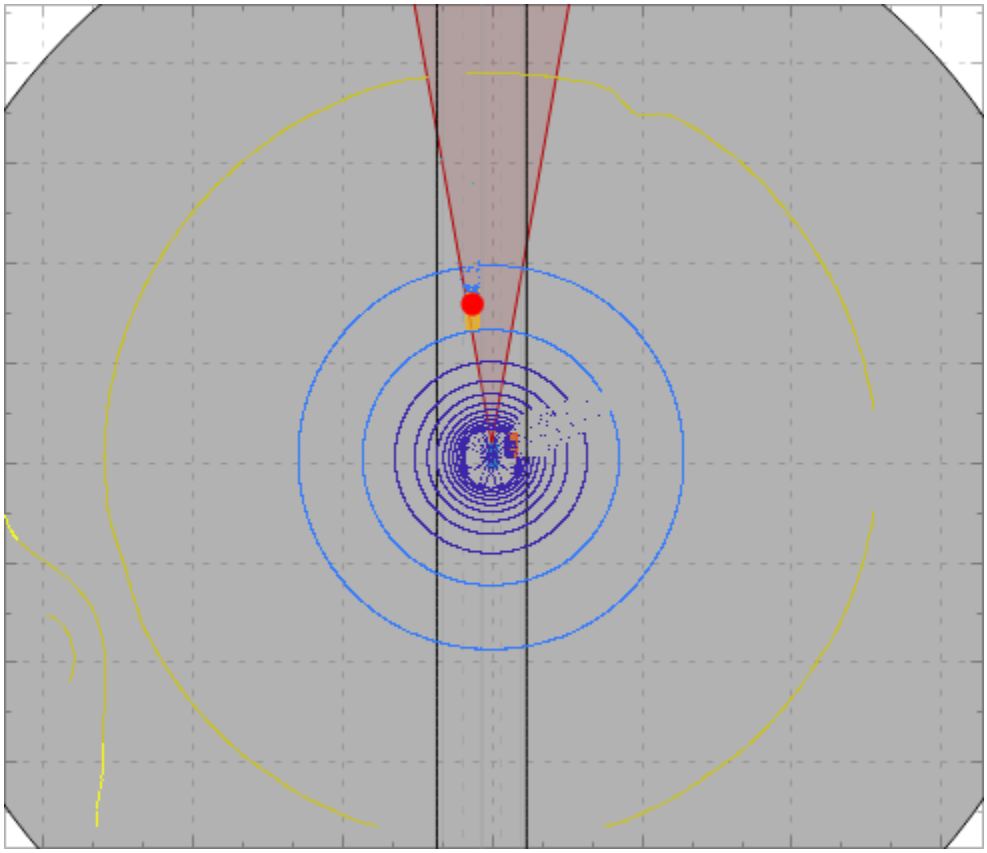
Simulate and Visualize Scenario

During simulation, you can visualize the scenario in both the 3D simulation window and the **Bird's-Eye Scope**.

First, open the scope. On the Simulink toolstrip, under **Review Results**, click **Bird's-Eye Scope**. Then, to find signals that the scope can display, click **Find Signals**.

To run the simulation, click **Run** in either the model or scope. When the simulation begins, it can take a few seconds for the 3D simulation window to initialize, especially when you run it for the first time in a Simulink session. When this window opens, it displays the scenario with high-fidelity graphics but does not display detections or sensor coverages.

The **Bird's-Eye Scope** displays detections and sensor coverages by using a cuboid representation. The radar coverage area and detections are in red. The lidar coverage area is in gray, and its point cloud detections display as a parula colormap.



The model runs the simulation at a pace of 0.5 seconds per wall-clock second. To adjust the pacing, from the Simulink toolstrip, select **Run > Simulation Pacing**, and then move the slider to increase or decrease the speed of the simulation.

Modify the Driving Scenario

When modifying your driving scenario, you might need to update the scenario in the **Driving Scenario Designer** app, the Simulink model, or in both places, depending on what you change.

- **Modify the road network** — In the app, select a new prebuilt scene from the **Simulation3D** folder. Do not modify these road networks or the roads will not match the roads in the selected 3D scene. In the model, in the Simulation 3D Scene Configuration block, select the corresponding scene in the **Scene name** parameter.
- **Modify vehicle trajectories** — In the app, modify the vehicle trajectories and resave the scenario. In the model, you do not need to update anything to account for this change. The Scenario Reader block automatically picks up these changes.
- **Modify vehicle appearances** — In the app, update the color and **3D Display Type** parameter of the vehicles. Also make sure that the **3D Display > Use 3D Simulation Actor Dimensions** option is selected. In the model, update the **Color** and **Type** parameters of the corresponding Simulation 3D Vehicle with Ground Following blocks.
- **Add a new vehicle** — In the app, create a new vehicle and specify a trajectory, color, and 3D display type. In the model, add a new Simulation 3D Vehicle with Ground Following block and corresponding Cuboid To 3D Simulation block. Set up these blocks similar to how the existing non-ego vehicles are set up. In the Cuboid To 3D Simulation block, set the `ActorID` of the new vehicle.

- **Set a new ego vehicle** — In the app, on the **Actors** tab, select the vehicle that you want to set as the ego vehicle and click **Set As Ego Vehicle**. In the model, in the Cuboid To 3D Simulation blocks, update the **ActorID used for conversion** parameters to account for which vehicle is the new ego vehicle. In the sensor blocks, set the **Parent name** parameters such that the sensors are mounted to the new ego vehicle.
- **Modify or add sensors** — In the app, you do not need to make any changes. In the model, modify or add sensor blocks. When adding sensor blocks, set the **Parent name** of all sensors to the ego vehicle.

To visualize any updated scenario in the **Bird's-Eye Scope**, you must click **Find Signals** again. If you modify a scenario or are interested in only visualizing sensor data, consider turning off the 3D window during simulation. In the Simulation 3D Scene Configuration block, clear the **Display 3D simulation window** parameter.

See Also

Apps

Bird's-Eye Scope | Driving Scenario Designer

Blocks

Cuboid To 3D Simulation | Scenario Reader | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following | Vehicle To World

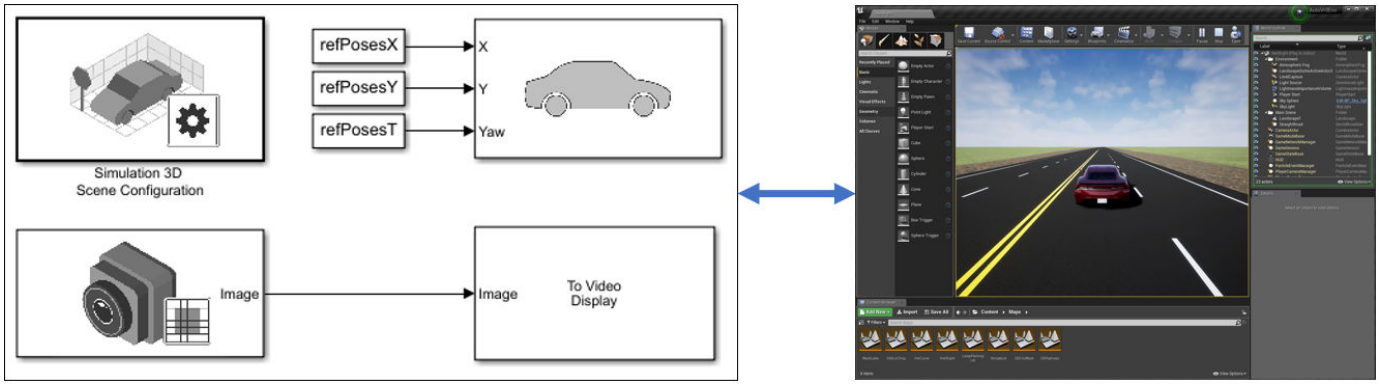
More About

- “Choose a Sensor for Unreal Engine Simulation” on page 6-16
- “Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer” on page 5-62
- “Highway Lane Following” on page 7-653

Customize Unreal Engine Scenes for Automated Driving

Automated Driving Toolbox comes installed with prebuilt scenes in which to simulate and visualize the performance of driving algorithms modeled in Simulink. These scenes are visualized using the Unreal Engine from Epic Games. By using the Unreal Editor and the Automated Driving Toolbox Interface for Unreal Engine 4 Projects, you can customize these scenes. You can also use the Unreal Editor and the support package to simulate within scenes from your own custom project.

With custom scenes, you can co-simulate in both Simulink and the Unreal Editor so that you can modify your scenes between simulation runs. You can also package your scenes into an executable file so that you do not have to open the editor to simulate with these scenes.



To customize Unreal Engine scenes for automated driving, follow these steps:

1. “Install Support Package for Customizing Scenes” on page 6-45
2. “Customize Scenes Using Simulink and Unreal Editor” on page 6-49
3. “Package Custom Scenes into Executable” on page 6-55

See Also

Simulation 3D Scene Configuration

More About

- “Unreal Engine Simulation for Automated Driving” on page 6-2

Install Support Package for Customizing Scenes

To customize scenes in the Unreal Editor and use them in Simulink, you must install the Automated Driving Toolbox Interface for Unreal Engine 4 Projects.

Verify Software and Hardware Requirements

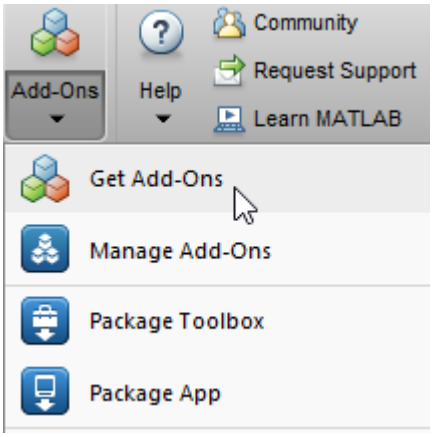
Before installing the support package, make sure that your environment meets the minimum software and hardware requirements described in “Unreal Engine Simulation Environment Requirements and Limitations” on page 6-6. In particular, verify that you have Visual Studio 2017 installed. This software is required for using the Unreal Editor to customize scenes.

In addition, verify that your project is compatible with Unreal Engine, Version 4.23. If your project was created with an older version of the Unreal Editor, upgrade your project to version 4.23.

Install Support Package

To install the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, follow these steps:

1. On the MATLAB **Home** tab, in the **Environment** section, select **Add-Ons** > **Get Add-Ons**.



2. In the Add-On Explorer window, search for the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package. Click **Install**.

Note You must have write permission for the installation folder.

Set Up Scene Customization Using Support Package

The Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package includes these components:

- `AutoVrtlEnv` folder — An Unreal Engine project folder containing the `AutoVrtlEnv.uproject` file and corresponding supporting files. This project contains editable versions of the prebuilt scenes that you can select from the **Scene name** parameter of the Simulation 3D Scene Configuration block.

- `MathWorkSimulation.uplugin` — A plugin file that establishes the connection between Simulink and the Unreal Editor. It is required for co-simulation.
- `RoadRunnerScenes` folder — A folder containing the Unreal Engine project and corresponding executable for a scene that was created by using the RoadRunner scene editing software. This folder contains these subfolders:
  - `RRScene` — An Unreal Engine project folder containing the `RRScene.uproject` file and corresponding supporting files. This project contains an editable version of the scene used in the “Highway Lane Following with RoadRunner Scene” on page 7-736 example.
  - `WindowsPackage` — A folder containing the executable `RRScene.exe` and supporting files. Use this executable to co-simulate the Simulink models explained in the “Highway Lane Following with RoadRunner Scene” on page 7-736 example.

To set up scene customization, you must copy the `AutoVrtlEnv` project and `MathWorksSimulation` plugin onto your local machine. To customize the RoadRunner scene used in the “Highway Lane Following with RoadRunner Scene” on page 7-736 example, you must also copy the `RRScene` project onto your local machine and download the `RoadRunnerMaterials` plugin and copy it into your local project.

Copy AutoVrtlEnv Project to Local Folder

Copy the `AutoVrtlEnv` project folder into a folder on your local machine.

1. Specify the path to the support package folder that contains the project. If you previously downloaded the support package, specify only the latest download path, as shown here. Also specify a local folder destination in which to copy the project. This code specifies the local folder of `C:\Local`.

   ```
   supportPackageFolder = fullfile( ...
       matlabshared.supportpkg.getSupportPackageRoot, ...
       "toolbox","shared","sim3dprojects","driving");
   localFolder = "C:\Local";
   ```

2. Copy the `AutoVrtlEnv` project from the support package folder to the local destination folder.

   ```
   projectFolderName = "AutoVrtlEnv";
   projectSupportPackageFolder = fullfile(supportPackageFolder,projectFolderName);
   projectLocalFolder = fullfile(localFolder,projectFolderName);
   if ~exist(projectLocalFolder,"dir")
       copyfile(projectSupportPackageFolder,projectLocalFolder);
   end
   ```

   The `AutoVrtlEnv.uproject` file and all of its supporting files are now located in a folder named `AutoVrtlEnv` within the specified local folder. For example: `C:\Local\AutoVrtlEnv`.

Copy MathWorksSimulation Plugin to Unreal Editor

Copy the `MathWorksSimulation` plugin into the `Plugins` folder of your Unreal Engine installation.

1. Specify the local folder containing your Unreal Engine installation. This code shows the default installation location for the editor on a Windows machine.

   ```
   ueInstallFolder = "C:\Program Files\Epic Games\UE_4.23";
   ```

2. Copy the plugin from the support package into the `Plugins` folder.

   ```
   mwPluginName = "MathWorksSimulation.uplugin";
   mwPluginFolder = fullfile(supportPackageFolder,"PluginResources","UE423");
   ```

```
uePluginFolder = fullfile(ueInstallFolder,"Engine","Plugins");
uePluginDestination = fullfile(uePluginFolder,"Marketplace","MathWorks");

cd(uePluginFolder)
foundPlugins = dir("**/" + mwPluginName);

if ~isempty(foundPlugins)
    numPlugins = size(foundPlugins,1);
    msg2 = cell(1,numPlugins);
    pluginCell = struct2cell(foundPlugins);

    msg1 = "Plugin(s) already exist here:" + newline + newline;
    for n = 1:numPlugins
        msg2{n} = "    " + pluginCell{2,n} + newline;
    end
    msg3 = newline + "Please remove plugin folder(s) and try again.";
    msg = msg1 + msg2 + msg3;
    warning(msg);
else
    copyfile(mwPluginFolder,uePluginDestination);
    disp("Successfully copied MathWorksSimulation plugin to UE4 engine plugins!")
end
```

(Optional) Copy RRScene Project to Local Folder

To customize the scene in the `RRScene` project folder, copy the project onto your local machine.

1. Specify the path to the support package folder that contains the project. Also specify a local folder destination to copy the project. This code uses the support package path and local folder path from previous sections.

   ```
   rrProjectSupportPackageFolder = fullfile(supportPackageFolder, ...
       "RoadRunnerScenes","RRScene");
   rrProjectLocalFolder = fullfile(localFolder,"RRScene");
   ```

2. Copy the `RRScene` project from the support package folder to the local destination folder.

   ```
   if ~exist(rrProjectLocalFolder,"dir")
       copyfile(rrProjectSupportPackageFolder,rrProjectLocalFolder);
   end
   ```

The `RRScene.uproject` file and all of its supporting files are now located in a folder named `RRScene` within the specified local folder. For example: `C:\Local\RRScene`.

(Optional) Copy RoadRunnerMaterials Plugin to Unreal Editor

When customizing the scene in the `RRScene` project folder, you must also copy the `RoadRunnerMaterials` plugin to your plugin project folder.

1. Download the ZIP file containing the latest RoadRunner plugins. See “Downloading Plugins” (RoadRunner). Extract the contents of the ZIP file to your local machine. The extracted folder name is of the form `RoadRunner Plugins x.x.x`, where `x.x.x` is the plugin version number.
2. Specify the path to the `RoadRunnerMaterials` plugin. This plugin is located in the `Unreal/Plugins` folder of the extracted folder. Update this code to match the location where you downloaded the plugin and the plugin version number.

   ```
   rrMaterialsPluginFolder = fullfile("C:","Local","RoadRunner Plugins 1.0.3", ...
       "Unreal","Plugins","RoadRunnerMaterials");
   ```

3. In your local `RRScene` project, create a `Plugins` folder in which to copy the plugin. This code uses the path to the local `RRScene` project specified in the previous section.

```
rrProjectPluginFolder = fullfile(rrProjectLocalFolder,"Plugins","RoadRunnerMaterials");
```

4. Copy the `RoadRunnerMaterials` plugin to the `Plugins` folder of your local project.

```
copyStatus = copyfile(rrMaterialsPluginFolder,rrProjectPluginFolder);
if copyStatus
    disp("Successfully copied RoadRunnerMaterials plugin to RRScene project plugins folder.")
else
    disp("Unable to copy RoadRunnerMaterials plugin to RRScene project plugins folder.")
end
```

After you install and set up the support package, you can begin customizing scenes. See “Customize Scenes Using Simulink and Unreal Editor” on page 6-49.

See Also

More About

- “Unreal Engine Simulation for Automated Driving” on page 6-2
- “Unreal Engine Simulation Environment Requirements and Limitations” on page 6-6

Customize Scenes Using Simulink and Unreal Editor

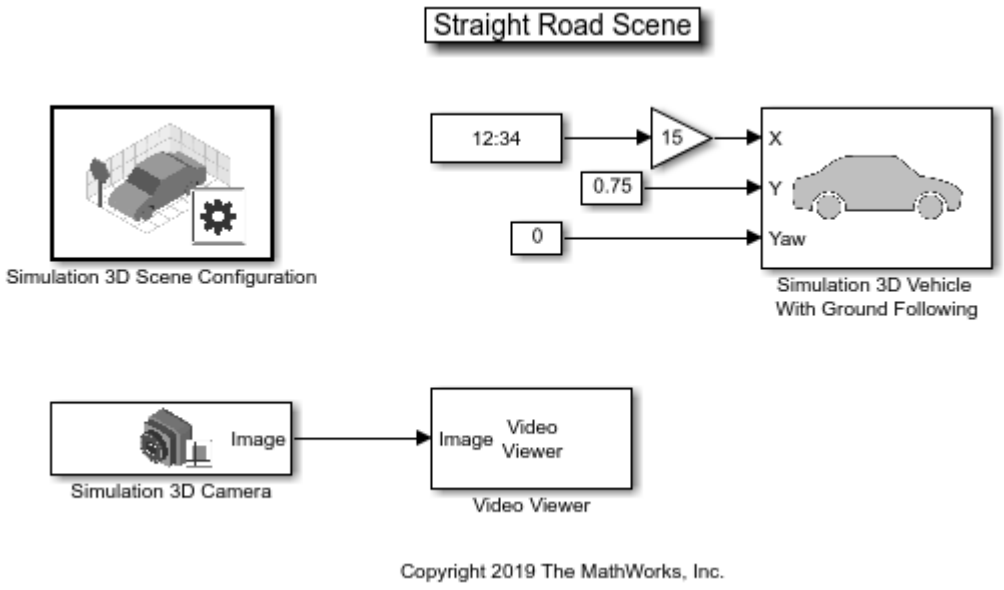
After you install the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package as described in “Install Support Package for Customizing Scenes” on page 6-45, you can simulate in custom scenes simultaneously from both the Unreal Editor and Simulink. By using this co-simulation framework, you can add vehicles and sensors to a Simulink model and then run this simulation in your custom scene.

Open Unreal Editor from Simulink

If you open your Unreal project file directly in the Unreal Editor, Simulink is unable to establish a connection with the editor. To establish this connection, you must open your project from a Simulink model.

1. Open a Simulink model configured to simulate in the Unreal Engine environment. At a minimum, the model must contain a Simulation 3D Scene Configuration block. For example, open a simple model that simulates a vehicle driving on a straight highway. This model is used in the “Design Lane Marker Detector Using Unreal Engine Simulation Environment” on page 7-617 example.

   ```
   openExample('driving/VisualPerceptionIn3DSimulationExample')
   open_system('straightRoadSim3D')
   ```



Copyright 2019 The MathWorks, Inc.

2. In the Simulation 3D Scene Configuration block of this model, set the **Scene source** parameter to `Unreal Editor`.
3. In the **Project** parameter, browse for the project file that contains the scenes that you want to customize.

   For example, this sample path specifies the `AutoVrtlEnv` project that comes installed with the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package.

   ```
   C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject
   ```

   This sample path specifies a custom project.

```
Z:\UnrealProjects\myProject\myProject.uproject
```

4 Click **Open Unreal Editor**. The Unreal Editor opens and loads a scene from your project.

The first time that you open the Unreal Editor from Simulink, you might be asked to rebuild `UE4Editor` DLL files or the `AutoVrtlEnv` module. Click **Yes** to rebuild these files or modules. The editor also prompts you that new plugins are available. Click **Manage Plugins** and verify that the **MathWorks Interface** plugin is installed. This plugin is the `MathWorksSimulation.uplugin` file that you copied into your Unreal Editor installation in “Install Support Package for Customizing Scenes” on page 6-45.

When the editor opens, you can ignore any warning messages about files with the name `'_BuiltData'` that failed to load.

If you receive a warning that the lighting needs to be rebuilt, from the toolbar above the editor window, select **Build** > **Build Lighting Only**. The editor issues this warning the first time you open a scene or when you add new elements to a scene.

Reparent Actor Blueprint

Note If you are using a scene from the `AutoVtrlEnv` or `RRScene` project that is part of the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, skip this section. However, if you create a new scene based off of one of the scenes in this project, then you must complete this section.

The first time that you open a custom scene from Simulink, you need to associate, or reparent, this project with the **Sim3dLevelScriptActor** level blueprint used in Automated Driving Toolbox. The level blueprint controls how objects interact with the Unreal Engine environment once they are placed in it. Simulink returns an error at the start of simulation if the project is not reparented. You must reparent each scene in a custom project separately.

To reparent the level blueprint, follow these steps:

1 In the Unreal Editor toolbar, select **Blueprints** > **Open Level Blueprint**.

2 In the Level Blueprint window, select **File** > **Reparent Blueprint**.

3 Click the **Sim3dLevelScriptActor** blueprint. If you do not see the **Sim3dLevelScriptActor** blueprint listed, use these steps to check that you have the `MathWorksSimulation` plugin installed and enabled:

   a In the Unreal Editor toolbar, select **Settings** > **Plugins**.

   b In the Plugins window, verify that the **MathWorks Interface** plugin is listed in the installed window. If the plugin is not already enabled, select the **Enabled** check box.

      If you do not see the **MathWorks Interface** plugin in this window, repeat the steps under “Copy MathWorksSimulation Plugin to Unreal Editor” on page 6-46 and reopen the editor from Simulink.

   c Close the editor and reopen it from Simulink.

4 Close the Level Blueprint window.

Create or Modify Scenes in Unreal Editor

After you open the editor from Simulink, you can modify the scenes in your project or create new scenes.

Open Scene

In the Unreal Editor, scenes within a project are referred to as levels. Levels come in several types, and scenes have a level type of map.

- To open a prebuilt scene from the `AutoVrtlEnv.uproject` or `RRScene.uproject` file, in the **Content Browser** pane below the editor window, navigate to the **Content** > **Maps** folder. Then, select the map that corresponds to the scene you want to modify.

  This table shows the map names in the `AutoVrtlEnv` project as they appear in the Unreal Editor. It also shows their corresponding scene names as they appear in the **Scene name** parameter of the Simulation 3D Scene Configuration block.

Unreal Editor Map	Automated Driving Toolbox Scene
`HwCurve`	**Curved Road**
`DblLnChng`	**Double Lane Change**
`BlackLake`	**Open Surface**
`LargeParkingLot`	**Large Parking Lot**
`SimpleLot`	**Parking Lot**
`HwStrght`	**Straight Road**
`USCityBlock`	**US City Block**
`USHighway`	**US Highway**

  Note The `AutoVrtlEnv.uproject` file does not include the **Virtual Mcity** scene.

  The `RRScene` project contains only one scene: `RRHighway`. This scene is used in the "Highway Lane Following with RoadRunner Scene" on page 7-736 example and is not selectable from the **Scene name** parameter of the Simulation 3D Scene Configuration block.
- To open a scene within your own project, in the **Content Browser** pane, navigate to the folder that contains your scenes.

Create New Scene

To create a new scene in your project, from the top-left menu of the editor, select **File** > **New Level**.

Alternatively, you can create a new scene from an existing one. This technique is useful, for example, if you want to use one of the prebuilt scenes in the `AutoVtrlEnv` project as a starting point for creating your own scene. To save a version of the currently opened scene to your project, from the top-left menu of the editor, select **File** > **Save Current As**. The new scene is saved to the same location as the existing scene.

Specify Vehicle Trajectories

In your scenes, you can specify trajectory waypoints that the vehicles in your scene can follow.

- If your scene is based off one of the prebuilt scenes in the `AutoVrtlEnv` project, then specify waypoints using the process described in the "Select Waypoints for Unreal Engine Simulation" on page 7-626 example. This example shows how to interactively draw waypoints on 2-D top-down maps of the prebuilt scenes.
- If your is not based off of a prebuilt scene, then before using the "Select Waypoints for Unreal Engine Simulation" on page 7-626 example, you must first generate a map of your scene. See "Create Top-Down Map of Unreal Engine Scene" on page 6-63.

Add Assets to Scene

In the Unreal Editor, elements within a scene are referred to as assets. To add assets to a scene, you can browse or search for them in the **Content Browser** pane at the bottom and drag them into the editor window.

When adding assets to a scene that is in the `AutoVrtlEnv` project, you can choose from a library of driving-related assets. These assets are built as static meshes and begin with the prefix `SM_`. Search for these objects in the **Content Browser** pane.

For example, add a stop sign to a scene in the `AutoVrtlEnv` project.

1. In the **Content Browser** pane at the bottom of the editor, navigate to the **Content** folder.
2. In the search bar, search for `SM_StopSign`. Drag the stop sign from the **Content Browser** into the editing window. You can then change the position of the stop sign in the editing window or on the **Details** pane on the right, in the **Transform** section.

The Unreal Editor uses a left-hand *Z*-up coordinate system, where the *Y*-axis points to the right. Automated Driving Toolbox uses a right-hand *Z*-up coordinate system, where the *Y*-axis points to the left. When positioning objects in a scene, keep this coordinate system difference in mind. In the two coordinate systems, the positive and negative signs for the *Y*-axis and pitch angle values are reversed.

For more information on modifying scenes and adding assets, see Unreal Engine 4 Documentation.

To migrate assets from the `AutoVrtlEnv` scene into the `RRScene` project or to your own project, see Migrating Assets in the Unreal Engine documentation.

To obtain semantic segmentation data from a scene, then you must apply stencil IDs to the objects added to a scene. For more information, see "Apply Semantic Segmentation Labels to Custom Scenes" on page 6-57.

Run Simulation

Verify that the Simulink model and Unreal Editor are configured to co-simulate by running a test simulation.

1. In the Simulink model, click **Run**.

   Because the source of the scenes is the project opened in the Unreal Editor, the simulation does not start. Instead, you must start the simulation from the editor.
2. Verify that the Diagnostic Viewer window in Simulink displays this message:

```
In the Simulation 3D Scene Configuration block, you set the scene source
to 'Unreal Editor'. In Unreal Editor, select 'Play' to view the scene.
```

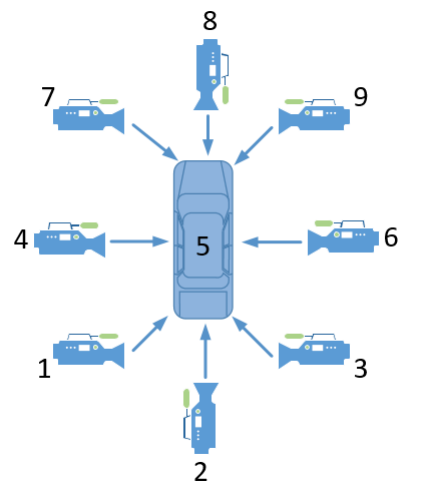
This message confirms that Simulink has instantiated vehicles and other objects in the Unreal Engine environment.

3. In the Unreal Editor, click **Play**. The simulation runs in the scene currently open in the Unreal Editor.

   - If your Simulink model contains vehicles, these vehicles drive through the scene that is open in the editor.
   - If your Simulink model includes sensors, these sensors capture data from the scene that is open in the editor.

To control the view of the scene during simulation, in the Simulation 3D Scene Configuration block, select the vehicle name from the **Scene view** parameter. To change the scene view as the simulation runs, use the numeric keypad in the editor. The table shows the position of the camera displaying the scene, relative to the vehicle selected in the **Scene view** parameter.

Key	Camera View
1	Back left
2	Back
3	Back right
4	Left
5	Internal
6	Right
7	Front left
8	Front
9	Front right
0	Overhead



To restart a simulation, click **Run** in the Simulink model, wait until the Diagnostic Viewer displays the confirmation message, and then click **Play** in the editor. If you click **Play** before starting the simulation in your model, the connection between Simulink and the Unreal Editor is not established, and the editor displays an empty scene.

If you are co-simulating a custom project, to enable the numeric keypad, copy the `DefaultInput.ini` file from the support package installation folder to your custom project folder. For example, copy `DefaultInput.ini` from:

```
C:\ProgramData\MATLAB\SupportPackages\<MATLABRelease>\toolbox\shared\sim3dprojects\driving\AutoV
```

to:

```
C:\<yourproject>.project\Config
```

After tuning your custom scene based on simulation results, you can then package the scene into an executable. For more details, see “Package Custom Scenes into Executable” on page 6-55.

See Also

Simulation 3D Scene Configuration

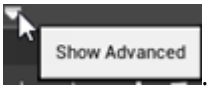
More About

- “Apply Semantic Segmentation Labels to Custom Scenes” on page 6-57
- “Create Top-Down Map of Unreal Engine Scene” on page 6-63
- “Select Waypoints for Unreal Engine Simulation” on page 7-626

Package Custom Scenes into Executable

When you finish modifying a custom scene, you can package the project file containing this scene into an executable. You can then configure your model to simulate from this executable by using the Simulation 3D Scene Configuration block. Executable files can improve simulation performance and do not require opening the Unreal Editor to simulate your scene. Instead, the scene runs by using the Unreal Engine that comes installed with Automated Driving Toolbox.

Package Scene into Executable Using Unreal Editor

1. Open the project containing the scene in the Unreal Editor. You must open the project from a Simulink model that is configured to co-simulate with the Unreal Editor. For more details on this configuration, see “Customize Scenes Using Simulink and Unreal Editor” on page 6-49.
2. In the Unreal Editor toolbar, select **Settings** > **Project Settings** to open the Project Settings window.
3. In the left pane, in the **Project** section, click **Packaging**.
4. In the **Packaging** section, set or verify the options in the table. If you do not see all these options, at the bottom of the **Packaging** section, click the **Show Advanced** expander 

.

Packaging Option	Enable or Disable
Use Pak File	Enable
Cook everything in the project content directory (ignore list of maps below)	Disable
Cook only maps (this only affects cookall)	Enable
Create compressed cooked packages	Enable
Exclude editor content while cooking	Enable

5. Specify the scene from the project that you want to package into an executable.
   a. In the **List of maps to include in a packaged build** option, click the **Adds Element** button .
   b. Specify the path to the scene that you want to include in the executable. By default, the Unreal Editor saves maps to the `/Game/Maps` folder. For example, if the `/Game/Maps` folder has a scene named `myScene` that you want to include in the executable, enter `/Game/Maps/myScene`.
   c. Add or remove additional scenes as needed.
6. Rebuild the lighting in your scenes. If you do not rebuild the lighting, the shadows from the light source in your executable file are incorrect and a warning about rebuilding the lighting displays during simulation. In the Unreal Editor toolbar, select **Build** > **Build Lighting Only**.
7. (Optional) If you plan to obtain semantic segmentation data from the scene by using a Simulation 3D Camera block, enable rendering of the stencil IDs. In the left pane, in the **Engine** section, click **Rendering**. Then, in the main window, in the **Postprocessing** section, set **Custom Depth-Stencil Pass** to `Enabled with Stencil`. For more details on applying stencil IDs for semantic segmentation, see “Apply Semantic Segmentation Labels to Custom Scenes” on page 6-57.

8. Close the **Project Settings** window.
9. In the top-left menu of the editor, select **File > Package Project > Windows > Windows (64-bit)**. Select a local folder in which to save the executable, such as to the root of the project file (for example, `C:/Local/myProject`).

   Note Packaging a project into an executable can take several minutes. The more scenes that you include in the executable, the longer the packaging takes.

   Once packaging is complete, the folder where you saved the package contains a `WindowsNoEditor` folder that includes the executable file. This file has the same name as the project file.

   Note If you repackage a project into the same folder, the new executable folder overwrites the old one.

   Suppose you package a scene that is from the `myProject.uproject` file and save the executable to the `C:/Local/myProject` folder. The editor creates a file named `myProject.exe` with this path:

   ```
   C:/Local/myProject/WindowsNoEditor/myProject.exe
   ```

Simulate Scene from Executable in Simulink

1. In the Simulation 3D Scene Configuration block of your Simulink model, set the **Scene source** parameter to `Unreal Executable`.
2. Set the **File name** parameter to the name of your Unreal Editor executable file. You can either browse for the file or specify the full path to the file by using backslashes. For example:

   ```
   C:\Local\myProject\WindowsNoEditor\myProject.exe
   ```

3. Set the **Scene** parameter to the name of a scene from within the executable file. For example:

   ```
   /Game/Maps/myScene
   ```

4. Run the simulation. The model simulates in the custom scene that you created.

If you are simulating a scene from a project that is not based on the `AutoVtrlEnv` project, then the scene simulates in full screen mode. To use the same window size as the default scenes, copy the `DefaultGameUserSettings.ini` file from the support package installation folder to your custom project folder. For example, copy `DefaultGameUserSettings.ini` from:

```
C:\ProgramData\MATLAB\SupportPackages\<MATLABrelease>\toolbox\shared\sim3dprojects\driving\AutoV
```

to:

```
C:\<yourproject>.project\Config
```

Then, package scenes from the project into an executable again and retry the simulation.

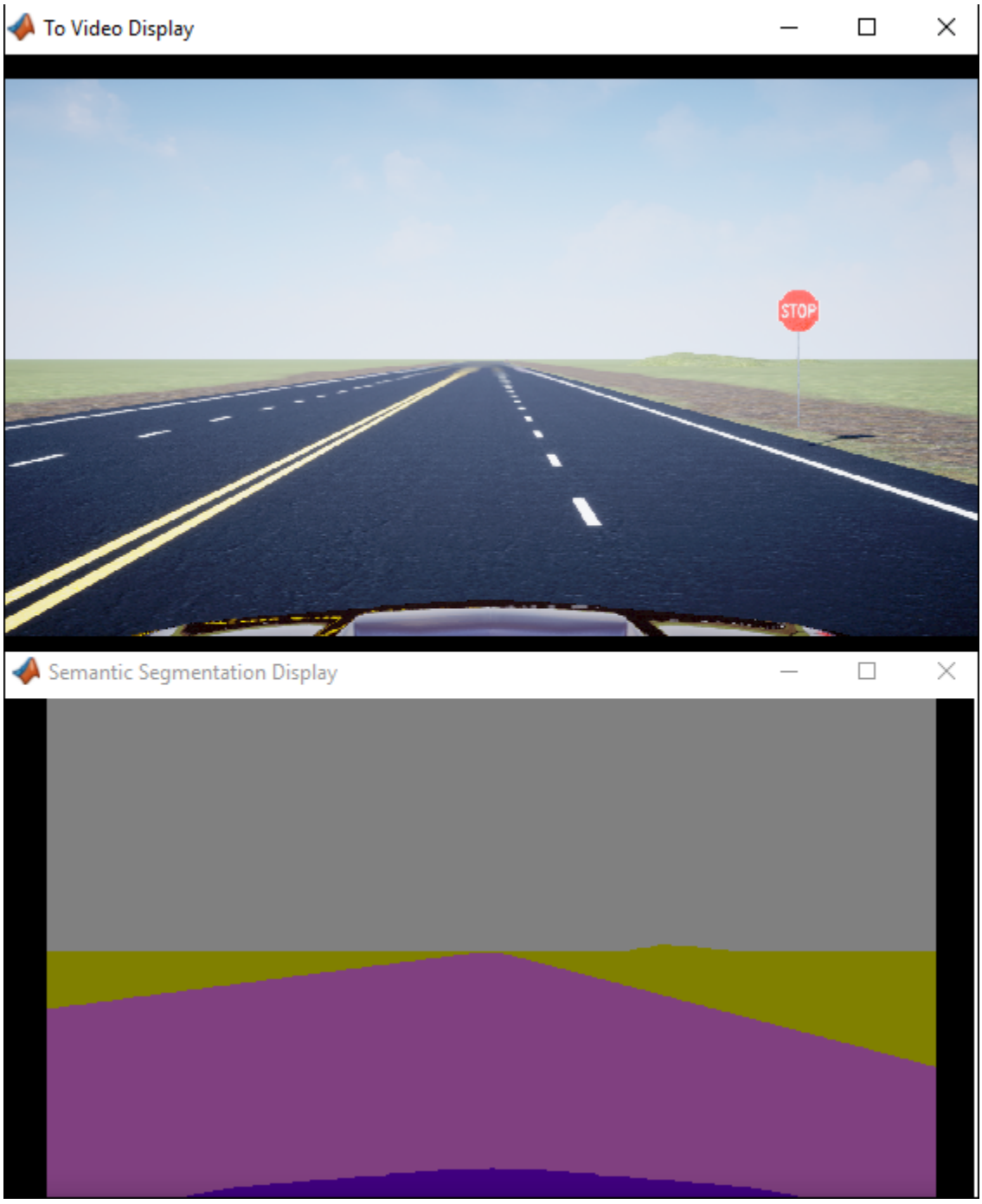
See Also

Simulation 3D Scene Configuration

Apply Semantic Segmentation Labels to Custom Scenes

The Simulation 3D Camera block provides an option to output semantic segmentation data from a scene. If you add new scene elements, or assets (such as traffic signs or roads), to a custom scene, then in the Unreal Editor, you must apply the correct ID to that element. This ID is known as a stencil ID. Without the correct stencil ID applied, the Simulation 3D Camera block does not recognize the scene element and does not display semantic segmentation data for it.

For example, this To Video Display window shows a stop sign that was added to a custom scene. The Semantic Segmentation Display window does not display the stop sign, because the stop sign is missing a stencil ID.



To apply a stencil ID label to a scene element, follow these steps:

1. Open the Unreal Editor from a Simulink model that is configured to simulate in the 3D environment. For more details, see “Customize Scenes Using Simulink and Unreal Editor” on page 6-49.
2. In the editor window, select the scene element with the missing stencil ID.
3. On the **Details** pane on the right, in the **Rendering** section, select **Render CustomDepth Pass**.

If you do not see this option, click the **Show Advanced** expander 

to show all rendering options.

4 In the **CustomDepth Stencil Value** box, enter the stencil ID that corresponds to the asset. If you are adding an asset to a scene from the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, then enter the stencil ID corresponding to that asset type, as shown in the table. If you are adding assets other than the ones shown, then you can assign them to unused IDs. If you do not assign a stencil ID to an asset, then the Unreal Editor assigns that asset an ID of `0`.

Note The Simulation 3D Camera block does not support the output of semantic segmentation data for lane markings. Even if you assign a stencil ID to lane markings, the block ignores this setting.

ID	Type
`0`	None/default
`1`	Building
`2`	*Not used*
`3`	Other
`4`	*Not used*
`5`	Pole
`6`	*Not used*
`7`	Road
`8`	Sidewalk
`9`	Vegetation
`10`	Vehicle
`11`	*Not used*
`12`	Generic traffic sign
`13`	Stop sign
`14`	Yield sign
`15`	Speed limit sign
`16`	Weight limit sign
`17-18`	*Not used*
`19`	Left and right arrow warning sign
`20`	Left chevron warning sign
`21`	Right chevron warning sign
`22`	*Not used*
`23`	Right one-way sign
`24`	*Not used*

ID	Type
25	School bus only sign
26-38	*Not used*
39	Crosswalk sign
40	*Not used*
41	Traffic signal
42	Curve right warning sign
43	Curve left warning sign
44	Up right arrow warning sign
45-47	*Not used*
48	Railroad crossing sign
49	Street sign
50	Roundabout warning sign
51	Fire hydrant
52	Exit sign
53	Bike lane sign
54-56	*Not used*
57	Sky
58	Curb
59	Flyover ramp
60	Road guard rail
61-66	*Not used*
67	Deer
68-70	*Not used*
71	Barricade
72	Motorcycle
73-255	*Not used*

For example, for a stop sign that is missing a stencil ID, enter 13.

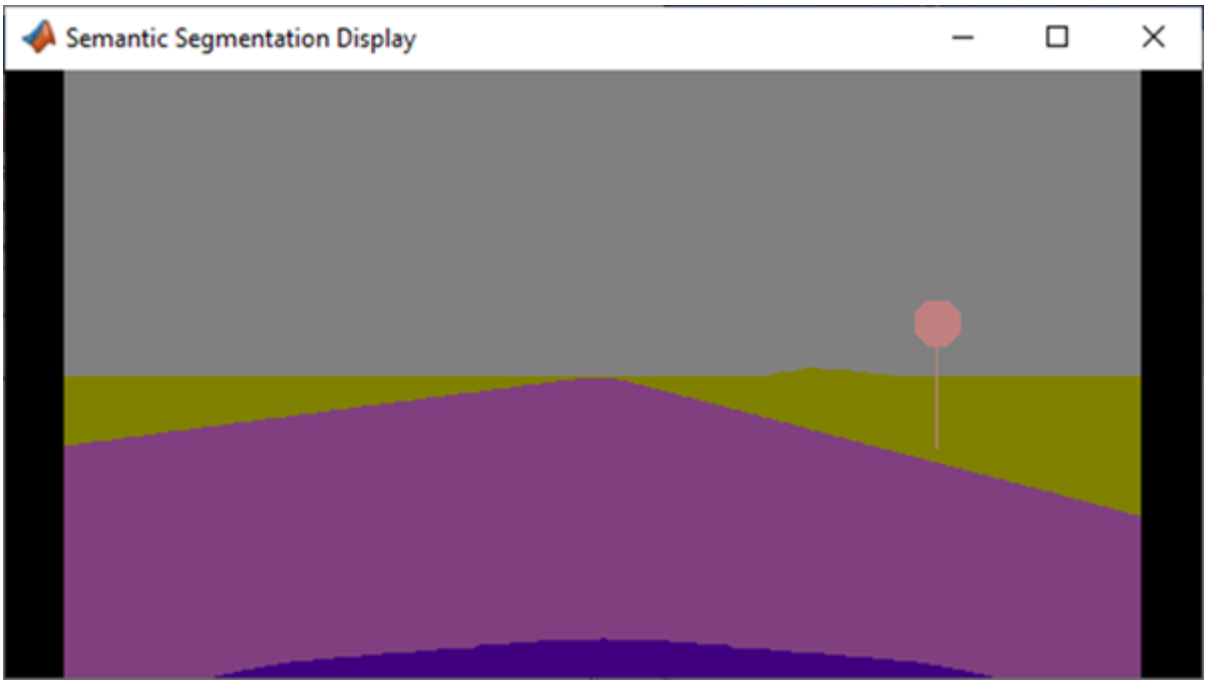
Tip If you are adding stencil ID for scene elements of the same type, you can copy **(Ctrl+C)** and paste **(Ctrl+V)** the element with the added stencil ID. The copied scene element includes the stencil ID.

5 Visually verify that the correct stencil ID shows by using the custom stencil view. In the top-left corner of the editor window, click 

and select **Buffer Visualization > Custom Stencil**. The scene displays the stencil IDs specified for each scene element. For example, if you added the correct stencil ID to a stop sign (13) then the editor window, the stop sign displays a stencil ID value of 13.

- If you did not set a stencil ID value for a scene element, then the element appears in black and displays no stencil ID.
- If you did not select **CustomDepth Stencil Value**, then the scene element does not appear at all in this view.

6. Turn off the custom stencil ID view. In the top-left corner of the editor window, click **Buffer Visualization** and then select **Lit**.
7. If you have not already done so, set up your Simulink model to display semantic segmentation data from a Simulation 3D Camera block. For an example setup, see “Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation” on page 6-31.
8. Run the simulation and verify that the Simulation 3D Camera block outputs the correct data. For example, here is the Semantic Segmentation Display window with the correct stencil ID applied to a stop sign.



See Also

Simulation 3D Camera | Simulation 3D Scene Configuration

More About

- “Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation” on page 6-31
- “Customize Scenes Using Simulink and Unreal Editor” on page 6-49

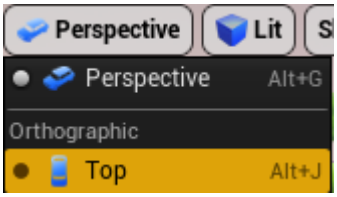
Create Top-Down Map of Unreal Engine Scene

3-D scenes developed for the Unreal Engine simulation environment can be large and complex. Using the Unreal Editor, you can create a 2-D, top-down map of the scene to get a big-picture view of your scene. You can also use this map to select waypoints of vehicles traveling along a path in your scene.

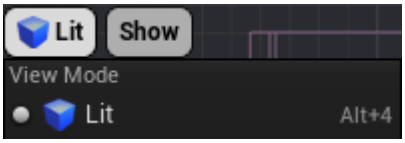
Capture Screenshot

To create your 2-D map, first capture a high-resolution screenshot of your 3-D scene from a top-down perspective.

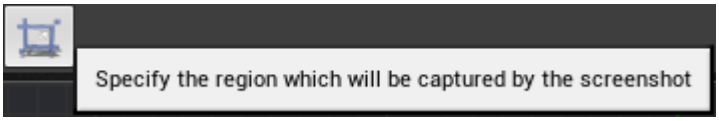
1. Open the Unreal Editor from a Simulink model that is configured to co-simulate with the Unreal Editor. For more details, see “Customize Scenes Using Simulink and Unreal Editor” on page 6-49.
2. Open your scene in the Unreal Editor.
3. Switch to a top-down view of the scene. In the top-left corner of the editing window, click **Perspective**, and then click **Top**.



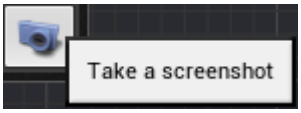
4. Verify that the scene is lit by the standard lighting. In the top-left corner of the editing window, click **Lit**.



5. Open the control panel for taking high-resolution screenshots of the scene. The screenshot acts as a 2-D scene map. In the top-left corner of the editing window, click the down arrow and select **High Resolution Screenshot**.
6. In the left corner of the control panel, click **Specify the region which will be captured by the screenshot**.



7. Manually select a region of the scene, and then click **Take a screenshot**.



The Unreal Editor displays a message that the screenshot is saved to a folder in your project. Click the folder location to access the image file. The folder containing screenshots is a path similar to this path:

```
myProject\Saved\Screenshots\Windows
```

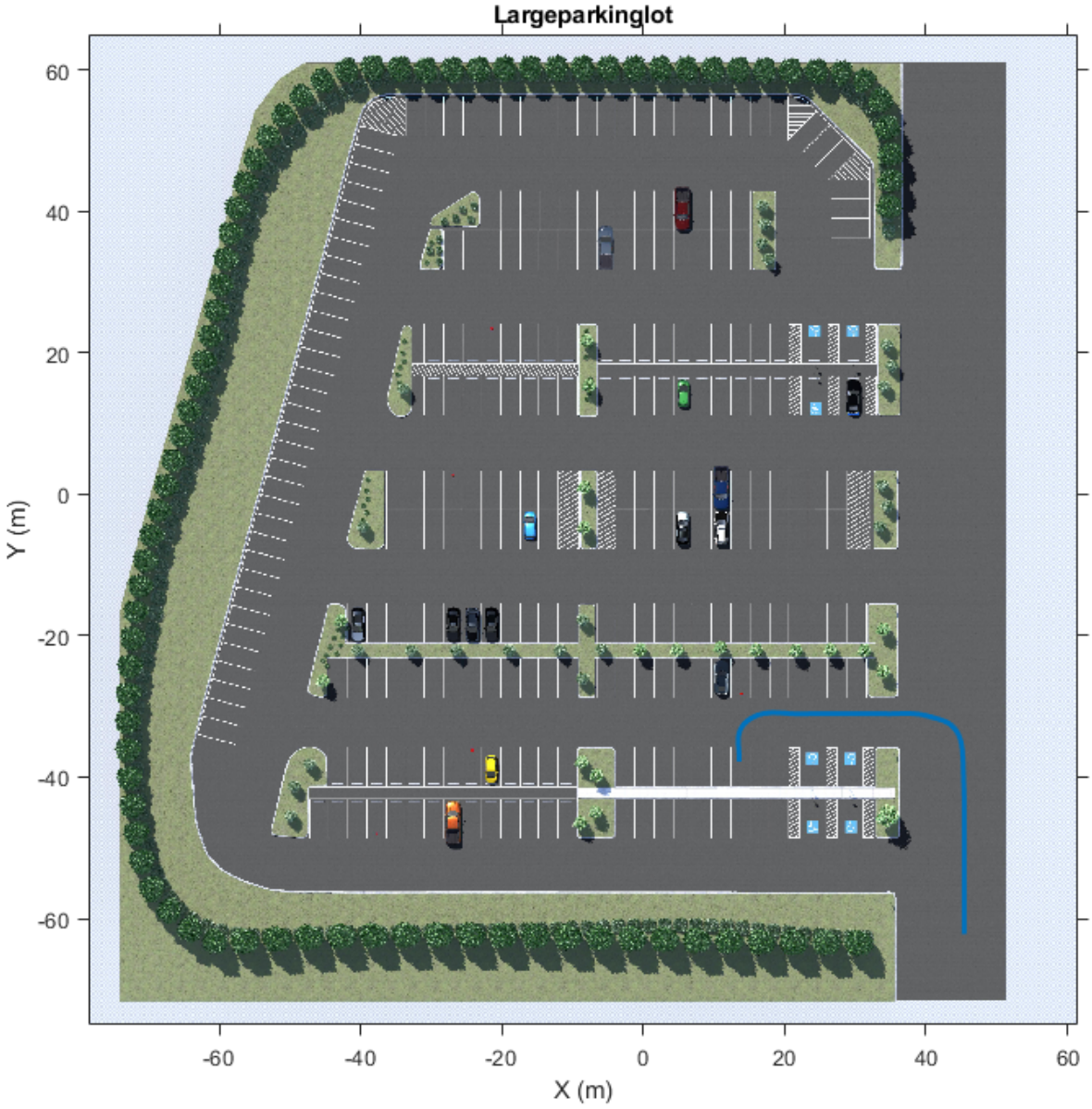
Convert Screenshot to Map

After you create your high-resolution screenshot, you can convert it to a map by creating a 2-D spatial referencing object, `imref2d`. This object describes the relationship between the pixels in the image and the world coordinates of the scene. To use this object to create a map, you must know the *X*-axis and *Y*-axis limits of the scene in world coordinates. For example, in this code, the scene captured by image `myScene.png` has *X*-coordinates of –80 to 60 meters and *Y*-coordinates of –75 to 65 meters.

```
sceneImage = imread('myScene.png');
imageSize = size(sceneImage);
xlims = [-80 60]; % in meters
ylims = [-75 65]; % in meters

sceneRef = imref2d(imageSize,xlims,ylims);
```

You can use the scene image and spatial referencing object to select waypoints for vehicles to follow in your scene. For details on this process, see the “Select Waypoints for Unreal Engine Simulation” on page 7-626 example. This code shows helper function calls in that example. These function calls enable you to display your scene and interactively specify waypoints for vehicles to follow. The image shows a sample map and drawn waypoints in blue that are from the example.

```
helperShowSceneImage(sceneImage, sceneRef);
hFig = helperSelectSceneWaypoints(sceneImage, sceneRef);
```



See Also

`imref2d`

More About

- “Select Waypoints for Unreal Engine Simulation” on page 7-626
- “Customize Scenes Using Simulink and Unreal Editor” on page 6-49

Featured Examples

Configure Monocular Fisheye Camera

This example shows how to convert a fisheye camera model to a pinhole model and construct a corresponding monocular camera sensor simulation. In this example, you learn how to calibrate a fisheye camera and configure a `monoCamera` object.

Overview

To simulate a monocular camera sensor mounted in a vehicle, follow these steps:

1. Estimate the intrinsic camera parameters by calibrating the camera using a checkerboard. The intrinsic parameters describe the properties of the fisheye camera itself.
2. Estimate the extrinsic camera parameters by calibrating the camera again, using the same checkerboard from the previous step. The extrinsic parameters describe the mounting position of the fisheye camera in the vehicle coordinate system.
3. Remove image distortion by converting the fisheye camera intrinsics to pinhole camera intrinsics. These intrinsics describe a synthetic pinhole camera that can hypothetically generate undistorted images.
4. Use the intrinsic pinhole camera parameters and the extrinsic parameters to configure the monocular camera sensor for simulation. You can then use this sensor to detect objects and lane boundaries.

Estimate Fisheye Camera Intrinsics

To estimate the intrinsic parameters, use a checkerboard for camera calibration. Alternatively, to better visualize the results, use the Camera Calibrator (Computer Vision Toolbox) app. For fisheye camera, it is useful to place the checkerboard close to the camera, in order to capture large noticeable distortion in the image.

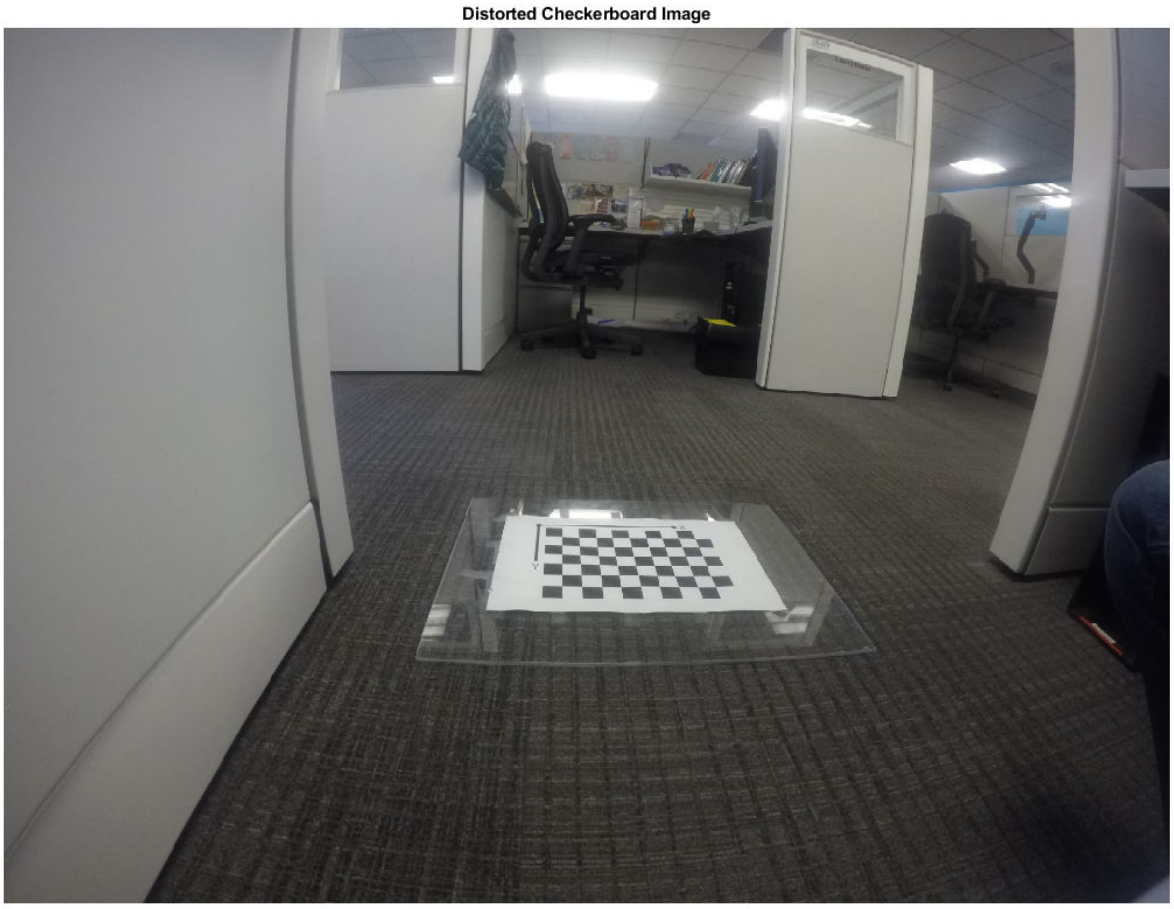
```
 % Gather a set of calibration images.
images = imageDatastore(fullfile(toolboxdir('vision'), 'visiondata', ...
      'calibration', 'gopro'));
imageFileNames = images.Files;

% Detect calibration pattern.
[imagePoints, boardSize] = detectCheckerboardPoints(imageFileNames);

% Generate world coordinates of the corners of the squares.
squareSize = 0.029; % Square size in meters
worldPoints = generateCheckerboardPoints(boardSize, squareSize);

% Calibrate the camera.
I = readimage(images, 1);
imageSize = [size(I, 1), size(I, 2)];
params = estimateFisheyeParameters(imagePoints, worldPoints, imageSize);
```

Estimate Fisheye Camera Extrinsics

To estimate the extrinsic parameters, use the same checkerboard to estimate the mounting position of the camera in the vehicle coordinate system. The following step estimates the parameters from one image. You can also take multiple checkerboard images to obtain multiple estimations, and average the results.

```
% Load a different image of the same checkerboard, where the checkerboard
% is placed on the flat ground. Its X-axis is pointing to the right of the
```

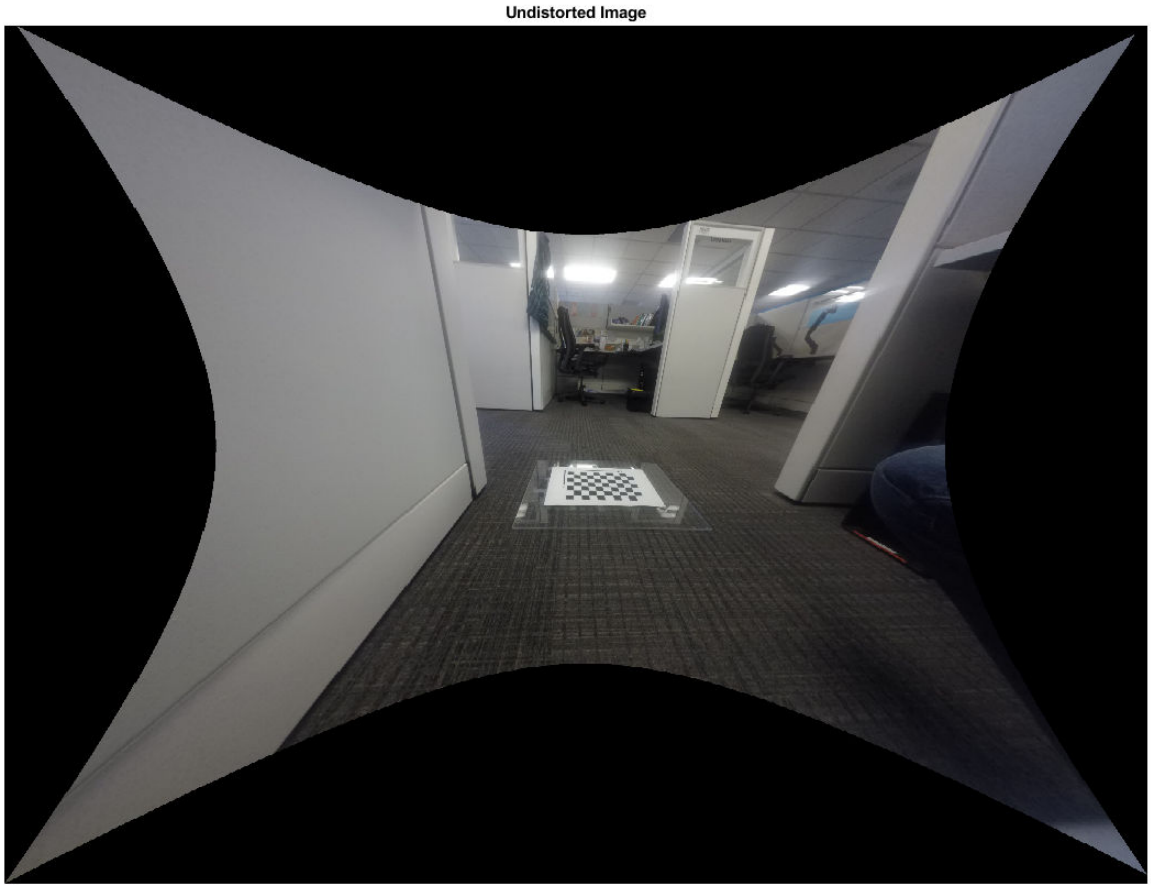
```
% vehicle, and its Y-axis is pointing to the camera. The image includes
% noticeable distortion, such as along the wall next to the checkerboard.

imageFileName = fullfile(toolboxdir('driving'), 'drivingdata', 'checkerboard.png');
I = imread(imageFileName);
imshow(I)
title('Distorted Checkerboard Image');
```




```
[imagePoints, boardSize] = detectCheckerboardPoints(I);

% Generate coordinates of the corners of the squares.
squareSize = 0.029; % Square size in meters
worldPoints = generateCheckerboardPoints(boardSize, squareSize);

% Estimate the parameters for configuring the monoCamera object.
% Height of the checkerboard is zero here, since the pattern is
% directly on the ground.
originHeight = 0;
[pitch, yaw, roll, height] = estimateMonoCameraParameters(params.Intrinsics, ...
                                imagePoints, worldPoints, originHeight);
```

Construct a Synthetic Pinhole Camera for the Undistorted Image

```
% Undistort the image and extract the synthetic pinhole camera intrinsics.
[J1, camIntrinsics] = undistortFisheyeImage(I, params.Intrinsics, 'Output', 'full');
imshow(J1)
title('Undistorted Image');
```




```
% Set up monoCamera with the synthetic pinhole camera intrinsics.
% Note that the synthetic camera has removed the distortion.
sensor = monoCamera(camIntrinsics, height, 'pitch', pitch, 'yaw', yaw, 'roll', roll);
```

Plot Bird's Eye View

Now you can validate the `monoCamera` by plotting a bird's-eye view.

```
% Define bird's-eye-view transformation parameters
distAheadOfSensor = 6; % in meters
spaceToOneSide = 2.5;  % look 2.5 meters to the right and 2.5 meters to the left
bottomOffset = 0.2;  % look 0.2 meters ahead of the sensor
outView = [bottomOffset, distAheadOfSensor, -spaceToOneSide, spaceToOneSide];
outImageSize = [NaN,1000]; % output image width in pixels

birdsEyeConfig = birdsEyeView(sensor, outView, outImageSize);
```

```
% Transform input image to bird's-eye-view image and display it
B = transformImage(birdsEyeConfig, J1);

% Place a 2-meter marker ahead of the sensor in bird's-eye view
imagePoint0 = vehicleToImage(birdsEyeConfig, [2, 0]);
offset = 5; % offset marker from text label by 5 pixels
annotatedB = insertMarker(B, imagePoint0 - offset);
annotatedB = insertText(annotatedB, imagePoint0, '2 meters');

figure
imshow(annotatedB)
title('Bird''s-Eye View')
```


Bird's-Eye View
2 meters

The plot above shows that the camera measures distances accurately. Now you can use the monocular camera for object and lane boundary detection. See the “Visual Perception Using Monocular Camera” on page 7-78 example.

See Also

Apps
Camera Calibrator

Functions
`detectCheckerboardPoints` | `estimateFisheyeParameters` | `estimateMonoCameraParameters` | `generateCheckerboardPoints` | `undistortFisheyeImage`

Objects
`birdsEyeView` | `monoCamera`

More About

- “Calibrate a Monocular Camera” on page 1-9
- “Visual Perception Using Monocular Camera” on page 7-78

Annotate Video Using Detections in Vehicle Coordinates

Configure and use a `monoCamera` object to display information provided in vehicle coordinates on a video display.

Overview

Displaying data recorded in vehicle coordinates on a recorded video is an integral part of ground truth labeling and analyzing tracking results. Using a two-dimensional bird's-eye view can help you understand the overall environment, but it is sometimes hard to correlate the video with the bird's-eye view display. In particular, this problem becomes worse when using a third-party sensor where you cannot access the raw video captured by the sensor, and you need to use a video captured by a separate camera.

Automated Driving Toolbox™ provides the `monoCamera` object that facilitates the conversion between vehicle coordinates and image coordinates. This example reads data recorded by a video sensor installed on a test vehicle. Then it displays the data on a video captured by a separate video camera installed on the same car. The data and video were recorded at the following rates:

- Reported lane information: 20 times per second
- Reported vision objects: 10 times per second
- Video frame rate: 20 frames per second

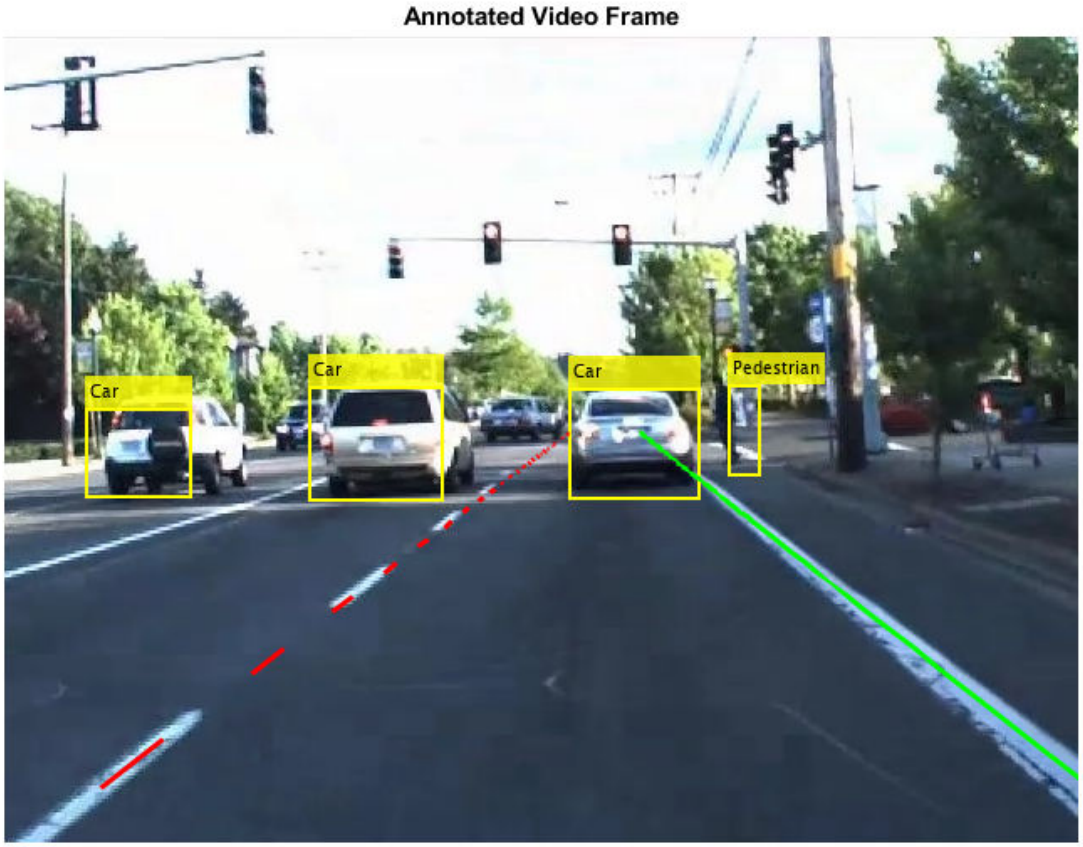
Display a Frame with Video Annotations

The selected frame corresponds to 5.9 seconds into the video clip, when there are several objects to show on the video.

```
% Set up video reader and player
videoFile      = '01_city_c2s_fcw_10s.mp4';
videoReader = VideoReader(videoFile);
videoPlayer = vision.DeployableVideoPlayer;

% Jump to the desired frame
time = 5.9;
videoReader.CurrentTime = time;
frameWithoutAnnotations = readFrame(videoReader);

imshow(frameWithoutAnnotations);
title('Original Video Frame')
```

Get the corresponding recorded data.

```
recordingFile  = '01_city_c2s_fcw_10s_sensor.mat';
[visionObjects, laneReports, timeStep, numSteps] = readDetectionsFile(recordingFile);
currentStep = round(time / timeStep) + 1;
videoDetections = processDetections(visionObjects(currentStep));
laneBoundaries = processLanes(laneReports(currentStep));

% Set up the monoCamera object for on-video display
sensor = setupMonoCamera(videoReader);

frameWithAnnotations = updateDisplay(frameWithoutAnnotations, sensor, videoDetections, laneBounda

imshow(frameWithAnnotations);
title('Annotated Video Frame')
```

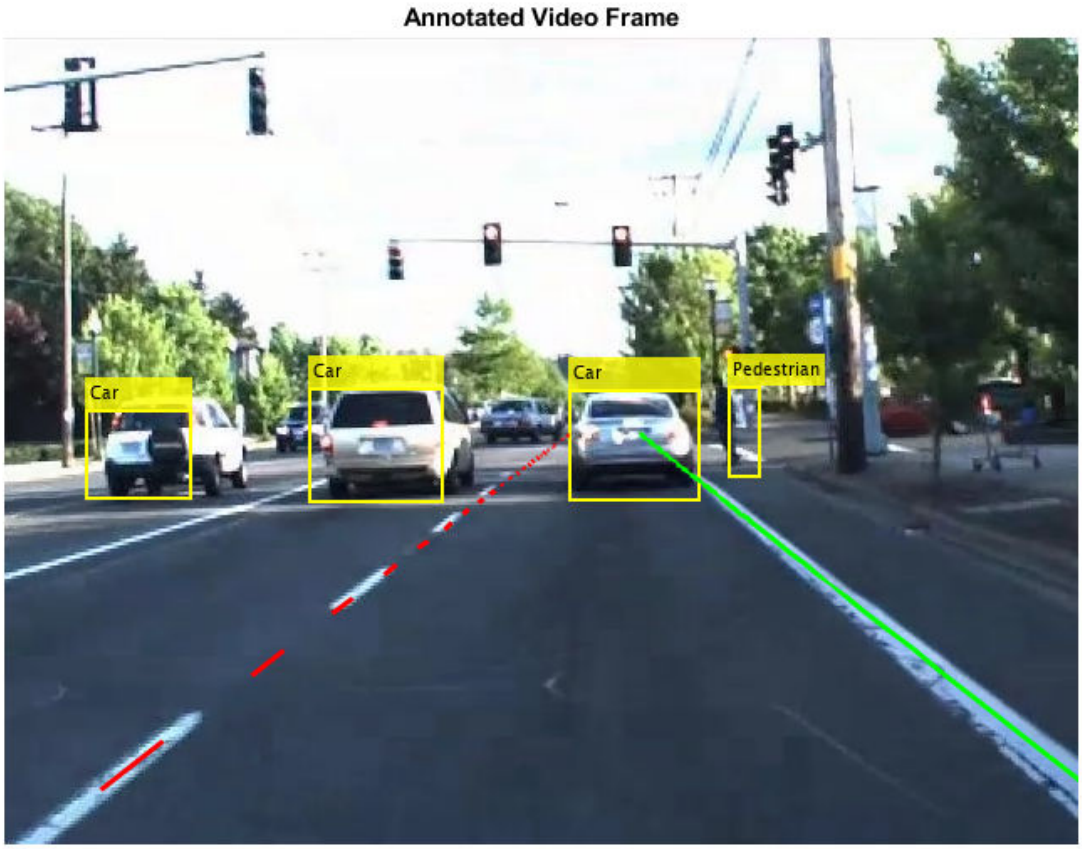


Display a Clip with Video Annotations

To display the video clip with annotations, simply repeat the annotation frame-by-frame. The video shows that the car pitches slightly up and down, which changes the pitch angle. No attempt has been made to compensate for this pitch motion. As a result, the conversion from vehicle coordinates to image coordinates is a little inaccurate on some of the frames.

```
% Reset the time back to zero
currentStep = 0;                % Reset the recorded data timestep
videoReader.CurrentTime = 0;    % Reset the video reader time
while currentStep < numSteps && hasFrame(videoReader)
    % Update scenario counters
    currentStep = currentStep + 1;

    % Get the current time
    tic

    % Prepare the detections to the tracker
    videoDetections = processDetections(visionObjects(currentStep), videoDetections);

    % Process lanes
    laneBoundaries = processLanes(laneReports(currentStep));
```

```
    % Update video frame with annotations from the reported objects
    frameWithoutAnnotations = readFrame(videoReader);
    frameWithAnnotations = updateDisplay(frameWithoutAnnotations, sensor, videoDetections, laneBo

    % The recorded data was obtained at a rate of 20 frames per second.
    % Pause for 50 milliseconds for a more realistic display rate. If you
    % process data and form tracks in this loop, you do not need this
    % pause.
    pause(0.05 - toc);

    % Display annotated frame
    videoPlayer(frameWithAnnotations);
end
```



Create the Mono Camera for On-Video Display

The `setupMonoCamera` function returns a `monoCamera` sensor object, which is used for converting positions in vehicle coordinates to image coordinates.

Knowing the camera's intrinsic and extrinsic calibration parameters is critical to accurate conversion between pixel and vehicle coordinates.

Start by defining the camera intrinsic parameters. The parameters in this function were estimated based on the camera model. To obtain the parameters for your camera, use the Camera Calibrator (Computer Vision Toolbox) app.

Because the data in this example has little distortion, this function ignores the lens distortion coefficients. The parameters are next stored in a `cameraIntrinsics` object.

Next, define the camera extrinsics. Camera extrinsics relate to the way the camera is mounted on the car. The mounting includes the following properties:

- Height: Mounting height above the ground, in meters.
- Pitch: Pitch of the camera, in degrees, where positive is angled below the horizon and toward the ground. In most cases, the camera is pitched slightly below the horizon.
- Roll: Roll of the camera about its axis. For example, if the video is flipped upside down, use roll = 180.
- Yaw: Angle of the camera sideways, where positive is in the direction of the positive *y*-axis (to the left). For example, a forward-facing camera has a yaw angle of 0 degrees, and a backward-facing camera has a yaw angle of 180 degrees.

```
function sensor = setupMonoCamera(vidReader)
% Define the camera intrinsics from the video information
focalLength    = [1260 1100];                    % [fx, fy]              % pixels
principalPoint = [360 245];                      % [cx, cy]              % pixels
imageSize = [vidReader.height, vidReader.width]; % [numRows, numColumns] % pixels
intrinsics = cameraIntrinsics(focalLength, principalPoint, imageSize);

% Define the camera mounting (camera extrinsics)
mountingHeight = 1.45;   % height in meters from the ground
mountingPitch  = 1.25;   % pitch of the camera in degrees
mountingRoll   = 0.15;   % roll of the camera in degrees
mountingYaw    = 0;      % yaw of the camera in degrees
sensor = monoCamera(intrinsics, mountingHeight, ...
    'Pitch', mountingPitch, ...
    'Roll',  mountingRoll, ...
    'Yaw',   mountingYaw);
end
```

Using the Mono Camera Object to Update the Display

The `updateDisplay` function displays all the object annotations on top of the video frame.

The display update includes the following steps:

1. Using the `monoCamera` sensor to convert reported detections into bounding boxes and annotating the frame.
2. Using the `insertLaneBoundary` method of the `parabolicLaneBoundary` object to insert the lane annotations.

```
function frame = updateDisplay(frame, sensor, videoDetections, laneBoundaries)

% Allocate memory for bounding boxes
bboxes = zeros(numel(videoDetections), 4);

% Create the bounding boxes
for i = 1:numel(videoDetections)
```

```
        % Use monoCamera sensor to convert the position in vehicle coordinates
        % to the position in image coordinates.
        % Notes:
        %   1. The width of the object is reported and is used to calculate the
        %      size of the bounding box around the object (half width on each
        %      side). The height of the object is not reported. Instead, the
        %      function uses a height/width ratio of 0.85 for cars and 3 for
        %      pedestrians.
        %   2. The reported location is at the center of the object at ground
        %      level, i.e., the bottom of the bounding box.
        xyLocation1 = vehicleToImage(sensor, videoDetections(i).positions' + [0,videoDetections(i).wi
        xyLocation2 = vehicleToImage(sensor, videoDetections(i).positions' - [0,videoDetections(i).wi
        dx = xyLocation2(1) - xyLocation1(1);

        % Define the height/width ratio based on object class
        if strcmp(videoDetections(i).labels, 'Car')
            dy = dx * 0.85;
        elseif  strcmp(videoDetections(i).labels, 'Pedestrian')
            dy = dx * 3;
        else
            dy = dx;
        end

        % Estimate the bounding box around the vehicle. Subtract the height of
        % the bounding box to define the top-left corner.
        bboxes(i,:) =[(xyLocation1 - [0, dy]), dx, dy];
    end

    % Add labels
    labels = {videoDetections(:).labels}';

    % Add bounding boxes to the frame
    if ~isempty(labels)
        frame = insertObjectAnnotation(frame, 'rectangle', bboxes, labels,...
            'Color', 'yellow', 'FontSize', 10, 'TextBoxOpacity', .8, 'LineWidth', 2);
    end

    % Display the lane boundary on the video frame
    xRangeVehicle = [1, 100];
    xPtsInVehicle = linspace(xRangeVehicle(1), xRangeVehicle(2), 100)';
    frame = insertLaneBoundary(frame, laneBoundaries(1), sensor, xPtsInVehicle, ...
        'Color', 'red');
    frame = insertLaneBoundary(frame, laneBoundaries(2), sensor, xPtsInVehicle, ...
        'Color', 'green');
    end
```

Summary

This example showed how to create a `monoCamera` sensor object and use it to display objects described in vehicle coordinates on a video captured by a separate camera. Try using recorded data and a video camera of your own. Try calibrating your camera to create a `monoCamera` that allows for transformation from vehicle to image coordinates, and vice versa.

Supporting Functions

readDetectionsFile - Reads the recorded sensor data file. The recorded data is in a single structure that is divided into four `struct` arrays. This example uses only the following two arrays:

1. `laneReports`, a `struct` array that reports the boundaries of the lane. It has these fields: `left` and `right`. Each element of the array corresponds to a different timestep. Both `left` and `right` are structures with these fields: `isValid`, `confidence`, `boundaryType`, `offset`, `headingAngle`, and `curvature`.
2. `visionObjects`, a struct array that reports the detected vision objects. It has the fields `numObjects` (`integer`) and `object` (`struct`). Each element of the array corresponds to a different timestep. `object` is a `struct` array, where each element is a separate object with these fields: `id`, `classification`, `position (x;y;z)`, `velocity(vx;vy;vz)`, `size(dx;dy;dz)`. Note: `z=vy=vz=dx=dz=0`

```
function [visionObjects, laneReports, timeStep, numSteps] = readDetectionsFile(filename)
A = load(strcat(filename));
visionObjects = A.vision;
laneReports = A.lane;

% Prepare some time variables
timeStep = 0.05;                % Lane data is provided every 50 milliseconds
numSteps = numel(visionObjects); % Number of recorded timesteps
end
```

processDetections - Reads the recorded vision detections. This example extracts only the following properties:

1. Position: A two-dimensional `[x, y]` array in vehicle coordinates
2. Width: The width of the object as reported by the video sensor (Note: The sensor does not report any other dimension of the object size.)
3. Labels: The reported classification of the object

```
function videoDetections = processDetections(visionData, videoDetections)
% The video sensor reports a classification value as an integer
% according to the following enumeration (starting from 0)
ClassificationValues = {'Unknown', 'Unknown Small', 'Unknown Big', ...
    'Pedestrian', 'Bike', 'Car', 'Truck', 'Barrier'};

% The total number of objects reported by the sensor in this frame
numVideoObjects = visionData.numObjects;

% The video objects are reported only 10 times per second, but the video
% has a frame rate of 20 frames per second. To prevent the annotations from
% flickering on and off, this function returns the values from the previous
% timestep if there are no video objects.
if numVideoObjects == 0
    if nargin == 1 % Returning a result even if there is no previous value
        videoDetections = struct('positions', {}, 'labels', {}, 'widths', {});
    end
    return;
else
    % Prepare a container for the relevant properties of video detections
    videoDetections = struct('positions', [], 'labels', [], 'widths', []);
    for i = 1:numVideoObjects
        videoDetections(i).widths = visionData.object(i).size(2);
        videoDetections(i).positions = visionData.object(i).position(1:2);
        videoDetections(i).labels = ClassificationValues{visionData.object(i).classification + 1
    end
end
end
```

processLanes - Reads reported lane information and converts it into `parabolicLaneBoundary` objects.

Lane boundaries are updated based on the `laneReports` from the recordings. The sensor reports the lanes as parameters of a parabolic model: % $y = ax^2 + bx + c$

```
function laneBoundaries = processLanes(laneReports)
% Return processed lane boundaries

% Boundary type information
types = {'Unmarked', 'Solid', 'Dashed', 'Unmarked', 'BottsDots', ...
    'Unmarked', 'Unmarked', 'DoubleSolid'};

% Read the recorded lane reports for this frame
leftLane    = laneReports.left;
rightLane   = laneReports.right;

% Create parabolicLaneBoundary objects for left and right lane boundaries
leftParams = cast([leftLane.curvature, leftLane.headingAngle, leftLane.offset], 'double');
leftBoundaries = parabolicLaneBoundary(leftParams);
leftBoundaries.BoundaryType = types{leftLane.boundaryType};

rightParams  = cast([rightLane.curvature, rightLane.headingAngle, rightLane.offset], 'double');
rightBoundaries = parabolicLaneBoundary(rightParams);
rightBoundaries.BoundaryType = types{rightLane.boundaryType};

laneBoundaries = [leftBoundaries, rightBoundaries];
end
```

See Also

Apps
Camera Calibrator

Functions
`insertLaneBoundary` | `insertObjectAnnotation`

Objects
`VideoReader` | `monoCamera` | `vision.DeployableVideoPlayer`

More About

- “Calibrate a Monocular Camera” on page 1-9
- “Coordinate Systems in Automated Driving Toolbox” on page 1-2
- “Visual Perception Using Monocular Camera” on page 7-78

Automate Ground Truth Labeling of Lane Boundaries

This example shows how to develop an algorithm for the automated marking of lane boundaries in the Ground Truth Labeler app.

The Ground Truth Labeler App

Good ground truth data is crucial for developing driving algorithms and evaluating their performances. However, creating a rich and diverse set of annotated driving data requires significant time and resources. The Ground Truth Labeler app makes this process efficient. You can use this app as a fully manual annotation tool to mark lane boundaries, vehicle bounding boxes, and other objects of interest for a vision system. However, manual labeling requires a significant amount of time and resources. This app also provides a framework to create algorithms to extend and automate the labeling process. You can use the algorithms you create to quickly label entire data sets, and then follow it up with a more efficient, shorter manual verification step. You can also edit the results of the automation step to account for challenging scenarios that the automation algorithm might have missed. This example describes how to insert a lane detection algorithm into the automation workflow of the app.

Create a Lane Detection Algorithm

First, create a lane detection algorithm. The “Visual Perception Using Monocular Camera” on page 7-78 example describes the process of detecting lane boundaries, and the `helperMonoSensor` class packages that algorithm into a single, reusable class. Try out the algorithm on a single video frame to detect the left ego lane boundary.

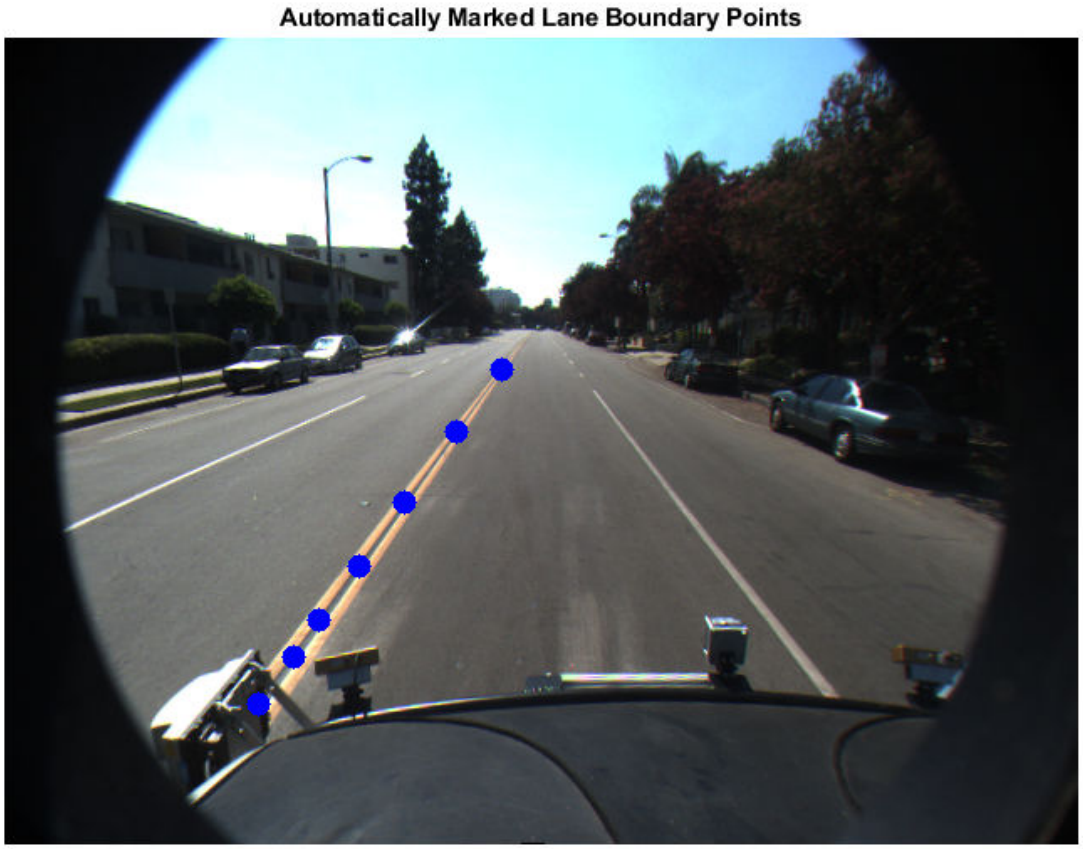
```
configData = load('birdsEyeConfig');
sensor     = configData.birdsEyeConfig.Sensor;
monoSensor = helperMonoSensor(sensor);
I          = imread('road.png');

sensorOut = processFrame(monoSensor, I);
lb = sensorOut.leftEgoBoundary;
figure
IwithLane = insertLaneBoundary(I, lb, sensor, [3 30], 'Color', 'blue');
imshow(IwithLane);
title('Detected Left Lane Boundary Model');
```

Mark Lane Boundary Points

The lane detected in the previous step is a *model* and must be converted to a set of discrete points. These points are similar to what a user might manually place on the image. In the camera view, parts of the lane boundary closer to the vehicle (lower part of the camera image) will span more pixels than the further parts. Consequently, a user would place more points with higher confidence in the lower parts of the camera image. To replicate this behavior, determine the lane boundary locations from the boundary model more densely at points closer to the vehicle.

```
ROI      = [3 30];
xPoints  = [3 3.5 4 5 7 12 30]'; % More dense closer to the vehicle
yPoints  = lb.computeBoundaryModel(xPoints);

% Find corresponding image locations
boundaryPointsOnImage = vehicleToImage(sensor, [xPoints, yPoints]);

imshow(I)
hold on
plot(boundaryPointsOnImage(:,1), boundaryPointsOnImage(:,2),...
    'o',...
    'MarkerEdgeColor','b',...
    'MarkerFaceColor','b',...
    'MarkerSize',10)
```

```
title('Automatically Marked Lane Boundary Points');
hold off
```



Prepare the Lane Detection Automation Class

To incorporate this lane detection algorithm into the automation workflow of the app, construct a class that inherits from the abstract base class `vision.labeler.AutomationAlgorithm` (Computer Vision Toolbox). This base class defines properties and signatures for methods that the app uses for configuring and running the custom algorithm. The Ground Truth Labeler app provides a convenient way to obtain an initial automation class template. For details, see “Create Automation Algorithm for Labeling” (Computer Vision Toolbox). The `AutoLaneMarking` class is based off of this template and provides you with a ready-to-use automation class for lane detection. The comments of the class outline the basic steps needed to implement each API call.

Step 1 contains properties that define the name and description of the algorithm, and the directions for using the algorithm.

```
    %--------------------------------------------------------------------
    % Step 1: Define required properties describing the algorithm. This
    %         includes Name, Description, and UserDirections.
    properties(Constant)
```

```
        % Name: Give a name for your algorithm.
        Name = 'Lane Detector';

        % Description: Provide a one-line description for your algorithm.
        Description = 'Automatically detect lane-like features';

        % UserDirections: Provide a set of directions that are displayed
        %                 when this algorithm is invoked. The directions
        %                 are to be provided as a cell array of character
        %                 vectors, with each element of the cell array
        %                 representing a step in the list of directions.
        UserDirections = {...
            'Load a MonoCamera configuration object from the workspace using the settings panel',
            'Specify additional parameters in the settings panel',...
            'Run the algorithm',...
            'Manually inspect and modify results if needed'};
    end
```

Step 2 contains the custom properties needed for the core algorithm. The necessary properties were determined from the lane detection and lane point creation section above.

```
    %--------------------------------------------------------------------
    % Step 2: Define properties to be used during the algorithm. These are
    % user-defined properties that can be defined to manage algorithm
    % execution.
    properties
        %MonoCamera
        %  The monoCamera object associated with this video
        MonoCamera          = [];
        %MonoCameraVarname
        %  The workspace variable name of the monoCamera object
        MonoCameraVarname   = '';
        %BirdsEyeConfig
        %  The birdsEyeView object needed to create the bird's-eye view
        BirdsEyeConfig      = [];
        %MaxNumLanes
        %  The maximum number of lanes the algorithm tries to annotate
        MaxNumLanes         = 2;
        %ROI
        %  The region of interest around the vehicle used to search for
        %  lanes
        ROI            = [3, 30, -3, 3];
        %LaneMaskSensitivity
        %  The sensitivity parameter used in the segmentLaneMarkerRidge function
        LaneMaskSensitivity = 0.25;
        %LaneBoundaryWidth
        %  The lane boundary width, used in findParabolicLaneBoundaries
        LaneBoundaryWidth   = 0.6;
        %XPoints
        %  The x-axis points along which to mark the lane boundaries
        XPoints             = [3 3.5 4 4.5 5 6 7 10 30];
    end
```

Step 3 deals with function definitions. The first function, `checkLabelDefinition`, ensures that only labels of the appropriate type are enabled for automation. For lane detection, you need to ensure that only labels of type `Line` are enabled, so this version of the function checks the `Type` of the labels:

```
        function TF = checkLabelDefinition(~, labelDef)
            % Lane detection only works with Line type labels
```

```
        TF = labelDef.Type == labelType.Line;
    end
```

The next function is `checkSetup`. Note that this algorithm *requires* a `monoCamera` sensor configuration to be available. All other properties have defined reasonable defaults.

```
    function TF = checkSetup(algObj, ~)
        % This is the only required input
        TF = ~isempty(algObj.MonoCamera);
    end
```

Next, the `settingsDialog` function obtains and modifies the properties defined in Step 2. This API call lets you create a dialog box that opens when a user clicks the **Settings** button in the **Automate** tab. To create this dialog box, use the `inputdlg` function to quickly create a simple modal window to ask a user to specify the `monoCamera` object. The following snippet of code outlines the basic syntax. The full `AutoLaneMarking` code extends this logic and also adds input validation steps.

```
% Describe the inputs
prompt = {...
    'Enter the MonoCamera variable name',...
    'Maximum number of Lanes',...
    };
defaultAnswer = {...
    '',...
    num2str(2),...
    };

% Create an input dialog
name = 'Settings for lane detection';
numLines = 1;
options.Resize      = 'on';
options.WindowStyle = 'normal';
options.Interpreter = 'none';
answer = inputdlg(prompt,name,numLines,defaultAnswer,options);

% Obtain the inputs
monoCameraVarname = answer{1};
maxNumberOfLanes  = answer{2};
```

Step 4 specifies the execution functions. Some automation algorithms need to implement an `initialize` routine to populate the initial algorithm state based on the existing labels in the app. This lane detection algorithm works on each frame independently, so the default version of the template has been trimmed to take no action.

```
    function initialize(~, ~, ~)
    end
```

Next, the `run` function defines the core lane detection algorithm of this automation class. `run` gets called for each video frame, and expects the automation class to return a set of labels. The `run` function in `AutoLaneMarking` contains the logic introduced previously for the lane detection and conversion to points. Code from `helperMonoSensor` has also been folded in for a more compact reference.

```
    function autoLabels = run(algObj, I)
        Ig = rgb2gray(I);
        birdsEyeViewImage = transformImage(algObj.BirdsEyeConfig, Ig);
        birdsEyeViewBW    = segmentLaneMarkerRidge(birdsEyeViewImage, ...
```

```
            algObj.BirdsEyeConfig, algObj.LaneBoundaryWidth, ...
            'Sensitivity', algObj.LaneMaskSensitivity);

        % Obtain lane candidate points in world coordinates
        [imageX, imageY] = find(birdsEyeViewBW);
        boundaryPointsxy = imageToVehicle(algObj.BirdsEyeConfig, [imageY, imageX]);

        % Fit requested number of boundaries to it
        lbs = findParabolicLaneBoundaries(...
            boundaryPointsxy,algObj.LaneBoundaryWidth, ...
            'MaxNumBoundaries',algObj.MaxNumLanes);
        numDetectedLanes = numel(lbs);

        % Convert the model to discrete set of points at the specified
        % x coordinates
        boundaryPoints = cell(1,numDetectedLanes);
        xPoints = algObj.XPoints';
        for ind = 1:numel(lbs)
            yPoints             = lbs(ind).computeBoundaryModel(xPoints);
            boundaryPoints{ind} = vehicleToImage(algObj.MonoCamera, [xPoints, yPoints]);
        end

        % Package up the results in a table
        autoLabels = table(...
            boundaryPoints',...
            repmat(labelType.Line, [numDetectedLanes,1]),...
            repmat(algObj.SelectedLabelDefinitions.Name, [numDetectedLanes,1]));
        autoLabels.Properties.VariableNames = {'Position','Type','Name'};
    end
```

Finally, the `terminate` function handles any cleanup or tear-down required after the automation is done. This algorithm does not require any cleanup, so the function is empty.

```
    function terminate(~)
    end
```

Use the AutoLaneMarking Automation Class in the App

The packaged version of the lane detection algorithm is now ready for use in the `AutoLaneMarking` class. To use this class in the app:

- Create the folder structure required under the current folder, and copy the automation class into it.

```
mkdir('+vision/+labeler');
copyfile(fullfile(matlabroot,'toolbox','driving','drivingdemos','AutoLaneMarking.m'),'+visio
```

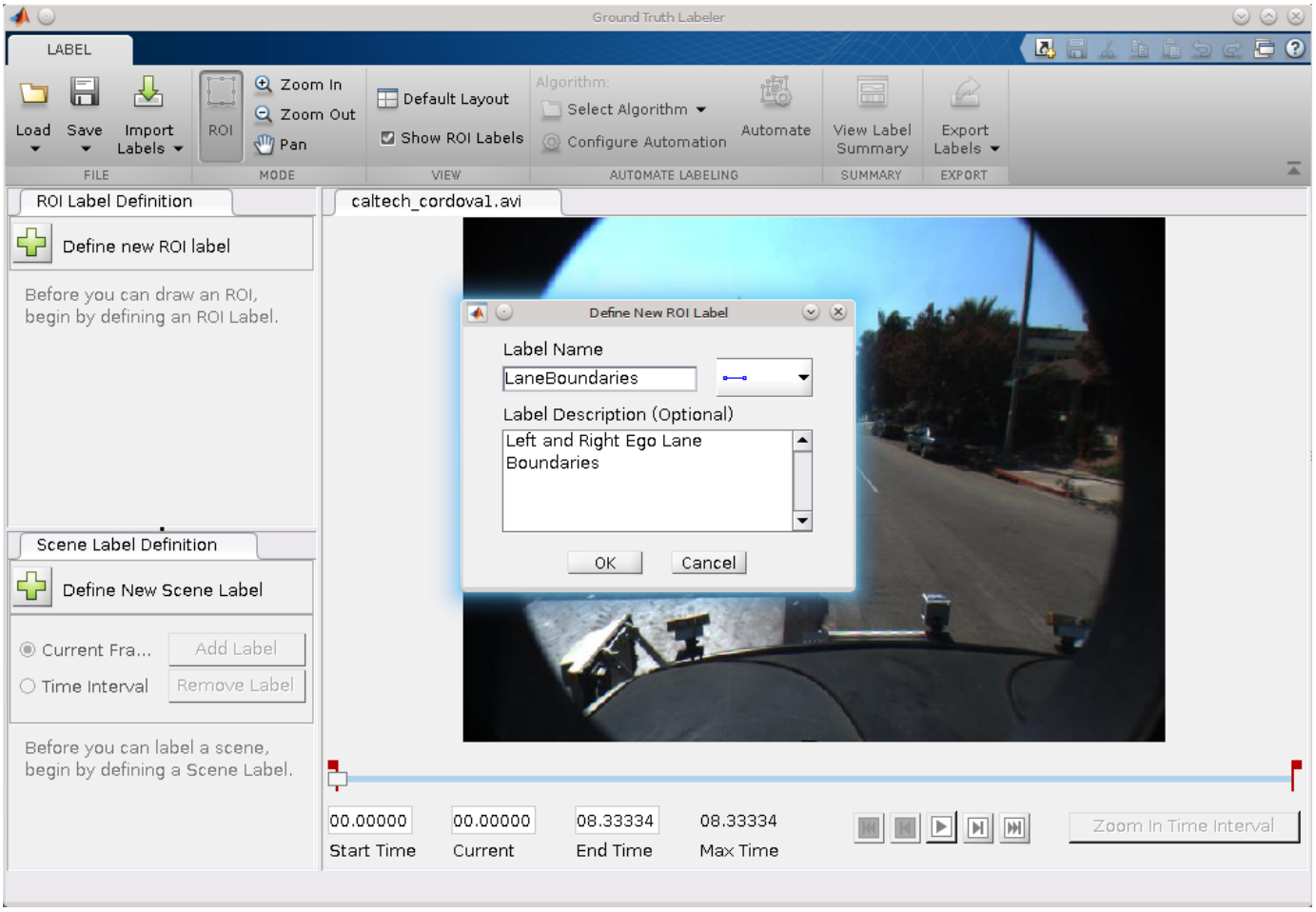
- Load the `monoCamera` information into the workspace.

```
configData = load('birdsEyeConfig');
sensor   = configData.birdsEyeConfig.Sensor;
```

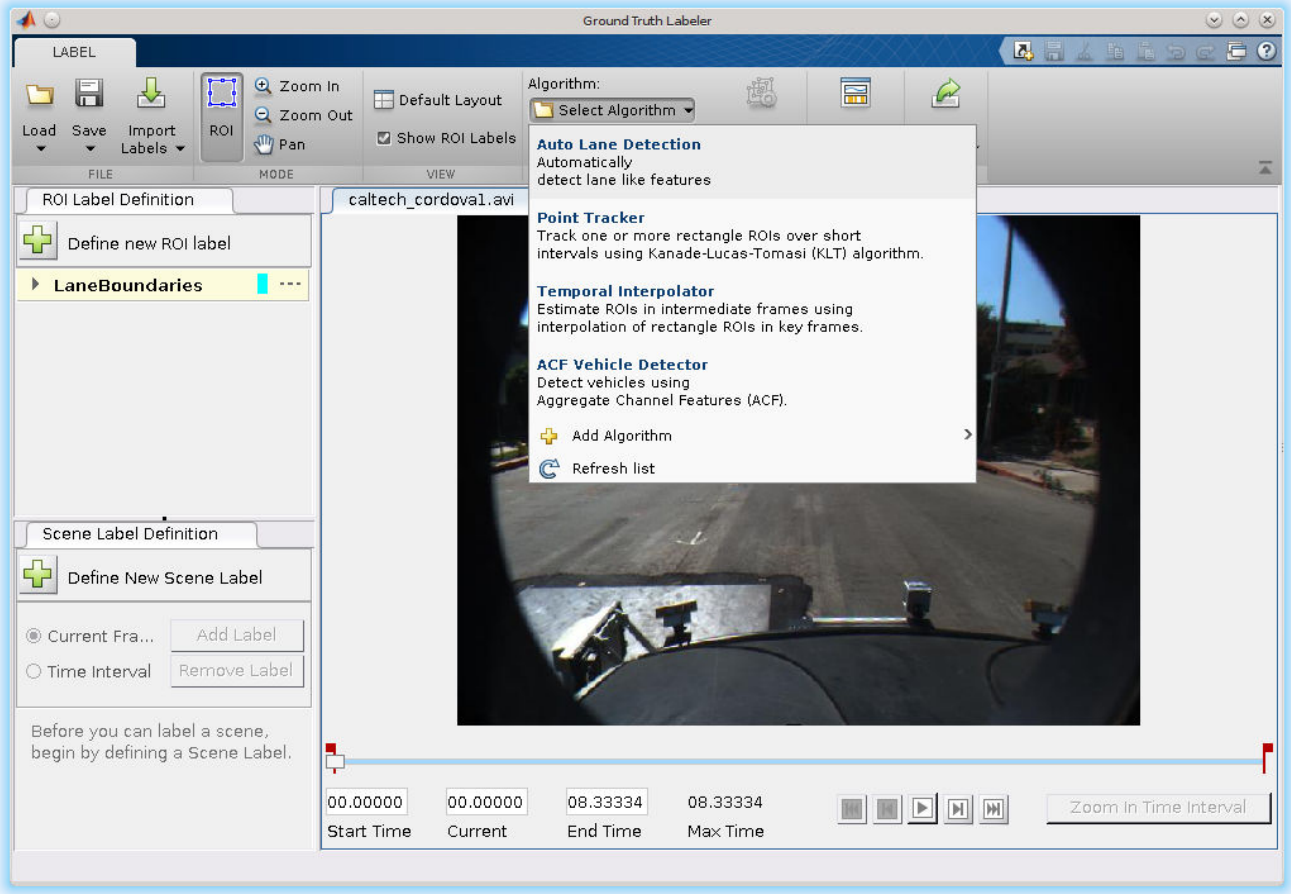
- Open the Ground Truth Labeler app.

```
groundTruthLabeler caltech_cordova1.avi
```

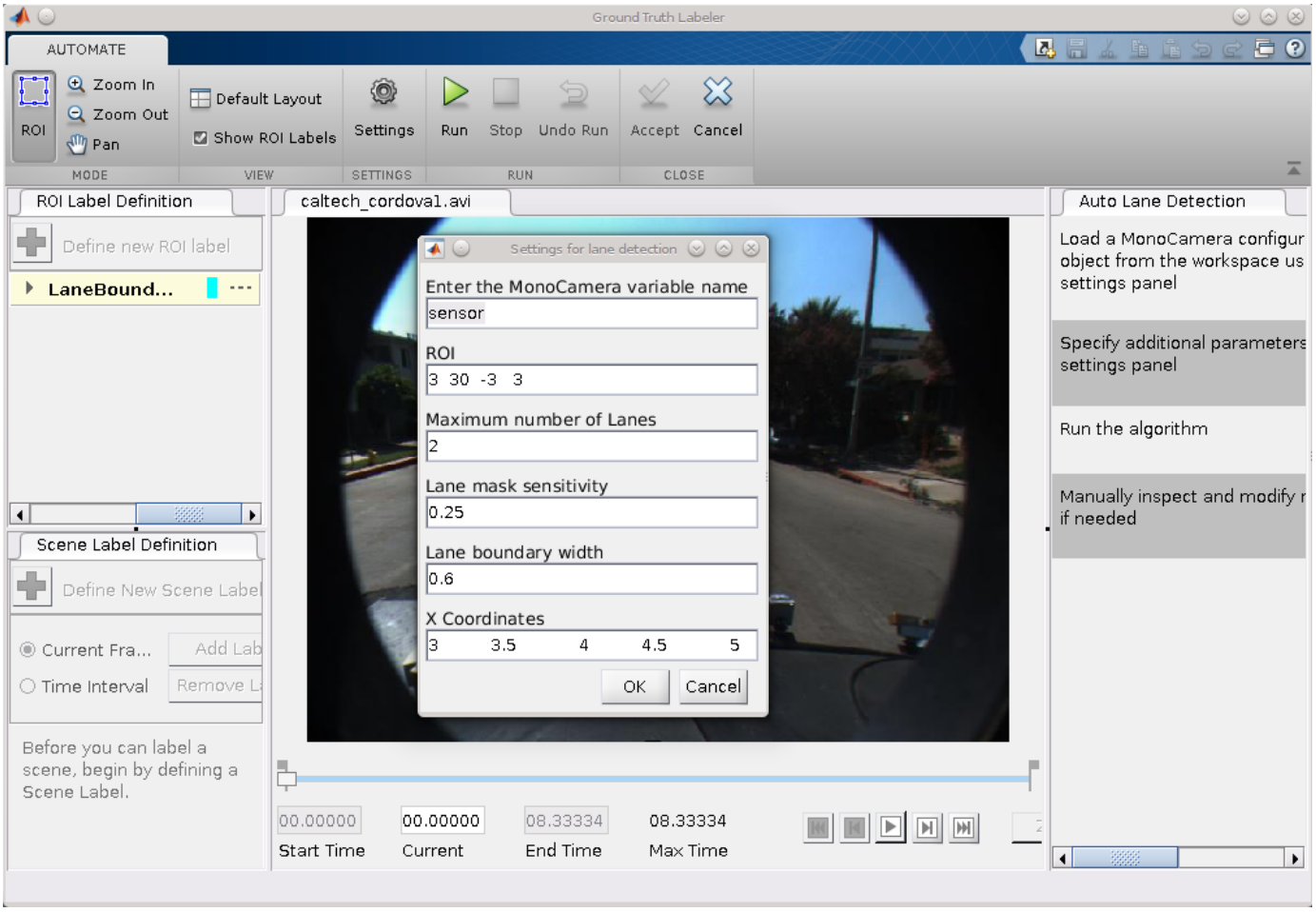
- On the left pane, click the **Define new ROI label** button and define the ROI line style shown. Then click OK.



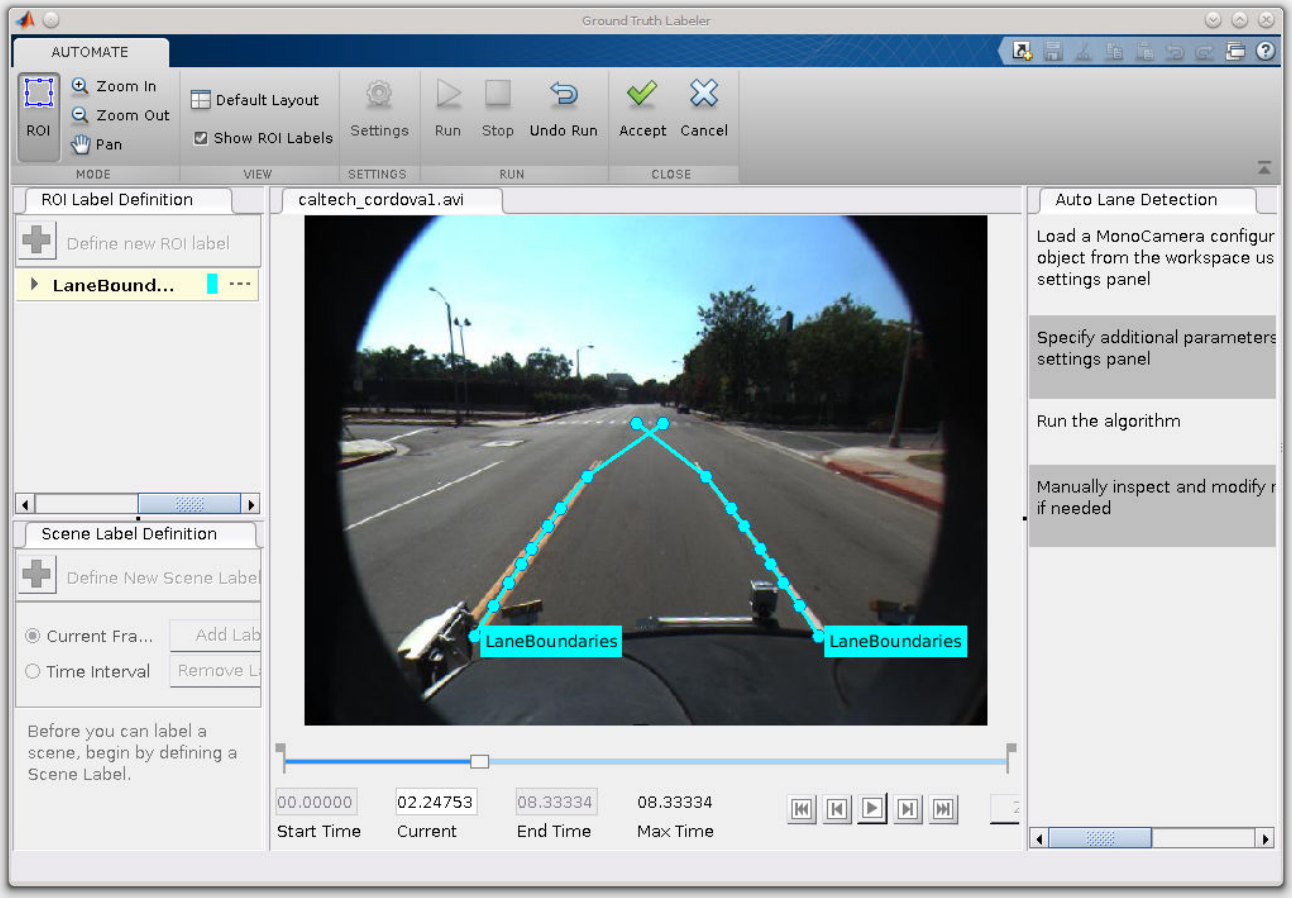
- Click **Algorithm > Select Algorithm > Refresh list**.
- Click **Algorithm > Auto Lane Detection**. If you do not see this option, ensure that the current working folder has a folder called `+vision/+labeler`, with a file named `AutoLaneMarking.m` in it.



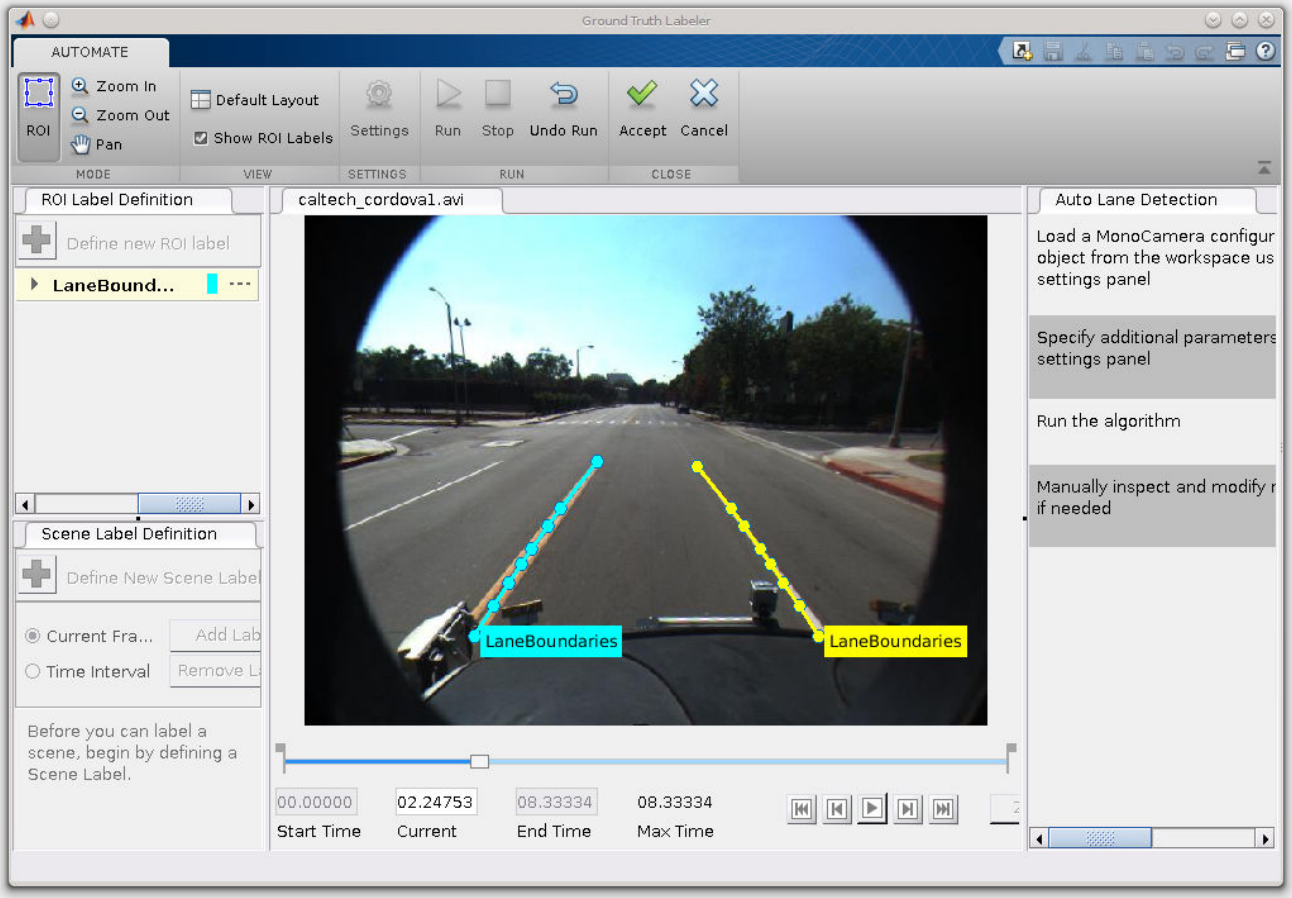
- Click **Automate**. A new tab will open, displaying directions for using the algorithm.
- Click **Settings**, and in the dialog box that opens, enter `sensor` in the first text box. Modify other parameters if needed before clicking **OK**.



- Click **Run**. The lane detection algorithm progresses on the video. Notice that the results are not satisfactory in some of the frames.
- After the run is completed, Use the slider or arrow keys to scroll across the video to locate the frames where the algorithm failed.



- Manually tweak the results by either moving the lane boundary points or deleting entire boundaries.



- Once you are satisfied with the lane boundaries for the entire video, click **Accept**.

The auto lane detection part of labeling the video is complete. You can proceed with labeling other objects of interest, save the session, or export the results of this labeling run.

Conclusion

This example showed the steps to incorporate a lane detection algorithm into the Ground Truth Labeler app. You can extend this concept to other custom algorithms to simplify and extend the functionality of the app.

See Also

Apps
Ground Truth Labeler

Objects
`monoCamera` | `vision.labeler.AutomationAlgorithm`

More About

- “Create Automation Algorithm for Labeling” (Computer Vision Toolbox)
- “Automate Ground Truth Labeling for Semantic Segmentation” on page 7-29
- “Automate Attributes of Labeled Objects” on page 7-39

Automate Ground Truth Labeling for Semantic Segmentation

This example shows how to use a pretrained semantic segmentation algorithm to segment the sky and road in an image, and use this algorithm to automate ground truth labeling in the Ground Truth Labeler app.

The Ground Truth Labeler App

Good ground truth data is crucial for developing automated driving algorithms and evaluating their performance. However, creating and maintaining a diverse and high-quality set of annotated driving data requires significant effort. The Ground Truth Labeler app makes this process easy and efficient. This app includes features to annotate objects as rectangles, lines, or pixel labels. Pixel labeling is a process in which each pixel in an image is assigned a class or category, which can then be used to train a pixel-level segmentation algorithm. Although you can use the app to manually label all your data, this process requires a significant amount of time and resources, especially for pixel labeling. As an alternative, the app also provides a framework to incorporate algorithms to extend and automate the labeling process. You can use the algorithms you create to automatically label entire data sets, and then end with a more efficient, shorter manual verification step. You can also edit the results of the automation step to account for challenging scenarios that the algorithm might have missed.

In this example, you will:

- Use a pretrained segmentation algorithm to segment pixels that belong to the categories 'Road' and 'Sky'.
- Create an automation algorithm that can be used in the Ground Truth Labeler app to automatically label road and sky pixels.

This ground truth data can then be used to train a new semantic segmentation network, or retrain an existing one.

Create a Road and Sky Detection Algorithm

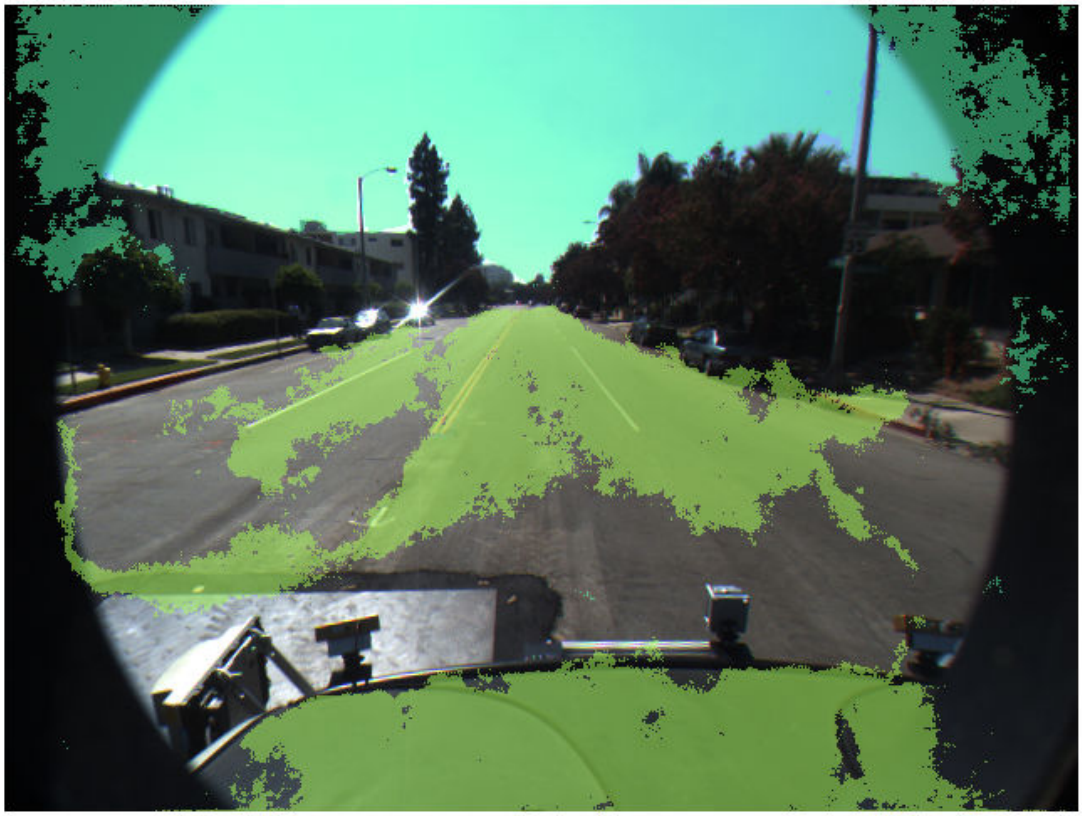
First, create a semantic segmentation algorithm that segments road and sky pixels in an image. The “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox) example describes how to train a deep learning network for semantic segmentation. This network has been trained to predict 11 classes of semantic labels including 'Road' and 'Sky'. The performance of these networks depends on how generalizable they are. Applying the networks to situations they did not encounter during training can lead to subpar results. Iteratively introducing custom training data to the learning process can make the network perform better on similar data sets.

Download a network, which was pretrained on the CamVid dataset [1][2] from the University of Cambridge.

```
pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/segnetVGG16CamVid.mat';
pretrainedFolder = fullfile(tempdir,'pretrainedSegNet');
pretrainedSegNet = fullfile(pretrainedFolder,'segnetVGG16CamVid.mat');
if ~exist(pretrainedSegNet,'file')
    if ~exist(pretrainedFolder,'dir')
        mkdir(pretrainedFolder);
    end
    disp('Downloading pretrained SegNet (107 MB)...');
    websave(pretrainedSegNet,pretrainedURL);
end
```

Segment an image and display it.

```
% Load the semantic segmentation network
data = load(pretrainedSegNet);

% Load a test image from drivingdata
roadSequenceData = fullfile(toolboxdir('driving'), 'drivingdata', 'roadSequence');
I = imread(fullfile(roadSequenceData, 'f00000.png'));

% Run the network on the image
automatedLabels = semanticseg(I, data.net);

% Display the labels overlaid on the image, choosing relevant categories
figure, imshow(labeloverlay(I, automatedLabels, 'IncludedLabels', ["Sky", "Road"]));
```



The output of the network is represented in MATLAB as a categorical matrix. The categories listed include all those that the semantic segmentation network has been trained on, not just the categories present in the output. This information is also available from the network object itself.

```
  data.net.Layers(end).ClassNames

% List categories of pixels labeled
categories(automatedLabels)

ans = 11×1 cell
    {'Sky'       }
```

```
    {'Building'  }
    {'Pole'      }
    {'Road'      }
    {'Pavement'  }
    {'Tree'      }
    {'SignSymbol'}
    {'Fence'     }
    {'Car'       }
    {'Pedestrian'}
    {'Bicyclist' }


% The blue overlay indicates the 'Sky' category, and the green overlay
% indicates 'Road'.
```

Review the Pixel Segmentation Automation Class

Incorporate this semantic segmentation algorithm into the automation workflow of the app by creating a class that inherits from the abstract base class `vision.labeler.AutomationAlgorithm` (Computer Vision Toolbox). This base class defines the API that the app uses to configure and run the algorithm. The Ground Truth Labeler app provides a convenient way to obtain an initial automation class template. For details, see “Create Automation Algorithm for Labeling” (Computer Vision Toolbox). The `RoadAndSkySegmentation` class is based on this template and provides a ready-to-use automation class for pixel label segmentation.

The first set of properties in the `RoadAndSkySegmentation` class specify the name of the algorithm, provide a brief description of it, and give directions for using it.

```
   properties(Constant)

       %Name
       %   Character vector specifying name of algorithm.
       Name = 'RoadAndSkySegmentation'

       %Description
       %   Character vector specifying short description of algorithm.
       Description = 'This algorithm uses semanticseg with a pretrained network to annotate roads

       %UserDirections
       %   Cell array of character vectors specifying directions for
       %   algorithm users to follow in order to use algorithm.
       UserDirections = {...
           ['Automation algorithms are a way to automate manual labeling ' ...
           'tasks. This AutomationAlgorithm automatically creates pixel ', ...
           'labels for road and sky.'], ...
           ['Review and Modify: Review automated labels over the interval ', ...
           'using playback controls. Modify/delete/add ROIs that were not ' ...
           'satisfactorily automated at this stage. If the results are ' ...
           'satisfactory, click Accept to accept the automated labels.'], ...
           ['Accept/Cancel: If results of automation are satisfactory, ' ...
           'click Accept to accept all automated labels and return to ' ...
           'manual labeling. If results of automation are not ' ...
           'satisfactory, click Cancel to return to manual labeling ' ...
           'without saving automated labels.']};
   end
```

The next section of the `RoadAndSkySegmentation` class specifies the custom properties needed by the core algorithm. The `PretrainedNetwork` property holds the pretrained network. The `AllCategories` property holds the names of the three categories.

```
properties
    % PretrainedNetwork saves the SeriesNetwork object that does the semantic
    % segmentation.
    PretrainedNetwork

    % Categories holds the default 'background', 'road', and 'sky'
    % categorical types.
    AllCategories = {'background'};

    % Store names for 'road' and 'sky'.
    RoadName
    SkyName
end
```

`checkLabelDefinition`, the first method defined in `RoadAndSkySegmentation` , checks that only labels of type `PixelLabel` are enabled for automation. `PixelLabel` is the only type needed for semantic segmentation.

```
function TF = checkLabelDefinition(~, labelDef)
    isValid = false;

    if (strcmpi(labelDef.Name, 'road') && labelDef.Type == labelType.PixelLabel)
        isValid = true;
        algObj.RoadName = labelDef.Name;
        algObj.AllCategories{end+1} = labelDef.Name;
    elseif (strcmpi(labelDef.Name, 'sky') && labelDef.Type == labelType.PixelLabel)
        isValid = true;
        algObj.SkyName = labelDef.Name;
        algObj.AllCategories{end+1} = labelDef.Name;
    elseif(labelDef.Type == labelType.PixelLabel)
        isValid = true;
    end
end
```

The next set of functions control the execution of the algorithm. The `vision.labeler.AutomationAlgorithm` class includes an interface that contains methods like `'initialize'`, `'run'`, and `'terminate'` for setting up and running the automation with ease. The `initialize` function populates the initial algorithm state based on the existing labels in the app. In the `RoadAndSkySegmentation` class, the `initialize` function has been customized to load the pretrained semantic segmentation network from `tempdir` and save it to the `PretrainedNetwork` property.

```
function initialize(algObj, ~, ~)

    % Point to tempdir where pretrainedSegNet was downloaded.
    pretrainedFolder = fullfile(tempdir,'pretrainedSegNet');
    pretrainedSegNet = fullfile(pretrainedFolder,'segnetVGG16CamVid.mat');
    data = load(pretrainedSegNet);
    % Store the network in the 'PretrainedNetwork' property of this object.
    algObj.PretrainedNetwork = data.net;
end
```

Next, the `run` function defines the core semantic segmentation algorithm of this automation class. `run` is called for each video frame, and expects the automation class to return a set of labels. The `run`

function in `RoadAndSkySegmentation` contains the logic introduced previously for creating a categorical matrix of pixel labels corresponding to "Road" and "Sky". This can be extended to any categories the network is trained on, and is restricted to these two for illustration only.

```
function autoLabels = run(algObj, I)
    % Setup categorical matrix with categories including road and
    % sky
    autoLabels = categorical(zeros(size(I,1), size(I,2)),0:2,algObj.AllCategories,'Ordina

    pixelCat = semanticseg(I, this.PretrainedNetwork);
    if ~isempty(pixelCat)
        % Add the selected label at the bounding box position(s)
        autoLabels(pixelCat == "Road") = algObj.RoadName;
        autoLabels(pixelCat == "Sky") = algObj.SkyName;
    end
end
```

This algorithm does not require any cleanup, so the `terminate` function is empty.

Use the Pixel Segmentation Automation Class in the App

The properties and methods described in the previous section have been implemented in the `RoadAndSkySegmentation` automation algorithm class file. To use this class in the app:

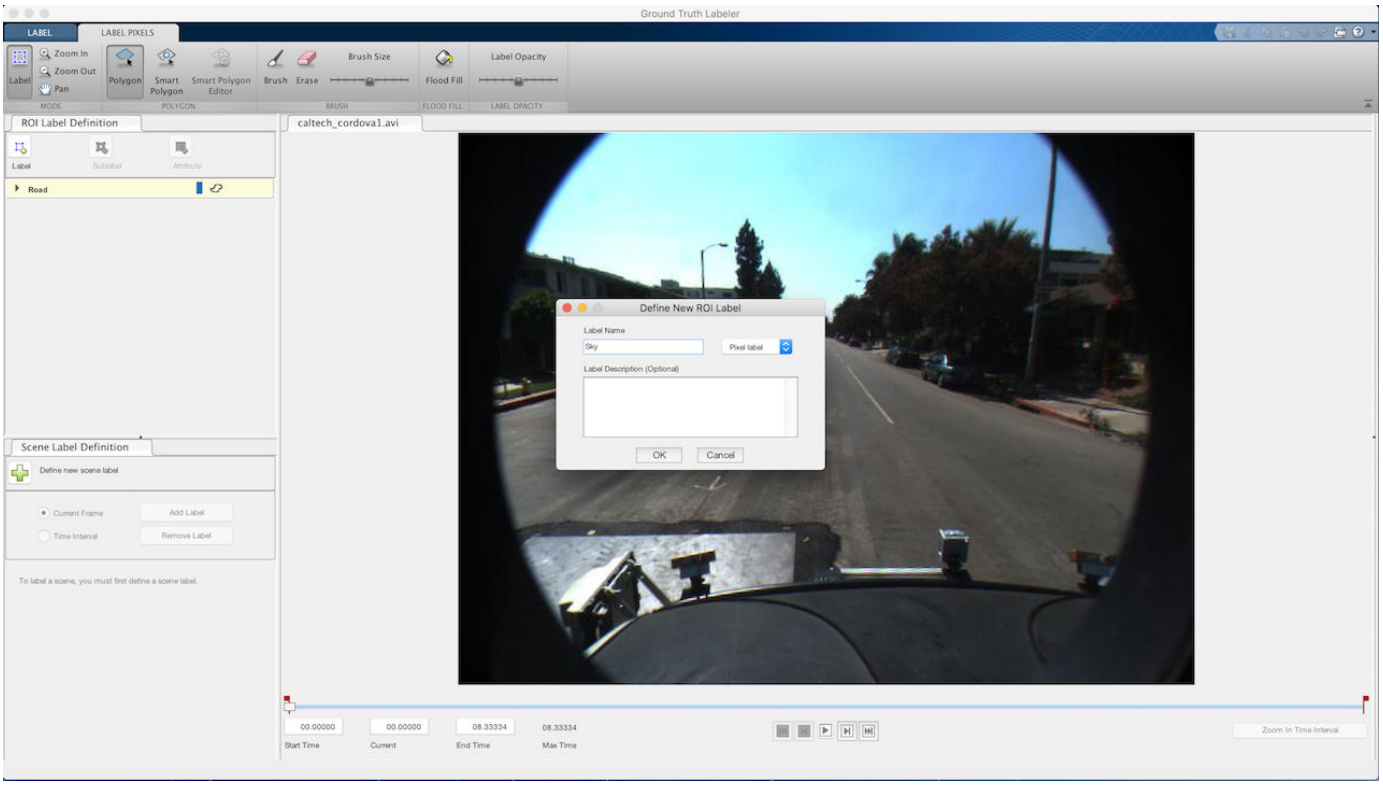
- Create the folder structure `+vision/+labeler` required under the current folder, and copy the automation class into it.

  ```
  mkdir('+vision/+labeler');
  copyfile('RoadAndSkySegmentation.m','+vision/+labeler');
  ```

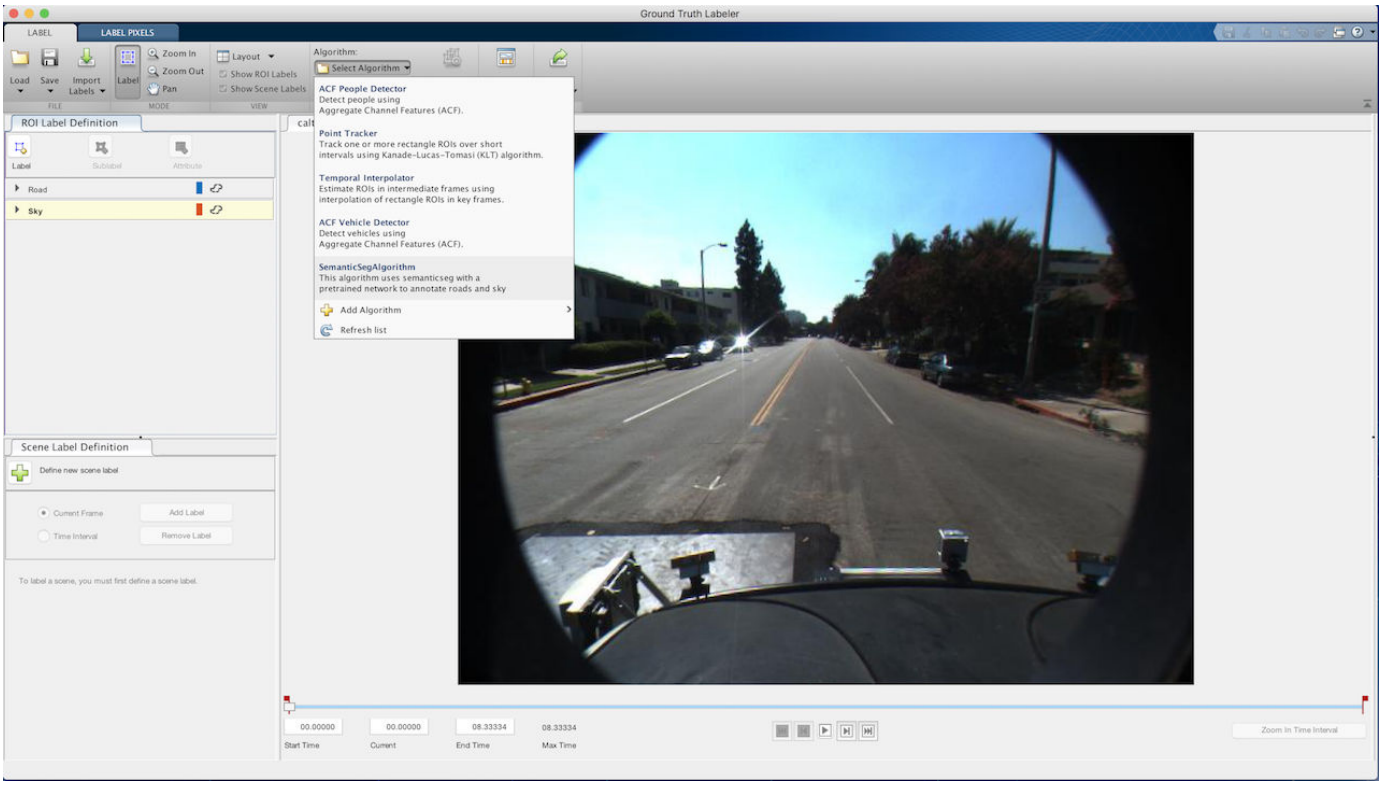
- Open the `groundTruthLabeler` app with custom data to label. For illustration purposes, open the `caltech_cordova1.avi` video.

  ```
  groundTruthLabeler caltech_cordova1.avi
  ```

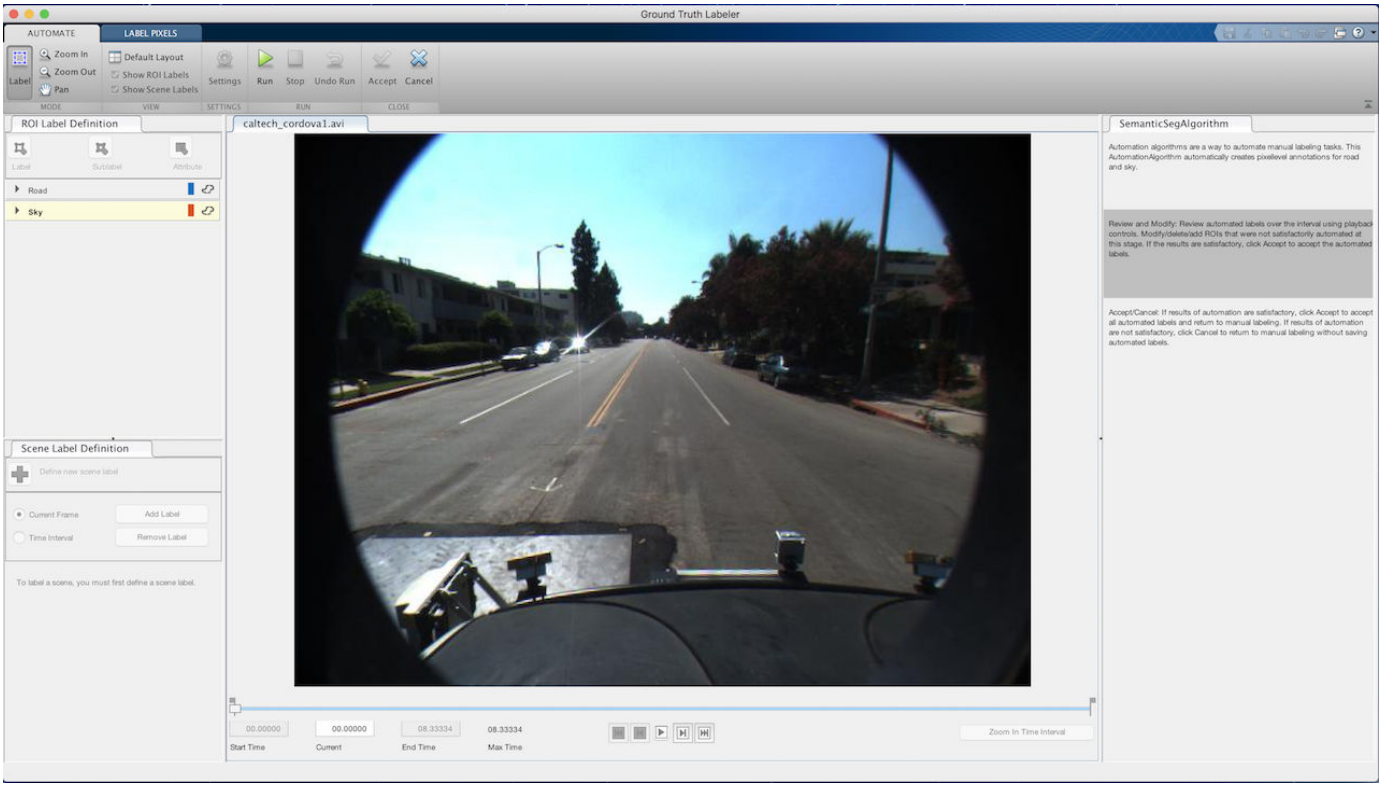
- On the left pane, click the **Define new ROI label** button and define two ROI labels with names `Road` and `Sky`, of type `Pixel label` as shown.



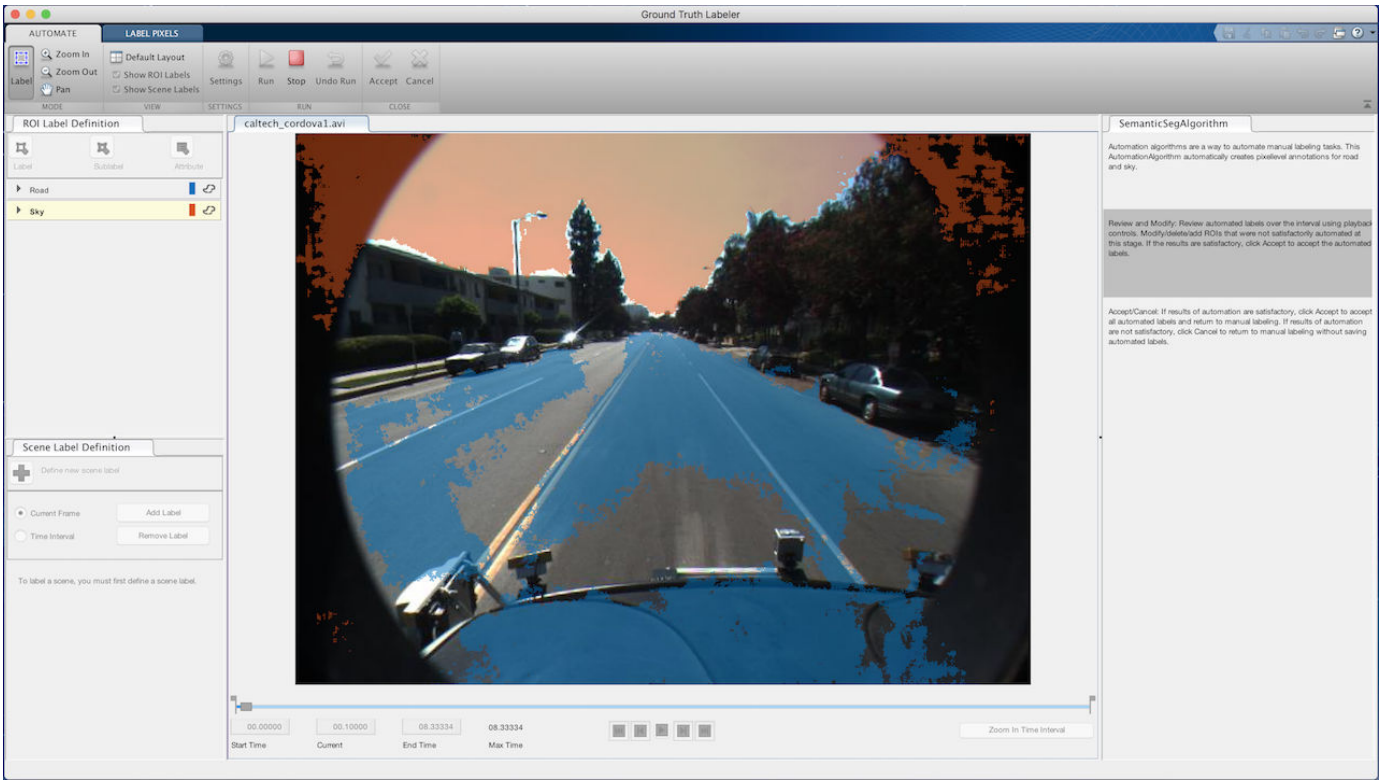
- Click **Algorithm > Select Algorithm > Refresh list**.
- Click **Algorithm > RoadAndSkySegmentation**. If you do not see this option, ensure that the current working folder has a folder called `+vision/+labeler`, with a file named `RoadAndSkySegmentation.m` in it.



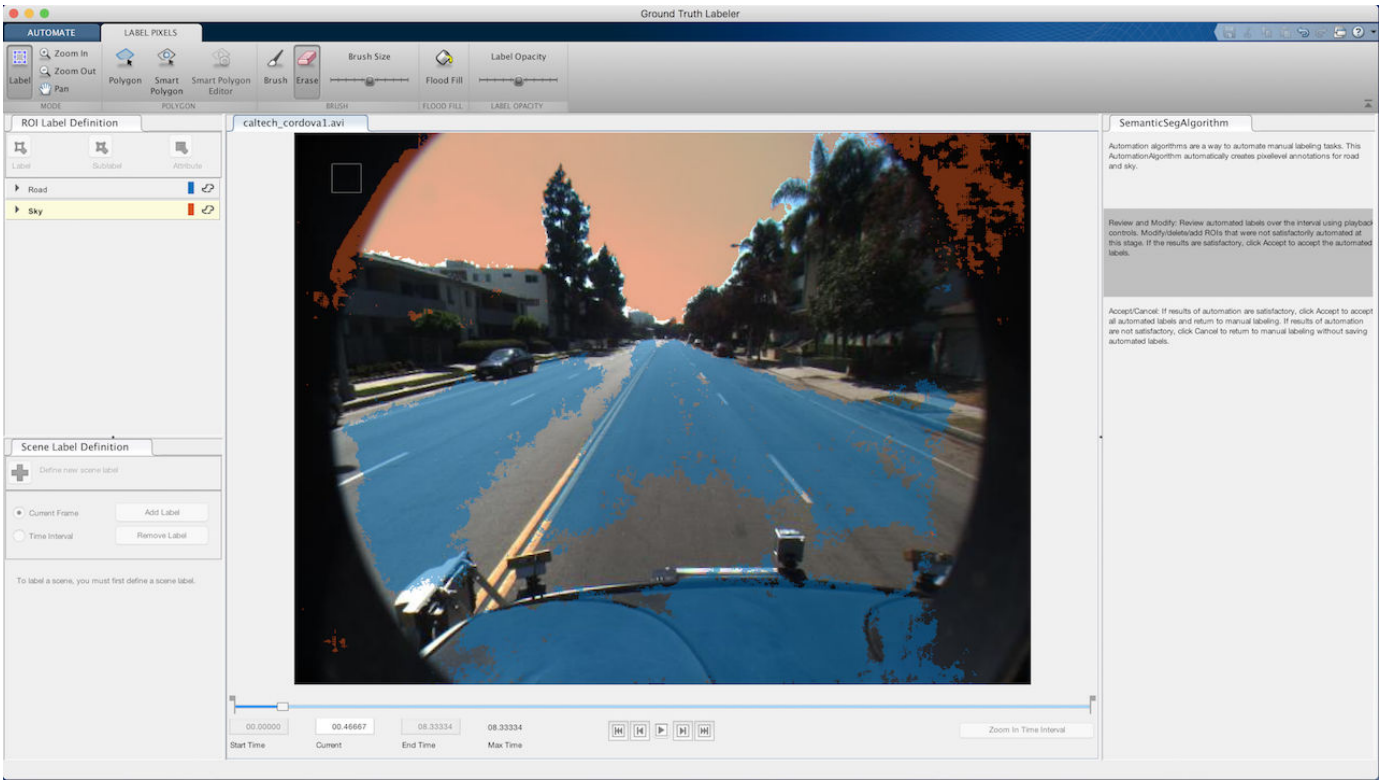
- Click **Automate**. A new panel opens, displaying directions for using the algorithm.



- Click **Run**. The created algorithm executes on each frame of the video, segmenting "Road" and "Sky" categories. After the run is completed, use the slider or arrow keys to scroll through the video and verify the result of the automation algorithm.



- It is evident that regions outside the camera field of view are incorrectly labeled as "Sky", and parts of the ego vehicle itself are marked as "Road". These results indicate that the network has not been previously trained on such data. This workflow allows for making manual corrections to these results, so that an iterative process of training and labeling (sometimes called *active learning* or *human in the loop*) can be used to further refine the accuracy of the network on custom data sets. You can manually tweak the results by using the brush tool in the **Label Pixels** tab and adding or removing pixel annotations. Other tools like flood fill and smart polygons are also available in the **Label Pixels** tab and can be used when appropriate.



- Once you are satisfied with the pixel label categories for the entire video, click **Accept**.

Automation for pixel labeling for the video is complete. You can now proceed with labeling other objects of interest, save the session, or export the results of this labeling run.

Conclusion

This example showed how to use a pretrained semantic segmentation network to accelerate labeling of road and sky pixels in the Ground Truth Labeler app using the `AutomationAlgorithm` interface.

References

1 Brostow, Gabriel J., Jamie Shotton, Julien Fauqueur, and Roberto Cipolla. "Segmentation and Recognition Using Structure from Motion Point Clouds." *ECCV*. 2008.

2 Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. "Semantic Object Classes in Video: A High-Definition Ground Truth Database." *Pattern Recognition Letters*. 2008.

See Also

Apps
Ground Truth Labeler

Objects
`vision.labeler.AutomationAlgorithm`

More About

- “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)
- “Create Automation Algorithm for Labeling” (Computer Vision Toolbox)
- “Automate Ground Truth Labeling of Lane Boundaries” on page 7-17
- “Automate Ground Truth Labeling for Semantic Segmentation” on page 7-29
- “Automate Attributes of Labeled Objects” on page 7-39
- “Train Deep Learning Semantic Segmentation Network Using 3-D Simulation Data” (Deep Learning Toolbox)

Automate Attributes of Labeled Objects

This example shows how to develop a vehicle detection and distance estimation algorithm and use it to automate labeling using the Ground Truth Labeler app. In this example, you will learn how to:

- Develop a computer vision algorithm to detect vehicles in a video, and use the monocular camera configuration to estimate distances to the detected vehicles.
- Use the `AutomationAlgorithm` API to create an automation algorithm. See "Create Automation Algorithm for Labeling" (Computer Vision Toolbox) for details. The created automation algorithm can be used with the Ground Truth Labeler app to automatically label vehicles, along with attributes to store the estimated distances.

The Ground Truth Labeler App

Good ground truth data is crucial for developing driving algorithms and evaluating their performances. However, creating a rich and diverse set of annotated driving data requires significant effort. The Ground Truth Labeler app makes this process efficient. You can use this app as a fully manual labeling tool to mark vehicle bounding boxes, lane boundaries, and other objects of interest for an automated driving system. You can also manually specify attributes of the labeled objects. However, manual labeling requires a significant amount of time and resources. As an alternative, this app provides a framework for creating algorithms to extend and automate the labeling process. You can use the algorithms you create to quickly label entire data sets, automatically annotate the labels with attributes, and then follow it up with a more efficient, shorter manual verification step. You can also edit the results of the automation step to account for challenging scenarios that the automation algorithm might have missed.

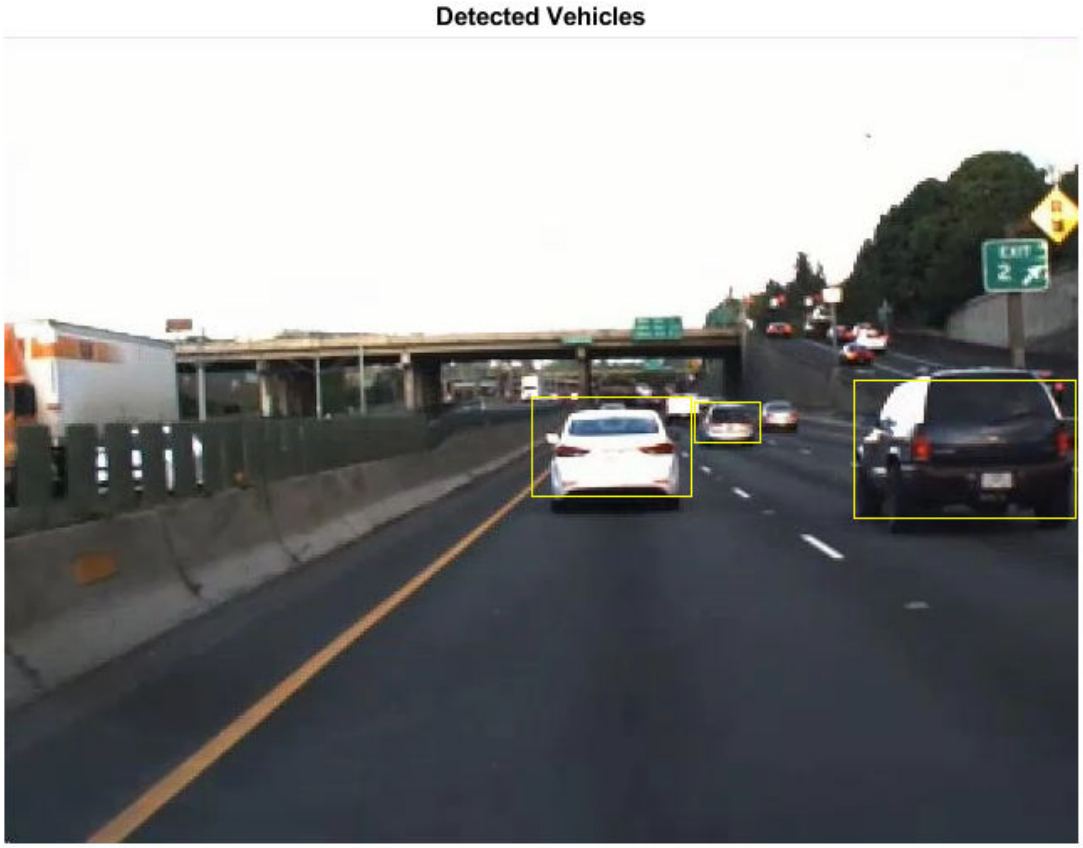
This example describes how to insert a vehicle detection and distance estimation automation algorithm into the automation workflow of the app. This example reuses the ACF Vehicle Detection automation algorithm to first detect vehicles and then automatically estimate the distances of the detected vehicles from the camera mounted on the ego vehicle. The algorithm then creates a label for each detected vehicle, with an attribute specifying the distance to the vehicle.

Detect Vehicles from a Monocular Camera

First, create a vehicle detection algorithm. The "Visual Perception Using Monocular Camera" on page 7-78 example describes how to create a pretrained vehicle detector and configure it to detect vehicle bounding boxes using the calibrated monocular camera configuration. To detect vehicles, try out the algorithm on a single video frame.

```
% Read a frame of interest from a video.
vidObj   = VideoReader('05_highway_lanechange_25s.mp4');
vidObj.CurrentTime = 0.1;
I = readFrame(vidObj);

% Load the monoCamera object.
data = load('FCWDemoMonoCameraSensor.mat', 'sensor');
sensor = data.sensor;

% Load the pretrained detector for vehicles.
detector = vehicleDetectorACF();

% Width of a common vehicle is between 1.5 to 2.5 meters.
vehicleWidth = [1.5, 2.5];
```

```
% Configure the detector to take into account configuration of the camera
% and expected vehicle width
detector = configureDetectorMonoCamera(detector, sensor, vehicleWidth);

% Detect vehicles and show the bounding boxes.
[bboxes, ~] = detect(detector, I);
Iout = insertShape(I, 'rectangle', bboxes);
figure;
imshow(Iout)
title('Detected Vehicles')
```



Estimate Distances to Detected Vehicles

Now that vehicles have been detected, estimate distances to the detected vehicles from the camera in world coordinates. `monoCamera` provides an `imageToVehicle` method to convert points from image coordinates to vehicle coordinates. This can be used to estimate the distance along the ground from the camera to the detected vehicles. The example specifies the distance as the center point of the detected vehicle, along the ground directly below it.

```
% Find the midpoint for each bounding box in image coordinates.
midPtsImg = [bboxes(:,1)+bboxes(:,3)/2  bboxes(:,2)+bboxes(:,4)./2];
midPtsWorld = imageToVehicle(sensor, midPtsImg);
x = midPtsWorld(:,1);
```

```
y = midPtsWorld(:,2);
distance  = sqrt(x.^2 + y.^2);

% Display vehicle bounding boxes and annotate them with distance in meters.
distanceStr = cellstr([num2str(distance) repmat(' m',[length(distance) 1])]);
Iout = insertObjectAnnotation(I, 'rectangle', bboxes, distanceStr);
imshow(Iout)
title('Distances of Vehicles from Camera')
```

Prepare the Vehicle Detection and Distance Estimation Automation Class

Incorporate the vehicle detection and distance estimation automation class into the automation workflow of the app. See “Create Automation Algorithm for Labeling” (Computer Vision Toolbox) for more details. Start with the existing ACF Vehicle Detection automation algorithm to perform vehicle detection with a calibrated monocular camera. Then modify the algorithm to perform attribute automation. In this example, use the distance of the vehicle from the camera as an attribute of the detected vehicle. This section describes the steps for making changes to the existing ACF Vehicle Detection automation algorithm class.

Step 1 contains properties that define the name and description of the algorithm, and the directions for using the algorithm.

```
%-----------------------------------------------------------------------
% Define algorithm Name, Description, and UserDirections.
properties(Constant)

    %Name: Algorithm Name
    %   Character vector specifying name of algorithm.
    Name = 'Vehicle Detection and Distance Estimation';

    % Description: Provide a one-line description for your algorithm.
    Description = 'Detect vehicles using a pretrained ACF vehicle detector and compute distanc

    % UserDirections: Provide a set of directions that are displayed
    %                 when this algorithm is invoked. The directions
    %                 are to be provided as a cell array of character
    %                 vectors, with each element of the cell array
    %                 representing a step in the list of directions.
    UserDirections = {...
        'Define a rectangle ROI Label to label vehicles.',...
        'For the label definition created, define an Attribute with name Distance, type Numer
        'Run the algorithm',...
        'Manually inspect and modify results if needed'};
end
```

Step 2 contains the custom properties needed to support vehicle detection and distance estimation automation

```
%-----------------------------------------------------------------------
% Vehicle Detector Properties
%-----------------------------------------------------------------------
properties
    %SelectedLabelName Selected label name
    %   Name of selected label. Vehicles detected by the algorithm will
    %   be assigned this variable name.
    SelectedLabelName

    %Detector Detector
    %   Pretrained vehicle detector, an object of class
    %   acfObjectDetector.
    Detector

    %VehicleModelName Vehicle detector model name
    %   Name of pretrained vehicle detector model.
    VehicleModelName = 'full-view';

    %OverlapThreshold Overlap threshold
    %   Threshold value used to eliminate overlapping bounding boxes
    %   around the reference bounding box, between 0 and 1. The
    %   bounding box overlap ratio denominator, 'RatioType' is set to
    %   'Min'
    OverlapThreshold = 0.65;

    %ScoreThreshold Classification Score Threshold
    %   Threshold value used to reject detections with low detection
    %   scores.
    ScoreThreshold = 30;

    %ConfigureDetector Boolean value to decide on configuring the detector
    %   Boolean value which decides if the detector is configured using
    %   monoCamera sensor.
    ConfigureDetector = true;
```

```
        %SensorObj monoCamera sensor
        %   Monocular Camera Sensor object used to configure the detector.
        %   A configured detector will run faster and can potentially
        %   result in better detections.
        SensorObj = [];

        %SensorStr monoCamera sensor variable name
        %   Monocular Camera Sensor object variable name used to configure
        %   the detector.
        SensorStr = '';

        %VehicleWidth Vehicle Width
        %   Vehicle Width used to configure the detector, specified as
        %   [minWidth, maxWidth] describing the approximate width of the
        %   object in world units.
        VehicleWidth = [1.5 2.5];

        %VehicleLength Vehicle Length
        %   Vehicle Length used to configure the detector, specified as
        %   [minLength, maxLength] describing the approximate length of the
        %   object in world units.
        VehicleLength = [ ];
    end

    %--------------------------------------------------------------------
    % Attribute automation Properties
    %--------------------------------------------------------------------
    properties (Constant, Access = private)

        % Flag to enable Distance attribute estimation automation
        AutomateDistanceAttriibute = true;

        % Supported Distance attribute name.
        % The label must have an attribute with the name specified.
        SupportedDistanceAttribName = 'Distance';
    end

    properties (Access = private)

        % Actual attribute name for distance
        DistanceAttributeName;

        % Flag to check if attribute specified is a valid distance
        % attribute
        HasValidDistanceAttribute = false;
    end
```

Step 3 initializes properties.

```
    %--------------------------------------------------------------------
    % Initialize sensor, detector and other relevant properties.
    function initialize(algObj, ~)

        % Store the name of the selected label definition. Use this
        % name to label the detected vehicles.
        algObj.SelectedLabelName = algObj.SelectedLabelDefinitions.Name;

        % Initialize the vehicle detector with a pretrained model.
        algObj.Detector = vehicleDetectorACF(algObj.VehicleModelName);
```

```
        % Initialize parameters to compute vehicle distance
        if algObj.AutomateDistanceAttriibute
            initializeAttributeParams(algObj);
        end
    end

    function initializeAttributeParams(algObj)
        % Initialize properties relevant to attribute automation.

        % The label must have an attribute with name Distance and type
        % Numeric Value.
        hasAttribute = isfield(algObj.ValidLabelDefinitions, 'Attributes') && ...
                       isstruct(algObj.ValidLabelDefinitions.Attributes);
        if hasAttribute
           attributeNames = fieldnames(algObj.ValidLabelDefinitions.Attributes);
           idx = find(contains(attributeNames, algObj.SupportedDistanceAttribName));
           if ~isempty(idx)
              algObj.DistanceAttributeName = attributeNames{idx};
              algObj.HasValidDistanceAttribute = validateDistanceType(algObj);
           end
        end
    end

    function tf = validateDistanceType(algObj)
        % Validate the attribute type.

        tf = isfield(algObj.ValidLabelDefinitions.Attributes, algObj.DistanceAttributeName) && ..
             isfield(algObj.ValidLabelDefinitions.Attributes.(algObj.DistanceAttributeName), 'Defa
             isnumeric(algObj.ValidLabelDefinitions.Attributes.(algObj.DistanceAttributeName).Defa
    end
```

Step 4 contains the updated `run` method to compute the distance of the detected cars and writes the label and attribute info to the output labels.

```
    %--------------------------------------------------------------------
    function autoLabels = run(algObj, I)

        autoLabels = [];

        % Configure the detector.
        if algObj.ConfigureDetector && ~isa(algObj.Detector,'acfObjectDetectorMonoCamera')
            vehicleSize = [algObj.VehicleWidth;algObj.VehicleLength];
            algObj.Detector = configureDetectorMonoCamera(algObj.Detector, algObj.SensorObj, vehic
        end

        % Detect vehicles using the initialized vehicle detector.
        [bboxes, scores] = detect(algObj.Detector, I,...
            'SelectStrongest', false);

        [selectedBbox, selectedScore] = selectStrongestBbox(bboxes, scores, ...
            'RatioType', 'Min', 'OverlapThreshold', algObj.OverlapThreshold);

        % Reject detections with detection score lower than
        % ScoreThreshold.
        detectionsToKeepIdx = (selectedScore > algObj.ScoreThreshold);
        selectedBbox = selectedBbox(detectionsToKeepIdx,:);

        if ~isempty(selectedBbox)
            % Add automated labels at bounding box locations detected
            % by the vehicle detector, of type Rectangle having name of
```

```
        % the selected label.
        autoLabels.Name     = algObj.SelectedLabelName;
        autoLabels.Type     = labelType.Rectangle;
        autoLabels.Position = selectedBbox;

        if (algObj.AutomateDistanceAttriibute && algObj.HasValidDistanceAttribute)
            attribName = algObj.DistanceAttributeName;
            % Attribute value is of type 'Numeric Value'
            autoLabels.Attributes = computeVehicleDistances(algObj, selectedBbox, attribName);
        end
    else
        autoLabels = [];
    end
end

function midPts = helperFindBottomMidpoint(bboxes)
   % Find midpoint of bottom edge of the bounding box.

   xBL = bboxes(:,1);
   yBL = bboxes(:,2);

   xM = xBL + bboxes(:,3)/2;
   yM = yBL + + bboxes(:,4)./2;
   midPts = [xM yM];

end

function distances= computeDistances(algObj, bboxes)
   % Helper function to compute vehicle distance.

   midPts = helperFindBottomMidpoint(bboxes);
   xy = algObj.SensorObj.imageToVehicle(midPts);
   distances  = sqrt(xy(:,1).^2 + xy(:,2).^2);

end

function attribS = computeVehicleDistances(algObj, bboxes, attribName)
    % Compute vehicle distance.

    numCars = size(bboxes, 1);
    attribS = repmat(struct(attribName, 0), [numCars, 1]);

    for i=1:numCars
        distanceVal = computeDistances(algObj, bboxes(i,:));
        attribS(i).(attribName) = distanceVal;
    end
end
```

Use the Vehicle Detection and Distance Estimation Automation Class in the App

The packaged version of the vehicle distance computation algorithm is available in the `VehicleDetectionAndDistanceEstimation` class. To use this class in the app:

- Create the folder structure required under the current folder, and copy the automation class into it.

```
mkdir('+vision/+labeler');
copyfile(fullfile(matlabroot,'examples','driving','main','VehicleDetectionAndDistanceEstimat
```

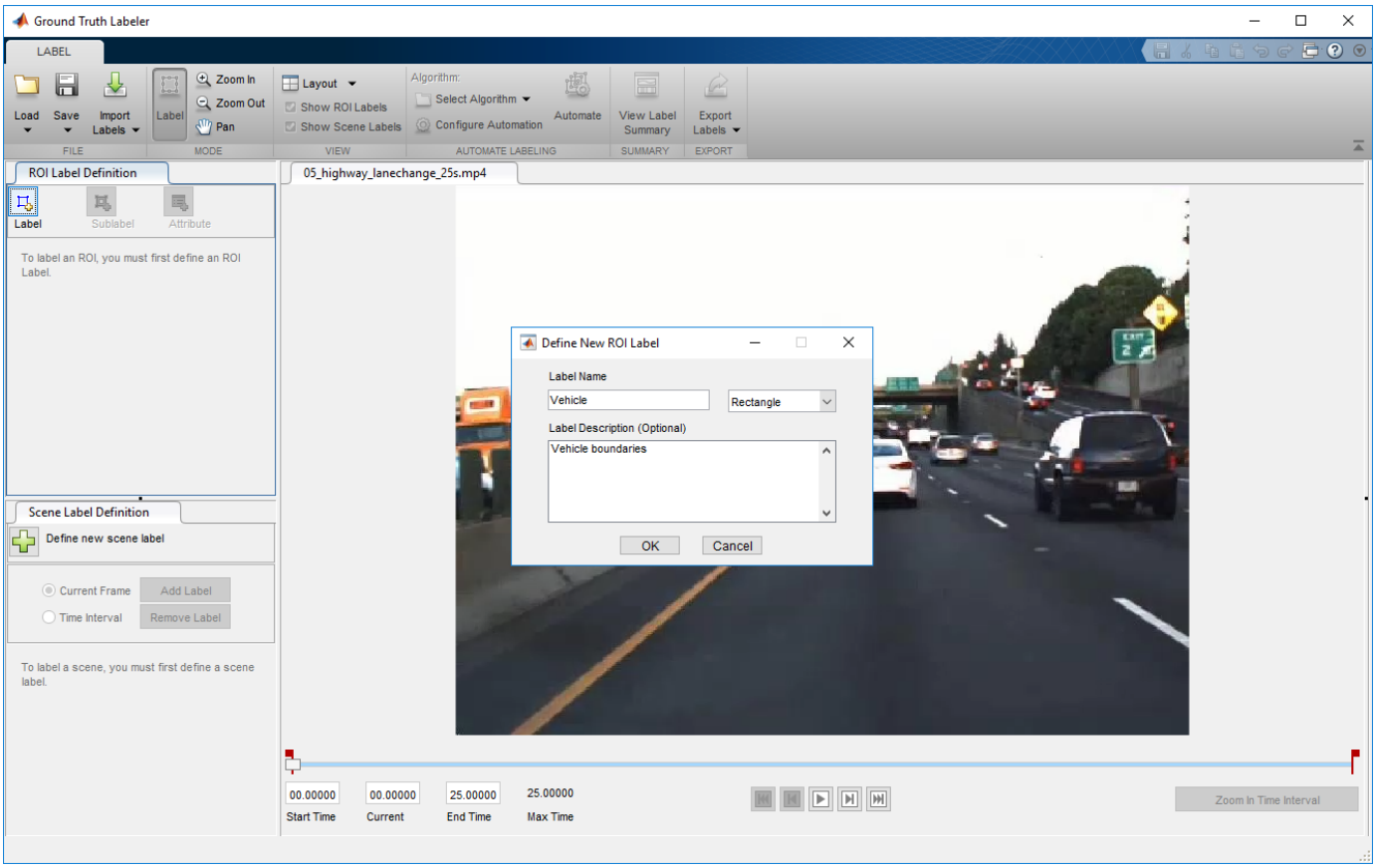
- Load the `monoCamera` information into the workspace. This camera sensor information is suitable for the camera used in the video used in this example, `05_highway_lanechange_25s.mp4`. If you load a different video, use the sensor information appropriate for that video.

```
load('FCWDemoMonoCameraSensor.mat', 'sensor')
```

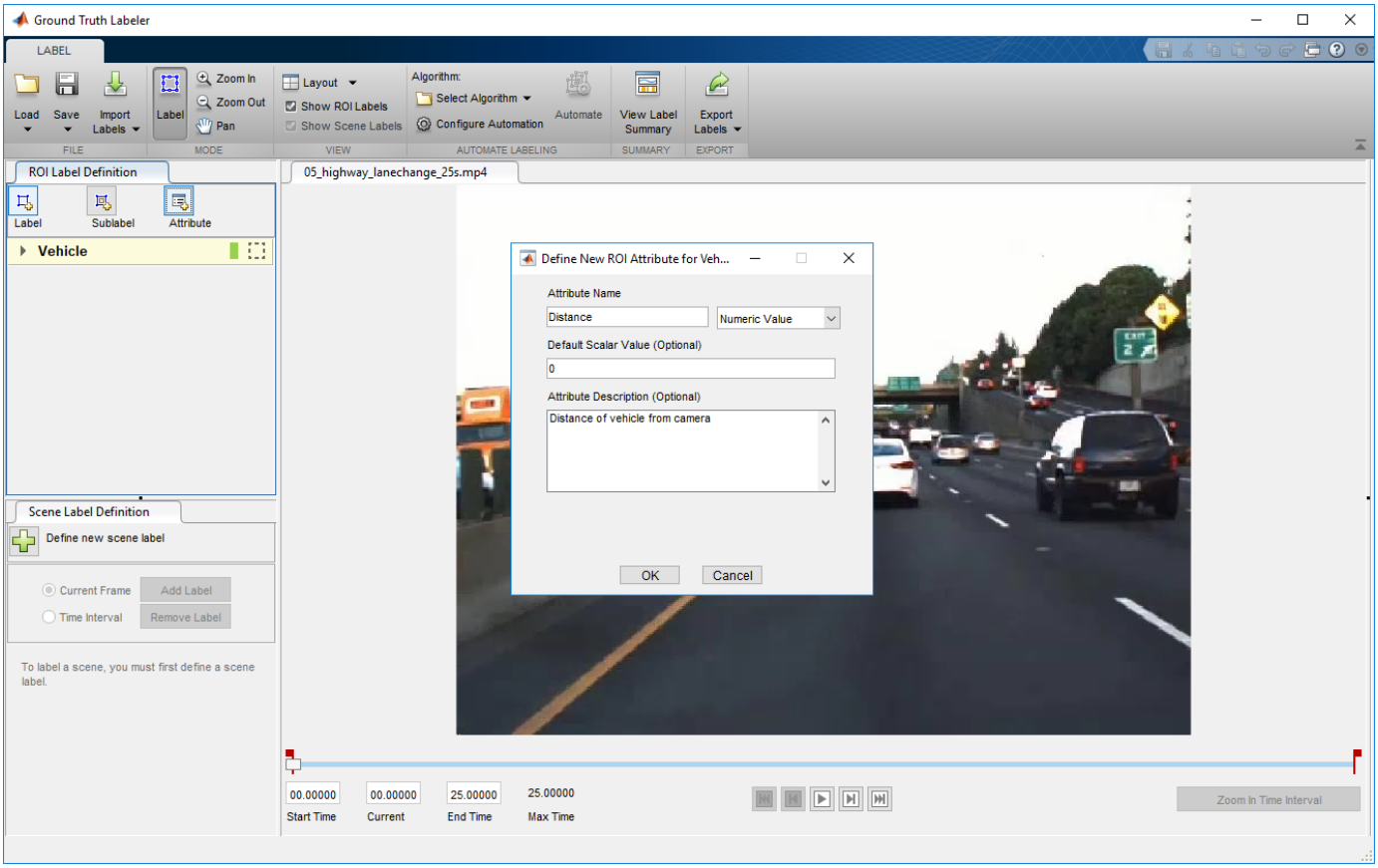
- Open the `groundTruthLabeler` app.

```
groundTruthLabeler 05_highway_lanechange_25s.mp4
```

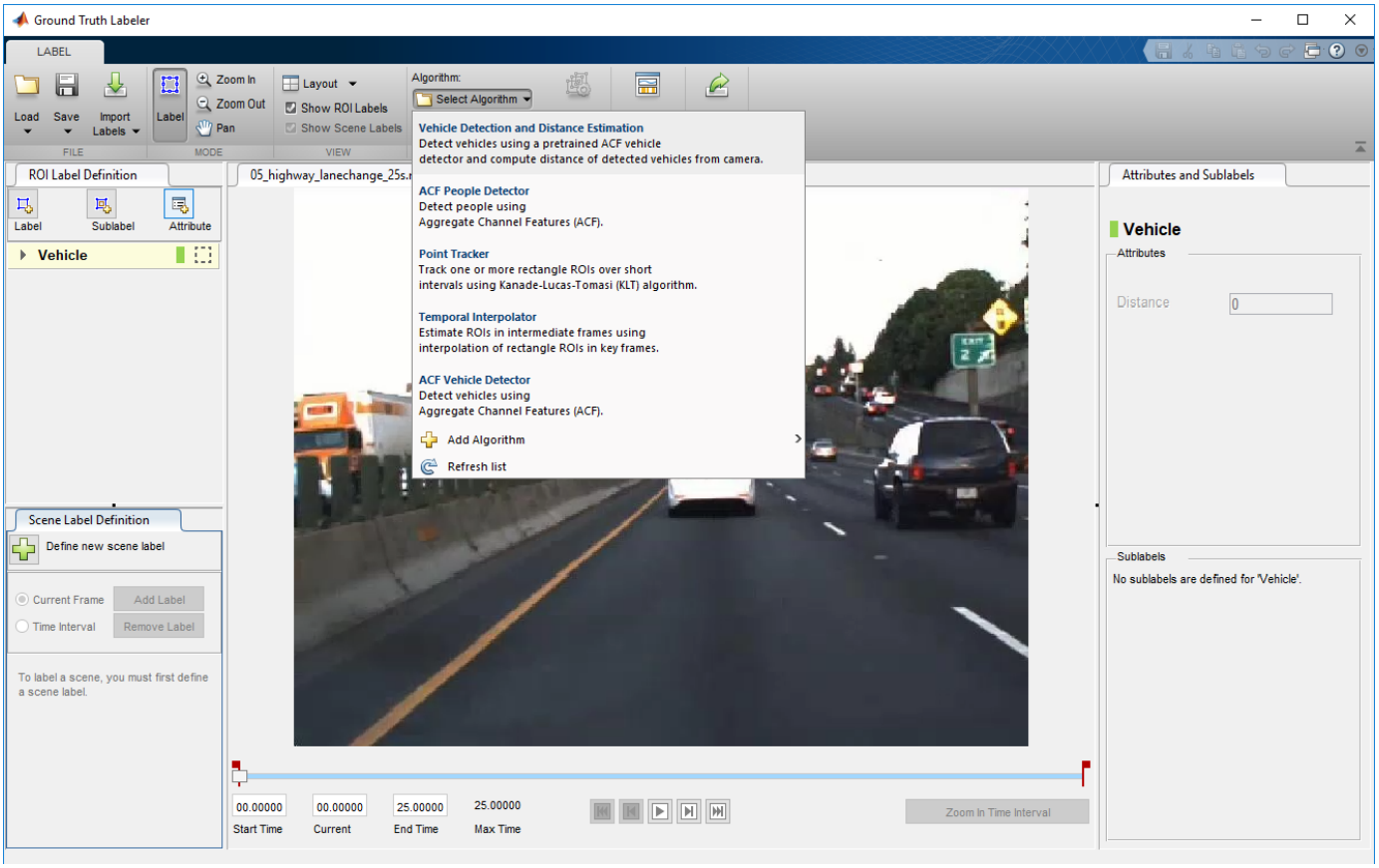
- In the **ROI Label Definition** pane on the left, click **Label**. Define a label with name `Vehicle` and type `Rectangle`. Optionally, add a label description. Then click **OK**.



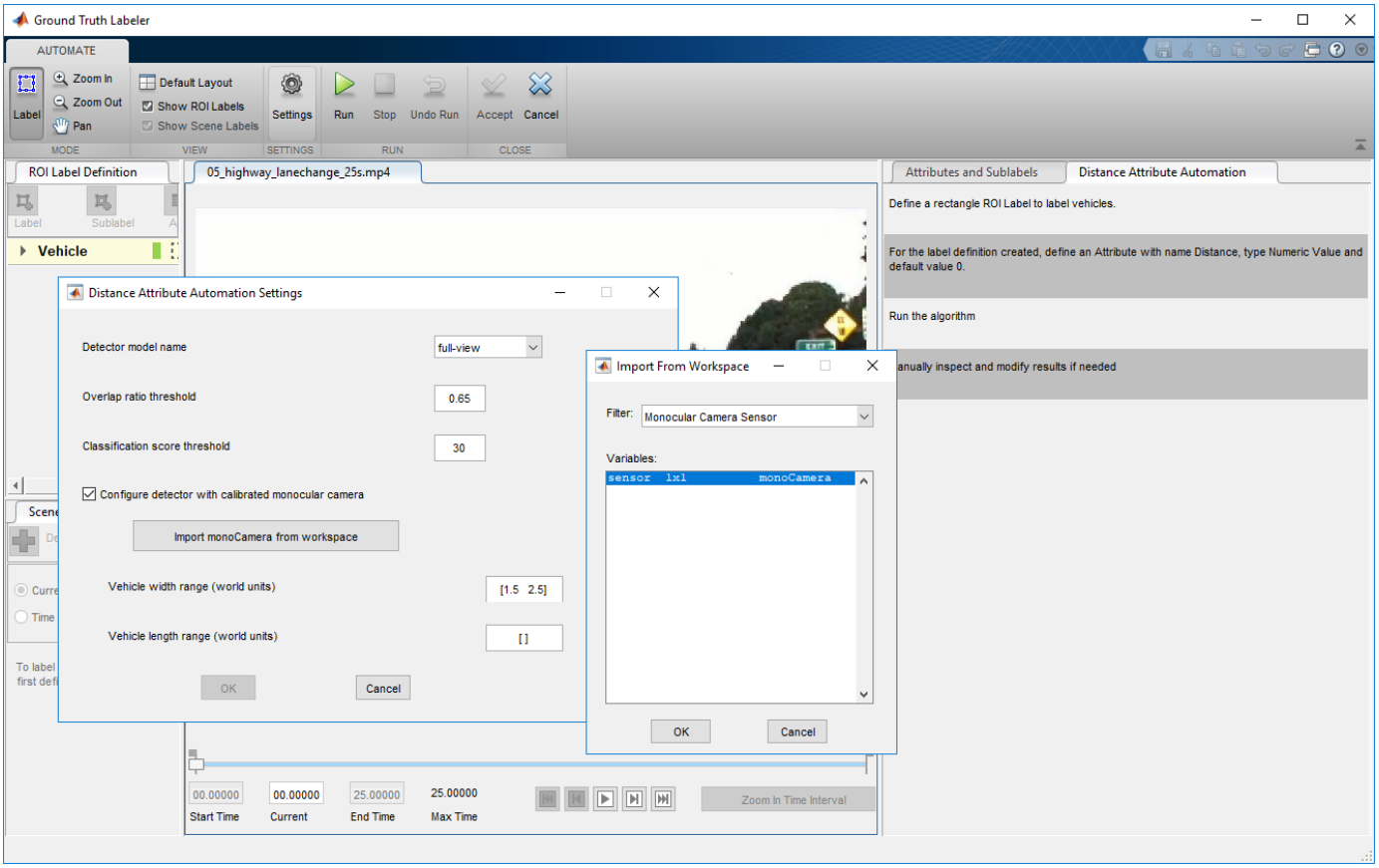
- In the **ROI Label Definition** pane on the left, click **Attribute**. Define an attribute with name `Distance`, type `Numeric Value`, and default value `0`. Optionally, add an attribute description. Then click **OK**.



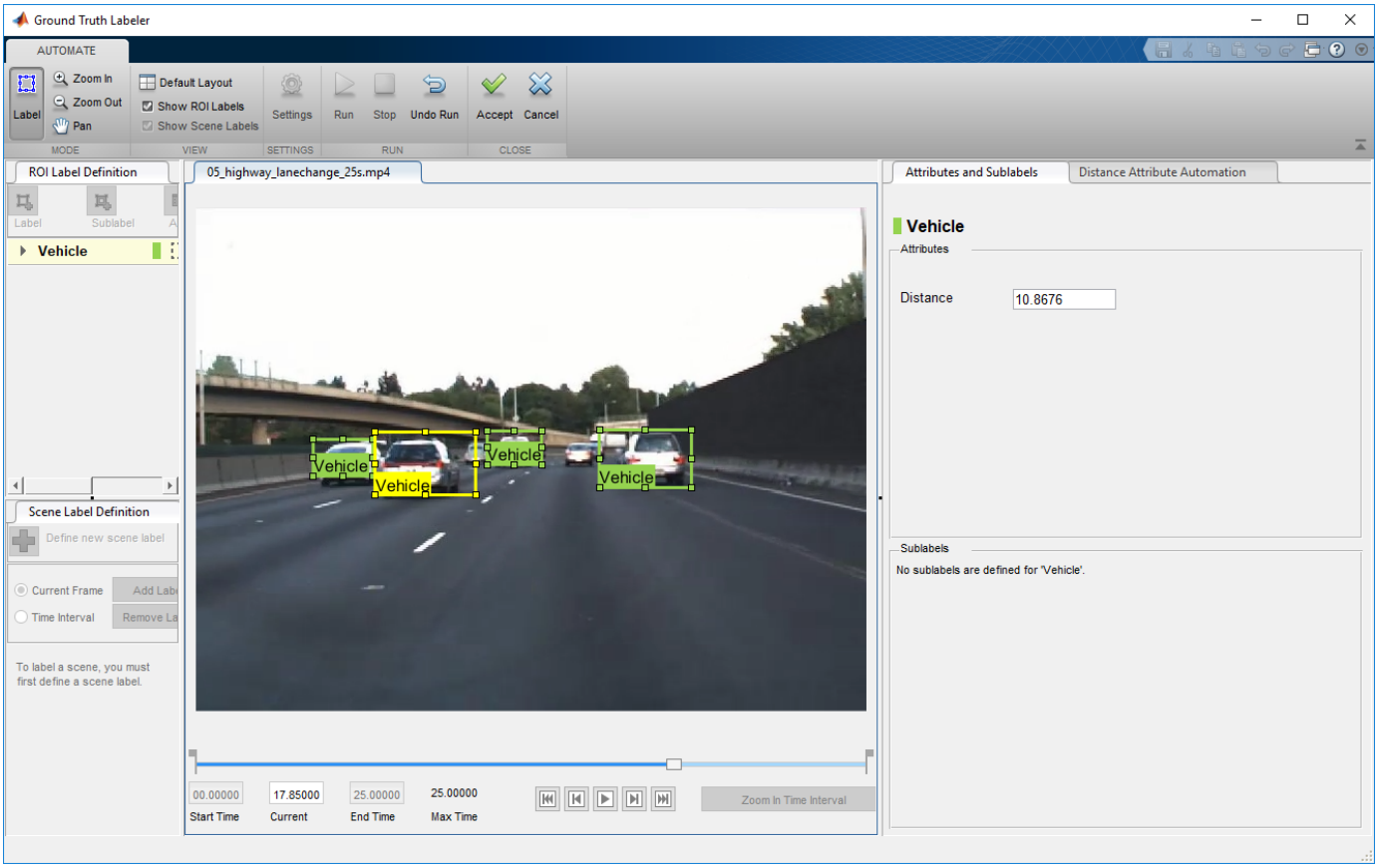
- Select **Algorithm > Select Algorithm > Refresh list**.
- Select **Algorithm > Vehicle Detection and Distance Estimation**. If you do not see this option, ensure that the current working folder has a folder called `+vision/+labeler`, with a file named `VehicleDetectionAndDistanceEstimation.m` in it.



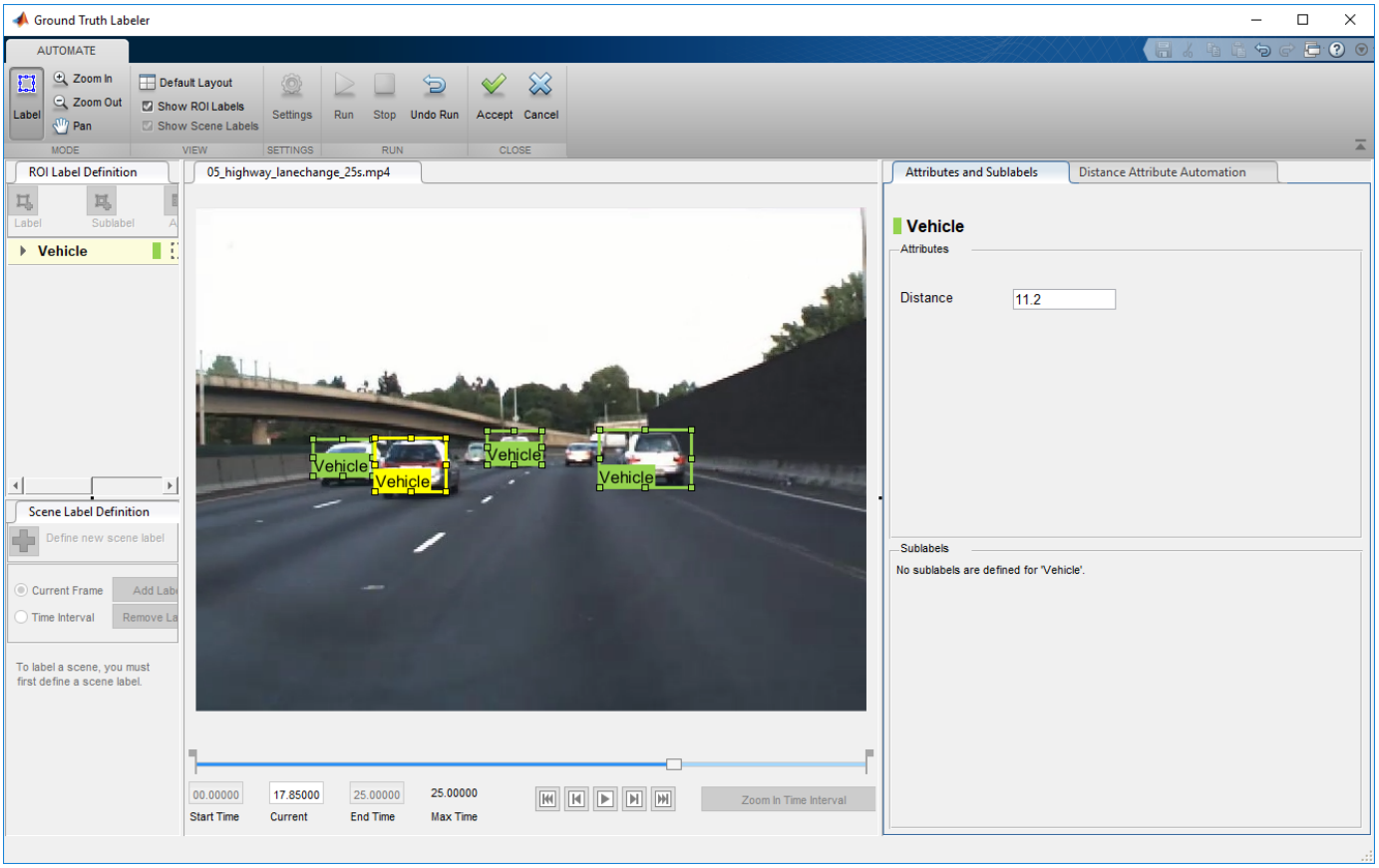
- Click **Automate**. A new tab opens, displaying directions for using the algorithm.
- Click **Settings**, and in the dialog box that opens, enter `sensor` in the first text box. Modify other parameters if needed before clicking **OK**.



- Click **Run**. The vehicle detection and distance computation algorithm progresses through the video. Notice that the results are not satisfactory in some of the frames.
- After the run is completed, use the slider or arrow keys to scroll across the video to locate the frames where the algorithm failed.



- Manually tweak the results by either moving the vehicle bounding box or by changing the distance value. You can also delete the bounding boxes and the associated distance values.



- Once you are satisfied with the vehicle bounding boxes and their distances for the entire video, click **Accept**.

The automated vehicle detection and distance attribute labeling on the video is complete. You can now label other objects of interest and set their attributes, save the session, or export the results of this labeling run.

Conclusion

This example showed the steps to incorporate a vehicle detection and distance attribute estimation automation algorithm into the Ground Truth Labeler app. You can extend this concept to other custom algorithms to extend the functionality of the app.

See Also

Apps
Ground Truth Labeler

Objects
`monoCamera` | `vision.labeler.AutomationAlgorithm`

More About

- “Create Automation Algorithm for Labeling” (Computer Vision Toolbox)

- “Use Sublabels and Attributes to Label Ground Truth Data” (Computer Vision Toolbox)
- “Visual Perception Using Monocular Camera” on page 7-78

Evaluate Lane Boundary Detections Against Ground Truth Data

This example shows how to compare ground truth data against results of a lane boundary detection algorithm. It also illustrates how this comparison can be used to tune algorithm parameters to get the best detection results.

Overview

Ground truth data is usually available in image coordinates, whereas boundaries are modeled in the vehicle coordinate system. Comparing the two involves a coordinate conversion and thus requires extra care in interpreting the results. Driving decisions are based on distances in the vehicle coordinate system. Therefore, it is more useful to express and understand accuracy requirements using physical units in the vehicle coordinates rather than pixel coordinates.

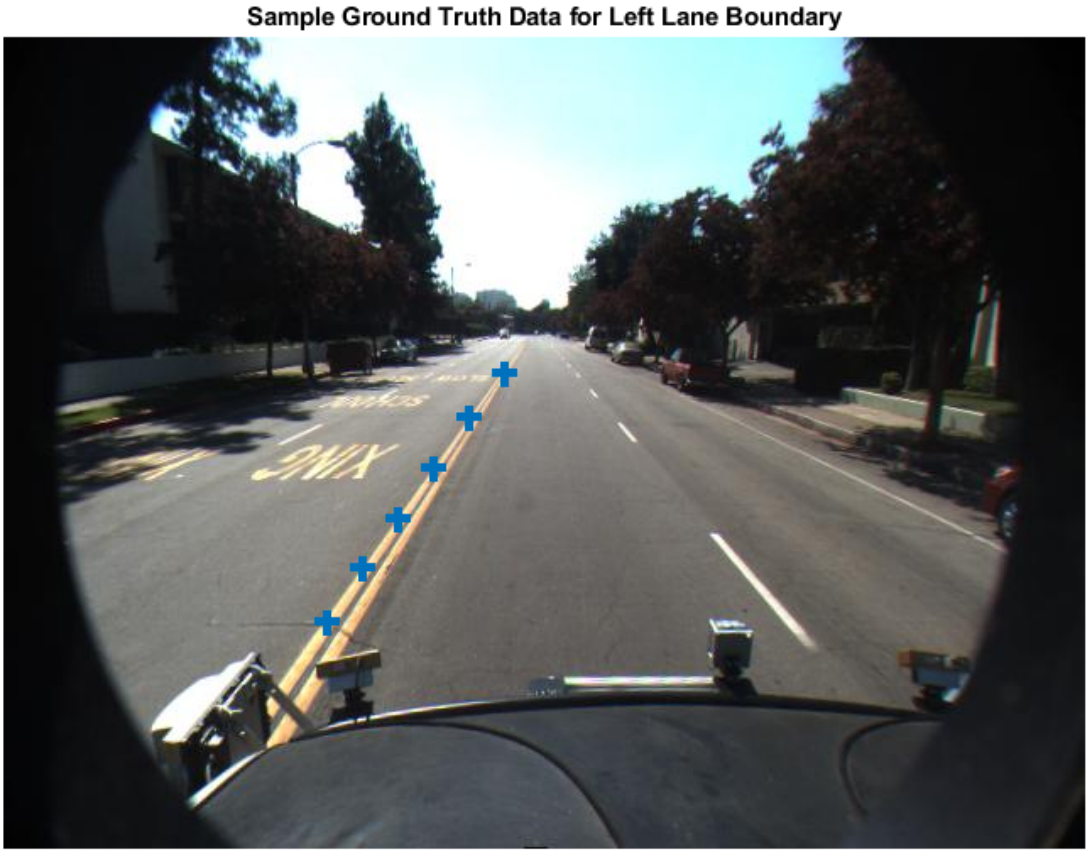
The `MonoCameraExample` describes the internals of a monocular camera sensor and the process of modeling lane boundaries. This example shows how to evaluate the accuracy of these models against manually validated ground truth data. After establishing a comparison framework, the framework is extended to fine-tune parameters of a boundary detection algorithm for optimal performance.

Load and Prepare Ground Truth Data

You can use the Ground Truth Labeler app to mark and label lane boundaries in a video. These annotated lane boundaries are represented as sets of points placed along the boundaries of interest. Having a rich set of manually annotated lane boundaries for various driving scenarios is critical in evaluating and fine-tuning automatic lane boundary detection algorithms. An example set for the `caltech_cordova1.avi` video file is available with the toolbox.

Load predefined left and right ego lane boundaries specified in image coordinates. Each boundary is represented by a set of M-by-2 numbers representing M pixel locations along that boundary. Each video frame has at most two such sets representing the left and the right lane.

```
loaded = load('caltech_cordova1_EgoBoundaries.mat');
sensor = loaded.sensor; % Associated monoCamera object
gtImageBoundaryPoints = loaded.groundTruthData.EgoLaneBoundaries;

% Show a sample of the ground truth at this frame index
frameInd = 36;

% Load the video frame
frameTimeStamp = seconds(loaded.groundTruthData(frameInd,:).Time);
videoReader = VideoReader(loaded.videoName);
videoReader.CurrentTime = frameTimeStamp;
frame = videoReader.readFrame();

% Obtain the left lane points for this frame
boundaryPoints = gtImageBoundaryPoints{frameInd};
leftLanePoints = boundaryPoints{1};

figure
imshow(frame)
hold on
plot(leftLanePoints(:,1), leftLanePoints(:,2),'+','MarkerSize',10,'LineWidth',4);
title('Sample Ground Truth Data for Left Lane Boundary');
```



Convert the ground truth points from image coordinates to vehicle coordinates to allow for direct comparison with boundary models. To perform this conversion, use the `imageToVehicle` function with the associated `monoCamera` object to perform this conversion.

```
gtVehicleBoundaryPoints = cell(numel(gtImageBoundaryPoints),1);
for frameInd = 1:numel(gtImageBoundaryPoints)
    boundaryPoints   = gtImageBoundaryPoints{frameInd};
    if ~isempty(boundaryPoints)
        ptsInVehicle = cell(1, numel(boundaryPoints));
        for cInd = 1:numel(boundaryPoints)
            ptsInVehicle{cInd} = imageToVehicle(sensor, boundaryPoints{cInd});
        end
        gtVehicleBoundaryPoints{frameInd} = ptsInVehicle;
    end
end
```

Model Lane Boundaries Using a Monocular Sensor

Run a lane boundary modeling algorithm on the sample video to obtain the test data for the comparison. Here, reuse the `helperMonoSensor` module introduced in the "Visual Perception Using Monocular Camera" on page 7-78 example. While processing the video, an additional step is needed to return the detected boundary models. This logic is wrapped in a helper function, `detectBoundaries`, defined at the end of this example.

```
monoSensor  = helperMonoSensor(sensor);
boundaries  = detectBoundaries(loaded.videoName, monoSensor);
```

Evaluate Lane Boundary Models

Use the `evaluateLaneBoundaries` function to find the number of boundaries that match those boundaries in ground truth. A ground truth is assigned to a test boundary only if all points of the ground truth are within a specified distance, laterally, from the corresponding test boundary. If multiple ground truth boundaries satisfy this criterion, the one with the smallest maximum lateral distance is chosen. The others are marked as false positives.

```
threshold = 0.25; % in vehicle coordinates (meters)
[numMatches, numMisses, numFalsePositives, assignments] = ...
    evaluateLaneBoundaries(boundaries, gtVehicleBoundaryPoints, threshold);
disp(['Number of matches: ', num2str(numMatches)]);
disp(['Number of misses: ', num2str(numMisses)]);
disp(['Number of false positives: ', num2str(numFalsePositives)]);
```

```
Number of matches: 321
Number of misses: 124
Number of false positives: 25
```

You can use these raw counts to compute other statistics such as precision, recall, and the F1 score:

```
precision = numMatches/(numMatches+numFalsePositives);
disp(['Precision: ', num2str(precision)]);

recall = numMatches/(numMatches+numMisses);
disp(['Sensitivity/Recall: ', num2str(recall)]);

f1Score = 2*(precision*recall)/(precision+recall);
disp(['F1 score: ', num2str(f1Score)]);
```

```
Precision: 0.92775
Sensitivity/Recall: 0.72135
F1 score: 0.81163
```

Visualize Results Using a Bird's-Eye Plot

`evaluateLaneBoundaries` additionally returns the assignment indices for every successful match between the ground truth and test boundaries. This can be used to visualize the detected and ground truth boundaries to gain a better understanding of failure modes.

Find a frame that has one matched boundary and one false positive. The ground truth data for each frame has two boundaries. So, a candidate frame will have two assignment indices, with one of them being 0 to indicate a false positive.

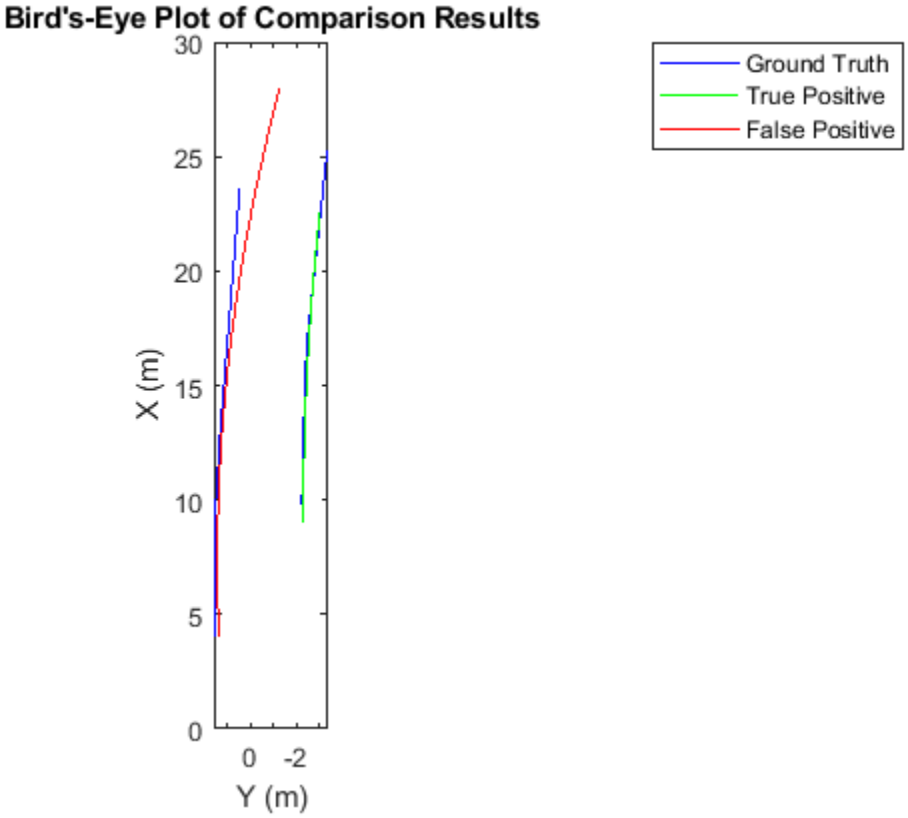
```
hasMatch           = cellfun(@(x)numel(x)==2, assignments);
hasFalsePositive   = cellfun(@(x)nnz(x)==1, assignments);
frameInd           = find(hasMatch&hasFalsePositive,1,'first');
frameVehiclePoints = gtVehicleBoundaryPoints{frameInd};
frameImagePoints   = gtImageBoundaryPoints{frameInd};
frameModels        = boundaries{frameInd};
```

Use the `assignments` output of `evaluateLaneBoundaries` to find the models that matched (true positives) and models that had no match (false positives) in ground truth.

```
matchedModels    = frameModels(assignments{frameInd}~=0);
fpModels         = frameModels(assignments{frameInd}==0);
```

Set up a bird's-eye plot and visualize the ground truth points and models on it.

```
bep = birdsEyePlot();
gtPlotter = laneBoundaryPlotter(bep,'DisplayName','Ground Truth',...
    'Color','blue');
tpPlotter = laneBoundaryPlotter(bep,'DisplayName','True Positive',...
    'Color','green');
fpPlotter = laneBoundaryPlotter(bep,'DisplayName','False Positive',...
    'Color','red');

plotLaneBoundary(gtPlotter, frameVehiclePoints);
plotLaneBoundary(tpPlotter, matchedModels);
plotLaneBoundary(fpPlotter, fpModels);
title('Bird''s-Eye Plot of Comparison Results');
```



Visualize Results on a Video in Camera and Bird's-Eye View

To get a better context of the result, you can also visualize ground truth points and the boundary models on the video.

Get the frame corresponding to the frame of interest.

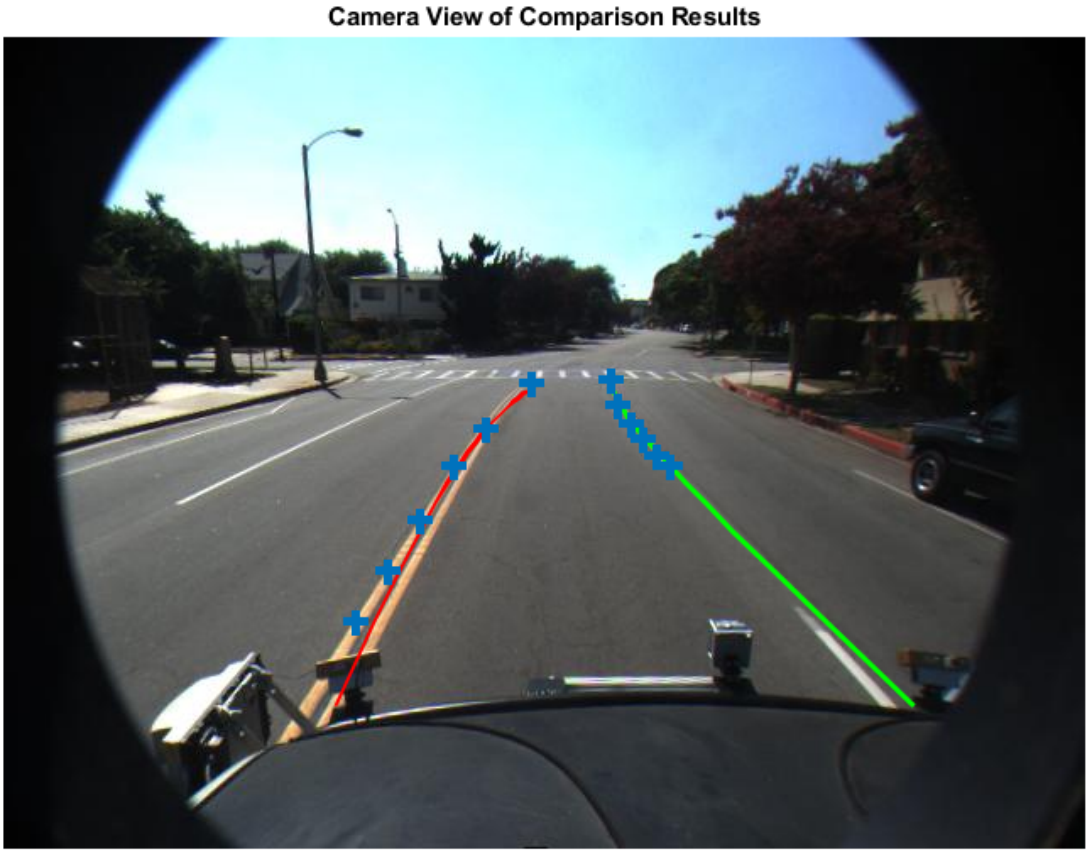
```
videoReader             = VideoReader(loaded.videoName);
videoReader.CurrentTime = seconds(loaded.groundTruthData.Time(frameInd));
frame                   = videoReader.readFrame();
```

Consider the boundary models as a solid line (irrespective of how the sensor classifies it) for visualization.

```
fpModels.BoundaryType      = 'Solid';
matchedModels.BoundaryType = 'Solid';
```

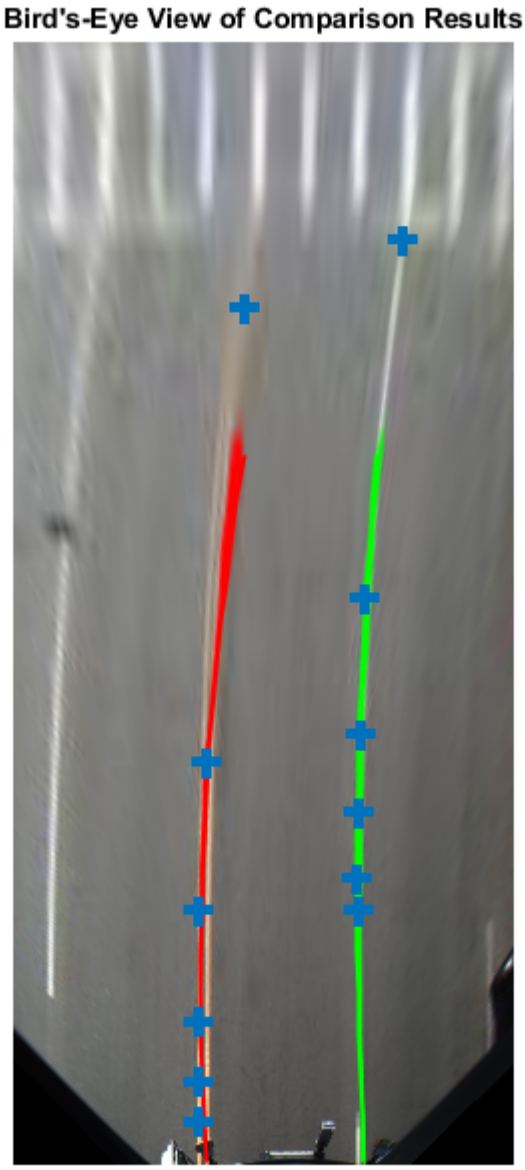
Insert the matched models, false positives and the ground truth points. This plot is useful in deducing that the presence of crosswalks poses a challenging scenario for the boundary modeling algorithm.

```
xVehicle = 3:20;
frame = insertLaneBoundary(frame, fpModels, sensor, xVehicle,'Color','Red');
frame = insertLaneBoundary(frame, matchedModels, sensor, xVehicle,'Color','Green');
figure
ha = axes;
imshow(frame,'Parent', ha);

% Combine the left and right boundary points
boundaryPoints = [frameImagePoints{1};frameImagePoints{2}];
hold on
plot(ha, boundaryPoints(:,1), boundaryPoints(:,2),'+','MarkerSize',10,'LineWidth',4);
title('Camera View of Comparison Results');
```



You can also visualize the results in the bird's-eye view of this frame.

```
birdsEyeImage = transformImage(monoSensor.BirdsEyeConfig,frame);

xVehicle = 3:20;
birdsEyeImage = insertLaneBoundary(birdsEyeImage, fpModels, monoSensor.BirdsEyeConfig, xVehicle,
birdsEyeImage = insertLaneBoundary(birdsEyeImage, matchedModels, monoSensor.BirdsEyeConfig, xVehi

% Combine the left and right boundary points
ptsInVehicle = [frameVehiclePoints{1};frameVehiclePoints{2}];
gtPointsInBEV  = vehicleToImage(monoSensor.BirdsEyeConfig, ptsInVehicle);

figure
imshow(birdsEyeImage);
hold on
plot(gtPointsInBEV(:,1), gtPointsInBEV(:,2),'+','MarkerSize', 10,'LineWidth',4);
title('Bird''s-Eye View of Comparison Results');
```



Tune Boundary Modeling Parameters

You can use the evaluation framework described previously to fine-tune parameters of the lane boundary detection algorithm. `helperMonoSensor` exposes three parameters that control the results of the lane-finding algorithm.

- `LaneSegmentationSensitivity` - Controls the sensitivity of `segmentLaneMarkerRidge` function. This function returns lane candidate points in the form of a binary lane feature mask. The sensitivity value can vary from 0 to 1, with a default of 0.25. Increasing this number results in more lane candidate points and potentially more false detections.

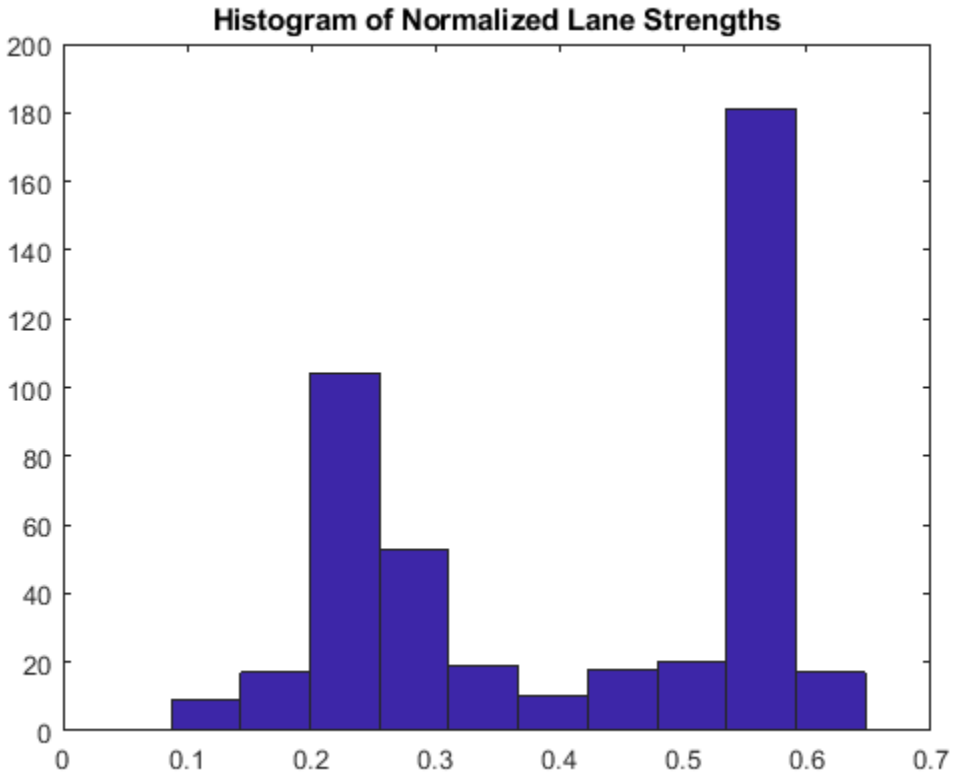
- `LaneXExtentThreshold` - Specifies the minimum extent (length) of a lane. It is expressed as a ratio of the detected lane length to the maximum lane length possible for the specified camera configuration. The default value is 0.4. Increase this number to reject shorter lane boundaries.
- `LaneStrengthThreshold` - Specifies the minimum normalized strength to accept a detected lane boundary.

`LaneXExtentThreshold` and `LaneStrengthThreshold` are derived from the `XExtent` and `Strength` properties of the `parabolicLaneBoundary` object. These properties are an example of how additional constraints can be placed on the boundary modeling algorithms to obtain acceptable results. The impact of varying `LaneStrengthThreshold` has additional nuances worth exploring. Typical lane boundaries are marked with either solid or dashed lines. When comparing to solid lines, dashed lines have a lower number of inlier points, leading to lower strength values. This makes it challenging to set a common strength threshold. To inspect the impact of this parameter, first generate all boundaries by setting `LaneStrengthThreshold` to 0. This setting ensures it has no impact on the output.

```
monoSensor.LaneStrengthThreshold = 0;
boundaries = detectBoundaries('caltech_cordova1.avi', monoSensor);
```

The `LaneStrengthThreshold` property of `helperMonoSensor` controls the normalized `Strength` parameter of each `parabolicLaneBoundary` model. The normalization factor, `MaxLaneStrength`, is the strength of a virtual lane that runs for the full extent of a bird's-eye image. This value is determined solely by the `birdsEyeView` configuration of `helperMonoSensor`. To assess the impact of `LaneStrengthThreshold`, first compute the distribution of the normalized lane strengths for all detected boundaries in the sample video. Note the presence of two clear peaks, one at a normalized strength of 0.3 and one at 0.7. These two peaks correspond to dashed and solid lane boundaries respectively. From this plot, you can empirically determine that to ensure dashed lane boundaries are detected, `LaneStrengthThreshold` should be below 0.3.

```
strengths = cellfun(@(b)[b.Strength], boundaries,'UniformOutput',false);
strengths = [strengths{:}];
normalizedStrengths = strengths/monoSensor.MaxLaneStrength;
figure;
hist(normalizedStrengths);
title('Histogram of Normalized Lane Strengths');
```



You can use the comparison framework to further assess the impact of the `LaneStrengthThreshold` parameters on the detection performance of the modeling algorithm. Note that the `threshold` value controlling the maximum physical distance between a model and a ground truth remains the same as before. This value is dictated by the accuracy requirements of an ADAS system and usually does not change.

```
threshold = .25;
[~, ~, ~, assignments] = ...
    evaluateLaneBoundaries(boundaries, gtVehicleBoundaryPoints, threshold);
```

Bin each boundary according to its normalized strength. The `assignments` information helps classify each boundary as either a true positive (matched) or a false positive. `LaneStrengthThreshold` is a "min" threshold, so a boundary classified as a true positive at a given value will continue to be a true positive for all lower threshold values.

```
nMatch = zeros(1,100); % Normalized lane strength is bucketed into 100 bins
nFP    = zeros(1,100); % ranging from 0.01 to 1.00.
for frameInd = 1:numel(boundaries)
    frameBoundaries  = boundaries{frameInd};
    frameAssignment  = assignments{frameInd};
    for bInd = 1:numel(frameBoundaries)
        normalizedStrength = frameBoundaries(bInd).Strength/monoSensor.MaxLaneStrength;
        strengthBucket     = floor(normalizedStrength*100);
        if frameAssignment(bInd)
            % This boundary was matched with a ground truth boundary,
            % record as a true positive for all values of strength above
            % its strength value.
```

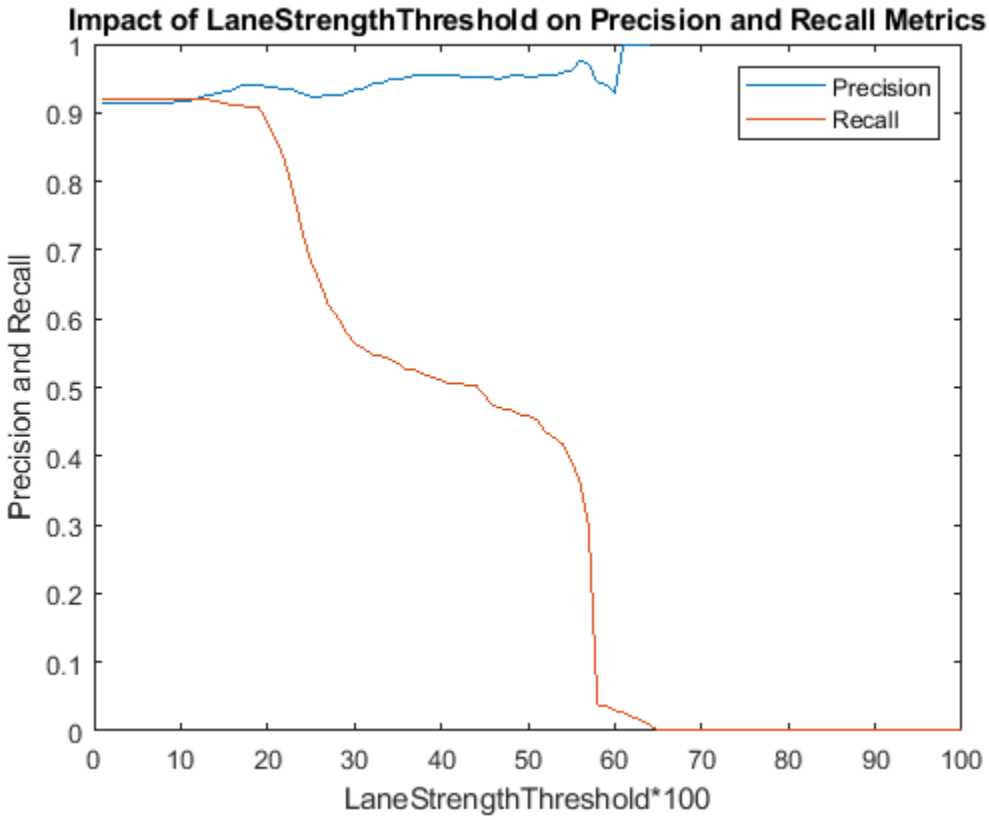
```
            nMatch(1:strengthBucket) = nMatch(1:strengthBucket)+1;
        else
            % This is a false positive
            nFP(1:strengthBucket) = nFP(1:strengthBucket)+1;
        end
    end
end
```

Use this information to compute the number of "missed" boundaries, that is, ground truth boundaries that the algorithm failed to detect at the specified `LaneStrengthThreshold` value. And with that information, compute the precision and recall metrics.

```
gtTotal = sum(cellfun(@(x)numel(x),gtVehicleBoundaryPoints));
nMiss   = gtTotal - nMatch;

precisionPlot = nMatch./(nMatch + nFP);
recallPlot    = nMatch./(nMatch + nMiss);
```

Plot the precision and recall metrics against various values of the lane strength threshold parameter. This plot is useful in determining an optimal value for the lane strength parameter. For this video clip, to maximize recall and precision metrics, `LaneStrengthThreshold` should be in the range 0.20 - 0.25.

```
figure;
plot(precisionPlot);
hold on;
plot(recallPlot);
xlabel('LaneStrengthThreshold*100');
ylabel('Precision and Recall');
legend('Precision','Recall');
title('Impact of LaneStrengthThreshold on Precision and Recall Metrics');
```



Supporting Function

Detect boundaries in a video.

`detectBoundaries` uses a preconfigured `helperMonoSensor` object to detect boundaries in a video.

```
function boundaries = detectBoundaries(videoName, monoSensor)
videoReader = VideoReader(videoName);
hwb         = waitbar(0,'Detecting and modeling boundaries in video...');
closeBar    = onCleanup(@()delete(hwb));
frameInd    = 0;
boundaries  = {};
while hasFrame(videoReader)
    frameInd  = frameInd+1;
    frame     = readFrame(videoReader);
    sensorOut = processFrame(monoSensor, frame);
    % Save the boundary models
    boundaries{end+1} =...
        [sensorOut.leftEgoBoundary, sensorOut.rightEgoBoundary];  %#ok<AGROW>
    waitbar(frameInd/(videoReader.Duration*videoReader.FrameRate), hwb);
```

```
end
end
```

See Also

Apps
Ground Truth Labeler

Functions
`evaluateLaneBoundaries` | `findParabolicLaneBoundaries` | `segmentLaneMarkerRidge`

More About

- “Evaluate and Visualize Lane Boundary Detections Against Ground Truth” on page 7-65

Evaluate and Visualize Lane Boundary Detections Against Ground Truth

This example shows how to evaluate the performance of lane boundary detection against known ground truth. In this example, you will characterize the performance of a lane boundary detection algorithm on a per-frame basis by computing a goodness-of-fit measure. This measure can be used to pinpoint, visualize, and understand failure modes in the underlying algorithm.

Overview

With increasing interest in vision-based solutions to automated driving problems, being able to evaluate and verify the accuracy of detection algorithms has become very important. Verifying accuracy is especially important in detection algorithms that have several parameters that can be tuned to achieve results that satisfy predefined quality requirements. This example walks through one such workflow, where lane boundaries can be measured for their level of accuracy. This workflow helps pinpoint failure modes in these algorithms on a per-frame basis, as well as characterize its overall performance. This workflow also helps you visually and quantitatively understand the performance of the algorithm. You can then use this understanding to tune the underlying algorithm to improve its performance.

Load Ground Truth Data

The dataset used in this example is a video file from a front-mounted camera on a vehicle driving through a street. Ground truth for the lane boundaries has been manually marked on the video with the Ground Truth Labeler app, using a Line ROI labeled "LaneBoundary." This video is 8 seconds, or 250 frames long. It has three intersection crossings, several vehicles (parked and moving), and lane boundaries (double line, single, and dashed). To create a ground truth lane boundary dataset for your own video, you can use the Ground Truth Labeler app.

```
% Load MAT file with ground truth data
loaded = load('caltech_cordova1_laneAndVehicleGroundTruth.mat');
```

The `loaded` structure contains three fields:

1. `groundTruthData`, a timetable with two columns: `LaneBoundaries` and `Vehicles`. `LaneBoundaries` contains ground truth points for the ego lane boundaries (left and right), represented as a cell array of XY points forming a poly line. `Vehicles` contains ground truth bounding boxes for vehicles in the camera view, represented as M-by-4 arrays of [x,y,width,height].
2. `sensor`, a `monoCamera` object with properties about the calibrated camera mounted on the vehicle. This object lets you estimate the real-world distances between the vehicle and the objects in front of it.
3. `videoName`, a character array containing the file name of the video where the frames are stored.

From the data in this structure, open the video file by using `VideoReader` to loop through the frames. The `VideoReader` object uses a `helperMonoSensor` object to detect lanes and objects in the video frame, using the camera setup stored in `sensor`. A `timetable` variable stored in `gtdata` holds the ground truth data. This variable contains the per-frame data that is used for analysis later on.

```
% Create a VideoReader object to read frames of the video.
videoName  = loaded.videoName;
fileReader = VideoReader(videoName);
```

```
% The ground truth data is organized in a timetable.
gtdata = loaded.groundTruthData;

% Display the first few rows of the ground truth data.
head(gtdata)
```

```
ans =

  8x2 timetable

        Time          Vehicles      LaneBoundaries
    ______________    ______________    _______________

    0 sec             {6x4 double}      {2x1 cell}
    0.033333 sec      {6x4 double}      {2x1 cell}
    0.066667 sec      {6x4 double}      {2x1 cell}
    0.1 sec           {6x4 double}      {2x1 cell}
    0.13333 sec       {6x4 double}      {2x1 cell}
    0.16667 sec       {6x4 double}      {2x1 cell}
    0.2 sec           {6x4 double}      {2x1 cell}
    0.23333 sec       {5x4 double}      {2x1 cell}
```

The `gtdata` timetable has the columns `Vehicles` and `LaneBoundaries`. At each timestamp, the `Vehicles` column holds an M-by-4 array of vehicle bounding boxes and the `LaneBoundaries` column holds a two-element cell array of left and right lane boundary points.

First, visualize the loaded ground truth data for an image frame.

```
% Read the first frame of the video.
frame = readFrame(fileReader);

% Extract all lane points in the first frame.
lanePoints = gtdata.LaneBoundaries{1};

% Extract vehicle bounding boxes in the first frame.
vehicleBBox = gtdata.Vehicles{1};

% Superimpose the right lane points and vehicle bounding boxes.
frame = insertMarker(frame, lanePoints{2}, 'X');
frame = insertObjectAnnotation(frame, 'rectangle', vehicleBBox, 'Vehicle');

% Display ground truth data on the first frame.
figure
imshow(frame)
```

Run Lane Boundary Detection Algorithm

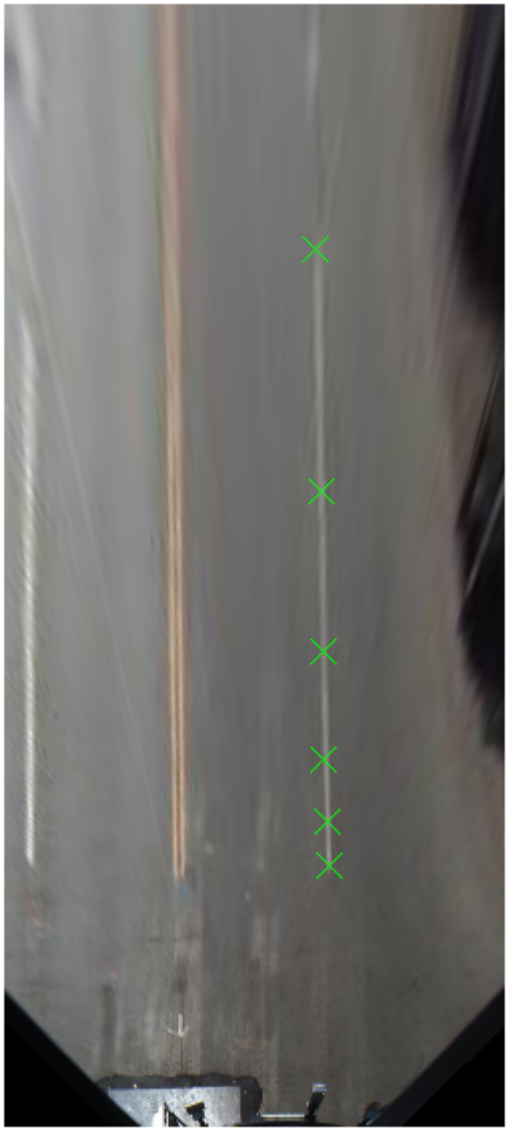
Using the video frames and the `monoCamera` parameters, you can automatically estimate locations of lane boundaries. For illustration, the `processFrame` method of the `helperMonoSensor` class is used here to detect lane boundaries (as `parabolicLaneBoundary` objects) and vehicles (as [x, y, width, height] bounding box matrices). For the purpose of this example, this is the lane boundary detection "algorithm under test." You can use the same pattern for evaluating a custom lane boundary detection algorithm, where `processFrame` is replaced with the custom detection function. The ground truth points in the vehicle coordinates are also stored in the `LanesInVehicleCoord` column of the `gtdata` timetable. That way, they can be visualized in a Bird's-Eye View display later on. First, configure the `helperMonoSensor` object with the `sensor`. The `helperMonoSensor` class assembles all the necessary steps required to run the lane boundary detection algorithm.

```
% Set up monoSensorHelper to process video.
monoCameraSensor = loaded.sensor;
monoSensorHelper = helperMonoSensor(monoCameraSensor);

% Create new timetable with same Time vector for measurements.
measurements = timetable(gtdata.Time);

% Set up timetable columns for holding lane boundary and vehicle data.
numFrames = floor(fileReader.FrameRate*fileReader.Duration);
```

```
measurements.LaneBoundaries    = cell(numFrames, 2);
measurements.VehicleDetections = cell(numFrames, 1);
gtdata.LanesInVehicleCoord     = cell(numFrames, 2);

% Rewind the video to t = 0, and create a frame index to hold current
% frame.
fileReader.CurrentTime = 0;
frameIndex  = 0;

% Loop through the videoFile until there are no new frames.
while hasFrame(fileReader)
    frameIndex = frameIndex+1;
    frame      = readFrame(fileReader);

    % Use the processFrame method to compute detections.
    % This method can be replaced with a custom lane detection method.
    detections = processFrame(monoSensorHelper, frame);

    % Store the estimated lane boundaries and vehicle detections.
    measurements.LaneBoundaries{frameIndex} = [detections.leftEgoBoundary ...
                                               detections.rightEgoBoundary];
    measurements.VehicleDetections{frameIndex} = detections.vehicleBoxes;

    % To facilitate comparison, convert the ground truth lane points to the
    % vehicle coordinate system.
    gtPointsThisFrame = gtdata.LaneBoundaries{frameIndex};
    vehiclePoints = cell(1, numel(gtPointsThisFrame));
    for ii = 1:numel(gtPointsThisFrame)
        vehiclePoints{ii} = imageToVehicle(monoCameraSensor, gtPointsThisFrame{ii});
    end

    % Store ground truth points expressed in vehicle coordinates.
    gtdata.LanesInVehicleCoord{frameIndex} = vehiclePoints;
end
```

Now that you have processed the video with a lane detection algorithm, verify that the ground truth points are correctly transformed into the vehicle coordinate system. The first entry in the `LanesInVehicleCoord` column of the `gtdata` timetable contains the vehicle coordinates for the first frame. Plot these ground truth points on the first frame in the Bird's-Eye View.

```
% Rewind video to t = 0.
fileReader.CurrentTime = 0;

% Read the first frame of the video.
frame = readFrame(fileReader);
birdsEyeImage = transformImage(monoSensorHelper.BirdsEyeConfig, frame);

% Extract right lane points for the first frame in Bird's-Eye View.
firstFrameVehiclePoints = gtdata.LanesInVehicleCoord{1};
pointsInBEV = vehicleToImage(monoSensorHelper.BirdsEyeConfig, firstFrameVehiclePoints{2});

% Superimpose points on the frame.
birdsEyeImage = insertMarker(birdsEyeImage, pointsInBEV, 'X', 'Size', 6);

% Display transformed points in Bird's-Eye View.
figure
imshow(birdsEyeImage)
```



Measure Detection Errors

Computing the errors in lane boundary detection is an essential step in verifying the performance of several downstream subsystems. Such subsystems include lane departure warning systems that depend on the accuracy of the lane detection subsystem.

You can estimate this accuracy by measuring the goodness of fit. With the ground truth points and the estimates computed, you can now compare and visualize them to find out how well the detection algorithms perform.

The goodness of fit can be measured either at the per-frame level or for the entire video. The per-frame statistics provide detailed information about specific scenarios, such as the behavior at road bends where the detection algorithm performance may vary. The global statistics provide a big picture estimate of number of lanes that missed detection.

Use the `evaluateLaneBoundaries` function to return global detection statistics and an `assignments` array. This array matches the estimated lane boundary objects with a corresponding ground truth points.

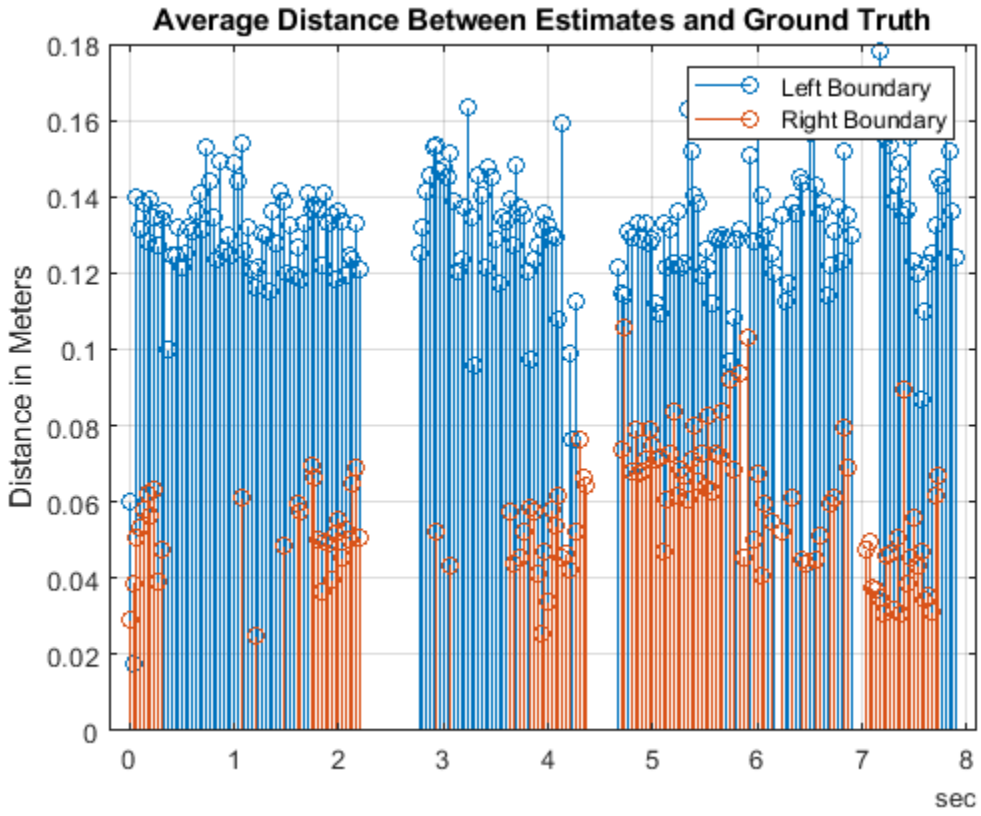
The threshold parameter in the `evaluateLaneBoundaries` function represents the maximum lateral distance in vehicle coordinates to qualify as a match with the estimated parabolic lane boundaries.

```
threshold = 0.25; % in meters

[numMatches, numMisses, numFalsePositives, assignments] = ...
    evaluateLaneBoundaries(measurements.LaneBoundaries, ...
                           gtdata.LanesInVehicleCoord, ...
                           threshold);

disp(['Number of matches: ', num2str(numMatches)]);
disp(['Number of misses: ', num2str(numMisses)]);
disp(['Number of false positives: ', num2str(numFalsePositives)]);
```

```
Number of matches: 321
Number of misses: 124
Number of false positives: 25
```

Using the `assignments` array, you can compute useful per-lane metrics, such as the average lateral distance between the estimates and the ground truth points. Such metrics indicate how well the algorithm is performing. To compute the average distance metric, use the helper function `helperComputeLaneStatistics`, which is defined at the end of this example.

```
averageDistance = helperComputeLaneStatistics(measurements.LaneBoundaries, ...
                                               gtdata.LanesInVehicleCoord, ...
                                               assignments, @mean);

% Plot average distance between estimates and ground truth.
figure
stem(gtdata.Time, averageDistance)
title('Average Distance Between Estimates and Ground Truth')
grid on
ylabel('Distance in Meters')
legend('Left Boundary','Right Boundary')
```



Visualize and Review Differences Between Ground Truth and Your Algorithm

You now have a quantitative understanding of the accuracy of the lane detection algorithm. However, it is not possible to completely understand the failures solely based on the plot in the previous section. Viewing the video and visualizing the errors on a per-frame basis is therefore crucial in identifying specific failure modes which can be improved by refining the algorithm.

You can use the Ground Truth Labeler app as a visualization tool to view the video containing the ground truth data and the estimated lane boundaries. The `driving.connector.Connector` class provides an interface to attach custom visualization tools to the Ground Truth Labeler.

Use the `parabolicLaneBoundary` array and the ground truth data to compute vehicle coordinate locations of the estimated points. The `parabolicLaneBoundary` array defines a line, and the ground truth data has discrete points marked on the road. The `helperGetCorrespondingPoints` function estimates points on the estimated lines that correspond to the same Y-axis distance from the vehicle. This helper function is defined at the end of the example.

The ground truth points and the estimated points are now included in a new `timetable` to be visualized in the Ground Truth Labeler app. The created `groundTruth` object is then stored as a MAT file.

```
% Compute the estimated point locations using the monoCamera.
[estVehiclePoints, estImagePoints] = helperGetCorrespondingPoints(monoCameraSensor, ...
                                     measurements.LaneBoundaries, ...
                                     gtdata.LanesInVehicleCoord, ...
                                     assignments);
```

```
% Add estimated lanes to the measurements timetable.
measurements.EstimatedLanes      = estImagePoints;
measurements.LanesInVehicleCoord = estVehiclePoints;

% Create a new timetable with all the variables needed for visualization.
names = {'LanePoints'; 'DetectedLanePoints'};
types = labelType({'Line'; 'Line'});
labelDefs = table(names, types, 'VariableNames', {'Name','Type'});

visualizeInFrame = timetable(gtdata.Time, ...
                             gtdata.LaneBoundaries, ...
                             measurements.EstimatedLanes, ...
                             'VariableNames', names);

% Create groundTruth object.
dataSource = groundTruthDataSource(videoName);
dataToVisualize = groundTruth(dataSource, labelDefs, visualizeInFrame);

% Save all the results of the previous section in distanceData.mat in a
% temporary folder.
dataToLoad = [tempdir 'distanceData.mat'];
save(dataToLoad, 'monoSensorHelper', 'videoName', 'measurements', 'gtdata', 'averageDistance');
```

The `helperCustomUI` class creates the plot and Bird's-Eye Views using data loaded from a MAT file, like the one you just created. The Connector interface of the Ground Truth Labeler app interacts with the `helperCustomUI` class through the `helperUIConnector` class to synchronize the video with the average distance plot and the Bird's-Eye View. This synchronization enables you to analyze per-frame results both analytically and visually.

Follow these steps to visualize the results as shown in the images that follow:

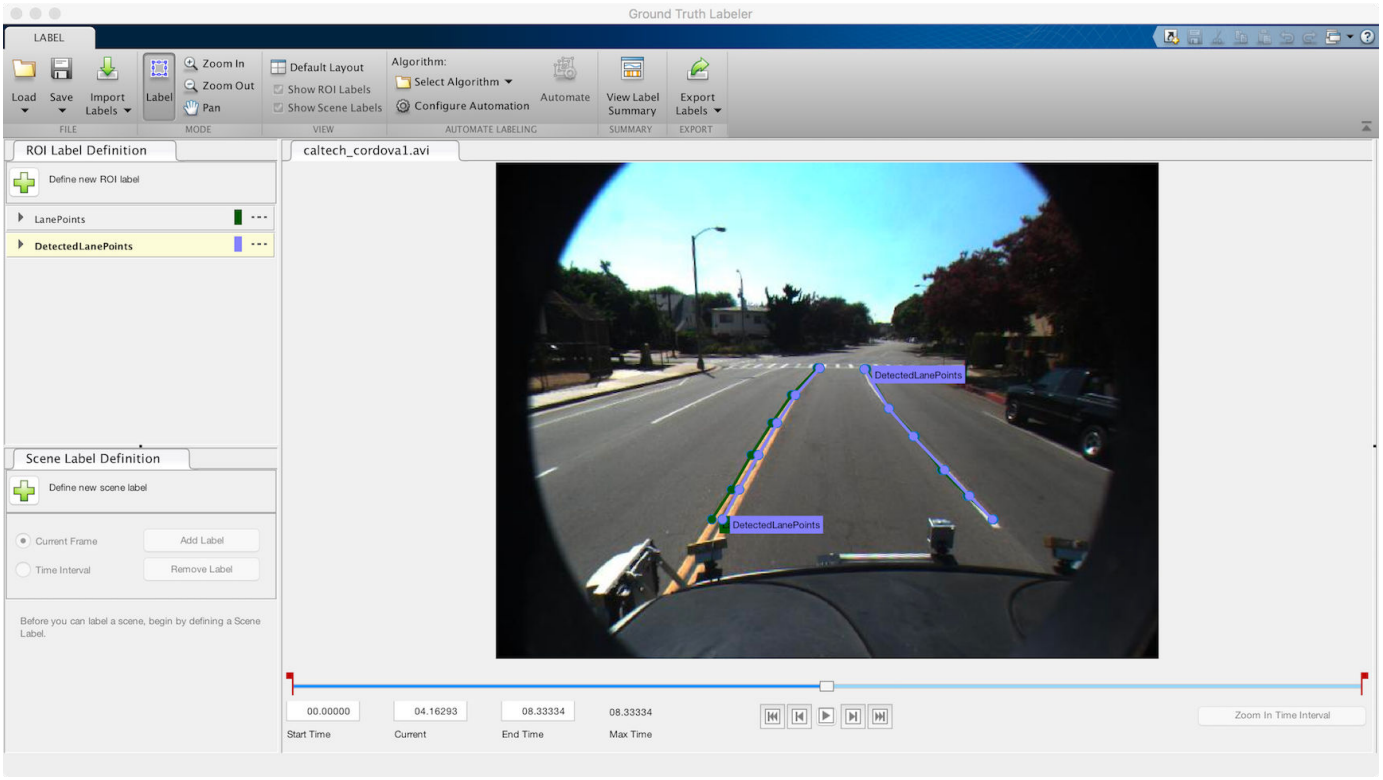
- Go to the temporary directory where `distanceData.mat` is saved and open the Ground Truth Labeler app. Then start the Ground Truth Labeler app, with the connector handle specified as `helperUIConnector` using the following commands:

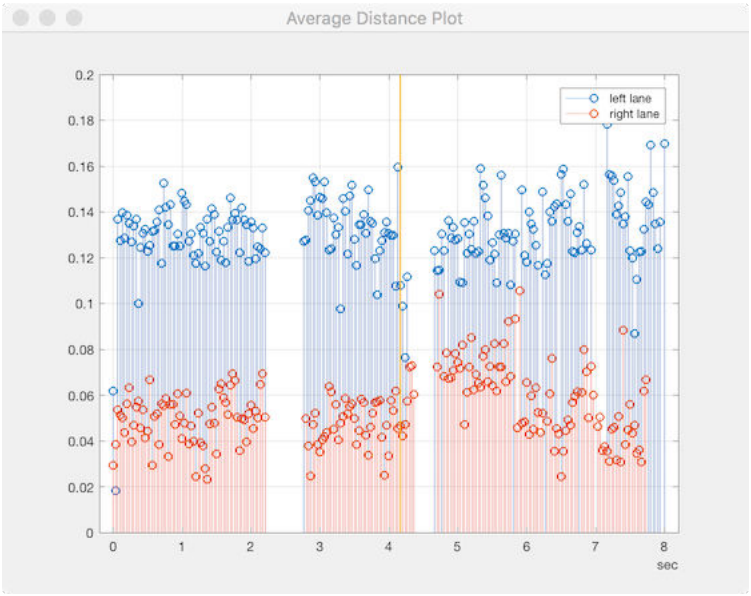
```
>> origdir = pwd;
>> cd(tempdir)
>> groundTruthLabeler(dataSource, 'ConnectorTargetHandle', @helperUIConnector);
```

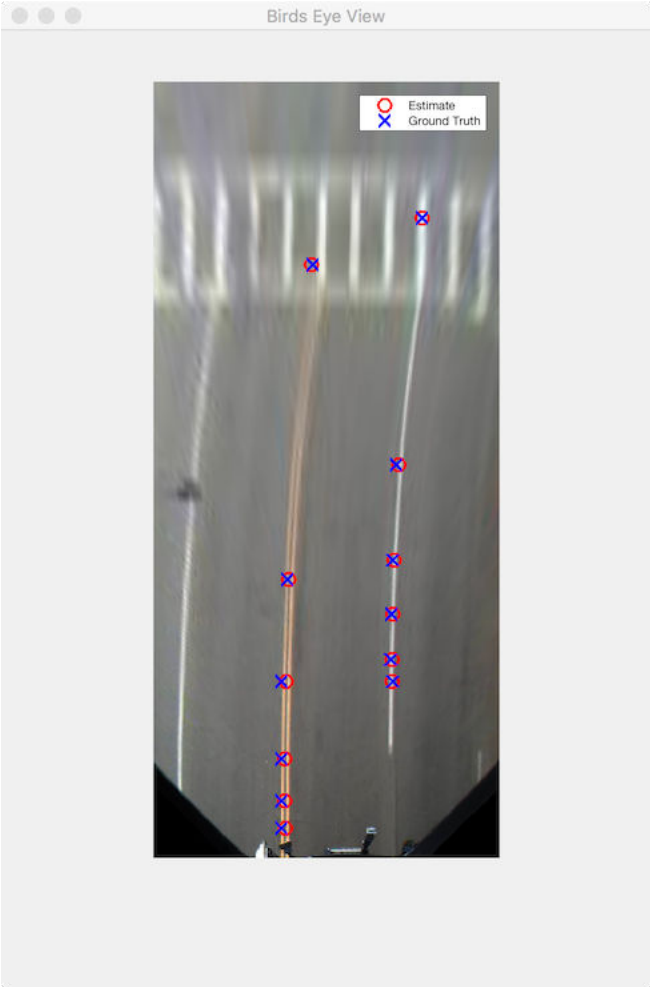
- Import labels: Visualize the ground truth lane markers and the estimated lanes in the image coordinates. From the app toolstrip, click **Import Labels**. Then select the **From Workspace** option and load the `dataToVisualize` ground truth into the app. The main app window now contains annotations for lane markers.

You can now navigate through the video and examine the errors. To return back to the original directory, you can type:

```
>> cd(origdir)
```


Ground Truth Labeler
LABEL
Load
Save
Import Labels
Label
Zoom In
Zoom Out
Pan
Default Layout
Show ROI Labels
Show Scene Labels
Algorithm:
Select Algorithm
Configure Automation
Automate
View Label Summary
Export Labels
FILE
MODE
VIEW
AUTOMATE LABELING
SUMMARY
EXPORT
ROI Label Definition
Define new ROI label
LanePoints
DetectedLanePoints
Scene Label Definition
Define new scene label
Current Frame
Time Interval
Add Label
Remove Label
Before you can label a scene, begin by defining a Scene Label.
caltech_cordova1.avi
DetectedLanePoints
DetectedLanePoints
00.00000
Start Time
04.16293
Current
08.33334
End Time
08.33334
Max Time
Zoom In Time Interval


Average Distance Plot
left lane
right lane
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
1
2
3
4
5
6
7
8
sec



From this visualization, you can make several inferences about the algorithm and the quality of the ground truth data.

- The left lane accuracy is consistently worse than the right lane accuracy. Upon closer observation in the Bird's-Eye View display, the ground truth data is marked as the outer boundary of the double line, whereas the estimated lane boundary lays generally at the center of the double line marker. This indicates that the left lane estimation is likely more accurate than the numbers portray, and that a clearly defined ground truth dataset is crucial for such observations.
- The detection gaps around 2.3 seconds and 4 seconds correspond to intersections on the road that are preceded by crosswalks. This indicates that the algorithm does not perform well in the presence of crosswalks.
- Around 6.8 seconds, as the vehicle approaches a third intersection, the ego lane diverges into a left-only lane and a straight lane. Here too, the algorithm fails to capture the left lane accurately, and the ground truth data also does not contain any information for five frames.

Conclusion

This example showed how to measure the accuracy of a lane boundary detection algorithm and visualize it using the Ground Truth Labeler app. You can extend this concept to other custom

algorithms to simplify these workflows and extend the functionality of the app for custom measurements.

Supporting Functions

helperComputeLaneStatistics

This helper function computes statistics for lane boundary detections as compared to ground truth points. It takes in a function handle that can be used to generalize the statistic that needs to be computed, including @mean and @median.

```
function stat = helperComputeLaneStatistics(estModels, gtPoints, assignments, fcnHandle)

    numFrames = length(estModels);
    % Make left and right estimates NaN by default to represent lack of
    % data.
    stat = NaN*ones(numFrames, 2);

    for frameInd = 1:numFrames
        % Make left and right estimates NaN by default.
        stat(frameInd, :) = NaN*ones(2, 1);

        for idx = 1:length(estModels{frameInd})
            % Ignore false positive assignments.
            if assignments{frameInd}(idx) == 0
                continue;
            end

            % The kth boundary in estModelInFrame is matched to kth
            % element indexed by assignments in gtPointsInFrame.
            thisModel = estModels{frameInd}(idx);
            thisGT = gtPoints{frameInd}{assignments{frameInd}(idx)};
            thisGTModel = driving.internal.piecewiseLinearBoundary(thisGT);
            if mean(thisGTModel.Points(:,2)) > 0
                % left lane
                xPoints = thisGTModel.Points(:,1);
                yDist = zeros(size(xPoints));
                for index = 1:numel(xPoints)
                    gtYPoints   = thisGTModel.computeBoundaryModel(xPoints(index));
                    testYPoints = thisModel.computeBoundaryModel(xPoints(index));
                    yDist(index) = abs(testYPoints-gtYPoints);
                end
                stat(frameInd, 1) = fcnHandle(yDist);
            else % right lane
                xPoints = thisGTModel.Points(:,1);
                yDist = zeros(size(xPoints));
                for index = 1:numel(xPoints)
                    gtYPoints   = thisGTModel.computeBoundaryModel(xPoints(index));
                    testYPoints = thisModel.computeBoundaryModel(xPoints(index));
                    yDist(index) = abs(testYPoints-gtYPoints);
                end
                stat(frameInd, 2) = fcnHandle(yDist);
            end
        end
    end
end
```

helperGetCorrespondingPoints

This helper function creates vehicle and image coordinate points at X-axis locations that match the ground truth points.

```
function [vehiclePoints, imagePoints] = helperGetCorrespondingPoints(monoCameraSensor, estModels,

    numFrames = length(estModels);
    imagePoints = cell(numFrames, 1);
    vehiclePoints = cell(numFrames, 1);

    for frameInd = 1:numFrames
        if isempty(assignments{frameInd})
            imagePointsInFrame = [];
            vehiclePointsInFrame = [];
        else
            estModelInFrame = estModels{frameInd};
            gtPointsInFrame = gtPoints{frameInd};
            imagePointsInFrame = cell(length(estModelInFrame), 1);
            vehiclePointsInFrame = cell(length(estModelInFrame), 1);
            for idx = 1:length(estModelInFrame)

                % Ignore false positive assignments.
                if assignments{frameInd}(idx) == 0
                    imagePointsInFrame{idx} = [NaN NaN];
                    continue;
                end

                % The kth boundary in estModelInFrame is matched to kth
                % element indexed by assignments in gtPointsInFrame.
                thisModel = estModelInFrame(idx);
                thisGT = gtPointsInFrame{assignments{frameInd}(idx)};
                xPoints = thisGT(:, 1);
                yPoints = thisModel.computeBoundaryModel(xPoints);

                vehiclePointsInFrame{idx} = [xPoints, yPoints];
                imagePointsInFrame{idx} = vehicleToImage(monoCameraSensor, [xPoints yPoints]);
            end
        end
        vehiclePoints{frameInd} = vehiclePointsInFrame;
        imagePoints{frameInd}   = imagePointsInFrame;
        % Make imagePoints [] instead of {} to comply with groundTruth object.
        if isempty(imagePoints{frameInd})
            imagePoints{frameInd} = [];
        end
        if isempty(vehiclePoints{frameInd})
            vehiclePoints{frameInd} = [];
        end
    end
end
```

See Also

Apps
Ground Truth Labeler

Functions
`evaluateLaneBoundaries` | `findParabolicLaneBoundaries`

Objects
`birdsEyeView` | `driving.connector.Connector` | `monoCamera`

More About

- “Evaluate Lane Boundary Detections Against Ground Truth Data” on page 7-53

Visual Perception Using Monocular Camera

This example shows how to construct a monocular camera sensor simulation capable of lane boundary and vehicle detections. The sensor will report these detections in vehicle coordinate system. In this example, you will learn about the coordinate system used by Automated Driving Toolbox™, and computer vision techniques involved in the design of a sample monocular camera sensor.

Overview

Vehicles that contain ADAS features or are designed to be fully autonomous rely on multiple sensors. These sensors can include sonar, radar, lidar and cameras. This example illustrates some of the concepts involved in the design of a monocular camera system. Such a sensor can accomplish many tasks, including:

- Lane boundary detection
- Detection of vehicles, people, and other objects
- Distance estimation from the ego vehicle to obstacles

Subsequently, the readings returned by a monocular camera sensor can be used to issue lane departure warnings, collision warnings, or to design a lane keep assist control system. In conjunction with other sensors, it can also be used to implement an emergency braking system and other safety-critical features.

The example implements a subset of features found on a fully developed monocular camera system. It detects lane boundaries and backs of vehicles, and reports their locations in the vehicle coordinate system.

Define Camera Configuration

Knowing the camera's intrinsic and extrinsic calibration parameters is critical to accurate conversion between pixel and vehicle coordinates.

Start by defining the camera's intrinsic parameters. The parameters below were determined earlier using a camera calibration procedure that used a checkerboard calibration pattern. You can use the Camera Calibrator (Computer Vision Toolbox) app to obtain them for your camera.

```
focalLength    = [309.4362, 344.2161]; % [fx, fy] in pixel units
principalPoint = [318.9034, 257.5352]; % [cx, cy] optical center in pixel coordinates
imageSize      = [480, 640];           % [nrows, mcols]
```

Note that the lens distortion coefficients were ignored, because there is little distortion in the data. The parameters are stored in a `cameraIntrinsics` (Computer Vision Toolbox) object.

```
camIntrinsics = cameraIntrinsics(focalLength, principalPoint, imageSize);
```

Next, define the camera orientation with respect to the vehicle's chassis. You will use this information to establish camera extrinsics that define the position of the 3-D camera coordinate system with respect to the vehicle coordinate system.

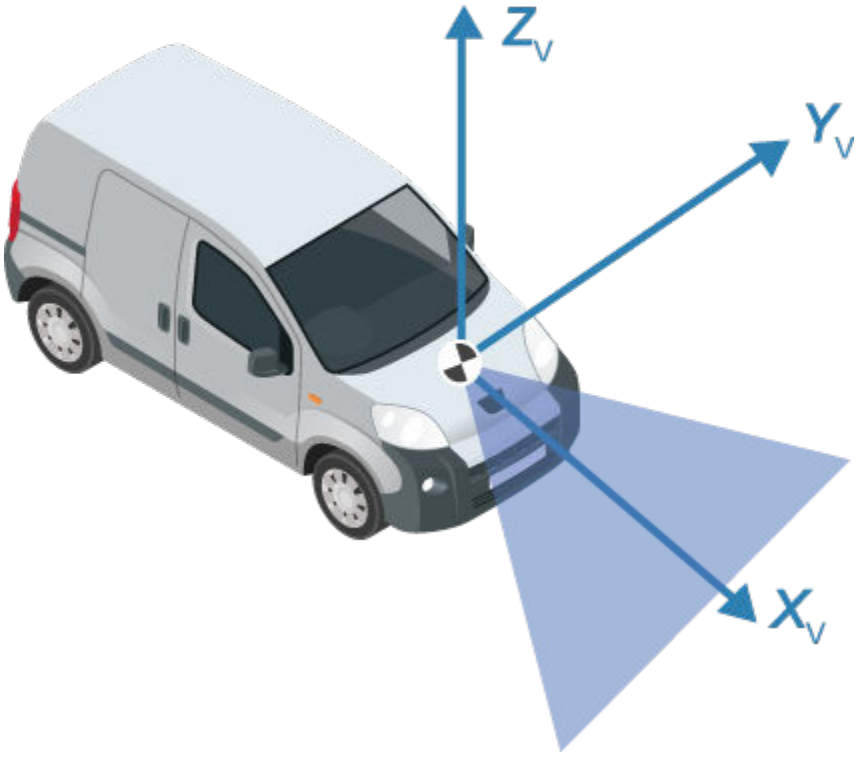
```
height = 2.1798;    % mounting height in meters from the ground
pitch  = 14;        % pitch of the camera in degrees
```

The above quantities can be derived from the rotation and translation matrices returned by the `extrinsics` (Computer Vision Toolbox) function. Pitch specifies the tilt of the camera from the

horizontal position. For the camera used in this example, the roll and yaw of the sensor are both zero. The entire configuration defining the intrinsics and extrinsics is stored in the `monoCamera` object.

```
sensor = monoCamera(camIntrinsics, height, 'Pitch', pitch);
```

Note that the `monoCamera` object sets up a very specific vehicle coordinate system, where the *X*-axis points forward from the vehicle, the *Y*-axis points to the left of the vehicle, and the *Z*-axis points up from the ground.



By default, the origin of the coordinate system is on the ground, directly below the camera center defined by the camera's focal point. The origin can be moved by using the `SensorLocation` property of the `monoCamera` object. Additionally, `monoCamera` provides `imageToVehicle` and `vehicleToImage` methods for converting between image and vehicle coordinate systems.

Note: The conversion between the coordinate systems assumes a flat road. It is based on establishing a homography matrix that maps locations on the imaging plane to locations on the road surface. Nonflat roads introduce errors in distance computations, especially at locations that are far from the vehicle.

Load a Frame of Video

Before processing the entire video, process a single video frame to illustrate the concepts involved in the design of a monocular camera sensor.

Start by creating a `VideoReader` object that opens a video file. To be memory efficient, `VideoReader` loads one video frame at a time.

```
videoName = 'caltech_cordova1.avi';
videoReader = VideoReader(videoName);
```

Read an interesting frame that contains lane markers and a vehicle.

```
timeStamp = 0.06667;                    % time from the beginning of the video
videoReader.CurrentTime = timeStamp;    % point to the chosen frame

frame = readFrame(videoReader); % read frame at timeStamp seconds
imshow(frame) % display frame
```

Note: This example ignores lens distortion. If you were concerned about errors in distance measurements introduced by the lens distortion, at this point you would use the `undistortImage` (Computer Vision Toolbox) function to remove the lens distortion.

Create Bird's-Eye-View Image

There are many ways to segment and detect lane markers. One approach involves the use of a bird's-eye-view image transform. Although it incurs computational cost, this transform offers one major advantage. The lane markers in the bird's-eye view are of uniform thickness, thus simplifying the segmentation process. The lane markers belonging to the same lane also become parallel, thus making further analysis easier.

Given the camera setup, the `birdsEyeView` object transforms the original image to the bird's-eye view. This object lets you specify the area that you want to transform using vehicle coordinates. Note

that the vehicle coordinate units were established by the `monoCamera` object, when the camera mounting height was specified in meters. For example, if the height was specified in millimeters, the rest of the simulation would use millimeters.

```
% Using vehicle coordinates, define area to transform
distAheadOfSensor = 30; % in meters, as previously specified in monoCamera height input
spaceToOneSide    = 6;  % all other distance quantities are also in meters
bottomOffset      = 3;

outView   = [bottomOffset, distAheadOfSensor, -spaceToOneSide, spaceToOneSide]; % [xmin, xmax, yn
imageSize = [NaN, 250]; % output image width in pixels; height is chosen automatically to preserv

birdsEyeConfig = birdsEyeView(sensor, outView, imageSize);
```

Generate bird's-eye-view image.

```
birdsEyeImage = transformImage(birdsEyeConfig, frame);
figure
imshow(birdsEyeImage)
```

The areas further away from the sensor are more blurry, due to having fewer pixels and thus requiring greater amount of interpolation.

Note that you can complete the latter processing steps without use of the bird's-eye view, as long as you can locate lane boundary candidate pixels in vehicle coordinates.

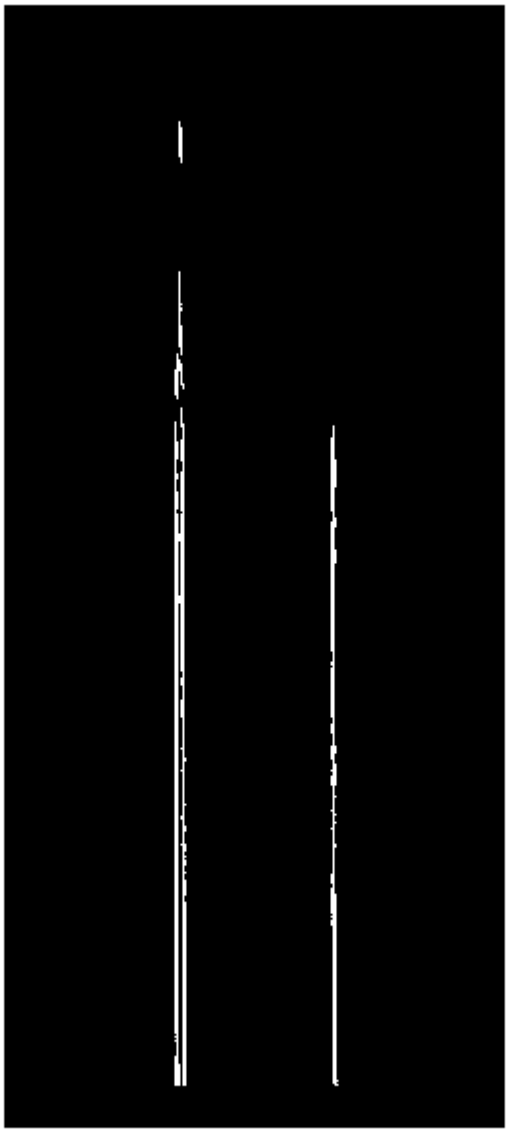
Find Lane Markers in Vehicle Coordinates

Having the bird's-eye-view image, you can now use the `segmentLaneMarkerRidge` function to separate lane marker candidate pixels from the road surface. This technique was chosen for its simplicity and relative effectiveness. Alternative segmentation techniques exist including semantic

segmentation (deep learning) and steerable filters. You can substitute these techniques below to obtain a binary mask needed for the next stage.

Most input parameters to the functions below are specified in world units, for example, the lane marker width fed into `segmentLaneMarkerRidge`. The use of world units allows you to easily try new sensors, even when the input image size changes. This is very important to making the design more robust and flexible with respect to changing camera hardware and handling varying standards across many countries.

```
% Convert to grayscale
birdsEyeImage = rgb2gray(birdsEyeImage);

% Lane marker segmentation ROI in world units
vehicleROI = outView - [-1, 2, -3, 3]; % look 3 meters to left and right, and 4 meters ahead of t
approxLaneMarkerWidthVehicle = 0.25; % 25 centimeters

% Detect lane features
laneSensitivity = 0.25;
birdsEyeViewBW = segmentLaneMarkerRidge(birdsEyeImage, birdsEyeConfig, approxLaneMarkerWidthVehic
    'ROI', vehicleROI, 'Sensitivity', laneSensitivity);

figure
imshow(birdsEyeViewBW)
```



Locating individual lane markers takes place in vehicle coordinates that are anchored to the camera sensor. This example uses a parabolic lane boundary model, ax^2 + bx + c, to represent the lane markers. Other representations, such as a third-degree polynomial or splines, are possible. Conversion to vehicle coordinates is necessary, otherwise lane marker curvature cannot be properly represented by a parabola while it is affected by a perspective distortion.

The lane model holds for lane markers along a vehicle's path. Lane markers going across the path or road signs painted on the asphalt are rejected.

```
% Obtain lane candidate points in vehicle coordinates
[imageX, imageY] = find(birdsEyeViewBW);
xyBoundaryPoints = imageToVehicle(birdsEyeConfig, [imageY, imageX]);
```

Since the segmented points contain many outliers that are not part of the actual lane markers, use the robust curve fitting algorithm based on random sample consensus (RANSAC).

Return the boundaries and their parabola parameters (a, b, c) in an array of `parabolicLaneBoundary` objects, `boundaries`.

```
maxLanes      = 2; % look for maximum of two lane markers
boundaryWidth = 3*approxLaneMarkerWidthVehicle; % expand boundary width

[boundaries, boundaryPoints] = findParabolicLaneBoundaries(xyBoundaryPoints,boundaryWidth, ...
    'MaxNumBoundaries', maxLanes, 'validateBoundaryFcn', @validateBoundaryFcn);
```

Notice that the `findParabolicLaneBoundaries` takes a function handle, `validateBoundaryFcn`. This example function is listed at the end of this example. Using this additional input lets you reject some curves based on the values of the a, b, c parameters. It can also be used to take advantage of temporal information over a series of frames by constraining future a, b, c values based on previous video frames.

Determine Boundaries of the Ego Lane

Some of the curves found in the previous step might still be invalid. For example, when a curve is fit into crosswalk markers. Use additional heuristics to reject many such curves.

```
% Establish criteria for rejecting boundaries based on their length
maxPossibleXLength = diff(vehicleROI(1:2));
minXLength         = maxPossibleXLength * 0.60; % establish a threshold

% Reject short boundaries
isOfMinLength = arrayfun(@(b)diff(b.XExtent) > minXLength, boundaries);
boundaries    = boundaries(isOfMinLength);
```

Remove additional boundaries based on the strength metric computed by the `findParabolicLaneBoundaries` function. Set a lane strength threshold based on ROI and image size.

```
% To compute the maximum strength, assume all image pixels within the ROI
% are lane candidate points
birdsImageROI = vehicleToImageROI(birdsEyeConfig, vehicleROI);
[laneImageX,laneImageY] = meshgrid(birdsImageROI(1):birdsImageROI(2),birdsImageROI(3):birdsImageR

% Convert the image points to vehicle points
vehiclePoints = imageToVehicle(birdsEyeConfig,[laneImageX(:),laneImageY(:)]);

% Find the maximum number of unique x-axis locations possible for any lane
% boundary
maxPointsInOneLane = numel(unique(vehiclePoints(:,1)));

% Set the maximum length of a lane boundary to the ROI length
maxLaneLength = diff(vehicleROI(1:2));

% Compute the maximum possible lane strength for this image size/ROI size
% specification
maxStrength   = maxPointsInOneLane/maxLaneLength;
```

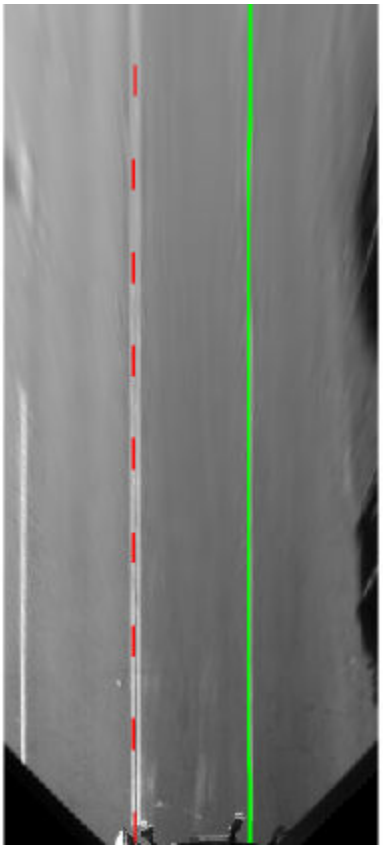
```
% Reject weak boundaries
isStrong      = [boundaries.Strength] > 0.4*maxStrength;
boundaries    = boundaries(isStrong);
```

The heuristics to classify lane marker type as solid/dashed are included in a helper function listed at the bottom of this example. Knowing the lane marker type is critical for steering the vehicle automatically. For example, crossing a solid marker is prohibited.

```
boundaries = classifyLaneTypes(boundaries, boundaryPoints);

% Locate two ego lanes if they are present
xOffset    = 0;   %  0 meters from the sensor
distanceToBoundaries  = boundaries.computeBoundaryModel(xOffset);

% Find candidate ego boundaries
leftEgoBoundaryIndex  = [];
rightEgoBoundaryIndex = [];
minLDistance = min(distanceToBoundaries(distanceToBoundaries>0));
minRDistance = max(distanceToBoundaries(distanceToBoundaries<=0));
if ~isempty(minLDistance)
    leftEgoBoundaryIndex  = distanceToBoundaries == minLDistance;
end
if ~isempty(minRDistance)
    rightEgoBoundaryIndex = distanceToBoundaries == minRDistance;
end
leftEgoBoundary       = boundaries(leftEgoBoundaryIndex);
rightEgoBoundary      = boundaries(rightEgoBoundaryIndex);
```

Show the detected lane markers in the bird's-eye-view image and in the regular view.

```
xVehiclePoints = bottomOffset:distAheadOfSensor;
birdsEyeWithEgoLane = insertLaneBoundary(birdsEyeImage, leftEgoBoundary , birdsEyeConfig, xVehic
birdsEyeWithEgoLane = insertLaneBoundary(birdsEyeWithEgoLane, rightEgoBoundary, birdsEyeConfig, x

frameWithEgoLane = insertLaneBoundary(frame, leftEgoBoundary, sensor, xVehiclePoints, 'Color','Re
frameWithEgoLane = insertLaneBoundary(frameWithEgoLane, rightEgoBoundary, sensor, xVehiclePoints

figure
subplot('Position', [0, 0, 0.5, 1.0]) % [left, bottom, width, height] in normalized units
imshow(birdsEyeWithEgoLane)
subplot('Position', [0.5, 0, 0.5, 1.0])
imshow(frameWithEgoLane)
```



Locate Vehicles in Vehicle Coordinates

Detecting and tracking vehicles is critical in front collision warning (FCW) and autonomous emergency braking (AEB) systems.

Load an aggregate channel features (ACF) detector that is pretrained to detect the front and rear of vehicles. A detector like this can handle scenarios where issuing a collision warning is important. It is not sufficient, for example, for detecting a vehicle traveling across a road in front of the ego vehicle.

```
detector = vehicleDetectorACF();

% Width of a common vehicle is between 1.5 to 2.5 meters
vehicleWidth = [1.5, 2.5];
```

Use the `configureDetectorMonoCamera` function to specialize the generic ACF detector to take into account the geometry of the typical automotive application. By passing in this camera configuration, this new detector searches only for vehicles along the road's surface, because there is no point searching for vehicles high above the vanishing point. This saves computational time and reduces the number of false positives.

```
monoDetector = configureDetectorMonoCamera(detector, sensor, vehicleWidth);

[bboxes, scores] = detect(monoDetector, frame);
```

Because this example shows how to process only a single frame for demonstration purposes, you cannot apply tracking on top of the raw detections. The addition of tracking makes the results of returning vehicle locations more robust, because even when the vehicle is partly occluded, the

tracker continues to return the vehicle's location. For more information, see the “Track Multiple Vehicles Using a Camera” on page 7-170 example.

Next, convert vehicle detections to vehicle coordinates. The `computeVehicleLocations` function, included at the end of this example, calculates the location of a vehicle in vehicle coordinates given a bounding box returned by a detection algorithm in image coordinates. It returns the center location of the bottom of the bounding box in vehicle coordinates. Because we are using a monocular camera sensor and a simple homography, only distances along the surface of the road can be computed accurately. Computation of an arbitrary location in 3-D space requires use of stereo camera or another sensor capable of triangulation.

```
locations = computeVehicleLocations(bboxes, sensor);

% Overlay the detections on the video frame
imgOut = insertVehicleDetections(frame, locations, bboxes);
figure;
imshow(imgOut);
```

Simulate a Complete Sensor with Video Input

Now that you have an idea about the inner workings of the individual steps, let's put them together and apply them to a video sequence where we can also take advantage of temporal information.

Rewind the video to the beginning, and then process the video. The code below is shortened because all the key parameters were defined in the previous steps. Here, the parameters are used without further explanation.

```
videoReader.CurrentTime = 0;

isPlayerOpen = true;
snapshot     = [];
while hasFrame(videoReader) && isPlayerOpen

    % Grab a frame of video
    frame = readFrame(videoReader);

    % Compute birdsEyeView image
    birdsEyeImage = transformImage(birdsEyeConfig, frame);
    birdsEyeImage = rgb2gray(birdsEyeImage);

    % Detect lane boundary features
    birdsEyeViewBW = segmentLaneMarkerRidge(birdsEyeImage, birdsEyeConfig, ...
        approxLaneMarkerWidthVehicle, 'ROI', vehicleROI, ...
        'Sensitivity', laneSensitivity);

    % Obtain lane candidate points in vehicle coordinates
    [imageX, imageY] = find(birdsEyeViewBW);
    xyBoundaryPoints = imageToVehicle(birdsEyeConfig, [imageY, imageX]);

    % Find lane boundary candidates
    [boundaries, boundaryPoints] = findParabolicLaneBoundaries(xyBoundaryPoints,boundaryWidth, .
        'MaxNumBoundaries', maxLanes, 'validateBoundaryFcn', @validateBoundaryFcn);

    % Reject boundaries based on their length and strength
    isOfMinLength = arrayfun(@(b)diff(b.XExtent) > minXLength, boundaries);
    boundaries    = boundaries(isOfMinLength);
    isStrong      = [boundaries.Strength] > 0.2*maxStrength;
    boundaries    = boundaries(isStrong);

    % Classify lane marker type
    boundaries = classifyLaneTypes(boundaries, boundaryPoints);

    % Find ego lanes
    xOffset    = 0;   %  0 meters from the sensor
    distanceToBoundaries  = boundaries.computeBoundaryModel(xOffset);
    % Find candidate ego boundaries
    leftEgoBoundaryIndex  = [];
    rightEgoBoundaryIndex = [];
    minLDistance = min(distanceToBoundaries(distanceToBoundaries>0));
    minRDistance = max(distanceToBoundaries(distanceToBoundaries<=0));
    if ~isempty(minLDistance)
        leftEgoBoundaryIndex  = distanceToBoundaries == minLDistance;
    end
    if ~isempty(minRDistance)
        rightEgoBoundaryIndex = distanceToBoundaries == minRDistance;
    end
    leftEgoBoundary       = boundaries(leftEgoBoundaryIndex);
    rightEgoBoundary      = boundaries(rightEgoBoundaryIndex);

    % Detect vehicles
    [bboxes, scores] = detect(monoDetector, frame);
```

```
    locations = computeVehicleLocations(bboxes, sensor);

    % Visualize sensor outputs and intermediate results. Pack the core
    % sensor outputs into a struct.
    sensorOut.leftEgoBoundary  = leftEgoBoundary;
    sensorOut.rightEgoBoundary = rightEgoBoundary;
    sensorOut.vehicleLocations = locations;

    sensorOut.xVehiclePoints   = bottomOffset:distAheadOfSensor;
    sensorOut.vehicleBoxes     = bboxes;

    % Pack additional visualization data, including intermediate results
    intOut.birdsEyeImage   = birdsEyeImage;
    intOut.birdsEyeConfig  = birdsEyeConfig;
    intOut.vehicleScores   = scores;
    intOut.vehicleROI      = vehicleROI;
    intOut.birdsEyeBW      = birdsEyeViewBW;

    closePlayers = ~hasFrame(videoReader);
    isPlayerOpen = visualizeSensorResults(frame, sensor, sensorOut, ...
        intOut, closePlayers);

    timeStamp = 7.5333; % take snapshot for publishing at timeStamp seconds
    if abs(videoReader.CurrentTime - timeStamp) < 0.01
        snapshot = takeSnapshot(frame, sensor, sensorOut);
    end
end
```

Display the video frame. Snapshot is taken at `timeStamp` seconds.

```
if ~isempty(snapshot)
    figure
    imshow(snapshot)
end
```

Try the Sensor Design on a Different Video

The `helperMonoSensor` class assembles the setup and all the necessary steps to simulate the monocular camera sensor into a complete package that can be applied to any video. Since most parameters used by the sensor design are based on world units, the design is robust to changes in camera parameters, including the image size. Note that the code inside the `helperMonoSensor` class is different from the loop in the previous section, which was used to illustrate basic concepts.

Besides providing a new video, you must supply a camera configuration corresponding to that video. The process is shown here. Try it on your own videos.

```
% Sensor configuration
focalLength    = [309.4362, 344.2161];
principalPoint = [318.9034, 257.5352];
imageSize      = [480, 640];
height         = 2.1798;    % mounting height in meters from the ground
pitch          = 14;        % pitch of the camera in degrees

camIntrinsics = cameraIntrinsics(focalLength, principalPoint, imageSize);
sensor        = monoCamera(camIntrinsics, height, 'Pitch', pitch);

videoReader = VideoReader('caltech_washington1.avi');
```

Create the `helperMonoSensor` object and apply it to the video.

```
monoSensor   = helperMonoSensor(sensor);
monoSensor.LaneXExtentThreshold = 0.5;
% To remove false detections from shadows in this video, we only return
% vehicle detections with higher scores.
monoSensor.VehicleDetectionThreshold = 20;

isPlayerOpen = true;
snapshot     = [];
while hasFrame(videoReader) && isPlayerOpen

    frame = readFrame(videoReader); % get a frame

    sensorOut = processFrame(monoSensor, frame);

    closePlayers = ~hasFrame(videoReader);

    isPlayerOpen = displaySensorOutputs(monoSensor, frame, sensorOut, closePlayers);

    timeStamp = 11.1333; % take snapshot for publishing at timeStamp seconds
    if abs(videoReader.CurrentTime - timeStamp) < 0.01
        snapshot = takeSnapshot(frame, sensor, sensorOut);
    end

end
```

Display the video frame. Snapshot is taken at `timeStamp` seconds.

```
if ~isempty(snapshot)
    figure
    imshow(snapshot)
end
```

Supporting Functions

visualizeSensorResults displays core information and intermediate results from the monocular camera sensor simulation.

```
function isPlayerOpen = visualizeSensorResults(frame, sensor, sensorOut,...
    intOut, closePlayers)

    % Unpack the main inputs
    leftEgoBoundary  = sensorOut.leftEgoBoundary;
    rightEgoBoundary = sensorOut.rightEgoBoundary;
    locations        = sensorOut.vehicleLocations;

    xVehiclePoints   = sensorOut.xVehiclePoints;
    bboxes           = sensorOut.vehicleBoxes;

    % Unpack additional intermediate data
    birdsEyeViewImage = intOut.birdsEyeImage;
    birdsEyeConfig          = intOut.birdsEyeConfig;
    vehicleROI        = intOut.vehicleROI;
    birdsEyeViewBW    = intOut.birdsEyeBW;

    % Visualize left and right ego-lane boundaries in bird's-eye view
```

```
    birdsEyeWithOverlays = insertLaneBoundary(birdsEyeViewImage, leftEgoBoundary , birdsEyeConfig
    birdsEyeWithOverlays = insertLaneBoundary(birdsEyeWithOverlays, rightEgoBoundary, birdsEyeCon

    % Visualize ego-lane boundaries in camera view
    frameWithOverlays = insertLaneBoundary(frame, leftEgoBoundary, sensor, xVehiclePoints, 'Color
    frameWithOverlays = insertLaneBoundary(frameWithOverlays, rightEgoBoundary, sensor, xVehicleP

    frameWithOverlays = insertVehicleDetections(frameWithOverlays, locations, bboxes);

    imageROI = vehicleToImageROI(birdsEyeConfig, vehicleROI);
    ROI = [imageROI(1) imageROI(3) imageROI(2)-imageROI(1) imageROI(4)-imageROI(3)];

    % Highlight candidate lane points that include outliers
    birdsEyeViewImage = insertShape(birdsEyeViewImage, 'rectangle', ROI); % show detection ROI
    birdsEyeViewImage = imoverlay(birdsEyeViewImage, birdsEyeViewBW, 'blue');

    % Display the results
    frames = {frameWithOverlays, birdsEyeViewImage, birdsEyeWithOverlays};

    persistent players;
    if isempty(players)
        frameNames = {'Lane marker and vehicle detections', 'Raw segmentation', 'Lane marker dete
        players = helperVideoPlayerSet(frames, frameNames);
    end
    update(players, frames);

    % Terminate the loop when the first player is closed
    isPlayerOpen = isOpen(players, 1);

    if (~isPlayerOpen || closePlayers) % close down the other players
        clear players;
    end
end
```

computeVehicleLocations calculates the location of a vehicle in vehicle coordinates, given a bounding box returned by a detection algorithm in image coordinates. It returns the center location of the bottom of the bounding box in vehicle coordinates. Because a monocular camera sensor and a simple homography are used, only distances along the surface of the road can be computed. Computation of an arbitrary location in 3-D space requires use of a stereo camera or another sensor capable of triangulation.

```
function locations = computeVehicleLocations(bboxes, sensor)

locations = zeros(size(bboxes,1),2);
for i = 1:size(bboxes, 1)
    bbox  = bboxes(i, :);

    % Get [x,y] location of the center of the lower portion of the
    % detection bounding box in meters. bbox is [x, y, width, height] in
    % image coordinates, where [x,y] represents upper-left corner.
    yBottom = bbox(2) + bbox(4) - 1;
    xCenter = bbox(1) + (bbox(3)-1)/2; % approximate center

    locations(i,:) = imageToVehicle(sensor, [xCenter, yBottom]);
end
end
```

insertVehicleDetections inserts bounding boxes and displays [x,y] locations corresponding to returned vehicle detections.

```
function imgOut = insertVehicleDetections(imgIn, locations, bboxes)

imgOut = imgIn;

for i = 1:size(locations, 1)
    location = locations(i, :);
    bbox     = bboxes(i, :);

    label = sprintf('X=%0.2f, Y=%0.2f', location(1), location(2));

    imgOut = insertObjectAnnotation(imgOut, ...
        'rectangle', bbox, label, 'Color','g');
end
end
```

vehicleToImageROI converts ROI in vehicle coordinates to image coordinates in bird's-eye-view image.

```
function imageROI = vehicleToImageROI(birdsEyeConfig, vehicleROI)

vehicleROI = double(vehicleROI);

loc2 = abs(vehicleToImage(birdsEyeConfig, [vehicleROI(2) vehicleROI(4)]));
loc1 = abs(vehicleToImage(birdsEyeConfig, [vehicleROI(1) vehicleROI(4)]));
loc4 =     vehicleToImage(birdsEyeConfig, [vehicleROI(1) vehicleROI(4)]);
loc3 =     vehicleToImage(birdsEyeConfig, [vehicleROI(1) vehicleROI(3)]);

[minRoiX, maxRoiX, minRoiY, maxRoiY] = deal(loc4(1), loc3(1), loc2(2), loc1(2));

imageROI = round([minRoiX, maxRoiX, minRoiY, maxRoiY]);

end
```

validateBoundaryFcn rejects some of the lane boundary curves computed using the RANSAC algorithm.

```
function isGood = validateBoundaryFcn(params)

if ~isempty(params)
    a = params(1);

    % Reject any curve with a small 'a' coefficient, which makes it highly
    % curved.
    isGood = abs(a) < 0.003; % a from ax^2+bx+c
else
    isGood = false;
end
end
```

classifyLaneTypes determines lane marker types as `solid`, `dashed`, etc.

```
function boundaries = classifyLaneTypes(boundaries, boundaryPoints)

for bInd = 1 : numel(boundaries)
    vehiclePoints = boundaryPoints{bInd};
```

```
    % Sort by x
    vehiclePoints = sortrows(vehiclePoints, 1);

    xVehicle = vehiclePoints(:,1);
    xVehicleUnique = unique(xVehicle);

    % Dashed vs solid
    xdiff  = diff(xVehicleUnique);
    % Sufficiently large threshold to remove spaces between points of a
    % solid line, but not large enough to remove spaces between dashes
    xdifft = mean(xdiff) + 3*std(xdiff);
    largeGaps = xdiff(xdiff > xdifft);

    % Safe default
    boundaries(bInd).BoundaryType= LaneBoundaryType.Solid;
    if largeGaps>2
        % Ideally, these gaps should be consistent, but you cannot rely
        % on that unless you know that the ROI extent includes at least 3 dashes.
        boundaries(bInd).BoundaryType = LaneBoundaryType.Dashed;
    end
end

end
```

takeSnapshot captures the output for the HTML publishing report.

```
function I = takeSnapshot(frame, sensor, sensorOut)

    % Unpack the inputs
    leftEgoBoundary  = sensorOut.leftEgoBoundary;
    rightEgoBoundary = sensorOut.rightEgoBoundary;
    locations        = sensorOut.vehicleLocations;
    xVehiclePoints   = sensorOut.xVehiclePoints;
    bboxes           = sensorOut.vehicleBoxes;

    frameWithOverlays = insertLaneBoundary(frame, leftEgoBoundary, sensor, xVehiclePoints, 'Colo
    frameWithOverlays = insertLaneBoundary(frameWithOverlays, rightEgoBoundary, sensor, xVehicleP
    frameWithOverlays = insertVehicleDetections(frameWithOverlays, locations, bboxes);

    I = frameWithOverlays;

end
```

See Also

Apps
Camera Calibrator

Functions
`configureDetectorMonoCamera` | `estimateMonoCameraParameters` | `extrinsics` | `findParabolicLaneBoundaries` | `segmentLaneMarkerRidge`

Objects
`VideoReader` | `birdsEyeView` | `cameraIntrinsics` | `monoCamera` | `parabolicLaneBoundary`

More About

- “Track Multiple Vehicles Using a Camera” on page 7-170
- “Lane Keeping Assist with Lane Detection” on page 7-363
- “Forward Collision Warning Using Sensor Fusion” on page 7-125
- “Highway Lane Following” on page 7-653
- “Coordinate Systems in Automated Driving Toolbox” on page 1-2
- “Calibrate a Monocular Camera” on page 1-9

Train a Deep Learning Vehicle Detector

This example shows how to train a vision-based vehicle detector using deep learning.

Overview

Vehicle detection using computer vision is an important component for tracking vehicles around the ego vehicle. The ability to detect and track vehicles is required for many autonomous driving applications, such as for forward collision warning, adaptive cruise control, and automated lane keeping. Automated Driving Toolbox™ provides pretrained vehicle detectors (`vehicleDetectorFasterRCNN` and `vehicleDetectorACF`) to enable quick prototyping. However, the pretrained models might not suit every application, requiring you to train from scratch. This example shows how to train a vehicle detector from scratch using deep learning.

Deep learning is a powerful machine learning technique that you can use to train robust object detectors. Several deep learning techniques for object detection exist, including Faster R-CNN and you only look once (YOLO) v2. This example trains a Faster R-CNN vehicle detector using the `trainFasterRCNNObjectDetector` function. For more information, see “Object Detection using Deep Learning” (Computer Vision Toolbox).

Download Pretrained Detector

Download a pretrained detector to avoid having to wait for training to complete. If you want to train the detector, set the `doTrainingAndEval` variable to true.

```
doTrainingAndEval = false;
if ~doTrainingAndEval && ~exist('fasterRCNNResNet50EndToEndVehicleExample.mat','file')
    disp('Downloading pretrained detector (118 MB)...');
    pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/fasterRCNNResNet50EndToEn
    websave('fasterRCNNResNet50VehicleExample.mat',pretrainedURL);
end
```

Load Dataset

This example uses a small labeled dataset that contains 295 images. Each image contains one or two labeled instances of a vehicle. A small dataset is useful for exploring the Faster R-CNN training procedure, but in practice, more labeled images are needed to train a robust detector. Unzip the vehicle images and load the vehicle ground truth data.

```
unzip vehicleDatasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

The vehicle data is stored in a two-column table, where the first column contains the image file paths and the second column contains the vehicle bounding boxes.

Split the data set into a training set for training the detector and a test set for evaluating the detector. Select 60% of the data for training. Use the rest for evaluation.

```
rng(0)
shuffledIdx = randperm(height(vehicleDataset));
idx = floor(0.6 * height(vehicleDataset));
trainingDataTbl = vehicleDataset(shuffledIdx(1:idx),:);
testDataTbl = vehicleDataset(shuffledIdx(idx+1:end),:);
```

Use `imageDatastore` and `boxLabelDatastore` to create datastores for loading the image and label data during training and evaluation.

```
imdsTrain = imageDatastore(trainingDataTbl{:,'imageFilename'});
bldsTrain = boxLabelDatastore(trainingDataTbl(:,'vehicle'));

imdsTest = imageDatastore(testDataTbl{:,'imageFilename'});
bldsTest = boxLabelDatastore(testDataTbl(:,'vehicle'));
```

Combine image and box label datastores.

```
trainingData = combine(imdsTrain,bldsTrain);
testData = combine(imdsTest,bldsTest);
```

Display one of the training images and box labels.

```
data = read(trainingData);
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```

Create Faster R-CNN Detection Network

A Faster R-CNN object detection network is composed of a feature extraction network followed by two subnetworks. The feature extraction network is typically a pretrained CNN, such as ResNet-50 or

Inception v3. The first subnetwork following the feature extraction network is a region proposal network (RPN) trained to generate object proposals - areas in the image where objects are likely to exist. The second subnetwork is trained to predict the actual class of each object proposal.

The feature extraction network is typically a pretrained CNN (for details, see “Pretrained Deep Neural Networks” (Deep Learning Toolbox)). This example uses ResNet-50 for feature extraction. You can also use other pretrained networks such as MobileNet v2 or ResNet-18, depending on your application requirements.

Use `fasterRCNNLayers` to create a Faster R-CNN network automatically given a pretrained feature extraction network. `fasterRCNNLayers` requires you to specify several inputs that parameterize a Faster R-CNN network:

- Network input size
- Anchor boxes
- Feature extraction network

First, specify the network input size. When choosing the network input size, consider the minimum size required to run the network itself, the size of the training images, and the computational cost incurred by processing data at the selected size. When feasible, choose a network input size that is close to the size of the training image and larger than the input size required for the network. To reduce the computational cost of running the example, specify a network input size of [224 224 3], which is the minimum size required to run the network.

```
inputSize = [224 224 3];
```

Note that the training images used in this example are bigger than 224-by-224 and vary in size, so you must resize the images in a preprocessing step prior to training.

Next, use `estimateAnchorBoxes` to estimate anchor boxes based on the size of objects in the training data. To account for the resizing of the images prior to training, resize the training data for estimating anchor boxes. Use `transform` to preprocess the training data, then define the number of anchor boxes and estimate the anchor boxes.

```
preprocessedTrainingData = transform(trainingData, @(data)preprocessData(data,inputSize));
numAnchors = 4;
anchorBoxes = estimateAnchorBoxes(preprocessedTrainingData,numAnchors)
```

```
anchorBoxes = 4×2

    96    91
    68    65
   150   125
    38    29
```

For more information on choosing anchor boxes, see “Estimate Anchor Boxes From Training Data” (Computer Vision Toolbox) (Computer Vision Toolbox™) and “Anchor Boxes for Object Detection” (Computer Vision Toolbox).

Now, use `resnet50` to load a pretrained ResNet-50 model.

```
featureExtractionNetwork = resnet50;
```

Select `'activation_40_relu'` as the feature extraction layer. This feature extraction layer outputs feature maps that are downsampled by a factor of 16. This amount of downsampling is a good trade-

off between spatial resolution and the strength of the extracted features, as features extracted further down the network encode stronger image features at the cost of spatial resolution. Choosing the optimal feature extraction layer requires empirical analysis. You can use `analyzeNetwork` to find the names of other potential feature extraction layers within a network.

```
featureLayer = 'activation_40_relu';
```

Define the number of classes to detect.

```
numClasses = width(vehicleDataset)-1;
```

Create the Faster R-CNN object detection network.

```
lgraph = fasterRCNNLayers(inputSize,numClasses,anchorBoxes,featureExtractionNetwork,featureLayer
```

You can visualize the network using `analyzeNetwork` or Deep Network Designer from Deep Learning Toolbox™.

If more control is required over the Faster R-CNN network architecture, use Deep Network Designer to design the Faster R-CNN detection network manually. For more information, see “Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN” (Computer Vision Toolbox).

Data Augmentation

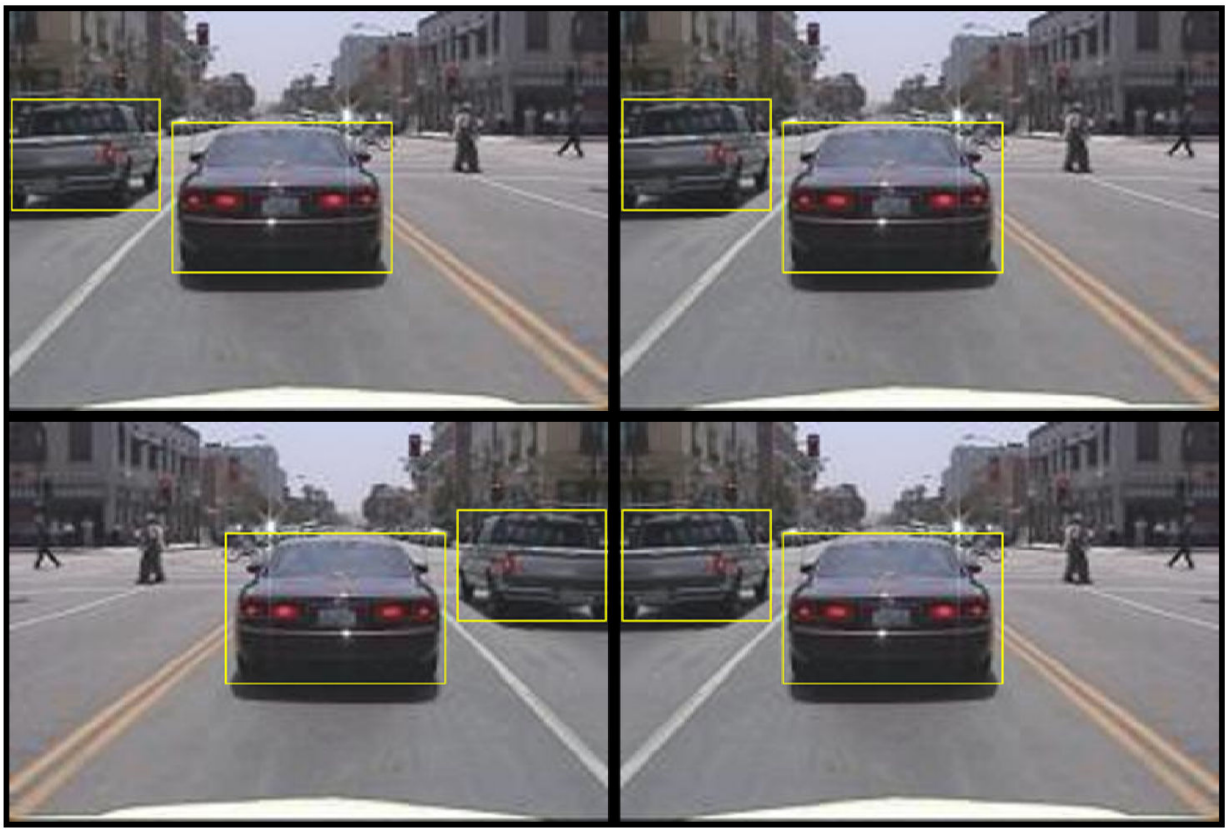
Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

Use `transform` to augment the training data by randomly flipping the image and associated box labels horizontally. Note that data augmentation is not applied to test data. Ideally, test data is representative of the original data and is left unmodified for unbiased evaluation.

```
augmentedTrainingData = transform(trainingData,@augmentData);
```

Read the same image multiple times and display the augmented training data.

```
augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedTrainingData);
    augmentedData{k} = insertShape(data{1},'Rectangle',data{2});
    reset(augmentedTrainingData);
end
figure
montage(augmentedData,'BorderSize',10)
```



Preprocess Training Data

Preprocess the augmented training data to prepare for training.

```
trainingData = transform(augmentedTrainingData,@(data)preprocessData(data,inputSize));
```

Read the preprocessed data.

```
data = read(trainingData);
```

Display the image and box bounding boxes.

```
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```

Train Faster R-CNN

Use `trainingOptions` to specify network training options. Set `'CheckpointPath'` to a temporary location. This enables the saving of partially trained detectors during the training process. If training is interrupted, such as by a power outage or system failure, you can resume training from the saved checkpoint.

```
options = trainingOptions('sgdm',...
    'MaxEpochs',7,...
    'MiniBatchSize',1,...
    'InitialLearnRate',1e-3,...
    'CheckpointPath',tempdir);
```

Use `trainFasterRCNNObjectDetector` to train Faster R-CNN object detector if `doTrainingAndEval` is true. Otherwise, load the pretrained network.

```
if doTrainingAndEval
    % Train the Faster R-CNN detector.
    % * Adjust NegativeOverlapRange and PositiveOverlapRange to ensure
```

```
    %   that training samples tightly overlap with ground truth.
    [detector, info] = trainFasterRCNNObjectDetector(trainingData,lgraph,options, ...
        'NegativeOverlapRange',[0 0.3], ...
        'PositiveOverlapRange',[0.6 1]);
else
    % Load pretrained detector for the example.
    pretrained = load('fasterRCNNResNet50EndToEndVehicleExample.mat');
    detector = pretrained.detector;
end
```

This example was verified on an Nvidia(TM) Titan X GPU with 12 GB of memory. Training the network took approximately 20 minutes. The training time varies depending on the hardware you use.

As a quick check, run the detector on one test image. Make sure you resize the image to the same size as the training images.

```
I = imread(testDataTbl.imageFilename{1});
I = imresize(I,inputSize(1:2));
[bboxes,scores] = detect(detector,I);
```

Display the results.

```
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I)
```

Evaluate Detector Using Test Set

Evaluate the trained object detector on a large set of images to measure the performance. Computer Vision Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). For this example, use the average precision metric to evaluate performance. The average precision provides a single number that incorporates the ability of the detector to make correct classifications (precision) and the ability of the detector to find all relevant objects (recall).

Apply the same preprocessing transform to the test data as for the training data.

```
testData = transform(testData,@(data)preprocessData(data,inputSize));
```

Run the detector on all the test images.

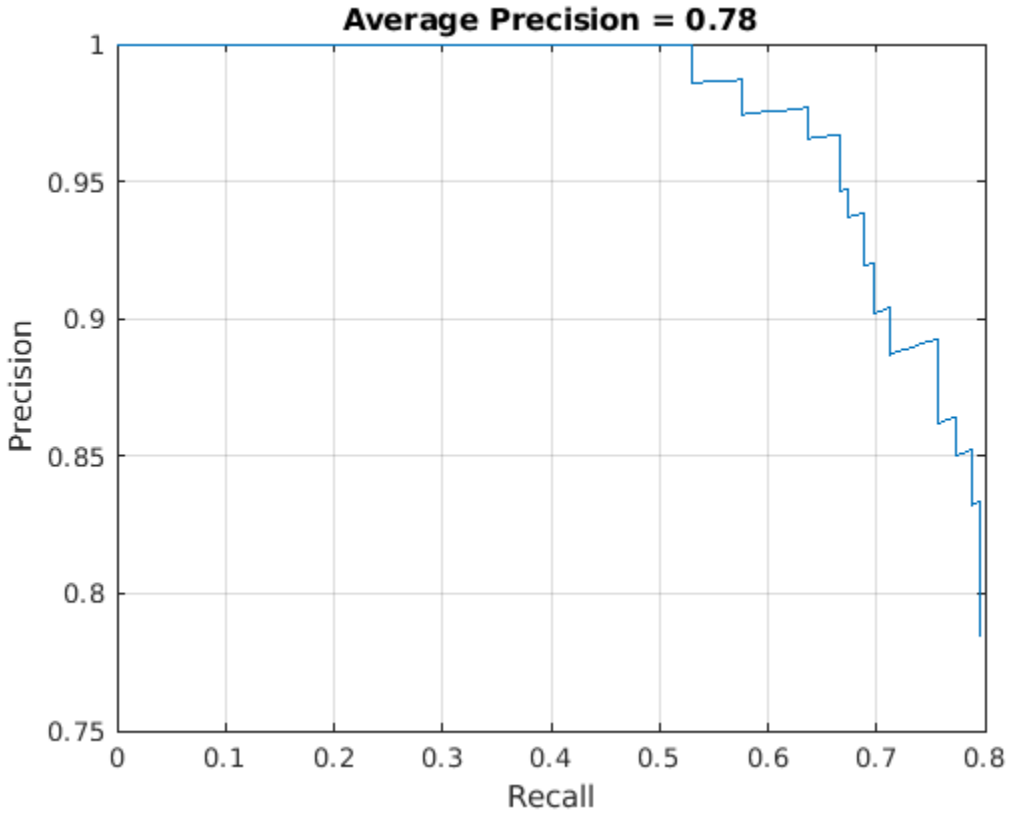
```
if doTrainingAndEval
    detectionResults = detect(detector,testData,'MinibatchSize',4);
else
    % Load pretrained detector for the example.
    pretrained = load('fasterRCNNResNet50EndToEndVehicleExample.mat');
    detectionResults = pretrained.detectionResults;
end
```

Evaluate the object detector using the average precision metric.

```
[ap, recall, precision] = evaluateDetectionPrecision(detectionResults,testData);
```

The precision/recall (PR) curve highlights how precise a detector is at varying levels of recall. The ideal precision is 1 at all recall levels. The use of more data can help improve the average precision but might require more training time. Plot the PR curve.

```
figure
plot(recall,precision)
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f', ap))
```



Supporting Functions

```
function data = augmentData(data)
% Randomly flip images and bounding boxes horizontally.
tform = randomAffine2d('XReflection',true);
rout = affineOutputView(size(data{1}),tform);
data{1} = imwarp(data{1},tform,'OutputView',rout);
data{2} = bboxwarp(data{2},tform,rout);
end

function data = preprocessData(data,targetSize)
% Resize image and bounding boxes to targetSize.
scale = targetSize(1:2)./size(data{1},[1 2]);
data{1} = imresize(data{1},targetSize(1:2));
data{2} = bboxresize(data{2},scale);
end
```

References

[1] Ren, S., K. He, R. Gershick, and J. Sun. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." *IEEE Transactions of Pattern Analysis and Machine Intelligence*. Vol. 39, Issue 6, June 2017, pp. 1137-1149.

[2] Girshick, R., J. Donahue, T. Darrell, and J. Malik. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*. Columbus, OH, June 2014, pp. 580-587.

[3] Girshick, R. "Fast R-CNN." *Proceedings of the 2015 IEEE International Conference on Computer Vision*. Santiago, Chile, Dec. 2015, pp. 1440-1448.

[4] Zitnick, C. L., and P. Dollar. "Edge Boxes: Locating Object Proposals from Edges." *European Conference on Computer Vision*. Zurich, Switzerland, Sept. 2014, pp. 391-405.

[5] Uijlings, J. R. R., K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders. "Selective Search for Object Recognition." *International Journal of Computer Vision*. Vol. 104, Number 2, Sept. 2013, pp. 154-171.

See Also

Functions

`trainFastRCNNObjectDetector` | `trainFasterRCNNObjectDetector` | `trainRCNNObjectDetector`

More About

- “Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN” (Computer Vision Toolbox)
- “Train Object Detector Using R-CNN Deep Learning” (Computer Vision Toolbox)
- “Object Detection Using Faster R-CNN Deep Learning” (Computer Vision Toolbox)

Ground Plane and Obstacle Detection Using Lidar

This example shows how to process 3-D lidar data from a sensor mounted on a vehicle by segmenting the ground plane and finding nearby obstacles. This can facilitate drivable path planning for vehicle navigation. The example also shows how to visualize streaming lidar data.

Create a Velodyne File Reader

The lidar data used in this example was recorded using a Velodyne HDL32E sensor mounted on a vehicle. Set up a `velodyneFileReader (Computer Vision Toolbox)` object to read the recorded PCAP file.

```
fileName    = 'lidarData_ConstructionRoad.pcap';
deviceModel = 'HDL32E';

veloReader = velodyneFileReader(fileName, deviceModel);
```

Read a Lidar Scan

Each scan of lidar data is stored as a 3-D point cloud. Efficiently processing this data using fast indexing and search is key to the performance of the sensor processing pipeline. This efficiency is achieved using the `pointCloud (Computer Vision Toolbox)` object, which internally organizes the data using a K-d tree data structure.

The `veloReader` constructs an organized `pointCloud` for each lidar scan. The `Location` property of the `pointCloud` is an M-by-N-by-3 matrix, containing the XYZ coordinates of points in meters. The point intensities are stored in `Intensity`.

```
% Read a scan of lidar data
ptCloud = readFrame(veloReader) %#ok<NOPTS>
```

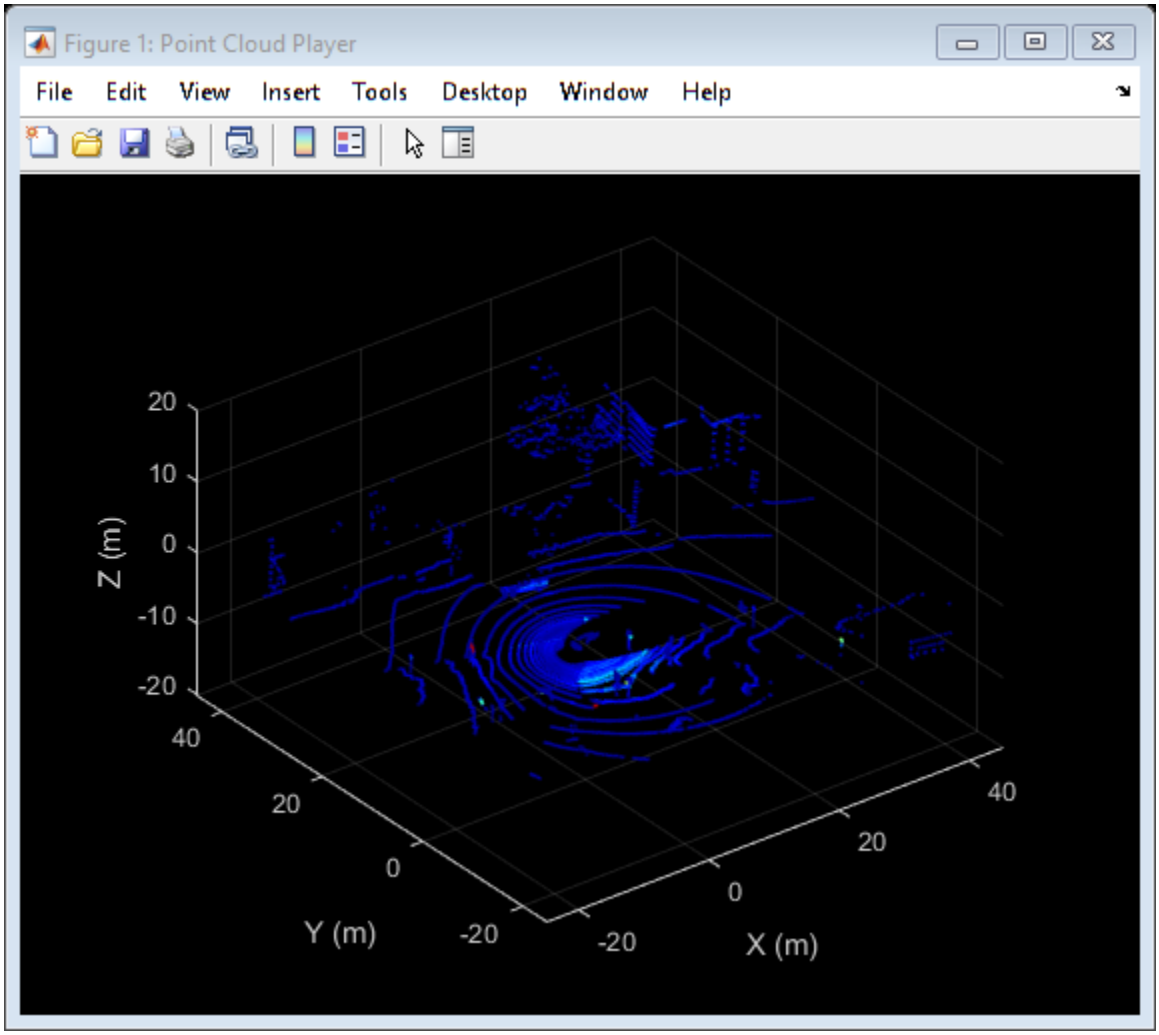
```
ptCloud =

  pointCloud with properties:

     Location: [32×1083×3 single]
        Color: []
       Normal: []
    Intensity: [32×1083 single]
        Count: 34656
      XLimits: [-80.0444 87.1780]
      YLimits: [-85.6287 92.8721]
      ZLimits: [-21.6060 14.3558]
```

Setup Streaming Point Cloud Display

The `pcplayer` can be used to visualize streaming point cloud data. Setup the region around the vehicle to display by configuring `pcplayer (Computer Vision Toolbox)`.

```
% Specify limits of point cloud display
xlimits = [-25 45]; % meters
ylimits = [-25 45];
zlimits = [-20 20];

% Create a pcplayer
```

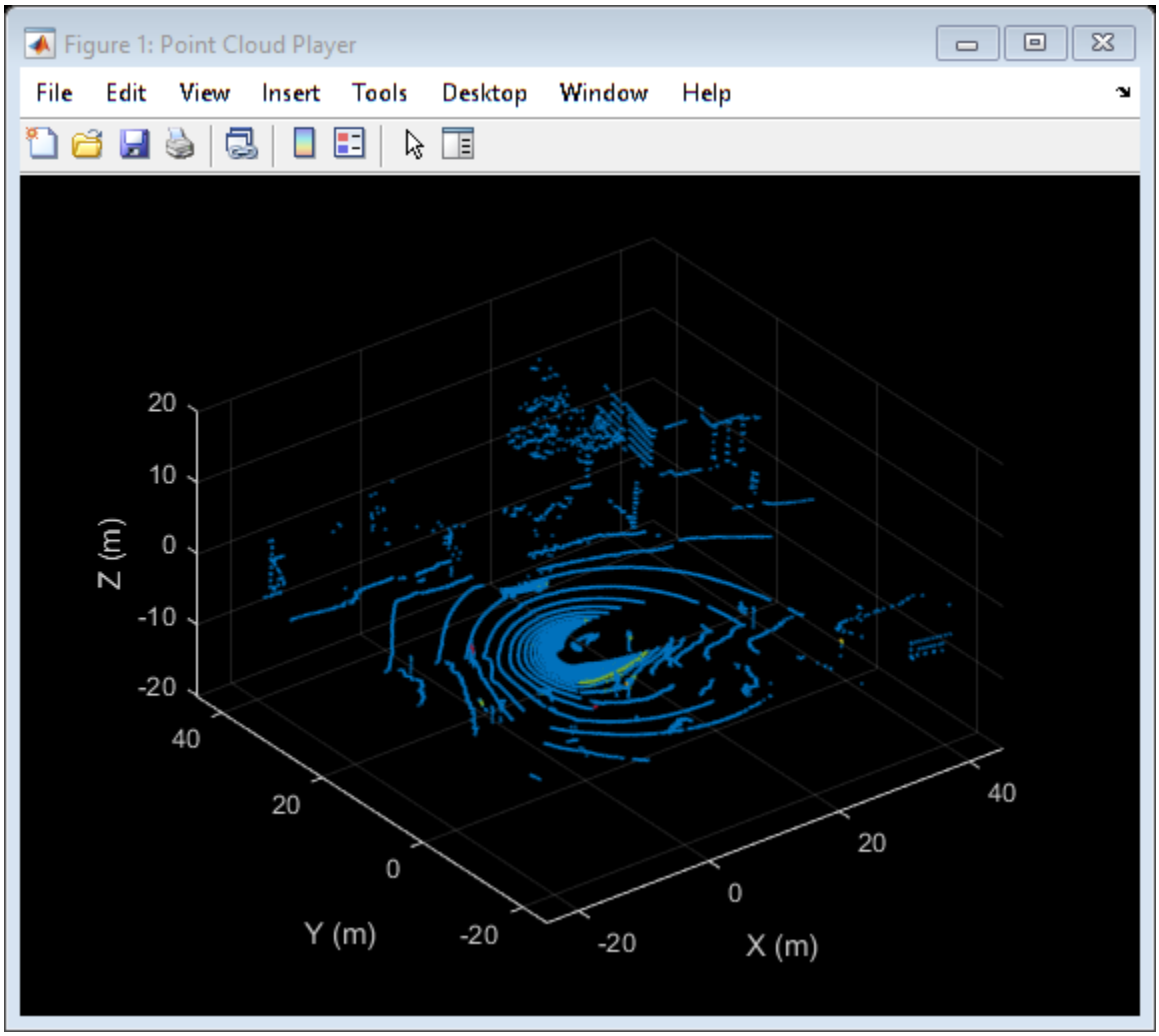
```
lidarViewer = pcplayer(xlimits, ylimits, zlimits);

% Customize player axes labels
xlabel(lidarViewer.Axes, 'X (m)')
ylabel(lidarViewer.Axes, 'Y (m)')
zlabel(lidarViewer.Axes, 'Z (m)')

% Display the raw lidar scan
view(lidarViewer, ptCloud)
```



In this example, we will be segmenting points belonging to the ground plane, the ego vehicle and nearby obstacles. Set the colormap for labeling these points.

```
% Define labels to use for segmented points
colorLabels = [...
    0      0.4470 0.7410; ... % Unlabeled points, specified as [R,G,B]
    0.4660 0.6740 0.1880; ... % Ground points
    0.9290 0.6940 0.1250; ... % Ego points
    0.6350 0.0780 0.1840];    % Obstacle points

% Define indices for each label
```

```
colors.Unlabeled = 1;
colors.Ground    = 2;
colors.Ego       = 3;
colors.Obstacle  = 4;

% Set the colormap
colormap(lidarViewer.Axes, colorLabels)
```



Segment the Ego Vehicle

The lidar is mounted on top of the vehicle, and the point cloud may contain points belonging to the vehicle itself, such as on the roof or hood. Knowing the dimensions of the vehicle, we can segment out points that are closest to the vehicle.

Create a `vehicleDimensions` object for storing dimensions of the vehicle.

```
vehicleDims = vehicleDimensions(); % Typical vehicle 4.7m by 1.8m by 1.4m
```

Specify the mounting location of the lidar in the vehicle coordinate system. The vehicle coordinate system is centered at the center of the rear-axle, on the ground, with positive X direction pointing

forward, positive Y towards the left, and positive Z upwards. In this example, the lidar is mounted on the top center of the vehicle, parallel to the ground.

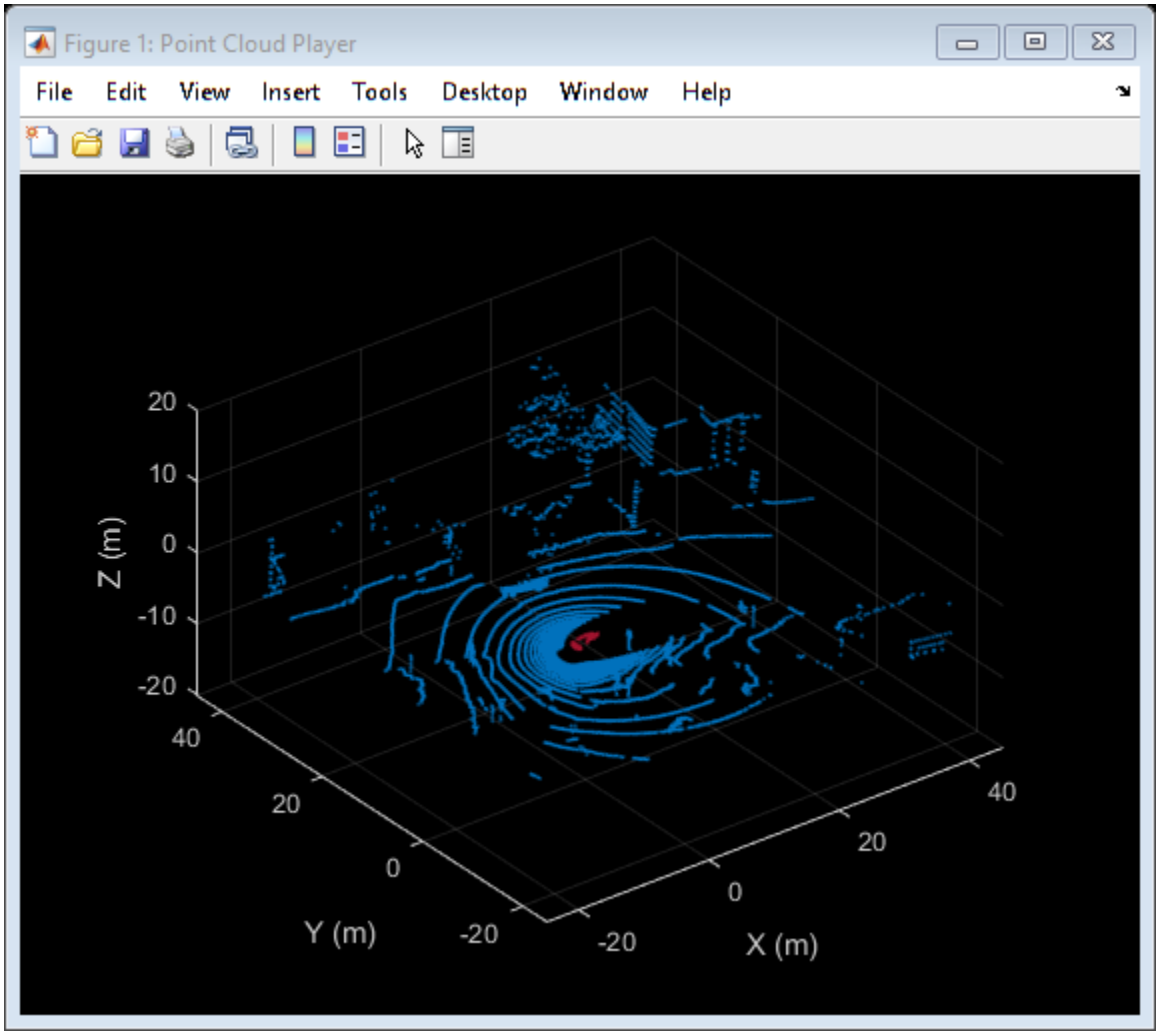
```
mountLocation = [...
    vehicleDims.Length/2 - vehicleDims.RearOverhang, ... % x
    0, ...                                               % y
    vehicleDims.Height];                                 % z
```

Segment the ego vehicle using the helper function `helperSegmentEgoFromLidarData`. This function segments all points within the cuboid defined by the ego vehicle. Store the segmented points in a struct `points`.

```
points = struct();
points.EgoPoints = helperSegmentEgoFromLidarData(ptCloud, vehicleDims, mountLocation);
```

Visualize the point cloud with segmented ego vehicle. Use the `helperUpdateView` helper function.

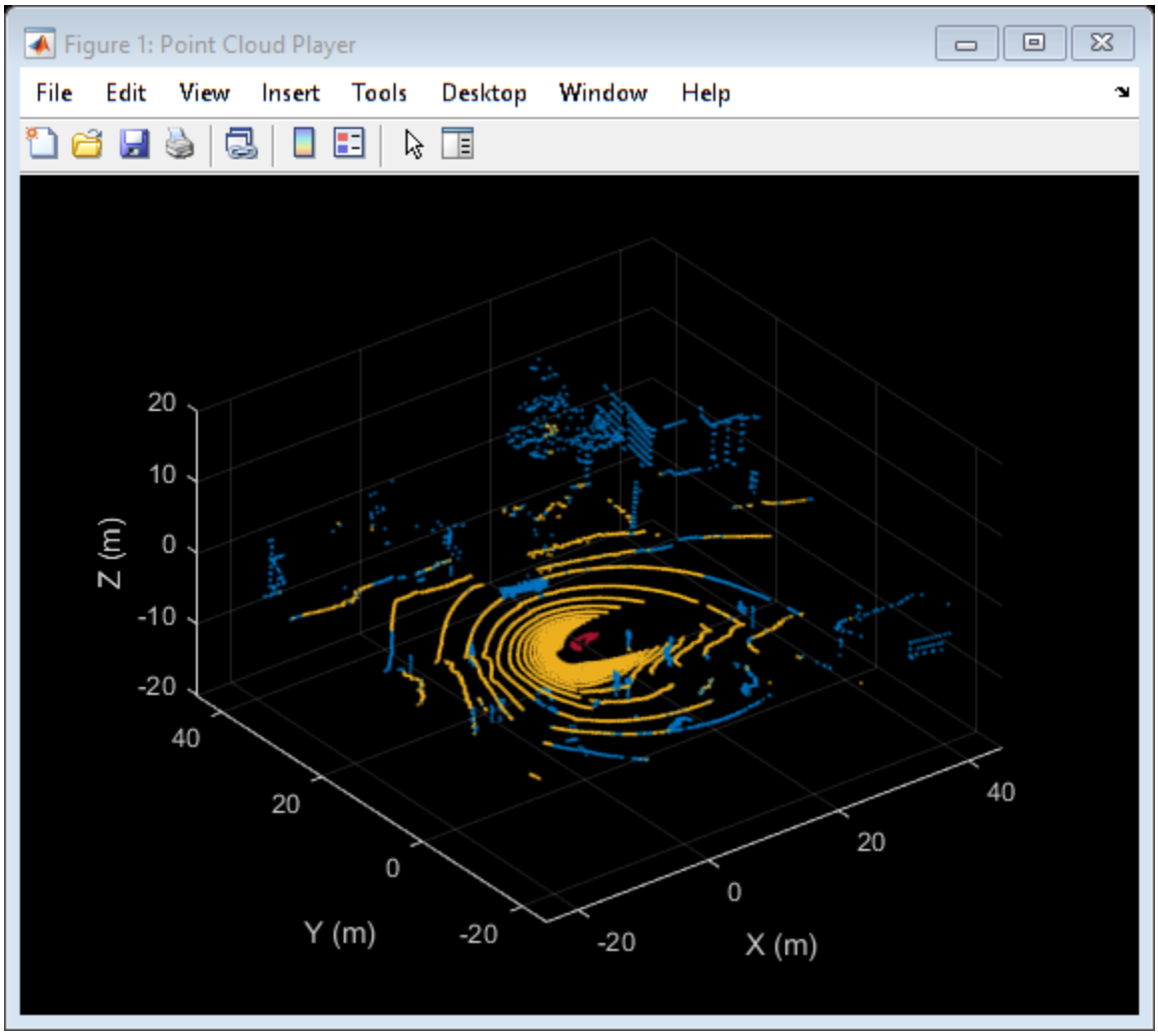
```
closePlayer = false;
helperUpdateView(lidarViewer, ptCloud, points, colors, closePlayer);
```



Segment Ground Plane and Nearby Obstacles

In order to identify obstacles from the lidar data, first segment the ground plane using the `segmentGroundFromLidarData (Computer Vision Toolbox)` function to accomplish this. This function segments points belonging to ground from organized lidar data.

```
elevationDelta = 10;
points.GroundPoints = segmentGroundFromLidarData(ptCloud, 'ElevationAngleDelta', elevationDelta);

% Visualize the segmented ground plane.
helperUpdateView(lidarViewer, ptCloud, points, colors, closePlayer);
```

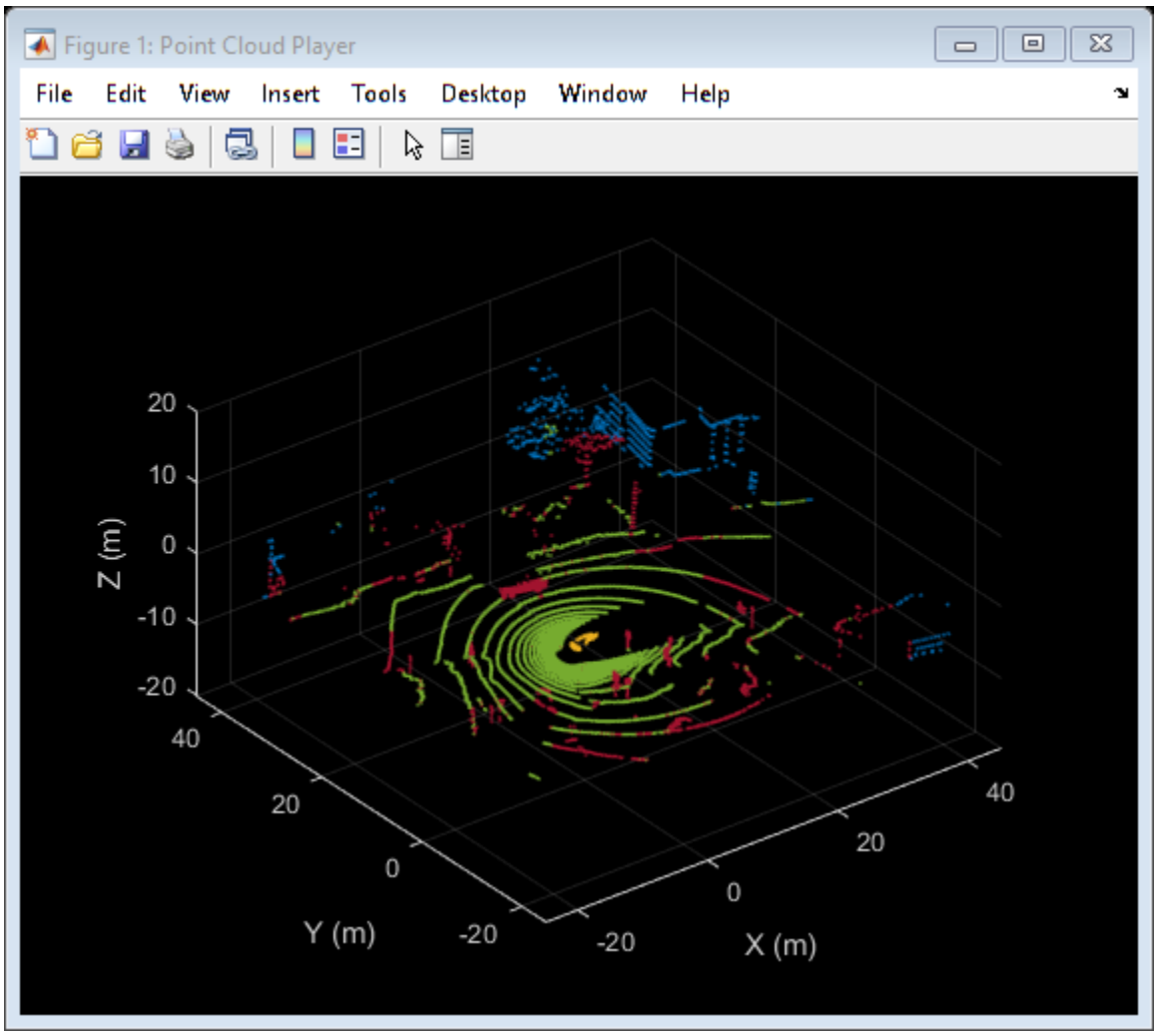


Remove points belonging to the ego vehicle and the ground plane by using the `select (Computer Vision Toolbox)` function on the point cloud. Specify the `'OutputSize'` as `'full'` to retain the organized nature of the point cloud.

```
nonEgoGroundPoints = ~points.EgoPoints & ~points.GroundPoints;
ptCloudSegmented = select(ptCloud, nonEgoGroundPoints, 'OutputSize', 'full');
```

Next, segment nearby obstacles by looking for all points that are not part of the ground or ego vehicle within some radius from the ego vehicle. This radius can be determined based on the range of the lidar and area of interest for further processing.

```
sensorLocation  = [0, 0, 0]; % Sensor is at the center of the coordinate system
radius          = 40; % meters

points.ObstaclePoints = findNeighborsInRadius(ptCloudSegmented, ...
    sensorLocation, radius);

% Visualize the segmented obstacles
helperUpdateView(lidarViewer, ptCloud, points, colors, closePlayer);
```



Process Lidar Sequence

Now that the point cloud processing pipeline for a single lidar scan has been laid out, put this all together to process 30 seconds from the sequence of recorded data. The code below is shortened since the key parameters have been defined in the previous steps. Here, the parameters are used without further explanation.

```
% Rewind the |veloReader| to start from the beginning of the sequence
reset(veloReader);

% Stop processing after 30 seconds
stopTime = veloReader.StartTime + seconds(30);

isPlayerOpen = true;
while hasFrame(veloReader) && veloReader.CurrentTime < stopTime && isPlayerOpen

    % Grab the next lidar scan
    ptCloud = readFrame(veloReader);

    % Segment points belonging to the ego vehicle
    points.EgoPoints = helperSegmentEgoFromLidarData(ptCloud, vehicleDims, mountLocation);

    % Segment points belonging to the ground plane
    points.GroundPoints = segmentGroundFromLidarData(ptCloud, 'ElevationAngleDelta', elevationDe

    % Remove points belonging to the ego vehicle and ground plane
    nonEgoGroundPoints = ~points.EgoPoints & ~points.GroundPoints;
    ptCloudSegmented = select(ptCloud, nonEgoGroundPoints, 'OutputSize', 'full');

    % Segment obstacles
    points.ObstaclePoints = findNeighborsInRadius(ptCloudSegmented, sensorLocation, radius);

    closePlayer = ~hasFrame(veloReader);

    % Update lidar display
    isPlayerOpen = helperUpdateView(lidarViewer, ptCloud, points, colors, closePlayer);
end
snapnow
```

Supporting Functions

`helperSegmentEgoFromLidarData` segments points belonging to the ego vehicle given the dimensions of the vehicle and mounting location.

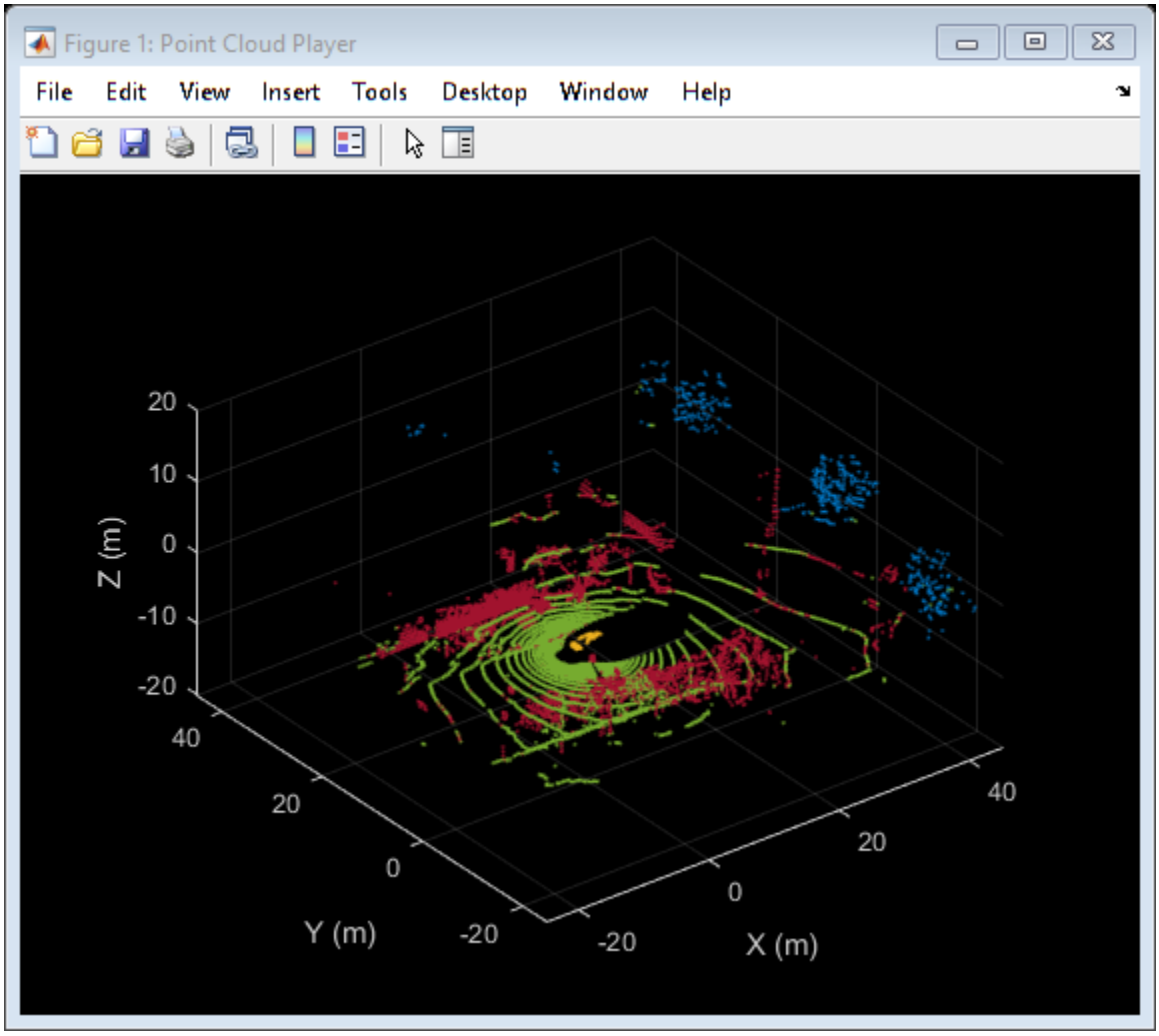
```
function egoPoints = helperSegmentEgoFromLidarData(ptCloud, vehicleDims, mountLocation)
%helperSegmentEgoFromLidarData segment ego vehicle points from lidar data
%   egoPoints = helperSegmentEgoFromLidarData(ptCloud,vehicleDims,mountLocation)
%   segments points belonging to the ego vehicle of dimensions vehicleDims
%   from the lidar scan ptCloud. The lidar is mounted at location specified
%   by mountLocation in the vehicle coordinate system. ptCloud is a
%   pointCloud object. vehicleDimensions is a vehicleDimensions object.
%   mountLocation is a 3-element vector specifying XYZ location of the
%   lidar in the vehicle coordinate system.
%
%   This function assumes that the lidar is mounted parallel to the ground
%   plane, with positive X direction pointing ahead of the vehicle,
%   positive Y direction pointing to the left of the vehicle in a
%   right-handed system.

% Buffer around ego vehicle
bufferZone = [0.1, 0.1, 0.1]; % in meters

% Define ego vehicle limits in vehicle coordinates
egoXMin = -vehicleDims.RearOverhang - bufferZone(1);
egoXMax = egoXMin + vehicleDims.Length + bufferZone(1);
```

```
egoYMin = -vehicleDims.Width/2 - bufferZone(2);
egoYMax = egoYMin + vehicleDims.Width + bufferZone(2);
egoZMin = 0 - bufferZone(3);
egoZMax = egoZMin + vehicleDims.Height + bufferZone(3);

egoXLimits = [egoXMin, egoXMax];
egoYLimits = [egoYMin, egoYMax];
egoZLimits = [egoZMin, egoZMax];

% Transform to lidar coordinates
egoXLimits = egoXLimits - mountLocation(1);
egoYLimits = egoYLimits - mountLocation(2);
egoZLimits = egoZLimits - mountLocation(3);

% Use logical indexing to select points inside ego vehicle cube
egoPoints = ptCloud.Location(:,:,1) > egoXLimits(1) ...
    & ptCloud.Location(:,:,1) < egoXLimits(2) ...
    & ptCloud.Location(:,:,2) > egoYLimits(1) ...
    & ptCloud.Location(:,:,2) < egoYLimits(2) ...
    & ptCloud.Location(:,:,3) > egoZLimits(1) ...
    & ptCloud.Location(:,:,3) < egoZLimits(2);
end
```

`helperUpdateView` updates the streaming point cloud display with the latest point cloud and associated color labels.

```
function isPlayerOpen = helperUpdateView(lidarViewer, ptCloud, points, colors, closePlayer)
%helperUpdateView update streaming point cloud display
%   isPlayerOpen = helperUpdateView(lidarViewer, ptCloud, points, colors, closePlayers)
%   updates the pcplayer object specified in lidarViewer with a new point
%   cloud ptCloud. Points specified in the struct points are colored
%   according to the colormap of lidarViewer using the labels specified by
%   the struct colors. closePlayer is a flag indicating whether to close
%   the lidarViewer.

if closePlayer
    hide(lidarViewer);
    isPlayerOpen = false;
    return;
end

scanSize = size(ptCloud.Location);
scanSize = scanSize(1:2);

% Initialize colormap
colormapValues = ones(scanSize, 'like', ptCloud.Location) * colors.Unlabeled;

if isfield(points, 'GroundPoints')
    colormapValues(points.GroundPoints) = colors.Ground;
end

if isfield(points, 'EgoPoints')
    colormapValues(points.EgoPoints) = colors.Ego;
end

if isfield(points, 'ObstaclePoints')
    colormapValues(points.ObstaclePoints) = colors.Obstacle;
end
```

```
% Update view
view(lidarViewer, ptCloud.Location, colormapValues)

% Check if player is open
isPlayerOpen = isOpen(lidarViewer);

end
```



See Also

Functions

`findNeighborsInRadius` | `pcfitplane` | `segmentGroundFromLidarData` | `segmentLidarData`

Objects

`pcplayer` | `pointCloud` | `vehicleDimensions` | `velodyneFileReader`

More About

- “Build a Map from Lidar Data” on page 7-539

Code Generation for Tracking and Sensor Fusion

This example shows how to generate C code for a MATLAB function that processes data recorded from a test vehicle and tracks the objects around it.

Automatic generation of code from MATLAB code has two key benefits:

1. Prototypes can be developed and debugged in the MATLAB environment. Once the MATLAB work is done, automatic C code generation makes the algorithms deployable to various targets. Additionally, the C code can be further tested by running the compiled MEX file in a MATLAB environment using the same visualization and analysis tools that were available during the prototyping phase.
2. After generating C code, you can generate executable code, which in many cases runs faster than the MATLAB code. The improved run time can be used to develop and deploy real-time sensor fusion and tracking systems. It also provides a better way to batch test the tracking systems on a large number of data sets.

The example explains how to modify the MATLAB code in the “Forward Collision Warning Using Sensor Fusion” on page 7-125 example to support code generation.

This example requires a MATLAB® Coder™ license for generating C code.

Modify and Run MATLAB Code

You can learn about the basics of code generation using MATLAB Coder from the “Introduction to Code Generation with Feature Matching and Registration” (Computer Vision Toolbox) example.

To generate C code, MATLAB Coder requires MATLAB code to be in the form of a function. Furthermore, the arguments of the function cannot be MATLAB classes.

In this example, the code for the forward collision warning (FCW) example has been restructured such that the functions that perform sensor fusion and tracking reside in a separate file, called `trackingForFCW_kernel.m`. Review this file for important information about memory allocation for code generation.

To preserve the state of the `multiObjectTracker` between calls to `trackingForFCW_kernel.m`, the tracker is defined as a `persistent` variable.

This function takes as an input a frame of the recorded data that includes:

1. Vision objects - A `struct` that contains 10 vision objects.
2. Radar objects - A `struct` that contains 36 radar objects.
3. Inertial measurement - A `struct` containing speed and yaw rate.
4. Lane reports - A `struct` array with parameters for the left and right lane boundaries.

Similarly, the outputs from a function that supports code generation cannot be objects. The outputs from `trackingForFCW_kernel.m` are:

1. Confirmed tracks - A `struct` array that contains a variable number of tracks.
2. Ego lane - A `struct` array with the parameters of left and right lane boundaries.
3. Number of tracks - An integer scalar.

4 Information about the most important object (MIO) and warning level from the FCW logic.

By restructuring the code this way, you can reuse the same display tools used in the FCW example. These tools still run in MATLAB and do not require code generation.

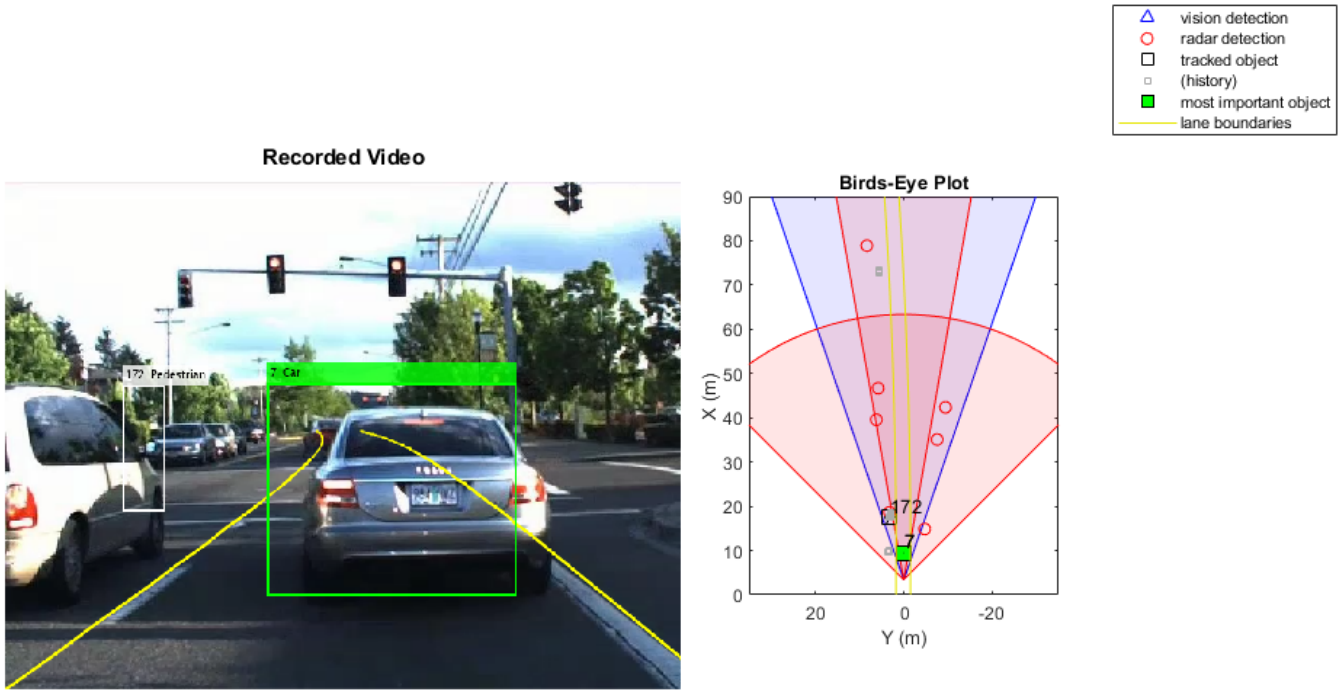
Run the following lines of code to load the recorded data and prepare the display in a similar way to the FCW example.

```
% If a previous tracker is defined, clear it
clear trackingForFCW_kernel

% Set up the display
videoFile = '01_city_c2s_fcw_10s.mp4';
sensorConfigFile = 'SensorConfigurationData.mat';
[videoReader, videoDisplayHandle, bepPlotters, sensor] = helperCreateFCWDemoDisplay(videoFile,se

% Read the recorded detections file
detfile = '01_city_c2s_fcw_10s_sensor.mat';
[visionObjects, radarObjects, imu, lanes, timeStep, numSteps] = helperReadSensorRecordingsFile(de

% An initial ego lane is calculated. If the recorded lane information is
% invalid, define the lane boundaries as straight lines half a lane
% distance on each side of the car.
laneWidth = 3.6; % meters
egoLane = struct('left', [0 0 laneWidth/2], 'right', [0 0 -laneWidth/2]);

% Prepare some time variables
timeStamp  = 0;    % Time since the beginning of the recording
index = 0;         % Index into the recorded sensor data

% Define the position and velocity selectors:
% The state vector is in constant acceleration: [x;vx;ax;y;vy;ay]
% Constant acceleration position: [x;y] = [1 0 0 0 0 0; 0 0 0 1 0 0] * State
positionSelector = [1 0 0 0 0 0; 0 0 0 1 0 0];
% Constant acceleration velocity: [x;y] = [0 1 0 0 0 0; 0 0 0 0 1 0] * State
velocitySelector = [0 1 0 0 0 0; 0 0 0 0 1 0];
```

Now run the example by calling the `trackingForFCW_kernel` function in MATLAB. This initial run provides a baseline to compare the results and enables you to collect some metrics about the performance of the tracker when it runs in MATLAB or as a MEX file.

```
% Allocate memory for number of tracks and time measurement in MATLAB
numTracks = zeros(1, numSteps);
runTimes  = zeros(1, numSteps);
while index < numSteps && ishghandle(videoDisplayHandle)
    % Update scenario counters
    index = index + 1;
    timeStamp = timeStamp + timeStep;
    tic;

    % Call the MATLAB tracking kernel file to perform the tracking
    [tracks, egoLane, numTracks(index), mostImportantObject] = trackingForFCW_kernel(...
        visionObjects(index), radarObjects(index), imu(index), lanes(index), egoLane, timeStamp,

    runTimes(index) = toc; % Gather MATLAB run time data

    % Update video and bird's-eye plot displays
    frame = readFrame(videoReader);     % Read video frame
    laneBoundaries = [parabolicLaneBoundary(egoLane.left);parabolicLaneBoundary(egoLane.right)];
```

```
    helperUpdateFCWDemoDisplay(frame, videoDisplayHandle, bepPlotters, laneBoundaries, sensor, .
        tracks, mostImportantObject, positionSelector, velocitySelector, visionObjects(index), ra
end
```



Compile the MATLAB Function into a MEX File

Use the `codegen` function to compile the `trackingForFCW_kernel` function into a MEX file. You can specify the `-report` option to generate a compilation report that shows the original MATLAB code and the associated files that were created during C code generation. Consider creating a temporary directory where MATLAB Coder can store generated files. Note that unless you use the `-o` option to specify the name of the executable, the generated MEX file has the same name as the original MATLAB file with `_mex` appended.

MATLAB Coder requires that you specify the properties of all the input arguments. The inputs are used by the tracker to create the correct data types and sizes for objects used in the tracking. The data types and sizes must not change between data frames. One easy way to do this is to define the input properties by example at the command line using the `-args` option. For more information, see "Define Input Properties by Example at the Command Line" (MATLAB Coder).

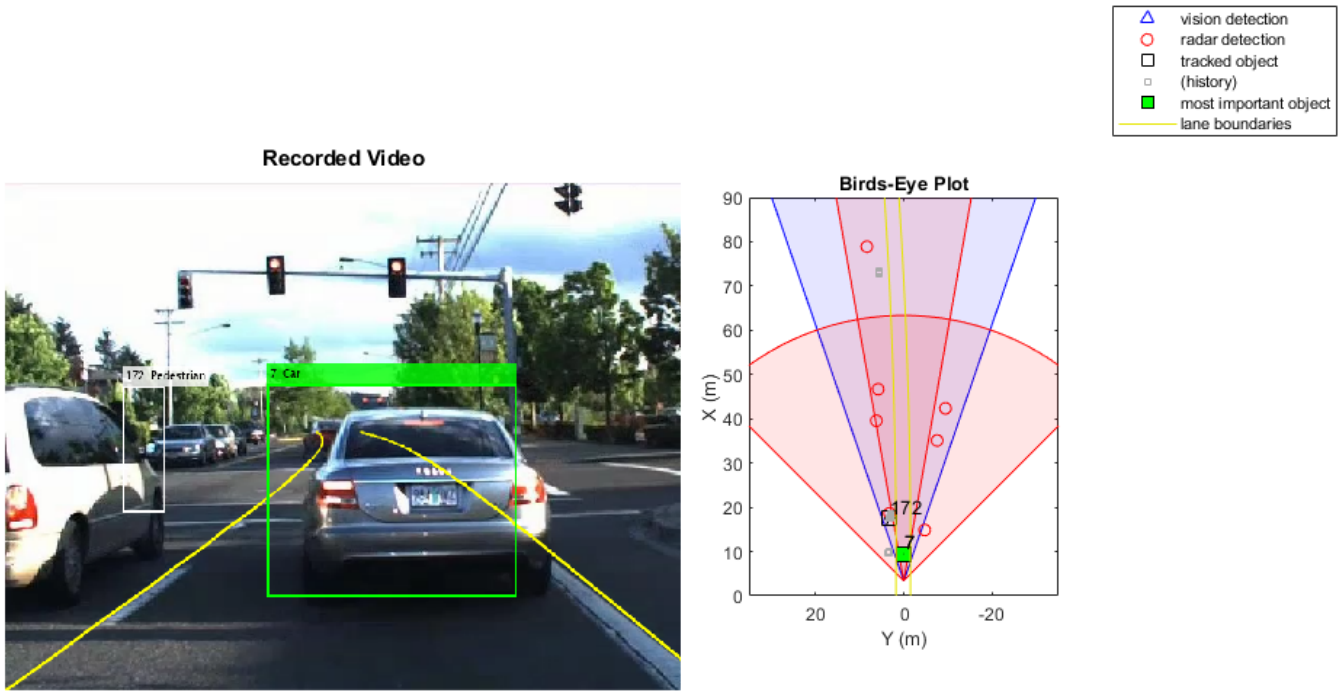
```
% Define the properties of the input based on the data in the first time frame.
compInputs  = {visionObjects(1), radarObjects(1), imu(1), lanes(1), egoLane, timeStamp, position
```

```
% Code generation may take some time.
h = msgbox({'Generating code. This may take a few minutes...';'This message box will close when c
% Generate code.
try
    codegen trackingForFCW_kernel -args compInputs;
    close(h)
catch ME
    close(h)
    delete(videoDisplayHandle.Parent.Parent)
    throw(ME)
end
```

Run the Generated Code

Now that the code has been generated, run the exact same scenario with the generated MEX file `trackingForFCW_kernel_mex`. Everything else remains the same.

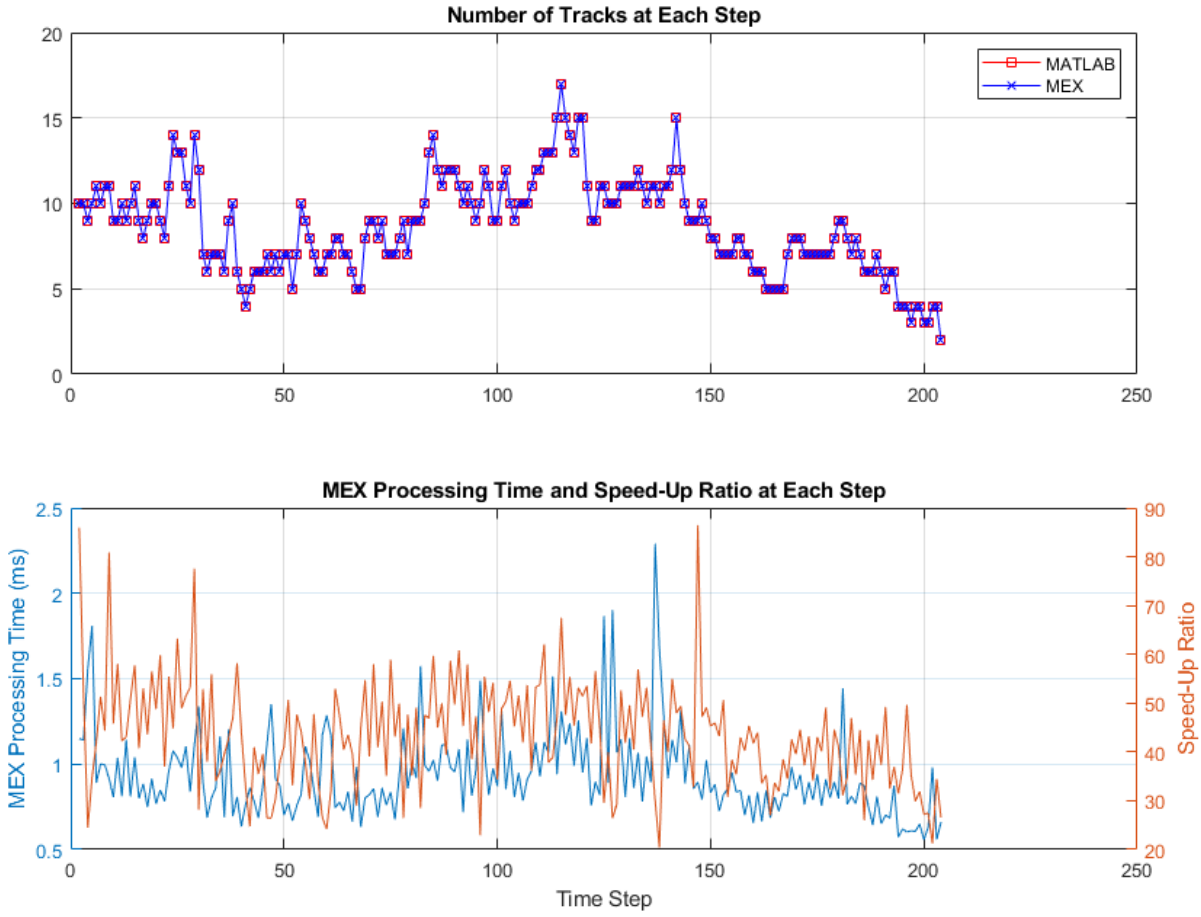
```
% If a previous tracker is defined, clear it
clear trackingForFCW_kernel_mex

% Allocate memory for number of tracks and time measurement
numTracksMex = zeros(1, numSteps);
runTimesMex  = zeros(1, numSteps);

% Reset the data and video counters
index = 0;
videoReader.CurrentTime = 0;

while index < numSteps && ishghandle(videoDisplayHandle)
    % Update scenario counters
    index = index + 1;
    timeStamp = timeStamp + timeStep;
    tic;

    % Call the generated MEX file to perform the tracking
    [tracks, egoLane, numTracksMex(index), mostImportantObject] = trackingForFCW_kernel_mex(...
        visionObjects(index), radarObjects(index), imu(index), lanes(index), egoLane, timeStamp,

    runTimesMex(index) = toc; % Gather MEX run time data

    % Update video and bird's-eye plot displays
    frame = readFrame(videoReader);     % Read video frame
    laneBoundaries = [parabolicLaneBoundary(egoLane.left);parabolicLaneBoundary(egoLane.right)];
    helperUpdateFCWDemoDisplay(frame, videoDisplayHandle, bepPlotters, laneBoundaries, sensor, .
        tracks, mostImportantObject, positionSelector, velocitySelector, visionObjects(index), ra
end
```



Compare the Results of the Two Runs

Compare the results and the performance of the generated code vs. the MATLAB code. The following plots compare the number of tracks maintained by the trackers at each time step. They also show the amount of time it took to process each call to the function.

```
figure(2)
subplot(2,1,1)
plot(2:numSteps, numTracks(2:end), 'rs-', 2:numSteps, numTracksMex(2:end), 'bx-')
title('Number of Tracks at Each Step');
legend('MATLAB', 'MEX')
grid
subplot(2,1,2)
yyaxis left
plot(2:numSteps, runTimesMex(2:end)*1e3);
ylabel('MEX Processing Time (ms)');
yyaxis right
plot(2:numSteps, runTimes(2:end) ./ runTimesMex(2:end))
ylabel('Speed-Up Ratio');
title('MEX Processing Time and Speed-Up Ratio at Each Step')
grid
xlabel('Time Step')
```



The top plot shows that the number of tracks that were maintained by each tracker are the same. It measures the size of the tracking problem in terms of number of tracks.

The bottom plot shows the time required for the MATLAB and generated code functions to process each step. Note that the first step requires a disproportionately longer time, because the trackers have to be constructed in the first step. Thus, the first time step is ignored.

The results show that the MEX code is much faster than the MATLAB code. They also show the number of milliseconds required by the MEX code to perform each update step on your computer. For example, on a computer with a CPU clock speed of 2.6 GHz running Windows 7, the time required for the MEX code to run an update step was less than 4 ms. As a reference, the recorded data used in this example was sampled every 50 ms, so the MEX run time is short enough to allow real-time tracking.

Display the CPU clock speed and average speed-up ratio.

```
p = profile('info');
speedUpRatio = mean(runTimes(2:end) ./ runTimesMex(2:end));
disp(['The generated code is ', num2str(speedUpRatio), ' times faster than the MATLAB code.']);
disp(['The computer clock speed is ', num2str(p.ClockSpeed / 1e9), ' GHz.']);
```

```
The generated code is 43.4007 times faster than the MATLAB code.
The computer clock speed is 3.601 GHz.
```

Summary

This example showed how to generate C code from MATLAB code for sensor fusion and tracking.

The main benefits of automatic code generation are the ability to prototype in the MATLAB environment, generating a MEX file that can run in the MATLAB environment, and deploying to a target using C code. In most cases, the generated code is faster than the MATLAB code, and can be used for batch testing of algorithms and generating real-time tracking systems.

See Also

`multiObjectTracker`

More About

- “Forward Collision Warning Using Sensor Fusion” on page 7-125
- “Introduction to Code Generation with Feature Matching and Registration” (Computer Vision Toolbox)

Forward Collision Warning Using Sensor Fusion

This example shows how to perform forward collision warning by fusing data from vision and radar sensors to track objects in front of the vehicle.

Overview

Forward collision warning (FCW) is an important feature in driver assistance and automated driving systems, where the goal is to provide correct, timely, and reliable warnings to the driver before an impending collision with the vehicle in front. To achieve the goal, vehicles are equipped with forward-facing vision and radar sensors. Sensor fusion is required to increase the probability of accurate warnings and minimize the probability of false warnings.

For the purposes of this example, a test car (the ego vehicle) was equipped with various sensors and their outputs were recorded. The sensors used for this example were:

1. Vision sensor, which provided lists of observed objects with their classification and information about lane boundaries. The object lists were reported 10 times per second. Lane boundaries were reported 20 times per second.
2. Radar sensor with medium and long range modes, which provided lists of unclassified observed objects. The object lists were reported 20 times per second.
3. IMU, which reported the speed and turn rate of the ego vehicle 20 times per second.
4. Video camera, which recorded a video clip of the scene in front of the car. Note: This video is not used by the tracker and only serves to display the tracking results on video for verification.

The process of providing a forward collision warning comprises the following steps:

1. Obtain the data from the sensors.
2. Fuse the sensor data to get a list of tracks, i.e., estimated positions and velocities of the objects in front of the car.
3. Issue warnings based on the tracks and FCW criteria. The FCW criteria are based on the Euro NCAP AEB test procedure and take into account the relative distance and relative speed to the object in front of the car.

For more information about tracking multiple objects, see “Multiple Object Tracking” (Computer Vision Toolbox).

The visualization in this example is done using `monoCamera` and `birdsEyePlot`. For brevity, the functions that create and update the display were moved to helper functions outside of this example. For more information on how to use these displays, see “Annotate Video Using Detections in Vehicle Coordinates” on page 7-9 and “Visualize Sensor Coverage, Detections, and Tracks” on page 7-226.

This example is a script, with the main body shown here and helper routines in the form of local functions in the sections that follow. For more details about local functions, see “Add Functions to Scripts”.

```
% Set up the display
[videoReader, videoDisplayHandle, bepPlotters, sensor] = helperCreateFCWDemoDisplay('01_city_c2s_

% Read the recorded detections file
[visionObjects, radarObjects, inertialMeasurementUnit, laneReports, ...
    timeStep, numSteps] = readSensorRecordingsFile('01_city_c2s_fcw_10s_sensor.mat');
```

```
% An initial ego lane is calculated. If the recorded lane information is
% invalid, define the lane boundaries as straight lines half a lane
% distance on each side of the car
laneWidth = 3.6; % meters
egoLane = struct('left', [0 0 laneWidth/2], 'right', [0 0 -laneWidth/2]);

% Prepare some time variables
time = 0;           % Time since the beginning of the recording
currentStep = 0;    % Current timestep
snapTime = 9.3;     % The time to capture a snapshot of the display

% Initialize the tracker
[tracker, positionSelector, velocitySelector] = setupTracker();

while currentStep < numSteps && ishghandle(videoDisplayHandle)
    % Update scenario counters
    currentStep = currentStep + 1;
    time = time + timeStep;

    % Process the sensor detections as objectDetection inputs to the tracker
    [detections, laneBoundaries, egoLane] = processDetections(...
        visionObjects(currentStep), radarObjects(currentStep), ...
        inertialMeasurementUnit(currentStep), laneReports(currentStep), ...
        egoLane, time);

    % Using the list of objectDetections, return the tracks, updated to time
    confirmedTracks = updateTracks(tracker, detections, time);

    % Find the most important object and calculate the forward collision
    % warning
    mostImportantObject = findMostImportantObject(confirmedTracks, egoLane, positionSelector, ve

    % Update video and birds-eye plot displays
    frame = readFrame(videoReader);     % Read video frame
    helperUpdateFCWDemoDisplay(frame, videoDisplayHandle, bepPlotters, ...
        laneBoundaries, sensor, confirmedTracks, mostImportantObject, positionSelector, ...
        velocitySelector, visionObjects(currentStep), radarObjects(currentStep));

    % Capture a snapshot
    if time >= snapTime && time < snapTime + timeStep
        snapnow;
    end
end
```

Create the Multi-Object Tracker

The `multiObjectTracker` tracks the objects around the ego vehicle based on the object lists reported by the vision and radar sensors. By fusing information from both sensors, the probability of a false collision warning is reduced.

The `setupTracker` function returns the `multiObjectTracker`. When creating a `multiObjectTracker`, consider the following:

1. `FilterInitializationFcn`: The likely motion and measurement models. In this case, the objects are expected to have a constant acceleration motion. Although you can configure a linear Kalman filter for this model, `initConstantAccelerationFilter` configures an extended Kalman filter. See the 'Define a Kalman filter' section.

2. `AssignmentThreshold`: How far detections can fall from tracks. The default value for this parameter is 30. If there are detections that are not assigned to tracks, but should be, increase this value. If there are detections that get assigned to tracks that are too far, decrease this value. This example uses 35.
3. `DeletionThreshold`: When a track is confirmed, it should not be deleted on the first update that no detection is assigned to it. Instead, it should be coasted (predicted) until it is clear that the track is not getting any sensor information to update it. The logic is that if the track is missed *P* out of *Q* times it should be deleted. The default value for this parameter is 5-out-of-5. In this case, the tracker is called 20 times a second and there are two sensors, so there is no need to modify the default.
4. `ConfirmationThreshold`: The parameters for confirming a track. A new track is initialized with every unassigned detection. Some of these detections might be false, so all the tracks are initialized as `'Tentative'`. To confirm a track, it has to be detected at least *M* times in *N* tracker updates. The choice of *M* and *N* depends on the visibility of the objects. This example uses the default of 2 detections out of 3 updates.

The outputs of `setupTracker` are:

- `tracker` - The `multiObjectTracker` that is configured for this case.
- `positionSelector` - A matrix that specifies which elements of the State vector are the position: `position = positionSelector * State`
- `velocitySelector` - A matrix that specifies which elements of the State vector are the velocity: `velocity = velocitySelector * State`

```
function [tracker, positionSelector, velocitySelector] = setupTracker()
    tracker = multiObjectTracker(...
        'FilterInitializationFcn', @initConstantAccelerationFilter, ...
        'AssignmentThreshold', 35, 'ConfirmationThreshold', [2 3], ...
        'DeletionThreshold', 5);

    % The State vector is:
    %   In constant velocity:     State = [x;vx;y;vy]
    %   In constant acceleration: State = [x;vx;ax;y;vy;ay]

    % Define which part of the State is the position. For example:
    %   In constant velocity:     [x;y] = [1 0 0 0; 0 0 1 0] * State
    %   In constant acceleration: [x;y] = [1 0 0 0 0 0; 0 0 0 1 0 0] * State
    positionSelector = [1 0 0 0 0 0; 0 0 0 1 0 0];

    % Define which part of the State is the velocity. For example:
    %   In constant velocity:     [x;y] = [0 1 0 0; 0 0 0 1] * State
    %   In constant acceleration: [x;y] = [0 1 0 0 0 0; 0 0 0 0 1 0] * State
    velocitySelector = [0 1 0 0 0 0; 0 0 0 0 1 0];
end
```

Define a Kalman Filter

The `multiObjectTracker` defined in the previous section uses the filter initialization function defined in this section to create a Kalman filter (linear, extended, or unscented). This filter is then used for tracking each object around the ego vehicle.

```
function filter = initConstantAccelerationFilter(detection)
% This function shows how to configure a constant acceleration filter. The
% input is an objectDetection and the output is a tracking filter.
% For clarity, this function shows how to configure a trackingKF,
```

```
% trackingEKF, or trackingUKF for constant acceleration.
%
% Steps for creating a filter:
%   1. Define the motion model and state
%   2. Define the process noise
%   3. Define the measurement model
%   4. Initialize the state vector based on the measurement
%   5. Initialize the state covariance based on the measurement noise
%   6. Create the correct filter

    % Step 1: Define the motion model and state
    % This example uses a constant acceleration model, so:
    STF = @constacc;     % State-transition function, for EKF and UKF
    STFJ = @constaccjac; % State-transition function Jacobian, only for EKF
    % The motion model implies that the state is [x;vx;ax;y;vy;ay]
    % You can also use constvel and constveljac to set up a constant
    % velocity model, constturn and constturnjac to set up a constant turn
    % rate model, or write your own models.

    % Step 2: Define the process noise
    dt = 0.05; % Known timestep size
    sigma = 1; % Magnitude of the unknown acceleration change rate
    % The process noise along one dimension
    Q1d = [dt^4/4, dt^3/2, dt^2/2; dt^3/2, dt^2, dt; dt^2/2, dt, 1] * sigma^2;
    Q = blkdiag(Q1d, Q1d); % 2-D process noise

    % Step 3: Define the measurement model
    MF = @fcwmeas;       % Measurement function, for EKF and UKF
    MJF = @fcwmeasjac;   % Measurement Jacobian function, only for EKF

    % Step 4: Initialize a state vector based on the measurement
    % The sensors measure [x;vx;y;vy] and the constant acceleration model's
    % state is [x;vx;ax;y;vy;ay], so the third and sixth elements of the
    % state vector are initialized to zero.
    state = [detection.Measurement(1); detection.Measurement(2); 0; detection.Measurement(3); de

    % Step 5: Initialize the state covariance based on the measurement
    % noise. The parts of the state that are not directly measured are
    % assigned a large measurement noise value to account for that.
    L = 100; % A large number relative to the measurement noise
    stateCov = blkdiag(detection.MeasurementNoise(1:2,1:2), L, detection.MeasurementNoise(3:4,3:4

    % Step 6: Create the correct filter.
    % Use 'KF' for trackingKF, 'EKF' for trackingEKF, or 'UKF' for trackingUKF
    FilterType = 'EKF';

    % Creating the filter:
    switch FilterType
        case 'EKF'
            filter = trackingEKF(STF, MF, state,...
                'StateCovariance', stateCov, ...
                'MeasurementNoise', detection.MeasurementNoise(1:4,1:4), ...
                'StateTransitionJacobianFcn', STFJ, ...
                'MeasurementJacobianFcn', MJF, ...
                'ProcessNoise', Q ...
                );
        case 'UKF'
            filter = trackingUKF(STF, MF, state, ...
```

```
                'StateCovariance', stateCov, ...
                'MeasurementNoise', detection.MeasurementNoise(1:4,1:4), ...
                'Alpha', 1e-1, ...
                'ProcessNoise', Q ...
                );
        case 'KF' % The ConstantAcceleration model is linear and KF can be used
            % Define the measurement model: measurement = H * state
            % In this case:
            %   measurement = [x;vx;y;vy] = H * [x;vx;ax;y;vy;ay]
            % So, H = [1 0 0 0 0 0; 0 1 0 0 0 0; 0 0 0 1 0 0; 0 0 0 0 1 0]
            %
            % Note that ProcessNoise is automatically calculated by the
            % ConstantAcceleration motion model
            H = [1 0 0 0 0 0; 0 1 0 0 0 0; 0 0 0 1 0 0; 0 0 0 0 1 0];
            filter = trackingKF('MotionModel', '2D Constant Acceleration', ...
                'MeasurementModel', H, 'State', state, ...
                'MeasurementNoise', detection.MeasurementNoise(1:4,1:4), ...
                'StateCovariance', stateCov);
    end
end
```

Process and Format the Detections

The recorded information must be processed and formatted before it can be used by the tracker. This has the following steps:

1. Filtering out unnecessary radar clutter detections. The radar reports many objects that correspond to fixed objects, which include: guard-rails, the road median, traffic signs, etc. If these detections are used in the tracking, they create false tracks of fixed objects at the edges of the road and therefore must be removed before calling the tracker. Radar objects are considered nonclutter if they are either stationary in front of the car or moving in its vicinity.
2. Formatting the detections as input to the tracker, i.e., an array of `objectDetection` elements. See the `processVideo` and `processRadar` supporting functions at the end of this example.

```
function [detections,laneBoundaries, egoLane] = processDetections...
        (visionFrame, radarFrame, IMUFrame, laneFrame, egoLane, time)
    % Inputs:
    %   visionFrame - objects reported by the vision sensor for this time frame
    %   radarFrame  - objects reported by the radar sensor for this time frame
    %   IMUFrame    - inertial measurement unit data for this time frame
    %   laneFrame   - lane reports for this time frame
    %   egoLane     - the estimated ego lane
    %   time        - the time corresponding to the time frame

    % Remove clutter radar objects
    [laneBoundaries, egoLane] = processLanes(laneFrame, egoLane);
    realRadarObjects = findNonClutterRadarObjects(radarFrame.object,...
        radarFrame.numObjects, IMUFrame.velocity, laneBoundaries);

    % Return an empty list if no objects are reported

    % Counting the total number of objects
    detections = {};
    if (visionFrame.numObjects + numel(realRadarObjects)) == 0
        return;
    end
```

```
        % Process the remaining radar objects
        detections = processRadar(detections, realRadarObjects, time);

        % Process video objects
        detections = processVideo(detections, visionFrame, time);
    end
```

Update the Tracker

To update the tracker, call the `updateTracks` method with the following inputs:

1. `tracker` - The `multiObjectTracker` that was configured earlier. See the 'Create the Multi-Object Tracker' section.
2. `detections` - A list of `objectDetection` objects that was created by `processDetections`
3. `time` - The current scenario time.

The output from the tracker is a `struct` array of tracks.

Find the Most Important Object and Issue a Forward Collision Warning

The most important object (MIO) is defined as the track that is in the ego lane and is closest in front of the car, i.e., with the smallest positive *x* value. To lower the probability of false alarms, only confirmed tracks are considered.

Once the MIO is found, the relative speed between the car and MIO is calculated. The relative distance and relative speed determine the forward collision warning. There are 3 cases of FCW:

1. Safe (green): There is no car in the ego lane (no MIO), the MIO is moving away from the car, or the distance to the MIO remains constant.
2. Caution (yellow): The MIO is moving closer to the car, but is still at a distance above the FCW distance. FCW distance is calculated using the Euro NCAP AEB Test Protocol. Note that this distance varies with the relative speed between the MIO and the car, and is greater when the closing speed is higher.
3. Warn (red): The MIO is moving closer to the car, and its distance is less than the FCW distance, d_{FCW}.

Euro NCAP AEB Test Protocol defines the following distance calculation:

$$d_{FCW} = 1.2 * v_{rel} + \frac{v_{rel}^2}{2a_{max}}$$

where:

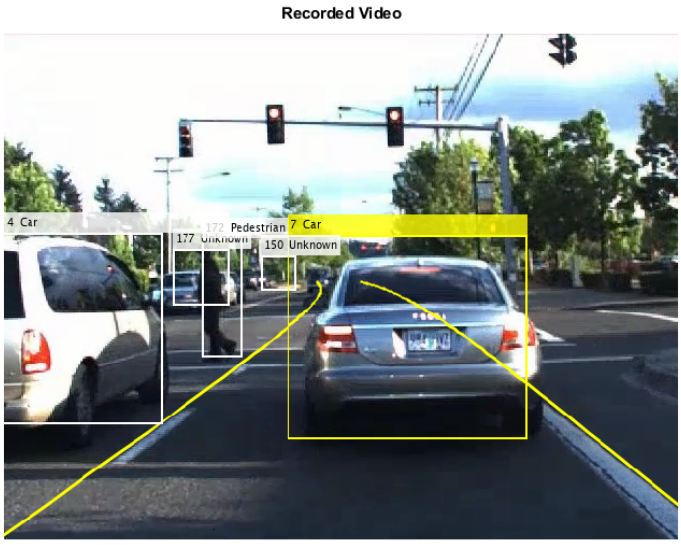
d_{FCW} is the forward collision warning distance.

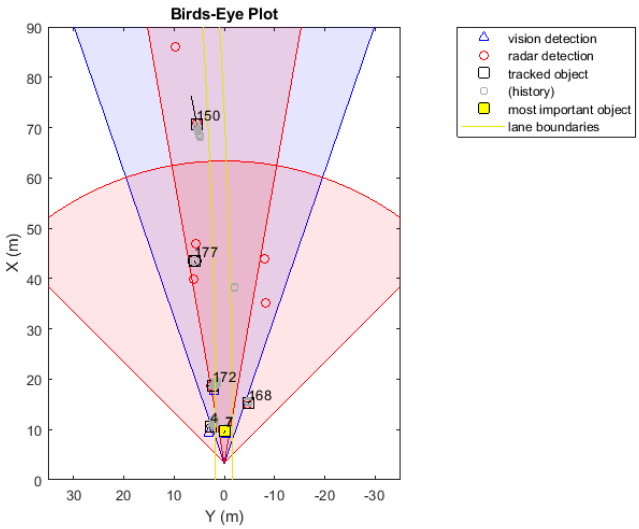
v_{rel} is the relative velocity between the two vehicles.

a_{max} is the maximum deceleration, defined to be 40% of the gravity acceleration.

```
    function mostImportantObject = findMostImportantObject(confirmedTracks,egoLane,positionSelec

        % Initialize outputs and parameters
        MIO = [];               % By default, there is no MIO
        trackID = [];           % By default, there is no trackID associated with an MIO
        FCW = 3;                % By default, if there is no MIO, then FCW is 'safe'
```

```
        threatColor = 'green';  % By default, the threat color is green
        maxX = 1000;  % Far enough forward so that no track is expected to exceed this distance
        gAccel = 9.8; % Constant gravity acceleration, in m/s^2
        maxDeceleration = 0.4 * gAccel; % Euro NCAP AEB definition
        delayTime = 1.2; % Delay time for a driver before starting to brake, in seconds

        positions = getTrackPositions(confirmedTracks, positionSelector);
        velocities = getTrackVelocities(confirmedTracks, velocitySelector);

        for i = 1:numel(confirmedTracks)
            x = positions(i,1);
            y = positions(i,2);

            relSpeed = velocities(i,1); % The relative speed between the cars, along the lane

            if x < maxX && x > 0 % No point checking otherwise
                yleftLane  = polyval(egoLane.left,  x);
                yrightLane = polyval(egoLane.right, x);
                if (yrightLane <= y) && (y <= yleftLane)
                    maxX = x;
                    trackID = i;
                    MIO = confirmedTracks(i).TrackID;
                    if relSpeed < 0 % Relative speed indicates object is getting closer
                        % Calculate expected braking distance according to
                        % Euro NCAP AEB Test Protocol
                        d = abs(relSpeed) * delayTime + relSpeed^2 / 2 / maxDeceleration;
                        if x <= d % 'warn'
                            FCW = 1;
                            threatColor = 'red';
                        else % 'caution'
                            FCW = 2;
                            threatColor = 'yellow';
                        end
                    end
                end
            end
        end
        mostImportantObject = struct('ObjectID', MIO, 'TrackIndex', trackID, 'Warning', FCW, 'Th
    end
```





Summary

This example showed how to create a forward collision warning system for a vehicle equipped with vision, radar, and IMU sensors. It used `objectDetection` objects to pass the sensor reports to the `multiObjectTracker` object that fused them and tracked objects in front of the ego vehicle.

Try using different parameters for the tracker to see how they affect the tracking quality. Try modifying the tracking filter to use `trackingKF` or `trackingUKF`, or to define a different motion model, e.g., constant velocity or constant turn. Finally, you can try to define your own motion model.

Supporting Functions

readSensorRecordingsFile Reads recorded sensor data from a file

```
function [visionObjects, radarObjects, inertialMeasurementUnit, laneReports, ...
    timeStep, numSteps] = readSensorRecordingsFile(sensorRecordingFileName)
% Read Sensor Recordings
% The |ReadDetectionsFile| function reads the recorded sensor data file.
% The recorded data is a single structure that is divided into the
% following substructures:
%
% # |inertialMeasurementUnit|, a struct array with fields: timeStamp,
%   velocity, and yawRate. Each element of the array corresponds to a
%   different timestep.
% # |laneReports|, a struct array with fields: left and right. Each element
%   of the array corresponds to a different timestep.
%   Both left and right are structures with fields: isValid, confidence,
%   boundaryType, offset, headingAngle, and curvature.
% # |radarObjects|, a struct array with fields: timeStamp (see below),
%   numObjects (integer) and object (struct). Each element of the array
%   corresponds to a different timestep.
%   |object| is a struct array, where each element is a separate object,
%   with the fields: id, status, position(x;y;z), velocity(vx,vy,vz),
%   amplitude, and rangeMode.
%   Note: z is always constant and vz=0.
% # |visionObjects|, a struct array with fields: timeStamp (see below),
```

```
%   numObjects (integer) and object (struct). Each element of the array
%   corresponds to a different timestep.
%   |object| is a struct array, where each element is a separate object,
%   with the fields: id, classification, position (x;y;z),
%   velocity(vx;vy;vz), size(dx;dy;dz). Note: z=vy=vz=dx=dz=0
%
% The timeStamp for recorded vision and radar objects is a uint64 variable
% holding microseconds since the Unix epoch. Timestamps are recorded about
% 50 milliseconds apart. There is a complete synchronization between the
% recordings of vision and radar detections, therefore the timestamps are
% not used in further calculations.

A = load(sensorRecordingFileName);
visionObjects = A.vision;
radarObjects = A.radar;
laneReports = A.lane;
inertialMeasurementUnit = A.inertialMeasurementUnit;

timeStep = 0.05;                % Data is provided every 50 milliseconds
numSteps = numel(visionObjects); % Number of recorded timesteps
end
```

processLanes Converts sensor-reported lanes to `parabolicLaneBoundary` lanes and maintains a persistent ego lane estimate

```
function [laneBoundaries, egoLane] = processLanes(laneReports, egoLane)
% Lane boundaries are updated based on the laneReports from the recordings.
% Since some laneReports contain invalid (isValid = false) reports or
% impossible parameter values (-1e9), these lane reports are ignored and
% the previous lane boundary is used.
leftLane    = laneReports.left;
rightLane   = laneReports.right;

% Check the validity of the reported left lane
cond = (leftLane.isValid && leftLane.confidence) && ...
    ~(leftLane.headingAngle == -1e9 || leftLane.curvature == -1e9);
if cond
    egoLane.left = cast([leftLane.curvature, leftLane.headingAngle, leftLane.offset], 'double');
end

% Update the left lane boundary parameters or use the previous ones
leftParams  = egoLane.left;
leftBoundaries = parabolicLaneBoundary(leftParams);
leftBoundaries.Strength = 1;

% Check the validity of the reported right lane
cond = (rightLane.isValid && rightLane.confidence) && ...
    ~(rightLane.headingAngle == -1e9 || rightLane.curvature == -1e9);
if cond
    egoLane.right  = cast([rightLane.curvature, rightLane.headingAngle, rightLane.offset], 'doub
end

% Update the right lane boundary parameters or use the previous ones
rightParams = egoLane.right;
rightBoundaries = parabolicLaneBoundary(rightParams);
rightBoundaries.Strength = 1;
```

```
laneBoundaries = [leftBoundaries, rightBoundaries];
end
```

findNonClutterRadarObjects Removes radar objects that are considered part of the clutter

```
function realRadarObjects = findNonClutterRadarObjects(radarObject, numRadarObjects, egoSpeed, la
% The radar objects include many objects that belong to the clutter.
% Clutter is defined as a stationary object that is not in front of the
% car. The following types of objects pass as nonclutter:
%
% # Any object in front of the car
% # Any moving object in the area of interest around the car, including
%   objects that move at a lateral speed around the car

    % Allocate memory
    normVs = zeros(numRadarObjects, 1);
    inLane = zeros(numRadarObjects, 1);
    inZone = zeros(numRadarObjects, 1);

    % Parameters
    LaneWidth = 3.6;            % What is considered in front of the car
    ZoneWidth = 1.7*LaneWidth;  % A wider area of interest
    minV = 1;                   % Any object that moves slower than minV is considered stationary
    for j = 1:numRadarObjects
        [vx, vy] = calculateGroundSpeed(radarObject(j).velocity(1),radarObject(j).velocity(2),ego
        normVs(j) = norm([vx,vy]);
        laneBoundariesAtObject = computeBoundaryModel(laneBoundaries, radarObject(j).position(1)
        laneCenter = mean(laneBoundariesAtObject);
        inLane(j) = (abs(radarObject(j).position(2) - laneCenter) <= LaneWidth/2);
        inZone(j) = (abs(radarObject(j).position(2) - laneCenter) <= max(abs(vy)*2, ZoneWidth));
    end
    realRadarObjectsIdx = union(...
        intersect(find(normVs > minV), find(inZone == 1)), ...
        find(inLane == 1));

    realRadarObjects = radarObject(realRadarObjectsIdx);
end
```

calculateGroundSpeed Calculates the true ground speed of a radar-reported object from the relative speed and the ego vehicle speed

```
function [Vx,Vy] = calculateGroundSpeed(Vxi,Vyi,egoSpeed)
% Inputs
%   (Vxi,Vyi) : relative object speed
%   egoSpeed  : ego vehicle speed
% Outputs
%   [Vx,Vy]   : ground object speed

    Vx = Vxi + egoSpeed;    % Calculate longitudinal ground speed
    theta = atan2(Vyi,Vxi); % Calculate heading angle
    Vy = Vx * tan(theta);   % Calculate lateral ground speed

end
```

processVideo Converts reported vision objects to a list of `objectDetection` objects

```
function postProcessedDetections = processVideo(postProcessedDetections, visionFrame, t)
% Process the video objects into objectDetection objects
```

```
numRadarObjects = numel(postProcessedDetections);
numVisionObjects = visionFrame.numObjects;
if numVisionObjects
    classToUse = class(visionFrame.object(1).position);
    visionMeasCov = cast(diag([2,2,2,100]), classToUse);
    % Process Vision Objects:
    for i=1:numVisionObjects
        object = visionFrame.object(i);
        postProcessedDetections{numRadarObjects+i} = objectDetection(t,...
            [object.position(1); object.velocity(1); object.position(2); 0], ...
            'SensorIndex', 1, 'MeasurementNoise', visionMeasCov, ...
            'MeasurementParameters', {1}, ...
            'ObjectClassID', object.classification, ...
            'ObjectAttributes', {object.id, object.size});
    end
end
end
```

processRadar Converts reported radar objects to a list of `objectDetection` objects

```
function postProcessedDetections = processRadar(postProcessedDetections, realRadarObjects, t)
% Process the radar objects into objectDetection objects
numRadarObjects = numel(realRadarObjects);
if numRadarObjects
    classToUse = class(realRadarObjects(1).position);
    radarMeasCov = cast(diag([2,2,2,100]), classToUse);
    % Process Radar Objects:
    for i=1:numRadarObjects
        object = realRadarObjects(i);
        postProcessedDetections{i} = objectDetection(t, ...
            [object.position(1); object.velocity(1); object.position(2); object.velocity(2)], ..
            'SensorIndex', 2, 'MeasurementNoise', radarMeasCov, ...
            'MeasurementParameters', {2}, ...
            'ObjectAttributes', {object.id, object.status, object.amplitude, object.rangeMode});
    end
end
end
```

fcwmeas The measurement function used in this forward collision warning example

```
function measurement = fcwmeas(state, sensorID)
% The example measurements depend on the sensor type, which is reported by
% the MeasurementParameters property of the objectDetection. The following
% two sensorID values are used:
%   sensorID=1: video objects, the measurement is [x;vx;y].
%   sensorID=2: radar objects, the measurement is [x;vx;y;vy].
% The state is:
%   Constant velocity       state = [x;vx;y;vy]
%   Constant turn           state = [x;vx;y;vy;omega]
%   Constant acceleration   state = [x;vx;ax;y;vy;ay]

    if numel(state) < 6 % Constant turn or constant velocity
        switch sensorID
            case 1 % video
                measurement = [state(1:3); 0];
            case 2 % radar
                measurement = state(1:4);
        end
```

```
    else % Constant acceleration
        switch sensorID
            case 1 % video
                measurement = [state(1:2); state(4); 0];
            case 2 % radar
                measurement = [state(1:2); state(4:5)];
        end
    end
end
```

fcwmeasjac The Jacobian of the measurement function used in this forward collision warning example

```
function jacobian = fcwmeasjac(state, sensorID)
% The example measurements depend on the sensor type, which is reported by
% the MeasurementParameters property of the objectDetection. We choose
% sensorID=1 for video objects and sensorID=2 for radar objects.  The
% following two sensorID values are used:
%   sensorID=1: video objects, the measurement is [x;vx;y].
%   sensorID=2: radar objects, the measurement is [x;vx;y;vy].
% The state is:
%   Constant velocity       state = [x;vx;y;vy]
%   Constant turn           state = [x;vx;y;vy;omega]
%   Constant acceleration   state = [x;vx;ax;y;vy;ay]

    numStates = numel(state);
    jacobian = zeros(4, numStates);

    if numel(state) < 6 % Constant turn or constant velocity
        switch sensorID
            case 1 % video
                jacobian(1,1) = 1;
                jacobian(2,2) = 1;
                jacobian(3,3) = 1;
            case 2 % radar
                jacobian(1,1) = 1;
                jacobian(2,2) = 1;
                jacobian(3,3) = 1;
                jacobian(4,4) = 1;
        end
    else % Constant acceleration
        switch sensorID
            case 1 % video
                jacobian(1,1) = 1;
                jacobian(2,2) = 1;
                jacobian(3,4) = 1;
            case 2 % radar
                jacobian(1,1) = 1;
                jacobian(2,2) = 1;
                jacobian(3,4) = 1;
                jacobian(4,5) = 1;
        end
```

```
    end
end
```

See Also

Functions
`updateTracks`

Objects
`birdsEyePlot` | `monoCamera` | `multiObjectTracker` | `objectDetection` | `trackingEKF` | `trackingKF` | `trackingUKF`

More About

- “Code Generation for Tracking and Sensor Fusion” on page 7-117
- “Annotate Video Using Detections in Vehicle Coordinates” on page 7-9
- “Visualize Sensor Coverage, Detections, and Tracks” on page 7-226
- “Multiple Object Tracking Tutorial” on page 7-162
- “Multiple Object Tracking” (Computer Vision Toolbox)

Adaptive Cruise Control with Sensor Fusion

This example shows how to implement a sensor fusion-based automotive adaptive cruise controller for a vehicle traveling on a curved road using sensor fusion.

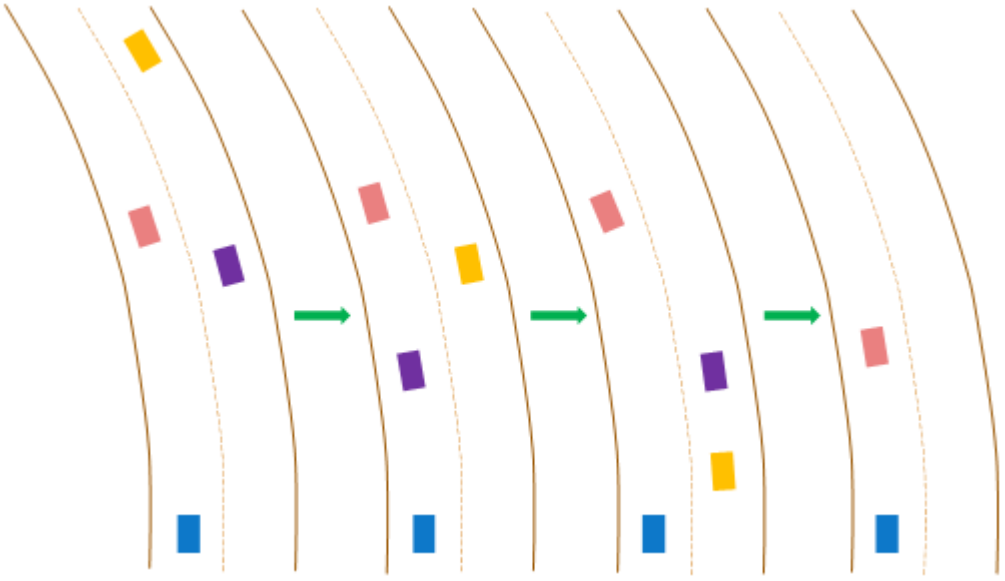
In this example, you:

1. Review a control system that combines sensor fusion and an adaptive cruise controller (ACC). Two variants of ACC are provided: a classical controller and an Adaptive Cruise Control System block from Model Predictive Control Toolbox.
2. Test the control system in a closed-loop Simulink model using synthetic data generated by the Automated Driving Toolbox.
3. Configure the code generation settings for software-in-the-loop simulation, and automatically generate code for the control algorithm.

Introduction

An adaptive cruise control system is a control system that modifies the speed of the ego vehicle in response to conditions on the road. As in regular cruise control, the driver sets a desired speed for the car; in addition, the adaptive cruise control system can slow the ego vehicle down if there is another vehicle moving slower in the lane in front of it.

For the ACC to work correctly, the ego vehicle must determine how the lane in front of it curves, and which car is the 'lead car', that is, in front of the ego vehicle in the lane. A typical scenario from the viewpoint of the ego vehicle is shown in the following figure. The ego vehicle (blue) travels along a curved road. At the beginning, the lead car is the pink car. Then the purple car cuts into the lane of the ego vehicle and becomes the lead car. After a while, the purple car changes to another lane, and the pink car becomes the lead car again. The pink car remains the lead car afterward. The ACC design must react to the change in the lead car on the road.



Current ACC designs rely mostly on range and range rate measurements obtained from radar, and are designed to work best along straight roads. An example of such a system is given in “Adaptive Cruise Control System Using Model Predictive Control” (Model Predictive Control Toolbox) and in

“Automotive Adaptive Cruise Control Using FMCW Technology” (Phased Array System Toolbox). Moving from advanced driver-assistance system (ADAS) designs to more autonomous systems, the ACC must address the following challenges:

1. Estimating the relative positions and velocities of the cars that are near the ego vehicle and that have significant lateral motion relative to the ego vehicle.
2. Estimating the lane ahead of the ego vehicle to find which car in front of the ego vehicle is the closest one in the same lane.
3. Reacting to aggressive maneuvers by other vehicles in the environment, in particular, when another vehicle cuts into the ego vehicle lane.

This example demonstrates two main additions to existing ACC designs that meet these challenges: adding a sensor fusion system and updating the controller design based on model predictive control (MPC). A sensor fusion and tracking system that uses both vision and radar sensors provides the following benefits:

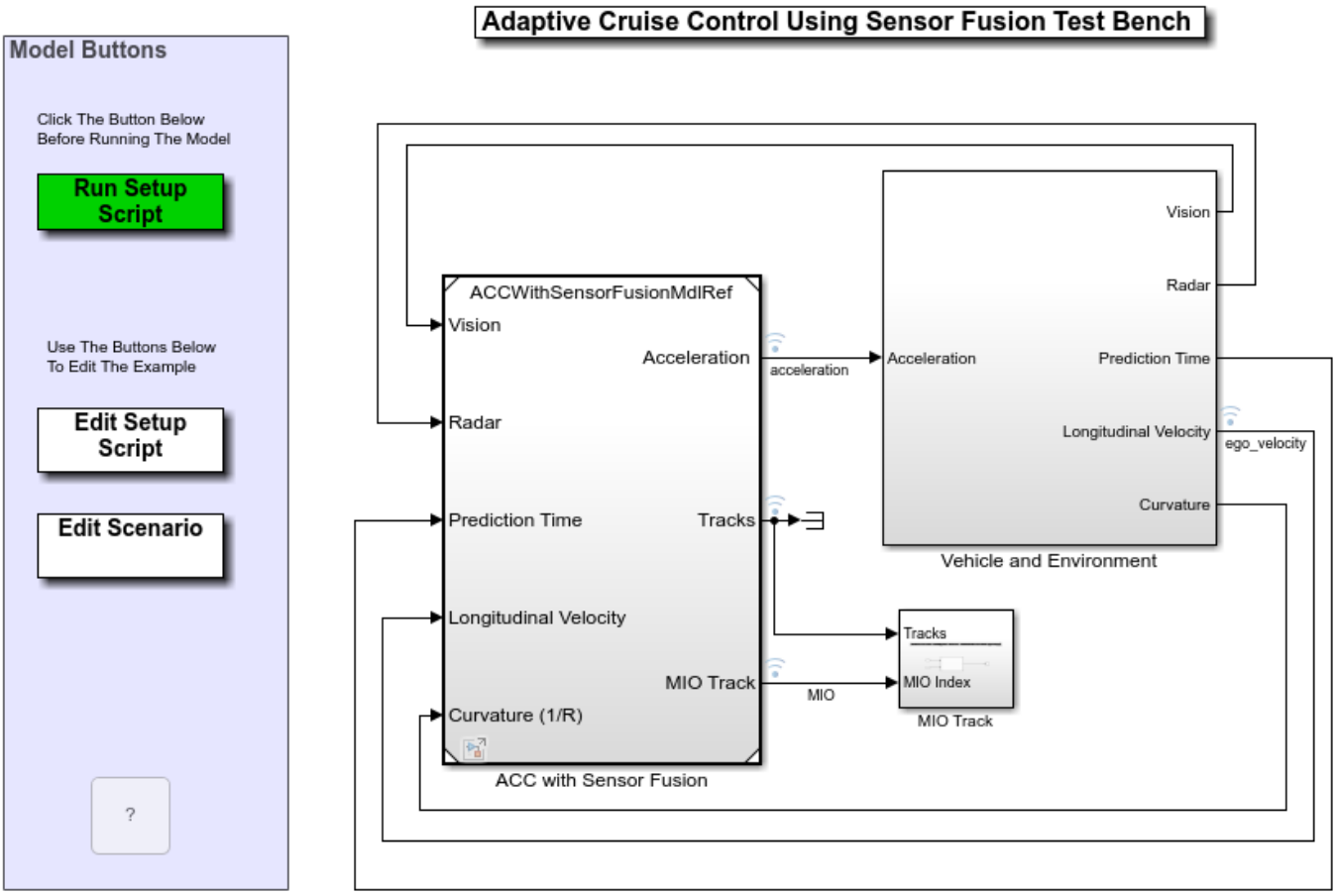
1. It combines the better lateral measurement of position and velocity obtained from vision sensors with the range and range rate measurement from radar sensors.
2. A vision sensor can detect lanes, provide an estimate of the lateral position of the lane relative to the ego vehicle, and position the other cars in the scene relative to the ego vehicle lane. This example assumes ideal lane detection.

An advanced MPC controller adds the ability to react to more aggressive maneuvers by other vehicles in the environment. In contrast to a classical controller that uses a PID design with constant gains, the MPC controller regulates the velocity of the ego vehicle while maintaining a strict safe distance constraint. Therefore, the controller can apply more aggressive maneuvers when the environment changes quickly in a similar way to what a human driver would do.

Overview of Test Bench Model and Simulation Results

To open the main Simulink model, use the following command:

```
open_system('ACCTestBenchExample')
```



The model contains two main subsystems:

1. ACC with Sensor Fusion, which models the sensor fusion and controls the longitudinal acceleration of the vehicle. This component allows you to select either a classical or model predictive control version of the design.
2. A Vehicle and Environment subsystem, which models the motion of the ego vehicle and models the environment. A simulation of radar and vision sensors provides synthetic data to the control subsystem.

To run the associated initialization script before running the model, in the Simulink model, click **Run Setup Script** or, at the command prompt, type the following:

```
helperACCSetUp
```

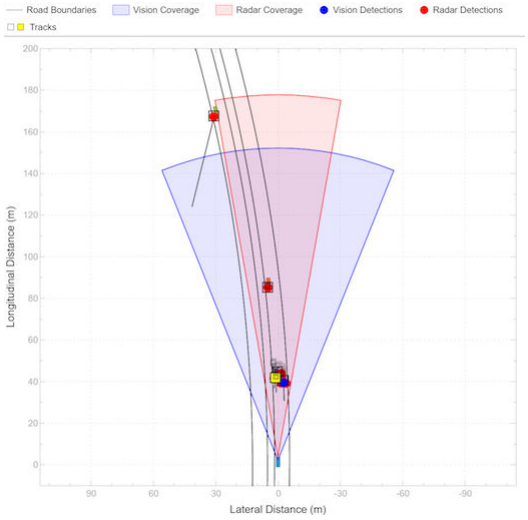
The script loads certain constants needed by the Simulink model, such as the vehicle and ACC design parameters. The default ACC is the classical controller. The script also creates buses that are required for defining the inputs into and outputs for the control system referenced model. These buses must be defined in the workspace before model compilation. When the model compiles, additional Simulink buses are automatically generated by their respective blocks.

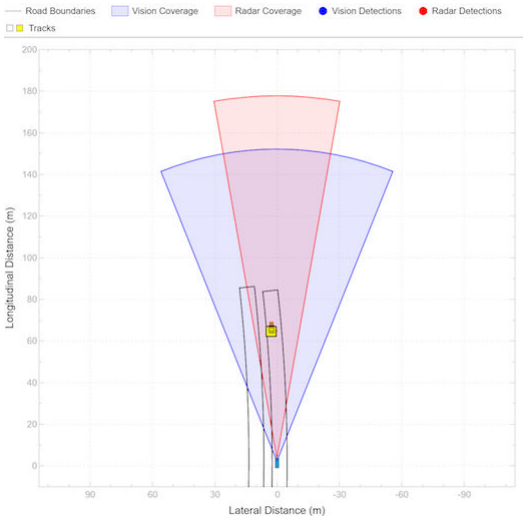
To plot the results of the simulation and depict the surroundings of the ego vehicle, including the tracked objects, use the Bird's-Eye Scope. The Bird's-Eye Scope is a model-level visualization tool that you can open from the Simulink toolstrip. On the **Simulation** tab, under **Review Results**, click

Bird's-Eye Scope. After opening the scope, click **Find Signals** to set up the signals. The following commands run the simulation to 15 seconds to get a mid-simulation picture and run again all the way to end of the simulation to gather results.

```
sim('ACCTestBenchExample','StopTime','15') %Simulate 15 seconds
sim('ACCTestBenchExample') %Simulate to end of scenario
```

```
ans =

  Simulink.SimulationOutput:
                logsout: [1x1 Simulink.SimulationData.Dataset]
                   tout: [151x1 double]

     SimulationMetadata: [1x1 Simulink.SimulationMetadata]
           ErrorMessage: [0x0 char]
```





The Bird's-Eye Scope shows the results of the sensor fusion. It shows how the radar and vision sensors detect the vehicles within their sensors coverage areas. It also shows the tracks maintained

by the Multi Object Tracker block. The yellow track shows the most important object (MIO): the closest track in front of the ego vehicle in its lane. We see that at the beginning of the scenario, the most important object is the fast-moving car ahead of the ego vehicle. When the passing car gets closer to the slow-moving car, it crosses to the left lane, and the sensor fusion system recognizes it to be the MIO. This car is much closer to the ego vehicle and much slower than it. Thus, the ACC must slow the ego vehicle.

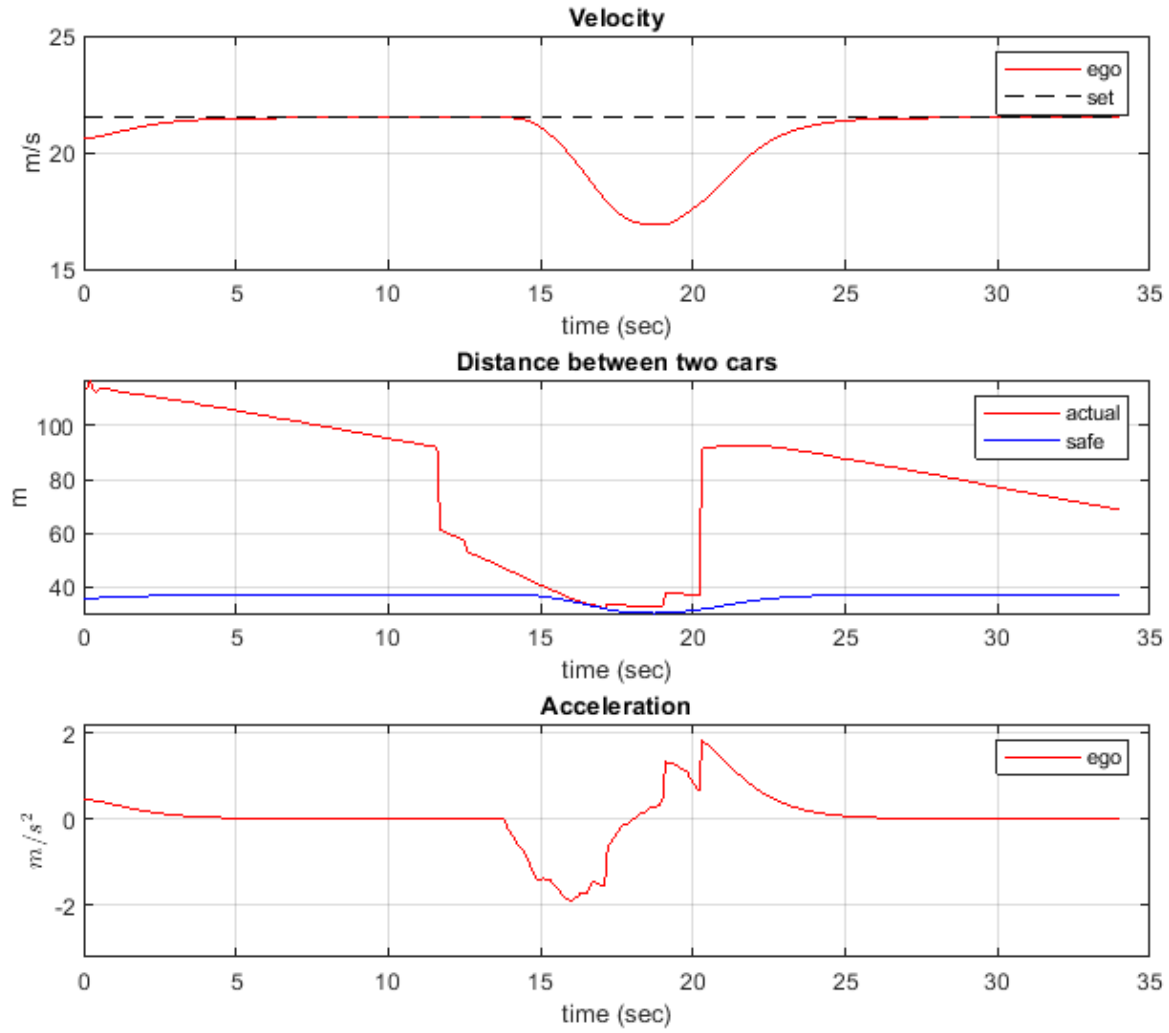
In the following results for the classical ACC system, the:

- Top plot shows the ego vehicle velocity.
- Middle plot shows the relative distance between the ego vehicle and lead car.
- Bottom plot shows the ego vehicle acceleration.

In this example, the raw data from the Tracking and Sensor Fusion system is used for ACC design without post-processing. You can expect to see some 'spikes' (middle plot) due to the uncertainties in the sensor model especially when another car cuts into or leaves the ego vehicle lane.

To view the simulation results, use the following command.

```
helperPlotACCResults(logsout,default_spacing,time_gap)
```

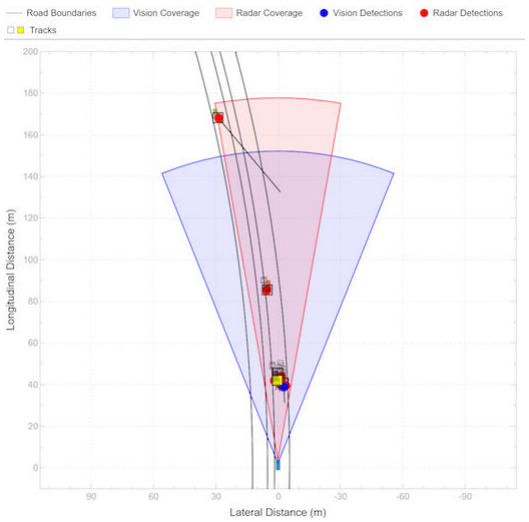


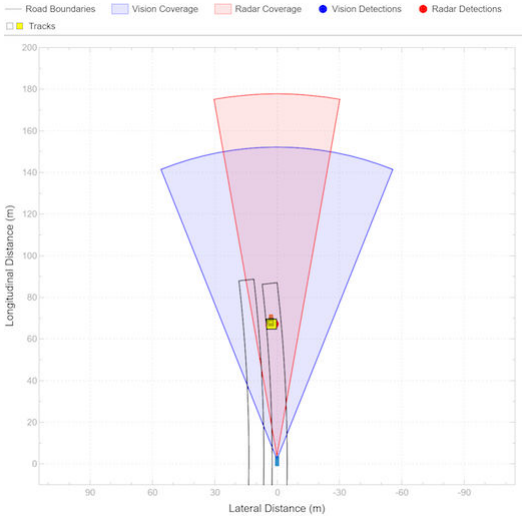
- In the first 11 seconds, the lead car is far ahead of the ego vehicle (middle plot). The ego vehicle accelerates and reaches the driver-set velocity V_set (top plot).
- Another car becomes the lead car from 11 to 20 seconds when the car cuts into the ego vehicle lane (middle plot). When the distance between the lead car and the ego vehicle is large (11-15 seconds), the ego vehicle still travels at the driver-set velocity. When the distance becomes small (15-20 seconds), the ego vehicle decelerates to maintain a safe distance from the lead car (top plot).
- From 20 to 34 seconds, the car in front moves to another lane, and a new lead car appears (middle plot). Because the distance between the lead car and the ego vehicle is large, the ego vehicle accelerates until it reaches the driver-set velocity at 27 seconds. Then, the ego vehicle continues to travel at the driver-set velocity (top plot).
- The bottom plot demonstrates that the acceleration is within the range [-3,2] m/s^2. The smooth transient behavior indicates that the driver comfort is satisfactory.

In the MPC-based ACC design, the underlying optimization problem is formulated by tracking the driver-set velocity subject to enforcing a safe distance from the lead car. The MPC controller design is described in the Adaptive Cruise Controller section. To run the model with the MPC design, first activate the MPC variant, and then run the following commands. This step requires Model Predictive Control Toolbox software. You can check the existence of this license using the following code. If no code exists, a sample of similar results is depicted.

```
hasMPCLicense = license('checkout','MPC_Toolbox');
if hasMPCLicense
    controller_type = 2;
    sim('ACCTestBenchExample','StopTime','15') %Simulate 15 seconds
    sim('ACCTestBenchExample') %Simulate to end of scenario
else
    load data_mpc
end
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #2 is integrated white noise.
   Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured

ans =

  Simulink.SimulationOutput:
                logsout: [1x1 Simulink.SimulationData.Dataset]
                   tout: [151x1 double]

     SimulationMetadata: [1x1 Simulink.SimulationMetadata]
           ErrorMessage: [0x0 char]

-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #2 is integrated white noise.
   Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

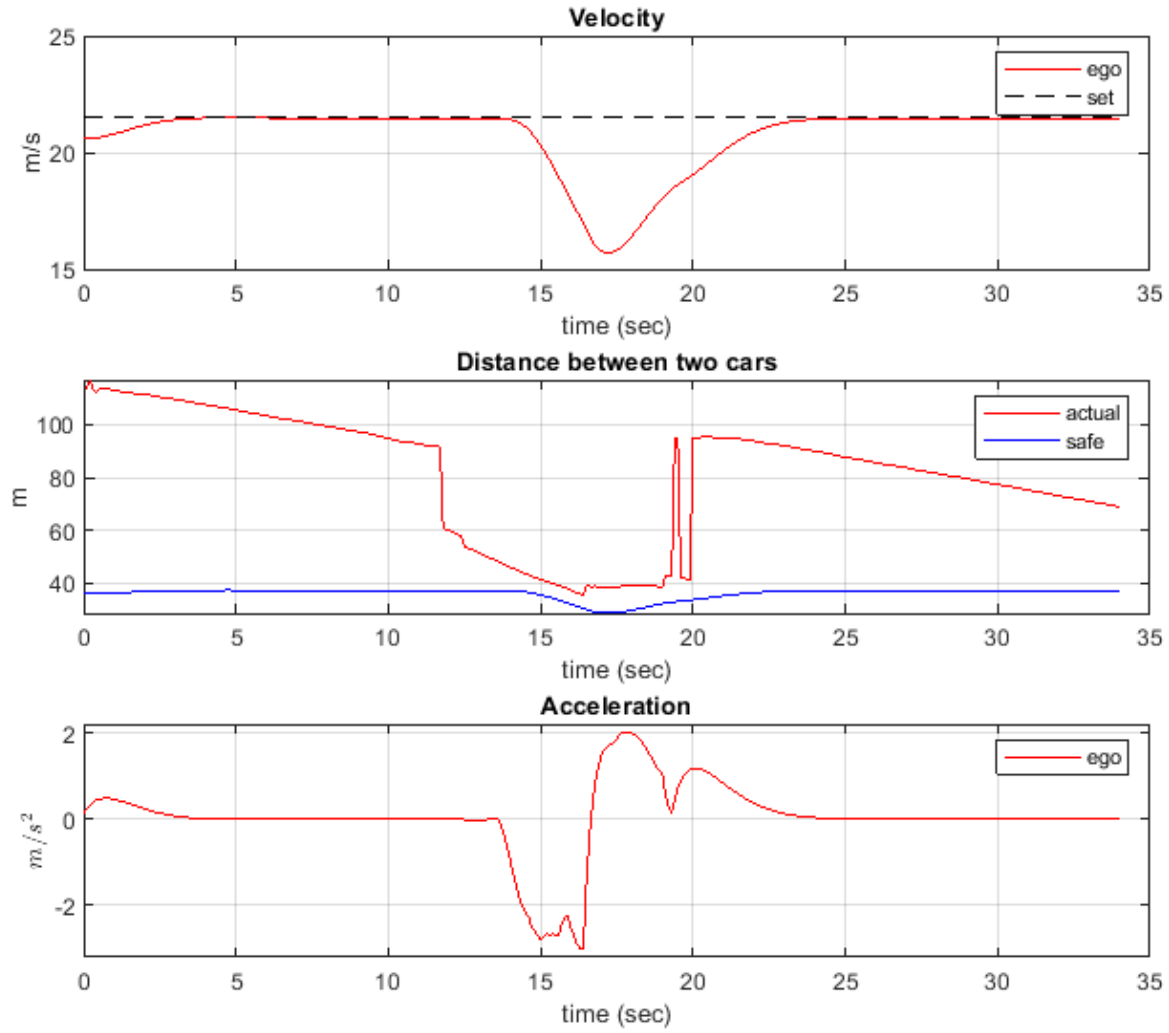




In the simulation results for the MPC-based ACC, similar to the classical ACC design, the objectives of speed and spacing control are achieved. Compared to the classical ACC design, the MPC-based ACC is more aggressive as it uses full throttle or braking for acceleration or deceleration. This behavior is due to the explicit constraint on the relative distance. The aggressive behavior may be preferred when sudden changes on the road occur, such as when the lead car changes to be a slow car. To make the controller less aggressive, open the mask of the Adaptive Cruise Control System block, and reduce the value of the **Controller Behavior** parameter. As previously noted, the spikes in the middle plot are due to the uncertainties in the sensor model.

To view the results of the simulation with the MPC-based ACC, use the following command.

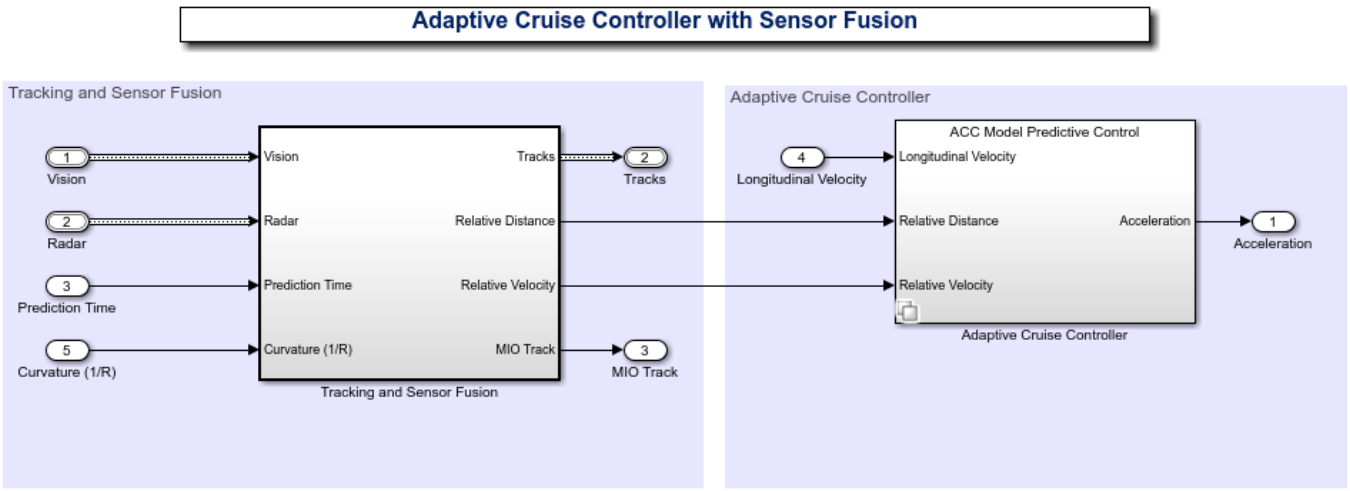
```
helperPlotACCResults(logsout,default_spacing,time_gap)
```



In the following, the functions of each subsystem in the Test Bench Model are described in more detail. The Adaptive Cruise Controller with Sensor Fusion subsystem contains two main components:

1. Tracking and Sensor Fusion subsystem
2. Adaptive Cruise Controller subsystem

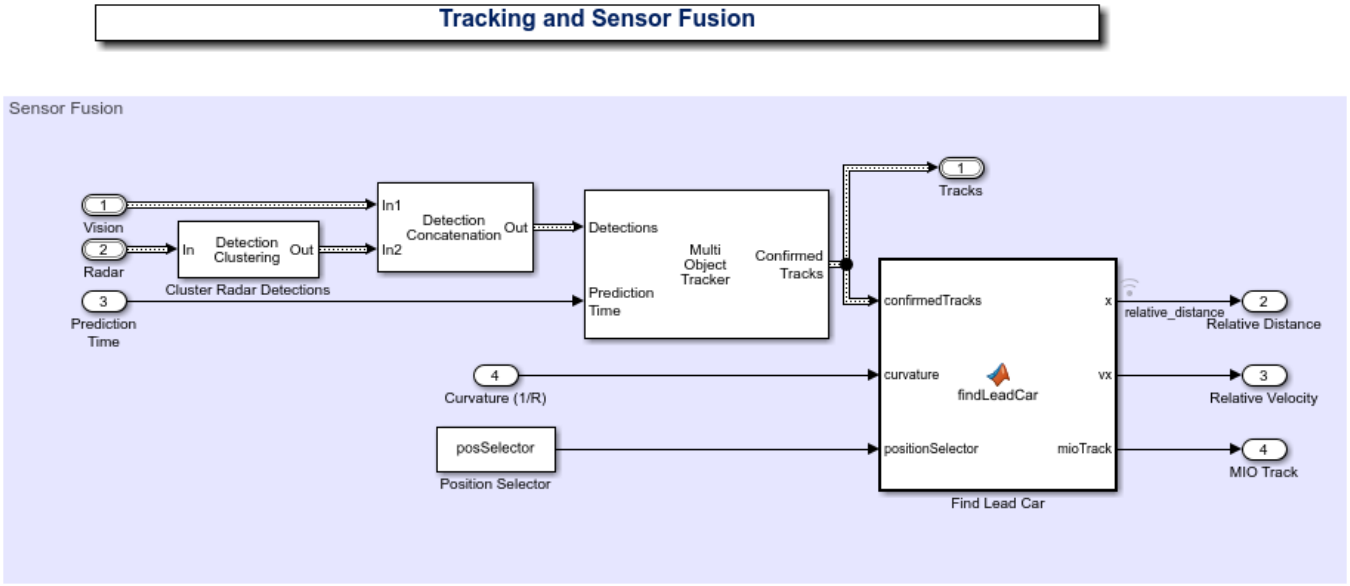
```
open_system('ACCTestBenchExample/ACC with Sensor Fusion')
```



Tracking and Sensor Fusion

The Tracking and Sensor Fusion subsystem processes vision and radar detections coming from the Vehicle and Environment subsystem and generates a comprehensive situation picture of the environment around the ego vehicle. Also, it provides the ACC with an estimate of the closest car in the lane in front of the ego vehicle.

```
open_system('ACCWithSensorFusionMdlRef/Tracking and Sensor Fusion')
```



The main block of the Tracking and Sensor Fusion subsystem is the Multi-Object Tracker block, whose inputs are the combined list of all the sensor detections and the prediction time. The output from the Multi Object Tracker block is a list of confirmed tracks.

The Detection Concatenation block concatenates the vision and radar detections. The prediction time is driven by a clock in the Vehicle and Environment subsystem.

The Detection Clustering block clusters multiple radar detections, since the tracker expects at most one detection per object per sensor.

The `findLeadCar` MATLAB function block finds which car is closest to the ego vehicle and ahead of it in same the lane using the list of confirmed tracks and the curvature of the road. This car is referred to as the lead car, and may change when cars move into and out of the lane in front of the ego vehicle. The function provides the position and velocity of the lead car relative to the ego vehicle and an index to the most important object (MIO) track.

Adaptive Cruise Controller

The adaptive cruise controller has two variants: a classical design (default) and an MPC-based design. For both designs, the following design principles are applied. An ACC equipped vehicle (ego vehicle) uses sensor fusion to estimate the relative distance and relative velocity to the lead car. The ACC makes the ego vehicle travel at a driver-set velocity while maintaining a safe distance from the lead car. The safe distance between lead car and ego vehicle is defined as

$$D_{safe} = D_{default} + T_{gap} \cdot V_x$$

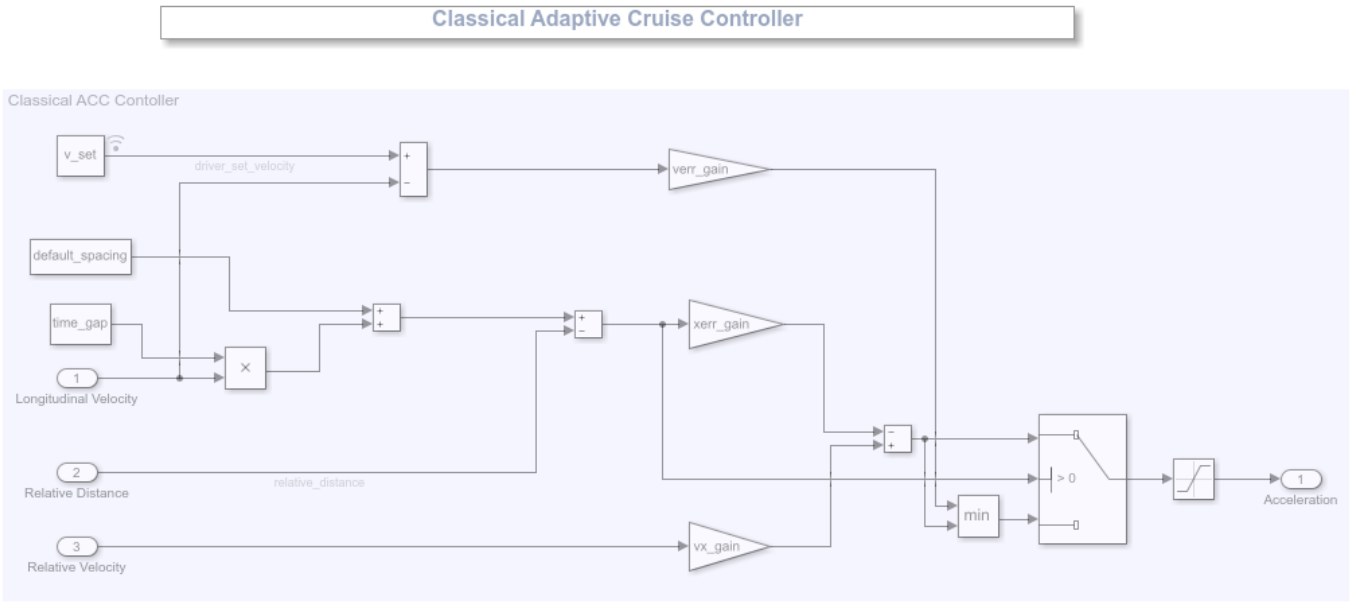
where the default spacing $D_{default}$, and time gap T_{gap} are design parameters and V_x is the longitudinal velocity of the ego vehicle. The ACC generates the longitudinal acceleration for the ego vehicle based on the following inputs:

- Longitudinal velocity of ego vehicle
- Relative distance between lead car and ego vehicle (from the Tracking and Sensor Fusion system)
- Relative velocity between lead car and ego vehicle (from the Tracking and Sensor Fusion system)

Considering the physical limitations of the ego vehicle, the longitudinal acceleration is constrained to the range [-3,2] m/s^2.

In the classical ACC design, if the relative distance is less than the safe distance, then the primary goal is to slow down and maintain a safe distance. If the relative distance is greater than the safe distance, then the primary goal is to reach driver-set velocity while maintaining a safe distance. These design principles are achieved through the Min and Switch blocks.

```
open_system('ACCWithSensorFusionMdlRef/Adaptive Cruise Controller/ACC Classical')
```

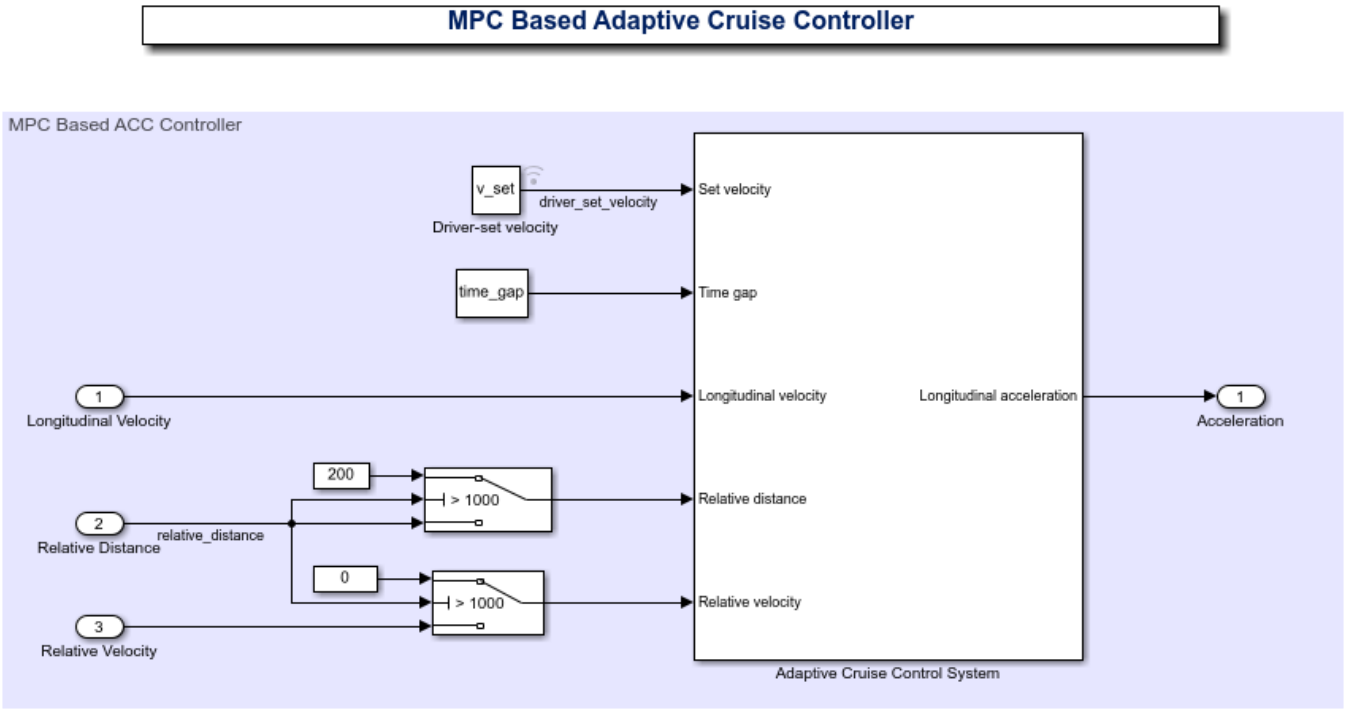


In the MPC-based ACC design, the underlying optimization problem is formulated by tracking the driver-set velocity subject to a constraint. The constraint enforces that relative distance is always greater than the safe distance.

$$\begin{aligned} \underset{u}{\text{minimize}} \quad & |V - V_{set}|^2 \\ \text{subject to} \quad & D_{relative} - D_{safe} \geq 0 \\ & -3 \leq u \leq 2 \end{aligned}$$

To configure the Adaptive Cruise Control System block, use the parameters defined in the `helperACCSetUp` file. For example, the linear model for ACC design G, and G is obtained from vehicle dynamics. The two Switch blocks implement simple logic to handle large numbers from the sensor (for example, the sensor may return `Inf` when it does not detect an MIO).

```
open_system('ACCWithSensorFusionMdlRef/Adaptive Cruise Controller/ACC Model Predictive Control')
```



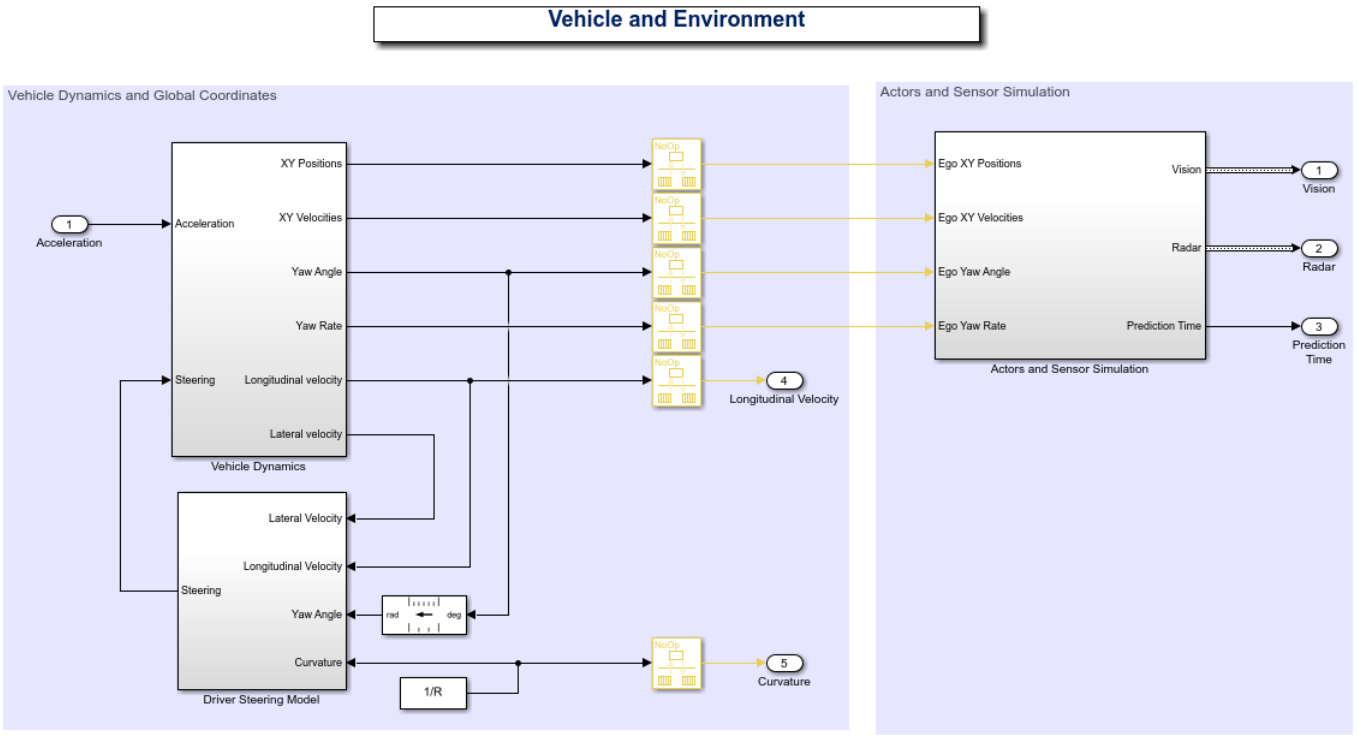
For more information on MPC design for ACC, see “Adaptive Cruise Control System Using Model Predictive Control” (Model Predictive Control Toolbox).

Vehicle and Environment

The Vehicle and Environment subsystem is composed of two parts:

1. Vehicle Dynamics and Global Coordinates
2. Actor and Sensor Simulation

```
open_system('ACCTestBenchExample/Vehicle and Environment')
```



The Vehicle Dynamics subsystem models the vehicle dynamics with the Bicycle Model - Force Input block from the Automated Driving Toolbox. The vehicle dynamics, with input u (longitudinal acceleration) and front steering angle δ, are approximated by:

$$\frac{d}{dt}\begin{bmatrix} V_y \\ \psi \\ \dot{\psi} \\ Vx \\ \dot{V}_x \end{bmatrix} = \begin{bmatrix} -\frac{2C_f+2C_r}{mV_x} & 0 & -V_x - \frac{2C_f\ell_f-2C_r\ell_r}{mV_x} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ -\frac{2C_f\ell_f-2C_r\ell_r}{I_z V_x} & 0 & -\frac{2C_f\ell_f^2+2C_r\ell_r^2}{I_z V_x} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -\frac{1}{\tau} \end{bmatrix} \begin{bmatrix} V_y \\ \psi \\ \dot{\psi} \\ Vx \\ \dot{V}_x \end{bmatrix} + \begin{bmatrix} \frac{2C_f}{m} \\ 0 \\ \frac{2C_f\ell_f}{I_z} \\ 0 \\ 0 \end{bmatrix} \delta + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{\tau} \end{bmatrix} u + \begin{bmatrix} 0 \\ 0 \\ 0 \\ V_y\dot{\psi} \\ 0 \end{bmatrix}$$

In the state vector, V_y denotes the lateral velocity, V_x denotes the longitudinal velocity and ψ denotes the yaw angle. The vehicle parameters are provided in the `helperACCSetUp` file.

The outputs from the vehicle dynamics (such as longitudinal velocity V_x and lateral velocity V_y) are based on body fixed coordinates. To obtain the trajectory traversed by the vehicle, the body fixed coordinates are converted into global coordinates through the following relations:

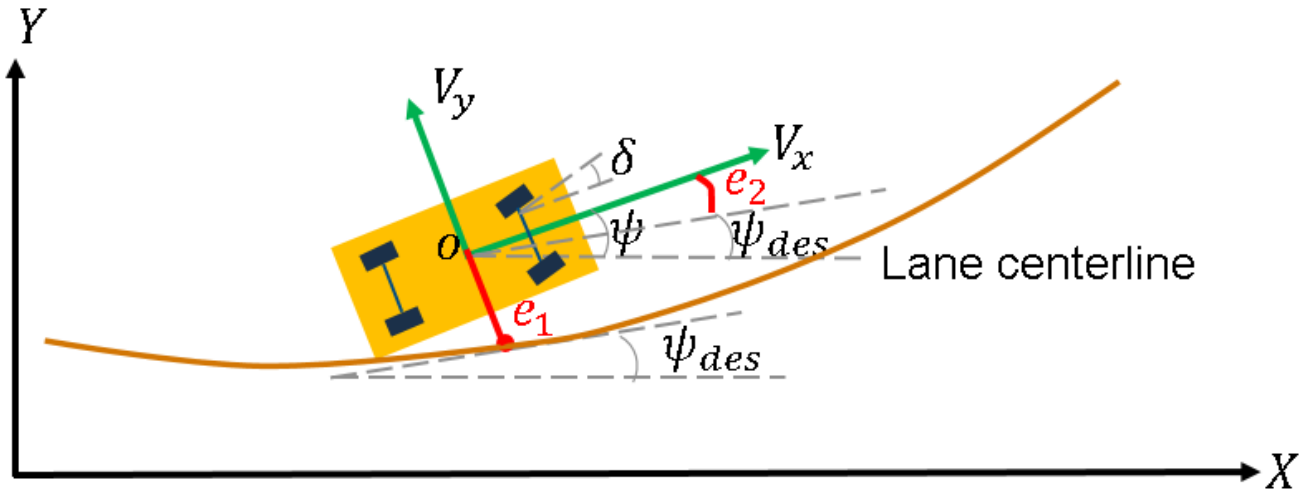
$$\dot{X} = V_x\cos(\psi) - V_y\sin(\psi), \quad \dot{Y} = V_x\sin(\psi) + V_y\cos(\psi)$$

The yaw angle ψ and yaw angle rate $\dot{\psi}$ are also converted into the units of degrees.

The goal for the driver steering model is to keep the vehicle in its lane and follow the curved road by controlling the front steering angle δ. This goal is achieved by driving the yaw angle error e_2 and lateral displacement error e_1 to zero (see the following figure), where

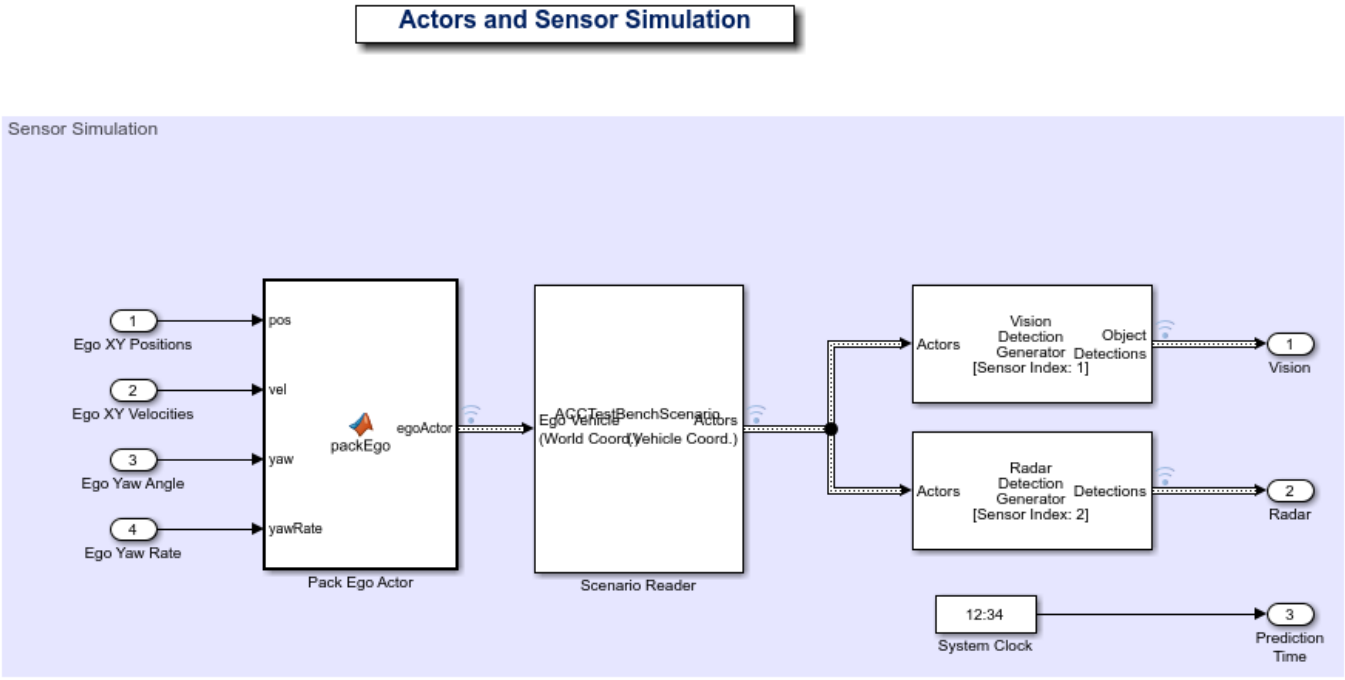
$$\dot{e}_1 = V_x e_2 + V_y, \quad e_2 = \psi - \psi_{des}$$

The desired yaw angle rate is given by Vx/R (R denotes the radius for the road curvature).



The Actors and Sensor Simulation subsystem generates the synthetic sensor data required for tracking and sensor fusion. Before running this example, the Driving Scenario Designer app was used to create a scenario with a curved road and multiple actors moving on the road. The roads and actors from this scenario were then saved to the scenario file `ACCTestBenchScenario.mat`. To see how you can define the scenario, see the Scenario Creation section.

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



The motion of the ego vehicle is controlled by the control system and is not read from the scenario file. Instead, the ego vehicle position, velocity, yaw angle, and yaw rate are received as inputs from the Vehicle Dynamics block and are packed into a single actor pose structure using the `packEgo` MATLAB function block.

The Scenario Reader block reads the actor pose data from the scenario file `ACCTestBenchScenario.mat`. The block converts the actor poses from the world coordinates of the scenario into ego vehicle coordinates. The actor poses are streamed on a bus generated by the block. In this example, you use a Vision Detection Generator block and Radar Detection Generator block. Both sensors are long-range and forward-looking, and provide good coverage of the front of the ego vehicle, as needed for ACC. The sensors use the actor poses in ego vehicle coordinates to generate lists of vehicle detections in front of the ego vehicle. Finally, a clock block is used as an example of how the vehicle would have a centralized time source. The time is used by the Multi Object Tracker block.

Scenario Creation

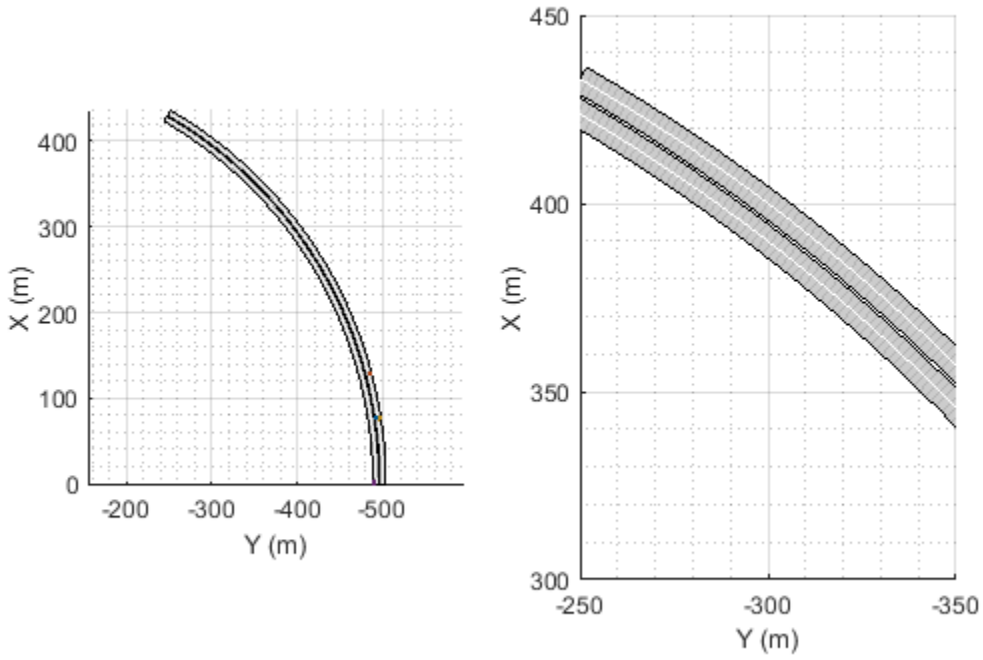
The **Driving Scenario Designer** app allows you to define roads and vehicles moving on the roads. For this example, you define two parallel roads of constant curvature. To define the road, you define the road centers, the road width, and banking angle (if needed). The road centers were chosen by sampling points along a circular arc, spanning a turn of 60 degrees of constant radius of curvature.

You define all the vehicles in the scenario. To define the motion of the vehicles, you define their trajectory by a set of waypoints and speeds. A quick way to define the waypoints is by choosing a subset of the road centers defined earlier, with an offset to the left or right of the road centers to control the lane in which the vehicles travel.

This example shows four vehicles: a fast-moving car in the left lane, a slow-moving car in the right lane, a car approaching on the opposite side of the road, and a car that starts on the right lane, but then moves to the left lane to pass the slow-moving car.

The scenario can be modified using the **Driving Scenario Designer** app and resaved to the same scenario file `ACCTestBenchScenario.mat`. The Scenario Reader block automatically picks up the changes when simulation is rerun. To build the scenario programmatically, you can use the `helperScenarioAuthoring` function.

```
plotACCScenario
```



Generating Code for the Control Algorithm

Although the entire model does not support code generation, the `ACCWithSensorFusionMdlRef` model is configured to support generating C code using Embedded Coder software. To check if you have access to Embedded Coder, run:

```
hasEmbeddedCoderLicense = license('checkout','RTW_Embedded_Coder')
```

You can generate a C function for the model and explore the code generation report by running:

```
if hasEmbeddedCoderLicense
    rtwbuild('ACCWithSensorFusionMdlRef')
end
```

You can verify that the compiled C code behaves as expected using software-in-the-loop (SIL) simulation. To simulate the `ACCWithSensorFusionMdlRef` referenced model in SIL mode, use:

```
if hasEmbeddedCoderLicense
    set_param('ACCTestBenchExample/ACC with Sensor Fusion',...
        'SimulationMode','Software-in-the-loop (SIL)')
end
```

When you run the `ACCTestBenchExample` model, code is generated, compiled, and executed for the `ACCWithSensorFusionMdlRef` model. This enables you to test the behavior of the compiled code through simulation.

Conclusions

This example shows how to implement an integrated adaptive cruise controller (ACC) on a curved road with sensor fusion, test it in Simulink using synthetic data generated by the Automated Driving Toolbox, componentize it, and automatically generate code for it.

```
bdclose all
```

See Also

Functions

`record` | `roadBoundaries`

Blocks

Adaptive Cruise Control System | Detection Concatenation | Multi-Object Tracker | Radar Detection Generator | Vision Detection Generator

Objects

`drivingScenario`

More About

- “Adaptive Cruise Control System Using Model Predictive Control” (Model Predictive Control Toolbox)
- “Automotive Adaptive Cruise Control Using FMCW Technology” (Phased Array System Toolbox)

Forward Collision Warning Application with CAN FD and TCP/IP

This example shows you how to execute a forward collision warning (FCW) application with sensor and vision data replayed live via CAN FD and TCP/IP protocols. Recorded data from a sensor suite mounted on a test vehicle are replayed live as if they were coming through the network interfaces of the vehicle. The Vehicle Network Toolbox™ and Instrument Control Toolbox™ provide these interfaces. This setup is used to test an FCW system developed using features from the Automated Driving Toolbox™. For assistance with the design and development of actual FCW algorithms, refer to the Forward Collision Warning Using Sensor Fusion example.

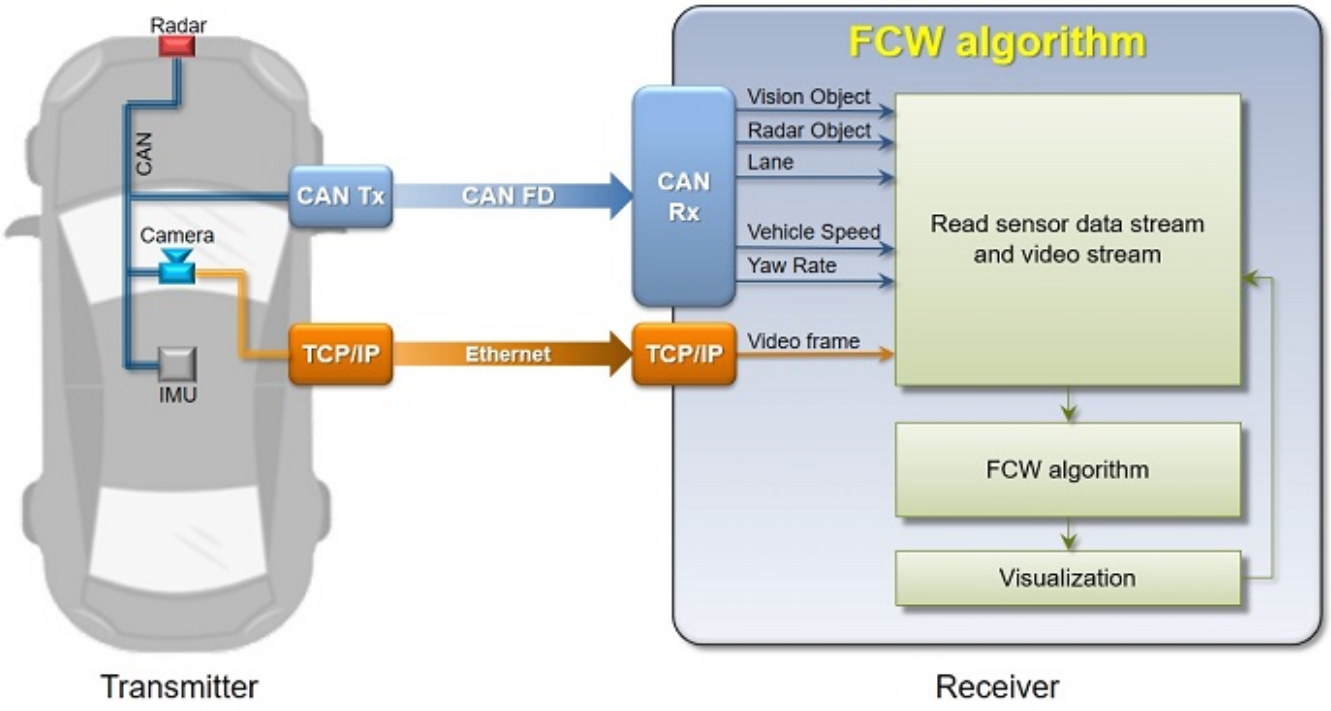
System Configuration

This example uses virtual CAN FD channels from Vector. These virtual device channels are available with the installation of the Vector Driver Setup package from www.vector.com.

This example has two primary components:

1. Transmitter: Sends the sensor and vision data via CAN FD and TCP/IP. This portion represents a sample vehicle environment. It replays prerecorded data as if it were a live vehicle.
2. Receiver: Collects all the data and executes the FCW algorithm and visualizations. This portion represents the application component.

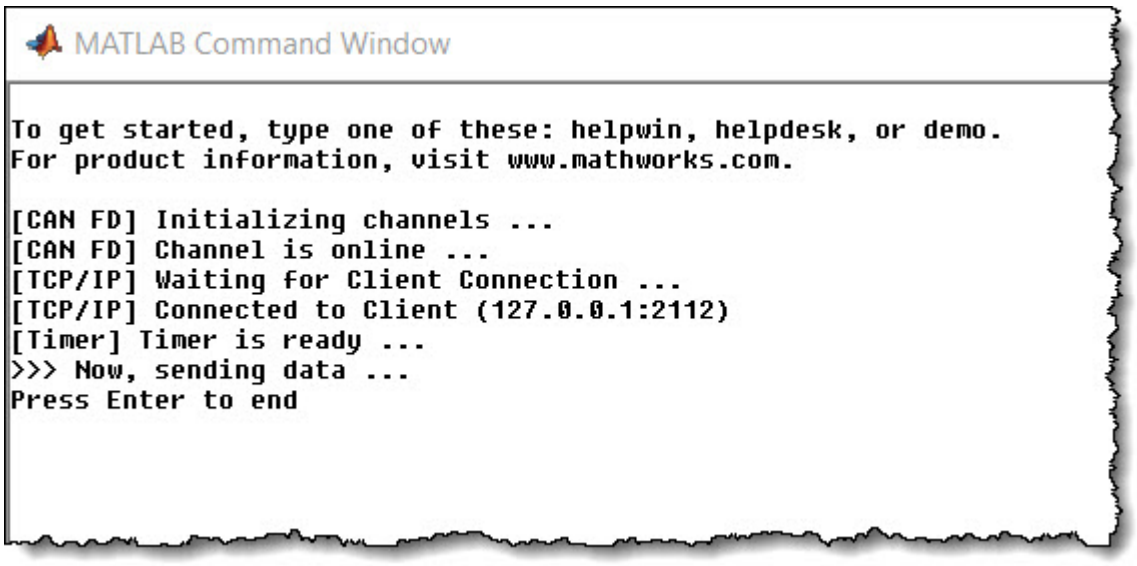
To execute the example, the transmitter and receiver portions run from separate sessions of MATLAB. This replicates the data source existing outside the MATLAB session serving as the development tool. Furthermore, this example allows you to run the FCW application in multiple execution modes (interpreted and MEX) with different performance characteristics.



Generate Data

The transmitting application executes via the `helperStartTransmitter` function. It launches a separate MATLAB process to run outside of the current MATLAB session. The transmitter initializes itself and begins sending sensor and vision data automatically. To run the transmitter, use the `system` command.

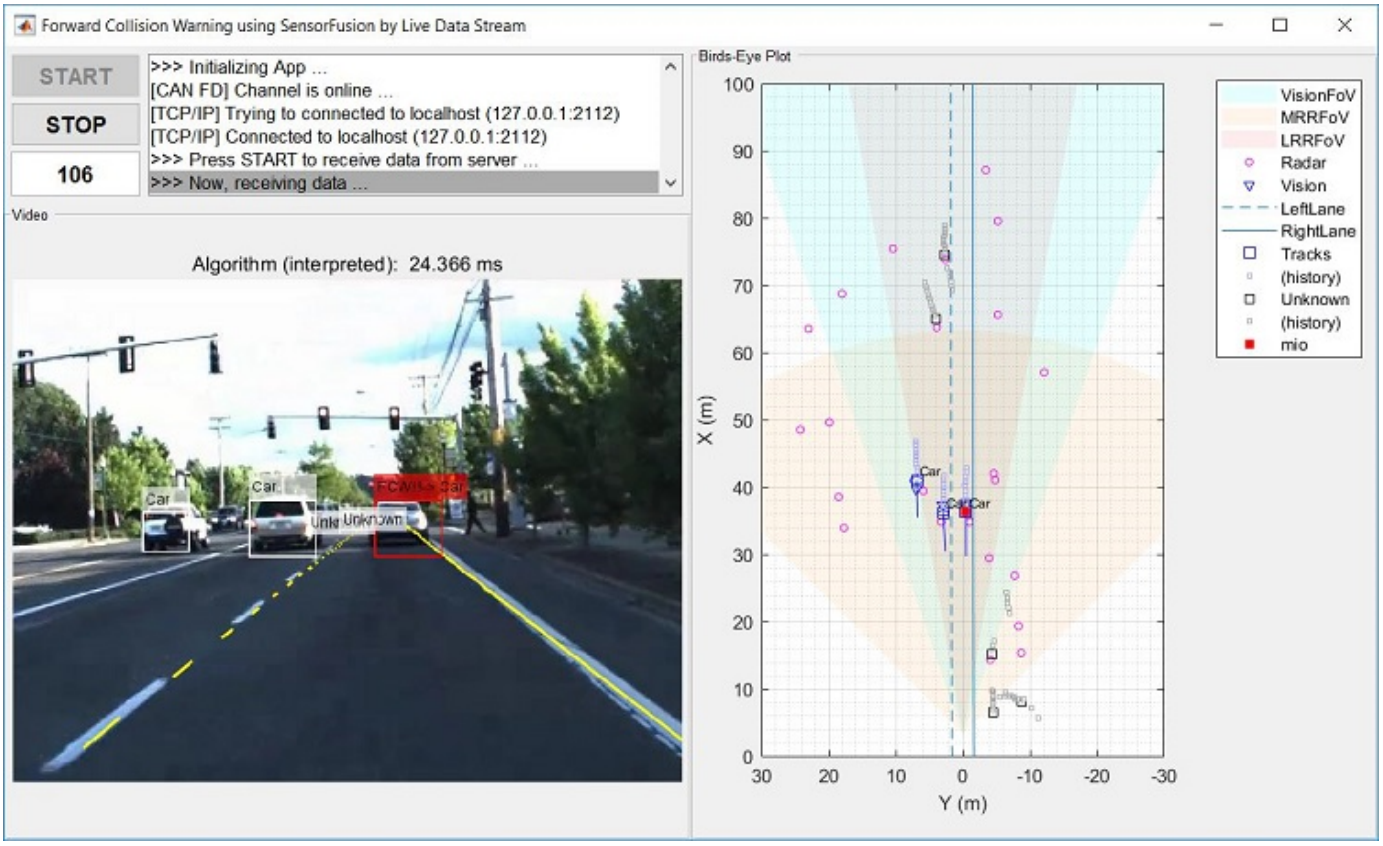
```
system('matlab -nodesktop -nosplash -r helperStartTransmitter &')
```



Execute Forward Collision Warning System (Interpreted Mode)

To open the receiving FCW application, execute the `helperStartReceiver` function. You can click **START** to begin data reception, processing, and visualization. You can explore the `helperStartReceiver` function to see how the Vehicle Network Toolbox CAN FD functions, Instrument Control Toolbox TCP/IP functions, and Automated Driving Toolbox capabilities are used in concert with one another.

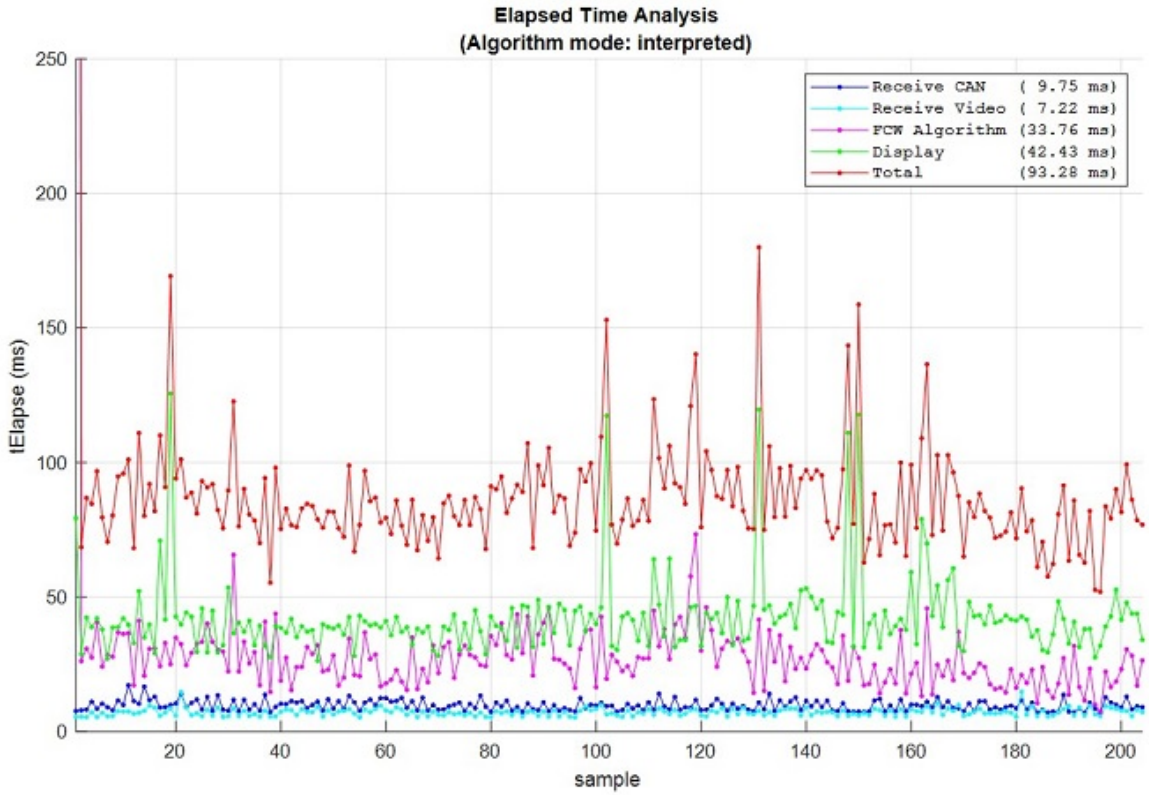
```
helperStartReceiver('interpreted')
```



Review Results

When ready, stop the transmitter application using the close window button on its command window. Click **STOP** on the receiving FCW application, and then close its window as well.

When the receiving FCW application is stopped, a plot appears detailing performance characteristics of the application. It shows time spent receiving data, processing the FCW algorithm, and performing visualizations. Benchmarking is useful to show parts of the setup that need performance improvement. It is clear that a significant portion of time is spent executing the FCW algorithm. In the next section, explore code generation as a strategy to improve performance.



Execute Forward Collision Warning System (MEX Mode)

If faster performance is a requirement in your workflow, you can use MATLAB Coder™ to generate and compile MATLAB code as MEX code. To build this example as MEX code, use the `helperGenerateCode` function. The build will compile the FCW application into a MEX function directly callable within MATLAB.

```
helperGenerateCode('mex')
```

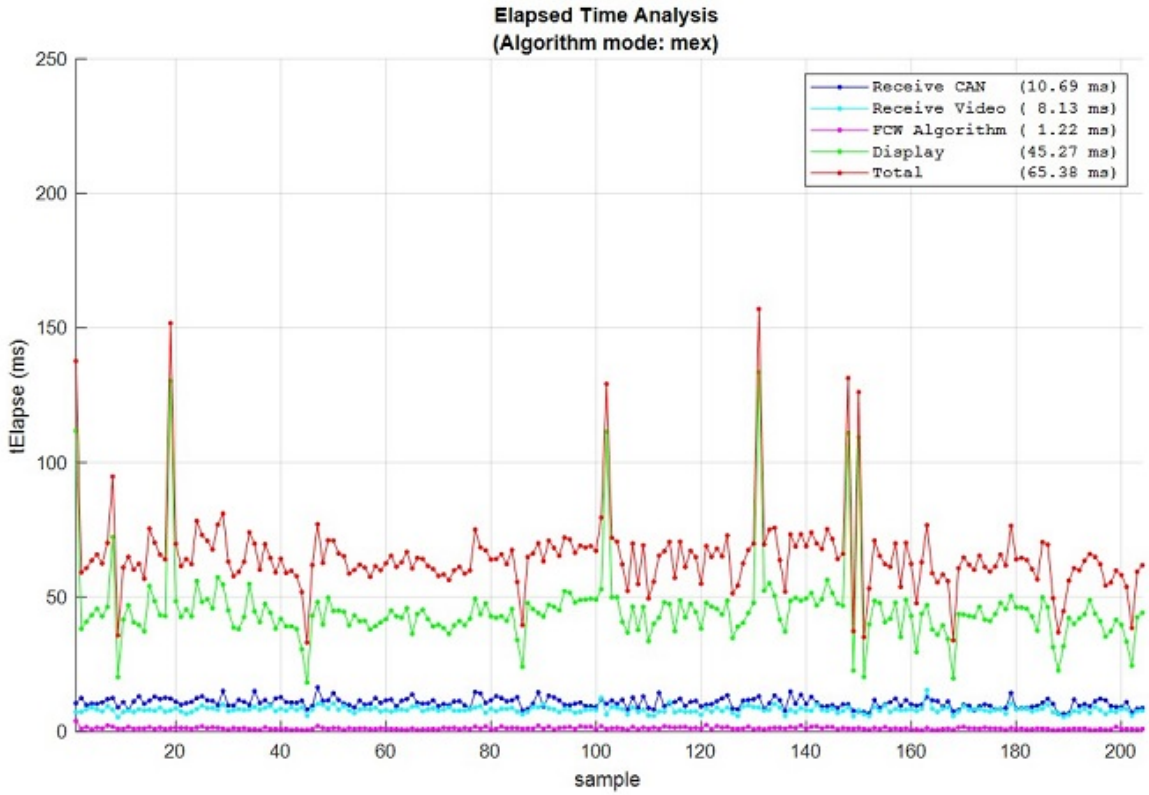
Restart the transmitter application.

```
system('matlab -nodesktop -nosplash -r helperStartTransmitter &')
```

The receiving FCW application can also be restarted. This time with an input argument to use the MEX compiled code built in the prior step.

```
helperStartReceiver('mex')
```

When ready, stop and close the transmitter and receiving FCW application. Comparing the time plot for MEX execution to the interpreted mode plot, you can see the performance improvement for the FCW algorithm.



Use Physical Hardware and Multiple Computers

The example uses a single computer to simulate the entire system with virtual connectivity. As such, its performance is meant as an approximation. You can also execute this example using two computers (one as transmitter, one as receiver). This would represent more of a real live data scenario. To achieve this, you can make simple modifications to the example code.

Changing the CAN FD communication from virtual to physical devices requires editing the transmission and reception code to invoke `canChannel` (Vehicle Network Toolbox) using a hardware device instead of the virtual channels. You may also need to modify the call to `configBusSpeed` (Vehicle Network Toolbox) depending on the capabilities of the hardware. These calls are found in the `helperStartReceiver` and `dataTransmitter` functions of the example.

Changing TCP/IP communication for multiple computers requires adjusting the TCP/IP address of the transmitter from local host (127.0.0.1) to a static value (192.168.1.2 recommended). This address is set first on the host transmitting computer. After, modify the `tcpipAddr` variable in the `helperStartReceiver` function to match.

Once configured and connected physically, you can run the transmitter application on one computer and the FCW application on the other.

See Also

Functions
`canChannel` | `canDatabase` | `configBusSpeed` | `tcpip`

Objects
`birdsEyePlot` | `multiObjectTracker`

More About

- “Forward Collision Warning Using Sensor Fusion” on page 7-125
- “Code Generation for Tracking and Sensor Fusion” on page 7-117

Multiple Object Tracking Tutorial

This example shows how to perform automatic detection and motion-based tracking of moving objects in a video. It simplifies the example “Motion-Based Multiple Object Tracking” (Computer Vision Toolbox) and uses the `multiObjectTracker` available in Automated Driving Toolbox™.

Detection of moving objects and motion-based tracking are important components of many computer vision applications, including activity recognition, traffic monitoring, and automotive safety. The problem of motion-based object tracking can be divided into two parts:

1. Detecting moving objects in each frame
2. Tracking the moving objects from frame to frame

The detection of moving objects uses a background subtraction algorithm based on Gaussian mixture models. Morphological operations are applied to the resulting foreground mask to eliminate noise. Finally, blob analysis detects groups of connected pixels, which are likely to correspond to moving objects.

The tracking of moving objects from frame to frame is done by the `multiObjectTracker` object that is responsible for the following:

1. Assigning detections to tracks.
2. Initializing new tracks based on unassigned detections. All tracks are initialized as `'Tentative'`, accounting for the possibility that they resulted from a false detection.
3. Confirming tracks if they have more than *M* assigned detections in *N* frames.
4. Updating existing tracks based on assigned detections.
5. Coasting (predicting) existing unassigned tracks.
6. Deleting tracks if they have remained unassigned (coasted) for too long.

The assignment of detections to the same object is based solely on motion. The motion of each track is estimated by a Kalman filter. The filter predicts the track's location in each frame, and determines the likelihood of each detection being assigned to each track. To initialize the filter that you design, use the `FilterInitializationFcn` property of the `multiObjectTracker`.

For more information, see “Multiple Object Tracking” (Computer Vision Toolbox).

This example is a function, with the main body at the top and helper routines in the form of nested functions below. See “Nested Functions” for more details.

```
function MultipleObjectTrackingExample()

% Create objects used for reading video and displaying the results.
videoObjects = setupVideoObjects('atrium.mp4');

% Create objects used for detecting objects in the foreground of the video.
minBlobArea = 400; % Minimum blob size, in pixels, to be considered as a detection
detectorObjects = setupDetectorObjects(minBlobArea);
```

Create the Multi-Object Tracker

When creating a `multiObjectTracker`, consider the following:

1. `FilterInitializationFcn`: The likely motion and measurement models. In this case, the objects are expected to have a constant speed motion. The `initDemoFilter` function configures a linear Kalman filter to track the motion. See the 'Define a Kalman filter' section for details.
2. `AssignmentThreshold`: How far detections may fall from tracks. The default value for this parameter is 30. If there are detections that are not assigned to tracks, but should be, increase this value. If there are detections that get assigned to tracks that are too far, decrease this value.
3. `DeletionThreshold`: How long a track is maintained before deletion. In this case, since the video has 30 frames per second, a reasonable value is about 0.75 seconds (22 frames).
4. `ConfirmationThreshold`: The parameters controlling track confirmation. A track is initialized with every unassigned detection. Some of these detections might be false, so initially, all tracks are `'Tentative'`. To confirm a track, it has to be detected at least *M* out of *N* frames. The choice of *M* and *N* depends on the visibility of the objects. This example assumes a visibility of 6 out of 10 frames.

```
tracker = multiObjectTracker(...
    'FilterInitializationFcn', @initDemoFilter, ...
    'AssignmentThreshold', 30, ...
    'DeletionThreshold', 22, ...
    'ConfirmationThreshold', [6 10] ...
    );
```

Define a Kalman Filter

When defining a tracking filter for the motion, complete the following steps:

Step 1: Define the motion model and state

In this example, use a constant velocity model in a 2-D rectangular frame.

1. The state is `[x;vx;y;vy]`.
2. The state transition model matrix is `A = [1 dt 0 0; 0 1 0 0; 0 0 1 dt; 0 0 0 1]`.
3. Assume that `dt = 1`.

Step 2: Define the process noise

The process noise represents the parts of the process that are not taken into account in the model. For example, in a constant velocity model, the acceleration is neglected.

Step 3: Define the measurement model

In this example, only the position (`[x;y]`) is measured. So, the measurement model is `H = [1 0 0 0; 0 0 1 0]`.

Note: To preconfigure these parameters, define the `'MotionModel'` property as `'2D Constant Velocity'`.

Step 4: Initialize the state vector based on the sensor measurement

In this example, because the measurement is `[x;y]` and the state is `[x;vx;y;vy]`, initializing the state vector is straightforward. Because there is no measurement of the velocity, initialize the `vx` and `vy` components to 0.

Step 5: Define an initial state covariance

In this example, the measurements are quite noisy, so define the initial state covariance to be quite large: `stateCov = diag([50, 50, 50, 50])`

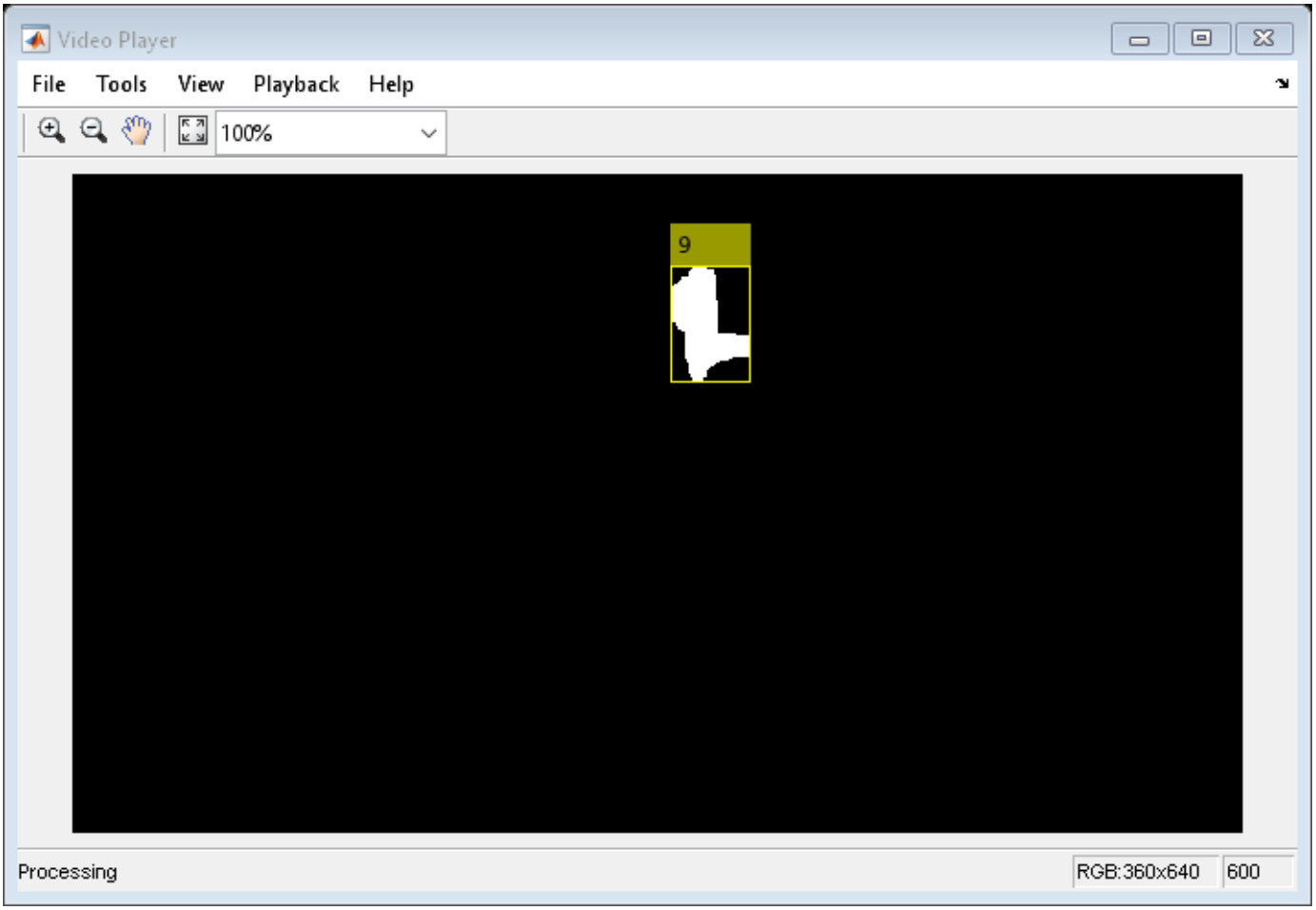
Step 6: Create the correct filter

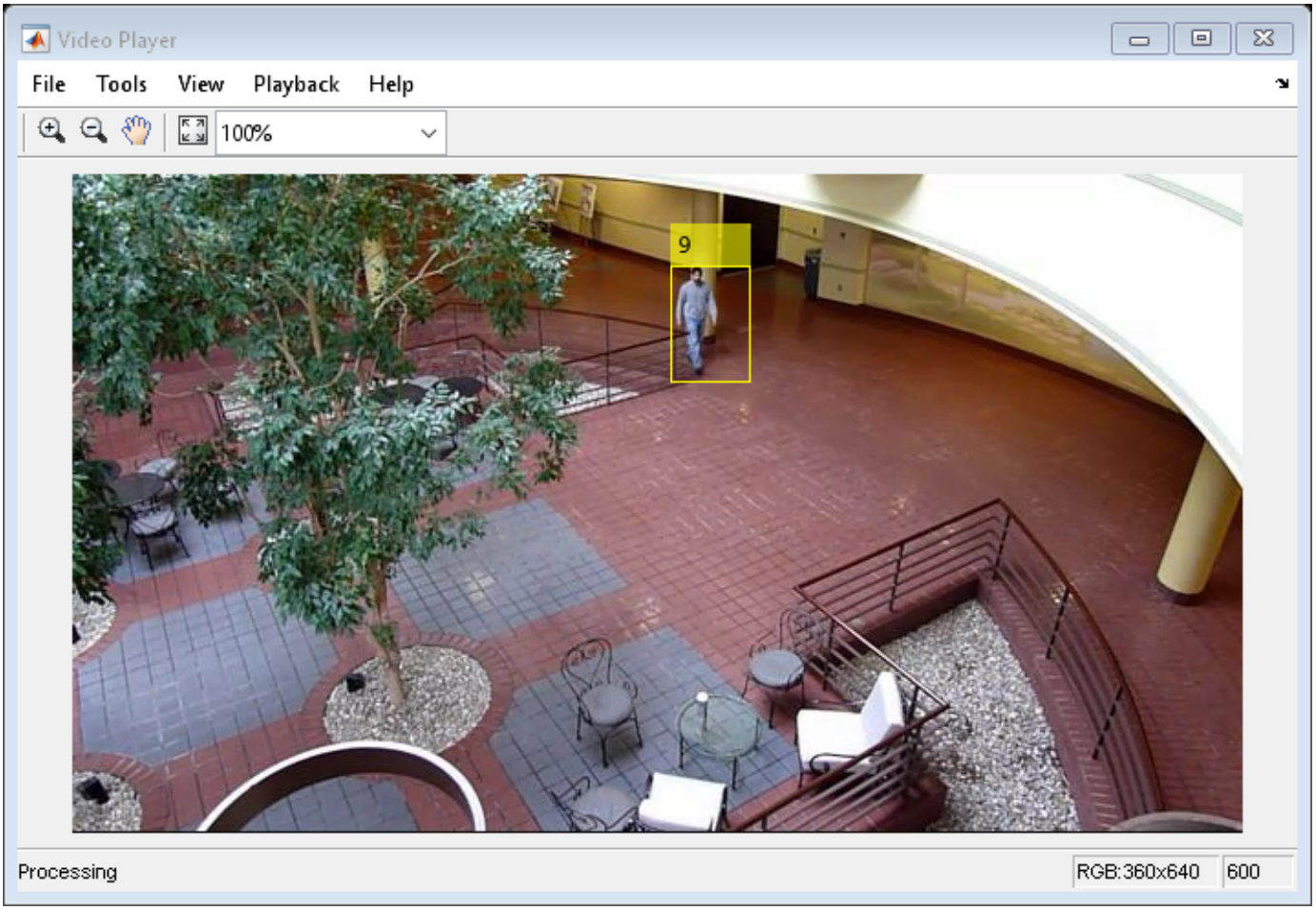
In this example, all the models are linear, so use `trackingKF` as the tracking filter.

```
function filter = initDemoFilter(detection)
% Initialize a Kalman filter for this example.

% Define the initial state.
state = [detection.Measurement(1); 0; detection.Measurement(2); 0];

% Define the initial state covariance.
stateCov = diag([50, 50, 50, 50]);

% Create the tracking filter.
filter = trackingKF('MotionModel', '2D Constant Velocity', ...
    'State', state, ...
    'StateCovariance', stateCov, ...
    'MeasurementNoise', detection.MeasurementNoise(1:2,1:2) ...
    );
end
```

The following loop runs the video clip, detects moving objects in the video, and tracks them across video frames.

```
% Count frames to create a sense of time.
frameCount = 0;
while hasFrame(videoObjects.reader)
    % Read a video frame and detect objects in it.
    frameCount = frameCount + 1;                                % Promote frame count
    frame = readFrame(videoObjects.reader);                     % Read frame
    [detections, mask] = detectObjects(detectorObjects, frame); % Detect objects in video frame

    % Run the tracker on the preprocessed detections.
    confirmedTracks = updateTracks(tracker, detections, frameCount);

    % Display the tracking results on the video.
    displayTrackingResults(videoObjects, confirmedTracks, frame, mask);
end
```


Video Player
File
Tools
View
Playback
Help
100%
9
Processing
RGB:360x640
600



Create Video Objects

Create objects used for reading and displaying the video frames.

```
function videoObjects = setupVideoObjects(filename)
    % Initialize video I/O
    % Create objects for reading a video from a file, drawing the tracked
    % objects in each frame, and playing the video.

    % Create a video file reader.
    videoObjects.reader = VideoReader(filename);

    % Create two video players: one to display the video,
    % and one to display the foreground mask.
    videoObjects.maskPlayer  = vision.VideoPlayer('Position', [20, 400, 700, 400]);
    videoObjects.videoPlayer = vision.VideoPlayer('Position', [740, 400, 700, 400]);
end
```

Create Detector Objects

Create objects used for detecting foreground objects. Use `minBlobArea` to define the size of the blob, in pixels, that is considered to be a detection.

- Increase `minBlobArea` to avoid detecting small blobs, which are more likely to be false detections, or if several detections are created for the same object due to partial occlusion.
- Decrease `minBlobArea` if objects are detected too late or not at all.

```
function detectorObjects = setupDetectorObjects(minBlobArea)
    % Create System objects for foreground detection and blob analysis

    % The foreground detector segments moving objects from the
    % background. It outputs a binary mask, where the pixel value of 1
    % corresponds to the foreground and the value of 0 corresponds to
    % the background.

    detectorObjects.detector = vision.ForegroundDetector('NumGaussians', 3, ...
        'NumTrainingFrames', 40, 'MinimumBackgroundRatio', 0.7);

    % Connected groups of foreground pixels are likely to correspond to
    % moving objects.  The blob analysis System object finds such
    % groups (called 'blobs' or 'connected components') and computes
    % their characteristics, such as their areas, centroids, and the
    % bounding boxes.

    detectorObjects.blobAnalyzer = vision.BlobAnalysis('BoundingBoxOutputPort', true, ...
        'AreaOutputPort', true, 'CentroidOutputPort', true, ...
        'MinimumBlobArea', minBlobArea);
end
```

Detect Objects

The `detectObjects` function returns the centroids and the bounding boxes of the detected objects as a list of `objectDetection` objects. You can supply this list as an input to the `multiObjectTracker`. The `detectObjects` function also returns the binary mask, which has the same size as the input frame. Pixels with a value of 1 correspond to the foreground. Pixels with a value of 0 correspond to the background.

The function performs motion segmentation using the foreground detector. It then performs morphological operations on the resulting binary mask to remove noisy pixels and to fill the holes in the remaining blobs.

When creating the `objectDetection` list, the `frameCount` serves as the time input, and the centroids of the detected blobs serve as the measurement. The list also has two optional name-value pairs:

- `MeasurementNoise` - Blob detection is noisy, and this example defines a large measurement noise value.
- `ObjectAttributes` - The detected bounding boxes that get passed to the track display are added to this argument.

```
function [detections, mask] = detectObjects(detectorObjects, frame)
    % Expected uncertainty (noise) for the blob centroid.
    measurementNoise = 100*eye(2);
    % Detect foreground.
    mask = detectorObjects.detector.step(frame);

    % Apply morphological operations to remove noise and fill in holes.
    mask = imopen(mask, strel('rectangle', [6, 6]));
    mask = imclose(mask, strel('rectangle', [50, 50]));
```

```
        mask = imfill(mask, 'holes');

        % Perform blob analysis to find connected components.
        [~, centroids, bboxes] = detectorObjects.blobAnalyzer.step(mask);

        % Formulate the detections as a list of objectDetection objects.
        numDetections = size(centroids, 1);
        detections = cell(numDetections, 1);
        for i = 1:numDetections
            detections{i} = objectDetection(frameCount, centroids(i,:), ...
                'MeasurementNoise', measurementNoise, ...
                'ObjectAttributes', {bboxes(i,:)});
        end
    end
```

Display Tracking Results

The `displayTrackingResults` function draws a bounding box and label ID for each track on the video frame and foreground mask. It then displays the frame and the mask in their respective video players.

```
    function displayTrackingResults(videoObjects, confirmedTracks, frame, mask)
        % Convert the frame and the mask to uint8 RGB.
        frame = im2uint8(frame);
        mask = uint8(repmat(mask, [1, 1, 3])) .* 255;

        if ~isempty(confirmedTracks)
            % Display the objects. If an object has not been detected
            % in this frame, display its predicted bounding box.
            numRelTr = numel(confirmedTracks);
            boxes = zeros(numRelTr, 4);
            ids = zeros(numRelTr, 1, 'int32');
            predictedTrackInds = zeros(numRelTr, 1);
            for tr = 1:numRelTr
                % Get bounding boxes.
                boxes(tr, :) = confirmedTracks(tr).ObjectAttributes{1}{1};

                % Get IDs.
                ids(tr) = confirmedTracks(tr).TrackID;

                if confirmedTracks(tr).IsCoasted
                    predictedTrackInds(tr) = tr;
                end
            end

            predictedTrackInds = predictedTrackInds(predictedTrackInds > 0);

            % Create labels for objects that display the predicted rather
            % than the actual location.
            labels = cellstr(int2str(ids));

            isPredicted = cell(size(labels));
            isPredicted(predictedTrackInds) = {' predicted'};
            labels = strcat(labels, isPredicted);

            % Draw the objects on the frame.
            frame = insertObjectAnnotation(frame, 'rectangle', boxes, labels);
```

```
            % Draw the objects on the mask.
            mask = insertObjectAnnotation(mask, 'rectangle', boxes, labels);
        end

        % Display the mask and the frame.
        videoObjects.maskPlayer.step(mask);
        videoObjects.videoPlayer.step(frame);
    end

end
```

Summary

In this example, you created a motion-based system for detecting and tracking multiple moving objects. Try using a different video to see if you can detect and track objects. Try modifying the parameters of the `multiObjectTracker`.

The tracking in this example was based solely on motion, with the assumption that all objects move in a straight line with constant speed. When the motion of an object significantly deviates from this model, the example can produce tracking errors. Notice the mistake in tracking the person occluded by the tree.

You can reduce the likelihood of tracking errors by using a more complex motion model, such as constant acceleration or constant turn. To do that, try defining a different tracking filter, such as `trackingEKF` or `trackingUKF`.

See Also

Objects
`VideoReader` | `multiObjectTracker` | `objectDetection` | `trackingEKF` | `trackingKF` | `trackingUKF` | `vision.VideoPlayer`

More About

- “Track Multiple Vehicles Using a Camera” on page 7-170
- “Multiple Object Tracking” (Computer Vision Toolbox)
- “Motion-Based Multiple Object Tracking” (Computer Vision Toolbox)

Track Multiple Vehicles Using a Camera

This example shows how to detect and track multiple vehicles with a monocular camera mounted in a vehicle.

Overview

Automated Driving Toolbox™ provides pretrained vehicle detectors and a multi-object tracker to facilitate tracking vehicles around ego vehicle. The vehicle detectors are based on ACF features and Faster R-CNN, a deep-learning-based object detection technique. The detectors can be easily interchanged to see their effect on vehicle tracking.

The tracking workflow consists of the following steps:

1. Define camera intrinsics and camera mounting position.
2. Load and configure a pretrained vehicle detector.
3. Set up a multi-object tracker.
4. Run the detector for each video frame.
5. Update the tracker with detection results.
6. Display the tracking results in a video.

Configure Vehicle Detector and Multi-Object Tracker

In this example, you use a pretrained ACF vehicle detector and configure this detector to incorporate camera information. By default, the detector scans the entire image at multiple scales. By knowing the camera parameters, you can configure the detector to detect vehicles on the ground plane only at reasonable scales.

```
% Load the monoCamera object that contains the camera information.
d = load('FCWDemoMonoCameraSensor.mat', 'sensor');

% Load a pretrained ACF vehicle detector. The ACF detector uses "Aggregate
% Channel Features", which is fast to compute in practice. The 'full-view'
% model is trained on images of the front, rear, left, and right side of
% vehicles.
detector = vehicleDetectorACF('full-view');
```

To try the Faster R-CNN vehicle detector, use `vehicleDetectorFasterRCNN` instead. This detector requires a Deep Learning Toolbox™ license.

Configure the detector using the sensor information. The detector only tries to find vehicles at image regions above the ground plane. This can reduce computation and prevent spurious detections.

```
% The width of common vehicles is between 1.5 to 2.5 meters. Only a
% bounding box of width within this range is considered as a detection
% candidate in image.
vehicleWidth = [1.5, 2.5];

% Configure the detector using the monoCamera sensor and desired width.
detector = configureDetectorMonoCamera(detector, d.sensor, vehicleWidth);

% Initialize an multi-object tracker including setting the filter,
% the detection-to-track assignment threshold, the coasting and
% confirmation parameters. You can find the |setupTracker| function at the
```

```
% end of this example.
[tracker, positionSelector] = setupTracker();
```

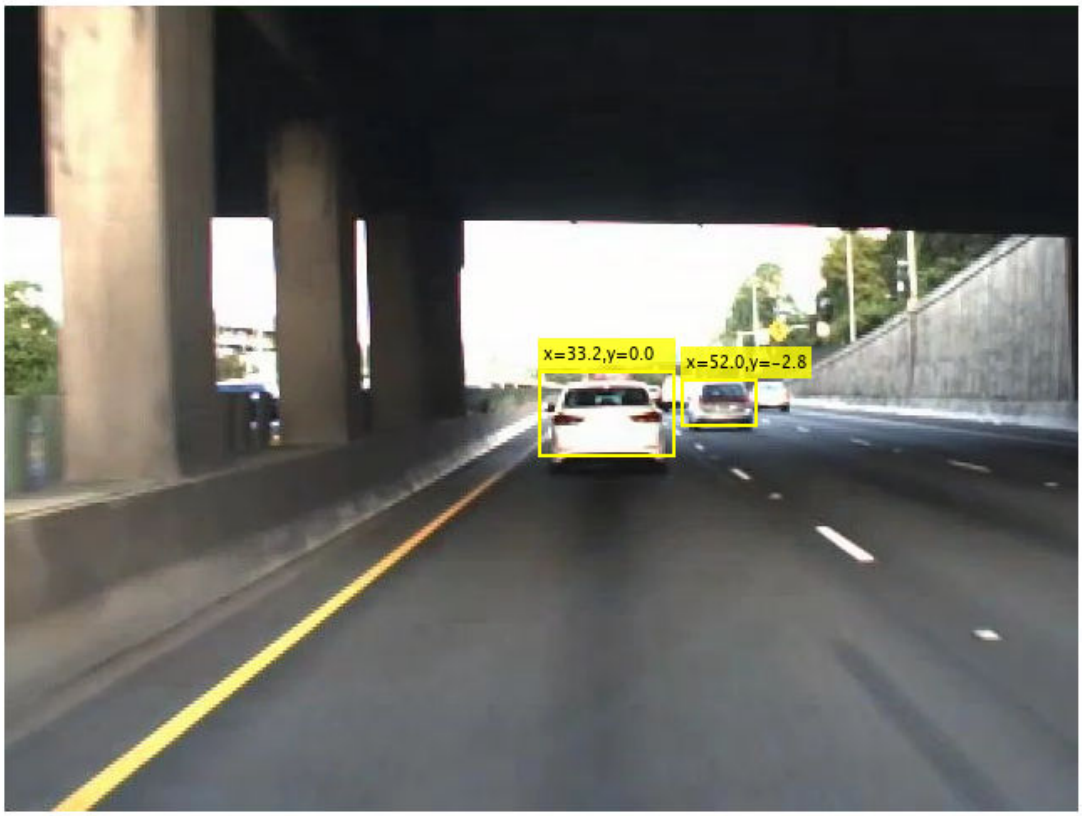
Track Vehicles in a Video

At each time step, run the detector, update the tracker with detection results, and display the tracking results in a video.

```
% Setup Video Reader and Player
videoFile   = '05_highway_lanechange_25s.mp4';
videoReader = VideoReader(videoFile);
videoPlayer = vision.DeployableVideoPlayer();

currentStep = 0;
snapshot = [];
snapTimeStamp = 120;
cont = hasFrame(videoReader);
while cont
    % Update frame counters.
    currentStep = currentStep + 1;

    % Read the next frame.
    frame = readFrame(videoReader);

    % Run the detector and package the returned results into an object
    % required by multiObjectTracker.  You can find the |detectObjects|
    % function at the end of this example.
    detections = detectObjects(detector, frame, currentStep);

    % Using the list of objectDetections, return the tracks, updated for
    % 'currentStep' time.
    confirmedTracks = updateTracks(tracker, detections, currentStep);

    % Remove the tracks for vehicles that are far away.
    confirmedTracks = removeNoisyTracks(confirmedTracks, positionSelector, d.sensor.Intrinsics.Ir

    % Insert tracking annotations.
    frameWithAnnotations = insertTrackBoxes(frame, confirmedTracks, positionSelector, d.sensor);

    % Display the annotated frame.
    videoPlayer(frameWithAnnotations);

    % Take snapshot for publishing at snapTimeStamp seconds
    if currentStep == snapTimeStamp
        snapshot = frameWithAnnotations;
    end

    % Exit the loop if the video player figure is closed by user.
    cont = hasFrame(videoReader) && isOpen(videoPlayer);
end
```

Show the tracked vehicles and display the distance to the ego vehicle.

```
if ~isempty(snapshot)
    figure
    imshow(snapshot)
end
```



The tracking workflow presented here can be easily integrated into the “Visual Perception Using Monocular Camera” on page 7-78 example, where the vehicle detection step can be enhanced with the tracker. To learn about additional tracking capabilities in Automated Driving Toolbox™, see `monoCamera` and `multiObjectTracker`.

Supporting Functions

setupTracker function creates a `multiObjectTracker` to track multiple objects with Kalman filters. When creating a `multiObjectTracker` consider the following:

- `FilterInitializationFcn`: The likely motion and measurement models. In this case, the objects are expected to have a constant velocity motion. See the 'Define a Kalman filter' section.
- `AssignmentThreshold`: How far detections can fall from tracks. The default value for this parameter is 30. If there are detections that are not assigned to tracks, but should be, increase this value. If there are detections that get assigned to tracks that are too far, decrease this value. This example uses 50.
- `DeletionThreshold`: How many times can the track be not assigned a detection (missed) in the last Q steps before its deletion. Coasting is a term used for updating the track without an assigned detection (predicting). The default value for this parameter is 5 misses out of 5 last updates.
- `ConfirmationThreshold`: The parameters for confirming a track. A new track is initialized with every unassigned detection. Some of these detections might be false, so all the tracks are

initialized as `Tentative`. To confirm a track, it has to be detected at least *M* times in *N* tracker updates. The choice of *M* and *N* depends on the visibility of the objects. This example uses the default of 3 detections out of 5 updates.

The outputs of `setupTracker` are:

- `tracker` - The `multiObjectTracker` that is configured for this case.
- `positionSelector` - A matrix that specifies which elements of the State vector are the position: `position = positionSelector * State`

```
function [tracker, positionSelector] = setupTracker()
    % Create the tracker object.
    tracker = multiObjectTracker('FilterInitializationFcn', @initBboxFilter, ...
        'AssignmentThreshold', 50, ...
        'DeletionThreshold', 5, ...
        'ConfirmationThreshold', [3 5]);

    % The State vector is: [x; vx; y; vy; w; vw; h; vh]
    % [x;y;w;h] = positionSelector * State
    positionSelector = [1 0 0 0 0 0 0 0; ...
                        0 0 1 0 0 0 0 0; ...
                        0 0 0 0 1 0 0 0; ...
                        0 0 0 0 0 0 1 0];
end
```

initBboxFilter function defines a Kalman filter to filter bounding box measurement.

```
function filter = initBboxFilter(Detection)
% Step 1: Define the motion model and state.
%   Use a constant velocity model for a bounding box on the image.
%   The state is [x; vx; y; vy; w; wv; h; hv]
%   The state transition matrix is:
%       [1 dt 0  0 0  0 0  0;
%        0  1 0  0 0  0 0  0;
%        0  0 1 dt 0  0 0  0;
%        0  0 0  1 0  0 0  0;
%        0  0 0  0 1 dt 0  0;
%        0  0 0  0 0  1 0  0;
%        0  0 0  0 0  0 1 dt;
%        0  0 0  0 0  0 0  1]
%   Assume dt = 1. This example does not consider time-variant transition
%   model for linear Kalman filter.
    dt = 1;
    cvel =[1 dt; 0 1];
    A = blkdiag(cvel, cvel, cvel, cvel);

% Step 2: Define the process noise.
%   The process noise represents the parts of the process that the model
%   does not take into account. For example, in a constant velocity model,
%   the acceleration is neglected.
    G1d = [dt^2/2; dt];
    Q1d = G1d*G1d';
    Q = blkdiag(Q1d, Q1d, Q1d, Q1d);

% Step 3: Define the measurement model.
%   Only the position ([x;y;w;h]) is measured.
%   The measurement model is
```

```
    H = [1 0 0 0 0 0 0 0; ...
         0 0 1 0 0 0 0 0; ...
         0 0 0 0 1 0 0 0; ...
         0 0 0 0 0 0 1 0];

% Step 4: Map the sensor measurements to an initial state vector.
%   Because there is no measurement of the velocity, the v components are
%   initialized to 0:
    state = [Detection.Measurement(1); ...
             0; ...
             Detection.Measurement(2); ...
             0; ...
             Detection.Measurement(3); ...
             0; ...
             Detection.Measurement(4); ...
             0];

% Step 5: Map the sensor measurement noise to a state covariance.
%   For the parts of the state that the sensor measured directly, use the
%   corresponding measurement noise components. For the parts that the
%   sensor does not measure, assume a large initial state covariance. That way,
%   future detections can be assigned to the track.
    L = 100; % Large value
    stateCov = diag([Detection.MeasurementNoise(1,1), ...
                     L, ...
                     Detection.MeasurementNoise(2,2), ...
                     L, ...
                     Detection.MeasurementNoise(3,3), ...
                     L, ...
                     Detection.MeasurementNoise(4,4), ...
                     L]);

% Step 6: Create the correct filter.
%   In this example, all the models are linear, so use trackingKF as the
%   tracking filter.
    filter = trackingKF(...
        'StateTransitionModel', A, ...
        'MeasurementModel', H, ...
        'State', state, ...
        'StateCovariance', stateCov, ...
        'MeasurementNoise', Detection.MeasurementNoise, ...
        'ProcessNoise', Q);
end
```

detectObjects function detects vehicles in an image.

```
function detections = detectObjects(detector, frame, frameCount)
    % Run the detector and return a list of bounding boxes: [x, y, w, h]
    bboxes = detect(detector, frame);

    % Define the measurement noise.
    L = 100;
    measurementNoise = [L 0  0  0; ...
                        0 L  0  0; ...
                        0 0 L/2 0; ...
                        0 0  0 L/2];

    % Formulate the detections as a list of objectDetection reports.
```

```
    numDetections = size(bboxes, 1);
    detections = cell(numDetections, 1);
    for i = 1:numDetections
        detections{i} = objectDetection(frameCount, bboxes(i, :), ...
            'MeasurementNoise', measurementNoise);
    end
end
```

removeNoisyTracks function removes noisy tracks. A track is considered to be noisy if its predicted bounding box is too small. Typically, this implies the vehicle is far away.

```
function tracks = removeNoisyTracks(tracks, positionSelector, imageSize)

    if isempty(tracks)
        return
    end

    % Extract the positions from all the tracks.
    positions = getTrackPositions(tracks, positionSelector);
    % The track is 'invalid' if the predicted position is about to move out
    % of the image, or if the bounding box is too small.
    invalid = ( positions(:, 1) < 1 | ...
                positions(:, 1) + positions(:, 3) > imageSize(2) | ...
                positions(:, 3) <= 20 | ...
                positions(:, 4) <= 20 );
    tracks(invalid) = [];
end
```

insertTrackBoxes inserts bounding boxes in an image and displays the track's position in front of the car, in world units.

```
function I = insertTrackBoxes(I, tracks, positionSelector, sensor)

    if isempty(tracks)
        return
    end

    % Allocate memory.
    labels = cell(numel(tracks), 1);
    % Retrieve positions of bounding boxes.
    bboxes = getTrackPositions(tracks, positionSelector);

    for i = 1:numel(tracks)
        box = bboxes(i, :);

        % Convert to vehicle coordinates using monoCamera object.
        xyVehicle = imageToVehicle(sensor, [box(1)+box(3)/2, box(2)+box(4)]);

        labels{i} = sprintf('x=%.1f,y=%.1f',xyVehicle(1),xyVehicle(2));
    end

    I = insertObjectAnnotation(I, 'rectangle', bboxes, labels, 'Color', 'yellow', ...
```

```
        'FontSize', 10, 'TextBoxOpacity', .8, 'LineWidth', 2);
end
```

See Also

Functions
`configureDetectorMonoCamera` | `vehicleDetectorACF` | `vehicleDetectorFasterRCNN`

Objects
`VideoReader` | `acfObjectDetectorMonoCamera` | `fasterRCNNObjectDetectorMonoCamera` | `monoCamera` | `multiObjectTracker` | `objectDetection` | `trackingKF` | `vision.DeployableVideoPlayer`

More About

- “Visual Perception Using Monocular Camera” on page 7-78

Track Vehicles Using Lidar: From Point Cloud to Track List

This example shows you how to track vehicles using measurements from a lidar sensor mounted on top of an ego vehicle. Lidar sensors report measurements as a point cloud. The example illustrates the workflow in MATLAB® for processing the point cloud and tracking the objects. For a Simulink® version of the example, refer to “Track Vehicles Using Lidar Data in Simulink” (Sensor Fusion and Tracking Toolbox).The lidar data used in this example is recorded from a highway driving scenario. In this example, you use the recorded data to track vehicles with a joint probabilistic data association (JPDA) tracker and an interacting multiple model (IMM) approach.

3-D Bounding Box Detector Model

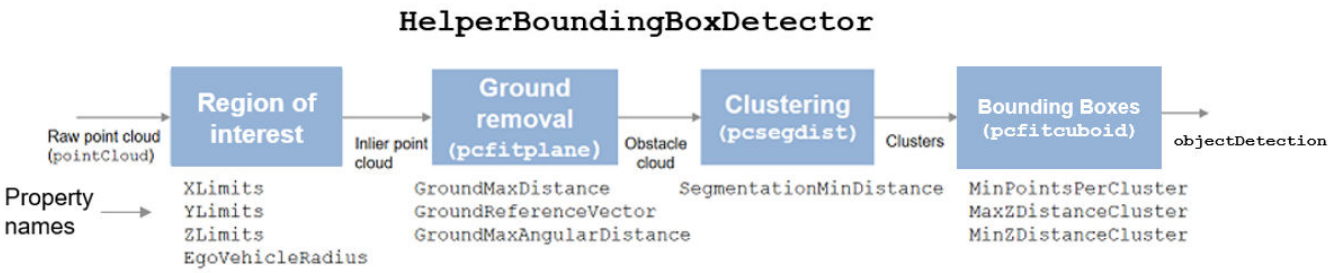
Due to high resolution capabilities of the lidar sensor, each scan from the sensor contains a large number of points, commonly known as a point cloud. This raw data must be preprocessed to extract objects of interest, such as cars, cyclists, and pedestrians. In this example, you use a classical segmentation algorithm using a distance-based clustering algorithm. For more details about segmentation of lidar data into objects such as the ground plane and obstacles, refer to the “Ground Plane and Obstacle Detection Using Lidar” on page 7-107 example. For a deep learning segmentation workflow, refer to the “Detect, Classify, and Track Vehicles Using Lidar” (Lidar Toolbox) example. In this example, the point clouds belonging to obstacles are further classified into clusters using the `pcsegdist` function, and each cluster is converted to a bounding box detection with the following format:

$$[x\ y\ z\ \theta\ l\ w\ h]$$

x, y and z refer to the x-, y- and z-positions of the bounding box, θ refers to its yaw angle and l, w and h refer to its length, width, and height, respectively. The `pcfitcuboid` (Lidar Toolbox) function uses L-shape fitting algorithm to determine the yaw angle of the bounding box.

The detector is implemented by a supporting class `HelperBoundingBoxDetector`, which wraps around point cloud segmentation and clustering functionalities. An object of this class accepts a `pointCloud` input and returns a list of `objectDetection` objects with bounding box measurements.

The diagram shows the processes involved in the bounding box detector model and the Lidar Toolbox™ functions used to implement each process. It also shows the properties of the supporting class that control each process.



The lidar data is available at the following location: https://ssd.mathworks.com/supportfiles/lidar/data/TrackVehiclesUsingLidarExampleData.zip

Download the data files into your temporary directory, whose location is specified by MATLAB's `tempdir` function. If you want to place the files in a different folder, change the directory name in the subsequent instructions.

```
% Load data if unavailable. The lidar data is stored as a cell array of
% pointCloud objects.
if ~exist('lidarData','var')
    dataURL = 'https://ssd.mathworks.com/supportfiles/lidar/data/TrackVehiclesUsingLidarExampleDa
    datasetFolder = fullfile(tempdir,'LidarExampleDataset');
    if ~exist(datasetFolder,'dir')
       unzip(dataURL,datasetFolder);
    end
    % Specify initial and final time for simulation.
    initTime = 0;
    finalTime = 35;
    [lidarData, imageData] = loadLidarAndImageData(datasetFolder,initTime,finalTime);
end

% Set random seed to generate reproducible results.
S = rng(2018);

% A bounding box detector model.
detectorModel = HelperBoundingBoxDetector(...
    'XLimits',[-50 75],...              % min-max
    'YLimits',[-5 5],...                % min-max
    'ZLimits',[-2 5],...                % min-max
    'SegmentationMinDistance',1.8,...   % minimum Euclidian distance
    'MinDetectionsPerCluster',1,...     % minimum points per cluster
    'MeasurementNoise',blkdiag(0.25*eye(3),25,eye(3)),...       % measurement noise in detection
    'GroundMaxDistance',0.3);           % maximum distance of ground points from ground plane
```

Target State and Sensor Measurement Model

The first step in tracking an object is defining its state, and the models that define the transition of state and the corresponding measurement. These two sets of equations are collectively known as the state-space model of the target. To model the state of vehicles for tracking using lidar, this example uses a cuboid model with following convention:

$$x = [x_{kin}\ \theta\ l\ w\ h]$$

x_{kin} refers to the portion of the state that controls the kinematics of the motion center, and θ is the yaw angle. The length, width, height of the cuboid are modeled as a constants, whose estimates evolve in time during correction stages of the filter.

In this example, you use two state-space models: a constant velocity (cv) cuboid model and a constant turn-rate (ct) cuboid model. These models differ in the way they define the kinematic part of the state, as described below:

$$x_{cv} = [x\ \dot{x}\ y\ \dot{y}\ z\ \dot{z}\ \theta\ l\ w\ h]$$

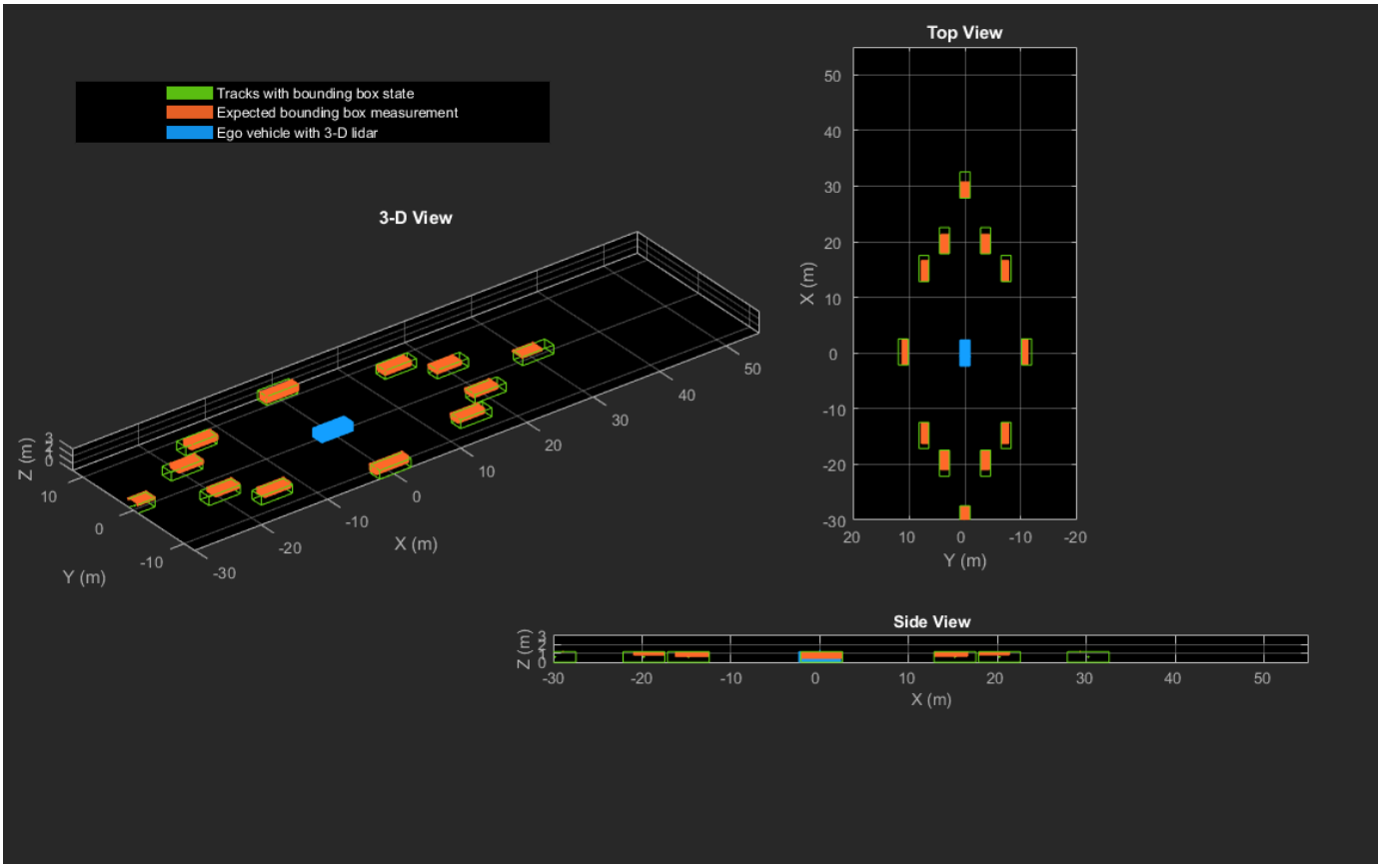
$$x_{ct} = [x\ \dot{x}\ y\ \dot{y}\ \dot{\theta}\ z\ \dot{z}\ \theta\ l\ w\ h]$$

For information about their state transition, refer to the `helperConstvelCuboid` and `helperConstturnCuboid` functions used in this example.

The `helperCvmeasCuboid` and `helperCtmeasCuboid` measurement models describe how the sensor perceives the constant velocity and constant turn-rate states respectively, and they return

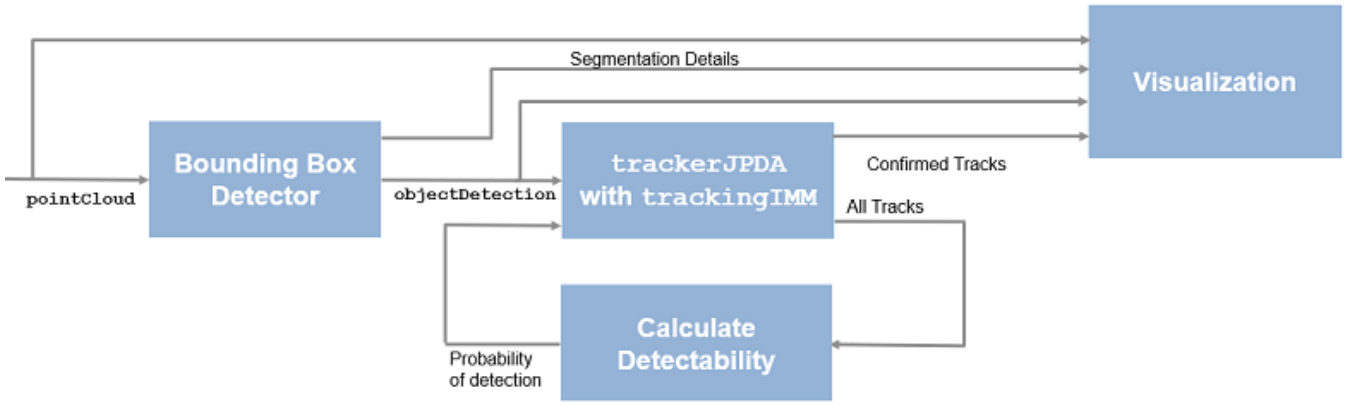
bounding box measurements. Because the state contains information about size of the target, the measurement model includes the effect of center-point offset and bounding box shrinkage, as perceived by the sensor, due to effects like self-occlusion [1]. This effect is modeled by a shrinkage factor that is directly proportional to the distance from the tracked vehicle to the sensor.

The image below demonstrates the measurement model operating at different state-space samples. Notice the modeled effects of bounding box shrinkage and center-point offset as the objects move around the ego vehicle.



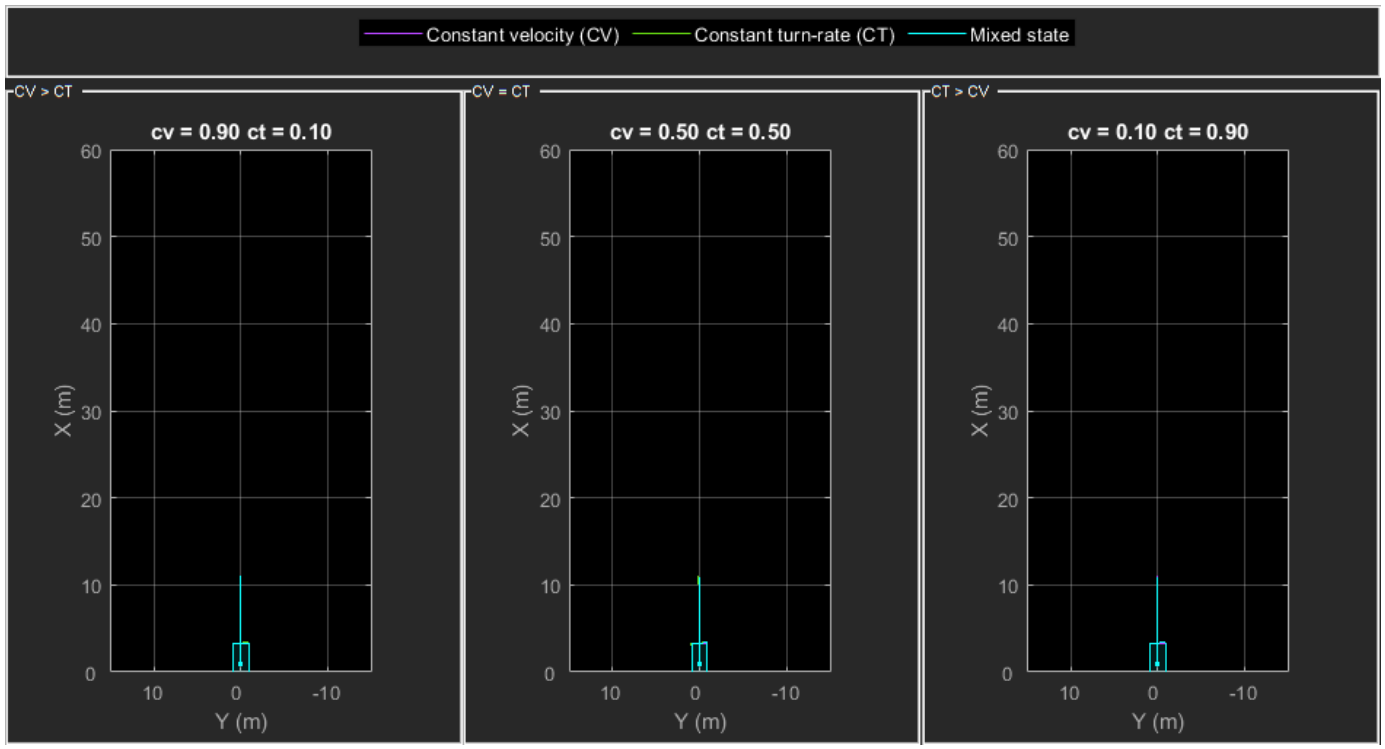
Set Up Tracker and Visualization

The image below shows the complete workflow to obtain a list of tracks from a pointCloud input.

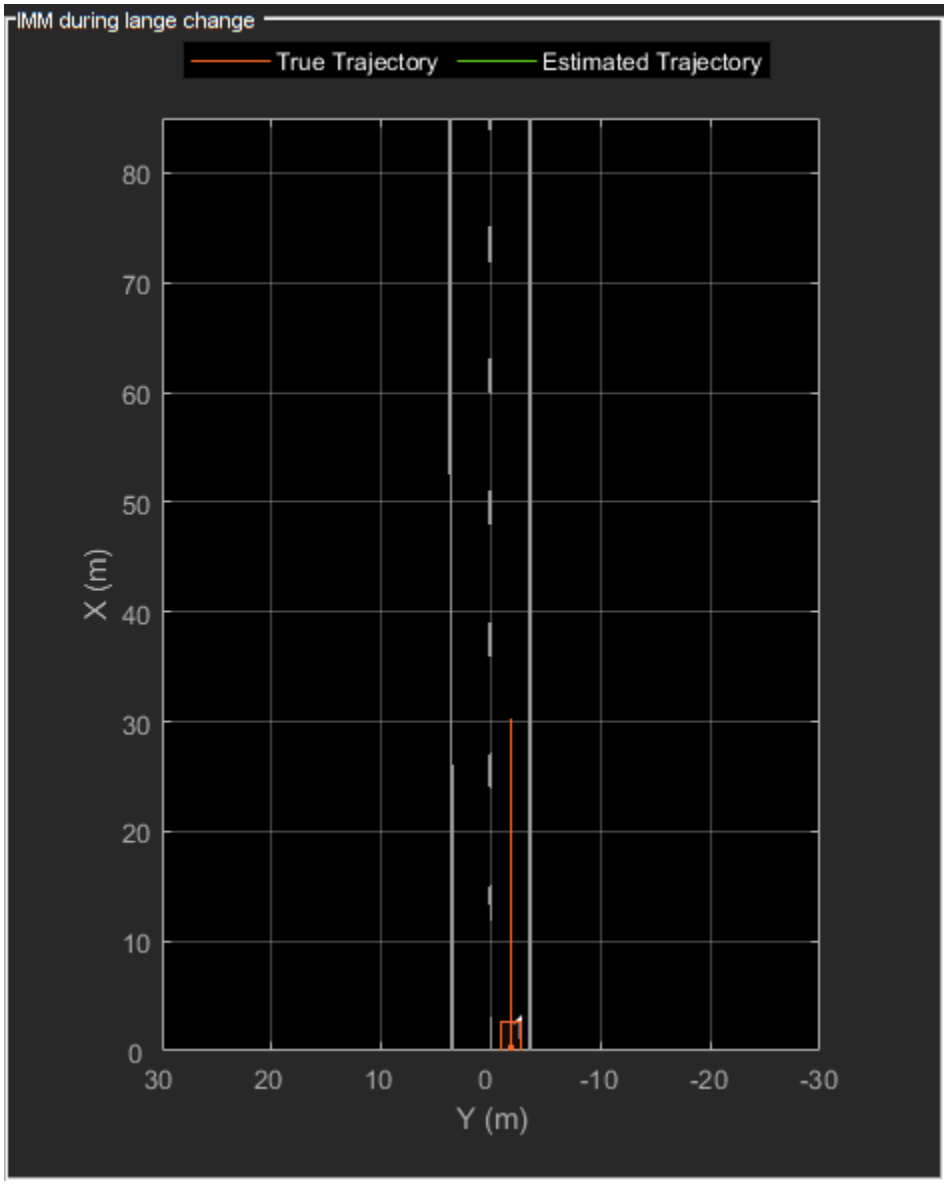


Now, set up the tracker and the visualization used in the example.

A joint probabilistic data association tracker (`trackerJPDA`) coupled with an IMM filter (`trackingIMM`) is used to track objects in this example. The IMM filter uses a constant velocity and constant turn-rate model and is initialized using the supporting function, `helperInitIMMFilter`, included with this example. The IMM approach helps a track to switch between motion models and thus achieve good estimation accuracy during events like maneuvering or lane changing. The animation below shows the effect of mixing the constant velocity and constant turn-rate model during prediction stages of the IMM filter.



The IMM filter updates the probability of each model when it is corrected with detections from the object. The animation below shows the estimated trajectory of a vehicle during a lane change event and the corresponding estimated probabilities of each model.



Set the `HasDetectableTrackIDsInput` property of the tracker as `true`, which enables you to specify a state-dependent probability of detection. The detection probability of a track is calculated by the `helperCalcDetectability` function, listed at the end of this example.

```
assignmentGate = [75 1000]; % Assignment threshold;
confThreshold = [7 10];     % Confirmation threshold for history logic
delThreshold = [8 10];      % Deletion threshold for history logic
Kc = 1e-9;                  % False-alarm rate per unit volume

% IMM filter initialization function
filterInitFcn = @helperInitIMMFilter;

% A joint probabilistic data association tracker with IMM filter
tracker = trackerJPDA('FilterInitializationFcn',filterInitFcn,...
    'TrackLogic','History',...
    'AssignmentThreshold',assignmentGate,...
```

```
    'ClutterDensity',Kc,...
    'ConfirmationThreshold',confThreshold,...
    'DeletionThreshold',delThreshold,...
    'HasDetectableTrackIDsInput',true,...
    'InitializationThreshold',0,...
    'HitMissThreshold',0.1);
```

The visualization is divided into these main categories:

1. Lidar Preprocessing and Tracking - This display shows the raw point cloud, segmented ground, and obstacles. It also shows the resulting detections from the detector model and the tracks of vehicles generated by the tracker.
2. Ego Vehicle Display - This display shows the 2-D bird's-eye view of the scenario. It shows the obstacle point cloud, bounding box detections, and the tracks generated by the tracker. For reference, it also displays the image recorded from a camera mounted on the ego vehicle and its field of view.
3. Tracking Details - This display shows the scenario zoomed around the ego vehicle. It also shows finer tracking details, such as error covariance in estimated position of each track and its motion model probabilities, denoted by cv and ct.

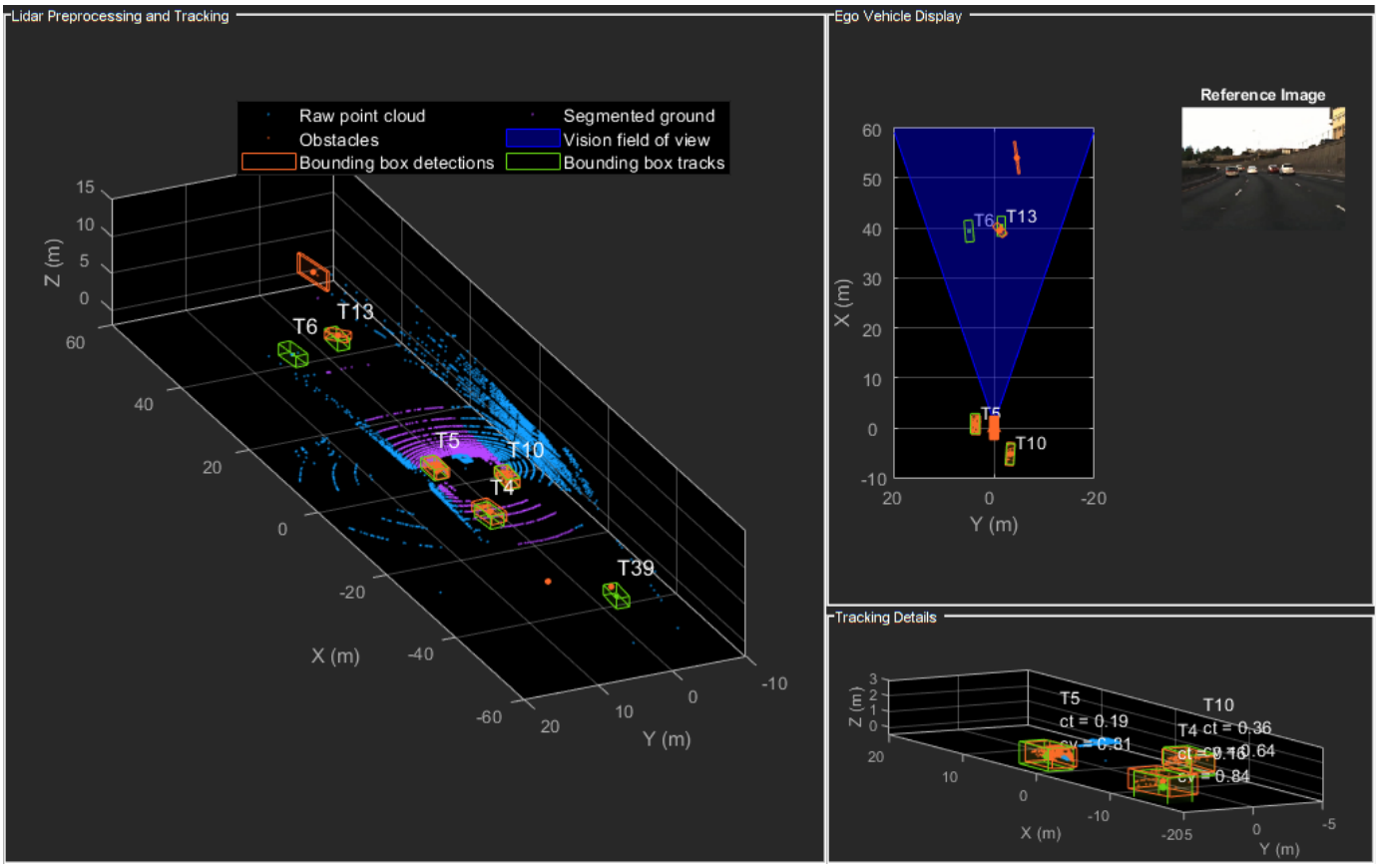
```
% Create display
displayObject = HelperLidarExampleDisplay(imageData{1},...
    'PositionIndex',[1 3 6],...
    'VelocityIndex',[2 4 7],...
    'DimensionIndex',[9 10 11],...
    'YawIndex',8,...
    'MovieName','',...  % Specify a movie name to record a movie.
    'RecordGIF',false); % Specify true to record new GIFs
```

Loop Through Data

Loop through the recorded lidar data, generate detections from the current point cloud using the detector model and then process the detections using the tracker.

```
time = 0;       % Start time
dT = 0.1;       % Time step

% Initiate all tracks.
allTracks = struct([]);

% Initiate variables for comparing MATLAB and MEX simulation.
numTracks = zeros(numel(lidarData),2);

% Loop through the data
for i = 1:numel(lidarData)
    % Update time
    time = time + dT;

    % Get current lidar scan
    currentLidar = lidarData{i};

    % Generator detections from lidar scan.
    [detections,obstacleIndices,groundIndices,croppedIndices] = detectorModel(currentLidar,time)

    % Calculate detectability of each track.
    detectableTracksInput = helperCalcDetectability(allTracks,[1 3 6]);
```

```
        % Pass detections to track.
        [confirmedTracks,tentativeTracks,allTracks,info] = tracker(detections,time,detectableTracksIn
        numTracks(i,1) = numel(confirmedTracks);

        % Get model probabilities from IMM filter of each track using
        % getTrackFilterProperties function of the tracker.
        modelProbs = zeros(2,numel(confirmedTracks));
        for k = 1:numel(confirmedTracks)
            c1 = getTrackFilterProperties(tracker,confirmedTracks(k).TrackID,'ModelProbabilities');
            modelProbs(:,k) = c1{1};
        end

        % Update display
        if isvalid(displayObject.PointCloudProcessingDisplay.ObstaclePlotter)
            % Get current image scan for reference image
            currentImage = imageData{i};

            % Update display object
            displayObject(detections,confirmedTracks,currentLidar,obstacleIndices,...
                groundIndices,croppedIndices,currentImage,modelProbs);
        end

        % Snap a figure at time = 18
        if abs(time - 18) < dT/2
            snapnow(displayObject);
        end
    end

    % Write movie if requested
    if ~isempty(displayObject.MovieName)
        writeMovie(displayObject);
    end

    % Write new GIFs if requested.
    if displayObject.RecordGIF
        % second input is start frame, third input is end frame and last input
        % is a character vector specifying the panel to record.
        writeAnimatedGIF(displayObject,10,170,'trackMaintenance','ego');
        writeAnimatedGIF(displayObject,310,330,'jpda','processing');
        writeAnimatedGIF(displayObject,120,140,'imm','details');
    end
```



The figure above shows the three displays at time = 18 seconds. The tracks are represented by green bounding boxes. The bounding box detections are represented by orange bounding boxes. The detections also have orange points inside them, representing the point cloud segmented as obstacles. The segmented ground is shown in purple. The cropped or discarded point cloud is shown in blue.

Generate C Code

You can generate C code from the MATLAB® code for the tracking and the preprocessing algorithm using MATLAB Coder™. C code generation enables you to accelerate MATLAB code for simulation. To generate C code, the algorithm must be restructured as a MATLAB function, which can be compiled into a MEX file or a shared library. For this purpose, the point cloud processing algorithm and the tracking algorithm is restructured into a MATLAB function, `mexLidarTracker`. Some variables are defined as `persistent` to preserve their state between multiple calls to the function (see `persistent`). The inputs and outputs of the function can be observed in the function description provided in the "Supporting Files" section at the end of this example.

MATLAB coder requires specifying the properties of all the input arguments. An easy way to do this is by defining the input properties by example at the command line using the `-args` option. For more information, see “Define Input Properties by Example at the Command Line” (MATLAB Coder). Note that the top-level input arguments cannot be objects of the `handle` class. Therefore, the function accepts the *x*, *y* and *z* locations of the point cloud as an input. From the stored point cloud, this information can be extracted using the `Location` property of the `pointCloud` object. This information is also directly available as the raw data from the lidar sensor.

```
% Input lists
inputExample = {lidarData{1}.Location, 0};

% Create configuration for MEX generation
cfg = coder.config('mex');

% Replace cfg with the following to generate static library and perform
% software-in-the-loop simulation. This requires Embedded Coder license.
%
% cfg = coder.config('lib'); % Static library
% cfg.VerificationMode = 'SIL'; % Software-in-the-loop

% Generate code if file does not exist.
if ~exist('mexLidarTracker_mex','file')
    h = msgbox({'Generating code. This may take a few minutes...';'This message box will close wh
    % -config allows specifying the codegen configuration
    % -o allows specifying the name of the output file
    codegen -config cfg -o mexLidarTracker_mex mexLidarTracker -args inputExample
    close(h);
else
    clear mexLidarTracker_mex;
end
```

Rerun simulation with MEX Code

Rerun the simulation using the generated MEX code, `mexLidarTracker_mex`. Reset time

```
time = 0;

for i = 1:numel(lidarData)
    time = time + dT;

    currentLidar = lidarData{i};

    [detectionsMex,obstacleIndicesMex,groundIndicesMex,croppedIndicesMex,...
        confirmedTracksMex, modelProbsMex] = mexLidarTracker_mex(currentLidar.Location,time);

    % Record data for comparison with MATLAB execution.
    numTracks(i,2) = numel(confirmedTracksMex);
end
```

Compare results between MATLAB and MEX Execution

```
disp(isequal(numTracks(:,1),numTracks(:,2)));
```

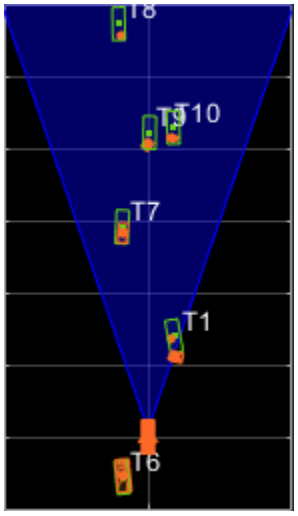
```
   1
```

Notice that the number of confirmed tracks is the same for MATLAB and MEX code execution. This assures that the lidar preprocessing and tracking algorithm returns the same results with generated C code as with the MATLAB code.

Results

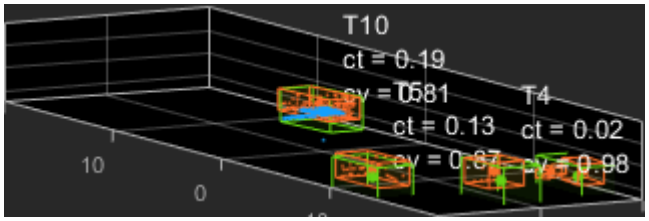
Now, analyze different events in the scenario and understand how the combination of lidar measurement model, joint probabilistic data association, and interacting multiple model filter, helps achieve a good estimation of the vehicle tracks.

Track Maintenance



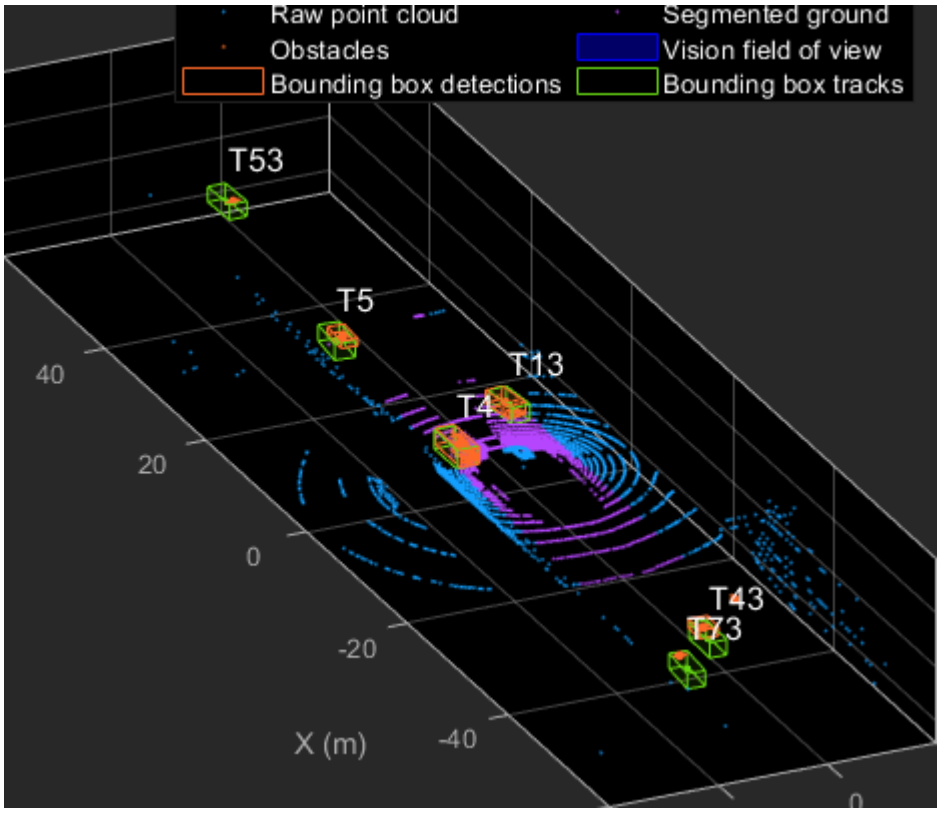
The animation above shows the simulation between time = 3 seconds and time = 16 seconds. Notice that tracks such as T10 and T6 maintain their IDs and trajectory during the time span. However, track T9 is lost because the tracked vehicle was missed (not detected) for a long time by the sensor. Also, notice that the tracked objects are able to maintain their shape and kinematic center by positioning the detections onto the visible portions of the vehicles. For example, as Track T7 moves forward, bounding box detections start to fall on its visible rear portion and the track maintains the actual size of the vehicle. This illustrates the offset and shrinkage effect modeled in the measurement functions.

Capturing Maneuvers



The animation shows that using an IMM filter helps the tracker to maintain tracks on maneuvering vehicles. Notice that the vehicle tracked by T4 changes lanes behind the ego vehicle. The tracker is able maintain a track on the vehicle during this maneuvering event. Also notice in the display that its probability of following the constant turn model, denoted by ct, increases during the lane change maneuver.

Joint Probabilistic Data Association



This animation shows that using a joint probabilistic data association tracker helps in maintaining tracks during ambiguous situations. Here, vehicles tracked by T43 and T73, have a low probability of detection due to their large distance from the sensor. Notice that the tracker is able to maintain tracks during events when one of the vehicles is not detected. During the event, the tracks first coalesce, which is a known phenomenon in JPDA, and then separate as soon as the vehicle was detected again.

Summary

This example showed how to use a JPDA tracker with an IMM filter to track objects using a lidar sensor. You learned how a raw point cloud can be preprocessed to generate detections for conventional trackers, which assume one detection per object per sensor scan. You also learned how to define a cuboid model to describe the kinematics, dimensions, and measurements of extended objects being tracked by the JPDA tracker. In addition, you generated C code from the algorithm and verified its execution results with the MATLAB simulation.

Supporting Files

helperLidarModel

This function defines the lidar model to simulate shrinkage of the bounding box measurement and center-point offset. This function is used in the `helperCvmeasCuboid` and `helperCtmeasCuboid` functions to obtain bounding box measurement from the state.

```
function meas = helperLidarModel(pos,dim,yaw)
% This function returns the expected bounding box measurement given an
% object's position, dimension, and yaw angle.
```

```
% Copyright 2019 The MathWorks, Inc.

% Get x,y and z.
x = pos(1,:);
y = pos(2,:);
z = pos(3,:) - 2; % lidar mounted at height = 2 meters.

% Get spherical measurement.
[az,~,r] = cart2sph(x,y,z);

% Shrink rate
s = 3/50; % 3 meters radial length at 50 meters.
sz = 2/50; % 2 meters height at 50 meters.

% Get length, width and height.
L = dim(1,:);
W = dim(2,:);
H = dim(3,:);

az = az - deg2rad(yaw);

% Shrink length along radial direction.
Lshrink = min(L,abs(s*r.*(cos(az))));
Ls = L - Lshrink;

% Shrink width along radial direction.
Wshrink = min(W,abs(s*r.*(sin(az))));
Ws = W - Wshrink;

% Shrink height.
Hshrink = min(H,sz*r);
Hs = H - Hshrink;

% Similar shift is for x and y directions.
shiftX = Lshrink.*cosd(yaw) + Wshrink.*sind(yaw);
shiftY = Lshrink.*sind(yaw) + Wshrink.*cosd(yaw);
shiftZ = Hshrink;

% Modeling the affect of box origin offset
x = x - sign(x).*shiftX/2;
y = y - sign(y).*shiftY/2;
z = z + shiftZ/2 + 2;

% Measurement format
meas = [x;y;z;yaw;Ls;Ws;Hs];

end
```

helperInverseLidarModel

This function defines the inverse lidar model to initiate a tracking filter using a lidar bounding box measurement. This function is used in the `helperInitIMMFilter` function to obtain state estimates from a bounding box measurement.

```
function [pos,posCov,dim,dimCov,yaw,yawCov] = helperInverseLidarModel(meas,measCov)
```

```
% This function returns the position, dimension, yaw using a bounding
% box measurement.

% Copyright 2019 The MathWorks, Inc.

% Shrink rate.
s = 3/50;
sz = 2/50;

% x,y and z of measurement
x = meas(1,:);
y = meas(2,:);
z = meas(3,:);

[az,~,r] = cart2sph(x,y,z);

% Shift x and y position.
Lshrink = abs(s*r.*(cos(az)));
Wshrink = abs(s*r.*(sin(az)));
Hshrink = sz*r;

shiftX = Lshrink;
shiftY = Wshrink;
shiftZ = Hshrink;

x = x + sign(x).*shiftX/2;
y = y + sign(y).*shiftY/2;
z = z - shiftZ/2;

pos = [x;y;z];
posCov = measCov(1:3,1:3,:);

yaw = meas(4,:);
yawCov = measCov(4,4,:);

% Dimensions are initialized for a standard passenger car with low
% uncertainity.
dim = [4.7;1.8;1.4];
dimCov = 0.01*eye(3);
end
```

HelperBoundingBoxDetector

This is the supporting class `HelperBoundingBoxDetector` to accept a point cloud input and return a list of `objectDetection`

```
classdef HelperBoundingBoxDetector < matlab.System
    % HelperBoundingBoxDetector A helper class to segment the point cloud
    % into bounding box detections.
    % The step call to the object does the following things:
    %
    % 1. Removes point cloud outside the limits.
    % 2. From the survived point cloud, segments out ground
    % 3. From the obstacle point cloud, forms clusters and puts bounding
    %    box on each cluster.
```

```
% Cropping properties
properties
    % XLimits XLimits for the scene
    XLimits = [-70 70];
    % YLimits YLimits for the scene
    YLimits = [-6 6];
    % ZLimits ZLimits fot the scene
    ZLimits = [-2 10];
end

% Ground Segmentation Properties
properties
    % GroundMaxDistance Maximum distance of point to the ground plane
    GroundMaxDistance = 0.3;
    % GroundReferenceVector Reference vector of ground plane
    GroundReferenceVector = [0 0 1];
    % GroundMaxAngularDistance Maximum angular distance of point to reference vector
    GroundMaxAngularDistance = 5;
end

% Bounding box Segmentation properties
properties
    % SegmentationMinDistance Distance threshold for segmentation
    SegmentationMinDistance = 1.6;
    % MinDetectionsPerCluster Minimum number of detections per cluster
    MinDetectionsPerCluster = 2;
    % MaxZDistanceCluster Maximum Z-coordinate of cluster
    MaxZDistanceCluster = 3;
    % MinZDistanceCluster Minimum Z-coordinate of cluster
    MinZDistanceCluster = -3;
end

% Ego vehicle radius to remove ego vehicle point cloud.
properties
    % EgoVehicleRadius Radius of ego vehicle
    EgoVehicleRadius = 3;
end

properties
    % MeasurementNoise Measurement noise for the bounding box detection
    MeasurementNoise = blkdiag(eye(3),10,eye(3));
end

properties (Nontunable)
    MeasurementParameters = struct.empty(0,1);
end

methods
    function obj = HelperBoundingBoxDetector(varargin)
        setProperties(obj,nargin,varargin{:})
    end
end

methods (Access = protected)
    function [bboxDets,obstacleIndices,groundIndices,croppedIndices] = stepImpl(obj,currentPo
        % Crop point cloud
        [pcSurvived,survivedIndices,croppedIndices] = cropPointCloud(currentPointCloud,obj.XL
        % Remove ground plane
```

```
            [pcObstacles,obstacleIndices,groundIndices] = removeGroundPlane(pcSurvived,obj.Ground
            % Form clusters and get bounding boxes
            detBBoxes = getBoundingBoxes(pcObstacles,obj.SegmentationMinDistance,obj.MinDetectio
            % Assemble detections
            if isempty(obj.MeasurementParameters)
                measParams = {};
            else
                measParams = obj.MeasurementParameters;
            end
            bboxDets = assembleDetections(detBBoxes,obj.MeasurementNoise,measParams,time);
        end
    end
end

function detections = assembleDetections(bboxes,measNoise,measParams,time)
% This method assembles the detections in objectDetection format.
numBoxes = size(bboxes,2);
detections = cell(numBoxes,1);
for i = 1:numBoxes
    detections{i} = objectDetection(time,cast(bboxes(:,i),'double'),...
        'MeasurementNoise',double(measNoise),'ObjectAttributes',struct,...
        'MeasurementParameters',measParams);
end
end

function bboxes = getBoundingBoxes(ptCloud,minDistance,minDetsPerCluster,maxZDistance,minZDistan
    % This method fits bounding boxes on each cluster with some basic
    % rules.
    % Cluster must have at least minDetsPerCluster points.
    % Its mean z must be between maxZDistance and minZDistance.
    % length, width and height are calculated using min and max from each
    % dimension.
    [labels,numClusters] = pcsegdist(ptCloud,minDistance);
    pointData = ptCloud.Location;
    bboxes = nan(7,numClusters,'like',pointData);
    isValidCluster = false(1,numClusters);
    for i = 1:numClusters
        thisPointData = pointData(labels == i,:);
        meanPoint = mean(thisPointData,1);
        if size(thisPointData,1) > minDetsPerCluster && ...
                meanPoint(3) < maxZDistance && meanPoint(3) > minZDistance
            cuboid = pcfitcuboid(pointCloud(thisPointData));
            yaw = cuboid.Orientation(3);
            L = cuboid.Dimensions(1);
            W = cuboid.Dimensions(2);
            H = cuboid.Dimensions(3);
            if abs(yaw) > 45
                possibles = yaw + [-90;90];
                [~,toChoose] = min(abs(possibles));
                yaw = possibles(toChoose);
                temp = L;
                L = W;
                W = temp;
            end
            bboxes(:,i) = [cuboid.Center yaw L W H]';
            isValidCluster(i) = L < 20 & W < 20;
        end
    end
```

```
    bboxes = bboxes(:,isValidCluster);
end

function [ptCloudOut,obstacleIndices,groundIndices] = removeGroundPlane(ptCloudIn,maxGroundDist,
    % This method removes the ground plane from point cloud using
    % pcfitplane.
    [~,groundIndices,outliers] = pcfitplane(ptCloudIn,maxGroundDist,referenceVector,maxAngularDis
    ptCloudOut = select(ptCloudIn,outliers);
    obstacleIndices = currentIndices(outliers);
    groundIndices = currentIndices(groundIndices);
end

function [ptCloudOut,indices,croppedIndices] = cropPointCloud(ptCloudIn,xLim,yLim,zLim,egoVehicle
    % This method selects the point cloud within limits and removes the
    % ego vehicle point cloud using findNeighborsInRadius
    locations = ptCloudIn.Location;
    locations = reshape(locations,[],3);
    insideX = locations(:,1) < xLim(2) & locations(:,1) > xLim(1);
    insideY = locations(:,2) < yLim(2) & locations(:,2) > yLim(1);
    insideZ = locations(:,3) < zLim(2) & locations(:,3) > zLim(1);
    inside = insideX & insideY & insideZ;

    % Remove ego vehicle
    nearIndices = findNeighborsInRadius(ptCloudIn,[0 0 0],egoVehicleRadius);
    nonEgoIndices = true(ptCloudIn.Count,1);
    nonEgoIndices(nearIndices) = false;
    validIndices = inside & nonEgoIndices;
    indices = find(validIndices);
    croppedIndices = find(~validIndices);
    ptCloudOut = select(ptCloudIn,indices);
end
```

mexLidarTracker

This function implements the point cloud preprocessing display and the tracking algorithm using a functional interface for code generation.

```
function [detections,obstacleIndices,groundIndices,croppedIndices,...
    confirmedTracks, modelProbs] = mexLidarTracker(ptCloudLocations,time)

persistent detectorModel tracker detectableTracksInput currentNumTracks


if isempty(detectorModel) || isempty(tracker) || isempty(detectableTracksInput) || isempty(curre

    % Use the same starting seed as MATLAB to reproduce results in SIL
    % simulation.
    rng(2018);

    % A bounding box detector model.
    detectorModel = HelperBoundingBoxDetector(...
                    'XLimits',[-50 75],...              % min-max
                    'YLimits',[-5 5],...                % min-max
                    'ZLimits',[-2 5],...                % min-max
```

```
                    'SegmentationMinDistance',1.8,...   % minimum Euclidian distance
                    'MinDetectionsPerCluster',1,...     % minimum points per cluster
                    'MeasurementNoise',blkdiag(0.25*eye(3),25,eye(3)),...       % measurement noi
                    'GroundMaxDistance',0.3);           % maximum distance of ground points from

    assignmentGate = [75 1000]; % Assignment threshold;
    confThreshold = [7 10];    % Confirmation threshold for history logic
    delThreshold = [8 10];     % Deletion threshold for history logic
    Kc = 1e-9;                 % False-alarm rate per unit volume

    filterInitFcn = @helperInitIMMFilter;

    tracker = trackerJPDA('FilterInitializationFcn',filterInitFcn,...
                      'TrackLogic','History',...
                      'AssignmentThreshold',assignmentGate,...
                      'ClutterDensity',Kc,...
                      'ConfirmationThreshold',confThreshold,...
                      'DeletionThreshold',delThreshold,...
                      'HasDetectableTrackIDsInput',true,...
                      'InitializationThreshold',0,...
                      'MaxNumTracks',30,...
                      'HitMissThreshold',0.1);

    detectableTracksInput = zeros(tracker.MaxNumTracks,2);

    currentNumTracks = 0;
end

ptCloud = pointCloud(ptCloudLocations);

% Detector model
[detections,obstacleIndices,groundIndices,croppedIndices] = detectorModel(ptCloud,time);

% Call tracker
[confirmedTracks,~,allTracks] = tracker(detections,time,detectableTracksInput(1:currentNumTracks,
% Update the detectability input
currentNumTracks = numel(allTracks);
detectableTracksInput(1:currentNumTracks,:) = helperCalcDetectability(allTracks,[1 3 6]);

% Get model probabilities
modelProbs = zeros(2,numel(confirmedTracks));
if isLocked(tracker)
    for k = 1:numel(confirmedTracks)
        c1 = getTrackFilterProperties(tracker,confirmedTracks(k).TrackID,'ModelProbabilities');
        probs = c1{1};
        modelProbs(1,k) = probs(1);
        modelProbs(2,k) = probs(2);
    end
end

end
```

helperCalcDetectability

The function calculate the probability of detection for each track. This function is used to generate the "DetectableTracksIDs" input for the `trackerJPDA`.

```
function detectableTracksInput = helperCalcDetectability(tracks,posIndices)
% This is a helper function to calculate the detection probability of
% tracks for the lidar tracking example. It may be removed in a future
% release.

% Copyright 2019 The MathWorks, Inc.

% The bounding box detector has low probability of segmenting point clouds
% into bounding boxes are distances greater than 40 meters. This function
% models this effect using a state-dependent probability of detection for
% each tracker. After a maximum range, the Pd is set to a high value to
% enable deletion of track at a faster rate.
if isempty(tracks)
    detectableTracksInput = zeros(0,2);
    return;
end
rMax = 75;
rAmbig = 40;
stateSize = numel(tracks(1).State);
posSelector = zeros(3,stateSize);
posSelector(1,posIndices(1)) = 1;
posSelector(2,posIndices(2)) = 1;
posSelector(3,posIndices(3)) = 1;
pos = getTrackPositions(tracks,posSelector);
if coder.target('MATLAB')
    trackIDs = [tracks.TrackID];
else
    trackIDs = zeros(1,numel(tracks),'uint32');
    for i = 1:numel(tracks)
        trackIDs(i) = tracks(i).TrackID;
    end
end
[~,~,r] = cart2sph(pos(:,1),pos(:,2),pos(:,3));
probDetection = 0.9*ones(numel(tracks),1);
probDetection(r > rAmbig) = 0.4;
probDetection(r > rMax) = 0.99;
detectableTracksInput = [double(trackIDs(:)) probDetection(:)];
end
```

loadLidarAndImageData

Stitches Lidar and Camera data for processing using initial and final time specified.

```
function [lidarData,imageData] = loadLidarAndImageData(datasetFolder,initTime,finalTime)
initFrame = max(1,floor(initTime*10));
lastFrame = min(350,ceil(finalTime*10));
load (fullfile(datasetFolder,'imageData_35seconds.mat'),'allImageData');
imageData = allImageData(initFrame:lastFrame);

numFrames = lastFrame - initFrame + 1;
lidarData = cell(numFrames,1);

% Each file contains 70 frames.
initFileIndex = floor(initFrame/70) + 1;
lastFileIndex = ceil(lastFrame/70);
```

```
frameIndices = [1:70:numFrames numFrames + 1];

counter = 1;
for i = initFileIndex:lastFileIndex
    startFrame = frameIndices(counter);
    endFrame = frameIndices(counter + 1) - 1;
    load(fullfile(datasetFolder,['lidarData_',num2str(i)]),'currentLidarData');
    lidarData(startFrame:endFrame) = currentLidarData(1:(endFrame + 1 - startFrame));
    counter = counter + 1;
end
end
```

References

[1] Arya Senna Abdul Rachman, Arya. "3D-LIDAR Multi Object Tracking for Autonomous Driving: Multi-target Detection and Tracking under Urban Road Uncertainties." (2017).

See Also

`pointCloud` | `trackerJPDA` | `trackingIMM`

More About

- “Ground Plane and Obstacle Detection Using Lidar” on page 7-107
- “Extended Object Tracking of Highway Vehicles with Radar and Camera” on page 7-234
- “Track Vehicles Using Lidar Data in Simulink” on page 7-310
- “Detect, Classify, and Track Vehicles Using Lidar” (Lidar Toolbox)

Sensor Fusion Using Synthetic Radar and Vision Data

This example shows how to generate a scenario, simulate sensor detections, and use sensor fusion to track simulated vehicles. The main benefit of using scenario generation and sensor simulation over sensor recording is the ability to create rare and potentially dangerous events and test the vehicle algorithms with them.

This example covers the entire programmatic workflow for generating synthetic data. To generate synthetic data interactively instead, use the Driving Scenario Designer app. For an example, see “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2.

Generate the Scenario

Scenario generation comprises generating a road network, defining vehicles that move on the roads, and moving the vehicles.

In this example, you test the ability of the sensor fusion to track a vehicle that is passing on the left of the ego vehicle. The scenario simulates a highway setting, and additional vehicles are in front of and behind the ego vehicle.

```
% Define an empty scenario.
scenario = drivingScenario;
scenario.SampleTime = 0.01;
```

Add a stretch of 500 meters of typical highway road with two lanes. The road is defined using a set of points, where each point defines the center of the road in 3-D space.

```
roadCenters = [0 0; 50 0; 100 0; 250 20; 500 40];
road(scenario, roadCenters, 'lanes',lanespec(2));
```

Create the ego vehicle and three cars around it: one that overtakes the ego vehicle and passes it on the left, one that drives right in front of the ego vehicle and one that drives right behind the ego vehicle. All the cars follow the trajectory defined by the road waypoints by using the `trajectory` driving policy. The passing car will start on the right lane, move to the left lane to pass, and return to the right lane.

```
% Create the ego vehicle that travels at 25 m/s along the road.  Place the
% vehicle on the right lane by subtracting off half a lane width (1.8 m)
% from the centerline of the road.
egoCar = vehicle(scenario, 'ClassID', 1);
trajectory(egoCar, roadCenters(2:end,:) - [0 1.8], 25); % On right lane

% Add a car in front of the ego vehicle
leadCar = vehicle(scenario, 'ClassID', 1);
trajectory(leadCar, [70 0; roadCenters(3:end,:)] - [0 1.8], 25); % On right lane

% Add a car that travels at 35 m/s along the road and passes the ego vehicle
passingCar = vehicle(scenario, 'ClassID', 1);
waypoints = [0 -1.8; 50 1.8; 100 1.8; 250 21.8; 400 32.2; 500 38.2];
trajectory(passingCar, waypoints, 35);

% Add a car behind the ego vehicle
chaseCar = vehicle(scenario, 'ClassID', 1);
trajectory(chaseCar, [25 0; roadCenters(2:end,:)] - [0 1.8], 25); % On right lane
```

Define Radar and Vision Sensors

In this example, you simulate an ego vehicle that has 6 radar sensors and 2 vision sensors covering the 360 degrees field of view. The sensors have some overlap and some coverage gap. The ego vehicle is equipped with a long-range radar sensor and a vision sensor on both the front and the back of the vehicle. Each side of the vehicle has two short-range radar sensors, each covering 90 degrees. One sensor on each side covers from the middle of the vehicle to the back. The other sensor on each side covers from the middle of the vehicle forward. The figure in the next section shows the coverage.

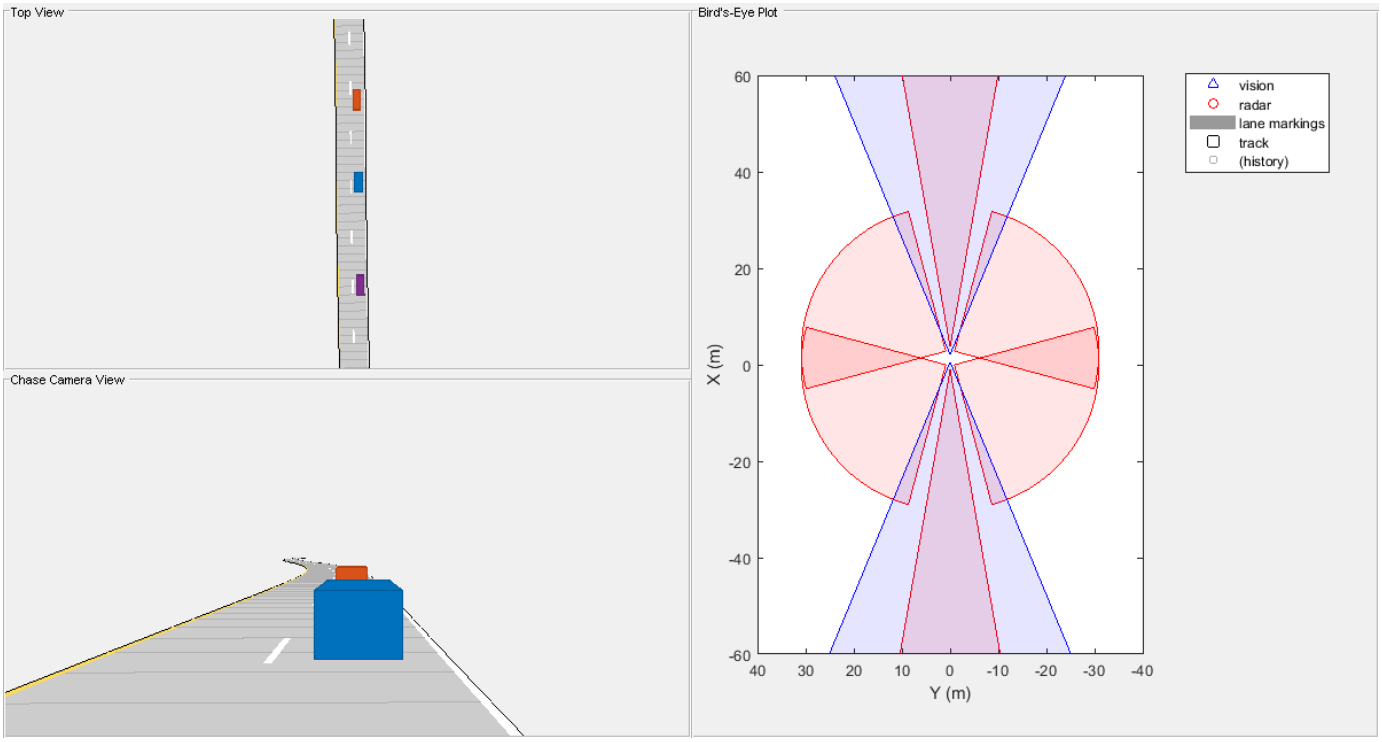
```
sensors = cell(8,1);
% Front-facing long-range radar sensor at the center of the front bumper of the car.
sensors{1} = radarDetectionGenerator('SensorIndex', 1, 'Height', 0.2, 'MaxRange', 174, ...
    'SensorLocation', [egoCar.Wheelbase + egoCar.FrontOverhang, 0], 'FieldOfView', [20, 5]);

% Rear-facing long-range radar sensor at the center of the rear bumper of the car.
sensors{2} = radarDetectionGenerator('SensorIndex', 2, 'Height', 0.2, 'Yaw', 180, ...
    'SensorLocation', [-egoCar.RearOverhang, 0], 'MaxRange', 174, 'FieldOfView', [20, 5]);

% Rear-left-facing short-range radar sensor at the left rear wheel well of the car.
sensors{3} = radarDetectionGenerator('SensorIndex', 3, 'Height', 0.2, 'Yaw', 120, ...
    'SensorLocation', [0, egoCar.Width/2], 'MaxRange', 30, 'ReferenceRange', 50, ...
    'FieldOfView', [90, 5], 'AzimuthResolution', 10, 'RangeResolution', 1.25);

% Rear-right-facing short-range radar sensor at the right rear wheel well of the car.
sensors{4} = radarDetectionGenerator('SensorIndex', 4, 'Height', 0.2, 'Yaw', -120, ...
    'SensorLocation', [0, -egoCar.Width/2], 'MaxRange', 30, 'ReferenceRange', 50, ...
    'FieldOfView', [90, 5], 'AzimuthResolution', 10, 'RangeResolution', 1.25);

% Front-left-facing short-range radar sensor at the left front wheel well of the car.
sensors{5} = radarDetectionGenerator('SensorIndex', 5, 'Height', 0.2, 'Yaw', 60, ...
    'SensorLocation', [egoCar.Wheelbase, egoCar.Width/2], 'MaxRange', 30, ...
    'ReferenceRange', 50, 'FieldOfView', [90, 5], 'AzimuthResolution', 10, ...
    'RangeResolution', 1.25);

% Front-right-facing short-range radar sensor at the right front wheel well of the car.
sensors{6} = radarDetectionGenerator('SensorIndex', 6, 'Height', 0.2, 'Yaw', -60, ...
    'SensorLocation', [egoCar.Wheelbase, -egoCar.Width/2], 'MaxRange', 30, ...
    'ReferenceRange', 50, 'FieldOfView', [90, 5], 'AzimuthResolution', 10, ...
    'RangeResolution', 1.25);

% Front-facing camera located at front windshield.
sensors{7} = visionDetectionGenerator('SensorIndex', 7, 'FalsePositivesPerImage', 0.1, ...
    'SensorLocation', [0.75*egoCar.Wheelbase 0], 'Height', 1.1);

% Rear-facing camera located at rear windshield.
sensors{8} = visionDetectionGenerator('SensorIndex', 8, 'FalsePositivesPerImage', 0.1, ...
    'SensorLocation', [0.2*egoCar.Wheelbase 0], 'Height', 1.1, 'Yaw', 180);

% Register actor profiles with the sensors.
profiles = actorProfiles(scenario);
for m = 1:numel(sensors)
    sensors{m}.ActorProfiles = profiles;
end
```

Create a Tracker

Create a `multiObjectTracker` to track the vehicles that are close to the ego vehicle. The tracker uses the `initSimDemoFilter` supporting function to initialize a constant velocity linear Kalman filter that works with position and velocity.

Tracking is done in 2-D. Although the sensors return measurements in 3-D, the motion itself is confined to the horizontal plane, so there is no need to track the height.

```
tracker = multiObjectTracker('FilterInitializationFcn', @initSimDemoFilter, ...
    'AssignmentThreshold', 30, 'ConfirmationThreshold', [4 5]);
positionSelector = [1 0 0 0; 0 0 1 0]; % Position selector
velocitySelector = [0 1 0 0; 0 0 0 1]; % Velocity selector

% Create the display and return a handle to the bird's-eye plot
BEP = createDemoDisplay(egoCar, sensors);
```



Simulate the Scenario

The following loop moves the vehicles, calls the sensor simulation, and performs the tracking.

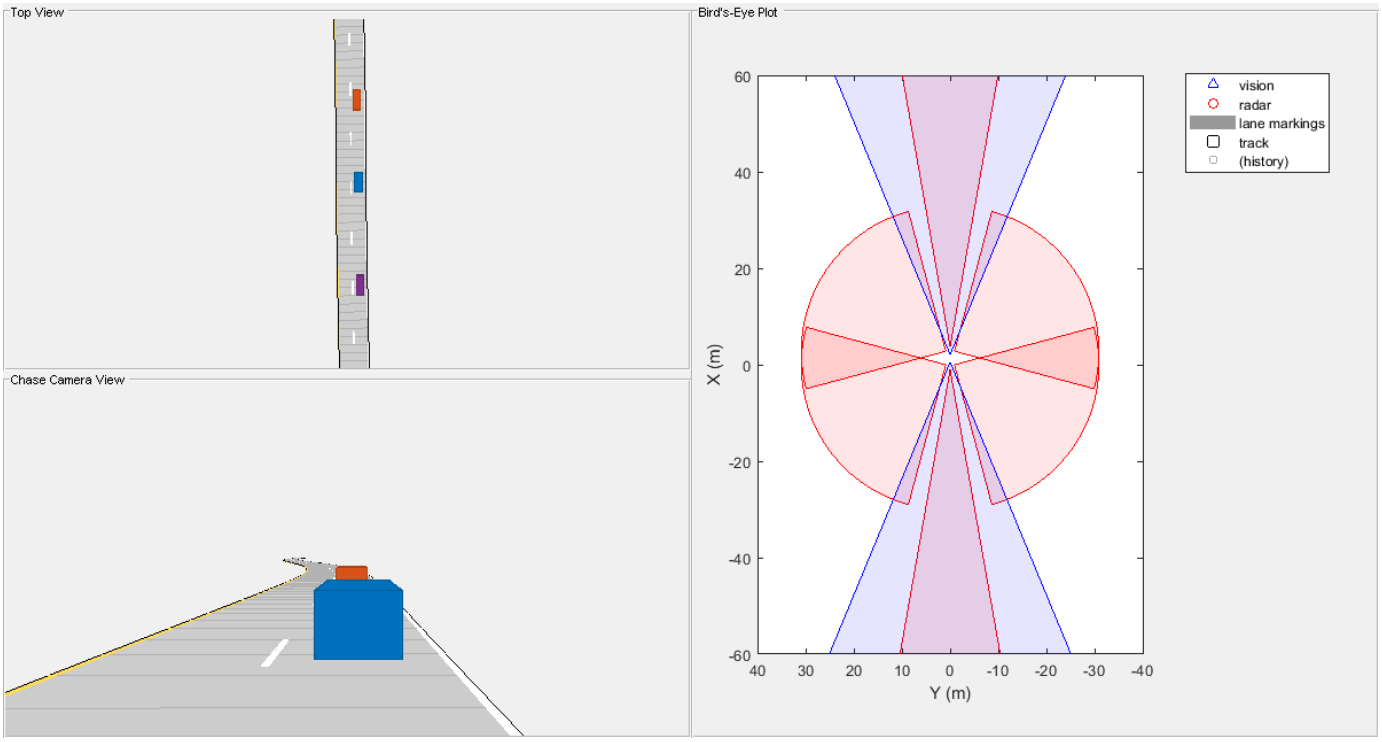
Note that the scenario generation and sensor simulation can have different time steps. Specifying different time steps for the scenario and the sensors enables you to decouple the scenario simulation from the sensor simulation. This is useful for modeling actor motion with high accuracy independently from the sensor's measurement rate.

Another example is when the sensors have different update rates. Suppose one sensor provides updates every 20 milliseconds and another sensor provides updates every 50 milliseconds. You can specify the scenario with an update rate of 10 milliseconds and the sensors will provide their updates at the correct time.

In this example, the scenario generation has a time step of 0.01 second, while the sensors detect every 0.1 second. The sensors return a logical flag, `isValidTime`, that is true if the sensors generated detections. This flag is used to call the tracker only when there are detections.

Another important note is that the sensors can simulate multiple detections per target, in particular when the targets are very close to the radar sensors. Because the tracker assumes a single detection per target from each sensor, you must cluster the detections before the tracker processes them. This is done by the function `clusterDetections`. See the 'Supporting Functions' section.

```
toSnap = true;
while advance(scenario) && ishghandle(BEP.Parent)
    % Get the scenario time
    time = scenario.SimulationTime;

    % Get the position of the other vehicle in ego vehicle coordinates
    ta = targetPoses(egoCar);

    % Simulate the sensors
    detections = {};
    isValidTime = false(1,8);
    for i = 1:8
        [sensorDets,numValidDets,isValidTime(i)] = sensors{i}(ta, time);
        if numValidDets
            for j = 1:numValidDets
                % Vision detections do not report SNR. The tracker requires
                % that they have the same object attributes as the radar
                % detections. This adds the SNR object attribute to vision
                % detections and sets it to a NaN.
                if ~isfield(sensorDets{j}.ObjectAttributes{1}, 'SNR')
                    sensorDets{j}.ObjectAttributes{1}.SNR = NaN;
                end
            end
            detections = [detections; sensorDets]; %#ok<AGROW>
        end
    end

    % Update the tracker if there are new detections
    if any(isValidTime)
        vehicleLength = sensors{1}.ActorProfiles.Length;
        detectionClusters = clusterDetections(detections, vehicleLength);
        confirmedTracks = updateTracks(tracker, detectionClusters, time);

        % Update bird's-eye plot
        updateBEP(BEP, egoCar, detections, confirmedTracks, positionSelector, velocitySelector);
    end

    % Snap a figure for the document when the car passes the ego vehicle
    if ta(1).Position(1) > 0 && toSnap
        toSnap = false;
        snapnow
    end
end
```



Summary

This example shows how to generate a scenario, simulate sensor detections, and use these detections to track moving vehicles around the ego vehicle.

You can try to modify the scenario road, or add or remove vehicles. You can also try to add, remove, or modify the sensors on the ego vehicle, or modify the tracker parameters.

Supporting Functions

`initSimDemoFilter`

This function initializes a constant velocity filter based on a detection.

```
function filter = initSimDemoFilter(detection)
% Use a 2-D constant velocity model to initialize a trackingKF filter.
% The state vector is [x;vx;y;vy]
% The detection measurement vector is [x;y;vx;vy]
% As a result, the measurement model is H = [1 0 0 0; 0 0 1 0; 0 1 0 0; 0 0 0 1]
H = [1 0 0 0; 0 0 1 0; 0 1 0 0; 0 0 0 1];
filter = trackingKF('MotionModel', '2D Constant Velocity', ...
    'State', H' * detection.Measurement, ...
    'MeasurementModel', H, ...
    'StateCovariance', H' * detection.MeasurementNoise * H, ...
    'MeasurementNoise', detection.MeasurementNoise);
end
```

`clusterDetections`

This function merges multiple detections suspected to be of the same vehicle to a single detection. The function looks for detections that are closer than the size of a vehicle. Detections that fit this

criterion are considered a cluster and are merged to a single detection at the centroid of the cluster. The measurement noises are modified to represent the possibility that each detection can be anywhere on the vehicle. Therefore, the noise should have the same size as the vehicle size.

In addition, this function removes the third dimension of the measurement (the height) and reduces the measurement vector to [x;y;vx;vy].

```
function detectionClusters = clusterDetections(detections, vehicleSize)
N = numel(detections);
distances = zeros(N);
for i = 1:N
    for j = i+1:N
        if detections{i}.SensorIndex == detections{j}.SensorIndex
            distances(i,j) = norm(detections{i}.Measurement(1:2) - detections{j}.Measurement(1:2
        else
            distances(i,j) = inf;
        end
    end
end
leftToCheck = 1:N;
i = 0;
detectionClusters = cell(N,1);
while ~isempty(leftToCheck)
    % Remove the detections that are in the same cluster as the one under
    % consideration
    underConsideration = leftToCheck(1);
    clusterInds = (distances(underConsideration, leftToCheck) < vehicleSize);
    detInds = leftToCheck(clusterInds);
    clusterDets = [detections{detInds}];
    clusterMeas = [clusterDets.Measurement];
    meas = mean(clusterMeas, 2);
    meas2D = [meas(1:2);meas(4:5)];
    i = i + 1;
    detectionClusters{i} = detections{detInds(1)};
    detectionClusters{i}.Measurement = meas2D;
    leftToCheck(clusterInds) = [];
end
detectionClusters(i+1:end) = [];

% Since the detections are now for clusters, modify the noise to represent
% that they are of the whole car
for i = 1:numel(detectionClusters)
    measNoise(1:2,1:2) = vehicleSize^2 * eye(2);
    measNoise(3:4,3:4) = eye(2) * 100 * vehicleSize^2;
    detectionClusters{i}.MeasurementNoise = measNoise;
end
end
```

`createDemoDisplay`

This function creates a three-panel display:

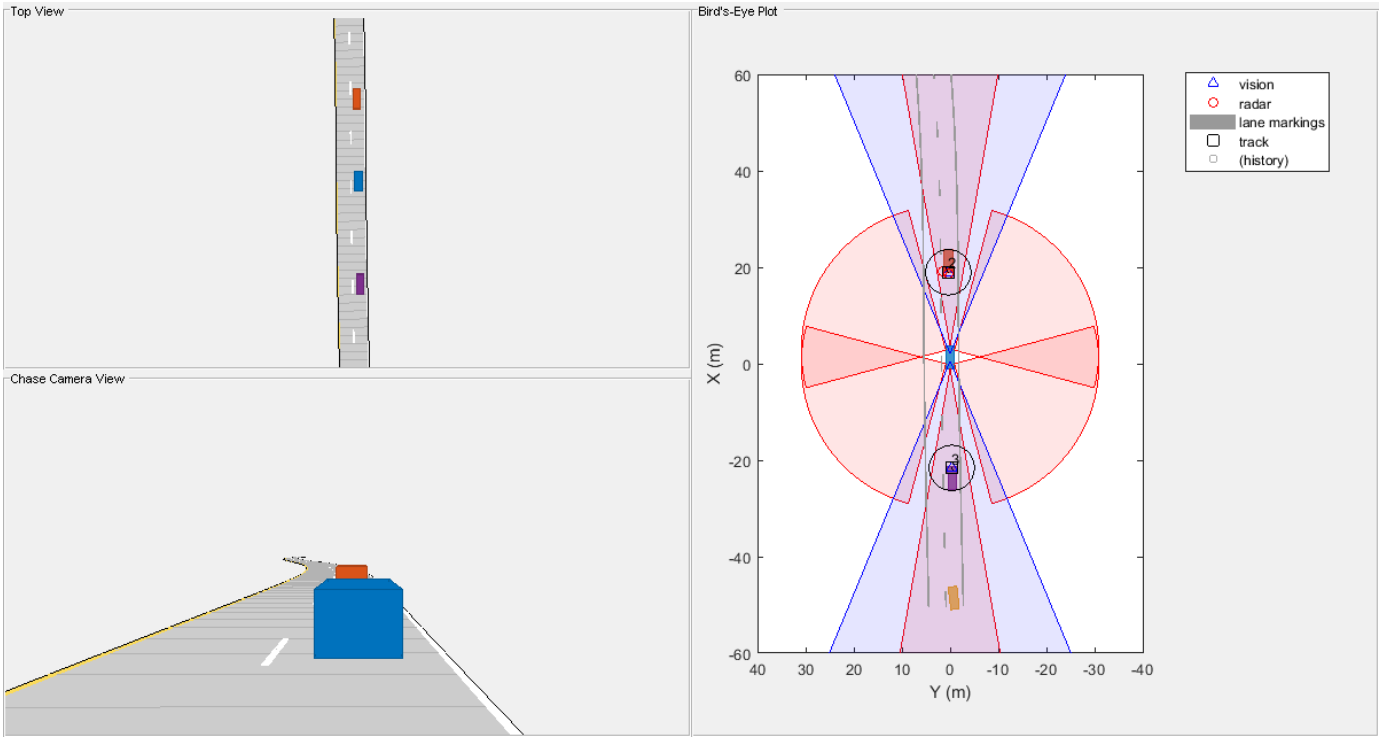
1. Top-left corner of display: A top view that follows the ego vehicle.
2. Bottom-left corner of display: A chase-camera view that follows the ego vehicle.
3. Right-half of display: A `birdsEyePlot` display.

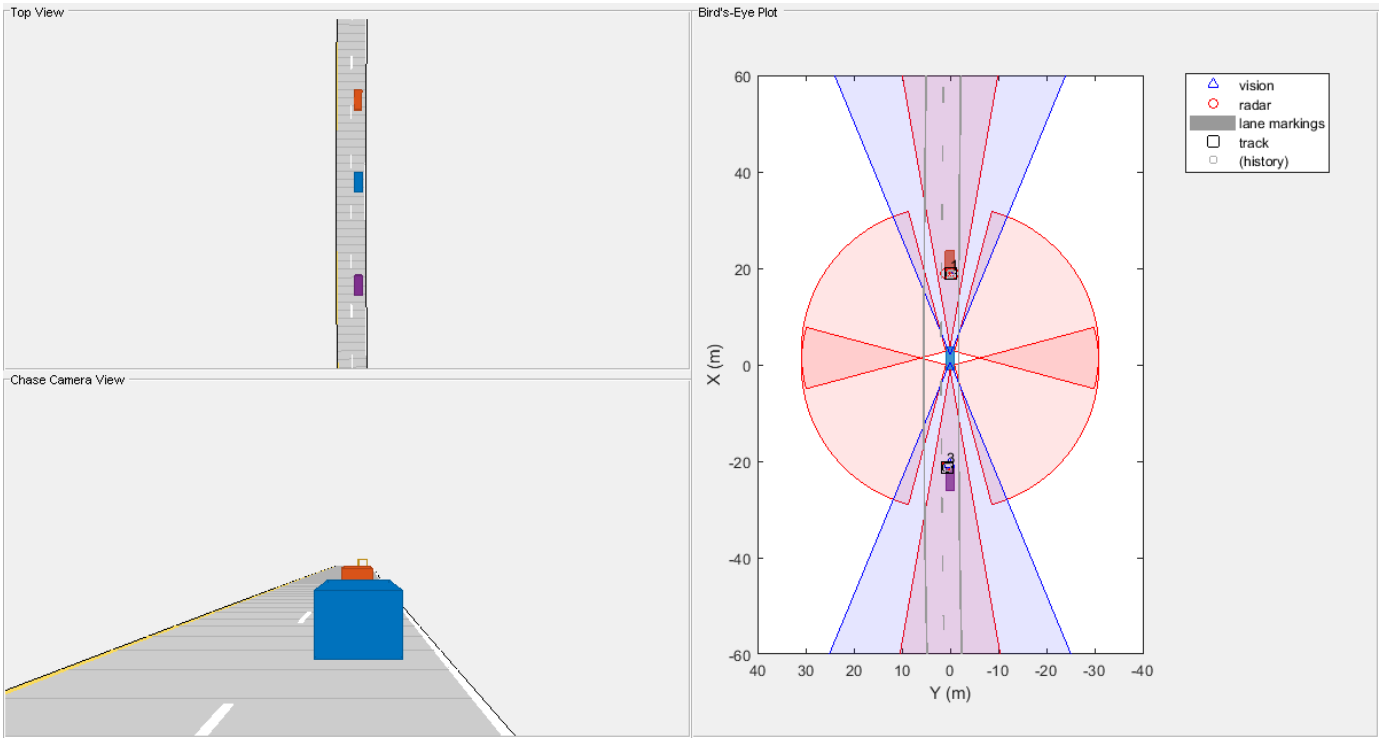
```
function BEP = createDemoDisplay(egoCar, sensors)
    % Make a figure
    hFigure = figure('Position', [0, 0, 1200, 640], 'Name', 'Sensor Fusion with Synthetic Data Ex
    movegui(hFigure, [0 -1]); % Moves the figure to the left and a little down from the top

    % Add a car plot that follows the ego vehicle from behind
    hCarViewPanel = uipanel(hFigure, 'Position', [0 0 0.5 0.5], 'Title', 'Chase Camera View');
    hCarPlot = axes(hCarViewPanel);
    chasePlot(egoCar, 'Parent', hCarPlot);

    % Add a car plot that follows the ego vehicle from a top view
    hTopViewPanel = uipanel(hFigure, 'Position', [0 0.5 0.5 0.5], 'Title', 'Top View');
    hCarPlot = axes(hTopViewPanel);
    chasePlot(egoCar, 'Parent', hCarPlot, 'ViewHeight', 130, 'ViewLocation', [0 0], 'ViewPitch',

    % Add a panel for a bird's-eye plot
    hBEVPanel = uipanel(hFigure, 'Position', [0.5 0 0.5 1], 'Title', 'Bird''s-Eye Plot');

    % Create bird's-eye plot for the ego vehicle and sensor coverage
    hBEVPlot = axes(hBEVPanel);
    frontBackLim = 60;
    BEP = birdsEyePlot('Parent', hBEVPlot, 'Xlimits', [-frontBackLim frontBackLim], 'Ylimits', [-

    % Plot the coverage areas for radars
    for i = 1:6
        cap = coverageAreaPlotter(BEP,'FaceColor','red','EdgeColor','red');
        plotCoverageArea(cap, sensors{i}.SensorLocation,...
            sensors{i}.MaxRange, sensors{i}.Yaw, sensors{i}.FieldOfView(1));
    end

    % Plot the coverage areas for vision sensors
    for i = 7:8
        cap = coverageAreaPlotter(BEP,'FaceColor','blue','EdgeColor','blue');
        plotCoverageArea(cap, sensors{i}.SensorLocation,...
            sensors{i}.MaxRange, sensors{i}.Yaw, 45);
    end

    % Create a vision detection plotter put it in a struct for future use
    detectionPlotter(BEP, 'DisplayName','vision', 'MarkerEdgeColor','blue', 'Marker','^');

    % Combine all radar detections into one entry and store it for later update
    detectionPlotter(BEP, 'DisplayName','radar', 'MarkerEdgeColor','red');

    % Add road borders to plot
    laneMarkingPlotter(BEP, 'DisplayName','lane markings');

    % Add the tracks to the bird's-eye plot. Show last 10 track updates.
    trackPlotter(BEP, 'DisplayName','track', 'HistoryDepth',10);

    axis(BEP.Parent, 'equal');
    xlim(BEP.Parent, [-frontBackLim frontBackLim]);
    ylim(BEP.Parent, [-40 40]);

    % Add an outline plotter for ground truth
    outlinePlotter(BEP, 'Tag', 'Ground truth');
end
```

updateBEP

This function updates the bird's-eye plot with road boundaries, detections, and tracks.

```
function updateBEP(BEP, egoCar, detections, confirmedTracks, psel, vsel)
    % Update road boundaries and their display
    [lmv, lmf] = laneMarkingVertices(egoCar);
    plotLaneMarking(findPlotter(BEP,'DisplayName','lane markings'),lmv,lmf);

    % update ground truth data
    [position, yaw, length, width, originOffset, color] = targetOutlines(egoCar);
    plotOutline(findPlotter(BEP,'Tag','Ground truth'), position, yaw, length, width, 'OriginOffse

    % Prepare and update detections display
    N = numel(detections);
    detPos = zeros(N,2);
    isRadar = true(N,1);
    for i = 1:N
        detPos(i,:) = detections{i}.Measurement(1:2)';
        if detections{i}.SensorIndex > 6 % Vision detections
            isRadar(i) = false;
        end
    end
    plotDetection(findPlotter(BEP,'DisplayName','vision'), detPos(~isRadar,:));
    plotDetection(findPlotter(BEP,'DisplayName','radar'), detPos(isRadar,:));

    % Prepare and update tracks display
    trackIDs = {confirmedTracks.TrackID};
    labels = cellfun(@num2str, trackIDs, 'UniformOutput', false);
    [tracksPos, tracksCov] = getTrackPositions(confirmedTracks, psel);
    tracksVel = getTrackVelocities(confirmedTracks, vsel);
    plotTrack(findPlotter(BEP,'DisplayName','track'), tracksPos, tracksVel, tracksCov, labels);
end
```





See Also

Apps
Driving Scenario Designer

Objects
`birdsEyePlot` | `drivingScenario` | `multiObjectTracker` | `radarDetectionGenerator` | `visionDetectionGenerator`

Functions
`targetPoses` | `trajectory` | `updateTracks` | `vehicle`

More About

- “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” on page 7-205

Sensor Fusion Using Synthetic Radar and Vision Data in Simulink

This example shows how to implement a synthetic data simulation for tracking and sensor fusion in Simulink® with Automated Driving Toolbox™. It closely follows the “Sensor Fusion Using Synthetic Radar and Vision Data” on page 7-196 MATLAB® example.

Introduction

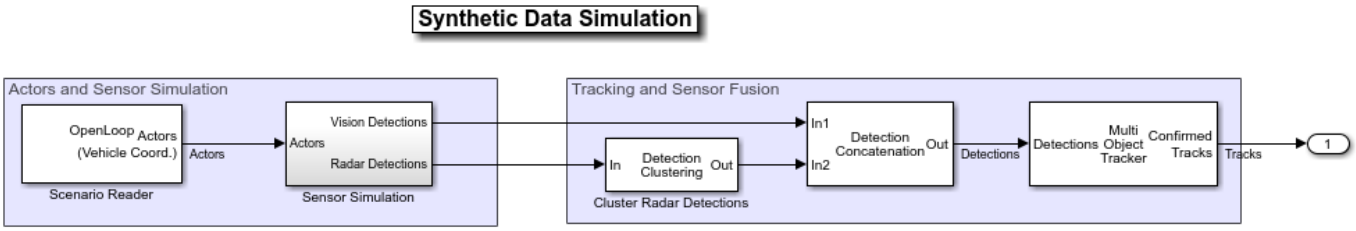
Simulating synthetic radar and vision detections provides the ability to create rare and potentially dangerous events and test the vehicle algorithms with them. This example covers the entire synthetic data workflow in Simulink.

Setup and Overview of the Model

Prior to running this example, the Driving Scenario Designer app was used to create the same scenario defined in “Sensor Fusion Using Synthetic Radar and Vision Data” on page 7-196. The roads and actors from this scenario were then saved to the scenario file `OpenLoop.mat`.

The Scenario Reader block reads the actor pose data from the saved file. The block converts the actor poses from the world coordinates of the scenario into ego vehicle coordinates. The actor poses are streamed on a bus generated by the block.

The actor poses are used by the Sensor Simulation subsystem, which generates synthetic radar and vision detections. The simulated detections are concatenated at the input to the Multi-Object Tracker block, whose output is a list of confirmed tracks. Finally, the Bird's-Eye Scope visualizes the actors, the vision and radar detections, the confirmed tracks and the road boundaries. The following sections describe the main blocks of this model.

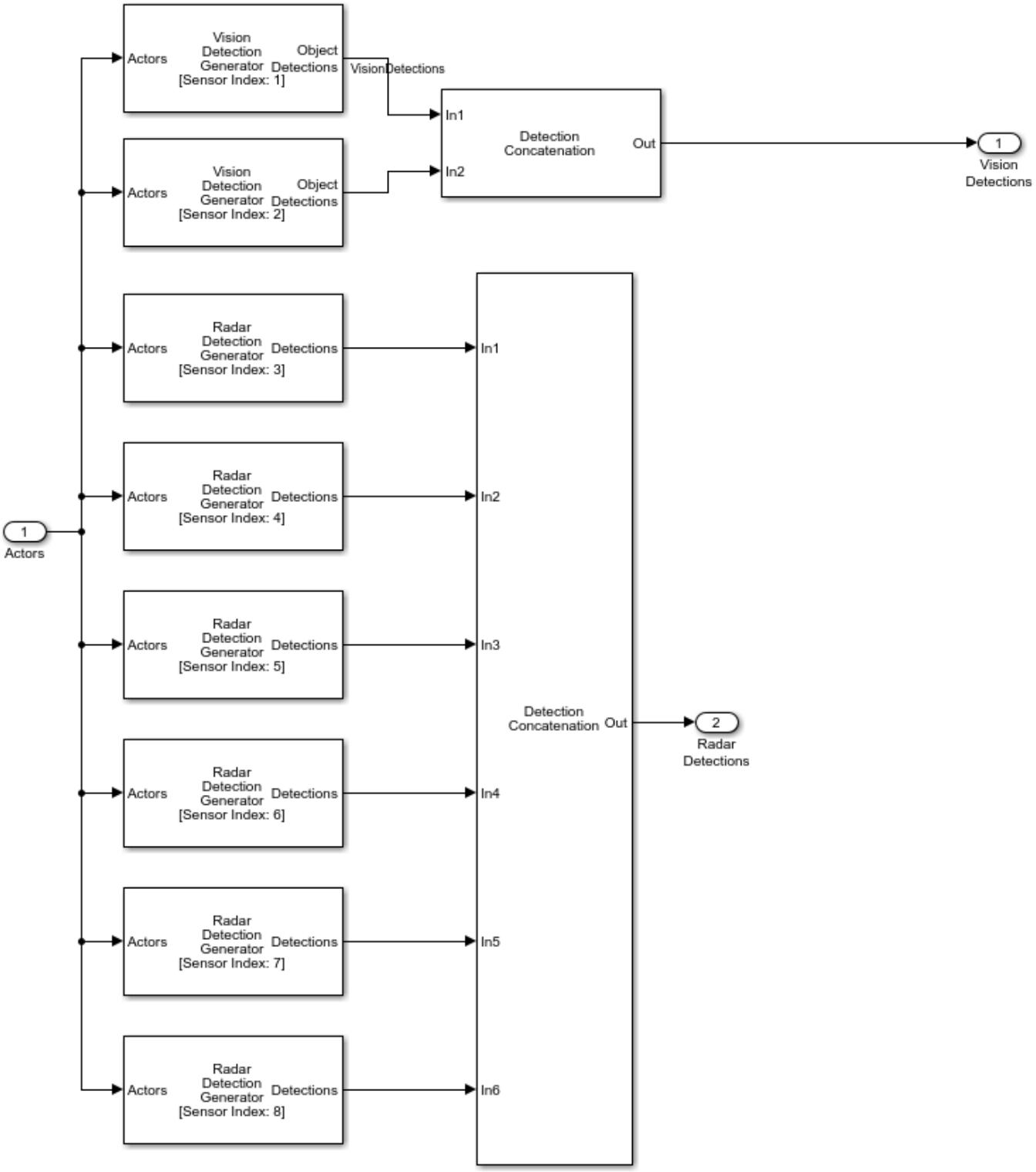


Simulating Sensor Detections

In this example, you simulate an ego vehicle that has 6 radar sensors and 2 vision sensors covering the 360 degrees field of view. The sensors have some overlap and some coverage gap. The ego vehicle is equipped with a long-range radar sensor and a vision sensor on both the front and the back of the vehicle. Each side of the vehicle has two short-range radar sensors, each covering 90 degrees. One sensor on each side covers from the middle of the vehicle to the back. The other sensor on each side covers from the middle of the vehicle forward.

When you open the Sensor Simulation subsystem, you can see the two Vision Detection Generator blocks, configured to generate detections from the front and the back of the ego vehicle. The output

from the vision detection generators is connected to a Detection Concatenation block. Next, the subsystem contains six Radar Detection Generator blocks, configured as described in the previous paragraph. The outputs of the radar detection generators are concatenated and then clustered using a Detection Clustering block.


Actors
Vision Detection Generator [Sensor Index: 1]
Object Detections
VisionDetections
Vision Detection Generator [Sensor Index: 2]
In1
In2
Detection Concatenation
Out
1
Vision Detections
Radar Detection Generator [Sensor Index: 3]
Radar Detection Generator [Sensor Index: 4]
Radar Detection Generator [Sensor Index: 5]
Radar Detection Generator [Sensor Index: 6]
Radar Detection Generator [Sensor Index: 7]
Radar Detection Generator [Sensor Index: 8]
Detections
In3
In4
In5
In6
Detection Concatenation
Out
2
Radar Detections
1
Actors

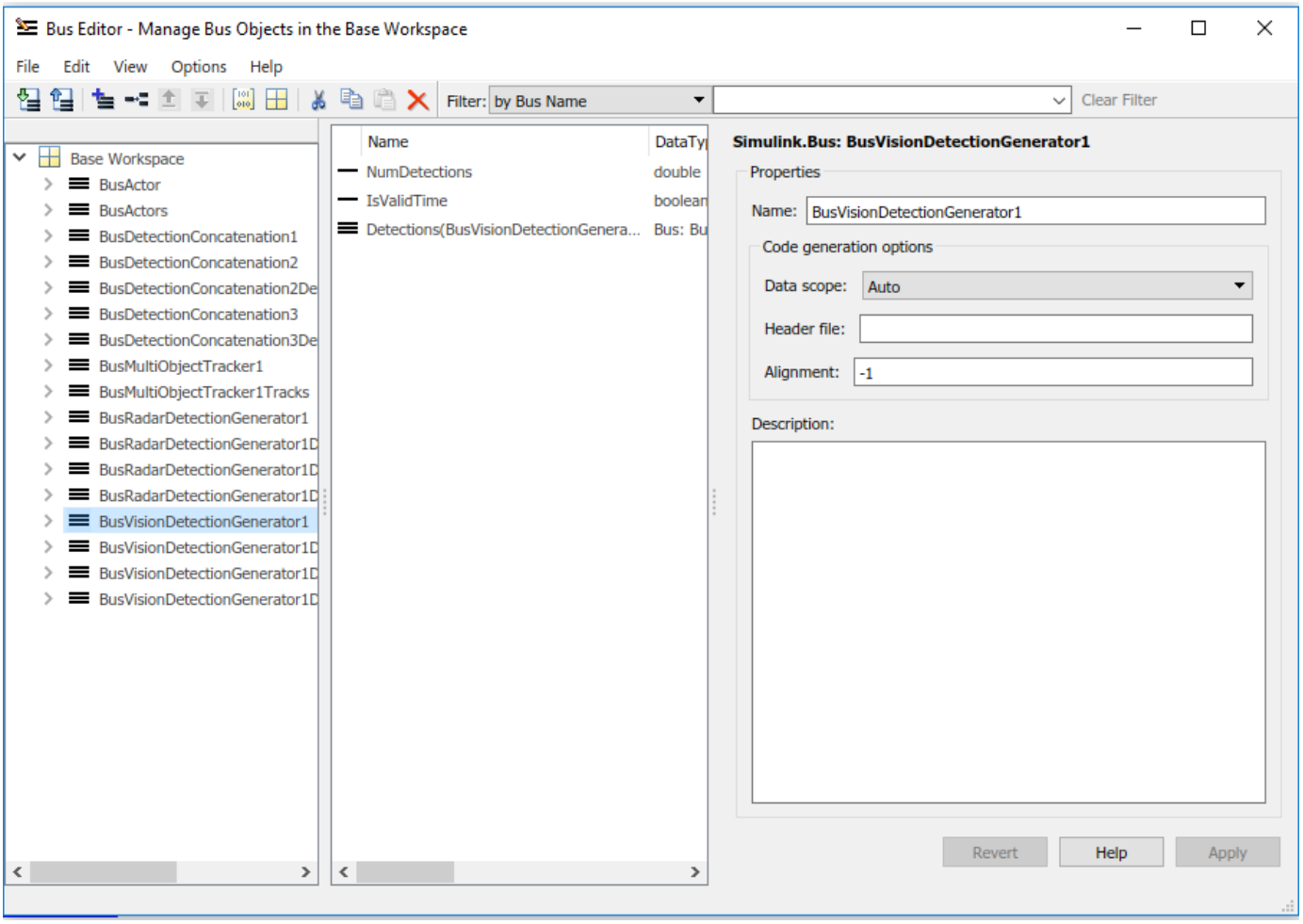
Tracking and Sensor Fusion

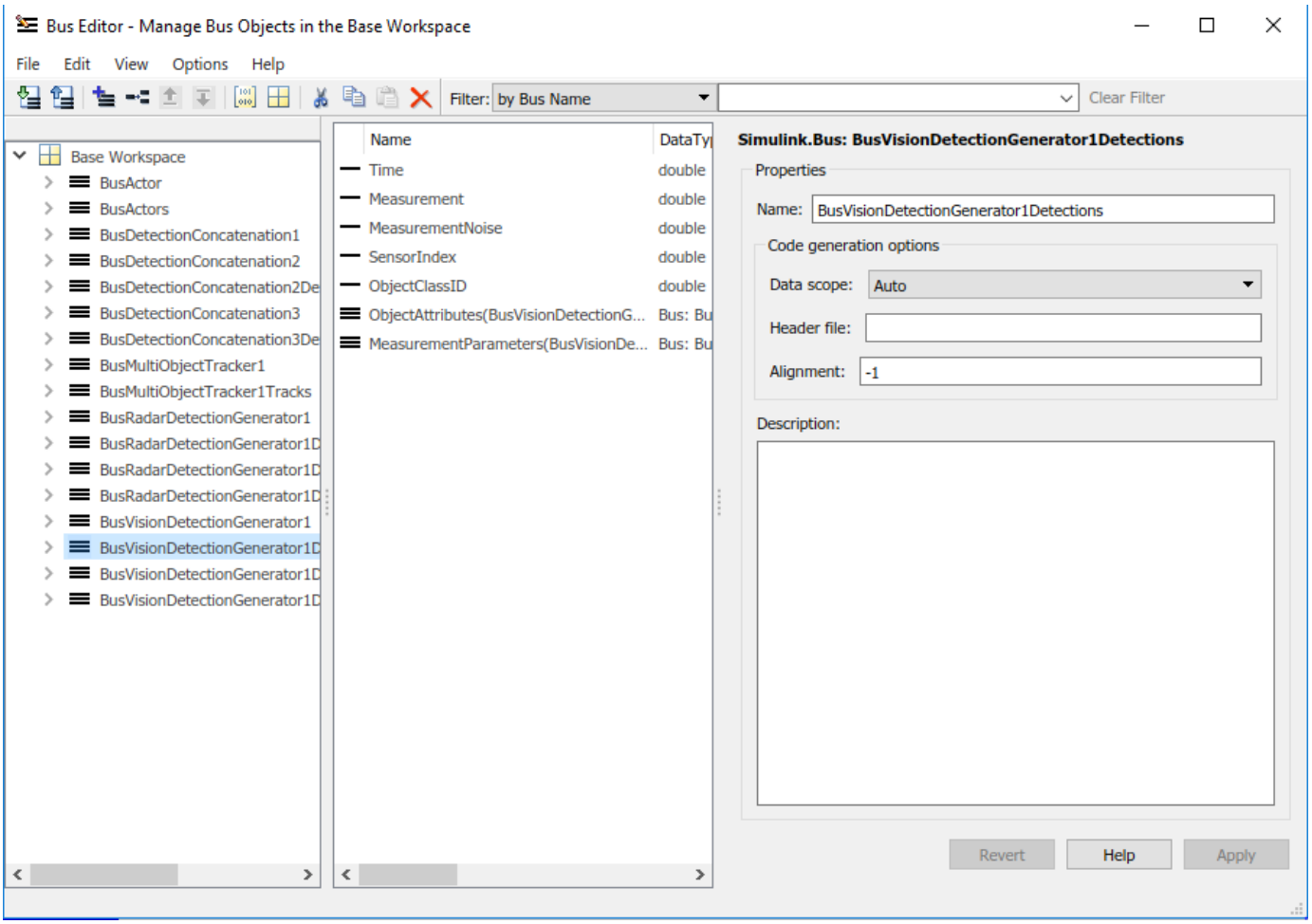
The detections from the vision and radar sensors must first be Concatenated to form a single input to the Multi-Object Tracker block. The Concatenation is done using an additional Detection Concatenation block.

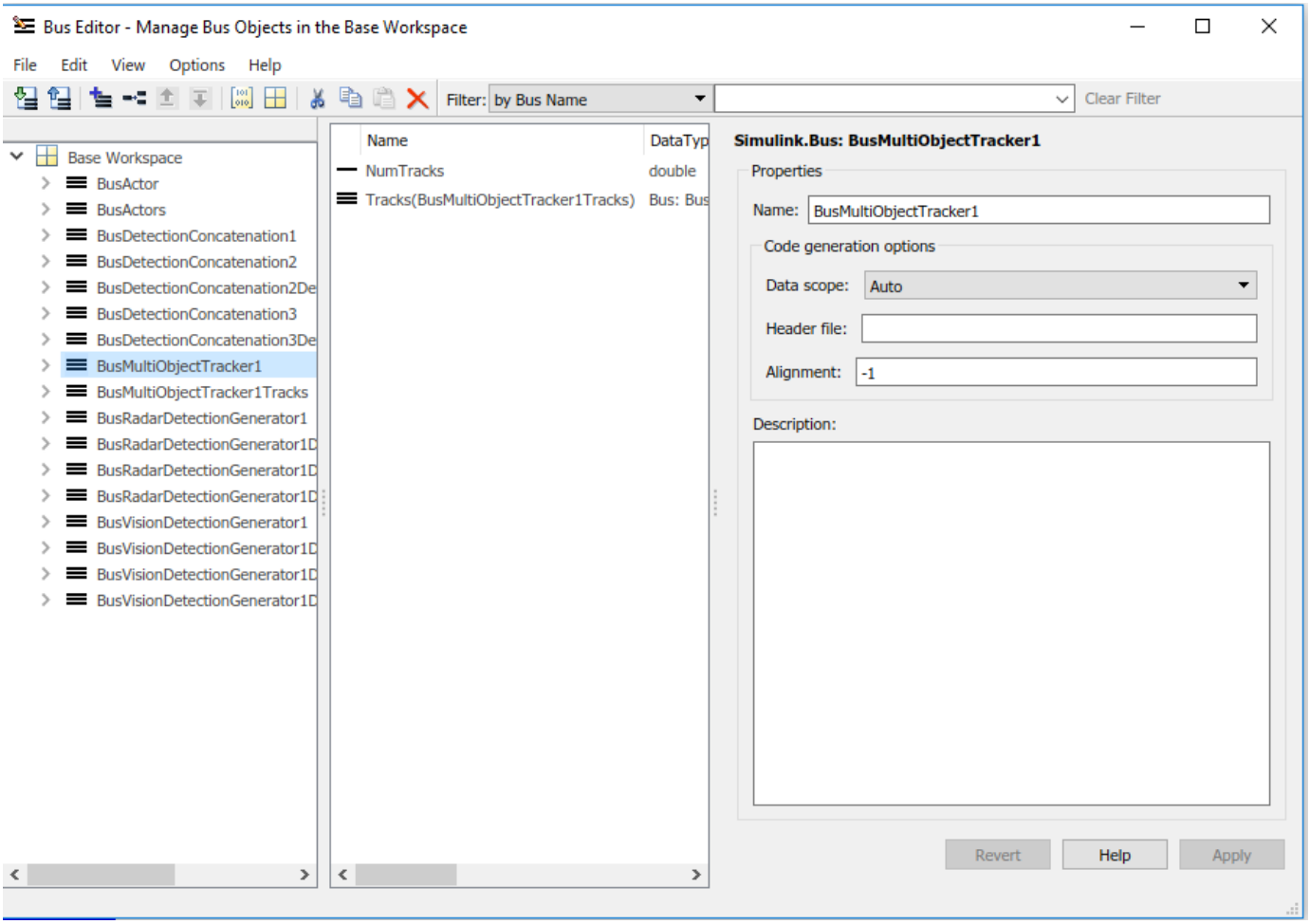
The Multi-Object Tracker block is responsible for fusing the data from all the detections and tracking the objects around the ego vehicle. The multi-object tracker is configured with the same parameters that were used in the corresponding MATLAB example, “Sensor Fusion Using Synthetic Radar and Vision Data” on page 7-196. The output from the Multi-Object Tracker block is a list of confirmed tracks.

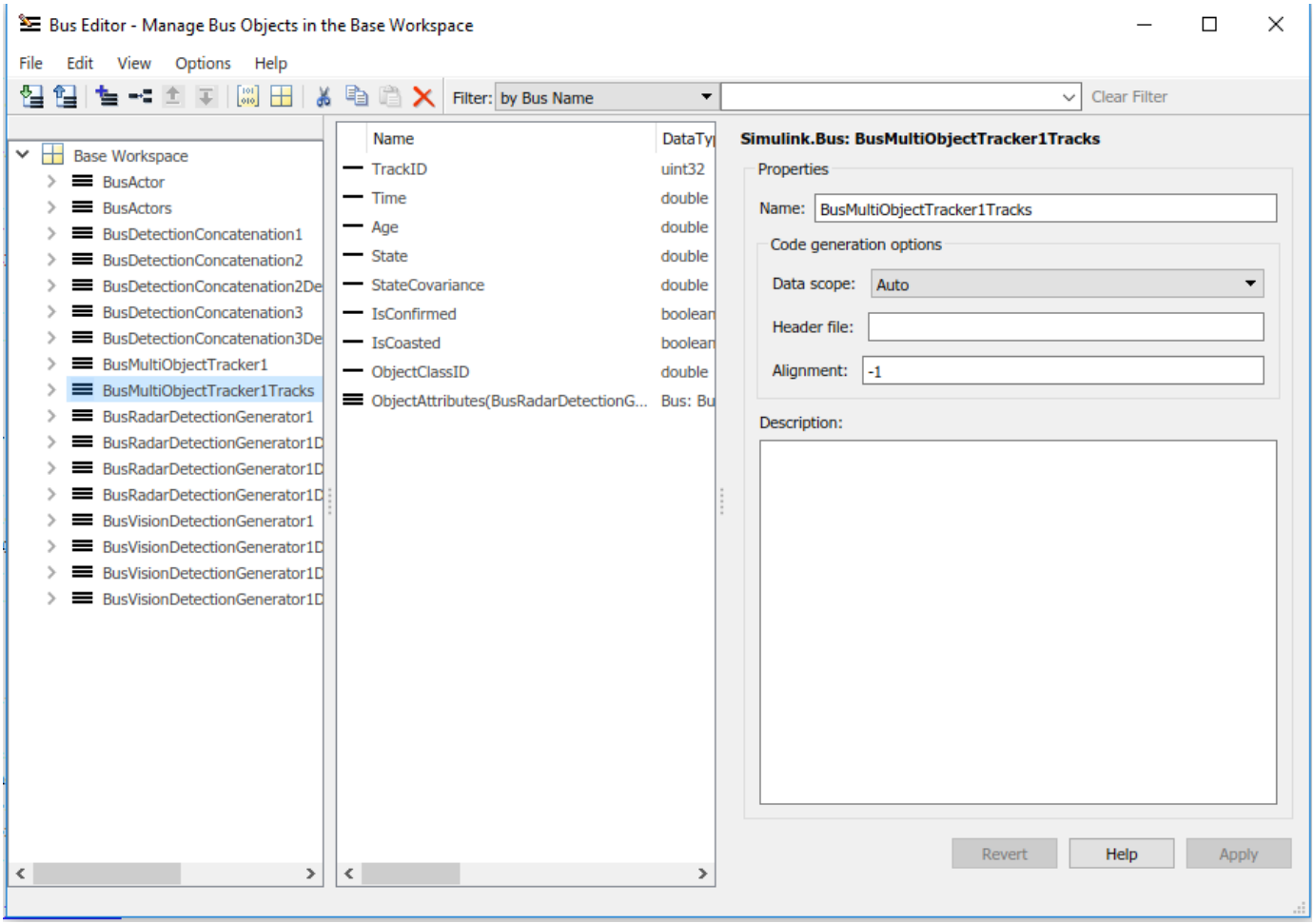
Creating and Propagating Buses

The inputs and outputs from the various blocks in this example are all `Simulink.Bus (Simulink)` objects. To simplify compiling the model and creating the buses, all the Vision Detection Generator, Radar Detection Generator, Multi-Object Tracker, and Detection Concatenation blocks have a property that defines the source of the output bus name. When set to `'Auto'`, the buses are created automatically and their names are propagated to the block that consumes this bus as an input. When set to `'Property'`, you can define the name of the output bus. The following images show the detections bus, a single detection bus, the tracks bus, and a single track bus.



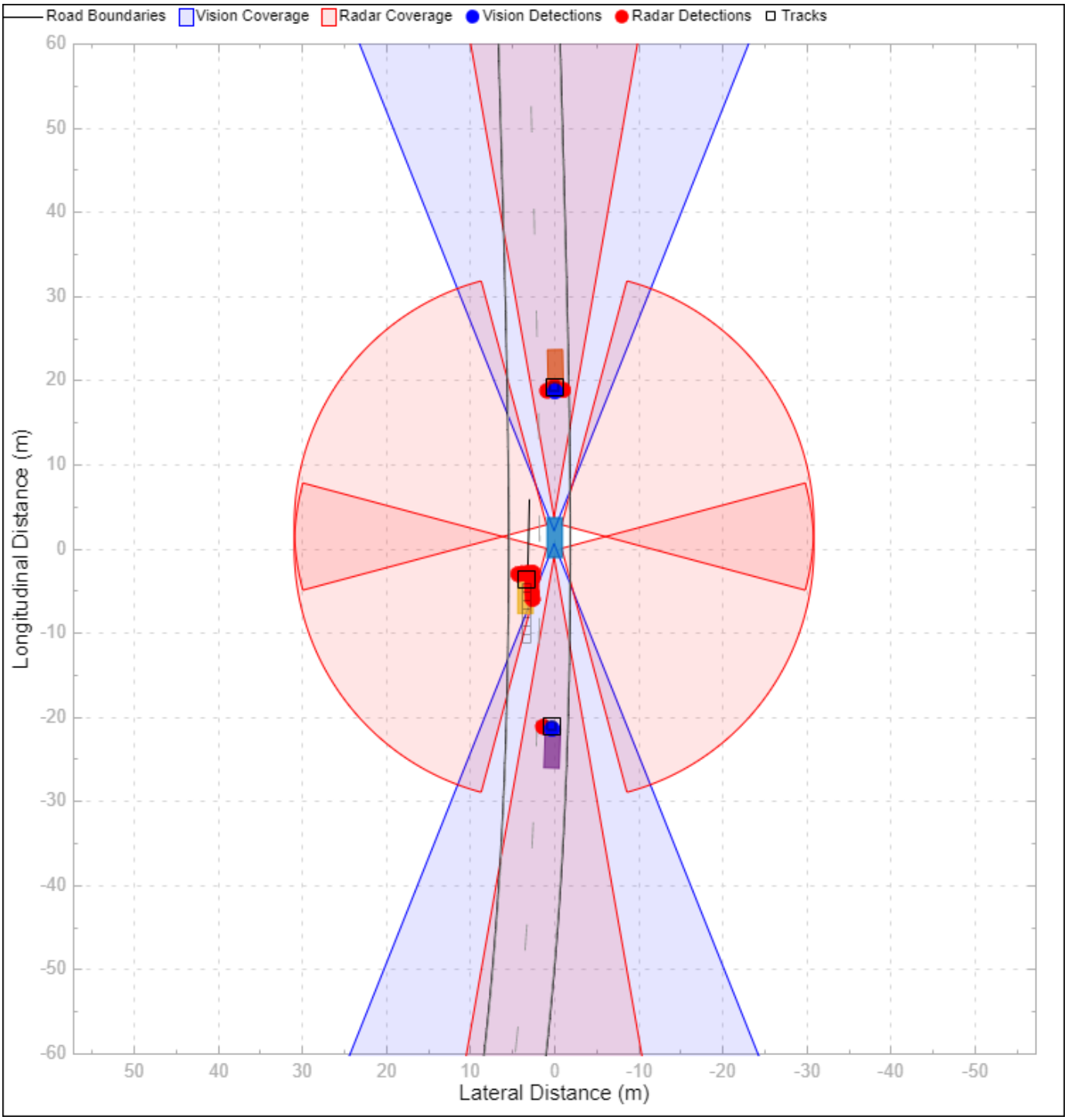

Bus Editor - Manage Bus Objects in the Base Workspace
File
Edit
View
Options
Help
Filter: by Bus Name
Clear Filter
Base Workspace
BusActor
BusActors
BusDetectionConcatenation1
BusDetectionConcatenation2
BusDetectionConcatenation2De
BusDetectionConcatenation3
BusDetectionConcatenation3De
BusMultiObjectTracker1
BusMultiObjectTracker1Tracks
BusRadarDetectionGenerator1
BusRadarDetectionGenerator1D
BusRadarDetectionGenerator1D
BusRadarDetectionGenerator1D
BusVisionDetectionGenerator1
BusVisionDetectionGenerator1D
BusVisionDetectionGenerator1D
BusVisionDetectionGenerator1D
Name
DataTy
Time
double
Measurement
double
MeasurementNoise
double
SensorIndex
double
ObjectClassID
double
ObjectAttributes(BusVisionDetectionG...
Bus: Bu
MeasurementParameters(BusVisionDe...
Bus: Bu
Simulink.Bus: BusVisionDetectionGenerator1Detections
Properties
Name:
BusVisionDetectionGenerator1Detections
Code generation options
Data scope:
Auto
Header file:
Alignment:
-1
Description:
Revert
Help
Apply


Bus Editor - Manage Bus Objects in the Base Workspace
File
Edit
View
Options
Help
Filter: by Bus Name
Clear Filter
Base Workspace
BusActor
BusActors
BusDetectionConcatenation1
BusDetectionConcatenation2
BusDetectionConcatenation2De
BusDetectionConcatenation3
BusDetectionConcatenation3De
BusMultiObjectTracker1
BusMultiObjectTracker1Tracks
BusRadarDetectionGenerator1
BusRadarDetectionGenerator1D
BusRadarDetectionGenerator1D
BusRadarDetectionGenerator1D
BusVisionDetectionGenerator1
BusVisionDetectionGenerator1D
BusVisionDetectionGenerator1D
BusVisionDetectionGenerator1D
Name
DataTyp
NumTracks
double
Tracks(BusMultiObjectTracker1Tracks)
Bus: Bus
Simulink.Bus: BusMultiObjectTracker1
Properties
Name:
BusMultiObjectTracker1
Code generation options
Data scope:
Auto
Header file:
Alignment:
-1
Description:
Revert
Help
Apply



Display

The Bird's-Eye Scope is a model-level visualization tool that you can open from the Simulink toolstrip. On the **Simulation** tab, under **Review Results**, click **Bird's-Eye Scope**. After opening the scope, click **Find Signals** to set up the signals. Then run the simulation to display the actors, vision and radar detections, tracks, and road boundaries. The following image shows the bird's-eye scope for this example.



See Also

Apps
Driving Scenario Designer

Blocks
Detection Concatenation | Multi-Object Tracker | Radar Detection Generator | Vision Detection Generator

Objects
`drivingScenario`

Functions
`record` | `roadBoundaries`

More About

- “Sensor Fusion Using Synthetic Radar and Vision Data” on page 7-196
- “Forward Collision Warning Using Sensor Fusion” on page 7-125
- “Adaptive Cruise Control with Sensor Fusion” on page 7-138
- “Code Generation for Tracking and Sensor Fusion” on page 7-117
- “Autonomous Emergency Braking with Sensor Fusion” on page 7-214

Autonomous Emergency Braking with Sensor Fusion

This example shows how to implement autonomous emergency braking (AEB) with a sensor fusion algorithm by using Automated Driving Toolbox.

In this example, you:

1. Integrate a Simulink® and Stateflow® based AEB controller, a sensor fusion algorithm, ego vehicle dynamics, a driving scenario reader, and radar and vision detection generators.
2. Test the AEB system in a closed-loop Simulink model using a series of test scenarios created by the Driving Scenario Designer app
3. Configure the code generation settings for software-in-the-loop simulation, and automatically generate C code for the control algorithm.

Introduction

Autonomous emergency braking (AEB) is an advanced active safety system that helps drivers avoid or mitigate collisions with other vehicles or vulnerable road users. AEB systems improve safety by:

1. Preventing accidents by identifying critical situations early and warning the driver.
2. Reducing the severity of unavoidable crashes by lowering the speed of collision. In some cases, AEB systems prepare the vehicle and restraint systems for impact [1].

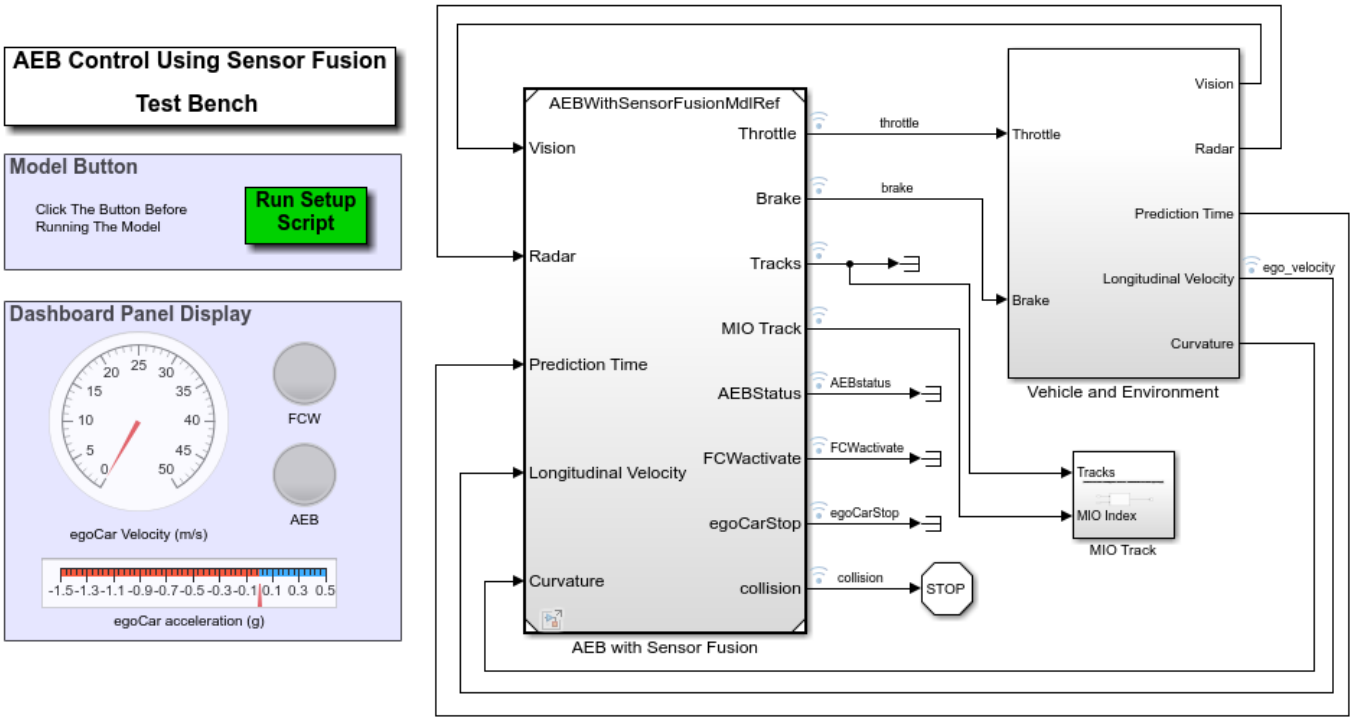
The European New Car Assessment Program (Euro NCAP) included the AEB city and inter-urban system in its safety rating from 2014. Euro NCAP continues to promote AEB systems for protecting vulnerable road users such as pedestrians and cyclists.

Today's AEB systems mostly use radar and vision sensors to identify potential collision partners ahead of the ego vehicle. Multiple sensors are often required for accurate, reliable, and robust detections while minimizing false positives. That is why sensor fusion technology plays an important role for the AEB system.

Overview of Simulink Model for AEB Test Bench

Add the example file folder to the MATLAB® search path. Then, open the main Simulink model used in this example.

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



The model consists of two main subsystems:

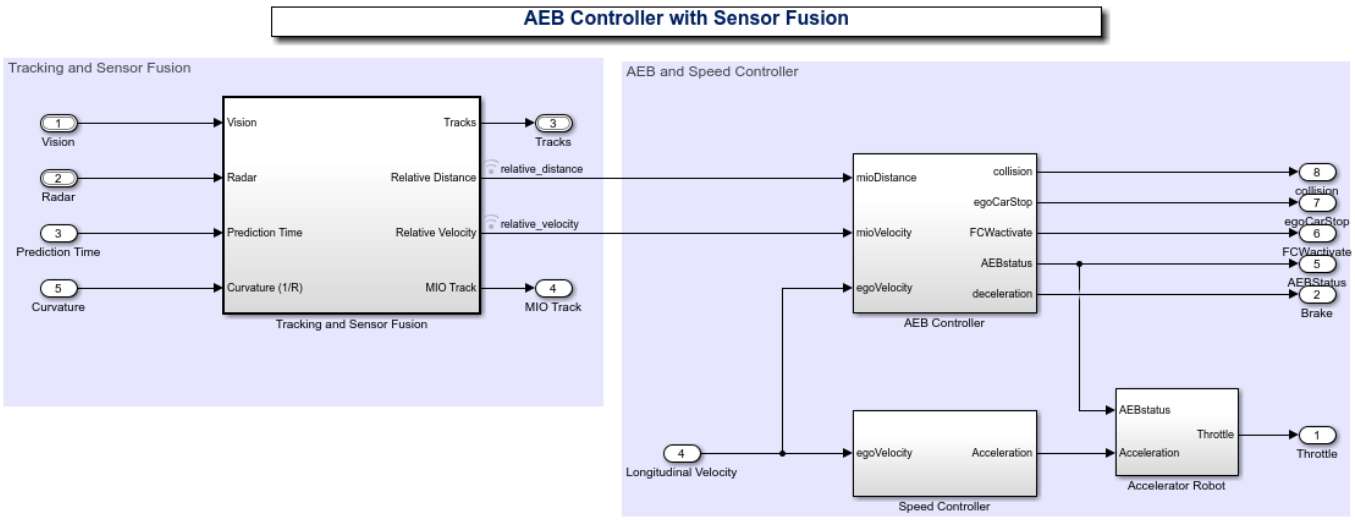
1. AEB with Sensor Fusion, which contains the sensor fusion algorithm and AEB controller.
2. Vehicle and Environment, which models the ego vehicle dynamics and the environment. It includes the driving scenario reader and radar and vision detection generators. These blocks provide synthetic sensor data for the objects.

To plot synthetic sensor detections, tracked objects and ground truth data, use the Bird's-Eye Scope. The Bird's-Eye Scope is a model-level visualization tool that you can open from the Simulink model toolbar. On the **Simulation** tab, under **Review Results**, click **Bird's-Eye Scope**. After opening the scope, click **Find Signals** to set up the signals. The Dashboard Panel displays ego vehicle velocity, acceleration, and the status of the autonomous emergency braking (AEB) and forward collision warning (FCW) controllers.

AEB Controller with Sensor Fusion

Open the AEB controller with Sensor Fusion subsystem.

```
open_system('AEBTestBenchExample/AEB with Sensor Fusion')
```



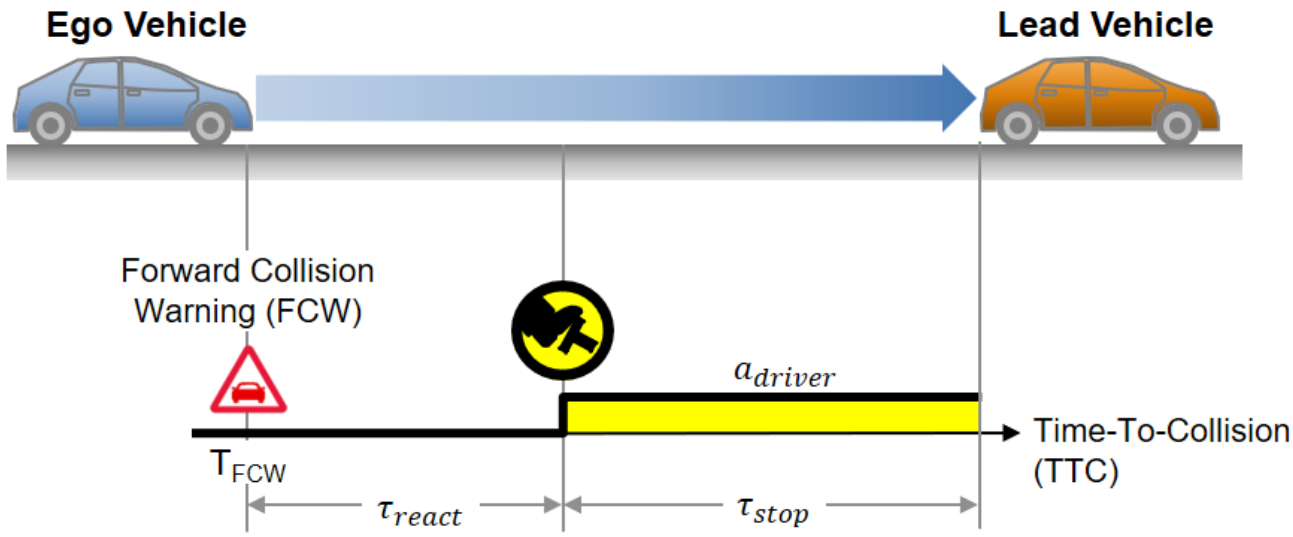
This subsystem contains the tracking and sensor fusion algorithm and the speed and AEB controllers.

- The Tracking and Sensor Fusion subsystem processes vision and radar detections coming from the Vehicle and Environment subsystem and generates the position and velocity of the most important object (MIO) track relative to the ego vehicle.
- The Speed Controller subsystem makes the ego vehicle travel at a driver's set velocity by using a proportional integral (PI) controller.
- The Accelerator Robot subsystem releases the vehicle accelerator when AEB is activated.
- The AEB Controller subsystem implements the forward collision warning (FCW) and AEB control algorithm based on stopping time calculation approach.

Stopping time refers to the time from when the ego vehicle first applies its brakes with deceleration, a_{brake}, to when it comes to a complete stop. Stopping time can be obtained by the following equation:

$$\tau_{stop} = v_{ego}/a_{brake}$$

The FCW system alerts the driver of an imminent collision with a lead vehicle. The driver is expected to react to the alert and apply the brake with a delay time, τ_{react}.

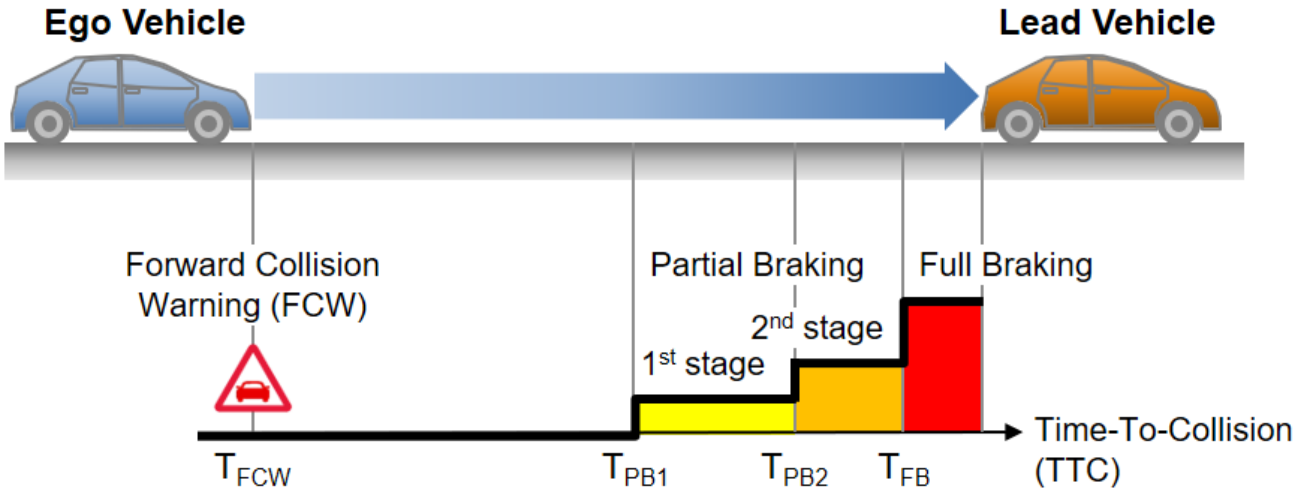


The total travel time of the ego vehicle before colliding with the lead vehicle can be expressed by:

$$\tau_{FCW} = \tau_{react} + \tau_{stop} = \tau_{react} + v_{ego}/a_{driver}$$

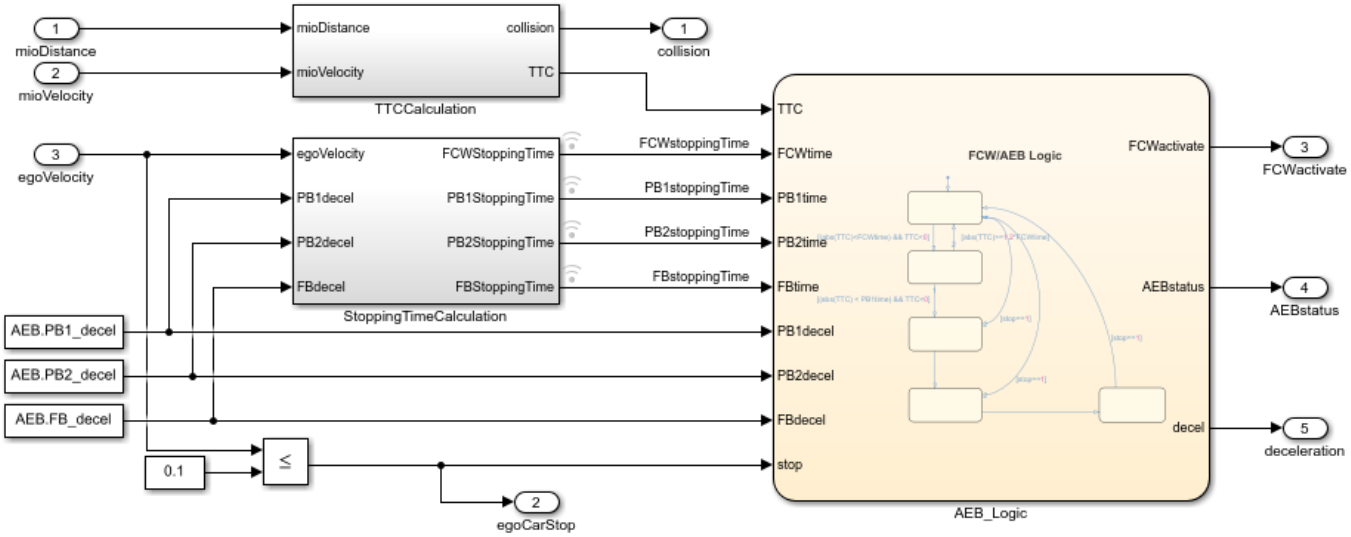
When the time-to-collision (TTC) of the lead vehicle becomes less than τ_{FCW}, the FCW alert is activated.

If the driver fails to apply the brakes in time, such as due to distractions, the AEB system acts independently of the driver to avoid or mitigate the collision. The AEB systems typically apply cascaded braking, which consists of the multi-stage partial braking followed by full braking [2].



Open the AEB Controller subsystem.

```
open_system('AEBWithSensorFusionMdlRef/AEB Controller')
```



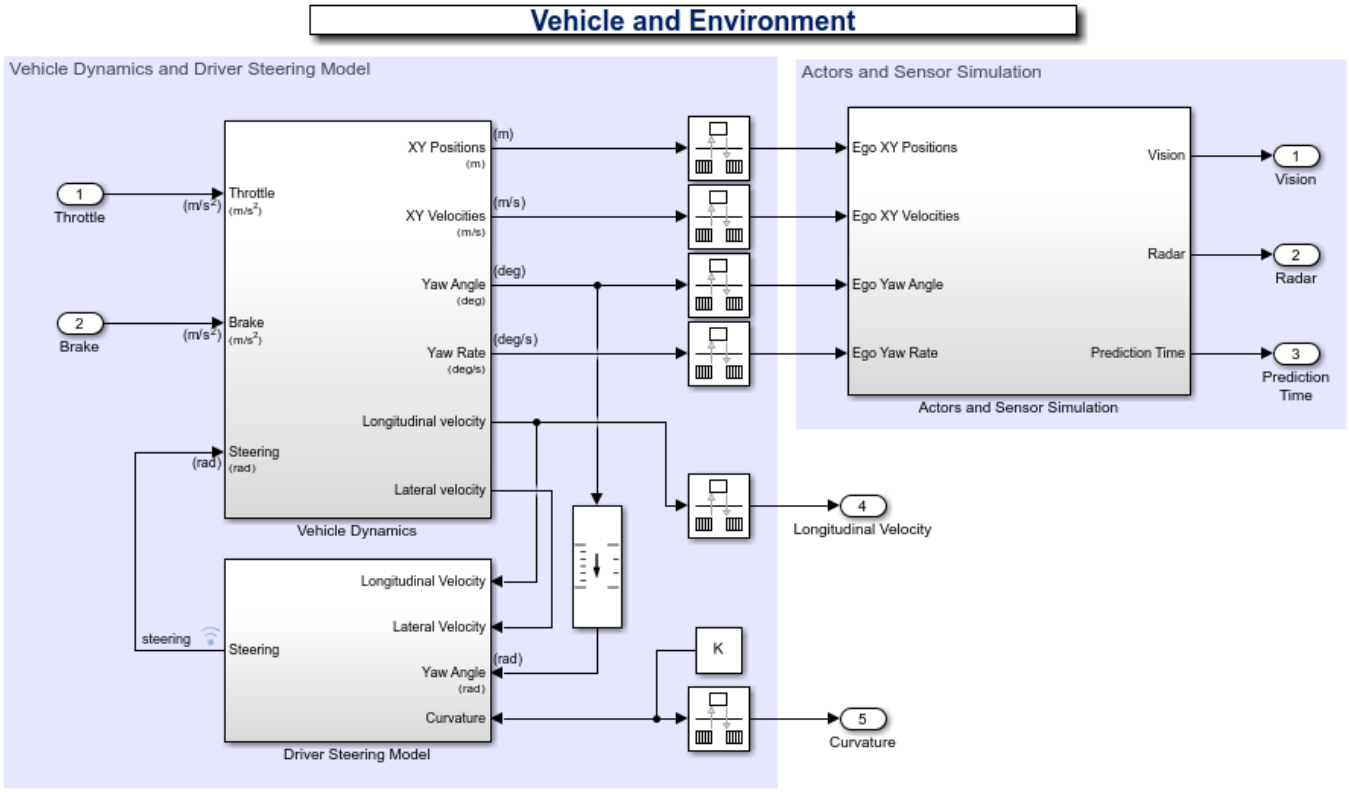
The AEB controller consists of multiple function blocks:

- TTCCalculation, which calculates the TTC using the relative distance and velocity of the lead vehicle or the most important object
- StoppingTimeCalculation, which calculates stopping times for FCW, first- and second-stage partial braking (PB), and full braking (FB), respectively
- AEB_Logic, which is a state machine comparing the TTC with the stopping times to determine FCW and AEB activations.

Vehicle and Environment

Open the Vehicle and Environment subsystem.

```
open_system('AEBTestBenchExample/Vehicle and Environment')
```



- The Vehicle Dynamics subsystem models the ego vehicle dynamics with Vehicle Body 3DOF (Vehicle Dynamics Blockset) Single Track block from Vehicle Dynamics Blockset.
- The Driver Steering Model subsystem generates the driver steering angle to keep the ego vehicle in its lane and follow the curved road defined by the curvature, K.
- The Actor and Sensor Simulation subsystem generates the synthetic sensor data required for tracking and sensor fusion. Right after the Simulink model is loaded, a callback function is executed to create a simulation environment with a road and multiple actors moving on the road.

You can also run the callback function by clicking **Run Setup Script** from the main Simulink model or type the following from the command prompt:

```
helperAEBSetUp
```

You can specify a scenario number corresponding to a desired scenario name from the list.

```
% Create driving scenario
scenariosNames = {                                  % scenarioNumber
    'AEB_CCRs_100overlap.mat',...                   % 1
    'AEB_CCRm_100overlap.mat',...                   % 2
    'AEB_CCRb_2_initialGap_12m_stop_inf.mat',...    % 3
    'AEB_CCRb_6_initialGap_40m_stop_inf.mat',...    % 4
    'AEB_PedestrianChild_Nearside_50width_overrun.mat'};    % 5

scenarioNumber = 5;
```

The scenario name is a scenario file created by the Driving Scenario Designer.

```
[scenario,egoCar,actor_Profiles] = ...
helperSessionToScenario(scenariosNames{scenarioNumber});
```

The scenario file can be converted to a `drivingScenario` object using the `helperSessionToScenario` script.

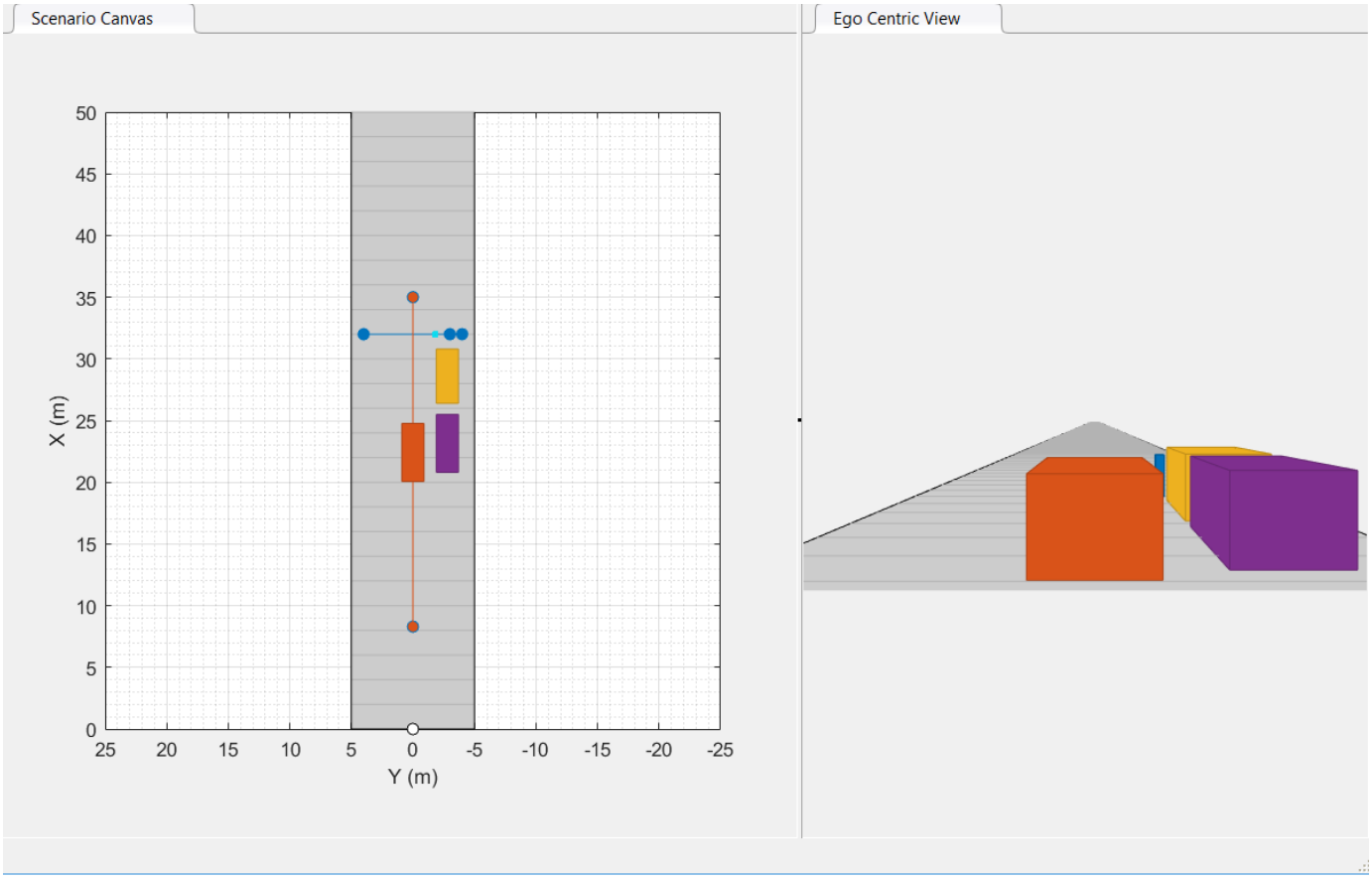
The Scenario Reader block reads the actor poses data from the scenario file. The block converts the actor poses from the world coordinates of the scenario into ego vehicle coordinates. The actor poses are streamed on a bus generated by the block. The Vision Detection Generator block and the Radar Detection Generator block synthesize vision and radar detections for the target actors respectively.

Test AEB System Based on Euro NCAP Test Protocol

Euro NCAP offers a series of test protocols that test the performance of AEB systems in car-to-car rear (CCR) and vulnerable road users (VRU) scenarios.

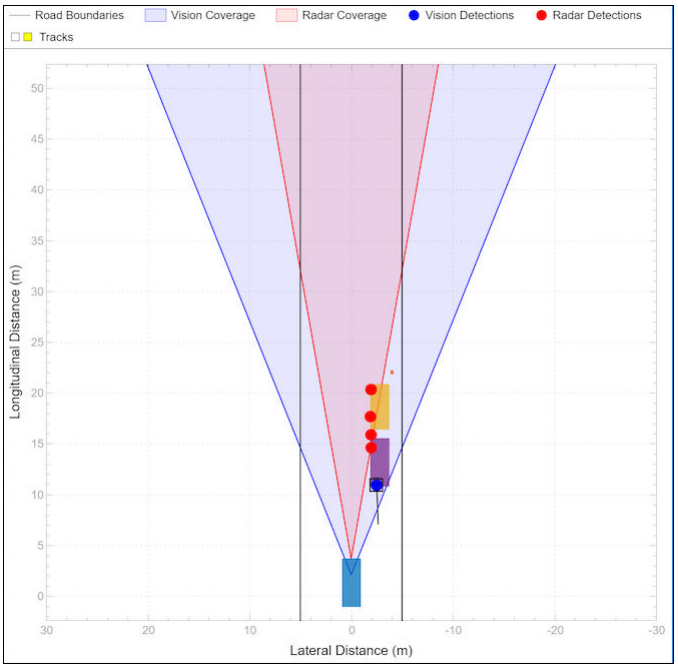
- Euro NCAP AEB - Car-to-Car Rear test protocol [3]
- Euro NCAP AEB - Vulnerable Road User test protocol [4]

Automated Driving Toolbox provides prebuilt driving scenarios according to the Euro NCAP test protocols for the AEB system. You can review the prebuilt scenarios using Driving Scenario Designer.



The AEB Simulink model reads the driving scenario file and runs a simulation.

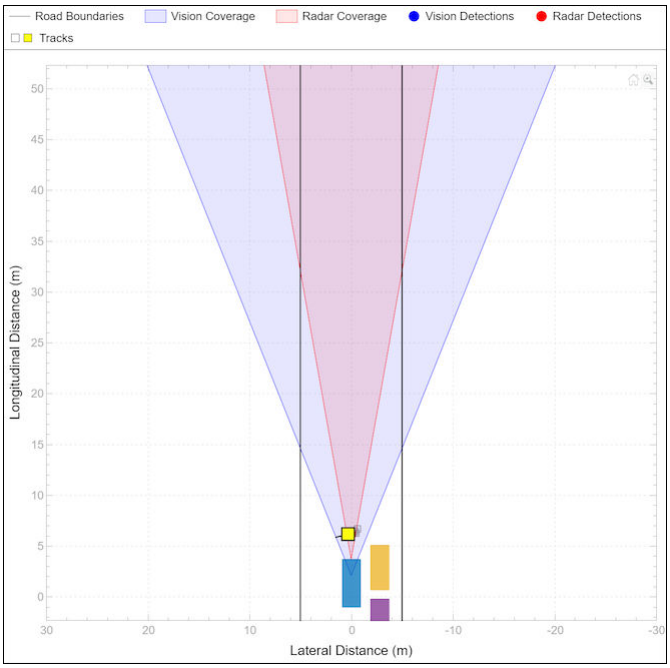
Simulate the model for 0.1 seconds.




```
sim('AEBTestBenchExample','StopTime','0.1'); % Simulate 0.1 seconds
```

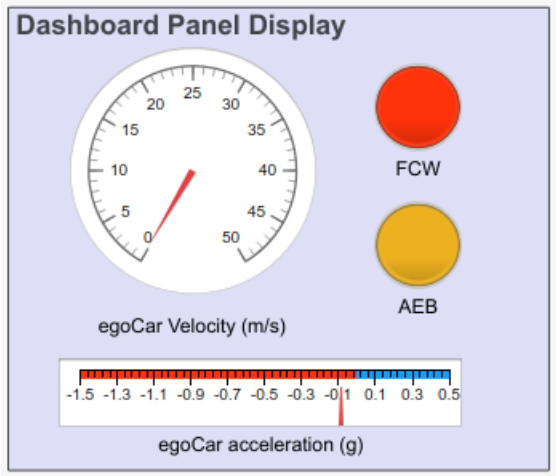
The Bird's-Eye Scope shows ground truth data of vehicles and a child pedestrian. It also shows radar detections, vision detections and objects tracked by the multi-object tracker. At the simulation time of 0.1 seconds, the vision and radar sensors fail to detect the child pedestrian as it is obstructed by the vehicles.

Simulate the model for 3.8 seconds.




```
sim('AEBTestBenchExample','StopTime','3.8'); % Simulate 3.8 seconds
```

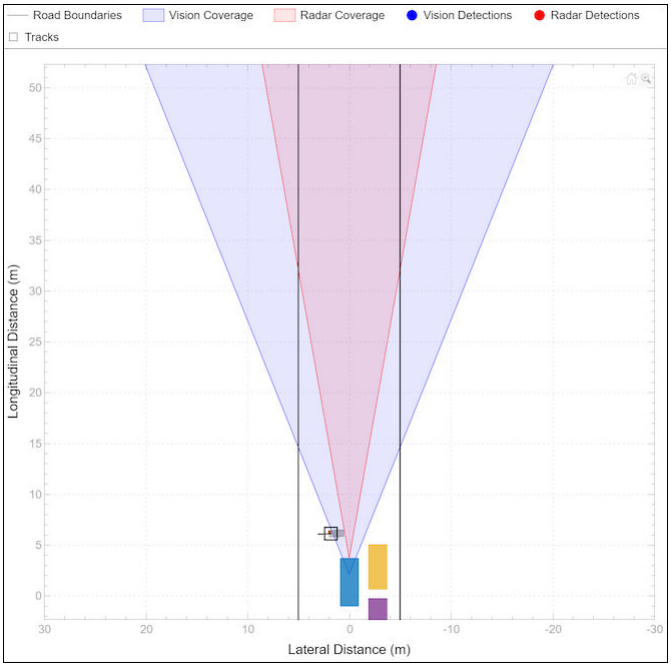
The Bird's-Eye Scope at the simulation time of 3.8 seconds shows that the sensor fusion and tracking algorithm detected the child pedestrian as the most important object and that the AEB system applied the brakes to avoid a collision.



The dashboard panel displayed along with the Bird's-Eye Scope showed that the AEB system applied a cascaded brake and the ego vehicle stopped right before a collision. The AEB status color indicates the level of AEB activation.

- Gray - No AEB is activated.
- Yellow - First stage partial brake is activated.
- Orange - Second stage partial brake is activated.
- Red - Full brake is activated.

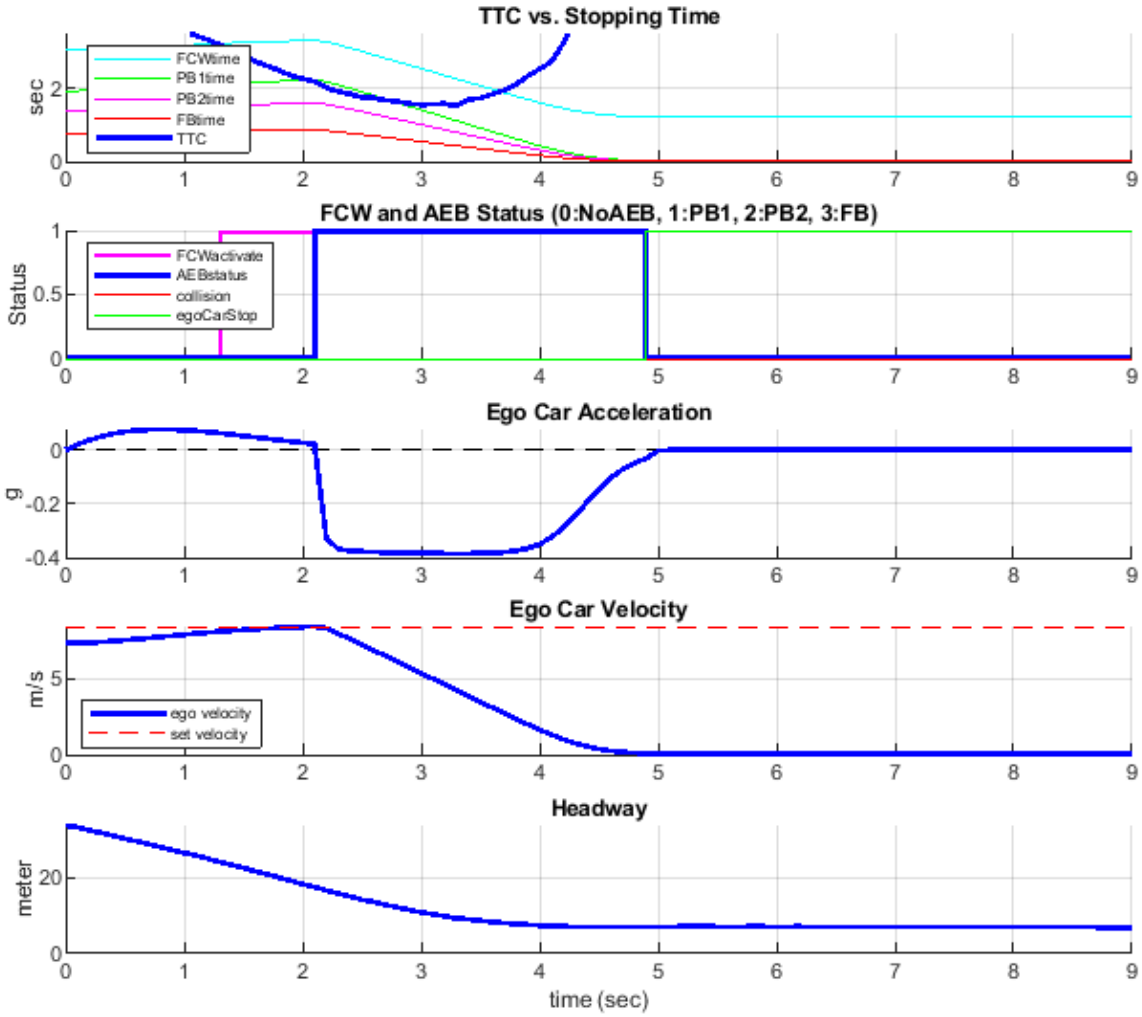
Complete the simulation all the way to the end to gather results.



```
sim('AEBTestBenchExample'); % Simulate to end of scenario
```

View the simulation results.

```
helperPlotAEBResults(logsout);
```



- The first plot (TTC vs. Stopping Time) shows a comparison between time-to-collision (TTC) and the stopping times for the FCW, first stage partial brake, second stage partial brake and full brake respectively.
- The second plot shows how the AEB state machine determines the activations for FCW and AEB based on the comparison results from the first plot.
- The third plot shows the velocity of the ego vehicle.
- The fourth plot shows the acceleration of the ego vehicle.
- The fifth plot shows the headway between the ego vehicle and the MIO.

In the first 2 seconds, the ego vehicle speeds up to reach the set velocity. At 2.3 seconds, the sensor fusion algorithm starts to detect the child pedestrian. Immediately after the detection, FCW is activated.

At 2.4 seconds, the first stage of partial brake is applied and the ego vehicle starts to slow down. The second stage of partial brake is again applied at 2.5 seconds.

When the ego vehicle finally stops at 3.9 seconds, the headway between the ego vehicle and the child pedestrian is about 2.4 meters. The AEB system has made a full collision avoidance in this scenario.

Generate Code for Control Algorithm

The `AEBWithSensorFusionMdlRef` model is configured to support generating C code using Embedded Coder® software. To check if you have access to Embedded Coder, run:

```
hasEmbeddedCoderLicense = license('checkout','RTW_Embedded_Coder')
```

You can generate a C function for the model and explore the code generation report by running:

```
if hasEmbeddedCoderLicense
    rtwbuild('AEBWithSensorFusionMdlRef')
end
```

You can verify that the compiled C code behaves as expected using a software-in-the-loop (SIL) simulation. To simulate the `ACCWithSensorFusionMdlRef` referenced model in SIL mode, use:

```
if hasEmbeddedCoderLicense
    set_param('AEBTestBenchExample/AEB with Sensor Fusion',...
        'SimulationMode','Software-in-the-loop (SIL)')
end
```

When you run the `AEBTestBenchExample` model, code is generated, compiled, and executed for the `AEBWithSensorFusionMdlRef` model. This enables you to test the behavior of the compiled code through simulation.

Conclusion

In this example, you implemented an AEB system with a closed-loop Simulink model. The model consisted of a Simulink and Stateflow based AEB controller, a sensor fusion algorithm, ego vehicle dynamics, a driving scenario reader and radar and vision detection generators.

You tested the AEB system using a series of test scenarios created by Driving Scenario Designer.

You can now test the AEB system with other Euro NCAP test scenarios for AEB. These can be accessed from Driving Scenario Designer.

Remove the example file folder from the MATLAB search path.

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

References

[1] Euro NCAP | The European New Car Assessment Programme. Euro NCAP

[2] W. Hulshof, et al., "Autonomous Emergency Braking Test Results," 23rd International Technical Conference on the Enhanced Safety of Vehicles (ESV), Paper Number 13-0168, 2013

[3] Euro NCAP Test Protocol - AEB systems, ver. 2.0.1, Nov. 2017.

[4] Euro NCAP Test Protocol - AEB VRU systems, ver. 2.0.2, Nov. 2017.

See Also

Apps
Driving Scenario Designer

Blocks
Radar Detection Generator | Vehicle Body 3DOF | Vision Detection Generator

Objects
`birdsEyePlot` | `drivingScenario`

More About

- “Forward Collision Warning Using Sensor Fusion” on page 7-125
- “Adaptive Cruise Control with Sensor Fusion” on page 7-138
- “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” on page 7-205
- “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-41

Visualize Sensor Coverage, Detections, and Tracks

Configure and use a Bird's-Eye Plot to display sensor coverage, detections and tracking results around the ego vehicle.

Overview

Displaying data recorded in vehicle coordinates on a 2-dimensional map around the ego vehicle is an important part of analyzing sensor coverages, detections and tracking results. Use `birdsEyePlot` to display a snapshot of this information for a certain time or to stream data and efficiently update the display.

This example reads pre-recorded sensor data and tracking results. It includes the following:

- Lane information
- Vision objects
- Radar objects
- Positions, velocities, covariance matrices, and labels of the tracks.
- Most important object

The above information was recorded at a high rate of 20 updates per second, except vision detections that were recorded at 10 updates per second.

A sensor configuration file defines the position and coverage areas of a vision sensor and a radar sensor with two coverage modes. These coverage areas will be displayed on the bird's-eye plot.

Note that the `birdsEyePlot` object sets up a very specific vehicle coordinate system, where the X-axis points forward from the vehicle, the Y-axis points to the left of the vehicle, and the Z-axis points up from the ground. The origin of the coordinate system is typically defined as the center of the rear axle, and the positions of the sensors are defined relative to the origin. For more details, see “Coordinate Systems in Automated Driving Toolbox” on page 1-2.

Defining Scene Limits and Sensor Coverage

Configuring a bird's-eye plot takes two steps. In the first step, the bird's-eye plot is created, which sets up the coordinate system described above, where the x-axis is directed upwards and y-axis is directed to the left. It is possible to define the axes limits in each direction. In this forward looking example, we define the scene up to 90 meters in front of the ego vehicle and 35 meters on each side.

```
% Create a bird's-eye plot and limit its axes
BEP = birdsEyePlot('Xlimits', [0 90], 'Ylimits', [-35 35]);
```

In the second step, the bird's-eye plotters are created. The bird's-eye plot offers the following variety of plotters, each configured for plotting a specific data type. They include:

- coverageAreaPlotter - Plot sensor coverage areas
- detectionPlotter - Plot object detections
- trackPlotter - Plot tracks, track uncertainties, and history trails
- laneBoundaryPlotter - Plot lane boundaries
- pathPlotter - Plot object trajectory

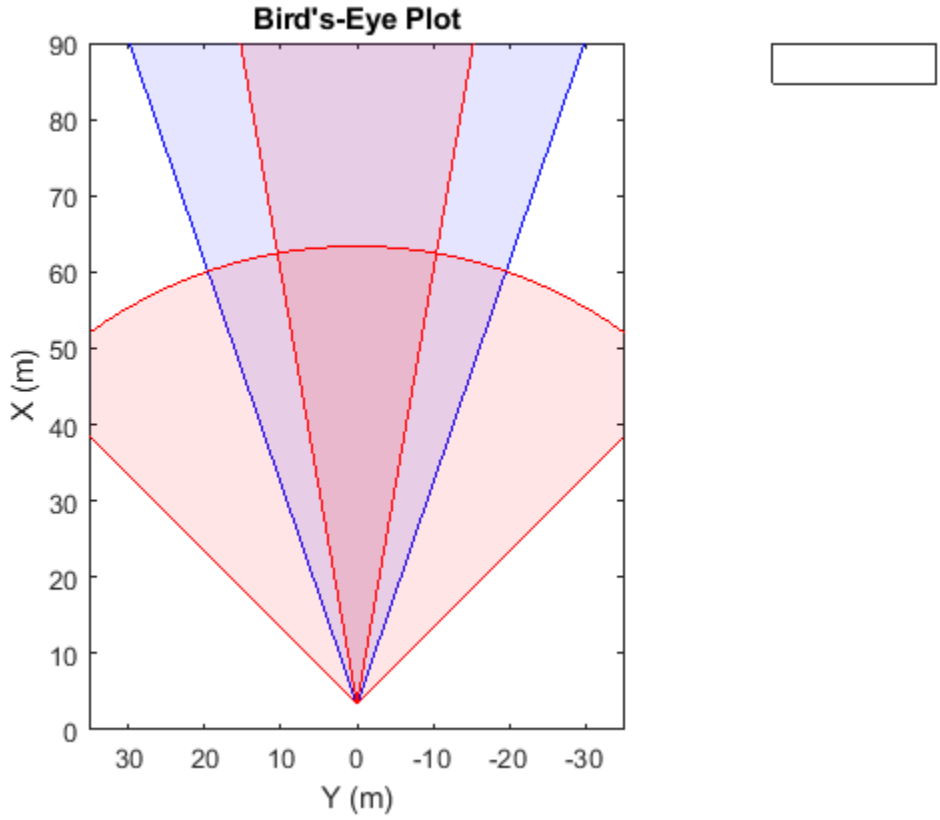
```
% Create a coverageAreaPlotter for a vision sensor and two radar modes
cap(1) = coverageAreaPlotter(BEP,'FaceColor','blue','EdgeColor','blue');
```

```
cap(2) = coverageAreaPlotter(BEP,'FaceColor','red','EdgeColor','red');
cap(3) = coverageAreaPlotter(BEP,'FaceColor','red','EdgeColor','red');
```

Load sensor configuration data. Sensor configuration includes:

- The position of the sensors relative to the axes origin (X,Y), in meters
- The sensor range, in meters
- The sensor yaw angle relative to the x-axis, in degrees
- The sensor field of view (FOV), in degrees

```
load('SensorConfigurationData.mat');
% Use the sensor configuration to plot the sensor coverage areas. Vision
% sensor uses the shaded blue coverage area and radar modes are shaded in
% red.
for i = 1:3
    plotCoverageArea(cap(i), [sensorParams(i).X, sensorParams(i).Y],...
        sensorParams(i).Range, sensorParams(i).YawAngle, sensorParams(i).FoV);
end

% Add title
title('Bird''s-Eye Plot')
```



The display above shows the coverage of the vision sensor and two radar sensor modes.

The vision sensor is positioned 3.30 meters in front of the origin (rear axle) at the center of the car, with a range of 150 meters and a FOV of 38 degrees.

The radar is positioned 3.38 meters in front of the origin at the center of the car. The radar long-range mode has a range of 174 meters and a FOV of 20 degrees, while the medium-range mode has a range of 60 meters and a FOV of 90 degrees. Note that the coverage areas are truncated at 90 meters in front of the ego vehicle and 35 meters on each side.

This example shows a forward looking scenario; however, you can define coverage area in 360° around the ego vehicle. For example, a sensor that covers from the rear of the vehicle backwards would be oriented with a yaw angle of 180°.

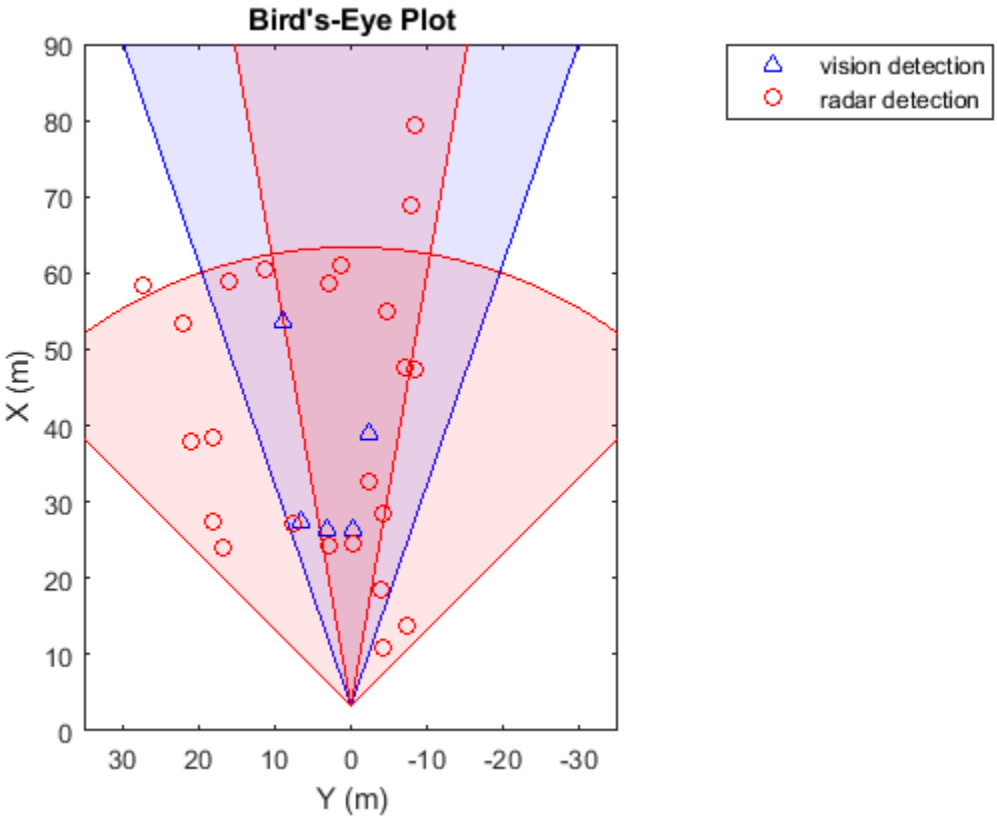
The next few lines read the recorded data in preparation for the next steps.

```
% Load recorded data from a file
load('BirdsEyePlotExampleData.mat', 'dataToDisplay');

% Skip to the 125th time step, where there are 5 vision detections and
% multiple radar objects and tracks.
timeStep = 125;

% Extract the various data from the recorded file for that time step
[visionObjectsPos, radarObjectsPos, laneBoundaries, trackPositions, ...
    trackVelocities, trackCovariances, trackLabels, MIOlabel, MIOposition, ...
    MIOvelocity] = readDataFrame(dataToDisplay(timeStep));
```

Plotting Detections

Next, create plotters to display the recorded vision and radar detections

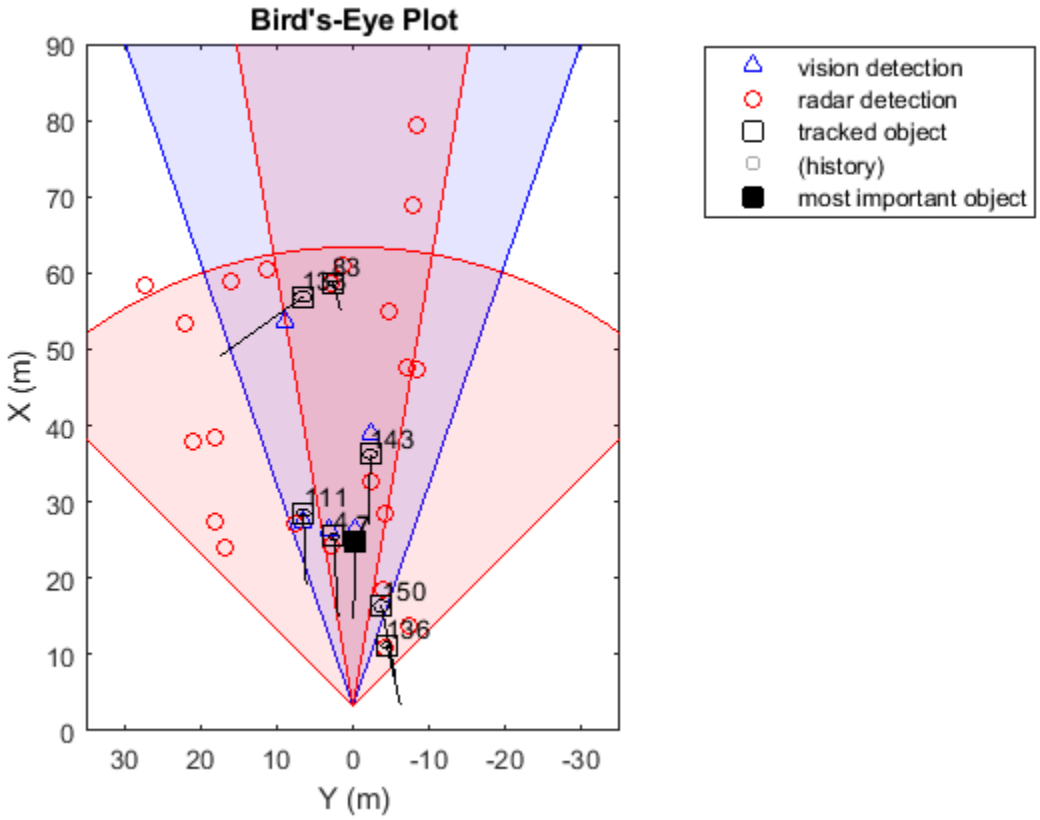
```
% create a vision detection plotter put it in a struct for future use
bepPlotters.Vision = detectionPlotter(BEP, 'DisplayName','vision detection', ...
    'MarkerEdgeColor','blue', 'Marker','^');

% Combine all radar detections into one entry and store it for later update
bepPlotters.Radar = detectionPlotter(BEP, 'DisplayName','radar detection', ...
    'MarkerEdgeColor','red');

% Call the vision detections plotter
plotDetection(bepPlotters.Vision, visionObjectsPos);

% Repeat the above for radar detections
plotDetection(bepPlotters.Radar, radarObjectsPos);
```



Plotting Tracks and Most-Important Objects

When adding the tracks to the Bird's-Eye Plot, we provide position, velocity and position covariance information. The plotter takes care of displaying the track history trail, but since this is a single frame, there will be no history.

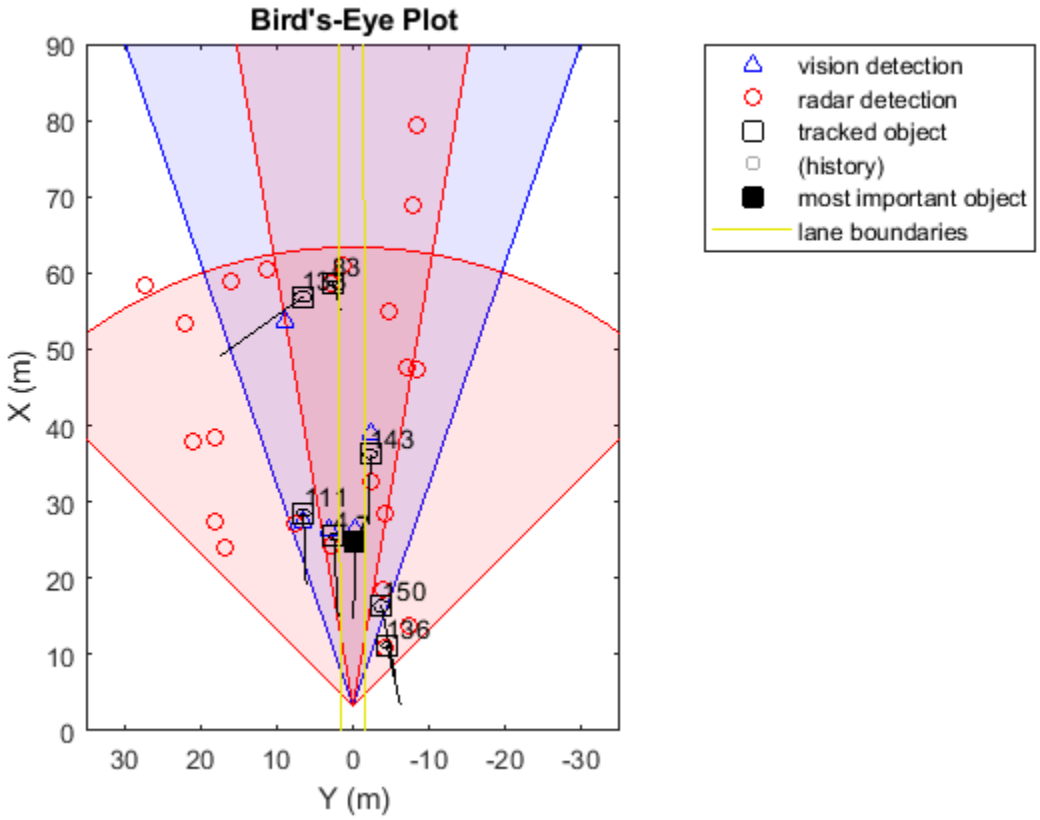
```
% Create a track plotter that shows the last 10 track updates
bepPlotters.Track = trackPlotter(BEP, 'DisplayName','tracked object', ...
    'HistoryDepth',10);

% Create a track plotter to plot the most important object
bepPlotters.MIO = trackPlotter(BEP, 'DisplayName','most important object', ...
    'MarkerFaceColor','black');

% Call the track plotter to plot all the tracks
plotTrack(bepPlotters.Track, trackPositions, trackVelocities, trackCovariances, trackLabels);

% Repeat for the Most Important Object (MIO)
plotTrack(bepPlotters.MIO, MIOposition, MIOvelocity, MIOlabel);
```



Plotting the Lane Boundaries

Plotting lane boundaries can utilize the `parabolicLaneBoundary` object. To use it, we saved the lane boundaries as parabolicLaneBoundary objects, and call the plotter with it.

```
% Create a plotter for lane boundaries
bepPlotters.LaneBoundary = laneBoundaryPlotter(BEP, ...
    'DisplayName','lane boundaries', 'Color',[.9 .9 0]);

% Call the lane boundaries plotter
plotLaneBoundary(bepPlotters.LaneBoundary, laneBoundaries);
```



Displaying a Scenario from a Recording File

The recording file contains time-dependent sensor detections, tracking information, and lane boundaries. The next code shows how to play back the recordings and display the results on the bird's-eye plot that was configured above.

Note: vision detections were provided every other frame. In such cases, it is beneficial to show the lack of new sensor detections. To do that, simply pass an empty array to the appropriate plotter to delete the previous detections from the display.

```
% Rewind to the beginning of the recording file
timeStep = 0;
numSteps = numel(dataToDisplay); % Number of steps in the scenario

% Loop through the scenario as long as the bird's eye plot is open
while timeStep < numSteps && isvalid(BEP.Parent)
    % Promote the timeStep
    timeStep = timeStep + 1;

    % Capture the current time for a realistic display rate
    tic;

    % Read the data for that time step
    [visionObjectsPos, radarObjectsPos, laneBoundaries, trackPositions, ...
        trackVelocities, trackCovariances, trackLabels, MIOlabel, MIOposition, ...
        MIOvelocity] = readDataFrame(dataToDisplay(timeStep));
```

```
    % Plot detections
    plotDetection(bepPlotters.Vision, visionObjectsPos);
    plotDetection(bepPlotters.Radar, radarObjectsPos);

    % Plot tracks and MIO
    plotTrack(bepPlotters.Track, trackPositions, trackVelocities, trackCovariances, trackLabels)
    plotTrack(bepPlotters.MIO, MIOposition, MIOvelocity, MIOlabel);

    % Plot lane boundaries
    plotLaneBoundary(bepPlotters.LaneBoundary, laneBoundaries);

    % The recorded data was obtained at a rate of 20 frames per second.
    % Pause for 50 milliseconds for a more realistic display rate. You
    % would not need this when you process data and form tracks in this
    % loop.
    pause(0.05 - toc)
end
```

Summary

This example demonstrated how to configure and use a bird's-eye plot object and some of the various plotters associated with it.

Try using the track and most-important object plotters or using the bird's-eye plot with a different recording file.

Supporting Functions

readDataFrame - extracts the separate fields from the data provided in dataFrame

```
function [visionObjectsPos, radarObjectsPos, laneBoundaries, trackPositions, ...
    trackVelocities, trackCovariances, trackLabels, MIOlabel, MIOposition, ...
    MIOvelocity] = readDataFrame(dataFrame)
    visionObjectsPos    = dataFrame.visionObjectsPos;
    radarObjectsPos     = dataFrame.radarObjectsPos;
    laneBoundaries      = dataFrame.laneBoundaries;
    trackPositions      = dataFrame.trackPositions;
    trackVelocities     = dataFrame.trackVelocities;
    trackCovariances    = dataFrame.trackCovariances;
    trackLabels         = dataFrame.trackLabels;
    MIOlabel            = dataFrame.MIOlabel;
    MIOposition         = dataFrame.MIOposition;
    MIOvelocity         = dataFrame.MIOvelocity;
end
```

See Also

Objects

`birdsEyePlot`

Functions

`plotCoverageArea` | `plotDetection` | `plotLaneBoundary` | `plotTrack`

More About

- “Coordinate Systems in Automated Driving Toolbox” on page 1-2

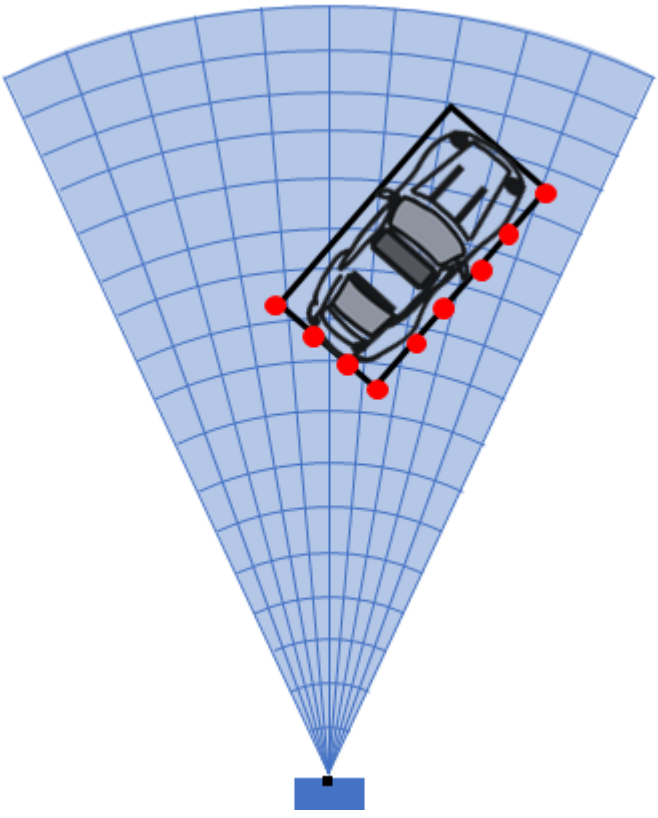
- “Visualize Sensor Data and Tracks in Bird's-Eye Scope” on page 3-2

Extended Object Tracking of Highway Vehicles with Radar and Camera

This example shows you how to track highway vehicles around an ego vehicle. Vehicles are extended objects, whose dimensions span multiple sensor resolution cells. As a result, the sensors report multiple detections of these objects in a single scan. In this example, you will use different extended object tracking techniques to track highway vehicles and evaluate the results of their tracking performance.

Introduction

In conventional tracking approaches such as global nearest neighbor (`multiObjectTracker`, `trackerGNN`), joint probabilistic data association (`trackerJPDA`) and multi-hypothesis tracking (`trackerTOMHT`), tracked objects are assumed to return one detection per sensor scan. With the development of sensors that have better resolution, such as a high-resolution radar, the sensors typically return more than one detection of an object. For example, the image below depicts multiple detections for a single vehicle that spans multiple radar resolution cells. In such cases, the technique used to track the objects is known as extended object tracking [1].



The key benefit of using a high-resolution sensor is getting more information about the object, such as its dimensions and orientation. This additional information can improve the probability of detection and reduce the false alarm rate.

Extended objects present new challenges to conventional trackers, because these trackers assume a single detection per object per sensor. In some cases, you can cluster the sensor data to provide the

conventional trackers with a single detection per object. However, by doing so, the benefit of using a high-resolution sensor may be lost.

In contrast, extended object trackers can handle multiple detections per object. In addition, these trackers can estimate not only the kinematic states, such as position and velocity of the object, but also the dimensions and orientation of the object. In this example, you track vehicles around the ego vehicle using the following trackers:

- A conventional multi-object tracker using a point-target model, `multiObjectTracker`
- A GGIW-PHD (Gamma Gaussian Inverse Wishart PHD) tracker, `trackerPHD` with `ggiwphd` filter
- A GM-PHD (Gaussian mixture PHD) tracker, `trackerPHD` with `gmphd` filter using rectangular target model

You will evaluate the tracking results of all trackers using `trackErrorMetrics` and `trackAssignmentMetrics`, which provide multiple measures of effectiveness of a tracker. You will also evaluate the results using the Optimal SubPattern Assignment Metric (OSPA), `trackOSPAMetric`, which aims to evaluate the performance of a tracker using a combined score.

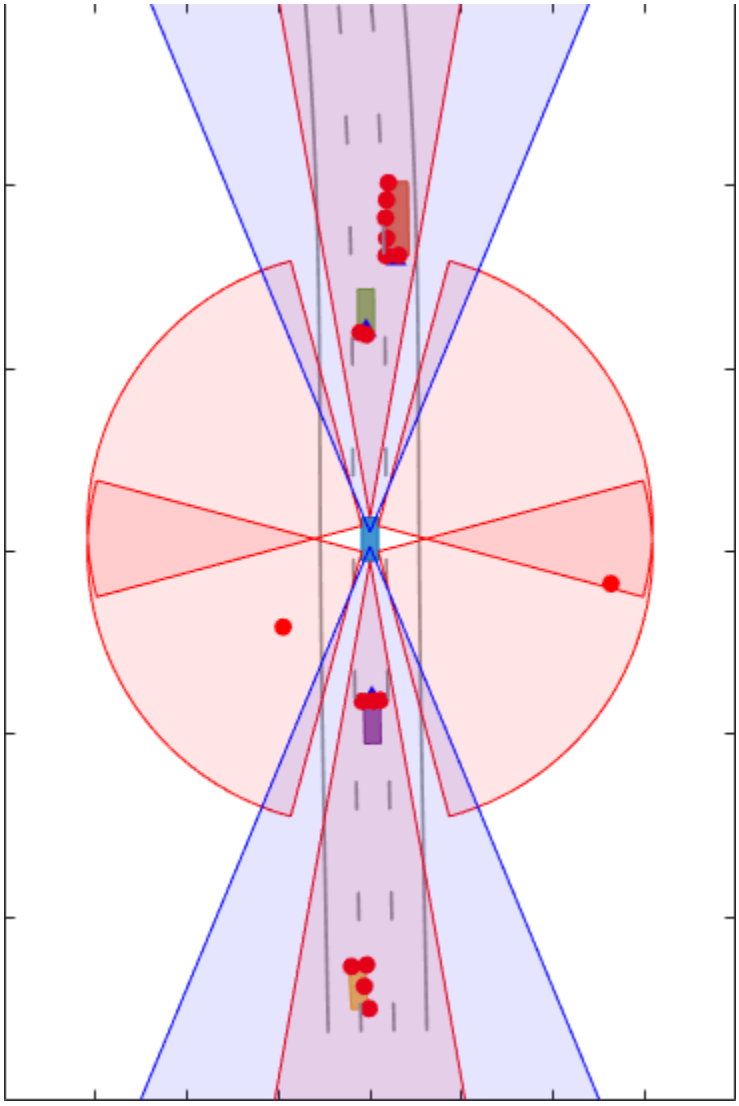
Setup

Scenario

In this example, there is an ego vehicle and four other vehicles: a vehicle ahead of the ego vehicle in the center lane, a vehicle behind the ego vehicle in the center lane, a truck ahead of the ego vehicle in the right lane and an overtaking vehicle in the left lane.

In this example, you simulate an ego vehicle that has 6 radar sensors and 2 vision sensors covering the 360-degree field of view. The sensors have some overlap and some coverage gap. The ego vehicle is equipped with a long-range radar sensor and a vision sensor on both the front and back of the vehicle. Each side of the vehicle has two short-range radar sensors, each covering 90 degrees. One sensor on each side covers from the middle of the vehicle to the back. The other sensor on each side covers from the middle of the vehicle forward.

```
% Create the scenario
exPath = fullfile(matlabroot,'examples','driving_fusion','main');
addpath(exPath)
[scenario, egoVehicle, sensors] = helperCreateScenario;

% Create the display object
display = helperExtendedTargetTrackingDisplay;

% Create the Animation writer to record each frame of the figure for
% animation writing. Set 'RecordGIF' to true to enable GIF writing.
gifWriter = helperGIFWriter('Figure',display.Figure,...
    'RecordGIF',false);
```



Metrics

In this example, you use some key metrics to assess the tracking performance of each tracker. In particular, you assess the trackers based on their accuracy in estimating the positions, velocities, dimensions (length and width) and orientations of the objects. These metrics can be evaluated using the `trackErrorMetrics` class. To define the error of a tracked target from its ground truth, this example uses a 'custom' error function, `helperExtendedTargetError`, listed at the end of this example.

You will also assess the performance based on metrics such as number of false tracks or redundant tracks. These metrics can be calculated using the `trackAssignmentMetrics` class. To define the distance between a tracked target and a truth object, this example uses a 'custom' error function, `helperExtendedTargetDistance`, listed at the end of this example. The function defines the distance metric as the sum of distances in position, velocity, dimension and yaw.

`trackErrorMetrics` and `trackAssignmentMetrics` provide multiple measures of effectiveness of a tracking algorithm. You will also assess the performance based on the Optimal SubPattern

Assignment Metric (OSPA), which provides a single score value for the tracking algorithm at each time step. This metric can be calculated using the `trackOSPAMetric` class. The 'custom' distance function defined for OSPA is same as the assignment metrics.

```
% actorProfiles to provide the error function size information about each
% truth.
aProfiles = actorProfiles(scenario);

% Function to return the errors given track and truth.
errorFcn = @(track,truth)helperExtendedTargetError(track,truth,aProfiles);

% Function to return the distance between track and truth
distFcn = @(track,truth)helperExtendedTargetDistance(track,truth,aProfiles);

% Function to return the IDs from the ground truth. The default
% identifier assumes that the truth is identified with PlatformID. In
% drivingScenario, truth is identified with an ActorID.
truthIdFcn = @(x)[x.ActorID];

% Create metrics object.
tem = trackErrorMetrics(...
    'ErrorFunctionFormat','custom',...
    'EstimationErrorLabels',{'PositionError','VelocityError','DimensionsError','YawError'},...
    'EstimationErrorFcn',errorFcn,...
    'TruthIdentifierFcn',truthIdFcn);

tam = trackAssignmentMetrics(...
    'DistanceFunctionFormat','custom',...
    'AssignmentDistanceFcn',distFcn,...
    'DivergenceDistanceFcn',distFcn,...
    'TruthIdentifierFcn',truthIdFcn,...
    'AssignmentThreshold',30,...
    'DivergenceThreshold',35);

% Create ospa metric object
tom = trackOSPAMetric(...
    'Distance','custom',...
    'DistanceFcn',distFcn,...
    'TruthIdentifierFcn',truthIdFcn);
```

Point Object Tracker

The `multiObjectTracker` System object™ assumes one detection per object per sensor and uses a global nearest neighbor approach to associate detections to tracks. It assumes that every object can be detected at most once by a sensor in a scan. In this case, the simulated radar sensors have a high enough resolution to generate multiple detections per object. If these detections are not clustered, the tracker generates multiple tracks per object. Clustering returns one detection per cluster, at the cost of having a larger uncertainty covariance and losing information about the true object dimensions. Clustering also makes it hard to distinguish between two objects when they are close to each other, for example, when one vehicle passes another vehicle.

```
trackerRunTimes = zeros(0,3);
ospaMetric = zeros(0,3);

% Create a multiObjectTracker
tracker = multiObjectTracker(...
    'FilterInitializationFcn', @helperInitPointFilter, ...
```

```
    'AssignmentThreshold', 30, ...
    'ConfirmationThreshold', [4 5], ...
    'DeletionThreshold', 3);

% Reset the random number generator for repeatable results
seed = 2018;
S = rng(seed);
timeStep = 1;

% For multiObjectTracker, the radar reports in Ego Cartesian frame and does
% not report velocity. This allows us to cluster detections from multiple
% sensors.
for i = 1:6
    sensors{i}.HasRangeRate = false;
    sensors{i}.DetectionCoordinates = 'Ego Cartesian';
end
```

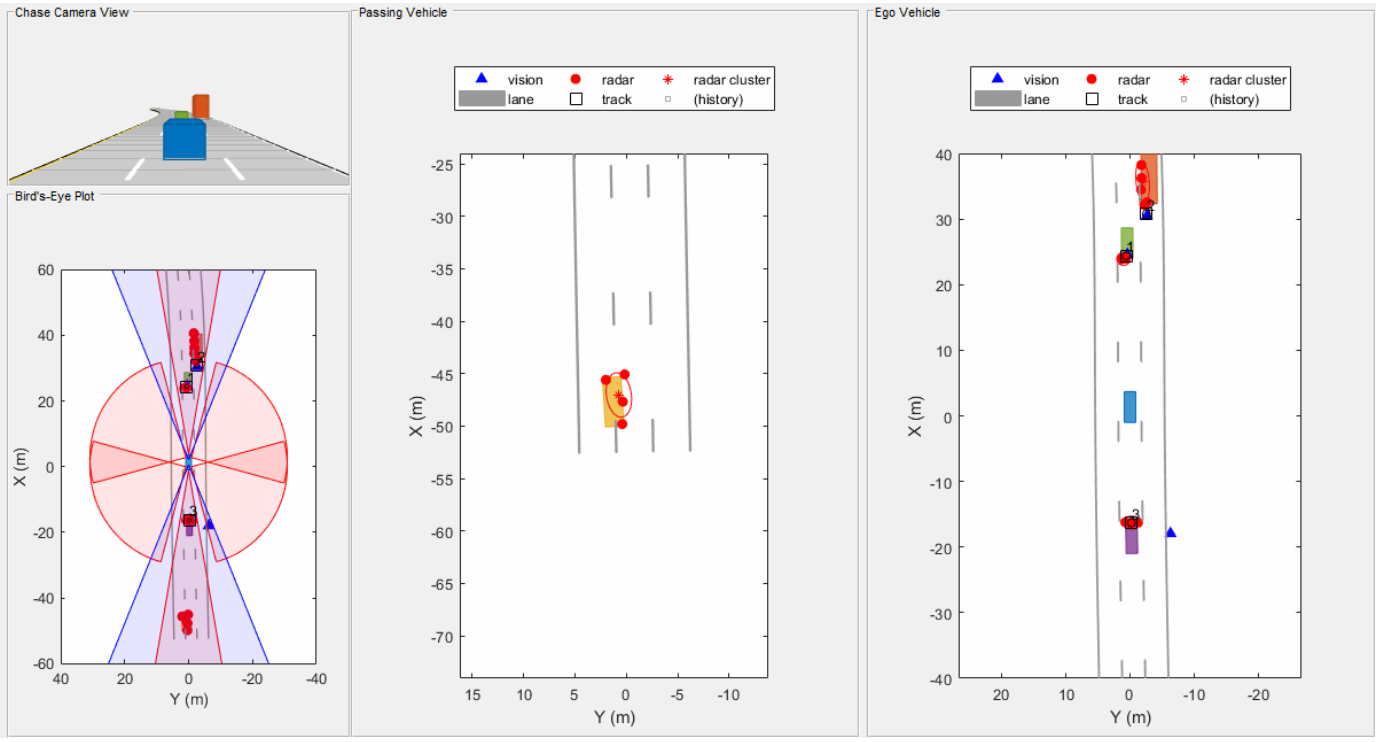
Run the scenario

```
while advance(scenario) && ishghandle(display.Figure)
    % Get the scenario time
    time = scenario.SimulationTime;

    % Collect detections from the ego vehicle sensors
    [detections,isValidTime] = helperDetect(sensors, egoVehicle, time);

    % Update the tracker if there are new detections
    if any(isValidTime)
        % Detections must be clustered first for the point tracker
        detectionClusters = helperClusterRadarDetections(detections);

        % Update the tracker
        tic
        % confirmedTracks are in scenario coordinates
        confirmedTracks = updateTracks(tracker, detectionClusters, time);
        t = toc;

        % Update the metrics
        % a. Obtain ground truth
        groundTruth = scenario.Actors(2:end); % All except Ego

        % b. Update assignment metrics
        tam(confirmedTracks,groundTruth);
        [trackIDs,truthIDs] = currentAssignment(tam);

        % c. Update error metrics
        tem(confirmedTracks,trackIDs,groundTruth,truthIDs);

        % d. Update ospa metric
        ospaMetric(timeStep,1) = tom(confirmedTracks, groundTruth);

        % Update bird's-eye-plot
        % Convert tracks to ego coordinates for display
        confirmedTracksEgo = helperConvertToEgoCoordinates(egoVehicle, confirmedTracks);
        display(egoVehicle, sensors, detections, confirmedTracksEgo, detectionClusters);
        drawnow;

        % Record tracker run times
```

```
            trackerRunTimes(timeStep,1) = t;
            timeStep = timeStep + 1;

            % Capture frames for animation
            gifWriter();
        end
    end

    % Capture the cumulative track metrics. The error metrics show the averaged
    % value of the error over the simulation.
    assignmentMetricsMOT = tam.trackMetricsTable;
    errorMetricsMOT = tem.cumulativeTruthMetrics;

    % Write GIF if requested
    writeAnimation(gifWriter,'multiObjectTracking');
```

These results show that, with clustering, the tracker can keep track of the objects in the scene. However, it also shows that the track associated with the overtaking vehicle (yellow) moves from the front of the vehicle at the beginning of the scenario to the back of the vehicle at the end. At the beginning of the scenario, the overtaking vehicle is behind the ego vehicle (blue), so radar and vision detections are made from its front. As the overtaking vehicle passes the ego vehicle, radar detections are made from the side of the overtaking vehicle and then from its back, and the track moves to the back of the vehicle.

You can also see that the clustering is not perfect. When the passing vehicle passes the vehicle that is behind the ego vehicle (purple), both tracks are slightly shifted to the left due to the imperfect clustering. A redundant track is created on the track initially due to multiple clusters created when part of the side edge is missed. Also, a redundant track appears on the passing vehicle during the end because the distances between its detections increase.



GGIW-PHD Extended Object Tracker

In this section, you use a GGIW-PHD tracker (`trackerPHD` with `ggiwphd`) to track objects. Unlike `multiObjectTracker`, which uses one filter per track, the GGIW-PHD is a multi-target filter which describes the probability hypothesis density (PHD) of the scenario. To model the extended target, GGIW-PHD uses the following distributions:

Gamma: Positive value to describe expected number of detections.

Gaussian: State vector to describe target's kinematic state

Inverse-Wishart: Positive-definite matrix to describe the elliptical extent.

The model assumes that each distribution is independent of each other. Thus, the probability hypothesis density (PHD) in GGIW-PHD filter is described by a weighted sum of the probability density functions of several GGIW components.

A PHD tracker requires calculating the detectability of each component in the density. The calculation of detectability requires configurations of each sensor used with the tracker. You define these configurations for `trackerPHD` using the `trackingSensorConfiguration` class. Review the `helperCreateSensorConfigurations` function to see how sensor properties can be utilized to define the sensor configurations for the tracker.

```
% Set up sensor configurations
%
sensorConfigurations = helperCreateSensorConfigurations(sensors,egoVehicle);

% The transform function and filter initialization functions are state and
% filter dependent. Therefore, they are not set in the helper function.
for i = 1:numel(sensorConfigurations)
    % You can use a different technique to initialize a filter for each
    % sensor by using a different function for each configuration.
    sensorConfigurations{i}.FilterInitializationFcn = @helperInitGGIWFilter;

    % Tracks are defined in constant turn-rate state-space in the scenario
    % coordinates. The MeasurementFcn for constant turn-rate model can be
    % used as the transform function.
    sensorConfigurations{i}.SensorTransformFcn = @ctmeas;
end
```

Define the tracker.

In contrast to a point object tracker, which usually takes into account one partition (cluster) of detections, the trackerPHD creates multiple possible partitions of a set of detections and evaluate it against the current components in the PHD filter. The 3 and 5 in the function below defines the lower and upper Mahalanobis distance between detections. This is equivalent to defining that each cluster of detection must be a minimum of 3 resolutions apart and maximum of 5 resolutions apart from each other.

```
partFcn = @(x)partitionDetections(x,3,5);

tracker = trackerPHD('SensorConfigurations', sensorConfigurations,...
    'PartitioningFcn',partFcn,...
    'AssignmentThreshold',220,...% Minimum negative log-likelihood of a detection cell (multiple
    'ExtractionThreshold',0.8,...% Weight threshold of a filter component to be declared a track
    'ConfirmationThreshold',0.95,...% Weight threshold of a filter component to be declared a co
```

```
    'MergingThreshold',50,...% Threshold to merge components
    'HasSensorConfigurationsInput',true... % Tracking is performed in scenario frame and hence se
    );
```

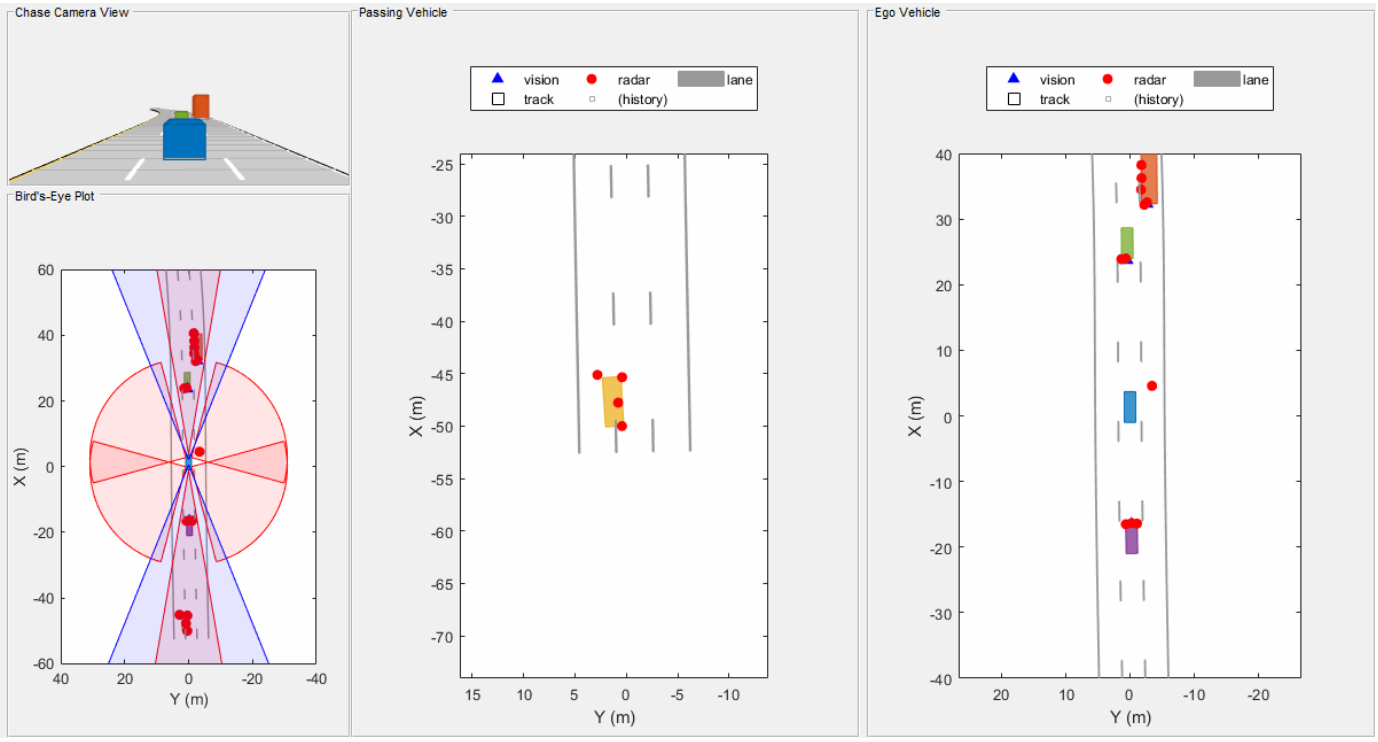
Run the simulation.

```
% Release and restart all objects.
restart(scenario);
release(tem);
release(tam);
% No penality for trackerPHD
tam.AssignmentThreshold = tam.AssignmentThreshold - 2;
release(display);
display.PlotClusteredDetection = false;
gifWriter.pFrames = {};
for i = 1:numel(sensors)
    release(sensors{i});
    if i <= 6
        sensors{i}.HasRangeRate = true;
        sensors{i}.DetectionCoordinates = 'Sensor spherical';
    end
end

% Restore random seed.
rng(seed)

% First time step
timeStep = 1;

% Run the scenario
while advance(scenario) && ishghandle(display.Figure)
    % Get the scenario time
    time = scenario.SimulationTime;

    % Get the poses of the other vehicles in ego vehicle coordinates
    ta = targetPoses(egoVehicle);

    % Collect detections from the ego vehicle sensors
    [detections, isValidTime, configurations] = helperDetect(sensors, egoVehicle, time, sensorCo

    % Update the tracker with all the detections. Note that there is no
    % need to cluster the detections before passing them to the tracker.
    % Also, the sensor configurations are passed as an input to the
    % tracker.
    tic
    % confirmedTracks are in scenario coordinates
    confirmedTracks = tracker(detections,configurations,time);
    t = toc;

    % Update the metrics
    % a. Obtain ground truth
    groundTruth = scenario.Actors(2:end); % All except Ego

    % b. Update assignment metrics
    tam(confirmedTracks,groundTruth);
    [trackIDs,truthIDs] = currentAssignment(tam);

    % c. Update error metrics
```

```
        tem(confirmedTracks,trackIDs,groundTruth,truthIDs);

        % d. Update ospa metric
        ospaMetric(timeStep,2) = tom(confirmedTracks, groundTruth);

        % Update the bird's-eye plot
        % Convert tracks to ego coordinates for display
        confirmedTracksEgo = helperConvertToEgoCoordinates(egoVehicle, confirmedTracks);
        display(egoVehicle, sensors, detections, confirmedTracksEgo);
        drawnow;

        % Record tracker run times
        trackerRunTimes(timeStep,2) = t;
        timeStep = timeStep + 1;

        % Capture frames for GIF
        gifWriter();
    end

    % Capture the truth and track metrics tables
    assignmentMetricsGGIWPHD = tam.trackMetricsTable;
    errorMetricsGGIWPHD = tem.cumulativeTruthMetrics;

    % Write GIF if requested
    writeAnimation(gifWriter,'ggiwphdTracking');
```

These results show that the GGIW-PHD can handle multiple detections per object per sensor, without the need to cluster these detections first. Moreover, by using the multiple detections, the tracker estimates the position, velocity, dimensions and orientation of each object. The dashed elliptical shape in the figure demonstrates the expected extent of the target. The filter initialization function specifies multiple possible sizes and their relative weights using multiple components. The list can be expanded to add more sizes with added computational complexity. In contrast, you can also initialize one component per detection with a higher uncertainty in dimensions. This will enable the tracker to estimate the dimensions of the objects automatically. That said, the accuracy of the estimate will depend on the observability of the target dimensions and is susceptible to shrinkage and enlargement of track dimensions as the targets move around the ego vehicle.

The GGIW-PHD filter assumes that detections are distributed around the target's elliptical center. Therefore, the tracks tend to follow observable portions of the vehicle. Such observable portions include rear face of the vehicle that is directly ahead of the ego vehicle or the front face of the vehicle directly behind the ego vehicle for example, the rear and front face of the vehicle directly ahead and behind of the ego vehicle respectively. In contrast, the length and width of the passing vehicle was fully observed during the simulation. Therefore, its estimated ellipse has a better overlap with the actual shape.



GM-PHD Rectangular Object Tracker

In this section, you use a GM-PHD tracker (`trackerPHD` with `gmphd`) and a rectangular target model (`initctrectgmphd` (Sensor Fusion and Tracking Toolbox)) to track objects. Unlike `ggiwphd`, which uses an elliptical shape to track extent, `gmphd` allows you to use a Gaussian distribution to define the shape of your choice. The rectangular target model is defined by motion models, `ctrect` (Sensor Fusion and Tracking Toolbox) and `ctrectjac` (Sensor Fusion and Tracking Toolbox) and measurement models, `ctrectmeas` (Sensor Fusion and Tracking Toolbox) and `ctrectmeasjac` (Sensor Fusion and Tracking Toolbox).

The sensor configurations defined for trackerPHD earlier remain the same, except for definition of `SensorTransformFcn` and `FilterInitializationFcn`.

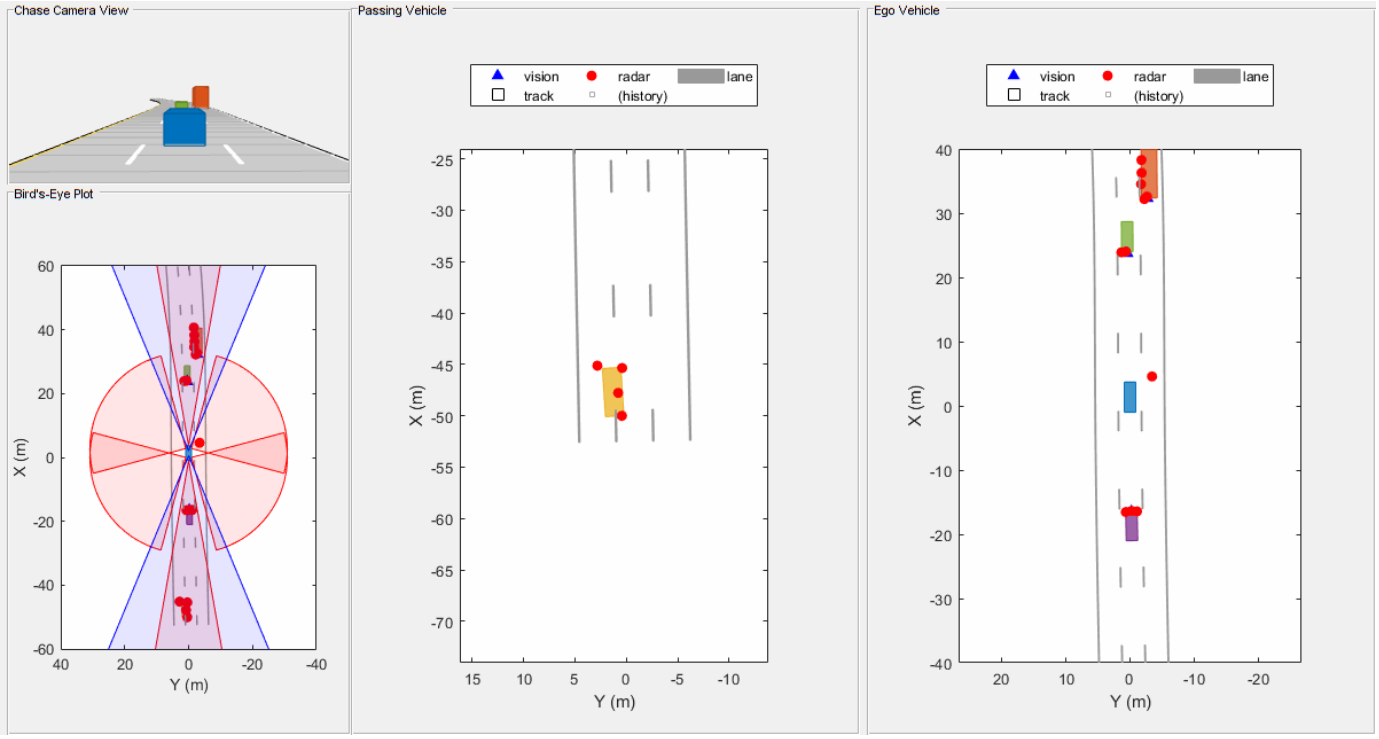
```
for i = 1:numel(sensorConfigurations)
    sensorConfigurations{i}.FilterInitializationFcn = @helperInitRectangularFilter; % Initialize
    sensorConfigurations{i}.SensorTransformFcn = @ctrectcorners; % Use corners to calculate detec
end

% Define tracker using new sensor configurations
tracker = trackerPHD('SensorConfigurations', sensorConfigurations,...
    'PartitioningFcn',partFcn,...
    'AssignmentThreshold',220,...% Minimum negative log-likelihood of a detection cell to add bi
    'ExtractionThreshold',0.8,...% Weight threshold of a filter component to be declared a track
    'ConfirmationThreshold',0.95,...% Weight threshold of a filter component to be declared a co
    'MergingThreshold',50,...% Threshold to merge components
    'HasSensorConfigurationsInput',true... % Tracking is performed in scenario frame and hence se
    );

% Release and restart all objects.
restart(scenario);
```

```
for i = 1:numel(sensors)
    release(sensors{i});
end
release(tem);
release(tam);
release(display);
display.PlotClusteredDetection = false;
gifWriter.pFrames = {};

% Restore random seed.
rng(seed)

% First time step
timeStep = 1;

% Run the scenario
while advance(scenario) && ishghandle(display.Figure)
    % Get the scenario time
    time = scenario.SimulationTime;

    % Get the poses of the other vehicles in ego vehicle coordinates
    ta = targetPoses(egoVehicle);

    % Collect detections from the ego vehicle sensors
    [detections, isValidTime, configurations] = helperDetect(sensors, egoVehicle, time, sensorCo

    % Update the tracker with all the detections. Note that there is no
    % need to cluster the detections before passing them to the tracker.
    % Also, the sensor configurations are passed as an input to the
    % tracker.
    tic
    % confirmedTracks are in scenario coordinates
    confirmedTracks = tracker(detections,configurations,time);
    t = toc;

    % Update the metrics
    % a. Obtain ground truth
    groundTruth = scenario.Actors(2:end); % All except Ego

    % b. Update assignment metrics
    tam(confirmedTracks,groundTruth);
    [trackIDs,truthIDs] = currentAssignment(tam);

    % c. Update error metrics
    tem(confirmedTracks,trackIDs,groundTruth,truthIDs);

    % d. Update ospa metric
    ospaMetric(timeStep,3) = tom(confirmedTracks, groundTruth);

    % Update the bird's-eye plot
    % Convert tracks to ego coordinates for display
    confirmedTracksEgo = helperConvertToEgoCoordinates(egoVehicle, confirmedTracks);
    display(egoVehicle, sensors, detections, confirmedTracksEgo);
    drawnow;

    % Record tracker run times
    trackerRunTimes(timeStep,3) = t;
    timeStep = timeStep + 1;
```

```
    % Capture frames for GIF
    gifWriter();
end

% Capture the truth and track metrics tables
assignmentMetricsGMPHD = tam.trackMetricsTable;
errorMetricsGMPHD = tem.cumulativeTruthMetrics;

% Write GIF if requested
writeAnimation(gifWriter,'gmphdTracking');

% Return the random number generator to its previous state
rng(S)
rmpath(exPath)
```

These results show that the GM-PHD can also handle multiple detections per object per sensor. Similar to GGIW-PHD, it also estimates the size and orientation of the object. The filter initialization function uses similar approach as GGIW-PHD tracker and initializes multiple components for different sizes.

You can notice that the estimated tracks, which are modeled as rectangles, have a good fit with the simulated ground truth object, depicted by the solid color patches. In particular, the tracks are able to correctly track the shape of the vehicle along with the kinematic center.

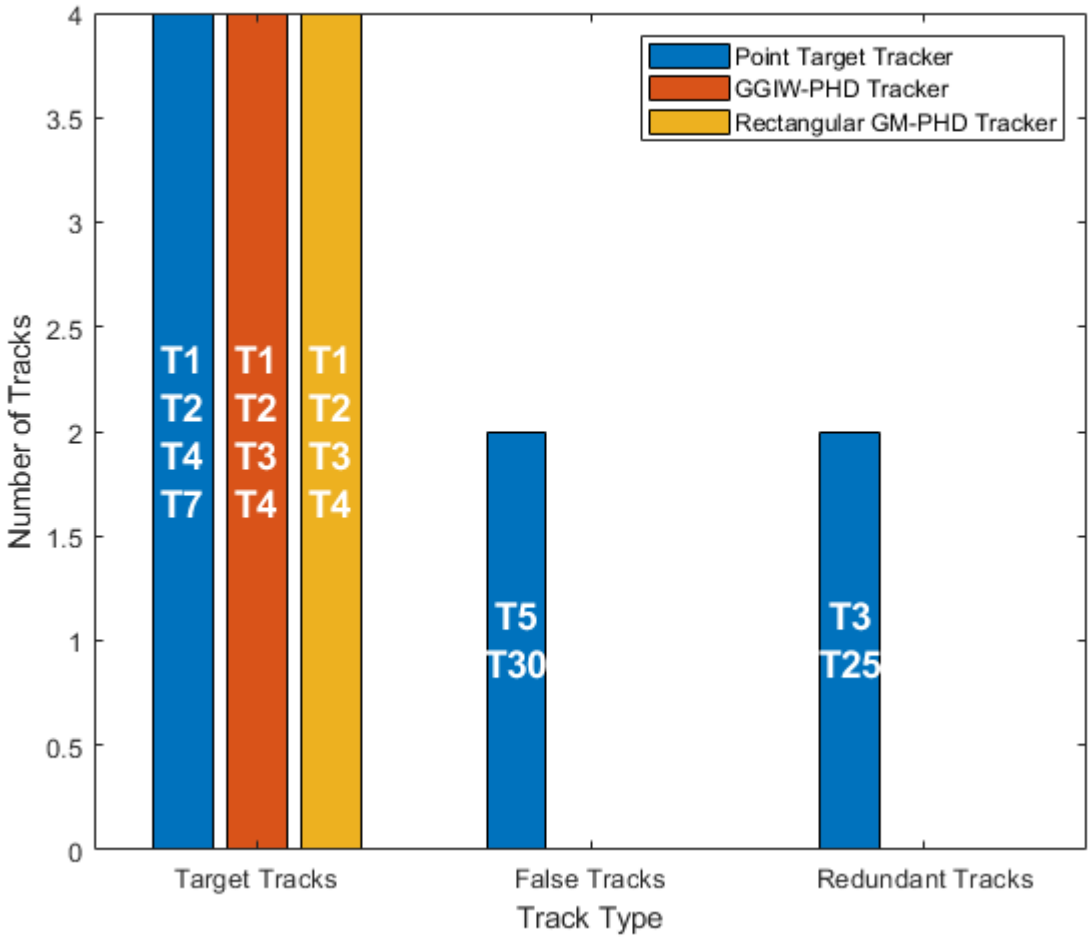


Evaluate Tracking Performance

Evaluate the tracking performance of each tracker using quantitative metrics such as the estimation error in position, velocity, dimensions and orientation. Also evaluate the track assignments using metrics such as redundant and false tracks.

Assignment metrics

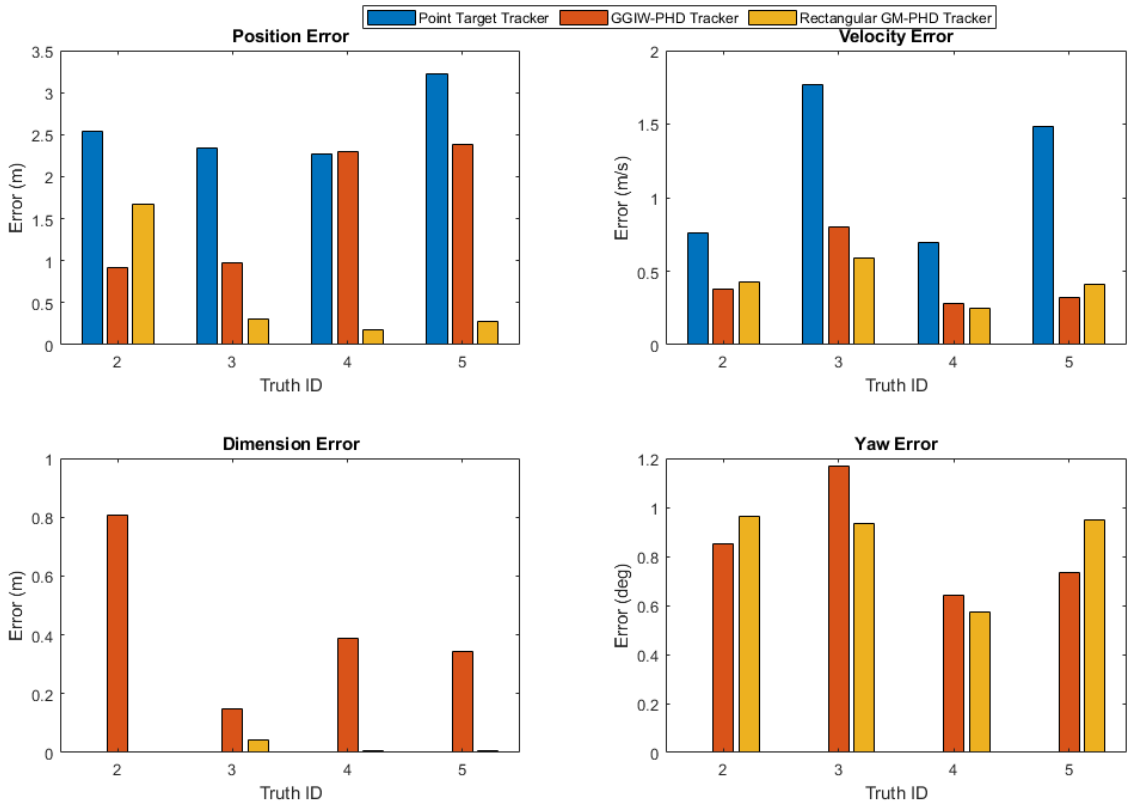
```
helperPlotAssignmentMetrics(assignmentMetricsMOT, assignmentMetricsGGIWPHD, assignmentMetricsGMPH
```



The assignment metrics illustrate that redundant and false tracks were initialized and confirmed by the point object tracker. These tracks result due to imperfect clustering, where detections belonging to the same target were clustered into more than one clustered detection. In contrast, the GGIW-PHD tracker and the GM-PHD tracker maintain tracks on all four targets and do not create any false or redundant tracks. These metrics show that both extended object trackers correctly partition the detections and associate them with the correct tracks.

Error metrics

```
helperPlotErrorMetrics(errorMetricsMOT, errorMetricsGGIWPHD, errorMetricsGMPHD);
```



The plot shows the average estimation errors for the three types of trackers used in this example. Because the point object tracker does not estimate the yaw and dimensions of the objects, they are now shown in the plots. The point object tracker is able to estimate the kinematics of the objects with a reasonable accuracy. The position error of the vehicle behind the ego vehicle is higher because it was dragged to the left when the passing vehicle overtakes this vehicle. This is also an artifact of imperfect clustering when the objects are close to each other.

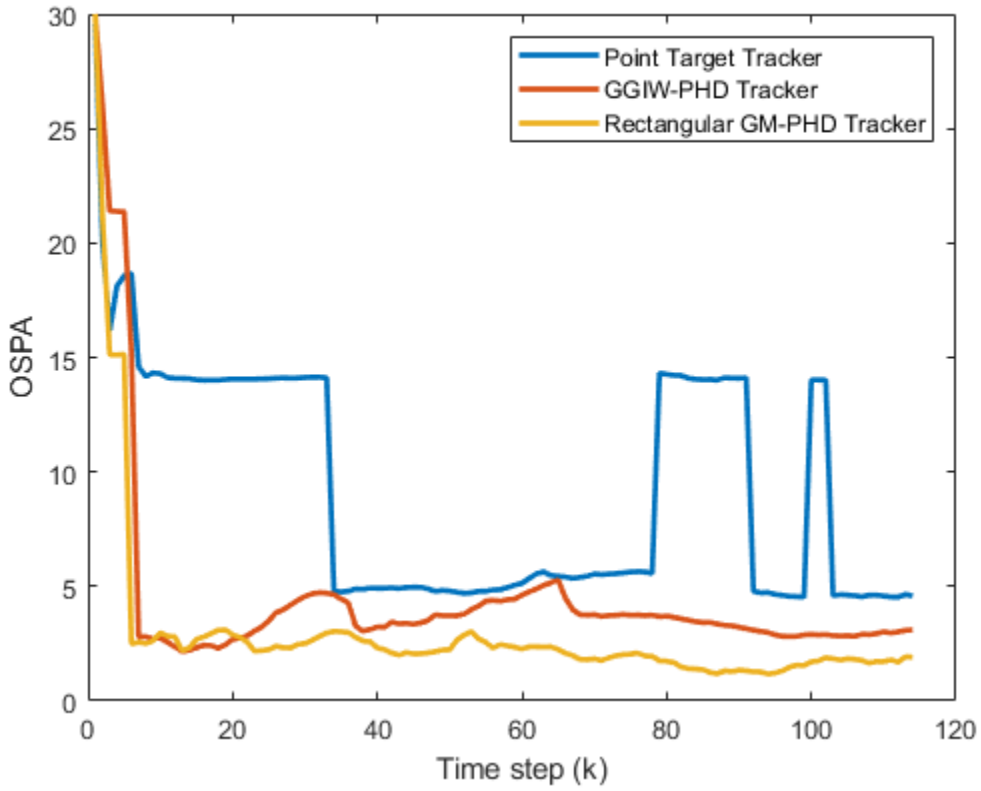
As described earlier, the GGIW-PHD tracker assumes that measurements are distributed around the object's extent, which results in center of the tracks on observable parts of the vehicle. This can also be seen in the position error metrics for TruthID 2 and 4. The tracker is able to estimate the dimensions of the object with about 0.3 meters accuracy for the vehicles ahead and behind the ego vehicle. Because of higher certainty defined for the vehicles' dimensions in the `helperInitGGIWFilter` function, the tracker does not collapse the length of these vehicles, even when the best-fit ellipse has a very low length. As passing vehicle (TruthID 3) was observed on all dimensions, its dimensions are measured more accurately than the other vehicles. However, as the passing vehicle maneuvers with respect to the ego vehicle, the error in yaw estimate is higher.

The GM-PHD in this example uses a rectangular shaped target model and uses received measurements to evaluate expected measurements on the boundary of the target. This model helps the tracker estimate the shape and orientation more accurately. However, the process of evaluating expected measurements on the edges of a rectangular target, is computationally more expensive.

OSPA Metric

As described earlier, the OSPA metric aims to describe the performance of a tracking algorithm using a single score. Notice that the OSPA sufficiently captures the performance of the tracking algorithm which decreases from GM-PHD to GGIW-PHD to the point-target tracker, as described using the error and assignment metrics.

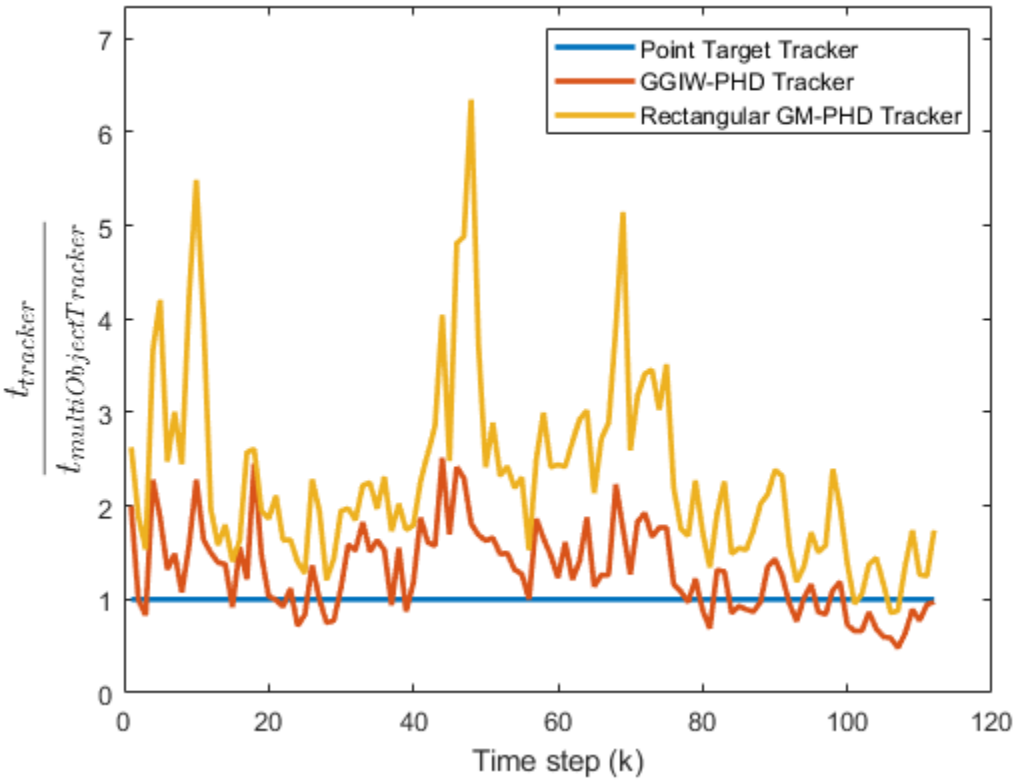
```
ospaFig = figure;
plot(ospaMetric,'LineWidth',2);
legend('Point Target Tracker','GGIW-PHD Tracker','Rectangular GM-PHD Tracker');
xlabel('Time step (k)');
ylabel('OSPA');
```



Compare Time Performance

Previously, you learned about different techniques, the assumptions they make about target models, and the resulting tracking performance. Now compare the run-times of the trackers. Notice that GGIW-PHD filter offers significant computational advantages over the GM-PHD, at the cost of decreased tracking performance.

```
runTimeFig = figure;
h = plot(trackerRunTimes(3:end,:)./trackerRunTimes(3:end,1),'LineWidth',2);
legend('Point Target Tracker','GGIW-PHD Tracker','Rectangular GM-PHD Tracker');
xlabel('Time step (k)');
ylabel('$$\frac{t_{tracker}}{t_{multiObjectTracker}}$$','interpreter','latex','fontsize',14);
ylim([0 max([h.YData]) + 1]);
```



Summary

This example showed how to track objects that return multiple detections in a single sensor scan using different approaches. These approaches can be used to track objects with high-resolution sensors, such as a radar or laser sensor.

References

[1] Granström, Karl, Marcus Baum, and Stephan Reuter. "Extended Object Tracking: Introduction, Overview and Applications." *Journal of Advances in Information Fusion*. Vol. 12, No. 2, December 2017.

[2] Granström, Karl, Christian Lundquist, and Umut Orguner. "Tracking rectangular and elliptical extended targets using laser measurements." 14th International Conference on Information Fusion. IEEE, 2011.

[3] Granström, Karl. "Extended target tracking using PHD filters." 2012

Supporting Functions

`helperExtendedTargetError`

Function to define the error between tracked target and the associated ground truth.

```
function [posError,velError,dimError,yawError] = helperExtendedTargetError(track,truth,aProfiles)
% Errors as a function of target track and associated truth.
```

```
% Get true information from the ground truth.
truePos = truth.Position(1:2)';
% Position is at the rear axle for all vehicles. We would like to compute
% the error from the center of the vehicle
rot = [cosd(truth.Yaw) -sind(truth.Yaw);sind(truth.Yaw) cosd(truth.Yaw)];
truePos = truePos + rot*[truth.Wheelbase/2;0];

trueVel = truth.Velocity(1:2);
trueYaw = truth.Yaw(:);
thisActor = truth.ActorID;
thisProfile = aProfiles([aProfiles.ActorID] == thisActor);
trueDims = [thisProfile.Length;thisProfile.Width];

% Get estimated value from track.
% GGIW-PHD tracker outputs a field 'Extent' and 'SourceIndex'
% GM-PHD tracker outputs 'SourceIndex', but not 'Extent'
% multiObjectTracker does not output 'SourceIndex'

if ~isfield(track,'SourceIndex')
    estPos = track.State([1 3]);
    estVel = track.State([2 4]);
    % No yaw or dimension information in multiObjectTracker.
    estYaw = nan;
    estDims = [nan;nan];
elseif isfield(track,'Extent') % trackerPHD with GGIWPHD
    estPos = track.State([1 3]);
    estVel = track.State([2 4]);
    estYaw = atan2d(estVel(2),estVel(1));
    d = eig(track.Extent);
    dims = 2*sqrt(d);
    estDims = [max(dims);min(dims)];
else % trackerPHD with GMPHD
    estPos = track.State(1:2);
    estYaw = track.State(4);
    estVel = [track.State(3)*cosd(estYaw);track.State(3)*sind(estYaw)];
    estDims = track.State(6:7);
end

% Compute 2-norm of error for each attribute.
posError = norm(truePos(:) - estPos(:));
velError = norm(trueVel(:) - estVel(:));
dimError = norm(trueDims(:) - estDims(:));
yawError = norm(trueYaw(:) - estYaw(:));
end
```

helperExtendedTargetDistance

Function to define the distance between a track and a ground truth.

```
function dist = helperExtendedTargetDistance(track,truth,aProfiles)
% This function computes the distance between track and a truth.

% Copyright 2019 The MathWorks, Inc.

% Errors in each aspect
```

```
[posError,velError,dimError,yawError] =  helperExtendedTargetError(track,truth,aProfiles);

% For multiObjectTracker, add a constant penalty for not estimating yaw
% and dimensions
if isnan(dimError)
    dimError = 1;
end
if isnan(yawError)
    yawError = 1;
end

% Distance is the sum of errors
dist = posError + velError + dimError + yawError;


end
```

helperInitGGIWFilter

Function to create a ggiwphd filter from a detection cell.

```
function phd = helperInitGGIWFilter(varargin)
% helperInitGGIWFilter A function to initialize the GGIW-PHD filter for the
% Extended Object Tracking example

% Create a ggiwphd filter using 5 states and the constant turn-rate models.
phd = ggiwphd(zeros(5,1),eye(5),...
    'StateTransitionFcn',@constturn,...
    'StateTransitionJacobianFcn',@constturnjac,...
    'MeasurementFcn',@ctmeas,...
    'MeasurementJacobianFcn',@ctmeasjac,...
    'HasAdditiveMeasurementNoise',true,...
    'HasAdditiveProcessNoise',false,...
    'ProcessNoise',diag([1 1 3]),...
    'MaxNumComponents',1000,...
    'ExtentRotationFcn',@extentRotFcn,...
    'PositionIndex',[1 3]);

% If the function is called with no inputs i.e. the predictive portion of
% the birth density, no components are added to the mixture.
if nargin == 0
    % Nullify to return 0 components.
    nullify(phd);
else
    % When called with detections input, add two components to the filter,
    % one for car and one for truck, More components can be added based on
    % prior knowledge of the scenario, example, pedestrian or motorcycle.
    % This is a "multi-model" type approach. Another approach can be to add
    % only 1 component with a higher covariance in the dimensions. The
    % later is computationally less demanding, but has a tendency to track
    % observable dimensions of the object. For example, if only the back is
    % visible, the measurement noise may cause the length of the object to
    % shrink.

    % Detections
    detections = varargin{1};
```

```
    % Enable elevation measurements to create a 3-D filter using
    % initctggiwphd
    if detections{1}.SensorIndex < 7
        for i = 1:numel(detections)
            detections{i}.Measurement = [detections{i}.Measurement(1);0;detections{i}.Measurement
            detections{i}.MeasurementNoise = blkdiag(detections{i}.MeasurementNoise(1,1),0.4,dete
            detections{i}.MeasurementParameters{1}(1).HasElevation = true;
        end
    end
    phd3d = initctggiwphd(detections);

    % Set states of the 2-D filter using 3-D filter
    phd.States = phd3d.States(1:5);
    phd.StateCovariances = phd3d.StateCovariances(1:5,1:5);

    phd.DegreesOfFreedom = 1000;
    phd.ScaleMatrices = (1000-4)*diag([4.7/2 1.8/2].^2);

    % Add truck dimensions as second component
    append(phd,phd);
    phd.ScaleMatrices(:,:,2) = (1000-4)*diag([8.1/2 2.45/2].^2);
    phd.GammaForgettingFactors = [1.03 1.03];

    % Relative weights of the components. Can be treated as probability of
    % existence of a car vs a truck on road.
    phd.Weights = [0.7 0.3];
end
end

function R = extentRotFcn(x,dT)
    % Rotation of the extent during prediction.
    w = x(5);
    theta = w*dT;
    R = [cosd(theta) -sind(theta);sind(theta) cosd(theta)];
end
```

helperInitRectangularFilter

Function to create a gmphd rectangular target filter from a detection cell

```
function filter = helperInitRectangularFilter(varargin)
% helperInitRectangularFilter A function to initialize the rectangular
% target PHD filter for the Extended Object Tracking example

% Copyright 2019 The MathWorks, Inc.

if nargin == 0
    % If called with no inputs, simply use the initctrectgmphd function to
    % create a PHD filter with no components.
    filter = initctrectgmphd;
    % Set process noise
    filter.ProcessNoise = diag([1 3]);
else
    % When called with detections input, add two components to the filter,
    % one for car and one for truck, More components can be added based on
```

```
    % prior knowledge of the scenario, example, pedestrian or motorcycle.
    % This is a "multi-model" type approach. Another approach can be to add
    % only 1 component with a higher covariance in the dimensions. The
    % later is computationally less demanding, but has a tendency to track
    % observable dimensions of the object. For example, if only the back is
    % visible, the measurement noise may cause the length of the object to
    % shrink.

    % Detections
    detections = varargin{1};

    % Create a GM-PHD filter with rectangular model
    filter = initctrectgmphd(detections);

    % Length width of a passenger car
    filter.States(6:7,1) = [4.7;1.8];

    % High certainty in dimensions
    lCov = 1e-4;
    wCov = 1e-4;
    lwCorr = 0.5;
    lwCov = sqrt(lCov*wCov)*lwCorr;
    filter.StateCovariances(6:7,6:7,1) = [lCov lwCov;lwCov wCov];

    % Add one more component by appending the filter with itself.
    append(filter,filter);

    % Set length and width to a truck dimensions
    filter.States(6:7,2) = [8.1;2.45];

    % Relative weights of each component
    filter.Weights = [0.7 0.3];
end

end
```

See Also

`drivingScenario` | `ggiwphd` | `multiObjectTracker` | `trackAssignmentMetrics` | `trackErrorMetrics`

More About

- “Sensor Fusion Using Synthetic Radar and Vision Data” on page 7-196
- “Track-Level Fusion of Radar and Lidar Data” on page 7-290

Track-to-Track Fusion for Automotive Safety Applications

This example shows how to fuse tracks from two vehicles in order to provide a more comprehensive estimate of the environment that can be seen by each vehicle. The example demonstrates the use of a track-level fuser and the object track data format. In this example, you use the driving scenario and models from Automated Driving Toolbox™ and the tracking and track fusion models from Sensor Fusion and Tracking Toolbox™.

Motivation

Automotive safety applications rely on the fusion of data from different sensor systems mounted on the vehicle. Individual vehicles fuse sensor detections by using either a centralized tracker or by taking a more decentralized approach and fusing tracks produced by individual sensors. In addition to intravehicle data fusion, the fusion of data from multiple vehicles provides added benefits, which include better coverage, situational awareness, and safety. [1] This intervehicle sensor fusion approach takes advantage of the variety of sensors and provides better coverage to each vehicle, because it uses data updated by sensors on other vehicles in the area. Governments and vehicle manufacturers have long recognized the need to share information between vehicles in order to increase automotive safety. For example, the Dedicated Short-Range Communications (DSRC) Service was established to provide a communications service for intervehicle information sharing. [2]

While sensor fusion across multiple vehicles is beneficial, most vehicles are required to meet certain safety requirements even if only internal sensors are available. Therefore, the vehicle is likely to be equipped with a tracker and/or a track fuser that provide situational awareness at the single vehicle level. As a result, the assumption made in this example is that vehicles share situational awareness by broadcasting tracks and performing track-to-track fusion.

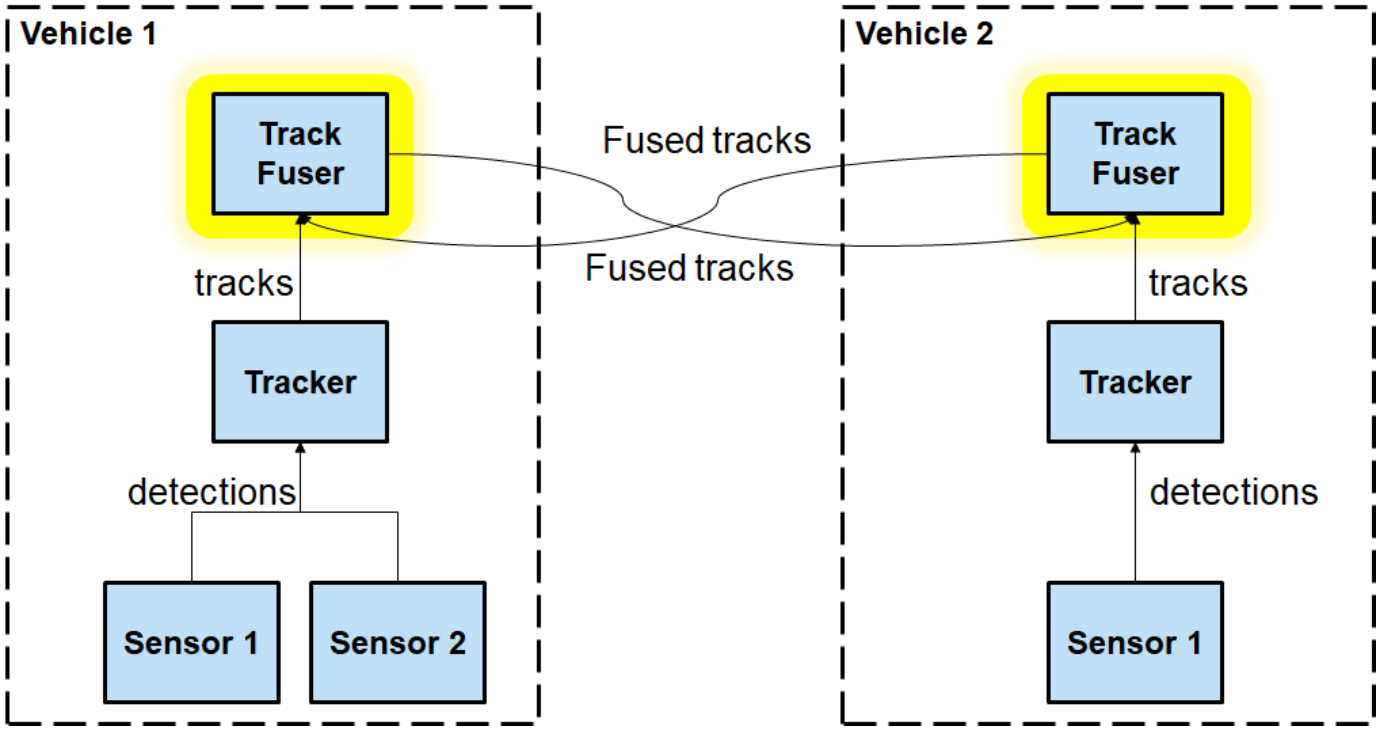
This example demonstrates the benefit of fusing tracks from two vehicles to enhance situational awareness and safety. Note that this example does not simulate the communications systems. Instead, the example assumes that a communications system provides the bandwidth required to transmit tracks between the two vehicles.

Track-to-Track Architecture

The following block diagram depicts the main functions in the two vehicles.

Vehicle 1 has two sensors, each providing detections to a local tracker. The tracker uses the detections from the local sensors to track objects and outputs these local tracks to the vehicle track fuser. Vehicle 2 has a single sensor, which feeds detections to the local tracker on vehicle 2. The local tracks from vehicle 2 are the input to the local track fuser on vehicle 2.

The track fuser on each vehicle fuses the local vehicle tracks with the tracks received from the other vehicle's track fuser. After each update, the track fuser on each vehicle broadcasts its fused tracks, which feed into the next update of the track fuser on the other vehicle.



In this example, you use a `trackerJPDA` (Sensor Fusion and Tracking Toolbox) object to define each vehicle tracker.

```
% Create trackers for each vehicle
v1Tracker = trackerJPDA('TrackerIndex',1, 'DeletionThreshold', [4 4]); % Vehicle 1 tracker
v2Tracker = trackerJPDA('TrackerIndex',2, 'DeletionThreshold', [4 4]); % Vehicle 2 tracker
posSelector = [1 0 0 0 0 0; 0 0 1 0 0 0];
```

Note that in this architecture, the fused tracks from one vehicle are used to update the fused tracks on the other vehicle. These fused tracks are then broadcast back to the first vehicle. To avoid rumor propagation, be careful how tracks from another vehicle are used to update the track fuser.

Consider the following rumor propagation example: at some update step, vehicle 1 tracks an object using its internal sensors. Vehicle 1 then fuses the object track and transmits it to vehicle 2, which now fuses the track with its own tracks and becomes aware of the object. Up to this point, this is exactly the goal of track-to-track fusion: to enhance the situational awareness of vehicle 2 with information from vehicle 1. Since vehicle 2 now knows about the object, it starts broadcasting the track as well, perhaps for the benefit of another vehicle (not shown in the example).

However, vehicle 1 now receives track information from vehicle 2 about the object that only vehicle 1 actually tracks. So, the track fuser on vehicle 1 has to be aware that the tracks it gets from vehicle 2 about this object do not actually contain any new information updated by an independent source. To make the distinction between tracks that contain new information and tracks that just repeat information, you must define vehicle 2 as an *external source* to the track fuser on vehicle 1. Similarly, vehicle 1 must be defined as an external source to the track fuser on vehicle 2. Furthermore, you need to define only tracks that are updated by a track fuser based on information from an internal source as *self-reported*. By doing so, the track fuser in each vehicle is able to ignore updates from tracks that bounce back and forth between the track fusers without any new information in them.

The local tracker of each vehicle tracks objects relative to the vehicle reference frame, called the ego frame. The track-to-track fusion is done at the scenario frame, which is the global level frame. The helper `egoToScenario` function transforms tracks from ego frame to the scenario frame. Similarly, the function `scenarioToEgo` transforms tracks from scenario frame to any of the ego frames. Both transformations rely on the `StateParameters` property of the `objectTrack` (Sensor Fusion and Tracking Toolbox) objects. Note that when the `trackFuser` object calculates the distance of a central track (in the scenario frame) to a local track (in any frame), it uses the `StateParameters` of the local track to perform the coordinate transformation.

To achieve the above `trackFuser` definitions, define the following sources as a `fuserSourceConfiguration` (Sensor Fusion and Tracking Toolbox) object.

```
% Define sources for each vehicle
v1TrackerConfiguration = fuserSourceConfiguration('SourceIndex',1,'IsInternalSource',true, ...
    "CentralToLocalTransformFcn", @scenarioToEgo, 'LocalToCentralTransformFcn', @egoToScenario);
v2FuserConfiguration = fuserSourceConfiguration('SourceIndex',4,'IsInternalSource',false);
v1Sources = {v1TrackerConfiguration; v2FuserConfiguration};
v2TrackerConfiguration = fuserSourceConfiguration('SourceIndex',2,'IsInternalSource',true, ...
    "CentralToLocalTransformFcn", @scenarioToEgo, 'LocalToCentralTransformFcn', @egoToScenario);
v1FuserConfiguration = fuserSourceConfiguration('SourceIndex',3,'IsInternalSource',false);
v2Sources = {v2TrackerConfiguration; v1FuserConfiguration};
```

You can now define each vehicle track fuser as a `trackFuser` (Sensor Fusion and Tracking Toolbox) object.

```
stateParams = struct('Frame','Rectangular','Position',[0 0 0],'Velocity',[0 0 0]);
v1Fuser = trackFuser('FuserIndex',3,...
    'MaxNumSources',2,'SourceConfigurations',v1Sources,...
    'StateFusion','Intersection','DeletionThreshold',[3 3],...
    'StateParameters',stateParams);
v2Fuser = trackFuser('FuserIndex',4,...
    'MaxNumSources',2,'SourceConfigurations',v2Sources,'StateFusion',...
    'Intersection','DeletionThreshold',[3 3],...
    'StateParameters',stateParams);

% Initialize the following variables
fusedTracks1 = objectTrack.empty(0,1);
fusedTracks2 = objectTrack.empty(0,1);
wasFuser1Updated = false;
wasFuser2Updated = false;
```

Scenario Definition

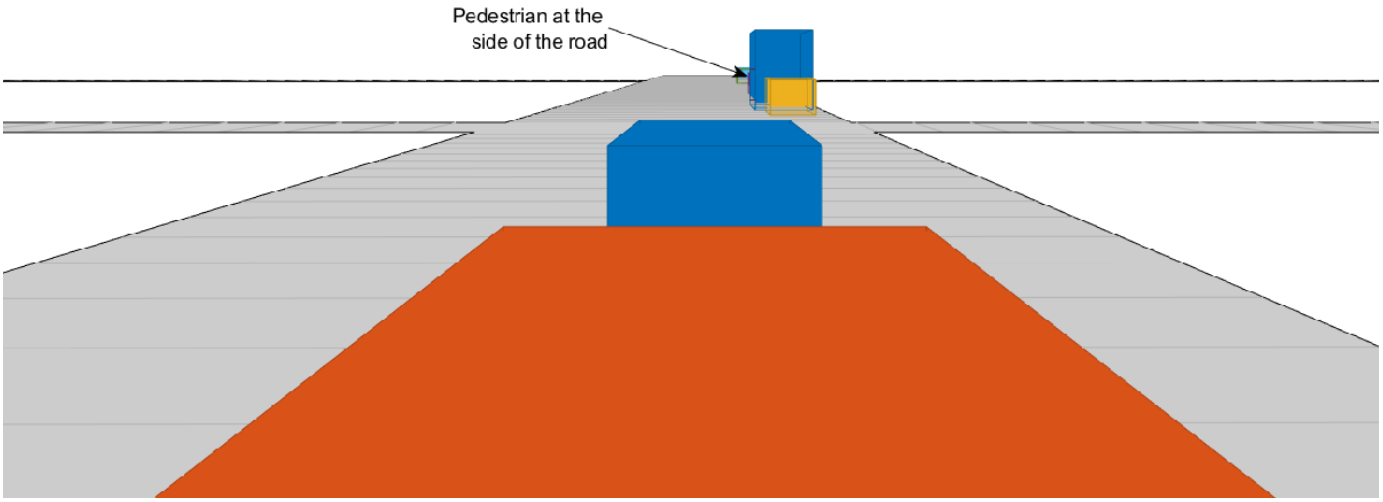
The following scenario shows two vehicles driving down a street. Vehicle 1 is in the lead, equipped with two forward-looking sensors: a short-range radar and a vision sensor. Vehicle 2, driving 10 meters behind vehicle 1, is equipped with a long-range radar. The right side of the street contains parked vehicles. A pedestrian stands between the vehicles. This pedestrian is shown as a dot at about X = 60 meters.

Due to the short distance between vehicle 2 and vehicle 1, most of the vehicle 2 radar sensor coverage is occluded by vehicle 1. As a result, most of the tracks that the track fuser on vehicle 2 maintains are first initialized by tracks broadcast from vehicle 1.

```
% Create the drivingScenario object and the two vehicles
[scenario, vehicle1, vehicle2] = createDrivingScenario;

% Create all the sensors
```

```
[sensors, numSensors, attachedVehicle] = createSensors(scenario);

% Create display
[f,plotters] = createT2TDisplay(scenario, sensors, attachedVehicle);
```

The following chase plot is seen from the point of view of the second vehicle. An arrow indicates the position of the pedestrian that is almost entirely occluded by the parked vehicles and the first vehicle.



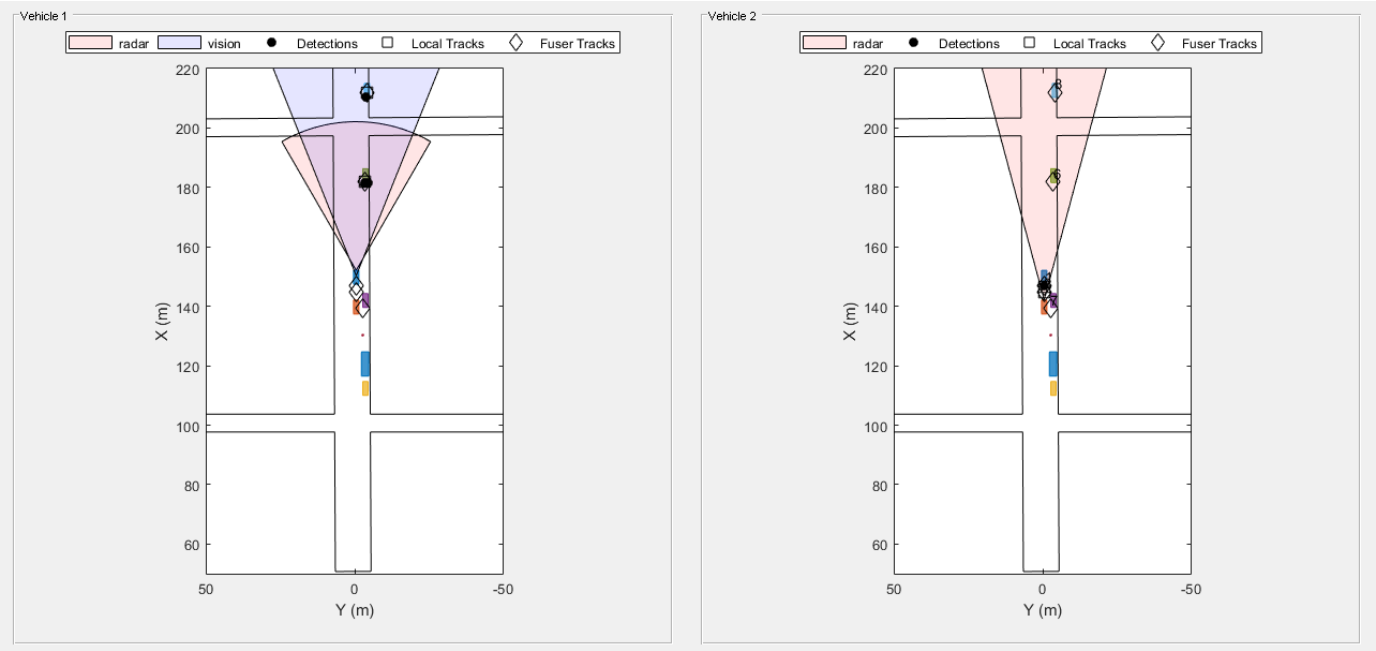

```
% Define each vehicle as a vehicle, sensors, a tracker, and plotters
v1 = struct('Actor', {vehicle1}, 'Sensors', {sensors(attachedVehicle==1)}, 'Tracker', {v1Tracker
v2 = struct('Actor', {vehicle2}, 'Sensors', {sensors(attachedVehicle==2)}, 'Tracker', {v2Tracker
```

Simulation and Results

The following code runs the simulation.

```
running = true;

% For repeatable results, set the random number seed
s = rng;
rng(2019)
snaptimes = [0.5, 2.6, 4.4, 6.3, inf];
snaps = cell(numel(snaptimes,1));
i = 1;
f.Visible = 'on';
while running && ishghandle(f)
    time  = scenario.SimulationTime;
```

```
        % Detect and track at the vehicle level
        [tracks1,wasTracker1Updated] = detectAndTrack(v1,time,posSelector);
        [tracks2,wasTracker2Updated] = detectAndTrack(v2,time,posSelector);

        % Keep the tracks from the previous fuser update
        oldFusedTracks1 = fusedTracks1;
        oldFusedTracks2 = fusedTracks2;

        % Update the fusers
        if wasTracker1Updated || wasFuser2Updated
            tracksToFuse1 = [tracks1;oldFusedTracks2];
            if isLocked(v1Fuser) || ~isempty(tracksToFuse1)
                [fusedTracks1,~,~,info1] = v1Fuser(tracksToFuse1,time);
                wasFuser1Updated = true;
                pos = getTrackPositions(fusedTracks1,posSelector);
                plotTrack(plotters.veh1FusePlotter,pos);
            else
                wasFuser1Updated = false;
                fusedTracks1 = objectTrack.empty(0,1);
            end
        else
            wasFuser1Updated = false;
            fusedTracks1 = objectTrack.empty(0,1);
        end
        if wasTracker2Updated || wasFuser1Updated
            tracksToFuse2 = [tracks2;oldFusedTracks1];
            if isLocked(v2Fuser) || ~isempty(tracksToFuse2)
                [fusedTracks2,~,~,info2] = v2Fuser(tracksToFuse2,time);
                wasFuser2Updated = true;
                pos = getTrackPositions(fusedTracks2,posSelector);
                ids = string([fusedTracks2.TrackID]');
                plotTrack(plotters.veh2FusePlotter,pos,ids);
            else
                wasFuser2Updated = false;
                fusedTracks2 = objectTrack.empty(0,1);
            end
        else
            wasFuser2Updated = false;
            fusedTracks2 = objectTrack.empty(0,1);
        end

        % Update the display
        updateT2TDisplay(plotters, scenario, sensors, attachedVehicle)

        % Advance the scenario one time step and exit the loop if the scenario is complete
        running = advance(scenario);

        % Snap a shot at required times
        if time >= snaptimes(i)
            snaps{i} = getframe(f);
            i = i + 1;
        end
    end
```

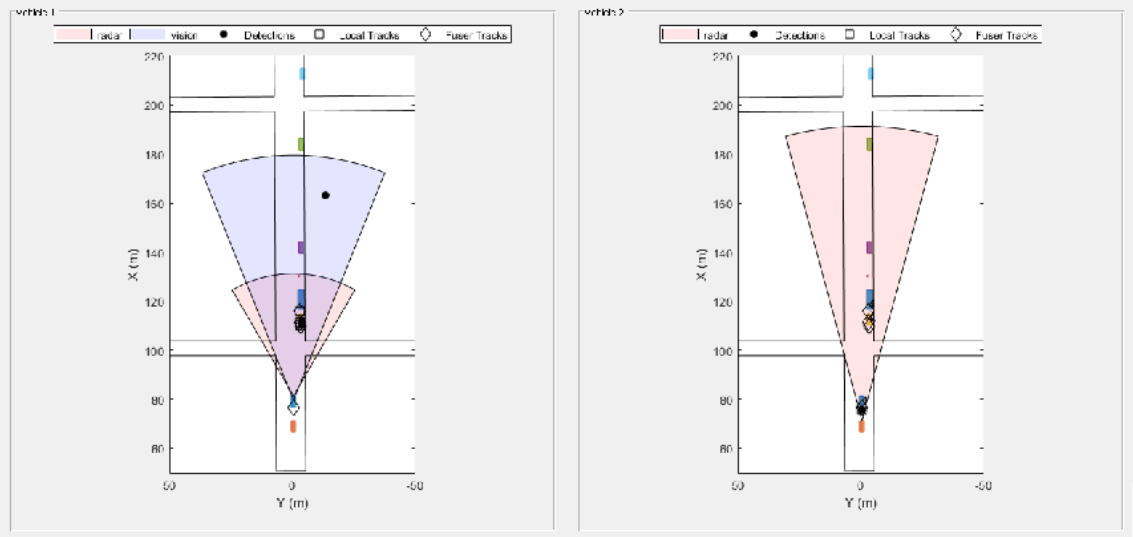


The figure shows the scene and tracking results at the end of the scenario.

Tracking at the Beginning of the Simulation

When the simulation begins, vehicle 1 detects the vehicles parked on the right side of the street, then the tracks associated with the parked vehicles are confirmed. At this time, the only object detected and tracked by vehicle 2 tracker is vehicle 1 immediately in front of it. Once the vehicle 1 track fuser confirms the tracks, it broadcasts them, and the vehicle 2 track fuser fuses them. As a result, vehicle 2 becomes aware of the parked vehicles before it can detect them on its own.

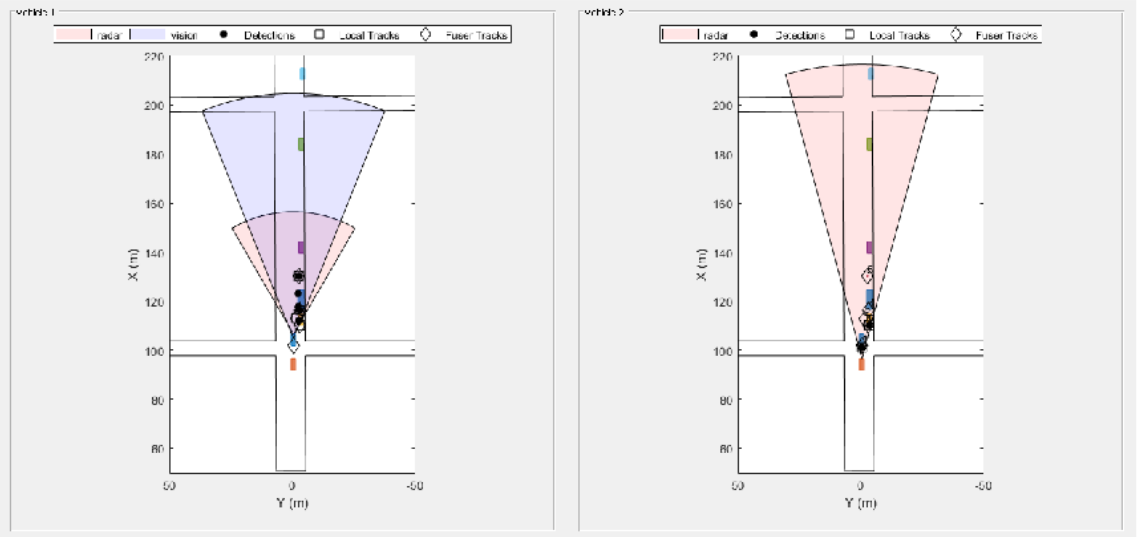
```
showsnap(snaps, 1)
```



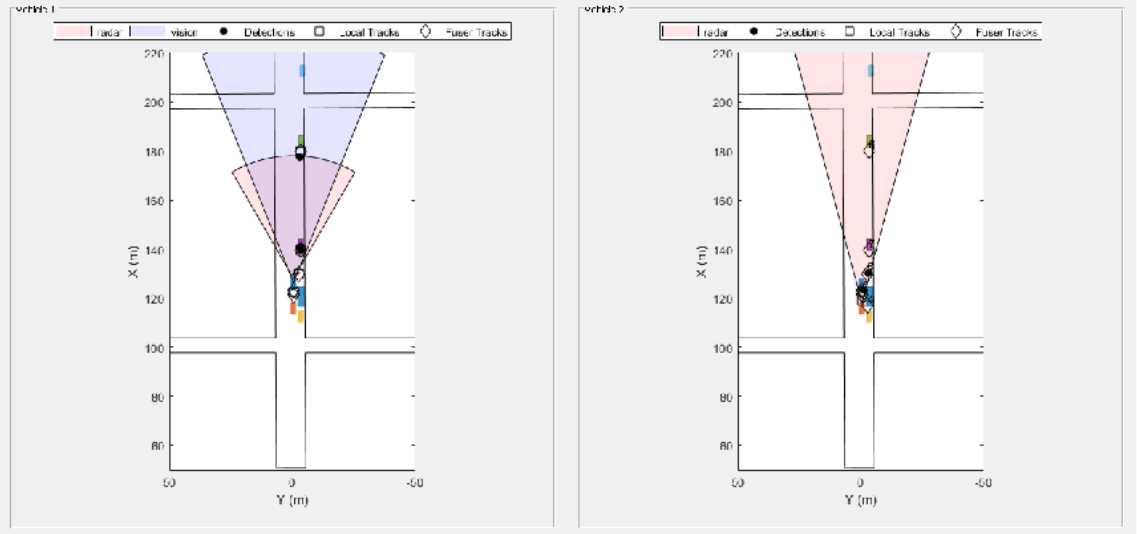
Tracking the Pedestrian at the Side of the Street

As the simulation continues, vehicle 2 is able to detect and track the vehicles parked at the side as well, and fuses them with the tracks coming from vehicle 1. Vehicle 2 is able to detect and track the pedestrian about 4 seconds into the simulation, and vehicle 2 fuses the track associated with the pedestrian around 4.4 seconds into the simulation (see snapshot 2). However, it takes vehicle 2 about two seconds before it can detect and track the pedestrian by its own sensors (see snapshot 3). These two seconds could make a huge impact on the safety of the pedestrian if that pedestrian started crossing the street.

```
showsnap(snaps, 2)
```



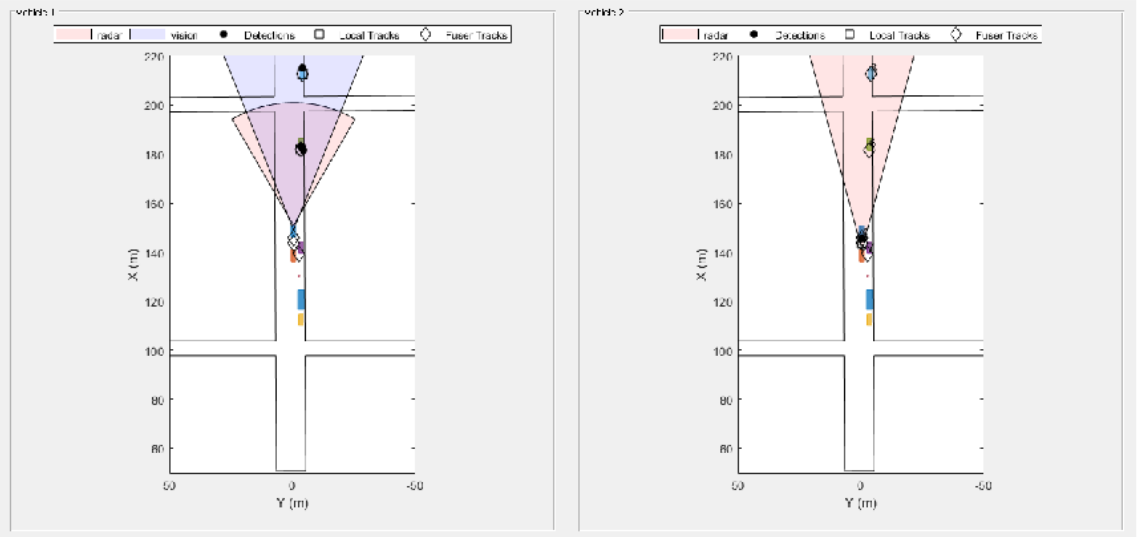

```
showsnap(snaps, 3)
```



Avoiding Rumor Propagation

Finally, note how as the vehicles pass the objects, and these objects go out of their field of view, the fused tracks associated with these objects are dropped by both trackers (see snapshot 4). Dropping the tracks demonstrates that the fused tracks broadcast back and forth between the two vehicles are not used to propagate rumors.

```
showsnap(snaps, 4)
```




```
% Restart the driving scenario to return the actors to their initial positions.
restart(scenario);

% Release all the sensor objects so they can be used again.
for sensorIndex = 1:numSensors
    release(sensors{sensorIndex});
end

% Return the random seed to its previous value
rng(s)
```

Summary

In this example, you saw how track-to-track fusion can enhance the situational awareness and increase the safety in automotive applications. You saw how to set up a `trackFuser` to perform track-to-track fusion and how to define sources as either internal or external by using the `fuserSourceConfiguration` object. By doing so, you avoid rumor propagation and keep only the fused tracks that are really observed by each vehicle to be maintained.

References

[1] Bharanidhar Duraisamy, Tilo Schwartz, and Christian Wohler, "Track level fusion algorithms for automotive safety applications", 2013 International Conference on Signal Processing, Image Processing & Pattern Recognition, IEEE, 2013.

[2] Federal Communications Commission, "Dedicated Short Range Communications Service", https://www.fcc.gov/wireless/bureau-divisions/mobility-division/dedicated-short-range-communications-dsrc-service.

Supporting Functions

createDrivingScenario

```
function [scenario, egoVehicle, secondVehicle] = createDrivingScenario
% createDrivingScenario Returns the drivingScenario defined in the Designer

% Construct a drivingScenario object.
scenario = drivingScenario('SampleTime', 0.05);

% Add all road segments
roadCenters = [50.8 0.5 0; 253.4 1.5 0];
roadWidth = 12;
road(scenario, roadCenters, roadWidth);

roadCenters = [100.7 -100.6 0; 100.7 103.7 0];
road(scenario, roadCenters);

roadCenters = [201.1 -99.2 0; 199.7 99.5 0];
road(scenario, roadCenters);

% Add the ego vehicle
egoVehicle = vehicle(scenario, 'ClassID', 1, 'Position', [65.1 -0.9 0]);
waypoints = [71 -0.5 0; 148.7 -0.5 0];
speed = 12;
trajectory(egoVehicle, waypoints, speed);

% Add the second vehicle
secondVehicle = vehicle(scenario, 'ClassID', 1, 'Position', [55.1 -0.9 0]);
waypoints = [61 -0.5 0; 138.7 -0.5 0];
speed = 12;
trajectory(secondVehicle, waypoints, speed);

% Add the parked cars
vehicle(scenario, 'ClassID', 1, 'Position', [111.0 -3.6 0]);
vehicle(scenario, 'ClassID', 1, 'Position', [140.6 -3.6 0]);
vehicle(scenario, 'ClassID', 1, 'Position', [182.6 -3.6 0]);
vehicle(scenario, 'ClassID', 1, 'Position', [211.3 -4.1 0]);

% Add pedestrian
actor(scenario, 'ClassID', 4, 'Length', 0.5, 'Width', 0.5, ...
    'Height', 1.7, 'Position', [130.3 -2.7 0], 'RCSPattern', [-8 -8;-8 -8]);

% Add parked truck
vehicle(scenario, 'ClassID', 2, 'Length', 8.2, 'Width', 2.5, ...
    'Height', 3.5, 'Position', [117.5 -3.5 0]);
end
```

createSensors

```
function [sensors, numSensors, attachedVehicle] = createSensors(scenario)
% createSensors Returns all sensor objects to generate detections
% Units used in createSensors and createDrivingScenario
% Distance/Position - meters
% Speed             - meters/second
% Angles            - degrees
% RCS Pattern       - dBsm
```

```
% Assign into each sensor the physical and radar profiles for all actors
profiles = actorProfiles(scenario);
sensors{1} = radarDetectionGenerator('SensorIndex', 1, ...
    'SensorLocation', [3.7 0], 'MaxRange', 50, 'FieldOfView', [60 5], ...
    'ActorProfiles', profiles, 'HasOcclusion', true, 'HasFalseAlarms', false);
sensors{2} = visionDetectionGenerator('SensorIndex', 2, ...
    'MaxRange', 100, 'SensorLocation', [1.9 0], 'DetectorOutput', 'Objects only', ...
    'ActorProfiles', profiles);
sensors{3} = radarDetectionGenerator('SensorIndex', 3, ...
    'SensorLocation', [3.7 0], 'MaxRange', 120, 'FieldOfView', [30 5], ...
    'ActorProfiles', profiles, 'HasOcclusion', true, 'HasFalseAlarms', false);
attachedVehicle = [1;1;2];
numSensors = numel(sensors);
end
```

scenarioToEgo

```
function trackInEgo = scenarioToEgo(trackInScenario)
% Performs coordinate transformation from scenario to ego coordinates
% trackInScenario has StateParameters defined to transform it from scenario
% coordinates to ego coordinates
% We assume a constant velocity model with state [x;vx;y;vy;z;vz]
egoPosInScenario = trackInScenario.StateParameters.Position;
egoVelInScenario = trackInScenario.StateParameters.Velocity;
stateInScenario = trackInScenario.State;
stateShift = [egoPosInScenario(1);egoVelInScenario(1);egoPosInScenario(2);egoVelInScenario(2);ego
stateInEgo = stateInScenario - stateShift;
trackInEgo = objectTrack('UpdateTime',trackInScenario.UpdateTime,'State',stateInEgo,'StateCovaria
end
```

egoToScenario

```
function trackInScenario = egoToScenario(trackInEgo)
% Performs coordinate transformation from ego to scenario coordinates
% trackInEgo has StateParameters defined to transform it from ego
% coordinates to scenario coordinates
% We assume a constant velocity model with state [x;vx;y;vy;z;vz]
egoPosInScenario = trackInEgo.StateParameters.Position;
egoVelInScenario = trackInEgo.StateParameters.Velocity;
stateInScenario = trackInEgo.State;
stateShift = [egoPosInScenario(1);egoVelInScenario(1);egoPosInScenario(2);egoVelInScenario(2);ego
stateInEgo = stateInScenario + stateShift;
trackInScenario = objectTrack('UpdateTime',trackInEgo.UpdateTime,'State',stateInEgo,'StateCovaria
end
```

detectAndTrack

```
function [tracks,wasTrackerUpdated] = detectAndTrack(agent,time,posSelector)
% Create detections from the vehicle
poses = targetPoses(agent.Actor);
[detections,isValid] = vehicleDetections(agent.Actor.Position,agent.Sensors,poses,time,agent.Det

% Update tracks for the vehicle
if isValid
    agent.Tracker.StateParameters = struct(...
        'Frame','Rectangular', ...
        'Position', agent.Actor.Position, ...
        'Velocity', agent.Actor.Velocity);
    tracks = agent.Tracker(detections,time);
```

```
    tracksInScenario = tracks;
    for i = 1:numel(tracks)
        tracksInScenario(i) = egoToScenario(tracks(i));
    end
    pos = getTrackPositions(tracksInScenario,posSelector);
    plotTrack(agent.TrkPlotter,pos)
    wasTrackerUpdated = true;
else
    tracks = objectTrack.empty(0,1);
    wasTrackerUpdated = false;
end
end

function [objectDetections,isValid] = vehicleDetections(position, sensors, poses, time, plotter)
% Provides the detections for each vehicle.

numSensors = numel(sensors);
objectDetections = {};
isValidTime      = false(1, numSensors);

% Generate detections for each sensor
for sensorIndex = 1:numSensors
    sensor = sensors{sensorIndex};
    [objectDets, ~, isValidTime(sensorIndex)] = sensor(poses, time);
    objectDets = cellfun(@(d) setAtt(d), objectDets, 'UniformOutput', false);

    if isa(sensors{sensorIndex},'radarDetectionGenerator')
        objectDets = helperClusterDetections(objectDets, 5);
    end
    numObjects = numel(objectDets);
    objectDetections = [objectDetections; objectDets(1:numObjects)]; %#ok<AGROW>
end
isValid = any(isValidTime);

% Plot detections
if numel(objectDetections)>0
    detPos = cellfun(@(d)d.Measurement(1:2), objectDetections, 'UniformOutput', false);
    detPos = cell2mat(detPos')' + position(1:2);
    plotDetection(plotter, detPos);
end
end

function d = setAtt(d)
% Set the attributes to be struct
d.ObjectAttributes = struct;
end

function detectionClusters = helperClusterDetections(detections, vehicleSize)
% helperClusterDetections  Helper to cluster detections in the example
N = numel(detections);
distances = zeros(N);
for i = 1:N
    for j = i+1:N
        if detections{i}.SensorIndex == detections{j}.SensorIndex
            distances(i,j) = norm(detections{i}.Measurement(1:2) - detections{j}.Measurement(1:2
        else
            distances(i,j) = inf;
        end
```

```
    end
end
leftToCheck = 1:N;
i = 0;
detectionClusters = cell(N,1);
while ~isempty(leftToCheck)
    % Remove the detections that are in the same cluster as the one under
    % consideration
    underConsideration = leftToCheck(1);
    clusterInds = (distances(underConsideration, leftToCheck) < vehicleSize);
    detInds = leftToCheck(clusterInds);
    clusterDets = [detections{detInds}];
    clusterMeas = [clusterDets.Measurement];
    meas = mean(clusterMeas, 2);
    i = i + 1;
    detectionClusters{i} = detections{detInds(1)};
    detectionClusters{i}.Measurement = meas;
    leftToCheck(clusterInds) = [];
end
detectionClusters(i+1:end) = [];

% Since the detections are now for clusters, modify the noise to represent
% that they are of the whole car
for i = 1:numel(detectionClusters)
    measNoise = eye(6);
    measNoise(1:2,1:2) = vehicleSize^2 * eye(2);
    measNoise(4:5,4:5) = eye(2) * vehicleSize^2;
    detectionClusters{i}.MeasurementNoise = measNoise;
end
end
```

See Also

Apps
Driving Scenario Designer

Objects
`fuserSourceConfiguration` | `objectTrack` | `trackFuser`

More About

- “Introduction to Track-To-Track Fusion” (Sensor Fusion and Tracking Toolbox)
- “Track-to-Track Fusion for Automotive Safety Applications in Simulink” on page 7-267

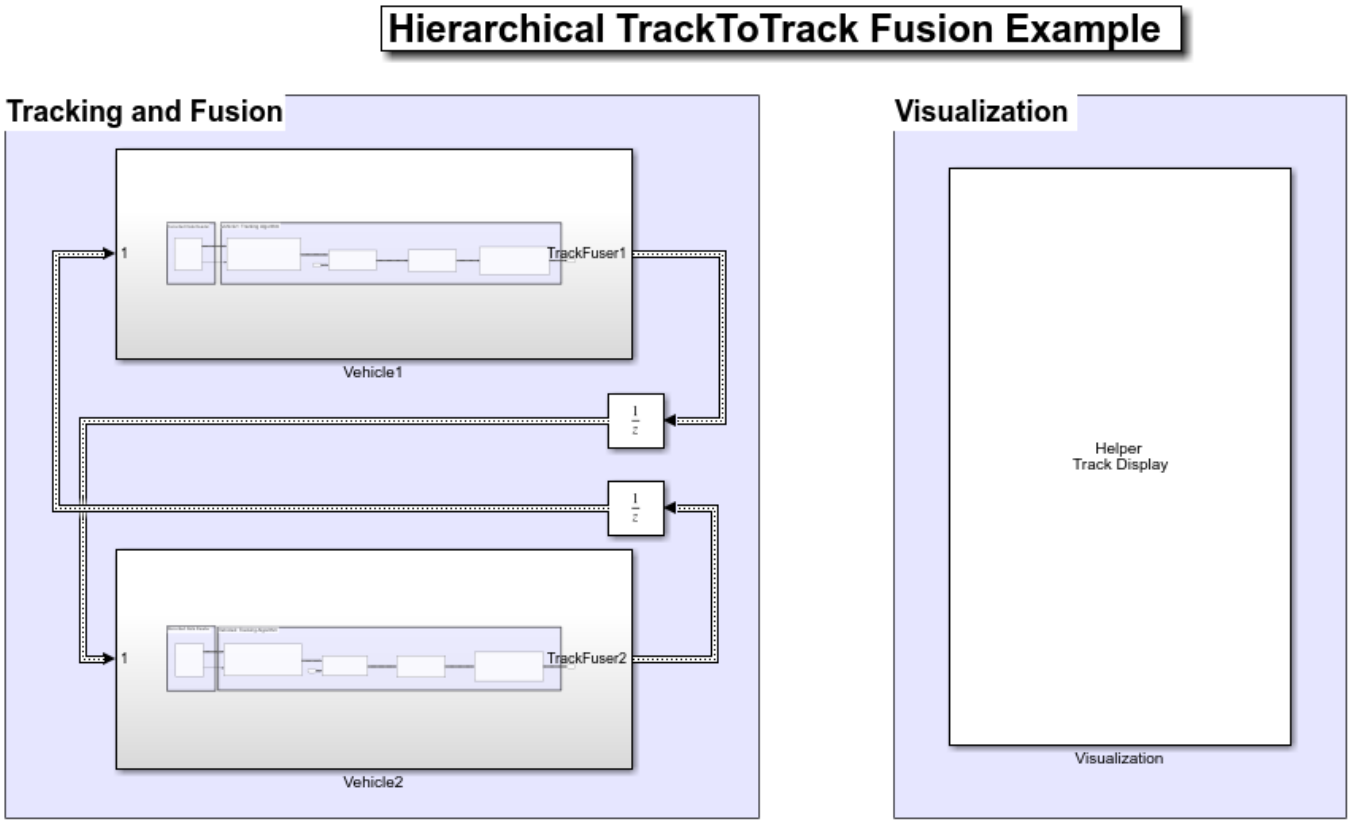
Track-to-Track Fusion for Automotive Safety Applications in Simulink

This example shows how to perform track-to-track fusion in Simulink® with Sensor Fusion and Tracking Toolbox™. In the context of autonomous driving, the example illustrates how to build a decentralized tracking architecture using a track fuser block. In the example, each vehicle perform tracking independently as well as fuse tracking information received from other vehicles. This example closely follows the “Track-to-Track Fusion for Automotive Safety Applications” (Sensor Fusion and Tracking Toolbox) MATLAB® example.

Introduction

Automotive safety applications largely rely on the situational awareness of the vehicle. A better situational awareness provides the basis to a successful decision-making for different situations. To achieve this, vehicles can benefit from intervehicle data fusion. This example illustrates the workflow in Simulink for fusing data from two vehicles to enhance situational awareness of the vehicle.

Setup and Overview of the Model

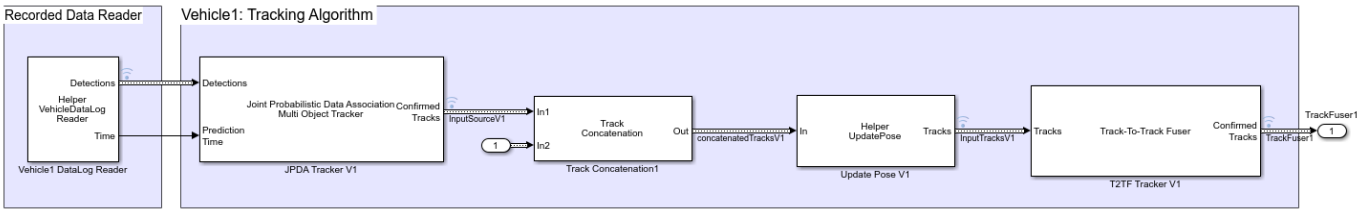


Prior to running this example, the `drivingScenario` object was used to create the same scenario defined in “Track-to-Track Fusion for Automotive Safety Applications” (Sensor Fusion and Tracking Toolbox). The detections and time data of objects detected from the sensors of `Vehicle1` and `Vehicle2` in the scenario were then saved to the data files `v1Data.mat` and `v2Data.mat`, respectively. Also, the pose information of vehicles were saved in files `v1Pose.mat` and `v2Pose.mat`.

Tracking and Fusion

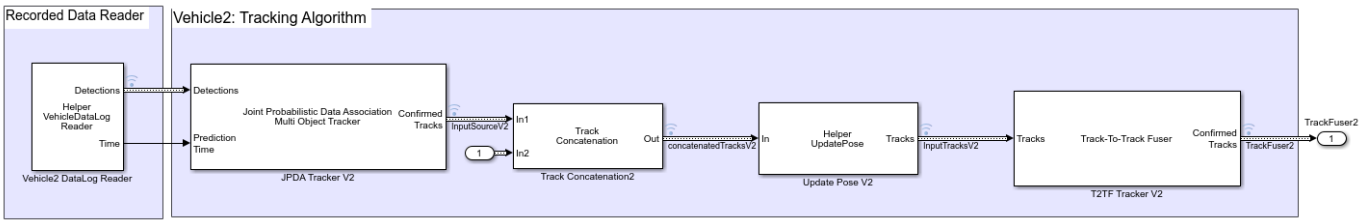
In Tracking and Fusion section of the model there are two subsystems which implements the target tracking and fusion capabilities of `Vehicle1` and `Vehicle2` in this scenario.

Vehicle1



This subsystem includes `Vehicle1 DataLog Reader` block that reads the prerecorded detection and time data form `v1Data.mat` file for `Vehicle1`. This data is then passed to the `JPDA Tracker V1` block which processes the detections to generate a list of tracks. The tracks are then passed into a `Track Concatenation1` block, which concatenates these input tracks. The first input to the `Track Concatenation1` block is the local tracks from the JPDA tracker and the second input is the tracks received from the other vehicle's track fuser. To transform local tracks to central tracks, the track fuser needs the parameter information about the local tracks. However, this information is not available from the direct outputs of the JPDA tracker. Therefore, a helper Update Pose block is used to supply these information by reading the data from the v1Pose.mat file. The updated tracks are then broadcasted to `T2TF Tracker V1` block as an input. Finally, the `trackFuser` (Sensor Fusion and Tracking Toolbox) `T2TF Tracker V1` block fuses the local vehicle tracks with the tracks received from the other vehicle's track fuser. After each update, the track fuser on each vehicle broadcasts its fused tracks to be fed into the update of the other vehicle's track fuser in the next time stamp.

Vehicle2



`Vehicle2` subsystem follows similar setup as `Vehicle1` subsystem as described above.

Visualization

The Visualization block is implemented using the MATLAB System block and is defined using `HelperTrackDisplay` block. The block uses `RunTimeObject` parameter of the blocks to display their outputs. See “Access Block Data During Simulation” (Simulink) for further information on how to access block outputs during simulation.

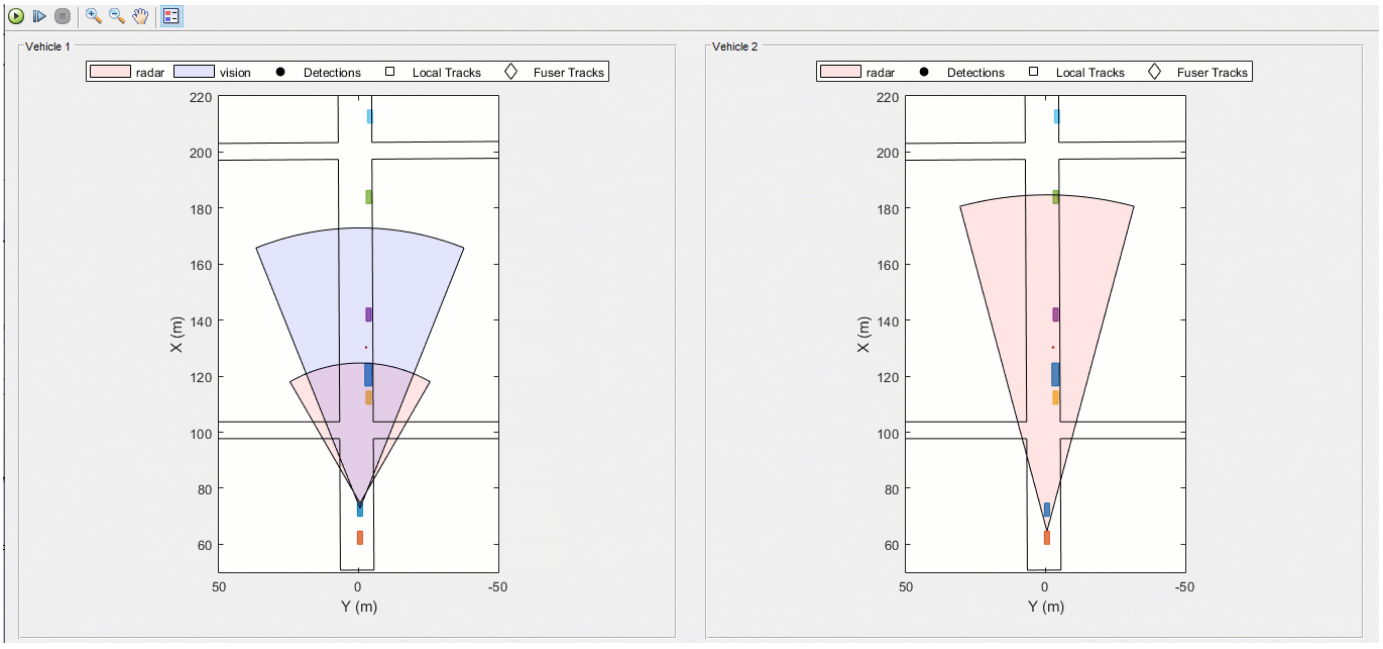
Results

After running the model, you visualize the results as on the figure. The animation below shows the results for this simulation.

The visualization includes two panels. The left panel shows the detections, local tracks, and fused tracks that `Vehicle1` generated during the simulation and represents the situational awareness of the `Vehicle1`. The right panel shows the situational awareness of `Vehicle2`.

The recorded detections are represented by black circles. The local and fused tracks from `Vehicle1` and `Vehicle2` are represented by square and diamond respectively. Notice that during the start of simulation, `Vehicle1` detects vehicles parked on the right side of the street, and tracks associated with the parked vehicles are confirmed. Currently `Vehicle2` only detects `Vehicle1` which is immediately in front of it. As the simulation continues the confirmed tracks from `Vehicle1` are broadcasts to the fuser on `Vehicle2`. After fusing the tracks, `vehicle2` becomes aware of the objects prior to detecting these objects on its own. Similarly, `Vehicle2` tracks are broadcasts to `Vehicle1`. `Vehicle1` fuses these tracks and becomes aware of the objects prior to detecting them on its own.

In particular, you observe that the pedestrian standing between the blue and purple car on the right side of the street is detected and tracked by `Vehicle1`. `Vehicle2` first becomes aware of the pedestrian by fusing the track from `Vehicle1` at around 2.5 seconds. It takes `Vehicle2` roughly 2 seconds before it starts detecting the pedestrian using its own sensor. The ability to track a pedestrian based on inputs from `Vehicle1` allows `Vehicle2` to extend its situational awareness and to mitigate the risk of accident.



Summary

This example showed how to perform track-to-track fusion in Simulink. You learned how to perform tracking using a decentralized tracking architecture, where each vehicle is responsible for maintaining its own local tracks, fuse tracks from other vehicles, and communicate the tracks to the other vehicle. You also used a JPDA tracker block to generate the local tracks.

See Also

Blocks

Joint Probabilistic Data Association Multi Object Tracker

Objects
`trackFuser`

More About

- “Introduction to Track-To-Track Fusion” (Sensor Fusion and Tracking Toolbox)
- “Track-to-Track Fusion for Automotive Safety Applications” on page 7-254

Visual-Inertial Odometry Using Synthetic Data

This example shows how to estimate the pose (position and orientation) of a ground vehicle using an inertial measurement unit (IMU) and a monocular camera. In this example, you:

1. Create a driving scenario containing the ground truth trajectory of the vehicle.
2. Use an IMU and visual odometry model to generate measurements.
3. Fuse these measurements to estimate the pose of the vehicle and then display the results.

Visual-inertial odometry estimates pose by fusing the visual odometry pose estimate from the monocular camera and the pose estimate from the IMU. The IMU returns an accurate pose estimate for small time intervals, but suffers from large drift due to integrating the inertial sensor measurements. The monocular camera returns an accurate pose estimate over a larger time interval, but suffers from a scale ambiguity. Given these complementary strengths and weaknesses, the fusion of these sensors using visual-inertial odometry is a suitable choice. This method can be used in scenarios where GPS readings are unavailable, such as in an urban canyon.

Create a Driving Scenario with Trajectory

Create a `drivingScenario` object that contains:

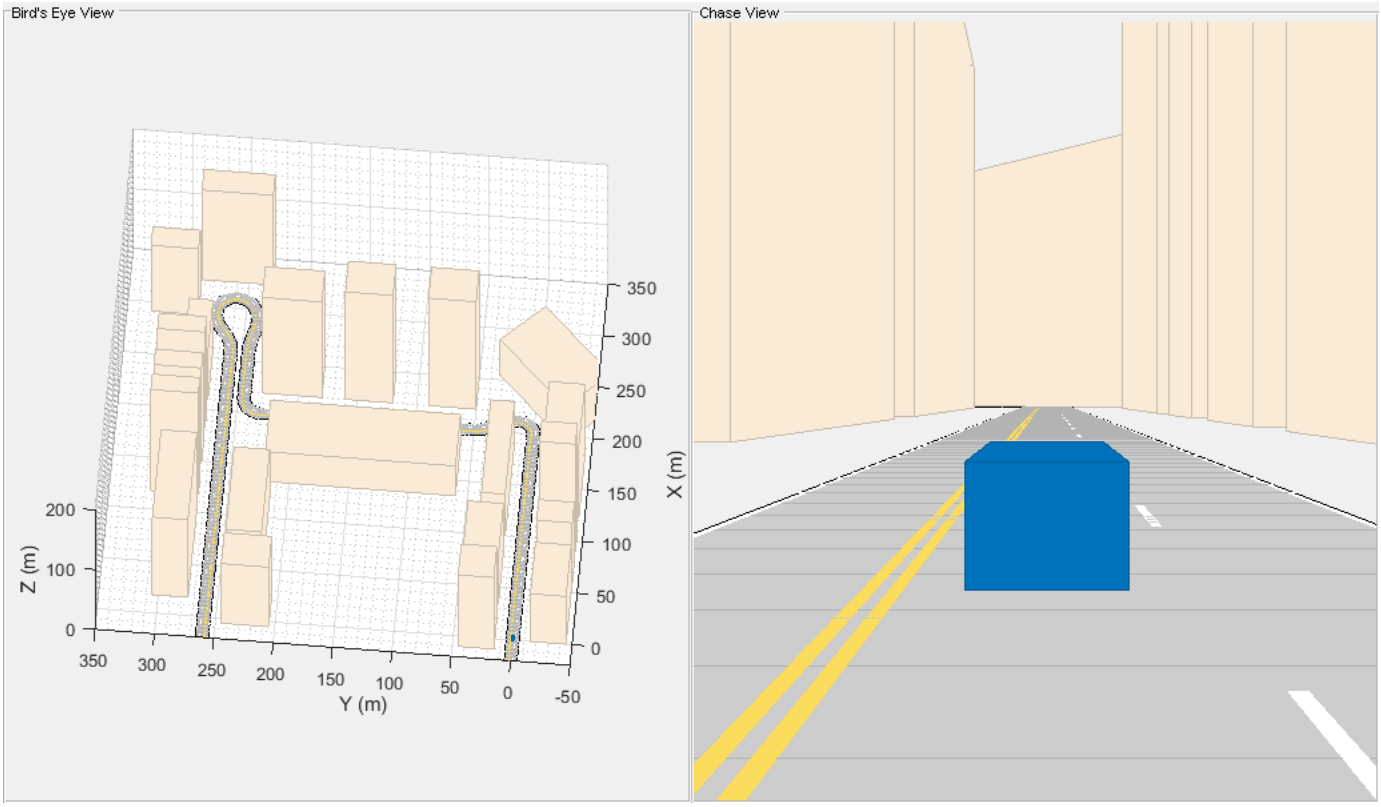
- The road the vehicle travels on
- The buildings surrounding either side of the road
- The ground truth pose of the vehicle
- The estimated pose of the vehicle

The ground truth pose of the vehicle is shown as a solid blue cuboid. The estimated pose is shown as a transparent blue cuboid. Note that the estimated pose does not appear in the initial visualization because the ground truth and estimated poses overlap.

Generate the baseline trajectory for the ground vehicle using the `waypointTrajectory` (Sensor Fusion and Tracking Toolbox) System object™. Note that the `waypointTrajectory` is used in place of `drivingScenario/trajectory` since the acceleration of the vehicle is needed. The trajectory is generated at a specified sampling rate using a set of waypoints, times of arrival, and velocities.

```
% Create the driving scenario with both the ground truth and estimated
% vehicle poses.
scene = drivingScenario;
groundTruthVehicle = vehicle(scene, 'PlotColor', [0 0.4470 0.7410]);
estVehicle = vehicle(scene, 'PlotColor', [0 0.4470 0.7410]);

% Generate the baseline trajectory.
sampleRate = 100;
wayPoints = [  0   0 0;
              200   0 0;
              200  50 0;
              200 230 0;
              215 245 0;
              260 245 0;
              290 240 0;
              310 258 0;
              290 275 0;
              260 260 0;
              -20 260 0];
```

```
t = [0 20 25 44 46 50 54 56 59 63 90].';
speed = 10;
velocities = [ speed     0 0;
               speed     0 0;
                   0 speed 0;
                   0 speed 0;
               speed     0 0;
               speed     0 0;
               speed     0 0;
                   0 speed 0;
              -speed     0 0;
              -speed     0 0;
              -speed     0 0];

traj = waypointTrajectory(wayPoints, 'TimeOfArrival', t, ...
    'Velocities', velocities, 'SampleRate', sampleRate);

% Add a road and buildings to scene and visualize.
helperPopulateScene(scene, groundTruthVehicle);
```



Create a Fusion Filter

Create the filter to fuse IMU and visual odometry measurements. This example uses a loosely coupled method to fuse the measurements. While the results are not as accurate as a tightly coupled method, the amount of processing required is significantly less and the results are adequate. The fusion filter uses an error-state Kalman filter to track orientation (as a quaternion), position, velocity, and sensor biases.

The `insfilterErrorState` object has the following functions to process sensor data: `predict` and `fusemvo`.

The `predict` function takes the accelerometer and gyroscope measurements from the IMU as inputs. Call the `predict` function each time the accelerometer and gyroscope are sampled. This function predicts the state forward by one time step based on the accelerometer and gyroscope measurements, and updates the error state covariance of the filter.

The `fusemvo` function takes the visual odometry pose estimates as input. This function updates the error states based on the visual odometry pose estimates by computing a Kalman gain that weighs the various inputs according to their uncertainty. As with the `predict` function, this function also updates the error state covariance, this time taking the Kalman gain into account. The state is then updated using the new error state and the error state is reset.

```
filt = insfilterErrorState('IMUSampleRate', sampleRate, ...
    'ReferenceFrame', 'ENU')
% Set the initial state and error state covariance.
helperInitialize(filt, traj);
```

```
filt =

  insfilterErrorState with properties:

        IMUSampleRate: 100              Hz
    ReferenceLocation: [0 0 0]          [deg deg m]
                State: [17x1 double]
      StateCovariance: [16x16 double]

   Process Noise Variances
            GyroscopeNoise: [1e-06 1e-06 1e-06]       (rad/s)²
        AccelerometerNoise: [0.0001 0.0001 0.0001]    (m/s²)²
        GyroscopeBiasNoise: [1e-09 1e-09 1e-09]       (rad/s)²
    AccelerometerBiasNoise: [0.0001 0.0001 0.0001]    (m/s²)²
```

Specify the Visual Odometry Model

Define the visual odometry model parameters. These parameters model a feature matching and tracking-based visual odometry system using a monocular camera. The `scale` parameter accounts for the unknown scale of subsequent vision frames of the monocular camera. The other parameters model the drift in the visual odometry reading as a combination of white noise and a first-order Gauss-Markov process.

```
% The flag useVO determines if visual odometry is used:
% useVO = false; % Only IMU is used.
useVO = true; % Both IMU and visual odometry are used.

paramsVO.scale = 2;
paramsVO.sigmaN = 0.139;
paramsVO.tau = 232;
paramsVO.sigmaB = sqrt(1.34);
paramsVO.driftBias = [0 0 0];
```

Specify the IMU Sensor

Define an IMU sensor model containing an accelerometer and gyroscope using the `imuSensor` System object. The sensor model contains properties to model both deterministic and stochastic noise sources. The property values set here are typical for low-cost MEMS sensors.

```
% Set the RNG seed to default to obtain the same results for subsequent
% runs.
rng('default')

imu = imuSensor('SampleRate', sampleRate, 'ReferenceFrame', 'ENU');

% Accelerometer
imu.Accelerometer.MeasurementRange =  19.6; % m/s^2
imu.Accelerometer.Resolution = 0.0024; % m/s^2/LSB
imu.Accelerometer.NoiseDensity = 0.01; % (m/s^2)/sqrt(Hz)

% Gyroscope
imu.Gyroscope.MeasurementRange = deg2rad(250); % rad/s
imu.Gyroscope.Resolution = deg2rad(0.0625); % rad/s/LSB
imu.Gyroscope.NoiseDensity = deg2rad(0.0573); % (rad/s)/sqrt(Hz)
imu.Gyroscope.ConstantBias = deg2rad(2); % rad/s
```

Set Up the Simulation

Specify the amount of time to run the simulation and initialize variables that are logged during the simulation loop.

```
% Run the simulation for 60 seconds.
numSecondsToSimulate = 60;
numIMUSamples = numSecondsToSimulate * sampleRate;

% Define the visual odometry sampling rate.
imuSamplesPerCamera = 4;
numCameraSamples = ceil(numIMUSamples / imuSamplesPerCamera);

% Preallocate data arrays for plotting results.
[pos, orient, vel, acc, angvel, ...
    posVO, orientVO, ...
    posEst, orientEst, velEst] ...
    = helperPreallocateData(numIMUSamples, numCameraSamples);

% Set measurement noise parameters for the visual odometry fusion.
RposVO = 0.1;
RorientVO = 0.1;
```

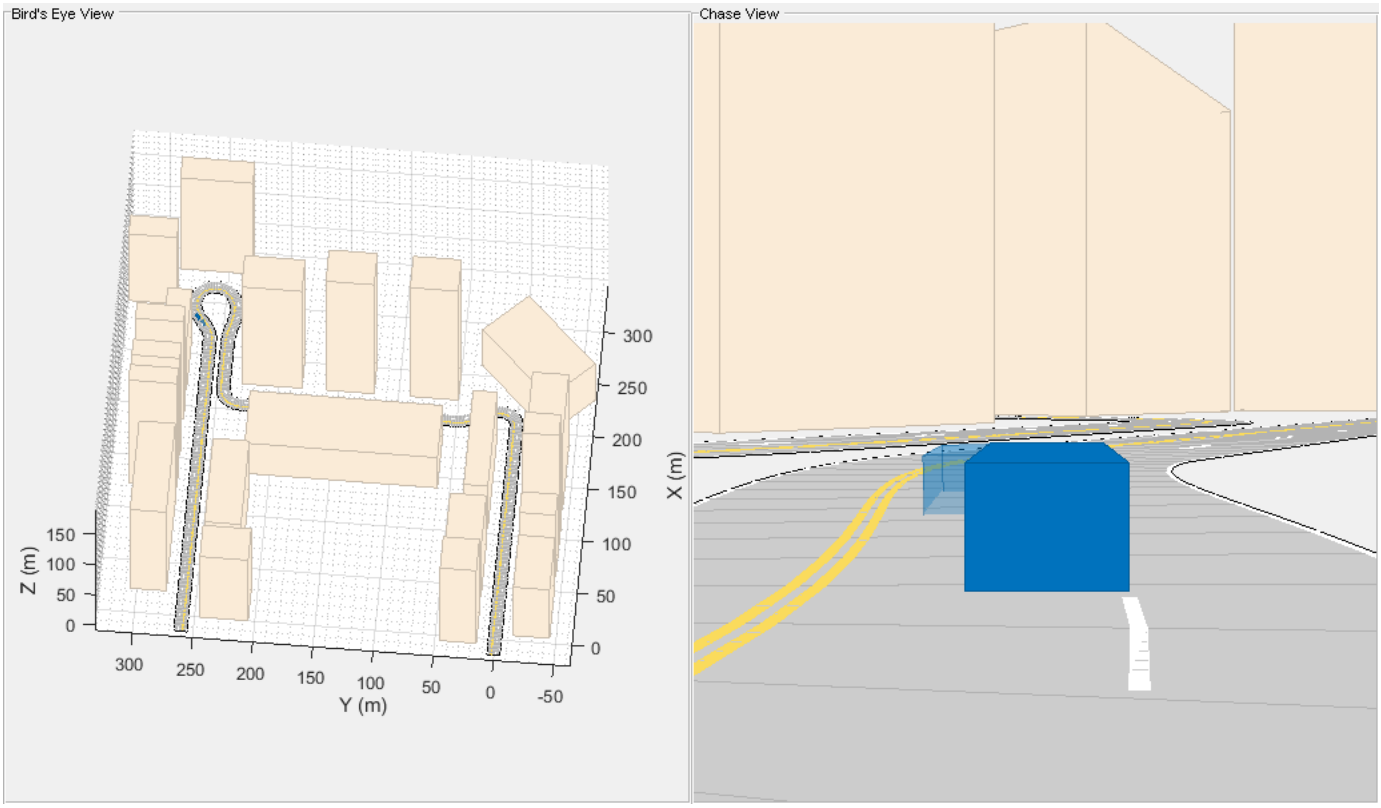
Run the Simulation Loop

Run the simulation at the IMU sampling rate. Each IMU sample is used to predict the filter's state forward by one time step. Once a new visual odometry reading is available, it is used to correct the current filter state.

There is some drift in the filter estimates that can be further corrected with an additional sensor such as a GPS or an additional constraint such as a road boundary map.

```
cameraIdx = 1;
for i = 1:numIMUSamples
    % Generate ground truth trajectory values.
```

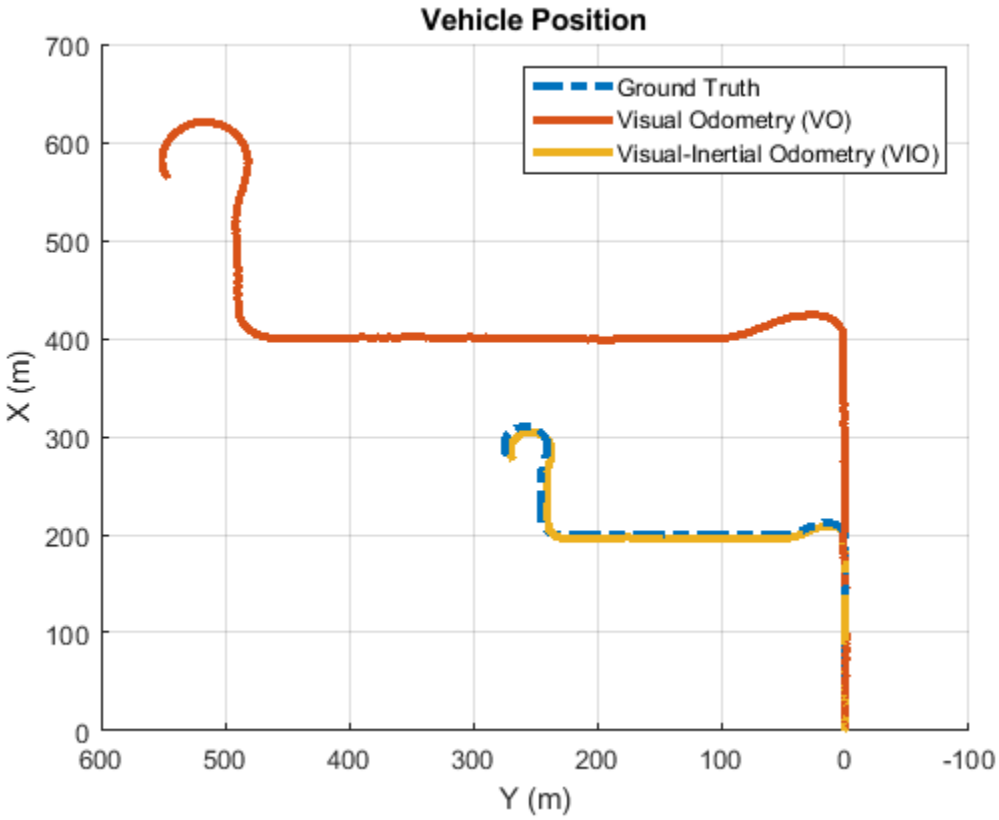
```
        [pos(i,:), orient(i,:), vel(i,:), acc(i,:), angvel(i,:)] = traj();

        % Generate accelerometer and gyroscope measurements from the ground truth
        % trajectory values.
        [accelMeas, gyroMeas] = imu(acc(i,:), angvel(i,:), orient(i));

        % Predict the filter state forward one time step based on the
        % accelerometer and gyroscope measurements.
        predict(filt, accelMeas, gyroMeas);

        if (1 == mod(i, imuSamplesPerCamera)) && useVO
            % Generate a visual odometry pose estimate from the ground truth
            % values and the visual odometry model.
            [posVO(cameraIdx,:), orientVO(cameraIdx,:), paramsVO] = ...
                helperVisualOdometryModel(pos(i,:), orient(i,:), paramsVO);

            % Correct filter state based on visual odometry data.
            fusemvo(filt, posVO(cameraIdx,:), RposVO, ...
                orientVO(cameraIdx), RorientVO);

            cameraIdx = cameraIdx + 1;
        end

        [posEst(i,:), orientEst(i,:), velEst(i,:)] = pose(filt);

        % Update estimated vehicle pose.
        helperUpdatePose(estVehicle, posEst(i,:), velEst(i,:), orientEst(i));

        % Update ground truth vehicle pose.
        helperUpdatePose(groundTruthVehicle, pos(i,:), vel(i,:), orient(i));

        % Update driving scenario visualization.
        updatePlots(scene);
        drawnow limitrate;
    end
```



Plot the Results

Plot the ground truth vehicle trajectory, the visual odometry estimate, and the fusion filter estimate.

```
figure
if useVO
    plot3(pos(:,1), pos(:,2), pos(:,3), '-.', ...
        posVO(:,1), posVO(:,2), posVO(:,3), ...
        posEst(:,1), posEst(:,2), posEst(:,3), ...
        'LineWidth', 3)
    legend('Ground Truth', 'Visual Odometry (VO)', ...
        'Visual-Inertial Odometry (VIO)', 'Location', 'northeast')
else
    plot3(pos(:,1), pos(:,2), pos(:,3), '-.', ...
        posEst(:,1), posEst(:,2), posEst(:,3), ...
        'LineWidth', 3)
    legend('Ground Truth', 'IMU Pose Estimate')
end
view(-90, 90)
title('Vehicle Position')
xlabel('X (m)')
ylabel('Y (m)')
grid on
```



The plot shows that the visual odometry estimate is relatively accurate in estimating the shape of the trajectory. The fusion of the IMU and visual odometry measurements removes the scale factor uncertainty from the visual odometry measurements and the drift from the IMU measurements.

Supporting Functions

`helperVisualOdometryModel`

Compute visual odometry measurement from ground truth input and parameters struct. To model the uncertainty in the scaling between subsequent frames of the monocular camera, a constant scaling factor combined with a random drift is applied to the ground truth position.

```
function [posVO, orientVO, paramsVO] ...
    = helperVisualOdometryModel(pos, orient, paramsVO)

% Extract model parameters.
scaleVO = paramsVO.scale;
sigmaN = paramsVO.sigmaN;
tau = paramsVO.tau;
sigmaB = paramsVO.sigmaB;
sigmaA = sqrt((2/tau) + 1/(tau*tau))*sigmaB;
b = paramsVO.driftBias;

% Calculate drift.
b = (1 - 1/tau).*b + randn(1,3)*sigmaA;
drift = randn(1,3)*sigmaN + b;
paramsVO.driftBias = b;
```

```
% Calculate visual odometry measurements.
posVO = scaleVO*pos + drift;
orientVO = orient;
end
```

helperInitialize

Set the initial state and covariance values for the fusion filter.

```
function helperInitialize(filt, traj)

% Retrieve the initial position, orientation, and velocity from the
% trajectory object and reset the internal states.
[pos, orient, vel] = traj();
reset(traj);

% Set the initial state values.
filt.State(1:4) = compact(orient(1)).';
filt.State(5:7) = pos(1,:).';
filt.State(8:10) = vel(1,:).';

% Set the gyroscope bias and visual odometry scale factor covariance to
% large values corresponding to low confidence.
filt.StateCovariance(10:12,10:12) = 1e6;
filt.StateCovariance(end) = 2e2;
end
```

helperPreallocateData

Preallocate data to log simulation results.

```
function [pos, orient, vel, acc, angvel, ...
    posVO, orientVO, ...
    posEst, orientEst, velEst] ...
    = helperPreallocateData(numIMUSamples, numCameraSamples)

% Specify ground truth.
pos = zeros(numIMUSamples, 3);
orient = quaternion.zeros(numIMUSamples, 1);
vel = zeros(numIMUSamples, 3);
acc = zeros(numIMUSamples, 3);
angvel = zeros(numIMUSamples, 3);

% Visual odometry output.
posVO = zeros(numCameraSamples, 3);
orientVO = quaternion.zeros(numCameraSamples, 1);

% Filter output.
posEst = zeros(numIMUSamples, 3);
orientEst = quaternion.zeros(numIMUSamples, 1);
velEst = zeros(numIMUSamples, 3);
end
```

helperUpdatePose

Update the pose of the vehicle.

```
function helperUpdatePose(veh, pos, vel, orient)

veh.Position = pos;
veh.Velocity = vel;
rpy = eulerd(orient, 'ZYX', 'frame');
veh.Yaw = rpy(1);
veh.Pitch = rpy(2);
veh.Roll = rpy(3);
end
```

References

- Sola, J. "Quaternion Kinematics for the Error-State Kalman Filter." ArXiv e-prints, arXiv:1711.02508v1 [cs.RO] 3 Nov 2017.
- R. Jiang, R., R. Klette, and S. Wang. "Modeling of Unbounded Long-Range Drift in Visual Odometry." 2010 Fourth Pacific-Rim Symposium on Image and Video Technology. Nov. 2010, pp. 121-126.

See Also

`drivingScenario` | `imuSensor` | `waypointTrajectory`

Lane Following Control with Sensor Fusion and Lane Detection

This example shows how to simulate and generate code for an automotive lane-following controller.

In this example, you:

1. Review a control algorithm that combines sensor fusion, lane detection, and a lane following controller from the Model Predictive Control Toolbox™ software.
2. Test the control system in a closed-loop Simulink® model using synthetic data generated by Automated Driving Toolbox™ software.
3. Configure the code generation settings for software-in-the-loop simulation and automatically generate code for the control algorithm.

Introduction

A lane following system is a control system that keeps the vehicle traveling within a marked lane of a highway, while maintaining a user-set velocity or safe distance from the preceding vehicle. A lane following system includes combined longitudinal and lateral control of the ego vehicle:

- Longitudinal control - Maintain a driver-set velocity and keep a safe distance from the preceding car in the lane by adjusting the acceleration of the ego vehicle.
- Lateral control - Keep the ego vehicle travelling along the centerline of its lane by adjusting the steering of the ego vehicle

The combined lane following control system achieves the individual goals for longitudinal and lateral control. Further, the lane following control system can adjust the priority of the two goals when they cannot be met simultaneously.

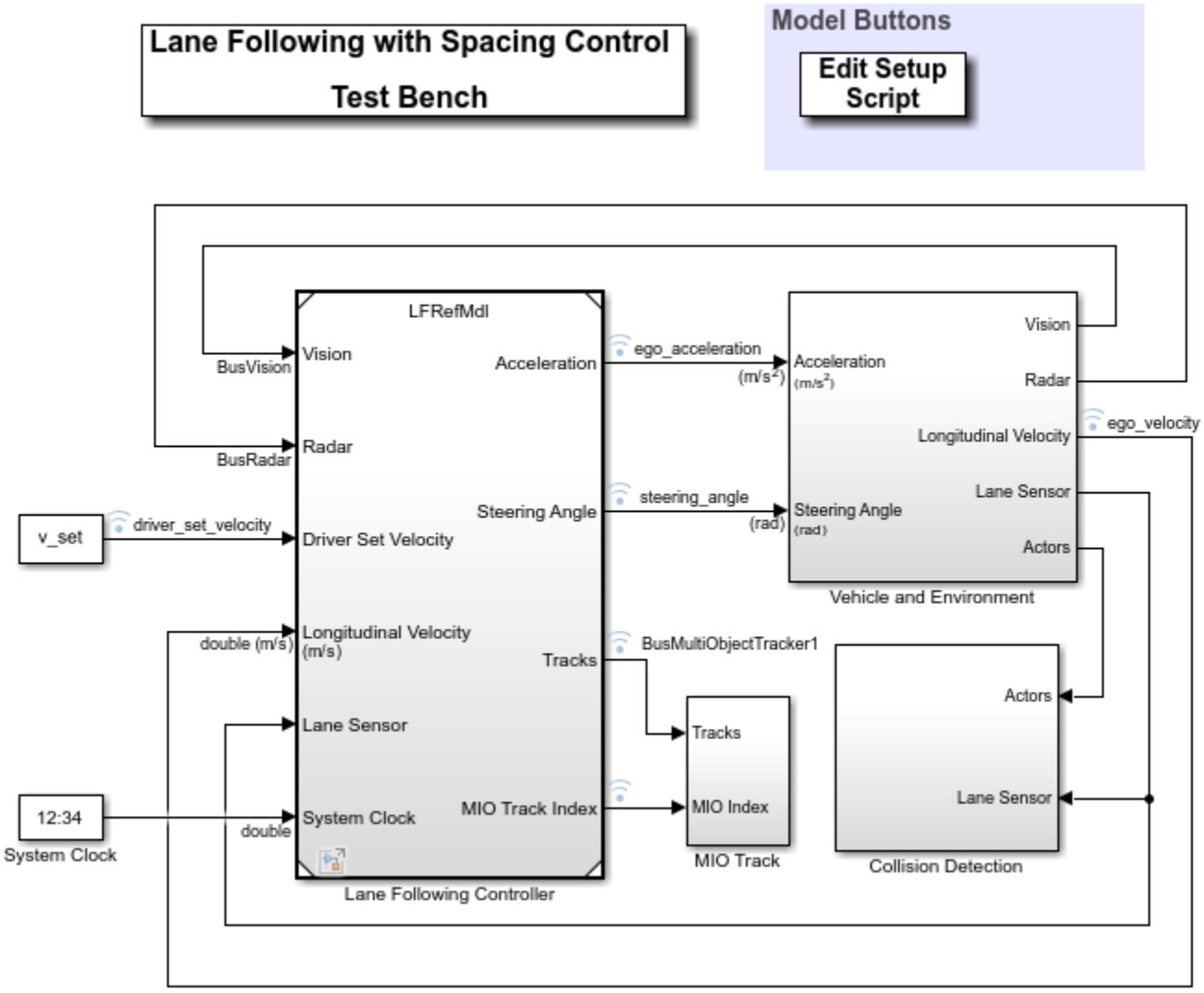
For an example of longitudinal control using adaptive cruise control (ACC) with sensor fusion, see “Adaptive Cruise Control with Sensor Fusion” (Model Predictive Control Toolbox). For an example of lateral control using a lane keeping assist (LKA) system with lane detection, see “Lane Keeping Assist with Lane Detection” (Model Predictive Control Toolbox). The ACC example assumes ideal lane detection, and the LKA example does not consider surrounding vehicles.

In this example, both lane detection and surrounding cars are considered. The lane following system synthesizes data from vision and radar detections, estimates the lane center and lead car distance, and calculates the longitudinal acceleration and steering angle of the ego vehicle.

Open Test Bench Model

Open the Simulink test bench model.

```
open_system('LaneFollowingTestBenchExample')
```



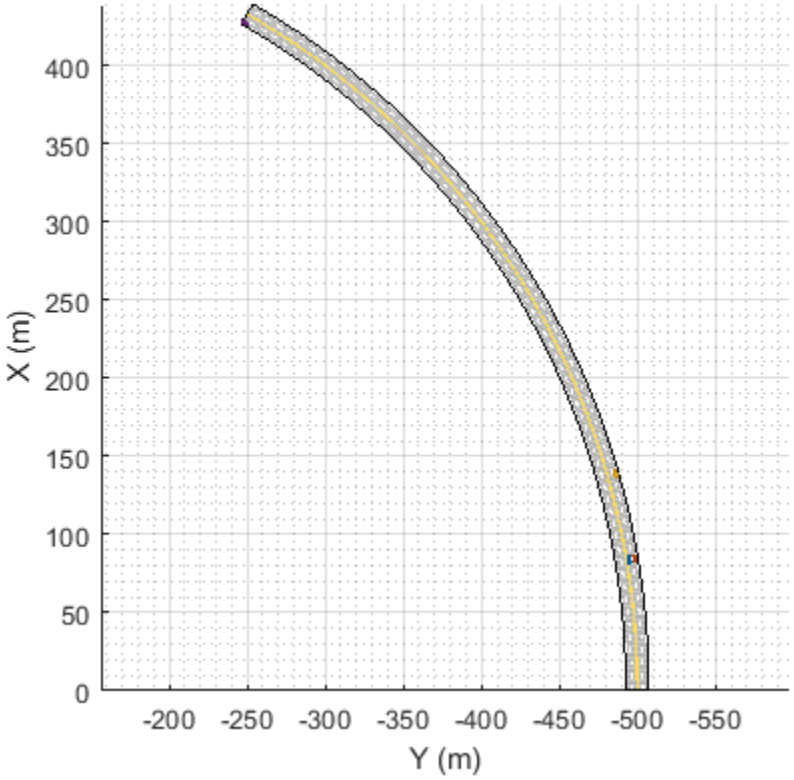
The model contains four main components:

1. Lane Following Controller - Controls both the longitudinal acceleration and front steering angle of the ego vehicle
2. Vehicle and Environment - Models the motion of the ego vehicle and models the environment
3. Collision Detection - Stops the simulation when a collision of the ego vehicle and lead vehicle is detected
4. MIO Track - Enables MIO track for display in the Bird's-Eye Scope.

Opening this model also runs the `helperLFSetUp` script, which initializes the data used by the model by loading constants needed by the Simulink model, such as the vehicle model parameters, controller design parameters, road scenario, and surrounding cars.

Plot the road and the path that the ego vehicle.

```
plot(scenario)
```

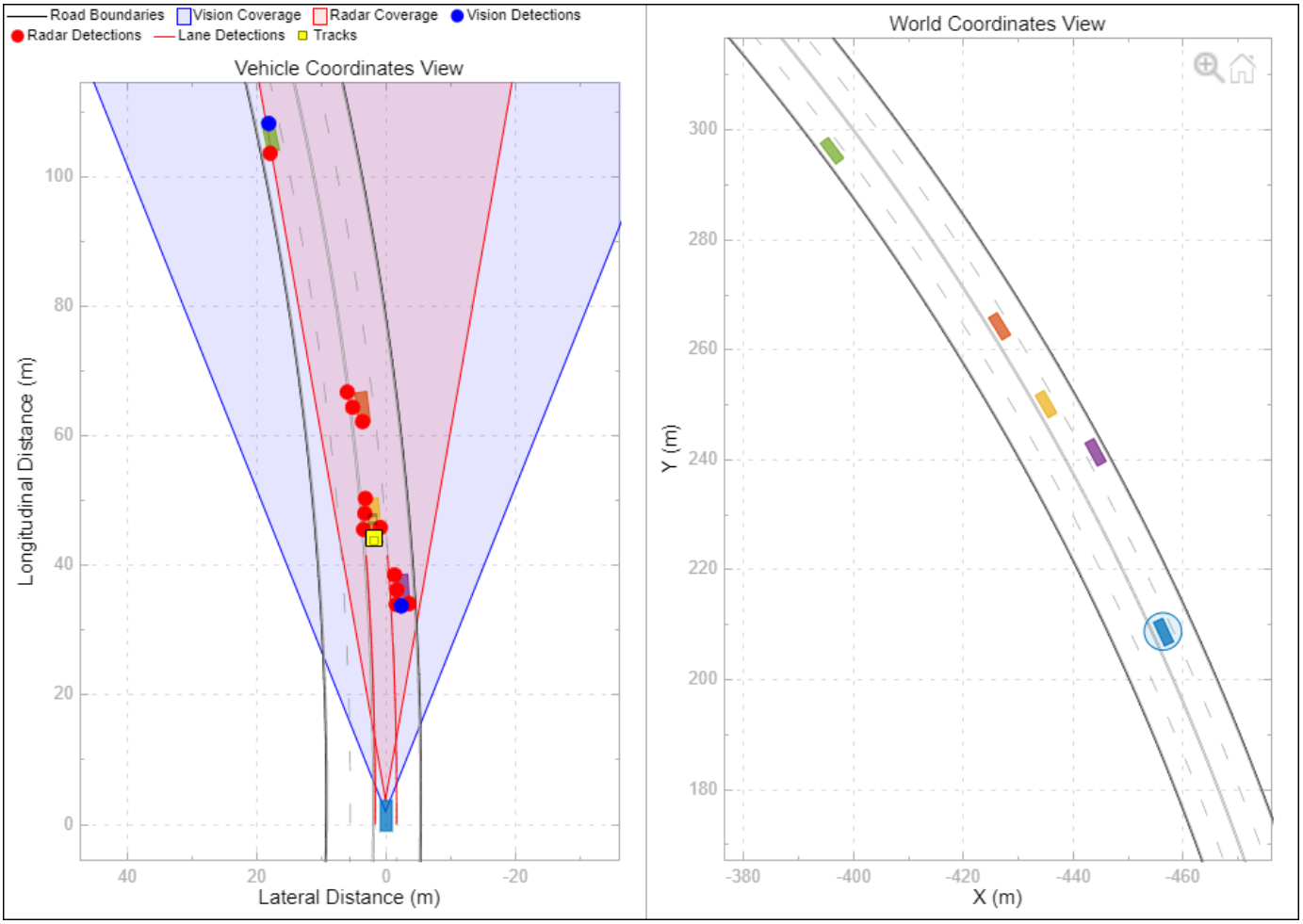


To plot the results of the simulation and depict the ego vehicle surroundings and tracked objects, use the Bird's-Eye Scope. The Bird's-Eye Scope is a model-level visualization tool that you can open from the Simulink toolstrip. On the **Simulation** tab, under **Review Results**, click **Bird's-Eye Scope**. After opening the scope, set up the signals by clicking **Find Signals**.

To get a mid-simulation view, simulate the model for `10` seconds.

```
sim('LaneFollowingTestBenchExample','StopTime','10')
```

After simulating the model for 10 seconds, open the Bird's-Eye Scope. In the scope toolstrip, to display the World Coordinates View of the scenario, click **World Coordinates**. In this view, the ego vehicle is circled. To display the legend for the Vehicle Coordinates View, click **Legend**.



The Bird's-Eye Scope shows the results of the sensor fusion. It shows how the radar and vision sensors detect the vehicles within their coverage areas. It also shows the tracks maintained by the Multi-Object Tracker block. The yellow track shows the most important object (MIO), which is the closest track in front of the ego vehicle in its lane. The ideal lane markings are also shown along with the synthetically detected left and right lane boundaries (shown in red).

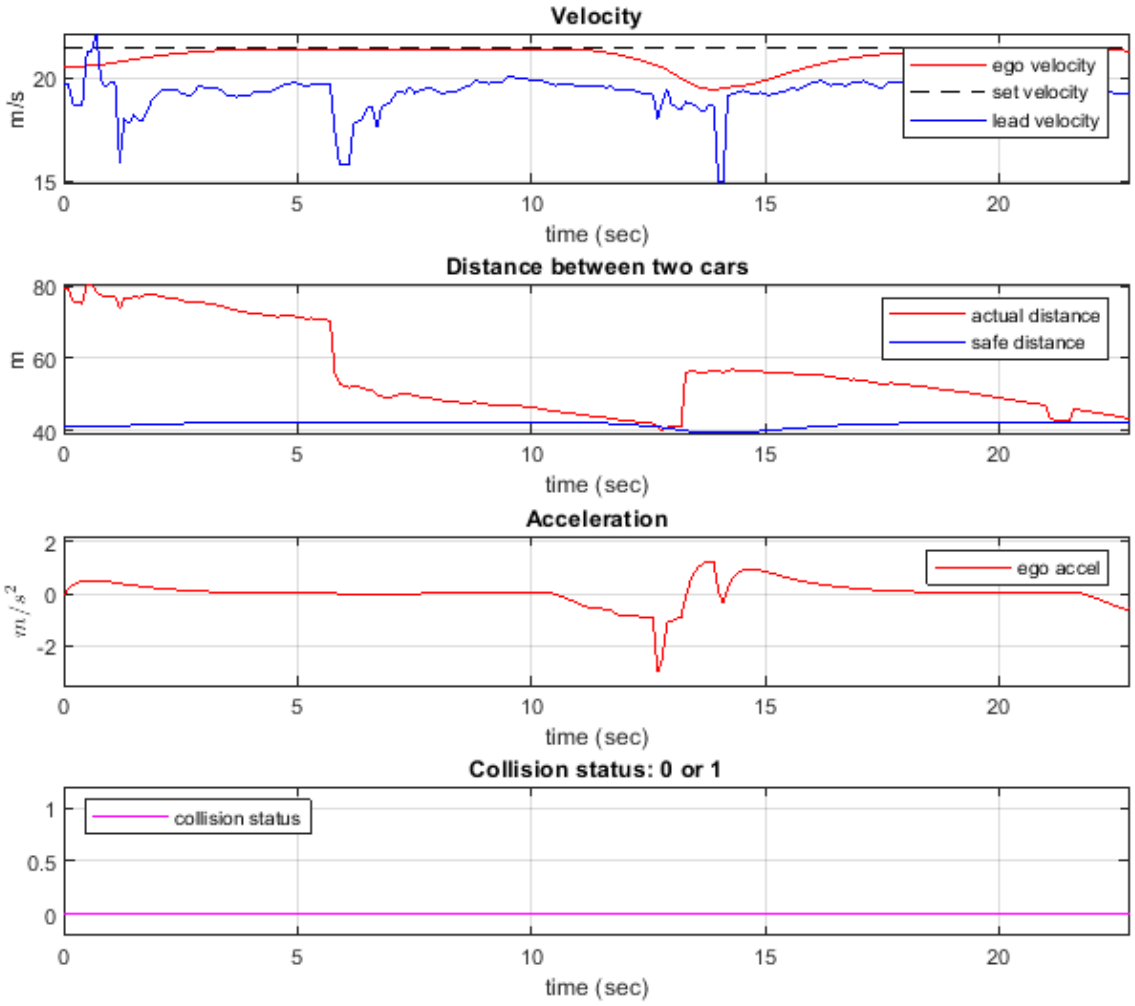
Simulate the model to the end of the scenario.

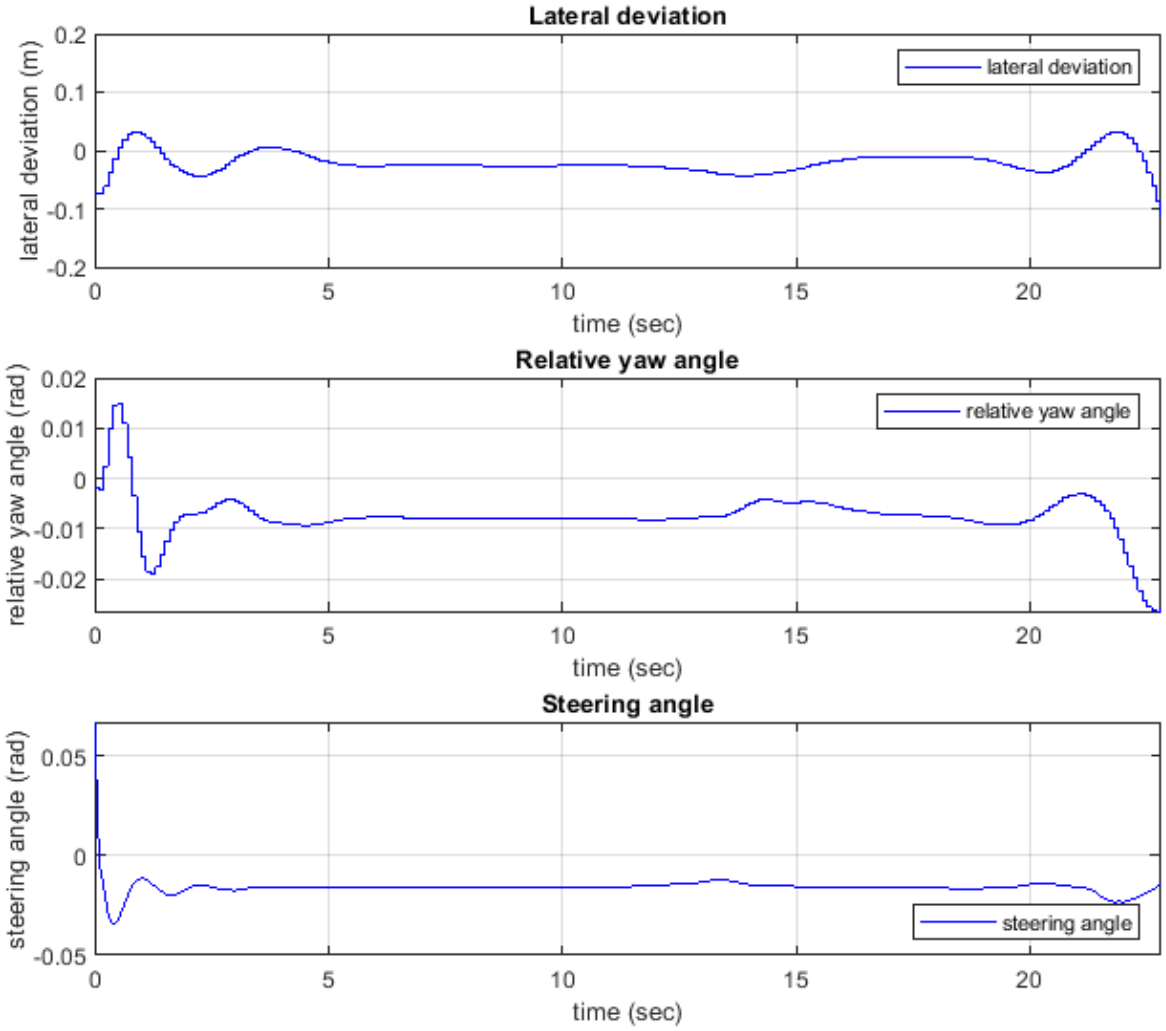
```
sim('LaneFollowingTestBenchExample')
```

```
   Assuming no disturbance added to measured output channel #3.
-->Assuming output disturbance added to measured output channel #2 is integrated white noise.
   Assuming no disturbance added to measured output channel #1.
-->Assuming output disturbance added to measured output channel #4 is integrated white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

Plot the controller performance.

```
plotLFResults(logsout,time_gap,default_spacing)
```


Velocity
m/s
20
15
0
5
10
15
20
time (sec)
ego velocity
set velocity
lead velocity
Distance between two cars
m
80
60
40
actual distance
safe distance
Acceleration
m/s^2
2
0
-2
ego accel
Collision status: 0 or 1
1
0.5
0
collision status



The first figure shows the following spacing control performance results.

- The **Velocity plot** shows that the ego vehicle maintains velocity control from 0 to 11 seconds, switches to spacing control from 11 to 16 seconds, then switches back to velocity control.
- The **Distance between two cars** plot shows that the actual distance between lead vehicle and ego vehicle is always greater than the safe distance.
- The **Acceleration** plot shows that the acceleration for ego vehicle is smooth.
- The **Collision status** plot shows that no collision between lead vehicle and ego vehicle is detected, thus the ego vehicle runs in a safe mode.

The second figure shows the following lateral control performance results.

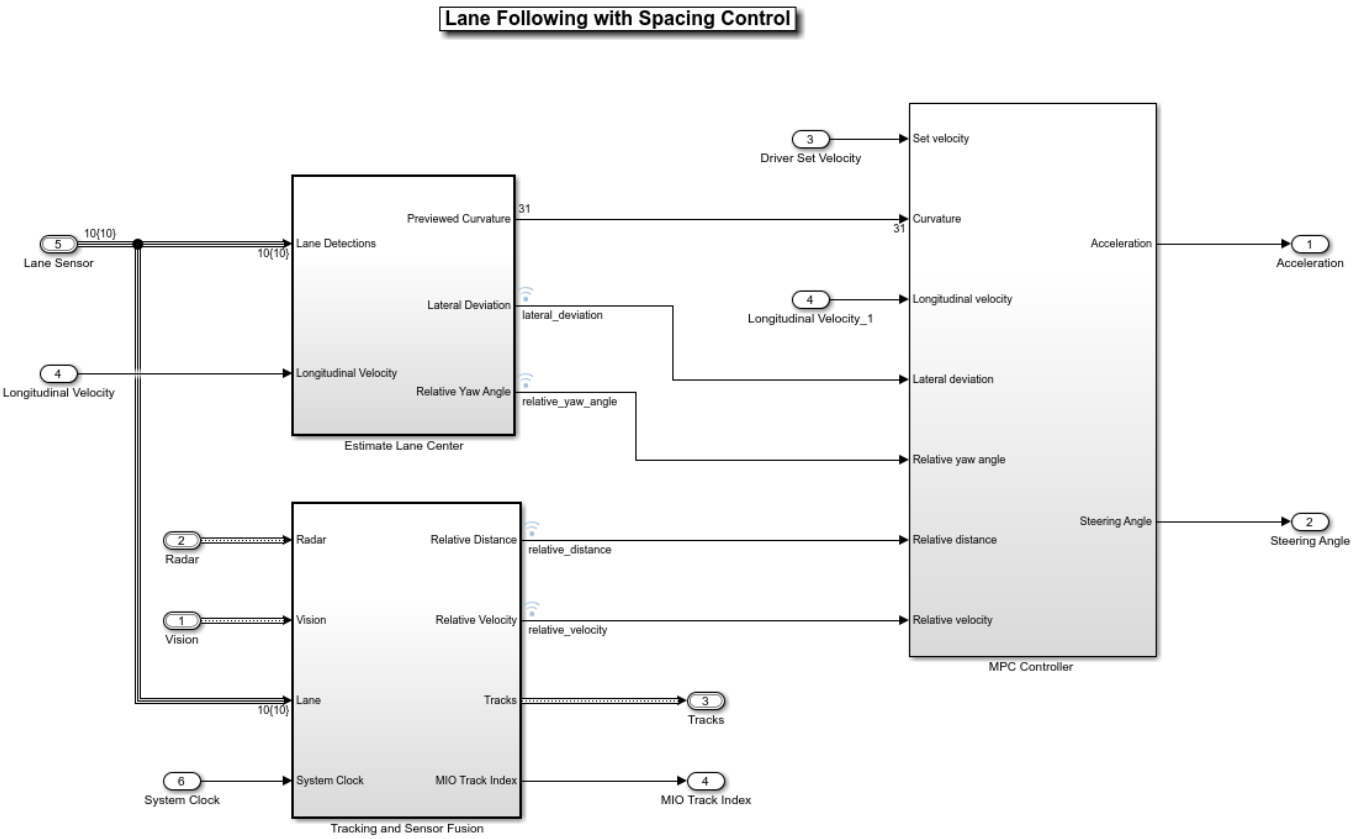
- The **Lateral deviation** plot shows that the distance to the lane centerline is within 0.2 m.
- The **Relative yaw angle** plot shows that the yaw angle error with respect to lane centerline is within 0.03 rad (less than 2 degrees).

- The **Steering angle** plot shows that the steering angle for ego vehicle is smooth.

Explore Lane Following Controller

The Lane Following Controller subsystem contains three main parts: 1) Estimate Lane Center 2) Tracking and Sensor Fusion 3) MPC Controller

```
open_system('LaneFollowingTestBenchExample/Lane Following Controller')
```

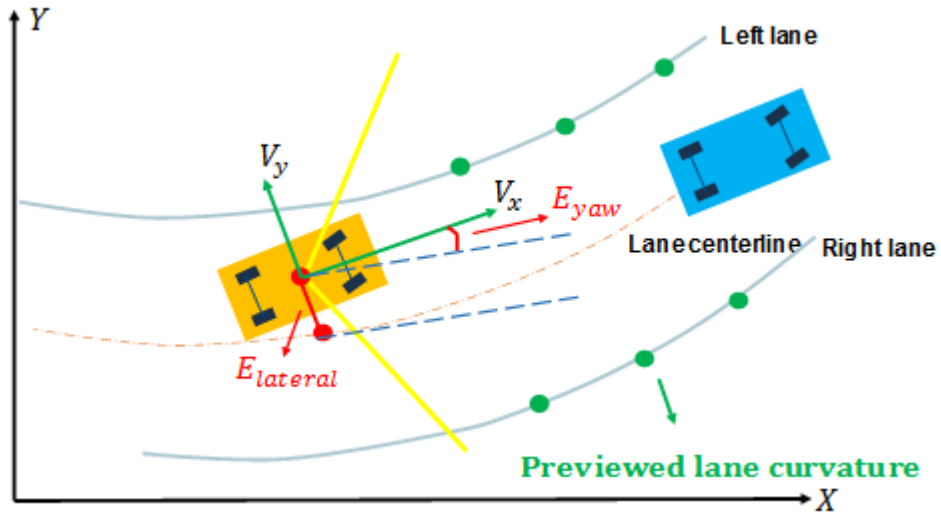


The Estimate Lane Center subsystem outputs the lane sensor data to the MPC controller. The previewed curvature provides the centerline of lane curvature ahead of the ego vehicle. In this example, the ego vehicle can look ahead for 3 seconds, which is the product of the prediction horizon and the controller sample time. The controller uses previewed information for calculating the ego vehicle steering angle, which improves the MPC controller performance. The lateral deviation measures the distance between the ego vehicle and the centerline of the lane. The relative yaw angle measures the yaw angle difference between the ego vehicle and the road. The ISO 8855 to SAE J670E block inside the subsystem converts the coordinates from Lane Detections, which use ISO 8855, to the MPC Controller which uses SAE J670E.

The Tracking and Sensor Fusion subsystem processes vision and radar detections coming from the Vehicle and Environment subsystem and generates a comprehensive situation picture of the environment around the ego vehicle. Also, it provides the lane following controller with an estimate of the closest vehicle in the lane in front of the ego vehicle.

The goals for the MPC Controller block are to:

- Maintain the driver-set velocity and keep a safe distance from lead vehicle. This goal is achieved by controlling the longitudinal acceleration.
- Keep the ego vehicle in the middle of the lane; that is reduce the lateral deviation $E_{lateral}$ and the relative yaw angle E_{yaw}, by controlling the steering angle.
- Slow down the ego vehicle when road is curvy. To achieve this goal, the MPC controller has larger penalty weights on lateral deviation than on longitudinal speed.



The MPC controller is designed within the Path Following Control (PFC) System block based on the entered mask parameters, and the designed MPC Controller is an adaptive MPC which updates the vehicle model at run time. The lane following controller calculates the longitudinal acceleration and steering angle for the ego vehicle based on the following inputs:

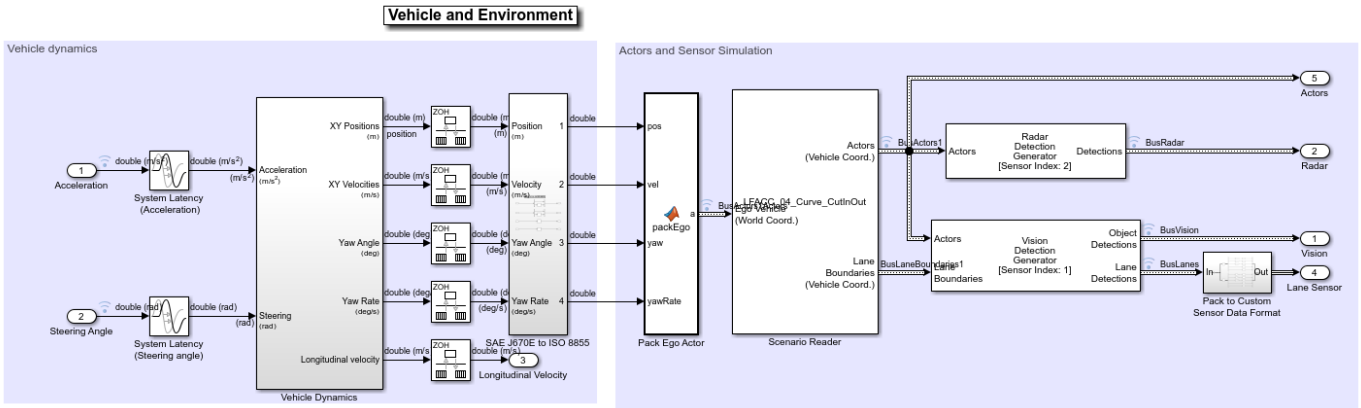
- Driver-set velocity
- Ego vehicle longitudinal velocity
- Previewed curvature (derived from Lane Detections)
- Lateral deviation (derived from Lane Detections)
- Relative yaw angle (derived from Lane Detections)
- Relative distance between lead vehicle and ego vehicle (from the Tracking and Sensor Fusion system)
- Relative velocity between lead vehicle and ego vehicle (from the Tracking and Sensor Fusion system)

Considering the physical limitations of the ego vehicle, the steering angle is constrained to be within [-0.26,0.26] rad, and the longitudinal acceleration is constrained to be within [-3,2] m/s^2.

Explore Vehicle and Environment

The Vehicle and Environment subsystem enables closed-loop simulation of the lane following controller.

```
open_system('LaneFollowingTestBenchExample/Vehicle and Environment')
```



The System Latency blocks model the latency in the system between model inputs and outputs. The latency can be caused by sensor delay or communication delay. In this example, the latency is approximated by one sample time $T_s = 0.1$ seconds.

The Vehicle Dynamics subsystem models the vehicle dynamics using a Bicycle Model - Force Input block from the Vehicle Dynamics Blockset™. The lower-level dynamics are modeled by a first-order linear system with a time constant of $\tau = 0.5$ seconds.

The SAE J670E to ISO 8855 subsystem converts the coordinates from Vehicle Dynamics, which uses SAE J670E, to Scenario Reader, which uses ISO 8855.

The Scenario Reader block reads the actor poses data from the scenario file. The block converts the actor poses from the world coordinates of the scenario into ego vehicle coordinates. The actor poses are streamed on a bus generated by the block. The Scenario Reader block also generates the ideal left and right lane boundaries based on the position of the vehicle with respect to the scenario used in `helperLFSetUp`.

The Vision Detection Generator block takes the ideal lane boundaries from the Scenario Reader block. The detection generator models the field of view of a monocular camera and determines the heading angle, curvature, curvature derivative, and valid length of each road boundary, accounting for any other obstacles. The Radar Detection block generates point detections from the ground-truth data present in the field-of-view of the radar based on the radar cross-section defined in the scenario.

Run Controller for Multiple Test Scenarios

This example uses multiple test scenarios based on ISO standards and real-world scenarios. To verify the controller performance, you can test the controller for multiple scenarios and tune the controller parameters if the performance is not satisfactory. To do so:

1. Select the scenario by changing `scenarioId` in `helperLFSetUp`.
2. Configure the simulation parameters by running `helperLFSetUp`.
3. Simulate the model with the selected scenario.
4. Evaluate the controller performance using `plotLFResults`
5. Tune the controller parameters if the performance is not satisfactory.

You can automate the verification and validation of the controller using Simulink Test™.

Generate Code for the Control Algorithm

The `LFRefMdl` model supports generating C code using Embedded Coder® software. To check if you have access to Embedded Coder, run:

```
hasEmbeddedCoderLicense = license('checkout','RTW_Embedded_Coder')
```

You can generate a C function for the model and explore the code generation report by running:

```
if hasEmbeddedCoderLicense
  rtwbuild('LFRefMdl')
end
```

You can verify that the compiled C code behaves as expected using software-in-the-loop (SIL) simulation. To simulate the `LFRefMdl` referenced model in SIL mode, use:

```
if hasEmbeddedCoderLicense
  set_param('LaneFollowingTestBenchExample/Lane Following Controller',...
            'SimulationMode','Software-in-the-loop (SIL)')
end
```

When you run the `LaneFollowingTestBenchExample` model, code is generated, compiled, and executed for the `LFRefMdl` model, which enables you to test the behavior of the compiled code through simulation.

Conclusions

This example shows how to implement an integrated lane following controller on a curved road with sensor fusion and lane detection, test it in Simulink using synthetic data generated using Automated Driving Toolbox software, componentize it, and automatically generate code for it.

```
close all
bdclose all
```

See Also

Apps
Bird's-Eye Scope

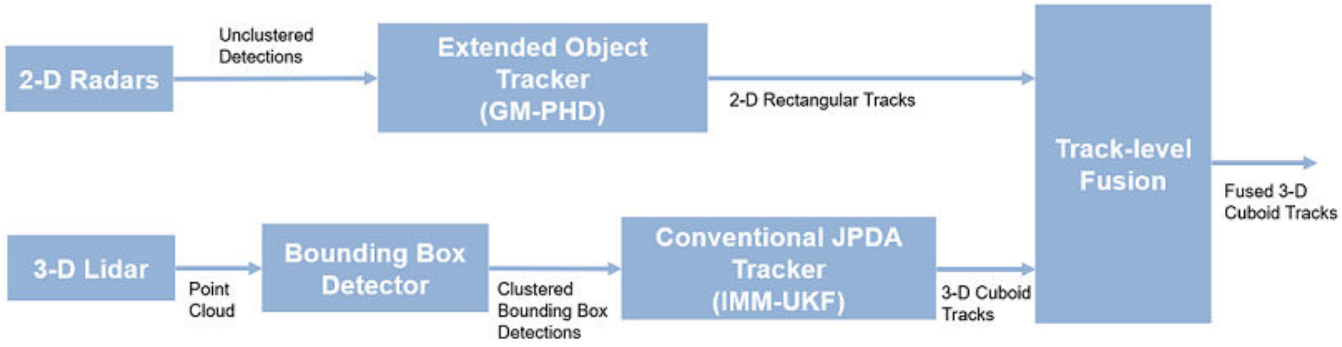
Blocks
Lane Keeping Assist System

More About

- “Adaptive Cruise Control with Sensor Fusion” on page 7-138
- “Lane Keeping Assist with Lane Detection” on page 7-363

Track-Level Fusion of Radar and Lidar Data

This example shows you how to generate an object-level track list from measurements of a radar and a lidar sensor and further fuse them using a track-level fusion scheme. You process the radar measurements using an extended object tracker and the lidar measurements using a joint probabilistic data association (JPDA) tracker. You further fuse these tracks using a track-level fusion scheme. The schematics of the workflow is shown below.

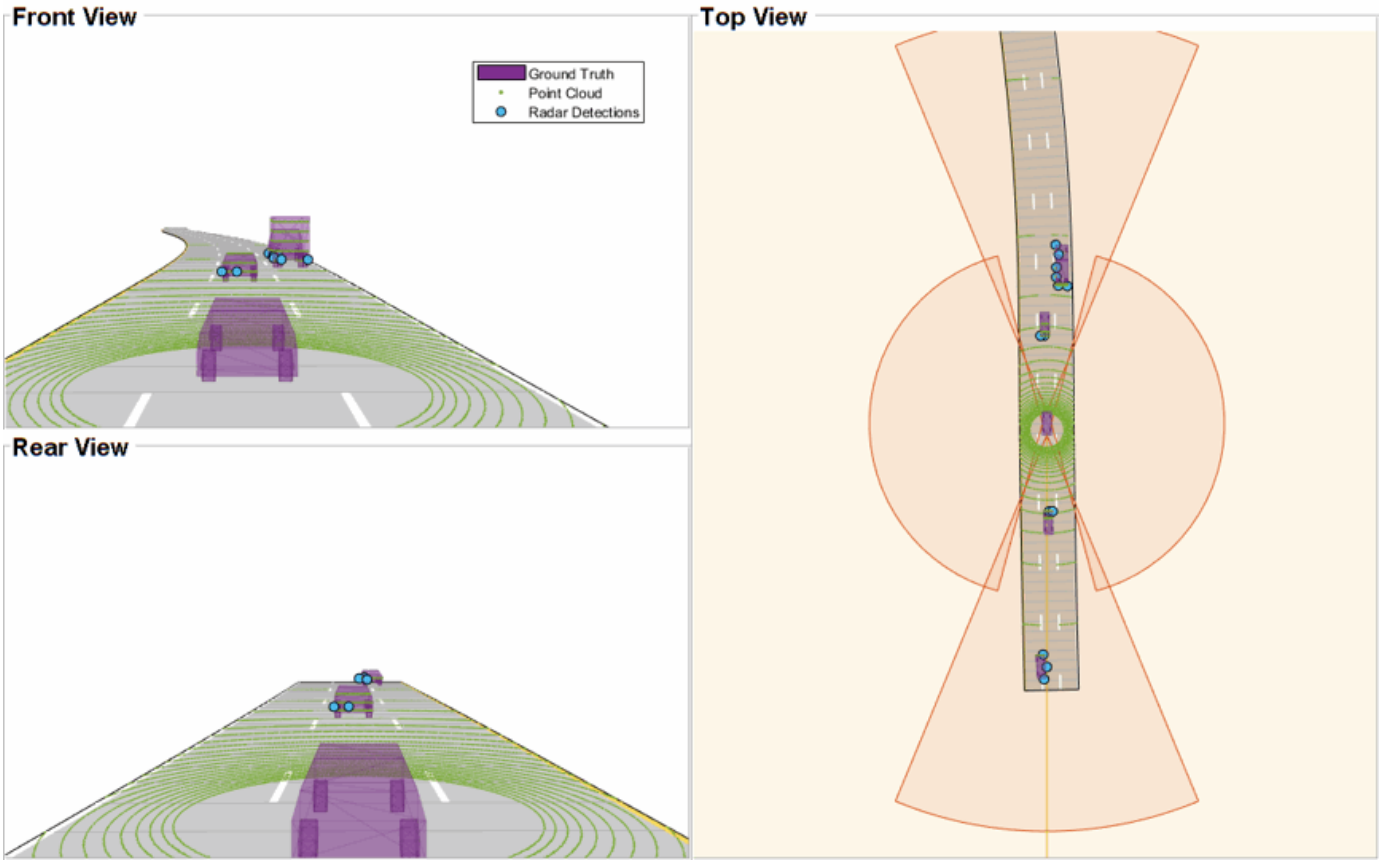


Setup Scenario for Synthetic Data Generation

The scenario used in this example is created using `drivingScenario`. The data from radar and lidar sensors is simulated using `radarDetectionGenerator` and `lidarPointCloudGenerator`, respectively. The creation of the scenario and the sensor models is wrapped in the helper function `helperCreateRadarLidarScenario`. For more information on scenario and synthetic data generation, refer to “Create Driving Scenario Programmatically” on page 7-423.

```
% For reproducible results
rng(2019);

% Create scenario, ego vehicle and get radars and lidar sensor
[scenario, egoVehicle, radars, lidar] = helperCreateRadarLidarScenario;
```

The Ego vehicle is mounted with four 2-D radar sensors. The front and rear radar sensors have a field of view of 45 degrees. The left and right radar sensors have a field of view of 150 degrees. Each radar has a resolution of 6 degrees in azimuth and 2.5 meters in range. The Ego is also mounted with one 3-D lidar sensor with a field of view of 360 degrees in azimuth and 40 degrees in elevation. The lidar has a resolution of 0.2 degrees in azimuth and 1.25 degrees in elevation (32 elevation channels). Visualize the configuration of the sensors and the simulated sensor data in the animation below. Notice that the radars have higher resolution than objects and therefore return multiple measurements per object. Also notice that the lidar interacts with the low-poly mesh of the actor as well as the road surface to return multiple points from these objects.



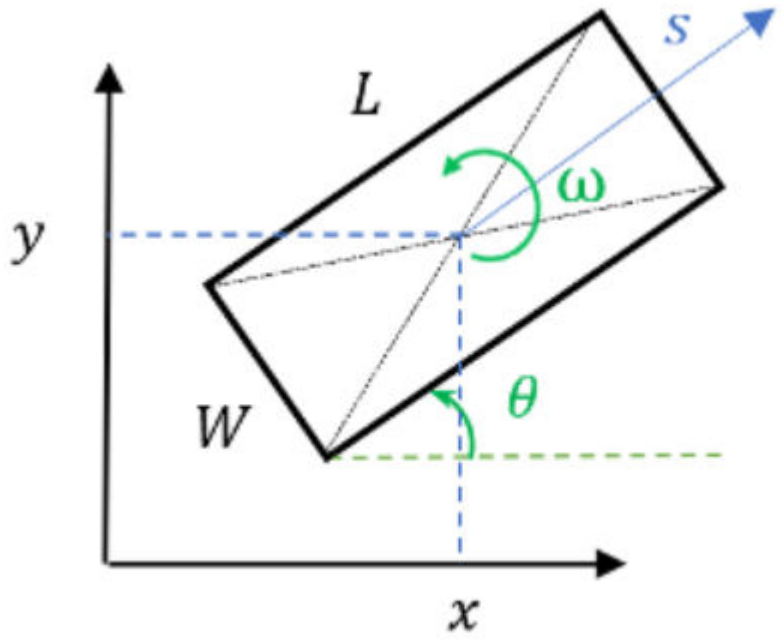
Radar Tracking Algorithm

As mentioned, the radars have higher resolution than the objects and return multiple detections per object. Conventional trackers such as Global Nearest Neighbor (GNN) and Joint Probabilistic Data Association (JPDA) assume that the sensors return at most one detection per object per scan. Therefore, the detections from high-resolution sensors must be either clustered before processing it with conventional trackers or must be processed using extended object trackers. Extended object trackers do not require pre-clustering of detections and usually estimate both kinematic states (for example, position and velocity) and the extent of the objects. For a more detailed comparison between conventional trackers and extended object trackers, refer to the “Extended Object Tracking of Highway Vehicles with Radar and Camera” (Sensor Fusion and Tracking Toolbox) example.

In general, extended object trackers offer better estimation of objects as they handle clustering and data association simultaneously using temporal history of tracks. In this example, the radar detections are processed using a Gaussian mixture probability hypothesis density (GM-PHD) tracker (`trackerPHD` (Sensor Fusion and Tracking Toolbox) and `gmphd` (Sensor Fusion and Tracking Toolbox)) with a rectangular target model. For more details on configuring the tracker, refer to the "GM-PHD Rectangular Object Tracker" section of the “Extended Object Tracking of Highway Vehicles with Radar and Camera” (Sensor Fusion and Tracking Toolbox) example.

The algorithm for tracking objects using radar measurements is wrapped inside the helper class, `helperRadarTrackingAlgorithm`, implemented as a System object™. This class outputs an array of `objectTrack` (Sensor Fusion and Tracking Toolbox) objects and define their state according to the following convention:

$$[x\ y\ s\ \theta\ \omega\ L\ W]$$




```
radarTrackingAlgorithm = helperRadarTrackingAlgorithm(radars);
```

Lidar Tracking Algorithm

Similar to radars, the lidar sensor also return multiple measurements per object. Further, the sensor returns a large number of points from the road, which must be removed to before used as inputs for an object-tracking algorithm. While lidar data from obstacles can be directly processed via extended object tracking algorithm, conventional tracking algorithms are still more prevalent for tracking using lidar data. The first reason for this trend is mainly observed due to higher computational complexity of extended object trackers for large data sets. The second reason is the investments into advanced Deep learning-based detectors such as PointPillars [1], VoxelNet [2] and PIXOR [3], which can segment a point cloud and return bounding box detections for the vehicles. These detectors can help in overcoming the performance degradation of conventional trackers due to improper clustering.

In this example, the lidar data is processed using a conventional joint probabilistic data association (JPDA) tracker, configured with an interacting multiple model (IMM) filter. The pre-processing of lidar data to remove point cloud is performed by using a RANSAC-based plane-fitting algorithm and bounding boxes are formed by performing a Euclidian-based distance clustering algorithm. For more information about the algorithm, refer to the “Track Vehicles Using Lidar: From Point Cloud to Track List” (Sensor Fusion and Tracking Toolbox) example. Compared the linked example, the tracking is performed in the scenario frame and the tracker is tuned differently to track objects of different sizes. Further the states of the variables are defined differently to constraint the motion of the tracks in the direction of its estimated heading angle.

The algorithm for tracking objects using lidar data is wrapped inside the helper class, `helperLidarTrackingAlgorithm` implemented as System object. This class outputs an array of

`objectTrack` (Sensor Fusion and Tracking Toolbox) objects and defines their state according to the following convention:

$$[x\ y\ s\ \theta\ \omega\ z\ \dot{z}\ L\ W\ H]$$

The states common to the radar algorithm are defined similarly. Also, as a 3-D sensor, the lidar tracker outputs three additional states, z, $\dot{z}$ and H, which refer to z-coordinate (m), z-velocity (m/s), and height (m) of the tracked object respectively.

```
lidarTrackingAlgorithm = helperLidarTrackingAlgorithm(lidar);
```

Set Up Fuser, Metrics, and Visualization

Fuser

Next, you will set up a fusion algorithm for fusing the list of tracks from radar and lidar trackers. Similar to other tracking algorithms, the first step towards setting up a track-level fusion algorithm is defining the choice of state vector (or state-space) for the fused or central tracks. In this case, the state-space for fused tracks is chosen to be same as the lidar. After choosing a central track state-space, you define the transformation of the central track state to the local track state. In this case, the local track state-space refers to states of radar and lidar tracks. To do this, you use a `fuserSourceConfiguration` (Sensor Fusion and Tracking Toolbox) object.

Define the configuration of the radar source. The `helperRadarTrackingAlgorithm` outputs tracks with `SourceIndex` set to 1. The `SourceIndex` is provided as a property on each tracker to uniquely identify it and allows a fusion algorithm to distinguish tracks from different sources. Therefore, you set the `SourceIndex` property of the radar configuration as same as those of the radar tracks. You set `IsInitializingCentralTracks` to `true` to let that unassigned radar tracks initiate new central tracks. Next, you define the transformation of a track in central state-space to the radar state-space and vice-versa. The helper functions `central2radar` and `radar2central` perform the two transformations and are included at the end of this example.

```
radarConfig = fuserSourceConfiguration('SourceIndex',1,...
    'IsInitializingCentralTracks',true,...
    'CentralToLocalTransformFcn',@central2radar,...
    'LocalToCentralTransformFcn',@radar2central);
```

Define the configuration of the lidar source. Since the state-space of a lidar track is same as central track, you do not define any transformations.

```
lidarConfig = fuserSourceConfiguration('SourceIndex',2,...
    'IsInitializingCentralTracks',true);
```

The next step is to define the state-fusion algorithm. The state-fusion algorithm takes multiple states and state covariances in the central state-space as input and return a fused estimate of the state and the covariances. In this example, you use a covariance intersection algorithm provided by the helper function, `helperRadarLidarFusionFcn`. A generic covariance intersection algorithm for two Gaussian estimates with mean x_i and covariance P_i can be defined according to the following equations:

$$P_F^{-1} = w_1 {P_1}^{-1} + w_2 {P_2}^{-1}$$

$$x_F = P_F(w_1 {P_1}^{-1} x_1 + w_2 {P_2}^{-1} x_2)$$

where x_F and P_F are the fused state and covariance and w_1 and w_2 are mixing coefficients from each estimate. Typically, these mixing coefficients are estimated by minimizing the determinant or the trace of the fused covariance. In this example, the mixing weights are estimated by minimizing the determinant of positional covariance of each estimate. Furthermore, as the radar does not estimate 3-D states, 3-D states are only fused with lidars. For more details, refer to the `helperRadarLidarFusionFcn` function shown at the end of this script.

Next, you assemble all the information using a `trackFuser` object.

```
% The state-space of central tracks is same as the tracks from the lidar,
% therefore you use the same state transition function. The function is
% defined inside the helperLidarTrackingAlgorithm class.
f = lidarTrackingAlgorithm.StateTransitionFcn;

% Create a trackFuser object
fuser = trackFuser('SourceConfigurations',{radarConfig;lidarConfig},...
    'StateTransitionFcn',f,...
    'ProcessNoise',diag([1 3 1]),...
    'HasAdditiveProcessNoise',false,...
    'AssignmentThreshold',[250 inf],...
    'ConfirmationThreshold',[3 5],...
    'DeletionThreshold',[5 5],...
    'StateFusion','Custom',...
    'CustomStateFusionFcn',@helperRadarLidarFusionFcn);
```

Metrics

In this example, you assess the performance of each algorithm using the Generalized Optimal SubPattern Assignment Metric (GOSPA) metric. You setup three separate metrics using `trackGOSPAMetric` (Sensor Fusion and Tracking Toolbox) for each of the trackers. GOSPA metric aims to evaluate the performance of a tracking system by providing a scalar cost. A lower value of the metric indicates better performance of the tracking algorithm.

To use the GOSPA metric with custom motion models like the one used in this example, you set the `Distance` property to 'custom' and define a distance function between a track and its associated ground truth. These distance functions, shown at the end of this example are `helperRadarDistance`, and `helperLidarDistance`.

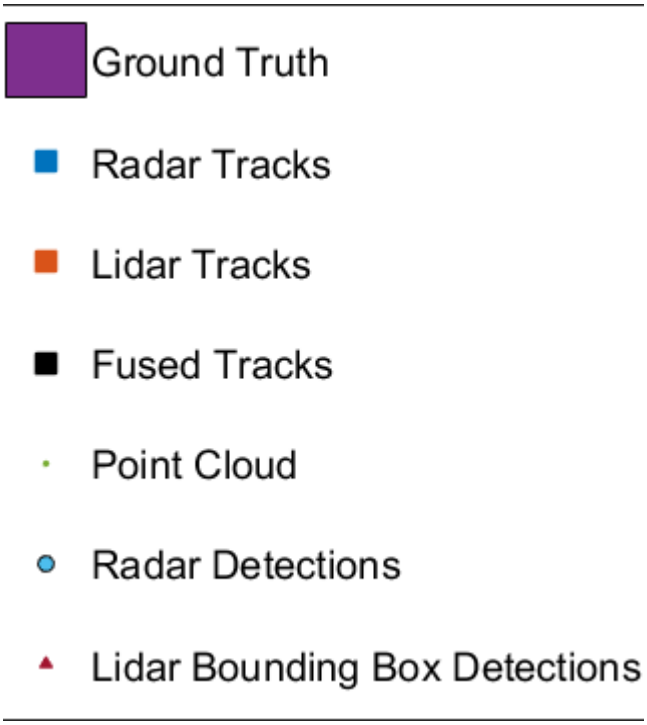
```
% Radar GOSPA
gospaRadar = trackGOSPAMetric('Distance','custom',...
    'DistanceFcn',@helperRadarDistance,...
    'CutoffDistance',25);

% Lidar GOSPA
gospaLidar = trackGOSPAMetric('Distance','custom',...
    'DistanceFcn',@helperLidarDistance,...
    'CutoffDistance',25);

% Central/Fused GOSPA
gospaCentral = trackGOSPAMetric('Distance','custom',...
    'DistanceFcn',@helperLidarDistance,...% State-space is same as lidar
    'CutoffDistance',25);
```

Visualization

The visualization for this example is implemented using a helper class `helperLidarRadarTrackFusionDisplay`. The display is divided into 4 panels. The display plots

the measurements and tracks from each sensor as well as the fused track estimates. The legend for the display is shown below. Furthermore, the tracks are annotated by their unique identity (`TrackID`) as well as a prefix. The prefixes "R", "L" and "F" stand for radar, lidar, and fused estimate, respectively.

```
% Create a display.
% FollowActorID controls the actor shown in the close-up
% display
display = helperLidarRadarTrackFusionDisplay('FollowActorID',3);

% Show persistent legend
showLegend(display,scenario);
```



Run Scenario and Trackers

Next, you advance the scenario, generate synthetic data from all sensors and process it to generate tracks from each of the system. You also compute the metric for each tracker using the ground truth available from the scenario.

```
% Initialzie GOSPA metric and its components for all tracking algorithms.
gospa = zeros(3,0);
missTarget = zeros(3,0);
falseTracks = zeros(3,0);

% Initialize fusedTracks
fusedTracks = objectTrack.empty(0,1);

% A counter for time steps elapsed for storing gospa metrics.
idx = 1;

% Ground truth for metrics. This variable updates every time-step
% automatically being a handle to the actors.
```

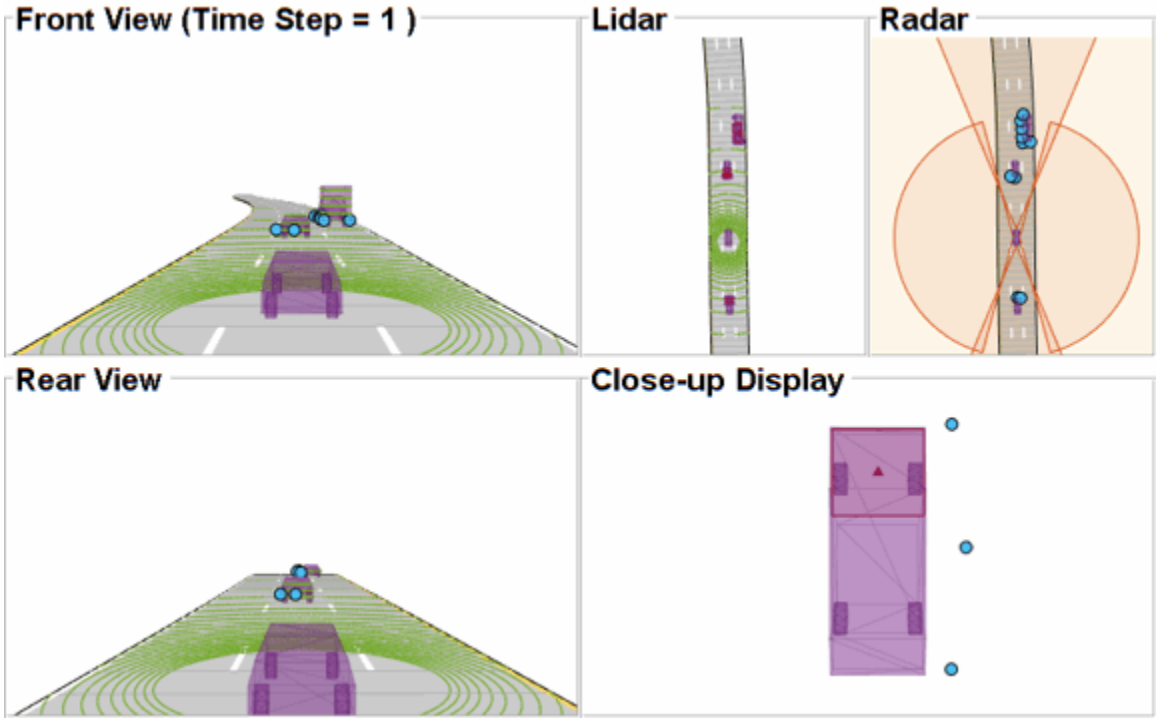
```
groundTruth = scenario.Actors(2:end);

while advance(scenario)
    % Current time
    time = scenario.SimulationTime;

    % Collect radar and lidar measurements and ego pose to track in
    % scenario frame. See helperCollectSensorData below.
    [radarDetections, ptCloud, egoPose] = helperCollectSensorData(egoVehicle, radars, lidar, time

    % Generate radar tracks
    radarTracks = radarTrackingAlgorithm(egoPose, radarDetections, time);

    % Generate lidar tracks and analysis information like bounding box
    % detections and point cloud segmentation information
    [lidarTracks, lidarDetections, segmentationInfo] = ...
        lidarTrackingAlgorithm(egoPose, ptCloud, time);

    % Concatenate radar and lidar tracks
    localTracks = [radarTracks;lidarTracks];

    % Update the fuser. First call must contain one local track
    if ~(isempty(localTracks) && ~isLocked(fuser))
        fusedTracks = fuser(localTracks,time);
    end

    % Capture GOSPA and its components for all trackers
    [gospa(1,idx),~,~,~,missTarget(1,idx),falseTracks(1,idx)] = gospaRadar(radarTracks, groundTru
    [gospa(2,idx),~,~,~,missTarget(2,idx),falseTracks(2,idx)] = gospaLidar(lidarTracks, groundTru
    [gospa(3,idx),~,~,~,missTarget(3,idx),falseTracks(3,idx)] = gospaCentral(fusedTracks, groundT

    % Update the display
    display(scenario, radars, radarDetections, radarTracks, ...
        lidar, ptCloud, lidarDetections, segmentationInfo, lidarTracks,...
        fusedTracks);

    % Update the index for storing GOSPA metrics
    idx = idx + 1;
end

% Update example animations
updateExampleAnimations(display);
```

Evaluate Performance

Evaluate the performance of each tracker using visualization as well as quantitative metrics. Analyze different events in the scenario and understand how the track-level fusion scheme helps achieve a better estimation of the vehicle state.

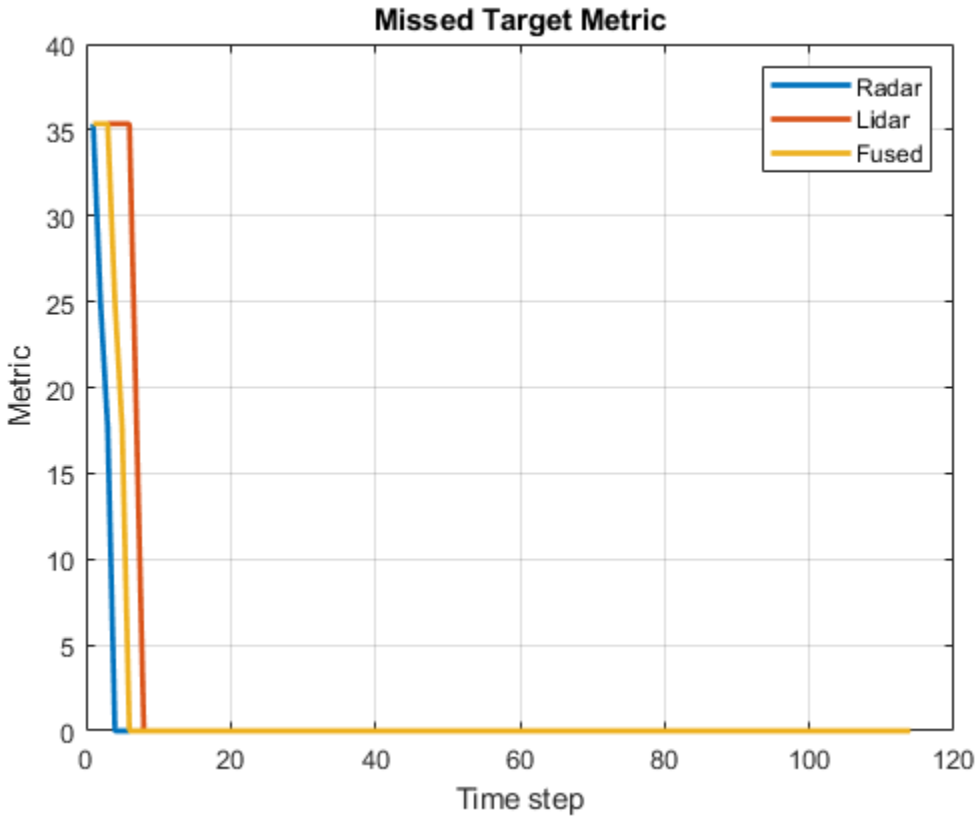
Track Maintenance

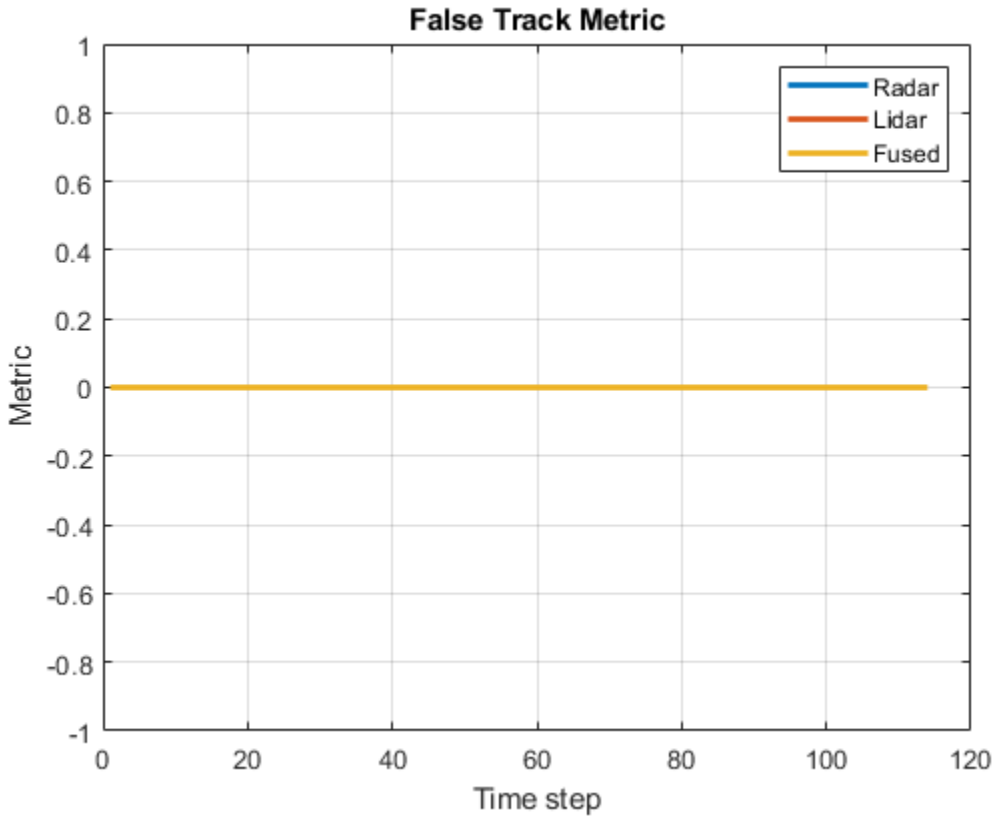
The animation below shows the entire run every three time-steps. Note that each of the three tracking systems - radar, lidar, and the track-level fusion - were able to track all four vehicles in the scenario and no false tracks were confirmed.



You can also quantitatively measure this aspect of the performance using "missed target" and "false track" components of the GOSPA metric. Notice in the figures below that missed target component starts from a higher value due to establishment delay and goes down to zero in about 5-10 steps for each tracking system. Also, notice that the false track component is zero for all systems, which indicates that no false tracks were confirmed.

```
% Plot missed target component
figure; plot(missTarget','LineWidth',2); legend('Radar','Lidar','Fused');
title("Missed Target Metric"); xlabel('Time step'); ylabel('Metric'); grid on;

% Plot false track component
figure; plot(falseTracks','LineWidth',2); legend('Radar','Lidar','Fused');
title("False Track Metric"); xlabel('Time step'); ylabel('Metric'); grid on;
```


Missed Target Metric
Radar
Lidar
Fused
Metric
Time step
0
5
10
15
20
25
30
35
40
0
20
40
60
80
100
120

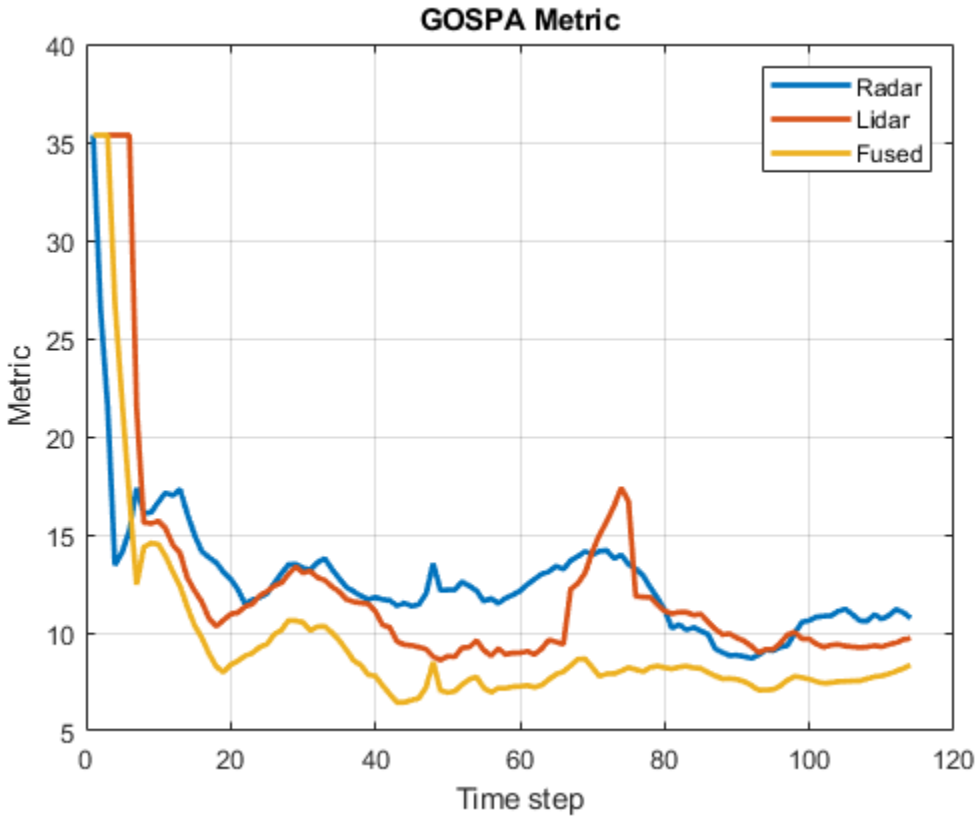


Track-level Accuracy

The track-level or localization accuracy of each tracker can also be quantitatively assessed by the GOSPA metric at each time step. A lower value indicates better tracking accuracy. As there were no missed targets or false tracks, the metric captures the localization errors resulting from state estimation of each vehicle.

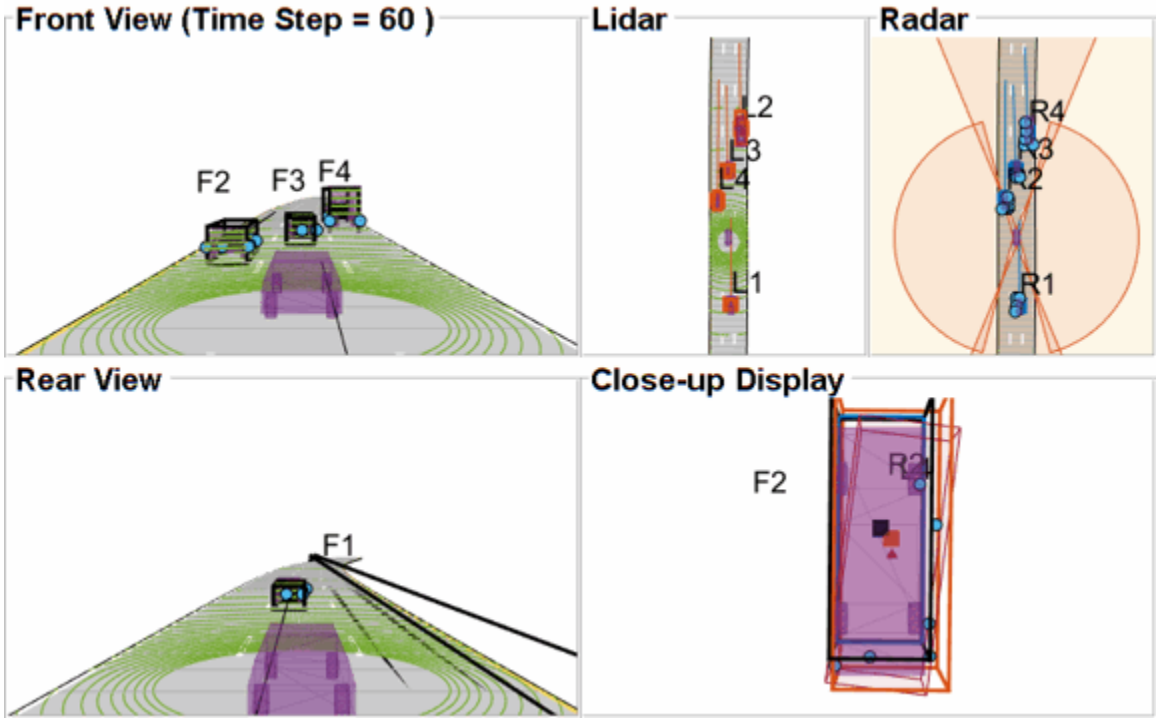
Note that the GOSPA metric for fused estimates is lower than the metric for individual sensor, which indicates that track accuracy increased after fusion of track estimates from each sensor.

```
% Plot GOSPA
figure; plot(gospa','LineWidth',2); legend('Radar','Lidar','Fused');
title("GOSPA Metric"); xlabel('Time step'); ylabel('Metric'); grid on;
```



Closely-spaced targets

As mentioned earlier, this example uses a Euclidian-distance based clustering and bounding box fitting to feed the lidar data to a conventional tracking algorithm. Clustering algorithms typically suffer when objects are closely-spaced. With the detector configuration used in this example, when the passing vehicle approaches the vehicle in front of the ego vehicle, the detector clusters the point cloud from each vehicle into a bigger bounding box. You can notice in the animation below that the track drifted away from the vehicle center. Because the track was reported with higher certainty in its estimate for a few steps, the fused estimated was also affected initially. However, as the uncertainty increases, its association with the fused estimate becomes weaker. This is because the covariance intersection algorithm chooses a mixing weight for each assigned track based on the certainty of each estimate.

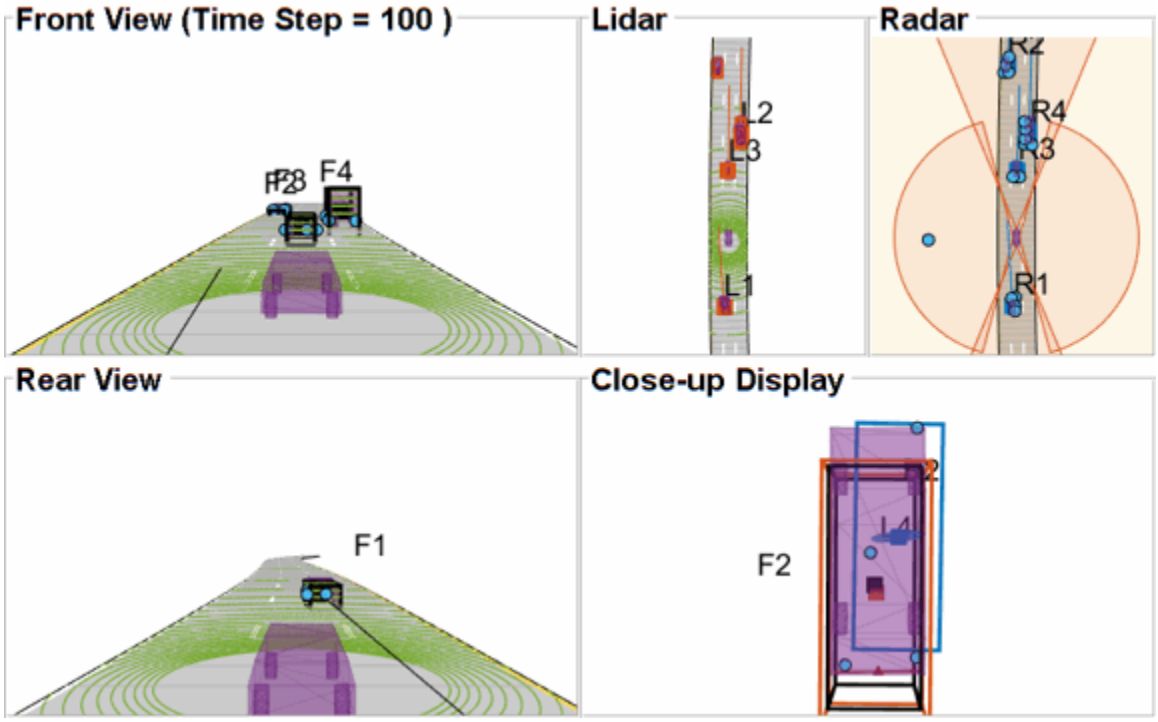


This effect is also captured in the GOSPA metric. You can notice in the GOSPA metric plot above that the lidar metric shows a peak around the 65th time step.

The radar tracks are not affected during this event because of two main reasons. Firstly, the radar sensor outputs range-rate information in each detection, which is different beyond noise-levels for the passing car as compared to the slower moving car. This results in an increased statistical distance between detections from individual cars. Secondly, extended object trackers evaluate multiple possible clustering hypothesis against predicted tracks, which results in rejection of improper clusters and acceptance of proper clusters. Note that for extended object trackers to properly choose the best clusters, the filter for the track must be robust to a degree that can capture the difference between two clusters. For example, a track with high process noise and highly uncertain dimensions may not be able to properly claim a cluster because of its premature age and higher flexibility to account for uncertain events.

Targets at long range

As targets recede away from the radar sensors, the accuracy of the measurements degrade because of reduced signal-to-noise ratio at the detector and the limited resolution of the sensor. This results in high uncertainty in the measurements, which in turn reduces the track accuracy. Notice in the close-up display below that the track estimate from the radar is further away from the ground truth for the radar sensor and is reported with a higher uncertainty. However, the lidar sensor reports enough measurements in the point cloud to generate a "shrunk" bounding box. The shrinkage effect modeled in the measurement model for lidar tracking algorithm allows the tracker to maintain a track with correct dimensions. In such situations, the lidar mixing weight is higher than the radar and allows the fused estimate to be more accurate than the radar estimate.



Summary

In this example, you learned how to setup a track-level fusion algorithm for fusing tracks from radar and lidar sensors. You also learned how to evaluate a tracking algorithm using the Generalized Optimal Subpattern Metric and its associated components.

Utility Functions

`collectSensorData`

A function to generate radar and lidar measurements at the current time-step.

```
function [radarDetections, ptCloud, egoPose] = helperCollectSensorData(egoVehicle, radars, lidar

% Current poses of targets with respect to ego vehicle
tgtPoses = targetPoses(egoVehicle);

radarDetections = cell(0,1);
for i = 1:numel(radars)
    thisRadarDetections = step(radars{i},tgtPoses,time);
    radarDetections = [radarDetections;thisRadarDetections]; %#ok<AGROW>
end

% Generate point cloud from lidar
rdMesh = roadMesh(egoVehicle);
ptCloud = step(lidar, tgtPoses, rdMesh, time);

% Compute pose of ego vehicle to track in scenario frame. Typically
% obtained using an INS system. If unavailable, this can be set to
% "origin" to track in Ego vehicle's frame.
egoPose = pose(egoVehicle);
```

```
end
```

radar2cental

A function to transform a track in the radar state-space to a track in the central state-space.

```
function centralTrack = radar2central(radarTrack)

% Initialize a track of the correct state size
centralTrack = objectTrack('State',zeros(10,1),...
    'StateCovariance',eye(10));

% Sync properties of radarTrack except State and StateCovariance with
% radarTrack See syncTrack defined below.
centralTrack = syncTrack(centralTrack,radarTrack);

xRadar = radarTrack.State;
PRadar = radarTrack.StateCovariance;

H = zeros(10,7); % Radar to central linear transformation matrix
H(1,1) = 1;
H(2,2) = 1;
H(3,3) = 1;
H(4,4) = 1;
H(5,5) = 1;
H(8,6) = 1;
H(9,7) = 1;

xCentral = H*xRadar;  % Linear state transformation
PCentral = H*PRadar*H'; % Linear covariance transformation

PCentral([6 7 10],[6 7 10]) = eye(3); % Unobserved states

% Set state and covariance of central track
centralTrack.State = xCentral;
centralTrack.StateCovariance = PCentral;

end
```

central2radar

A function to transform a track in the central state-space to a track in the radar state-space.

```
function radarTrack = central2radar(centralTrack)

% Initialize a track of the correct state size
radarTrack = objectTrack('State',zeros(7,1),...
    'StateCovariance',eye(7));

% Sync properties of centralTrack except State and StateCovariance with
% radarTrack See syncTrack defined below.
radarTrack = syncTrack(radarTrack,centralTrack);

xCentral = centralTrack.State;
PCentral = centralTrack.StateCovariance;

H = zeros(7,10); % Central to radar linear transformation matrix
```

```
H(1,1) = 1;
H(2,2) = 1;
H(3,3) = 1;
H(4,4) = 1;
H(5,5) = 1;
H(6,8) = 1;
H(7,9) = 1;

xRadar = H*xCentral;  % Linear state transformation
PRadar = H*PCentral*H'; % Linear covariance transformation

% Set state and covariance of radar track
radarTrack.State = xRadar;
radarTrack.StateCovariance = PRadar;
end
```

`syncTrack`

A function to syncs properties of one track with another except the "State" and "StateCovariance" property

```
function tr1 = syncTrack(tr1,tr2)
props = properties(tr1);
notState = ~strcmpi(props,'State');
notCov = ~strcmpi(props,'StateCovariance');

props = props(notState & notCov);
for i = 1:numel(props)
    tr1.(props{i}) = tr2.(props{i});
end
end
```

`pose`

A function to return pose of the ego vehicle as a structure.

```
function egoPose = pose(egoVehicle)
egoPose.Position = egoVehicle.Position;
egoPose.Velocity = egoVehicle.Velocity;
egoPose.Yaw = egoVehicle.Yaw;
egoPose.Pitch = egoVehicle.Pitch;
egoPose.Roll = egoVehicle.Roll;
end
```

`helperLidarDistance`

Function to calculate a normalized distance between the estimate of a track in radar state-space and the assigned ground truth.

```
function dist = helperLidarDistance(track, truth)

% Calculate the actual values of the states estimated by the tracker

% Center is different than origin and the trackers estimate the center
rOriginToCenter = -truth.OriginOffset(:) + [0;0;truth.Height/2];
rot = quaternion([truth.Yaw truth.Pitch truth.Roll],'eulerd','ZYX','frame');
actPos = truth.Position(:) + rotatepoint(rot,rOriginToCenter')';
```

```
% Actual speed and z-rate
actVel = [norm(truth.Velocity(1:2));truth.Velocity(3)];

% Actual yaw
actYaw = truth.Yaw;

% Actual dimensions.
actDim = [truth.Length;truth.Width;truth.Height];

% Actual yaw rate
actYawRate = truth.AngularVelocity(3);

% Calculate error in each estimate weighted by the "requirements" of the
% system. The distance specified using Mahalanobis distance in each aspect
% of the estimate, where covariance is defined by the "requirements". This
% helps to avoid skewed distances when tracks under/over report their
% uncertainty because of inaccuracies in state/measurement models.

% Positional error.
estPos = track.State([1 2 6]);
reqPosCov = 0.1*eye(3);
e = estPos - actPos;
d1 = sqrt(e'/reqPosCov*e);

% Velocity error
estVel = track.State([3 7]);
reqVelCov = 5*eye(2);
e = estVel - actVel;
d2 = sqrt(e'/reqVelCov*e);

% Yaw error
estYaw = track.State(4);
reqYawCov = 5;
e = estYaw - actYaw;
d3 = sqrt(e'/reqYawCov*e);

% Yaw-rate error
estYawRate = track.State(5);
reqYawRateCov = 1;
e = estYawRate - actYawRate;
d4 = sqrt(e'/reqYawRateCov*e);

% Dimension error
estDim = track.State([8 9 10]);
reqDimCov = eye(3);
e = estDim - actDim;
d5 = sqrt(e'/reqDimCov*e);

% Total distance
dist = d1 + d2 + d3 + d4 + d5;

end
```

helperRadarDistance

Function to calculate a normalized distance between the estimate of a track in radar state-space and the assigned ground truth.

```
function dist = helperRadarDistance(track, truth)

% Calculate the actual values of the states estimated by the tracker

% Center is different than origin and the trackers estimate the center
rOriginToCenter = -truth.OriginOffset(:) + [0;0;truth.Height/2];
rot = quaternion([truth.Yaw truth.Pitch truth.Roll],'eulerd','ZYX','frame');
actPos = truth.Position(:) + rotatepoint(rot,rOriginToCenter')';
actPos = actPos(1:2); % Only 2-D

% Actual speed
actVel = norm(truth.Velocity(1:2));

% Actual yaw
actYaw = truth.Yaw;

% Actual dimensions. Only 2-D for radar
actDim = [truth.Length;truth.Width];

% Actual yaw rate
actYawRate = truth.AngularVelocity(3);

% Calculate error in each estimate weighted by the "requirements" of the
% system. The distance specified using Mahalanobis distance in each aspect
% of the estimate, where covariance is defined by the "requirements". This
% helps to avoid skewed distances when tracks under/over report their
% uncertainty because of inaccuracies in state/measurement models.

% Positional error
estPos = track.State([1 2]);
reqPosCov = 0.1*eye(2);
e = estPos - actPos;
d1 = sqrt(e'/reqPosCov*e);

% Speed error
estVel = track.State(3);
reqVelCov = 5;
e = estVel - actVel;
d2 = sqrt(e'/reqVelCov*e);

% Yaw error
estYaw = track.State(4);
reqYawCov = 5;
e = estYaw - actYaw;
d3 = sqrt(e'/reqYawCov*e);

% Yaw-rate error
estYawRate = track.State(5);
reqYawRateCov = 1;
e = estYawRate - actYawRate;
d4 = sqrt(e'/reqYawRateCov*e);

% Dimension error
estDim = track.State([6 7]);
reqDimCov = eye(2);
e = estDim - actDim;
d5 = sqrt(e'/reqDimCov*e);
```

```
% Total distance
dist = d1 + d2 + d3 + d4 + d5;

% A constant penality for not measuring 3-D state
dist = dist + 3;

end
```

helperRadarLidarFusionFcn

Function to fuse states and state covariances in central track state-space

```
function [x,P] = helperRadarLidarFusionFcn(xAll,PAll)
n = size(xAll,2);
dets = zeros(n,1);

% Initialize x and P
x = xAll(:,1);
P = PAll(:,:,1);

onlyLidarStates = false(10,1);
onlyLidarStates([6 7 10]) = true;

% Only fuse this information with lidar
xOnlyLidar = xAll(onlyLidarStates,:);
POnlyLidar = PAll(onlyLidarStates,onlyLidarStates,:);

% States and covariances for intersection with radar and lidar both
xToFuse = xAll(~onlyLidarStates,:);
PToFuse = PAll(~onlyLidarStates,~onlyLidarStates,:);

% Sorted order of determinants. This helps to sequentially build the
% covariance with comparable determinations. For example, two large
% covariances may intersect to a smaller covariance, which is comparable to
% the third smallest covariance.
for i = 1:n
    dets(i) = det(PToFuse(1:2,1:2,i));
end
[~,idx] = sort(dets,'descend');
xToFuse = xToFuse(:,idx);
PToFuse = PToFuse(:,:,idx);

% Initialize fused estimate
thisX = xToFuse(:,1);
thisP = PToFuse(:,:,1);

% Sequential fusion
for i = 2:n
    [thisX,thisP] = fusecovintUsingPos(thisX, thisP, xToFuse(:,i), PToFuse(:,:,i));
end

% Assign fused states from all sources
x(~onlyLidarStates) = thisX;
P(~onlyLidarStates,~onlyLidarStates,:) = thisP;

% Fuse some states only with lidar source
valid = any(abs(xOnlyLidar) > 1e-6,1);
```

```
xMerge = xOnlyLidar(:,valid);
PMerge = POnlyLidar(:,:,valid);

if sum(valid) > 1
    [xL,PL] = fusecovint(xMerge,PMerge);
elseif sum(valid) == 1
    xL = xMerge;
    PL = PMerge;
else
    xL = zeros(3,1);
    PL = eye(3);
end

x(onlyLidarStates) = xL;
P(onlyLidarStates,onlyLidarStates) = PL;

end

function [x,P] = fusecovintUsingPos(x1,P1,x2,P2)
% Covariance intersection in general is employed by the following
% equations:
% P^-1 = w1*P1^-1 + w2*P2^-1
% x = P*(w1*P1^-1*x1 + w2*P2^-1*x2);
% where w1 + w2 = 1
% Usually a scalar representative of the covariance matrix like "det" or
% "trace" of P is minimized to compute w. This is offered by the function
% "fusecovint". However. in this case, the w are chosen by minimizing the
% determinants of "positional" covariances only.
n = size(x1,1);
idx = [1 2];
detP1pos = det(P1(idx,idx));
detP2pos = det(P2(idx,idx));
w1 = detP2pos/(detP1pos + detP2pos);
w2 = detP1pos/(detP1pos + detP2pos);
I = eye(n);

P1inv = I/P1;
P2inv = I/P2;

Pinv = w1*P1inv + w2*P2inv;
P = I/Pinv;

x = P*(w1*P1inv*x1 + w2*P2inv*x2);

end
```

References

[1] Lang, Alex H., et al. "PointPillars: Fast encoders for object detection from point clouds." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2019.

[2] Zhou, Yin, and Oncel Tuzel. "Voxelnet: End-to-end learning for point cloud based 3d object detection." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018.

[3] Yang, Bin, Wenjie Luo, and Raquel Urtasun. "Pixor: Real-time 3d object detection from point clouds." Proceedings of the IEEE conference on Computer Vision and Pattern Recognition. 2018.

See Also

`drivingScenario` | `lidarPointCloudGenerator` | `radarDetectionGenerator` | `trackGOSPAMetric`

More About

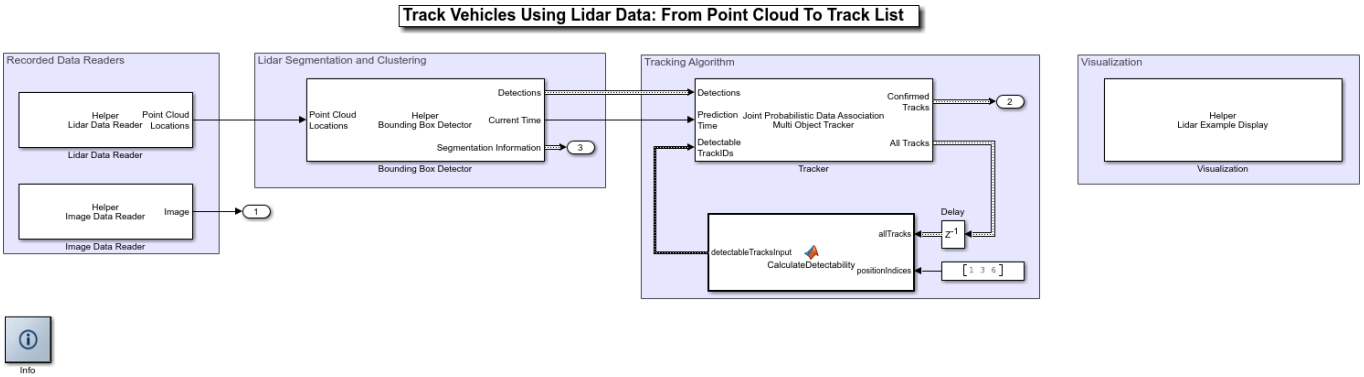
- “Extended Object Tracking of Highway Vehicles with Radar and Camera” on page 7-234
- “Track Vehicles Using Lidar: From Point Cloud to Track List” on page 7-177
- “Detect, Classify, and Track Vehicles Using Lidar” (Lidar Toolbox)

Track Vehicles Using Lidar Data in Simulink

This example shows you how to track vehicles using measurements from a lidar sensor mounted on top of an ego vehicle. Due to high resolution capabilities of the lidar sensor, each scan from the sensor contains a large number of points, commonly known as a point cloud. The example illustrates the workflow in Simulink for processing the point cloud and tracking the objects. The lidar data used in this example is recorded from a highway driving scenario. You use the recorded data to track vehicles with a joint probabilistic data association (JPDA) tracker and an interacting multiple model (IMM) approach. The example closely follows the "Track Vehicles Using Lidar: From Point Cloud to Track List" (Sensor Fusion and Tracking Toolbox) MATLAB® example.

Overview of the Model

```
load_system('TrackVehiclesSimulinkExample');
set_param('TrackVehiclesSimulinkExample','SimulationCommand','update');
open_system('TrackVehiclesSimulinkExample');
```

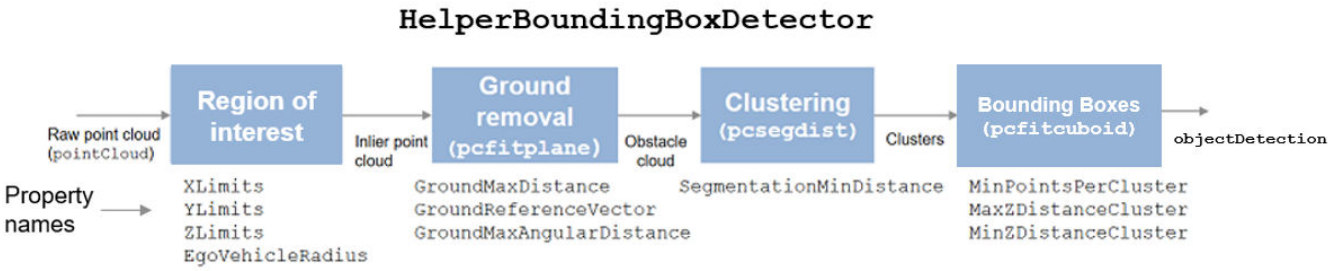


Lidar and Image Data Reader

The Lidar Data Reader and Image Data Reader blocks are implemented using a MATLAB System (Simulink) block. The code for the blocks is defined by helper classes, `HelperLidarDataReader` and `HelperImageDataReader` respectively. The image and lidar data readers read the recorded data from the MAT files and output the reference image and the locations of points in the point cloud respectively.

Bounding Box Detector

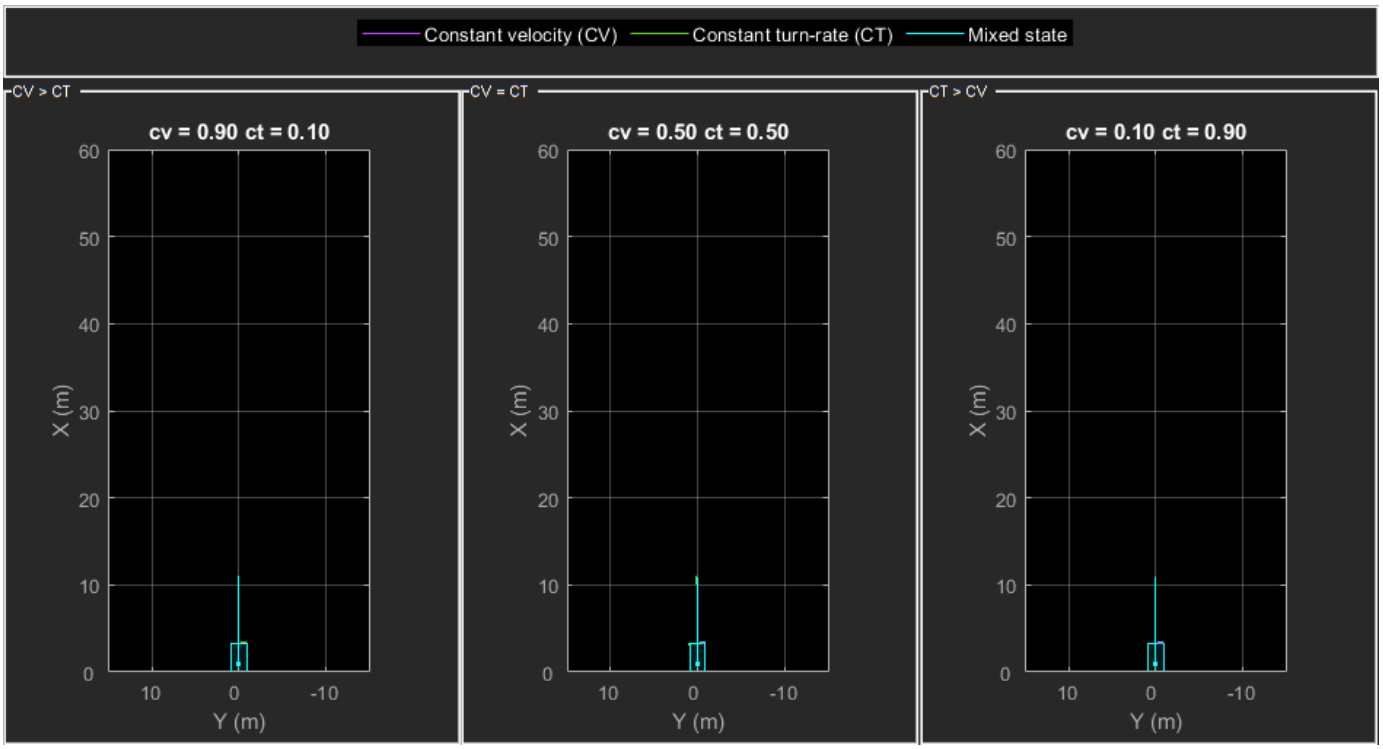
As described earlier, the raw data from sensor contains a large number of points. This raw data must be preprocessed to extract objects of interest, such as cars, cyclists, and pedestrian. The preprocessing is done using the Bounding Box Detector block. The Bounding Box Detector is also implemented as a MATLAB System™ block defined by a helper class, `HelperBoundingBoxDetectorBlock`. It accepts the point cloud locations as an input and outputs bounding box detections corresponding to obstacles. The diagram shows the processes involved in the bounding box detector model and the Computer Vision Toolbox™ functions used to implement each process. It also shows the parameters of the block that control each process.



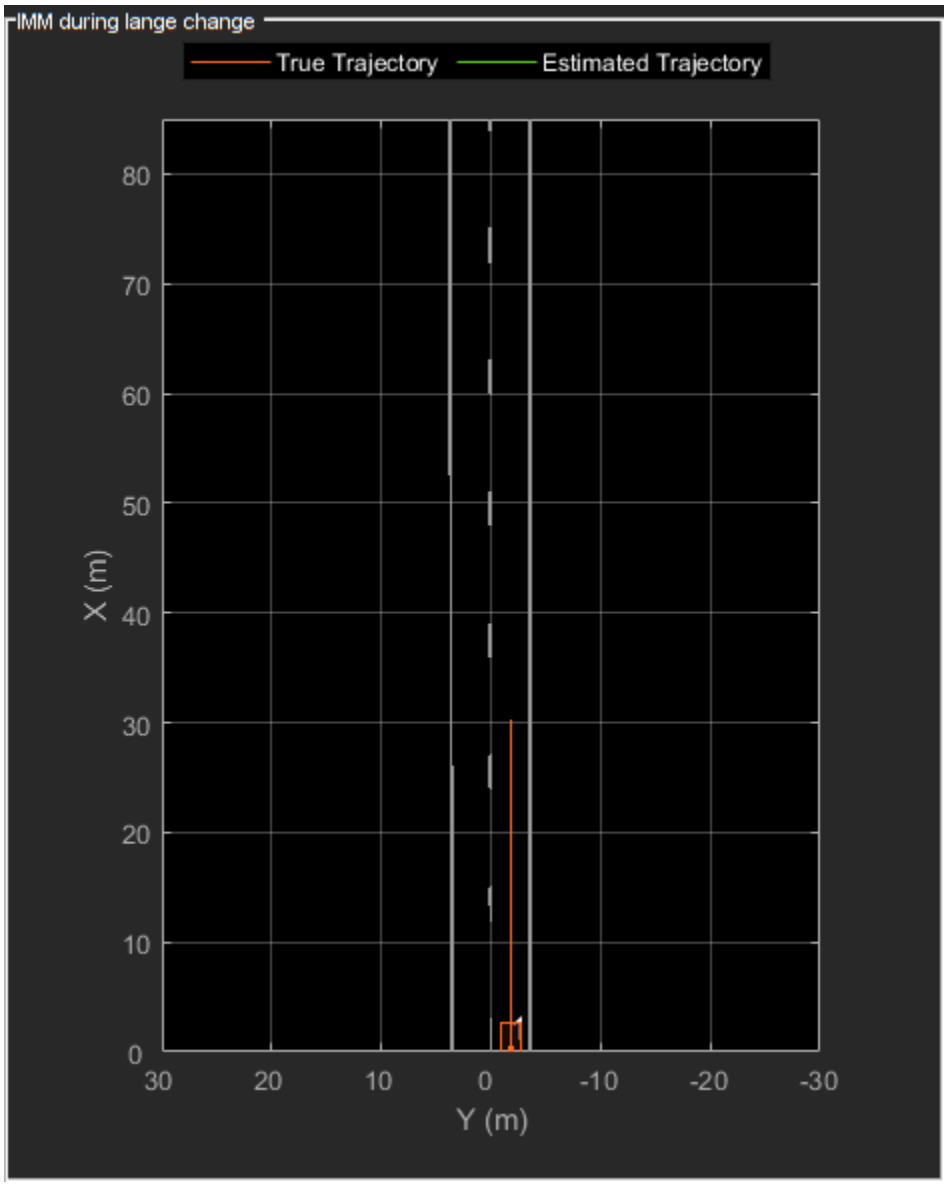
The block outputs the detections and segmentation information as a `Simulink.Bus` (Simulink) object named `detectionBus` and `segmentationBus`. These buses are created in the base workspace using helper function `helperCreateDetectorBus` specified in the `PreLoadFcn` callback. See "Model Callbacks" (Simulink) for more information about callback functions.

Tracking algorithm

The tracking algorithm is implemented using the joint probabilistic data association (JPDA) tracker, which uses an interacting multiple model (IMM) approach to track targets. The IMM filter is implemented by the `helperInitIMMFilter`, which is specified as the "Filter initialization function" parameter of the block. In this example, the IMM filter is configured to use two models, a constant velocity cuboid model and a constant turn-rate cuboid model. The models define the dimensions of the cuboid as constants during state-transition and their estimates evolve in time during correction stages of the filter. The animation below shows the effect of mixing the constant velocity and constant turn-rate models with different probabilities during prediction stages of the filter.



The IMM filter automatically computes the probability of each model when the filter is corrected with detections. The animation below shows the estimated trajectory and the probability of models during a lane change event.



For a detailed description of the state transition and measurement models, refer to the "Target State and Sensor Measurement Model" section of the MATLAB example.

The tracker block selects the check box "Enable all tracks output" and "Enable detectable track IDs input" to output all tracks from the tracker and calculate their detection probability as a function of their state.

Calculate Detectability

The Calculate Detectability block is implemented using a MATLAB Function (Simulink) block. The block calculates the Detectable TrackIDs input for the tracker and outputs it as an array with 2

columns. The first column represents the TrackIDs of the tracks and the second column specifies their probability of detection by the sensor and bounding box detector.

Visualization

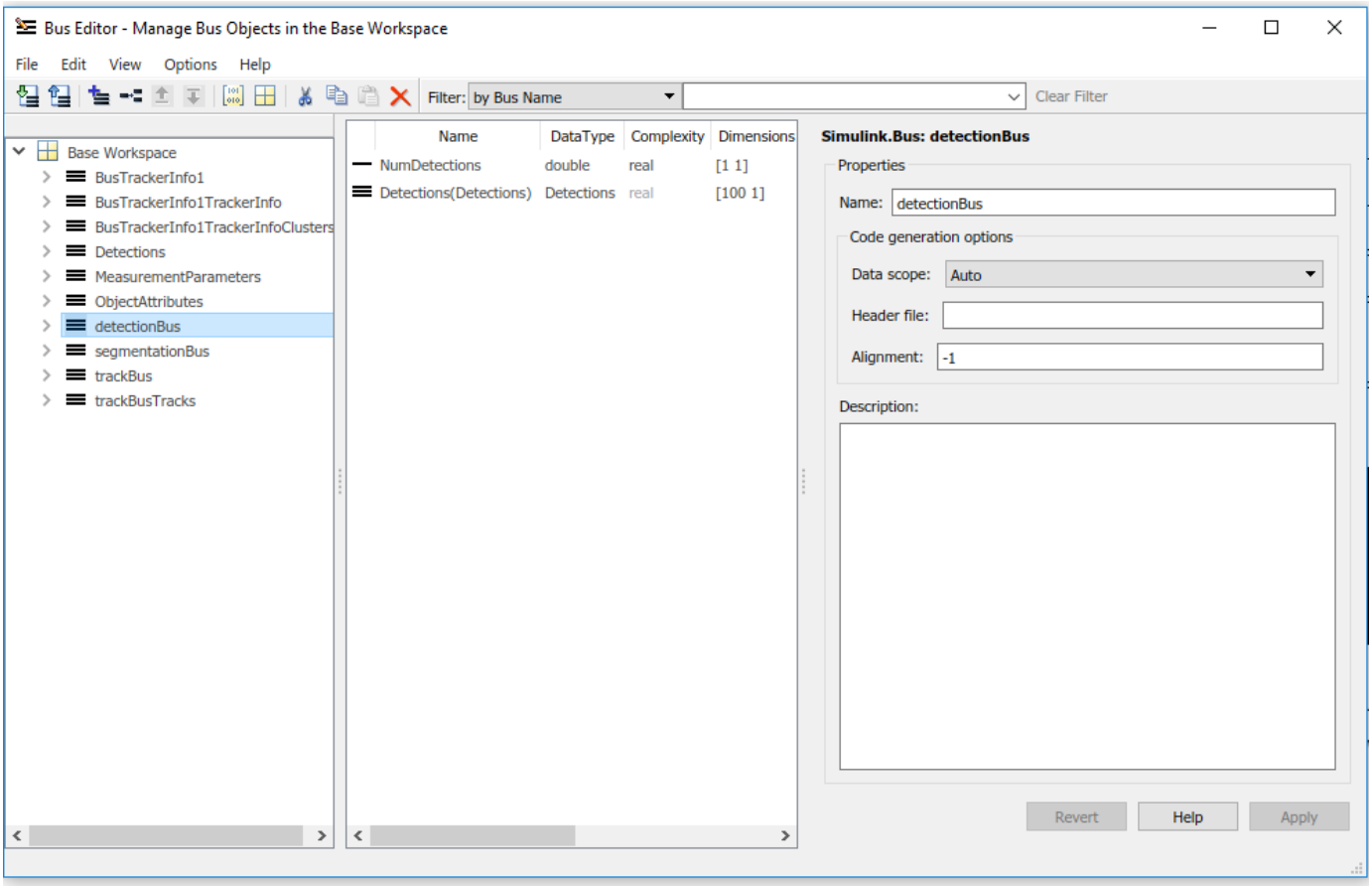
The Visualization block is also implemented using the MATLAB System block and is defined using `HelperLidarExampleDisplayBlock`. The block uses `RunTimeObject` parameter of the blocks to display their outputs. See “Access Block Data During Simulation” (Simulink) for further information on how to access block outputs during simulation.

Detections and Tracks Bus Objects

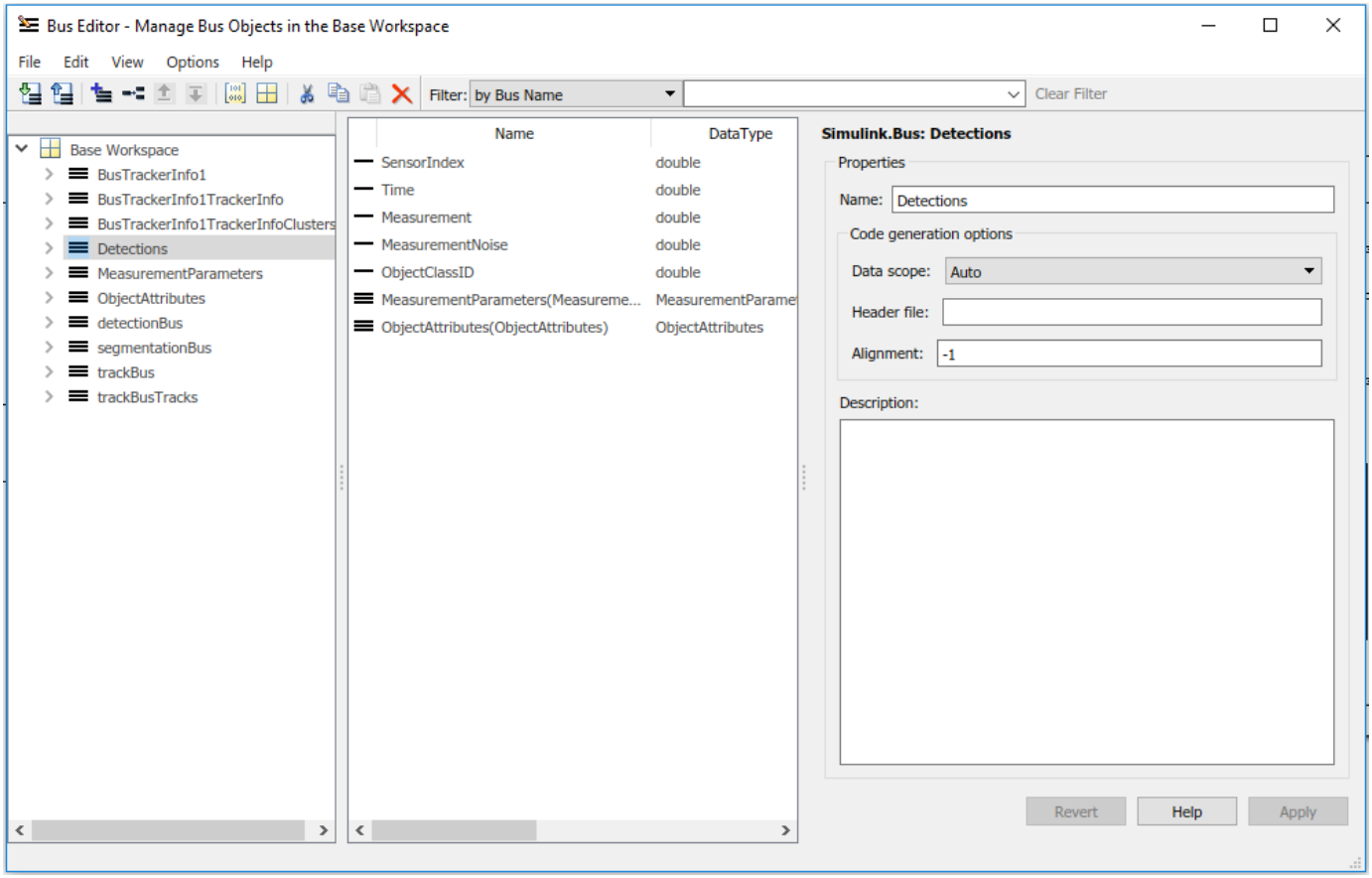
As described earlier, the inputs and outputs of different blocks are bus objects. You can visualize the structure of each bus using the Bus Editor (Simulink). The following images show the structure of the bus for detections and tracks.

Detections

The `detectionBus` outputs a nested bus object with 2 elements, `NumDetections` and `Detections`.

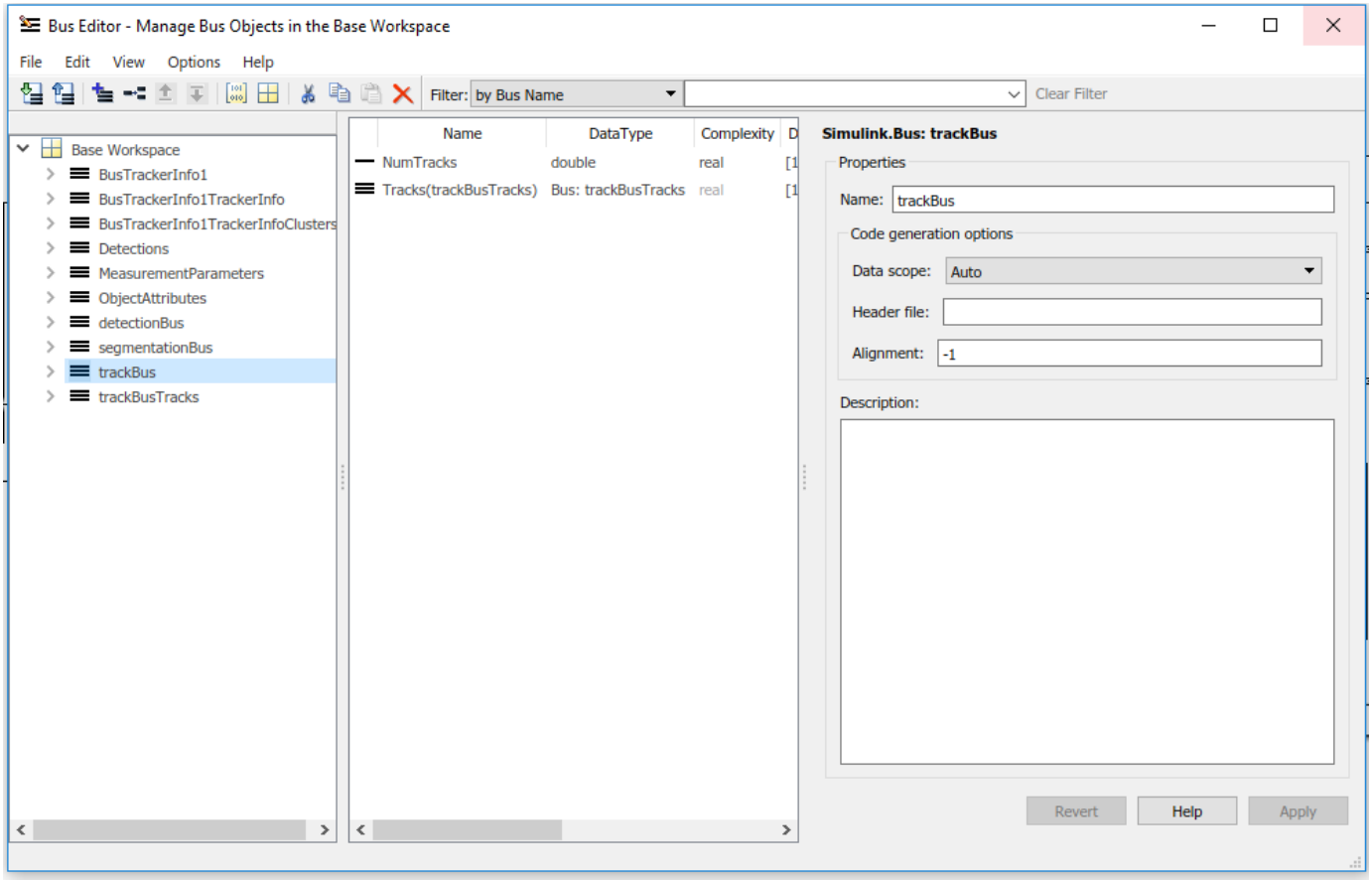


The first element, `NumDetections`, represents the number of detections. The second element `Detections` is a bus object of a fixed size representing all detections. The first `NumDetections` elements of the bus object represent the current set of detections. Notice that the structure of the bus is similar to the `objectDetection` (Sensor Fusion and Tracking Toolbox) class.

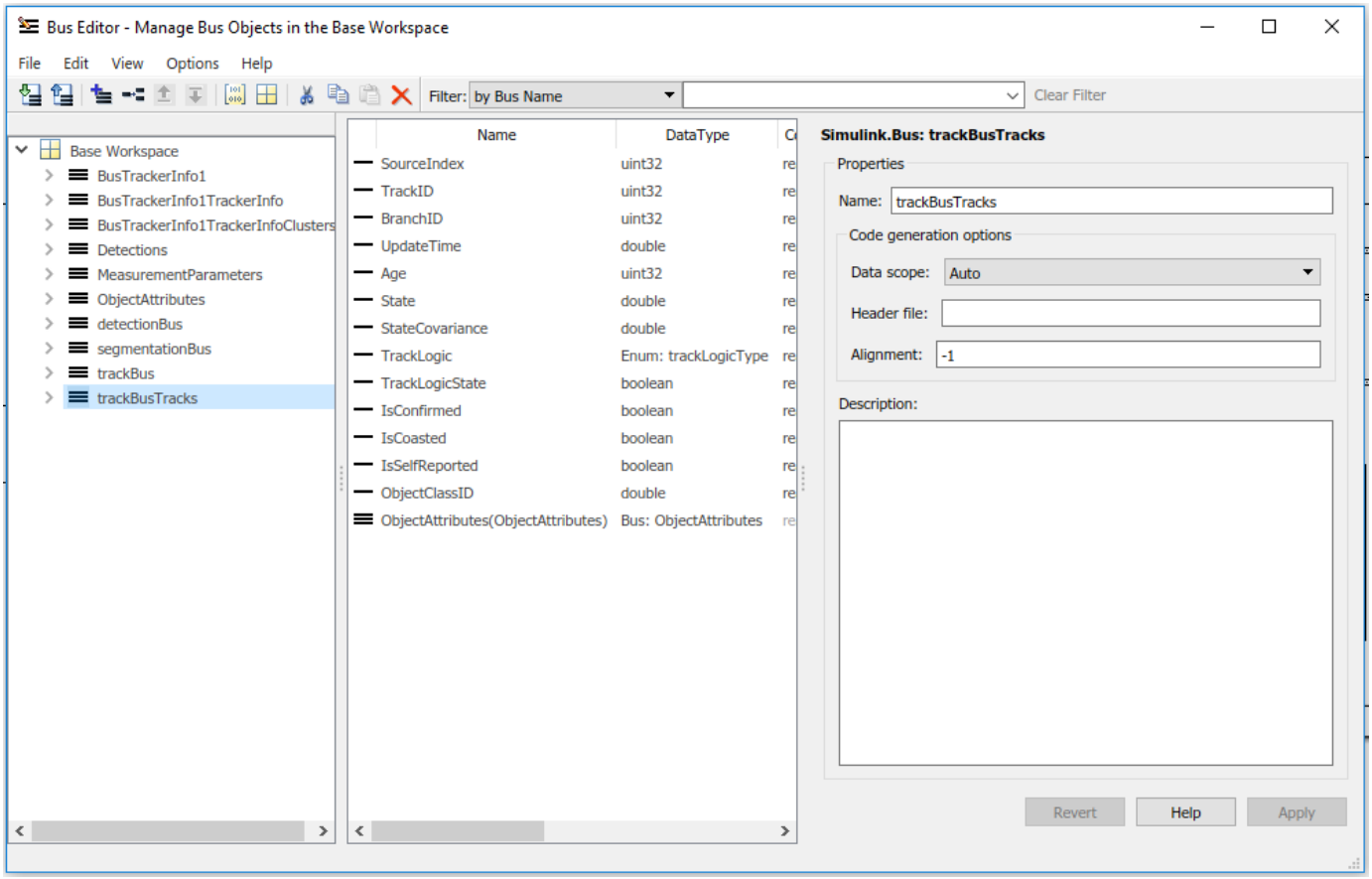


Tracks

The track bus is similar to the detections bus. It is a nested bus, where `NumTracks` defines the number of tracks in the bus and `Tracks` define a fixed size of tracks. The size of the tracks is governed by the block parameter "Maximum number of tracks".

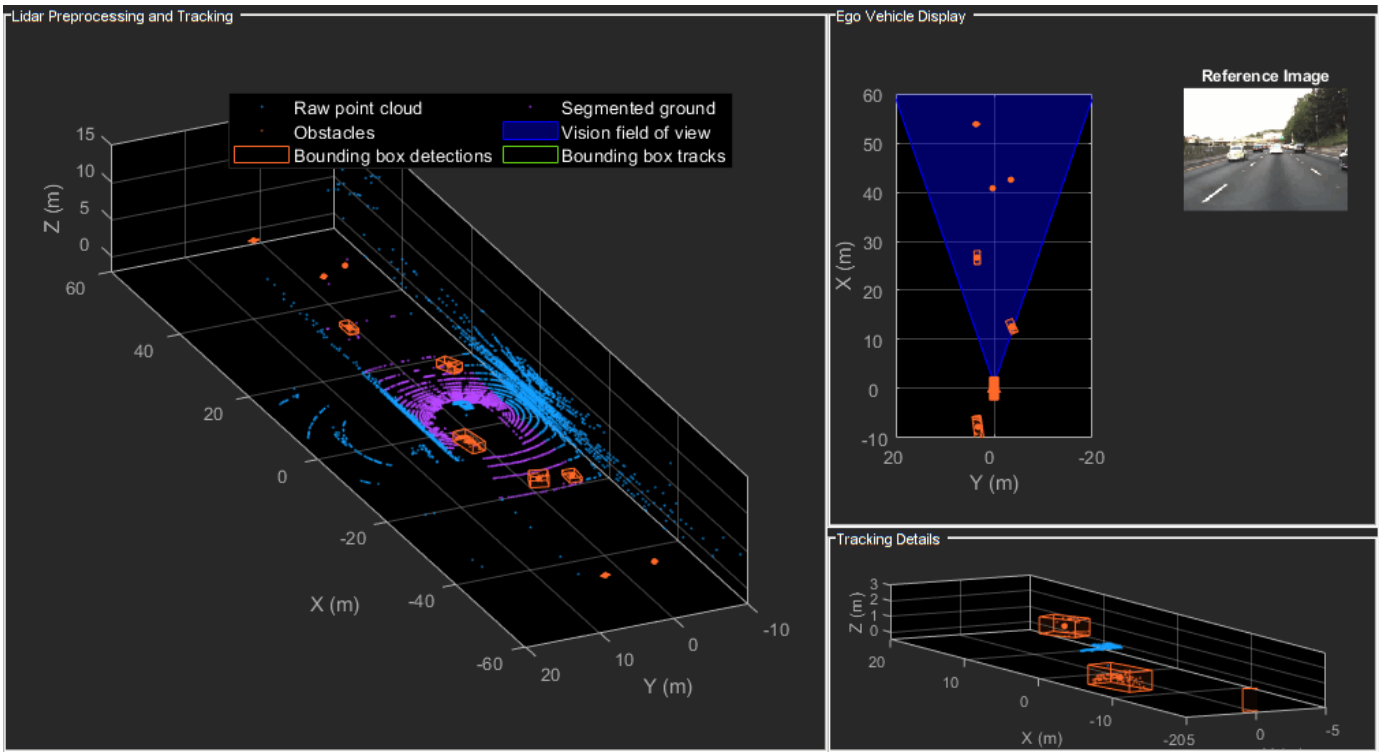


The second element `Tracks` is a bus object defined by `trackBusTracks`. This bus is automatically created by the tracker block by using the bus name specified as the prefix. Notice that the structure of the bus is similar to the `objectTrack` (Sensor Fusion and Tracking Toolbox) class.



Results

The detector and tracker algorithm is configured exactly as the “Track Vehicles Using Lidar: From Point Cloud to Track List” (Sensor Fusion and Tracking Toolbox) MATLAB example. After running the model, you can visualize the results on the figure. The animation below shows the results from time 0 to 4 seconds. The tracks are represented by green bounding boxes. The bounding box detections are represented by orange bounding boxes. The detections also have orange points inside them, representing the point cloud segmented as obstacles. The segmented ground is shown in purple. The cropped or discarded point cloud is shown in blue. Notice that the tracked objects are able to maintain their shape and kinematic center by positioning the detections onto visible portions of the vehicles. This illustrates the offset and shrinkage effect modeled in the measurement functions.




```
close_system('TrackVehiclesSimulinkExample');
```

Summary

This example showed how to use a JPDA tracker with an IMM filter to track objects using a lidar sensor. You learned how a raw point cloud can be preprocessed to generate detections for conventional trackers, which assume one detection per object per sensor scan. You also learned how to use a cuboid model to describe the extended objects being tracked by the JPDA tracker.

See Also

Joint Probabilistic Data Association Multi Object Tracker

More About

- “Track Vehicles Using Lidar: From Point Cloud to Track List” on page 7-177
- “Detect, Classify, and Track Vehicles Using Lidar” (Lidar Toolbox)

Grid-based Tracking in Urban Environments Using Multiple Lidars

This example shows how to track moving objects with multiple lidars using a grid-based tracker. A grid-based tracker enables early fusion of data from high-resolution sensors such as radars and lidars to create a global object list.

Introduction

Most multi-object tracking approaches represent the environment as a set of discrete and unknown number of objects. The job of the tracker is to estimate the number of objects and their corresponding states, such as position, velocity, and dimensions, using the sensor measurements. With high-resolution sensors such as radar or lidar, the tracking algorithm can be configured using point-object trackers or extended object trackers.

Point-Object Trackers

Point-object trackers assume that each object may give rise to at most one detection per sensor. Therefore, when using point-target trackers for tracking extended objects, features like bounding box detections are first extracted from the sensor measurements at the object-level. These object-level features then get fused with object-level hypothesis from the tracker. A poor object-level extraction algorithm at the sensor level (such as imperfect clustering) thus greatly impacts the performance of the tracker. For an example of this workflow, refer to “Track Vehicles Using Lidar: From Point Cloud to Track List” on page 7-177 example.

Extended Object Trackers

On the other hand, extended object trackers process the detections without extracting object-level hypothesis at the sensor level. Extended object trackers associate sensor measurements directly with the object-level hypothesis maintained by tracker. To do this, a class of algorithms typically requires complex measurement models of the object extents specific to each sensor modality. For example, refer to “Extended Object Tracking with Lidar for Airport Ground Surveillance” (Sensor Fusion and Tracking Toolbox) and “Extended Object Tracking of Highway Vehicles with Radar and Camera” on page 7-234 to learn how to configure a multi-object PHD tracker for lidar and radar respectively.

A grid-based tracker can be considered as a type of extended object tracking algorithm which uses a dynamic occupancy grid map as an intermediate representation of the environment. In a dynamic occupancy grid map, the environment is discretized using a set of 2-D grid cells. The dynamic map represents the occupancy as well as kinematics of the space represented by a grid cell. Using the dynamic map estimate and further classification of cells as static and dynamic serves as a preprocessing step to filter out measurements from static objects and to reduce the computational complexity.

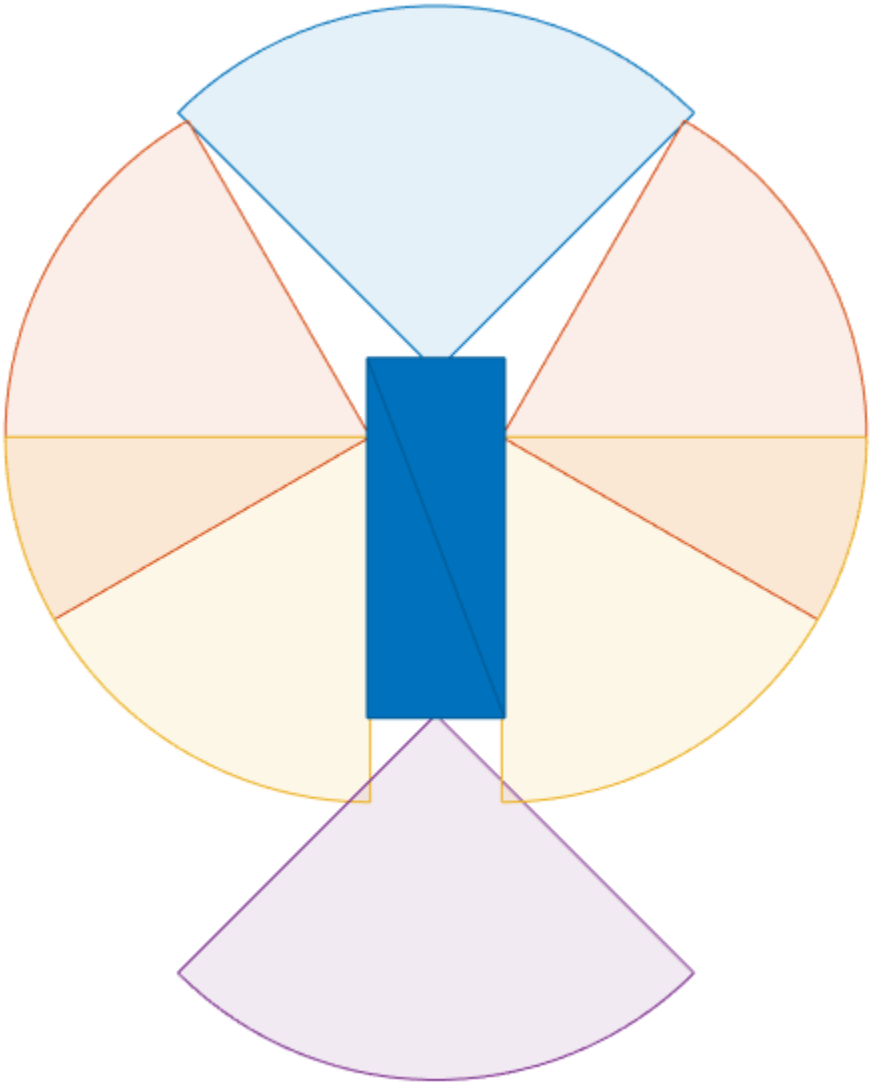
In this example, you use the `trackerGridRFS` (Sensor Fusion and Tracking Toolbox) System object™ to configure the grid-based tracker. This tracker uses the Random Finite Set (RFS) formulation with Dempster-Shafer approximation [1] to estimate the dynamic map. Further, it uses a nearest neighbor cell-to-track association [2] scheme to track dynamic objects in the scene. To initialize new tracks, the tracker uses the DBSCAN algorithm to cluster unassigned dynamic grid cells.

Set Up Scenario and Lidar Sensor Models

The scenario used in this example was created using the Driving Scenario Designer app and was exported to a MATLAB® function. This MATLAB function was wrapped as a helper function

`helperCreateMultiLidarDrivingScenario`. The scenario represents an urban intersection scene and contains a variety of objects that include like pedestrians, bicyclists, cars, and trucks.

The ego vehicle is equipped with 6 homogeneous lidars, each with a horizontal field of view of 90 degrees and a vertical field of view of 40 degrees. The lidars are simulated using the `lidarPointCloudGenerator` System object. Each lidar has 32 elevation channels and has a resolution of 0.16 degrees in azimuth. Under this configuration, each lidar sensor outputs approximately 18,000 points per scan. The configuration of each sensor is shown here.



```
% For reproducible results
rng(2020);

% Create scenario
[scenario, egoVehicle, lidars] = helperCreateMultiLidarDrivingScenario;
```

The scenario and the data from the different lidars can be visualized in the animation below. For brevity and to make the example easier to visualize, the lidar is configured to not return point cloud

from the ground by specifying the `HasRoadsInputPort` property as `false`. When using real data or if using simulated data from roads, the returns from ground and other environment must be removed using point cloud preprocessing. For more information, refer to the “Ground Plane and Obstacle Detection Using Lidar” on page 7-107 example.

Set Up Grid-Based Tracker

You define a grid-based tracker using `trackerGridRFS` to track dynamic objects in the scene. The first step of defining the tracker is setting up sensor configurations as `trackingSensorConfiguration` objects. The sensor configurations allow you to specify the mounting of each sensor with respect to the tracking coordinate frame. The sensor configurations also allow you to specify the detection limits - field of view and maximum range - of each sensor. In this example, you use the properties of the simulated lidar sensors to define these properties.

The utility function helperGetLidarConfig on page 7-0 uses the simulated lidar sensor model and returns its respective configuration. In this example, the targets are tracked in the global or world coordinate system by using the simulated pose of the vehicle. This information is typically obtained via an inertial navigation system. As the sensors move in the scenario system, their configuration must be updated each time by specifying the configurations as an input to the tracker.

```
% Store configurations of all sensor
sensorConfigs = cell(numel(lidars),1);

% Fill in sensor configurations
for i = 1:numel(sensorConfigs)
    sensorConfigs{i} = helperGetLidarConfig(lidars{i},egoVehicle);
```

```
end

% Create tracker. You can define the properties before using the tracker.
tracker = trackerGridRFS('SensorConfigurations',sensorConfigs,...
    'HasSensorConfigurationsInput',true);
```

The tracker uses a two-dimensional grid for the intermediate representation of the environment. The grid is defined by 3 attributes: its length, its width, and the resolution. The length and width describe the span of the grid in local X and local Y direction of the ego vehicle respectively. The resolution defines the number of cells per meter of the grid. In this example, you use a 120 m by 120 m grid with 2 cells per meter.

```
tracker.GridLength = 120; % meters
tracker.GridWidth = 120; % meters
tracker.GridResolution = 2; % 1/meters
```

In addition to defining the grid, you also define the relative position of the ego vehicle by specifying the origin of the grid (left corner) with respect to the origin of the ego vehicle. In this example, the ego vehicle is located at the center of the grid.

```
tracker.GridOriginInLocal = [-tracker.GridLength/2 -tracker.GridWidth/2];
```

The tracker uses particle-based methods to estimate the state of each grid cell and further classify them as dynamic or static. It uses a fixed number of persistent particles on the grid which defines the distribution of existing targets. It also uses a fixed number of particles to sample the distribution for newborn targets. These birth particles get sampled in different grid cells based on the probability of birth. Further, the velocity and other unknown states like turn-rate and acceleration (applicable when `MotionModel` of the tracker is not `constant-velocity`) of the particles is sampled uniformly using prior information supplied using prior limits. A resampling step assures that the number of particles on the grid remain constant.

```
tracker.NumParticles = 1e5; % Number of persistent particles
tracker.NumBirthParticles = 2e4; % Number of birth particles
tracker.VelocityLimits = [-15 15;-15 15]; % To sample velocity of birth particles (m/s)
tracker.BirthProbability = 0.025; % Probability of birth in each grid cell
tracker.ProcessNoise = 5*eye(2); % Process noise of particles for prediction as variance of [ax;a
```

The tracker uses the Dempster-Shafer approach to define the occupancy of each cell. The dynamic grid estimates the belief mass for occupancy and free state of the grid. During prediction, the occupancy belief mass of the grid cell updates due to prediction of the particle distribution. The `DeathRate` controls the probability of survival (P_s) of particles and results in a decay of occupancy belief mass during prediction. As the free belief mass is not linked to the particles, the free belief mass decays using a pre-specified, constant discount factor. This discount factor specifies the probability that free regions remain free during prediction.

```
tracker.DeathRate = 1e-3; % Per unit time. Translates to Ps = 0.9999 for 10 Hz
tracker.FreeSpaceDiscountFactor = 1e-2; % Per unit time. Translates to a discount factor of 0.63
```

After estimation of state of each grid cell, the tracker classifies each grid cell as static or dynamic by using its estimated velocity and associated uncertainty. Further, the tracker uses dynamic cells to extract object-level hypothesis using the following technique:

Each dynamic grid cell is considered for assignment with existing tracks. A dynamic grid cell is assigned to its nearest track if the negative log-likelihood between a grid cell and a track falls below an assignment threshold. A dynamic grid cell outside the assignment threshold is considered unassigned. The tracker uses unassigned grid cells at each step to initiate new tracks. Because

multiple unassigned grid cells can belong to the same object track, a DBSCAN clustering algorithm is used to assist in this step. Because there are false positives while classifying the cells as static or dynamic, the tracker filters those false alarms in two ways. First, only unassigned cells which form clusters with more than a specified number of points (`MinNumPointsPerCluster`) can create new tracks. Second, each track is initialized as a tentative track first and is only confirmed if its detected M out of N times.

```
tracker.AssignmentThreshold = 8; % Maximum distance or negative log-likelihood between cell and t
tracker.MinNumCellsPerCluster = 6; % Minimum number of grid cells per cluster for creating new tr
tracker.ClusteringThreshold = 1; % Minimum Euclidean distance between two cells for clustering
tracker.ConfirmationThreshold = [3 4]; % Threshold to confirm tracks
tracker.DeletionThreshold = [4 4]; % Threshold to delete confirmed tracks
```

You can also accelerate simulation by performing the dynamic map estimation on GPU by specifying the `UseGPU` property of the tracker.

```
tracker.UseGPU = false;
```

Visualization

The visualization used for this example is defined using a helper class, `helperGridTrackingDisplay`, attached with this example. The visualization contains three parts.

- Ground truth - Front View: This panel shows the front-view of the ground truth using a chase plot from the ego vehicle. To emphasize dynamic actors in the scene, the static objects are shown in gray.
- Lidar Views: These panels show the point cloud returns from each sensor.
- Grid-based tracker**:** This panel shows the grid-based tracker outputs. The tracks are shown as boxes, each annotated by their identity. The tracks are overlaid on the dynamic grid map. The colors of the dynamic grid cells are defined according to the color wheel, which represents the direction of motion of in the scenario frame. The static grid cells are represented using a grayscale according to their occupancy. The degree of grayness denotes the probability of the space occupied by the grid cell as free. The positions of the tracks are shown in the ego vehicle coordinate system, while the velocity vector corresponds to the velocity of the track in the scenario frame.

```
display = helperGridTrackingDisplay;
```

Run Scenario and Track Dynamic Objects

Next, run the scenario, simulate lidar sensor data from each lidar sensor, and process the data using the grid-based tracker.

```
% Initialize pointCloud ouputs from each sensor
ptClouds = cell(numel(lidars),1);
sensorConfigs = cell(numel(lidars),1);

while advance(scenario)
    % Current simulation time
    time = scenario.SimulationTime;

    % Poses of objects with respect to ego vehicle
    tgtPoses = targetPoses(egoVehicle);

    % Simulate point cloud from each sensor
    for i = 1:numel(lidars)
```

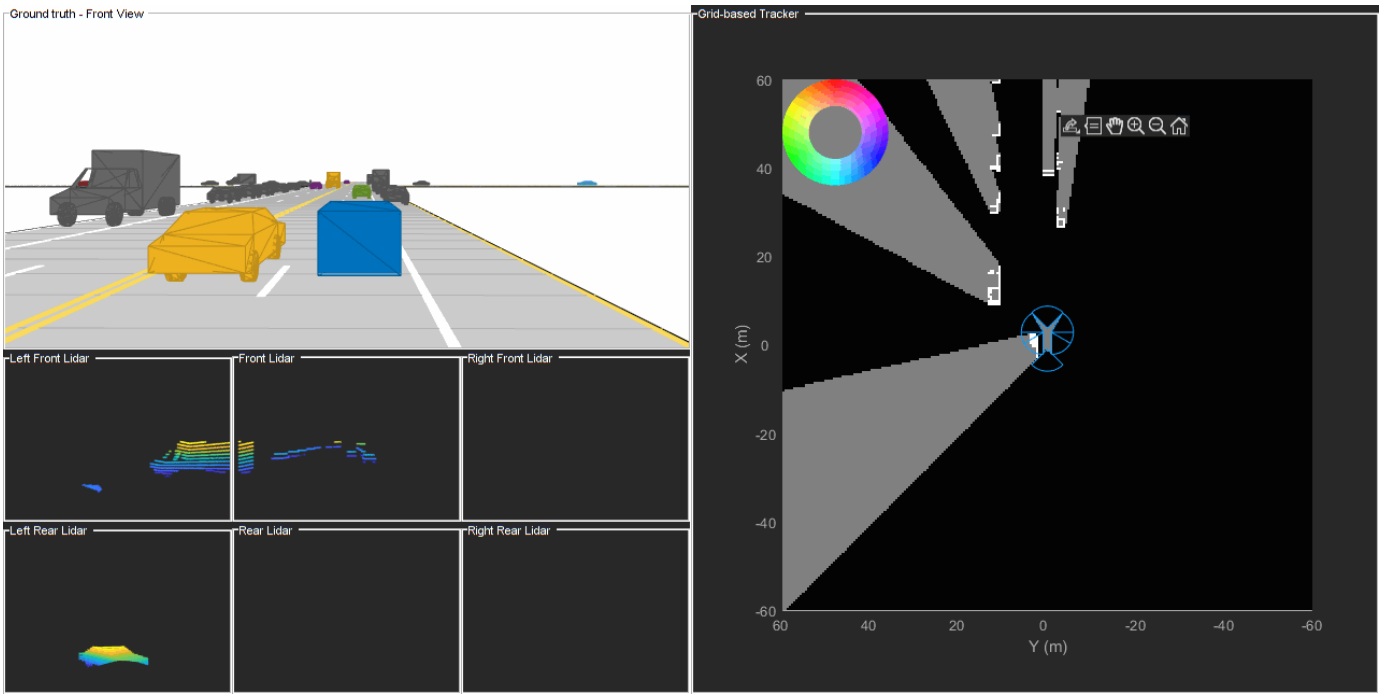
```
            [ptClouds{i}, isValidTime] = step(lidars{i},tgtPoses,time);
            sensorConfigs{i} = helperGetLidarConfig(lidars{i},egoVehicle);
        end

        % Pack point clouds as sensor data format required by the tracker
        sensorData = packAsSensorData(ptClouds,sensorConfigs,time);

        % Call the tracker
        tracks = tracker(sensorData,sensorConfigs,time);

        % Update the display
        display(scenario, egoVehicle, lidars, ptClouds, tracker, tracks);
        drawnow;
    end
```

Results

Next, analyze the performance of the tracker using the visualization used in this example.

The grid-based tracker uses the dynamic cells from the estimated grid map to extract object tracks. The animation below shows the results of the tracker in this scenario. The "Grid-based tracker" panel shows the estimated dynamic map as well as the estimated tracks of the objects. It also shows the configuration of the sensors mounted on the ego vehicle as blue circular sectors. Notice that the area encapsulated by these sensors is estimated as "gray" in the dynamic map, representing that this area is not observed by any of the sensors. This patch also serves as an indication of ego-vehicle's position on the dynamic grid.

Notice that the tracks are extracted only from the dynamic grid cells and hence the tracker is able to filter out static objects. Also notice that after a vehicle enters the grid region, its track establishment takes few time steps. This is due to two main reasons. First, there is an establishment delay in classification of the cell as dynamic. Second, the confirmation threshold for the object takes some steps to establish a track as a confirmed object.

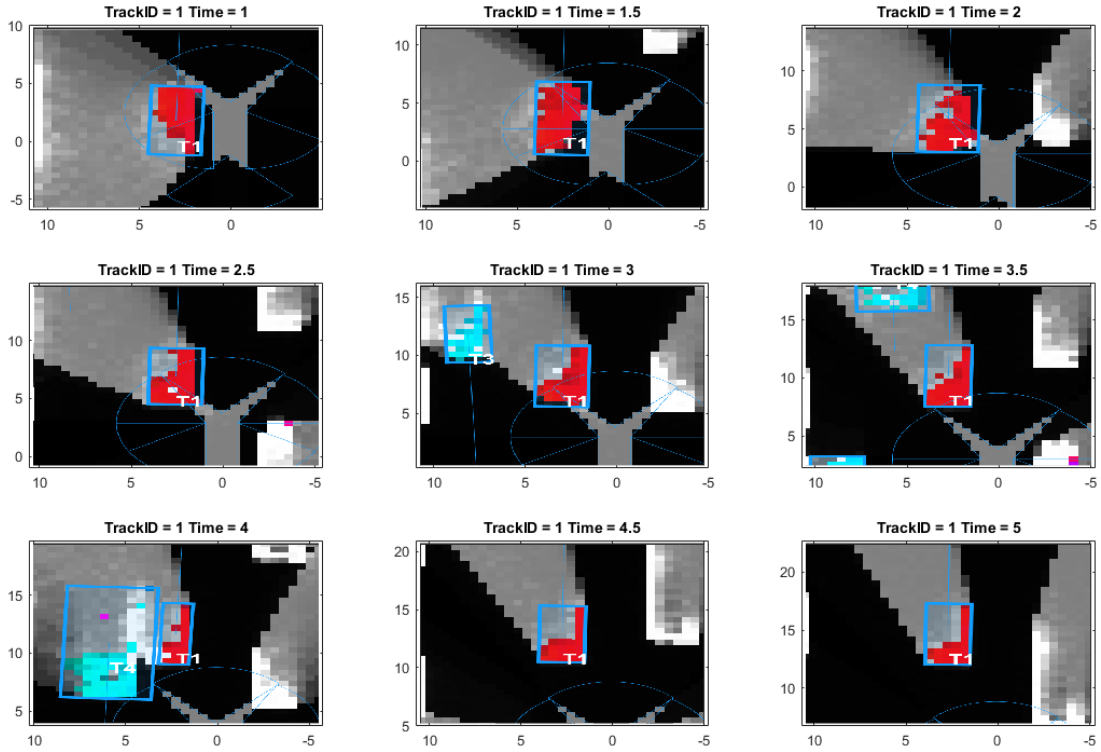


Next, you look at the history of a few tracks to understand how the state of a track gets affected by the estimation of the dynamic grid.

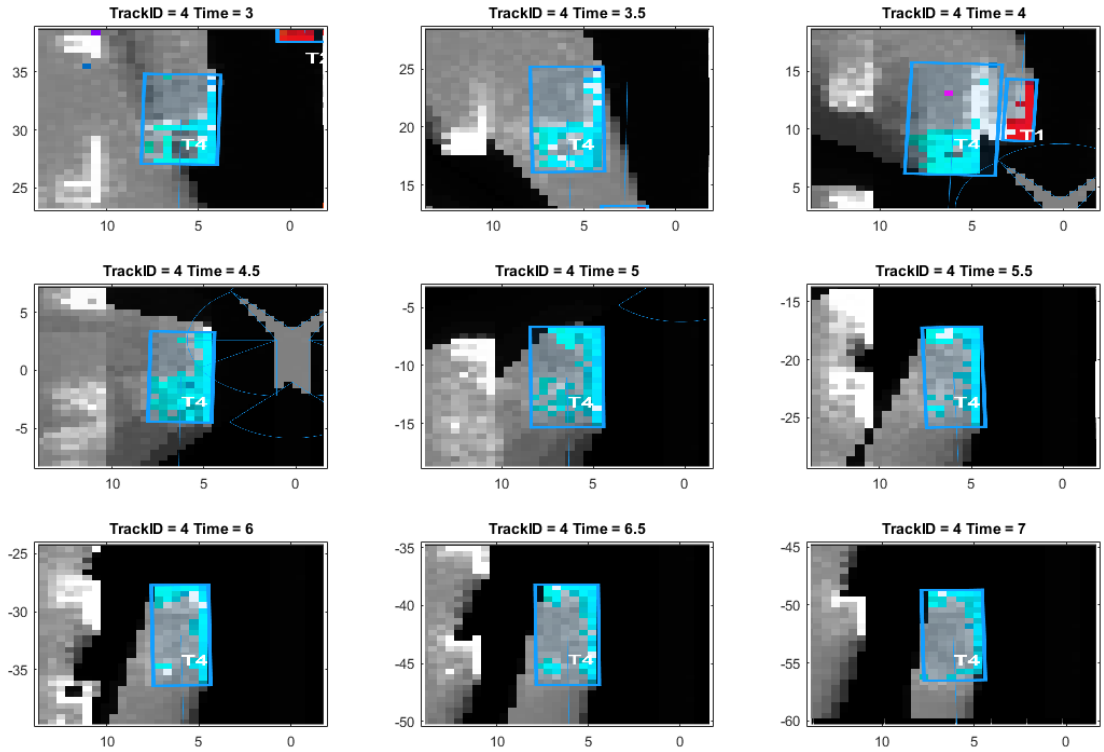
Longitudinally Moving Tracks

The following snapshots show the history for the track denoted by T1. The T1 track represents the yellow car that passes the ego vehicle on the left during the first few seconds of the simulation. Notice that the grid cells occupied by this track are colored in red, indicating their motion in the positive X direction. The track obtains the track's velocity and heading information using the velocity distribution of the assigned grid cells. It also obtains its length, width, and orientation using the spatial distribution of the assigned grid cells. The default `TrackUpdateFcn` of the `trackerGridRFS` extracts new length, width, and orientation information from the spatial distribution of associated grid cells at every step. This effect can be seen in the snapshots below, where the length and width of the track adjusts according to the bounding box of the associated grid cells. An additional filtering scheme can be added using the predicted length, width, and orientation of the track by using a custom `TrackUpdateFcn`.

```
% Show snapshots for TrackID = 1. Also shows close tracks like T3 and T4
% representing car and truck moving in the opposite direction.
showSnapshots(display.GridView,1);
```




```
showSnapshots(display.GridView,4);
```



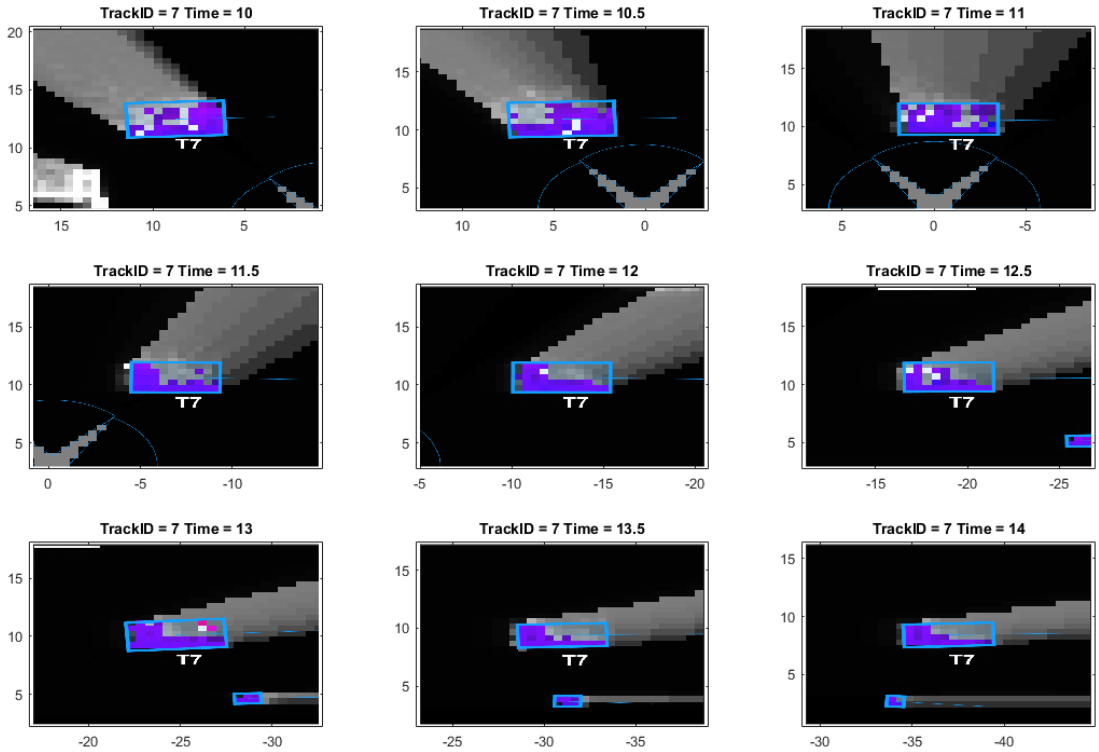
Next, take a closer look at the history of T4. The T4 track represents the truck moving in the opposite direction of the ego vehicle. Notice that the grid cells representing this track are colored in blue, representing the estimated motion direction of the grid cell. Also, notice that there are grid cells in the track that are misclassified by the tracker as static (white color). These misclassified grid cells often occur when sensors report previously occluded regions of an object, because the tracker has an establishment delay to classify these cells property.

Notice that at time = 4, when the truck and the vehicle came close to each other, the grid cells maintained their respective color, representing a stark difference between their estimated velocity directions. This also results in the correct data association between grid cells and predicted tracks of T1 and T4, which helps the tracker to resolve them as separate objects.

Laterally Moving Tracks

The following snapshots represent the track denoted by T7. This track represents the vehicle moving in the lateral direction, when the ego vehicle stops at the intersection. Notice that the grid cells of this track are colored in purple, representing the direction of motion in negative Y direction. Similar to other tracks, the track maintains its length and width using the spatial distribution of the assigned grid cells.

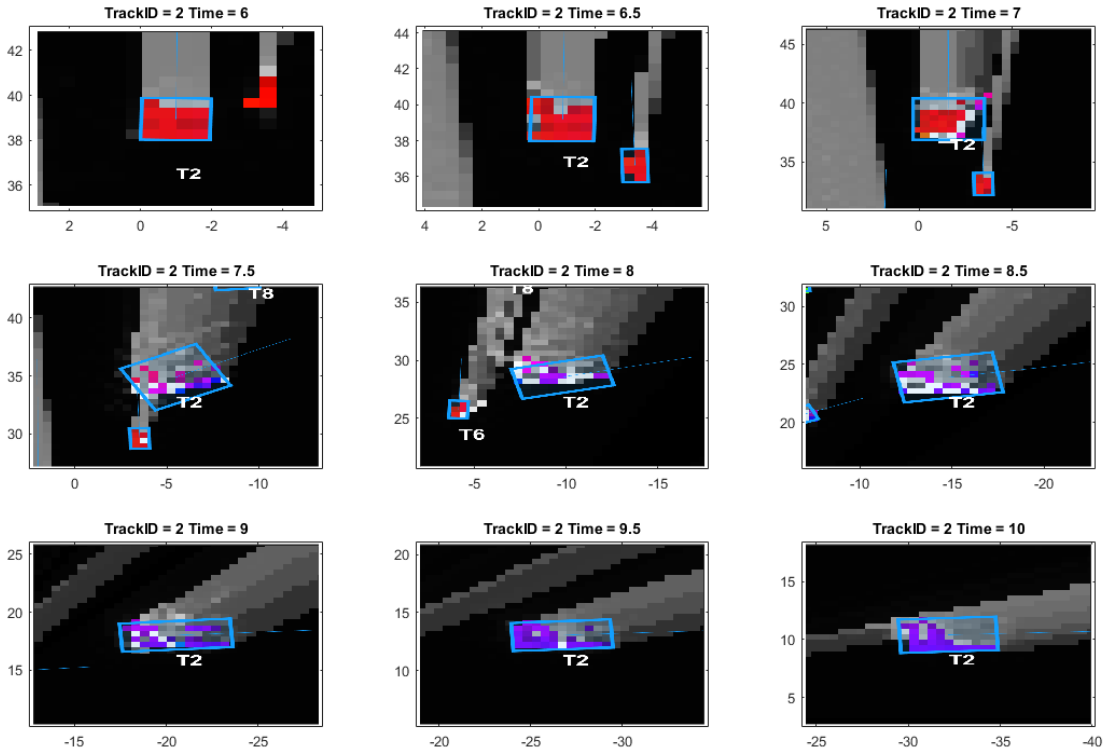
```
showSnapshots(display.GridView,7);
```



Tracks Changing Direction

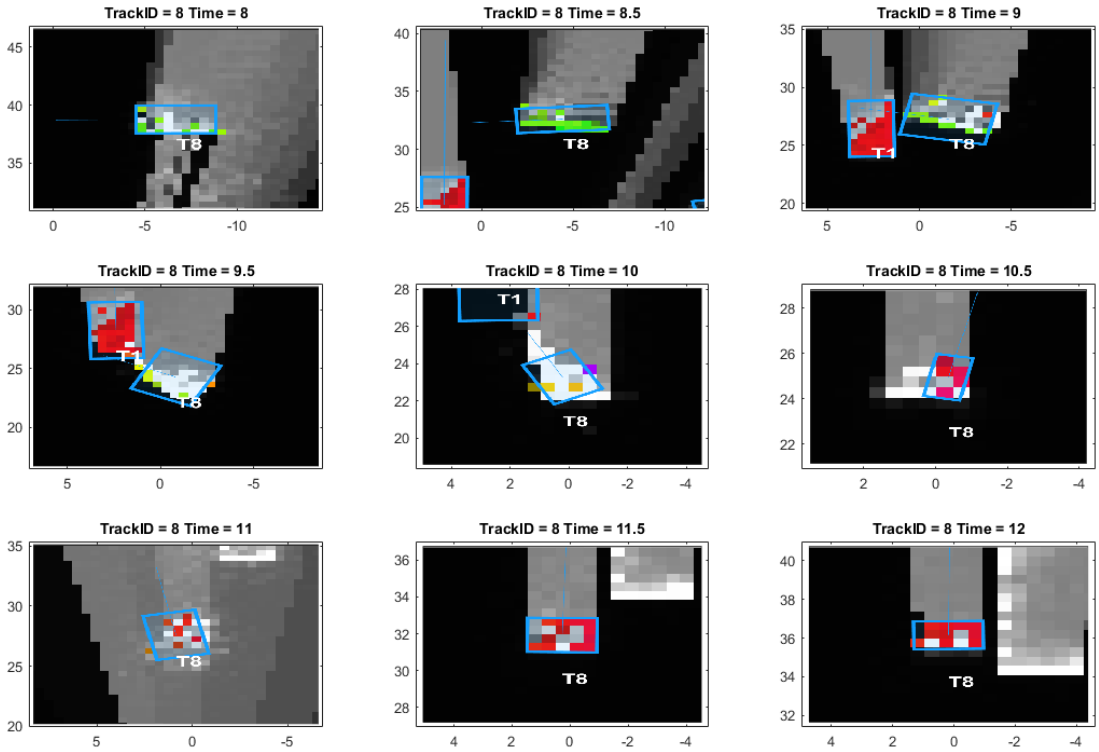
In this example, you used a "constant-velocity" model with the tracker. This motion model assumes that the targets move at a constant velocity, meaning constant speed and direction. However, in urban scenes, this assumption is usually not accurate. To compensate for the unknown acceleration of the objects, a process noise is specified on the tracker. The following snapshots show the history of track T2. This track represents the vehicle directly in front of the ego vehicle. Notice in the ground truth that this vehicle turns right at the intersection.

```
showSnapshots(display.GridView, 2);
```



Notice that the color of the grid cells associated with this track changes from red to purple. Also, the transition of colors results in a few misclassified cells, which can result in a poor estimate of length and width of the vehicle. The ability of the tracker to maintain the track on this vehicle is due to a coupled effect of three main reasons. First, the tracker allows to specify an assignment threshold. Even if the predicted track does not align with the dynamic grid cells, it can associate with them up to a certain threshold. Second, to create a new track from grid cells that remain outside the threshold requires meeting the minimum number of cells criteria. Third, the tracker has a deletion threshold, which allows a track to be coasted for a few steps before deleting it. If the classification of grid cells is very poor during the turn, the track can survive a few steps and can get re-associated with the grid cells. Note that misclassified grid cells are far more observable with Track T8, as shown below in its history. The T8 track represents the light blue car traveling in the positive Y direction before taking at right turn at the intersection. This vehicle was partially occluded before the turn and had another closely traveling vehicle while making the turn.

```
showSnapshots(display.GridView,8);
```



Summary

In this example, you learned the basics of a grid-based tracker and how it can be used to track dynamic objects in a complex urban driving environment. You also learned how to configure the tracker to track object using point clouds from multiple lidar sensors.

Supporting Functions

```
function sensorData = packAsSensorData(ptCloud, configs, time)
%The lidar simulation returns output as pointCloud object. The Location
%property of the point cloud is used to extract x,y and z locations of
%returns and pack them as structure with information required by a tracker.

sensorData = struct('SensorIndex',{},...
    'Time', {},...
    'Measurement', {},...
    'MeasurementParameters', {});

for i = 1:numel(ptCloud)
    % This sensor's cloud
    thisPtCloud = ptCloud{i};

    % Allows mapping between data and configurations without forcing an
    % ordered input and requiring configuration input for static sensors.
    sensorData(i).SensorIndex = configs{i}.SensorIndex;

    % Current time
```

```
    sensorData(i).Time = time;

    % Measurement as 3-by-N defininng locations of points
    sensorData(i).Measurement = reshape(thisPtCloud.Location,[],3)';

    % Data is reported in sensor coordinate frame and hence measurement
    % parameters are same as sensor transform parameters.
    sensorData(i).MeasurementParameters = configs{i}.SensorTransformParameters;
end

end

function config = helperGetLidarConfig(lidar, ego)
% Define transformation from sensor to ego
senToEgo = struct('Frame',fusionCoordinateFrameType(1),...
    'OriginPosition',[lidar.SensorLocation(:);lidar.Height],...
    'Orientation',rotmat(quaternion([lidar.Yaw lidar.Pitch lidar.Roll],'eulerd','ZYX','frame'),'
    'IsParentToChild',true);

% Define transformation from ego to tracking coordinates
egoToScenario = struct('Frame',fusionCoordinateFrameType(1),...
    'OriginPosition',ego.Position(:),...
    'Orientation',rotmat(quaternion([ego.Yaw ego.Pitch ego.Roll],'eulerd','ZYX','frame'),'frame'
    'IsParentToChild',true);

% Assemble using trackingSensorConfiguration.
config = trackingSensorConfiguration(...
    'SensorIndex',lidar.SensorIndex,...
    'IsValidTime', true,...
    'SensorLimits',[lidar.AzimuthLimits;0 lidar.MaxRange],...
    'SensorTransformParameters',[senToEgo;egoToScenario],...
    'DetectionProbability',0.95);
end
```

References

[1] Nuss, Dominik, et al. "A random finite set approach for dynamic occupancy grid maps with real-time application." The International Journal of Robotics Research 37.8 (2018): 841-866.

[2] Steyer, Sascha, Georg Tanzmeister, and Dirk Wollherr. "Object tracking based on evidential dynamic occupancy grids in urban environments." 2017 IEEE Intelligent Vehicles Symposium (IV). IEEE, 2017.

See Also

`lidarPointCloudGenerator` | `trackerGridRFS`

More About

- “Track Vehicles Using Lidar: From Point Cloud to Track List” on page 7-177
- “Ground Plane and Obstacle Detection Using Lidar” on page 7-107
- “Extended Object Tracking of Highway Vehicles with Radar and Camera” on page 7-234
- “Extended Object Tracking With Radar For Marine Surveillance” (Sensor Fusion and Tracking Toolbox)

- “Extended Object Tracking with Lidar for Airport Ground Surveillance” (Sensor Fusion and Tracking Toolbox)

Track Multiple Lane Boundaries with a Global Nearest Neighbor Tracker

This example shows how to design and test a multiple lane tracking algorithm. The algorithm is tested in a driving scenario with probabilistic lane detections.

Introduction

An automated lane change maneuver (LCM) system enables the ego vehicle to automatically move from one lane to another lane. To successfully change lanes, the system requires localization of the ego vehicle with respect to stationary features, such as lane markings. A lane detection algorithm typically provides offset and curvature information about the current and adjacent lane boundaries. During the lane change, a discontinuity in the lateral offset is introduced into the lane detections, because the lateral offset is always with respect to the current lane in which the vehicle is traveling. The discontinuity and the jump in offset values may cause the LCM system to become unstable. One technique to compensate for this discontinuity issue is to use a multi-lane tracker.

Detect Lanes in a Lane Change Scenario

You load a `drivingScenario` object, `scenario`, that contains an ego vehicle and its sensors from the `LaneTrackingScenario.mat` file. You use a `visionDetectionGenerator` object to detect lanes in the scenario.

```
load('LaneTrackingScenario.mat','scenario','egoVehicle','sensors');
laneDetector = sensors{1};
```

To visualize the scenario in a Driving Scenario Designer, use:

```
drivingScenarioDesigner(scenario)
```

In this scenario, the ego vehicle is driving along a curved road with multiple lanes. The ego vehicle is equipped with a lane detector that detects lane boundaries and reports two lane boundaries on each side of the ego vehicle. To pass a slower moving vehicle traveling in the ego lane, the ego vehicle changes lanes from its original lane to the one on its left. The measurement reported by the lane detector contains the offset, the heading, and the curvature of the lane.

The following block of code runs the scenario and display the results of the lane detections.

```
% Setup plotting area
egoCarBEP = createDisplay(scenario,egoVehicle);

% Setup data logs
timeHistory = 0:scenario.SampleTime:(scenario.StopTime-scenario.SampleTime);
laneDetectionOffsetHistory = NaN(5,length(timeHistory));
timeStep = 1;
restart(scenario)
running = true;
while running
    % Simulation time
    simTime  = scenario.SimulationTime;

    % Get the ground truth lane boundaries in the ego vehicle frame
    groundTruthLanes = laneBoundaries(egoVehicle,'XDistance',0:5:70,'AllBoundaries',true);
    [egoBoundaries,egoBoundaryExist] = findEgoBoundaries(groundTruthLanes); %% ego lane and adja
    laneBoundaryDetections = laneDetector(egoBoundaries(egoBoundaryExist),simTime); %% ego lane a
```

```
    laneObjectDetections = packLanesAsObjectDetections(laneBoundaryDetections); %% convert lane b

    % Log lane detections
    for laneIDX = 1:length(laneObjectDetections)
        laneDetectionOffsetHistory(laneIDX,timeStep) = laneObjectDetections{laneIDX}.Measurement(
    end

    % Visualization road and lane ground truth in ego frame
    updateDisplay(egoCarBEP,egoVehicle,egoBoundaries);

    % Advance to the next step
    timeStep = timeStep + 1;
    running = advance(scenario);
end
```

Since the lane detector always reports the two lane markings on each side of the ego vehicle, the lane change causes it to report discontinuous lane markings. You can observe it in the graph below.

```
f=figure;
plot(timeHistory,laneDetectionOffsetHistory(1:4,:),'LineWidth',2)
xlabel('Time (s)')
ylabel('Lane detections lateral offset (m)')
legend('Adjacent left', 'Left', 'Right', 'Adjacent Right','Orientation','horizontal','Location',
grid
p = snapnow;

close(f)
```

Define a Multi-Lane Tracker and Track Lanes

Define a multi-lane tracker using the `trackerGNN` (Sensor Fusion and Tracking Toolbox) object. To delete undetected lane boundaries quickly, set the tracker to delete tracked lanes after three misses in three updates. Also set the maximum number of tracks to 10.

Use the `singer` (Sensor Fusion and Tracking Toolbox) acceleration model to model the way lane boundaries change over time. The Singer acceleration model enables you to model accelerations that decay with time, and you can set the decay rate using the decay constant `tau`. You use the `initSingerLane` on page 7-0 function modified from the `initsingerekf` (Sensor Fusion and Tracking Toolbox) function by setting the decay constant `tau` to 1, because the lane change maneuver time is relatively short. The function is attached at the end of the script. Note that the three dimensions defined for the Singer acceleration state are the offset, the heading, and the curvature of the lane boundary, which are the same as those reported in the lane detection.

```
laneTracker = trackerGNN('FilterInitializationFcn', @initSingerLane, 'DeletionThreshold', [3 3],
```

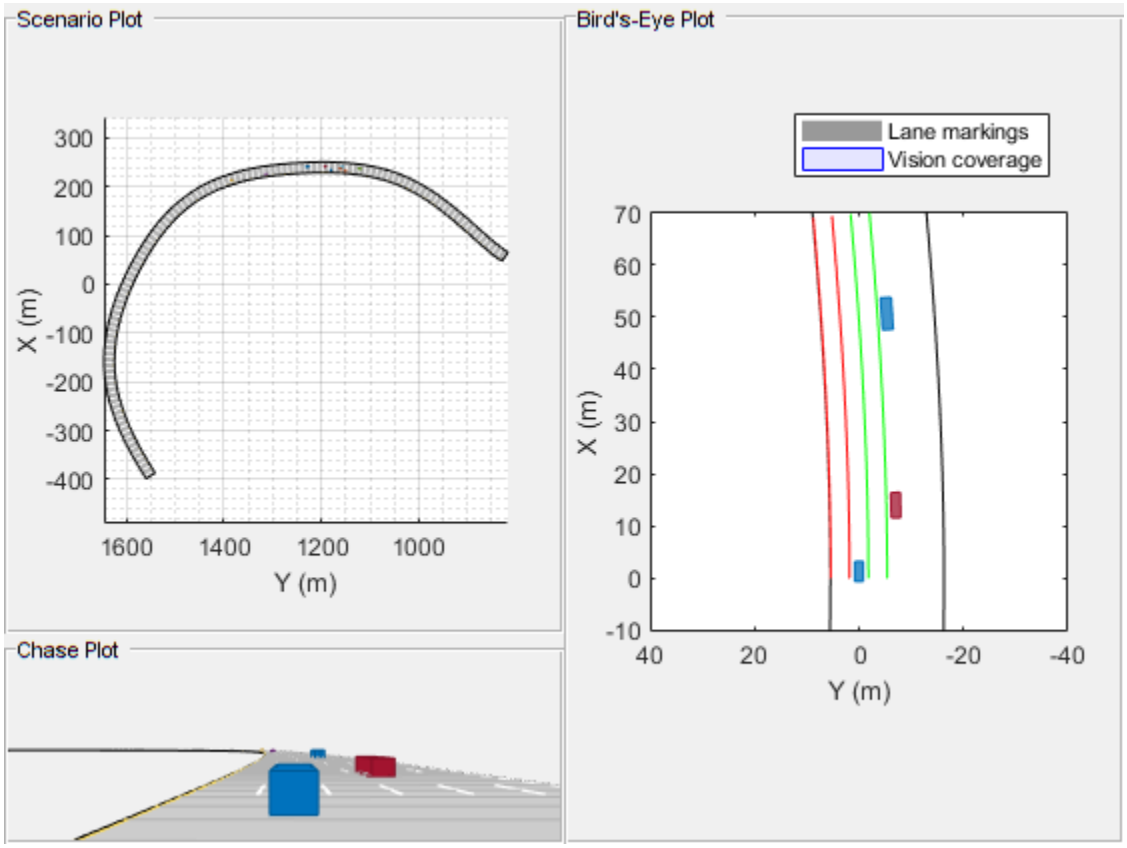
Rerun the scenario to track the lane boundaries.

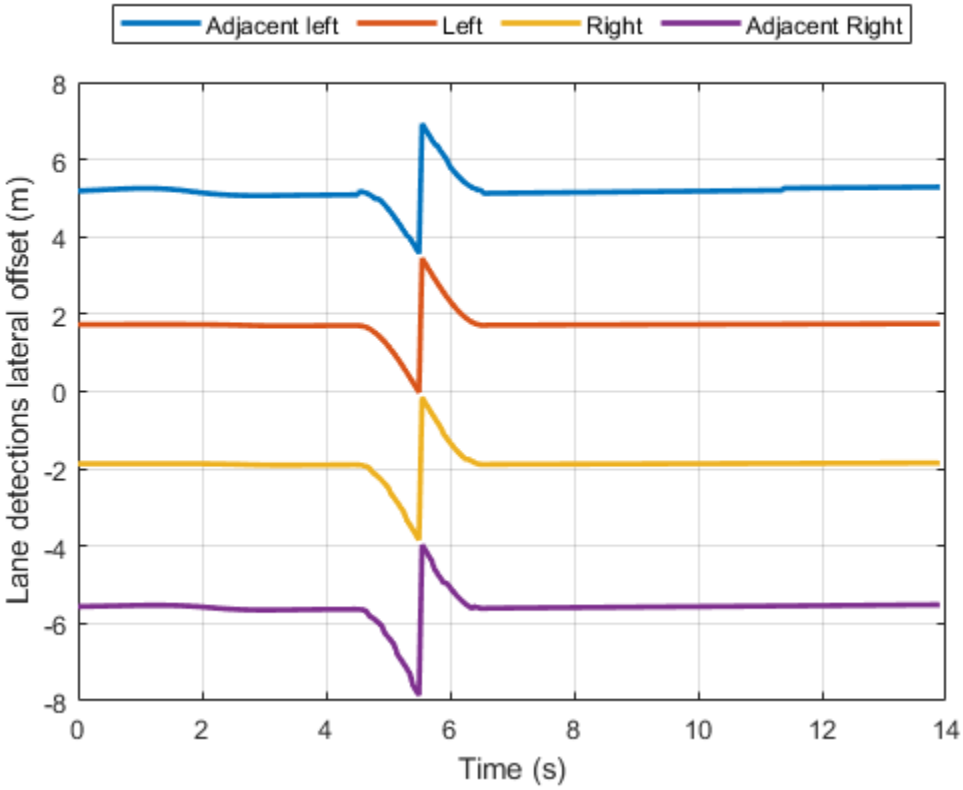
```
laneTrackOffsetHistory = NaN(5,length(timeHistory));
timeStep = 1;
restart(scenario)
restart(egoVehicle);
reset(laneDetector);
running = true;
while running
    % Simulation time
    simTime  = scenario.SimulationTime;
```

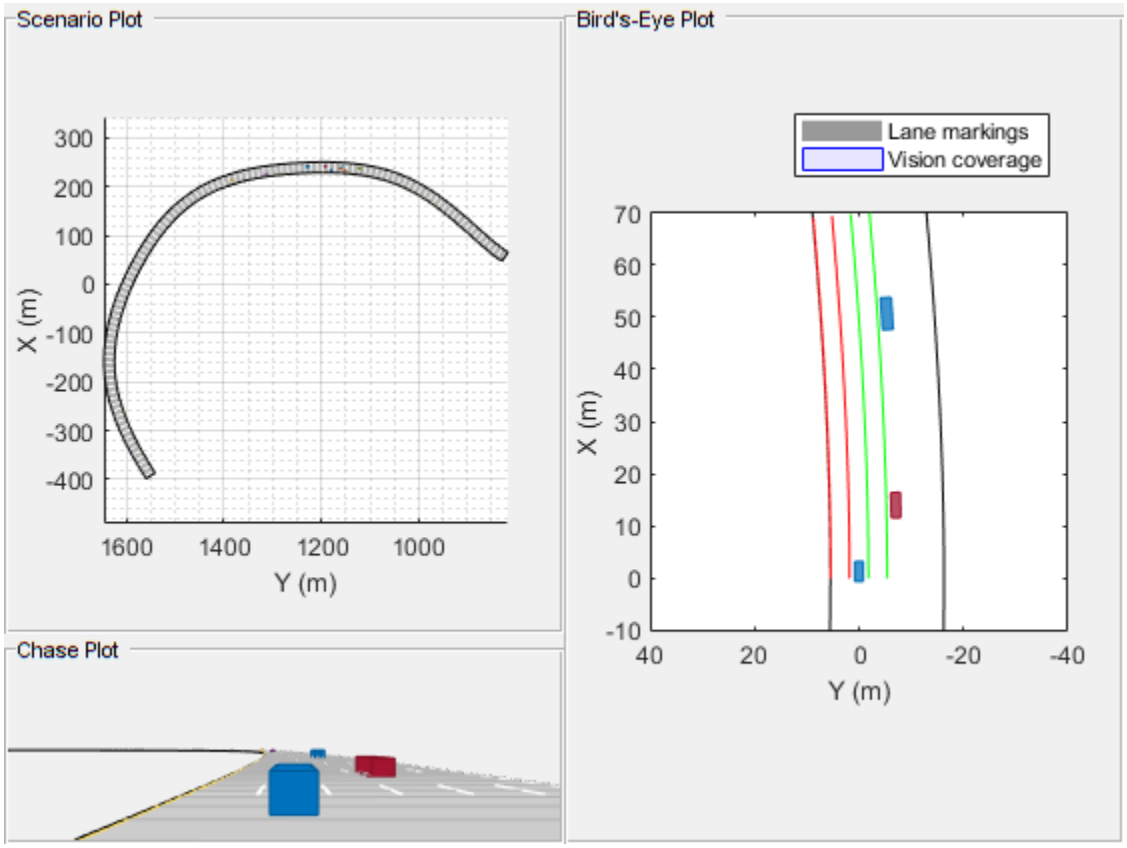
```
    % Get the ground truth lane boundaries in the ego vehicle frame
    groundTruthLanes = laneBoundaries(egoVehicle,'XDistance',0:5:70,'AllBoundaries',true);
    [egoBoundaries,egoBoundaryExist] = findEgoBoundaries(groundTruthLanes); %% ego lane and adja
    laneBoundaryDetections = laneDetector(egoBoundaries(egoBoundaryExist),simTime); %% ego lane a
    laneObjectDetections = packLanesAsObjectDetections(laneBoundaryDetections); %% convert lane b

    % Track the lanes
    laneTracks = laneTracker(laneObjectDetections,simTime);

    % Log data
    timeHistory(timeStep) = simTime;
    for laneIDX = 1:length(laneTracks)
        laneTrackOffsetHistory(laneTracks(laneIDX).TrackID,timeStep) = laneTracks(laneIDX).State
    end

    % Visualization road and lane ground truth in ego frame
    updateDisplay(egoCarBEP,egoVehicle,egoBoundaries);

    % Advance to the next step
    timeStep = timeStep + 1;
    running = advance(scenario);
end
```

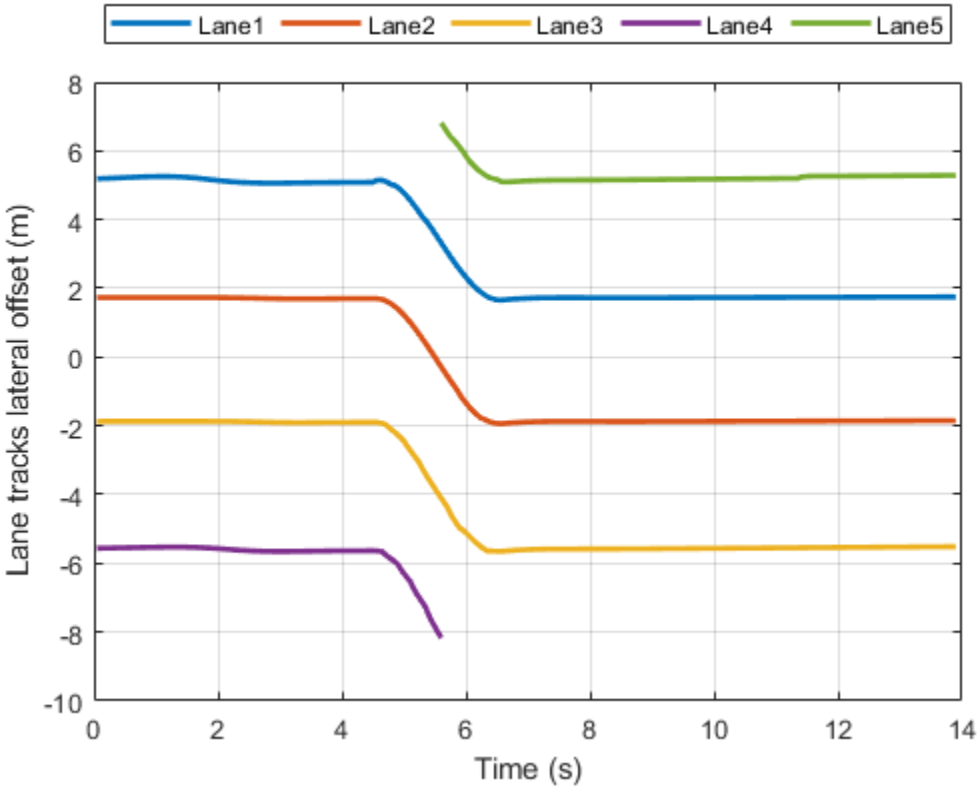
Plot the lateral offset of the tracked lane boundaries. Observe that the tracked lane boundaries are continuous and do not break when the ego vehicle performs the lane change maneuver.

```
figure
plot(timeHistory,laneTrackOffsetHistory(:,:),'LineWidth',2)
xlabel('Time (s)')
```


Scenario Plot
300
200
100
0
-100
-200
-300
-400
X (m)
1600
1400
1200
1000
Y (m)
Bird's-Eye Plot
Lane markings
Vision coverage
70
60
50
40
30
20
10
0
-10
X (m)
40
20
0
-20
-40
Y (m)
Chase Plot


Adjacent left
Left
Right
Adjacent Right
Lane detections lateral offset (m)
Time (s)
8
6
4
2
0
-2
-4
-6
-8
0
2
4
6
8
10
12
14


Scenario Plot
300
200
100
0
-100
-200
-300
-400
X (m)
1600
1400
1200
1000
Y (m)
Chase Plot
Bird's-Eye Plot
Lane markings
Vision coverage
70
60
50
40
30
20
10
0
-10
X (m)
40
20
0
-20
-40
Y (m)




```
ylabel('Lane tracks lateral offset (m)')
legend('Lane1', 'Lane2', 'Lane3', 'Lane4', 'Lane5', 'Orientation','horizontal','Location','northo
grid
```

Summary

In this example, you learned how to track multiple lanes. Without tracking the lanes, the lane detector reports discontinuous lane offsets relative to the ego vehicle when the ego vehicle changes lanes. The discontinuity in lane offsets can cause significant performance degradation of a closed-loop automated lane change system. You used a tracker to track the lanes and observed that the lane boundary offsets are continuous and can provide a stable input to the lane change system.

Supporting functions

`createDisplay` Create the display for this example

```
function egoCarBEP = createDisplay(scenario,egoVehicle)
hFigure = figure;
hPanel1 = uipanel(hFigure,'Units','Normalized','Position',[0 1/4 1/2 3/4],'Title','Scenario Plot
hPanel2 = uipanel(hFigure,'Units','Normalized','Position',[0 0 1/2 1/4],'Title','Chase Plot');
hPanel3 = uipanel(hFigure,'Units','Normalized','Position',[1/2 0 1/2 1],'Title','Bird''s-Eye Plo
hAxes1 = axes('Parent',hPanel1);
hAxes2 = axes('Parent',hPanel2);
hAxes3 = axes('Parent',hPanel3);
legend(hAxes3,'AutoUpdate','off')
scenario.plot('Parent',hAxes1)     % plot is a method of drivingScenario Class
chasePlot(egoVehicle,'Parent',hAxes2); % chase plot following the egoVehicle
```

```
egoCarBEP = birdsEyePlot('Parent',hAxes3,'XLimits',[-10 70],'YLimits',[-40 40]);
% Set up plotting type
outlinePlotter(egoCarBEP,'Tag','Platforms');
laneBoundaryPlotter(egoCarBEP,'Tag','Roads');
laneBoundaryPlotter(egoCarBEP,'Color','r','LineStyle','-','Tag','Left1');
laneBoundaryPlotter(egoCarBEP,'Color','g','LineStyle','-','Tag','Right1');
laneBoundaryPlotter(egoCarBEP,'Color','r','LineStyle','-','Tag','Left2');
laneBoundaryPlotter(egoCarBEP,'Color','g','LineStyle','-','Tag','Right2');
laneMarkingPlotter(egoCarBEP,'DisplayName','Lane markings','Tag','LaneMarkings');
coverageAreaPlotter(egoCarBEP,'DisplayName','Vision coverage','FaceAlpha',0.1,'FaceColor','b','Ed
end
```

`updateDisplay` Update the display for this example

```
function updateDisplay(egoCarBEP,egoVehicle,LaneBdryIn)
[position,yaw,leng,width,originOffset,color] = targetOutlines(egoVehicle);
outlineplotter = findPlotter(egoCarBEP,'Tag','Platforms');
plotOutline(outlineplotter, position, yaw, leng, width, ...
    'OriginOffset',originOffset,'Color',color)
rbdry = egoVehicle.roadBoundaries;
roadPlotter = findPlotter(egoCarBEP,'Tag','Roads');
roadPlotter.plotLaneBoundary(rbdry)
lbllPlotter = findPlotter(egoCarBEP,'Tag','Left2');
plotLaneBoundary(lbllPlotter,{LaneBdryIn(1).Coordinates})
lblPlotter = findPlotter(egoCarBEP,'Tag','Left1');
plotLaneBoundary(lblPlotter,{LaneBdryIn(2).Coordinates})
lbrPlotter = findPlotter(egoCarBEP,'Tag','Right1');
plotLaneBoundary(lbrPlotter,{LaneBdryIn(3).Coordinates})
lbrrPlotter = findPlotter(egoCarBEP,'Tag','Right2');
plotLaneBoundary(lbrrPlotter,{LaneBdryIn(4).Coordinates})
end
```

`findEgoBoundaries` Return the two nearest lane boundaries on each side of the ego vehicle

```
function [egoBoundaries,egoBoundaryExist] = findEgoBoundaries(groundTruthLanes)
%findEgoBoundaries  Find the two adjacent lane boundaries on each side of the ego
%  [egoBoundaries,egoBoundaryExist] = findEgoBoundaries(groundTruthLanes)
%  egoBoundaries - A 4x1 struct of lane boundaries ordered as: adjacent
%  left, left, right, and adjacent right
egoBoundaries = groundTruthLanes(1:4);
lateralOffsets = [groundTruthLanes.LateralOffset];
[sortedOffsets, inds] = sort(lateralOffsets);
egoBoundaryExist = [true;true;true;true];

% Left lane and left adjacent lane
idxLeft = find(sortedOffsets>0,2,'first');
numLeft = length(idxLeft);
egoBoundaries(2) = groundTruthLanes(inds(idxLeft(1)));
if numLeft>1
    egoBoundaries(1) = groundTruthLanes(inds(idxLeft(2)));
else % if left adjacent lane does not exist
    egoBoundaries(1) = egoBoundaries(2);
    egoBoundaryExist(1) = false;
end

% Right lane and right adjacent lane
idxRight = find(sortedOffsets<0,2,'last');
numRight = length(idxRight);
```

```
egoBoundaries(3) = groundTruthLanes(inds(idxRight(end)));
if numRight>1
    egoBoundaries(4) = groundTruthLanes(inds(idxRight(1)));
else % if right adjacent lane does not exist
    egoBoundaries(4) = egoBoundaries(3);
    egoBoundaryExist(4) = false;
end
end
```

`packLanesAsObjectDetections` Return lane boundary detections as a cell array of `objectDetection` objects

```
function laneObjectDetections = packLanesAsObjectDetections(laneBoundaryDetections)
%packLanesAsObjectDetections  Packs lane detections as a cell array of objectDetection
laneStrengths = [laneBoundaryDetections.LaneBoundaries.Strength];
IdxValid = find(laneStrengths>0);
numLaneDetections = length(IdxValid);
meas = zeros(3,1);
measurementParameters = struct(...
    'Frame', 'rectangular', ...
    'OriginPosition', [0 0 0]', ...
    'Orientation', eye(3,3), ...
    'HasVelocity', false, ...
    'HasElevation', false);

detection = objectDetection(laneBoundaryDetections.Time,meas, ...
    'MeasurementNoise', eye(3,3)/10, ...
    'SensorIndex', laneBoundaryDetections.SensorIndex, ...
    'MeasurementParameters' ,measurementParameters);
laneObjectDetections = repmat({detection},numLaneDetections,1);
for i = 1:numLaneDetections
    meas = [laneBoundaryDetections.LaneBoundaries(IdxValid(i)).LateralOffset ...
        laneBoundaryDetections.LaneBoundaries(IdxValid(i)).HeadingAngle/180*pi ...
        laneBoundaryDetections.LaneBoundaries(IdxValid(i)).Curvature/180*pi];
    laneObjectDetections{i}.Measurement = meas;
end
end
```

initSingerLane Define the Singer motion model for the lane boundary filter

```
function filter = initSingerLane(detection)
filter = initsingerekf(detection);
tau = 1;
filter.StateTransitionFcn = @(state,dt)singer(state,dt,tau);
filter.StateTransitionJacobianFcn = @(state,dt)singerjac(state,dt,tau);
filter.ProcessNoise = singerProcessNoise(zeros(9,1),1,tau,1);
end
```

See Also

`drivingScenario` | `singer` | `trackerGNN` | `visionDetectionGenerator`

More About

- “Multiple Object Tracking Tutorial” on page 7-162

Generate Code for a Track Fuser with Heterogeneous Source Tracks

This example shows how to generate code for a track-level fusion algorithm in a scenario where the tracks originate from heterogeneous sources with different state definitions. This example is based on the “Track-Level Fusion of Radar and Lidar Data” (Sensor Fusion and Tracking Toolbox) example, in which the state spaces of the tracks generated from lidar and radar sources are different.

Define a Track Fuser for Code Generation

You can generate code for a `trackFuser` (Sensor Fusion and Tracking Toolbox) using MATLAB® Coder™. To do so, you must modify your code to comply with the following limitations:

Code Generation Entry Function

Follow the instructions on how to use “System Objects in MATLAB Code Generation” (MATLAB Coder). For code generation, you must first define an entry-level function, in which the object is defined. Also, the function cannot use arrays of objects as inputs or outputs. In this example, you define the entry-level function as the heterogeneousInputsFuser function. The function must be on the path when you generate code for it. Therefore, it cannot be part of this live script and is attached in this example. The function accepts local tracks and current time as input and outputs central tracks.

To preserve the state of the fuser between calls to the function, you define the fuser as a `persistent` variable. On the first call, you must define the fuser variable because it is empty. The rest of the following code steps the `trackFuser` and returns the fused tracks.

```
function tracks = heterogeneousInputsFuser(localTracks,time)
%#codegen

persistent fuser
if isempty(fuser)
    % Define the radar source configuration
    radarConfig = fuserSourceConfiguration('SourceIndex',1,...
        'IsInitializingCentralTracks',true,...
        'CentralToLocalTransformFcn',@central2local,...
        'LocalToCentralTransformFcn',@local2central);

    % Define the lidar source configuration
    lidarConfig = fuserSourceConfiguration('SourceIndex',2,...
        'IsInitializingCentralTracks',true,...
        'CentralToLocalTransformFcn',@central2local,...
        'LocalToCentralTransformFcn',@local2central);

    % Create a trackFuser object
    fuser = trackFuser(...
        'MaxNumSources', 2, ...
        'SourceConfigurations',{radarConfig;lidarConfig},...
        'StateTransitionFcn',@helperctcuboid,...
        'StateTransitionJacobianFcn',@helperctcuboidjac,...
        'ProcessNoise',diag([1 3 1]),...
        'HasAdditiveProcessNoise',false,...
        'AssignmentThreshold',[250 inf],...
        'ConfirmationThreshold',[3 5],...
        'DeletionThreshold',[5 5],...
```

```
        'StateFusion','Custom',...
        'CustomStateFusionFcn',@helperRadarLidarFusionFcn);
end

tracks = fuser(localTracks, time);
end
```

Homogeneous Source Configurations

In this example, you define the radar and lidar source configurations differently than in the original "Track-Level Fusion of Radar and Lidar Data" (Sensor Fusion and Tracking Toolbox) example. In the original example, the `CentralToLocalTransformFcn` and `LocalToCentralTransformFcn` properties of the two source configurations are different because they use different function handles. This makes the source configurations a heterogeneous cell array. Such a definition is correct and valid when executing in MATLAB. However, in code generation, all source configurations must use the same function handles. To avoid the different function handles, you define one function to transform tracks from central (fuser) definition to local (source) definition and one function to transform from local to central. Each of these functions switches between the transform functions defined for the individual sources in the original example. Both functions are part of the heterogeneousInputsFuser function.

Here is the code for the `local2central` function, which uses the `SourceIndex` property to determine the correct function to use. Since the two types of local tracks transform to the same definition of central track, there is no need to predefine the central track.

```
function centralTrack = local2central(localTrack)
switch localTrack.SourceIndex
    case 1 % radar
        centralTrack = radar2central(localTrack);
    otherwise % lidar
        centralTrack = lidar2central(localTrack);
end
end
```

The function `central2local` transforms the central track into a radar track if `SourceIndex` is 1 or into a lidar track if `SourceIndex` is 2. Since the two tracks have a different definition of `State`, `StateCovariance`, and `TrackLogicState`, you must first predefine the output. Here is the code snippet for the function:

```
function localTrack = central2local(centralTrack)
state = 0;
stateCov = 1;
coder.varsize('state', [10, 1], [1 0]);
coder.varsize('stateCov', [10 10], [1 1]);
localTrack = objectTrack('State', state, 'StateCovariance', stateCov);

switch centralTrack.SourceIndex
    case 1
        localTrack = central2radar(centralTrack);
    case 2
        localTrack = central2lidar(centralTrack);
    otherwise
        % This branch is never reached but is necessary to force code
        % generation to use the predefined localTrack.
end
end
```

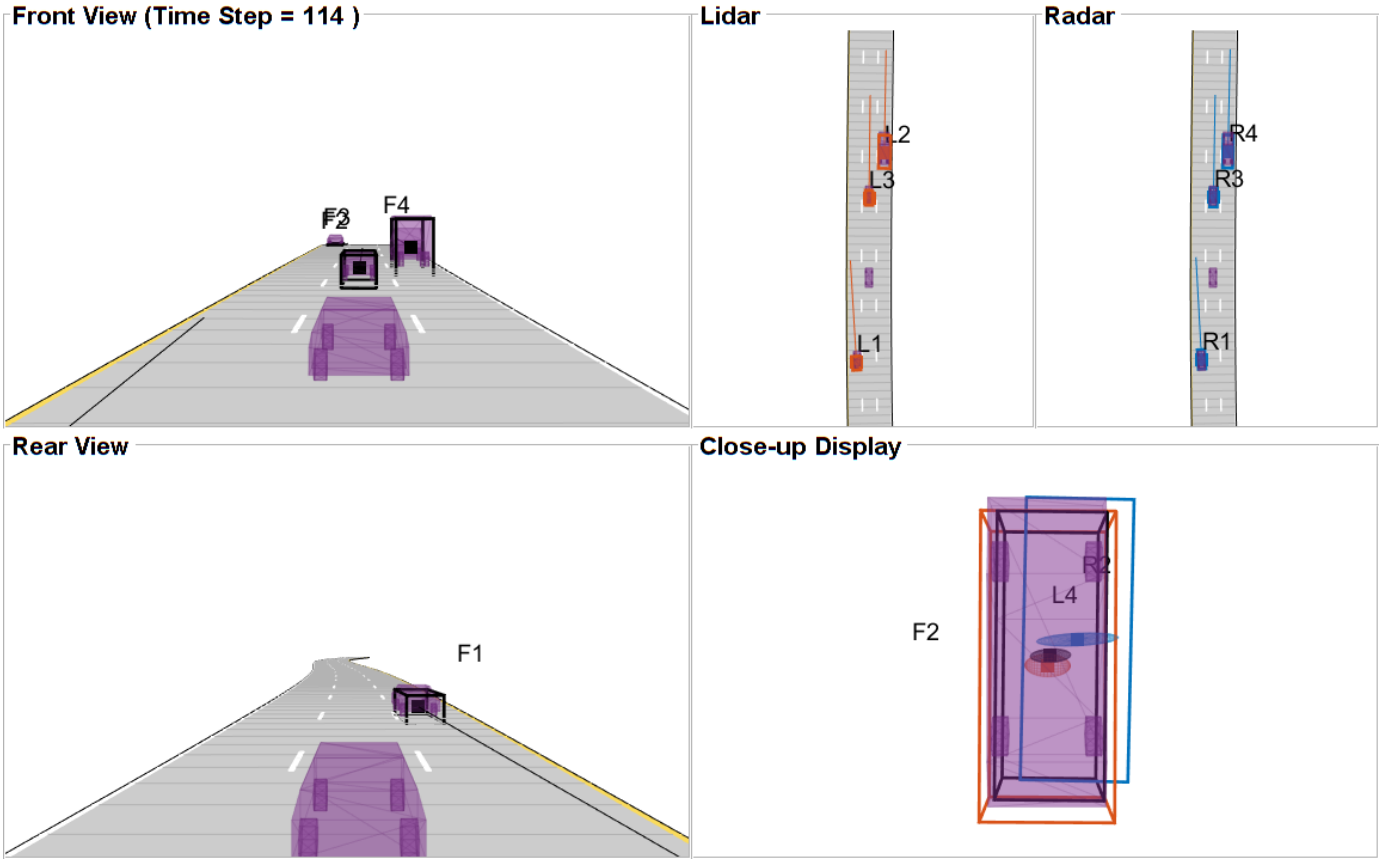
The functions `radar2central` and `central2radar` are the same as in the original example but moved from the live script to the heterogeneousInputsFuser function. You also add the `lidar2central` and `central2lidar` functions to the heterogeneousInputsFuser function. These two functions convert from the track definition that the fuser uses to the lidar track definition.

Run the Example in MATLAB

Before generating code, make sure that the example still runs after all the changes made to the fuser. The file `lidarRadarData.mat` contains the same scenario as in the original example. It also contains a set of radar and lidar tracks recorded at each step of that example. You also use a similar display to visualize the example and define the same `trackGOSPAMetric` objects to evaluate the tracking performance.

```
% Load the scenario and recorded local tracks
load('lidarRadarData.mat','scenario','localTracksCollection')
display = helperTrackFusionCodegenDisplay('FollowActorID',3);
showLegend(display,scenario);

% Radar GOSPA
gospaRadar = trackGOSPAMetric('Distance','custom',...
    'DistanceFcn',@helperRadarDistance,...
    'CutoffDistance',25);

% Lidar GOSPA
gospaLidar = trackGOSPAMetric('Distance','custom',...
    'DistanceFcn',@helperLidarDistance,...
    'CutoffDistance',25);

% Central/Fused GOSPA
gospaCentral = trackGOSPAMetric('Distance','custom',...
    'DistanceFcn',@helperLidarDistance,... % State space is same as lidar
    'CutoffDistance',25);


gospa = zeros(3,0);
missedTargets = zeros(3,0);
falseTracks = zeros(3,0);
% Ground truth for metrics. This variable updates every time step
% automatically, because it is a handle to the actors.
groundTruth = scenario.Actors(2:end);

fuserStepped = false;
fusedTracks = objectTrack.empty;
idx = 1;
clear heterogeneousInputsFuser
while advance(scenario)
    time = scenario.SimulationTime;
    localTracks = localTracksCollection{idx};

    if ~isempty(localTracks) || fuserStepped
        fusedTracks = heterogeneousInputsFuser(localTracks,time);
        fuserStepped = true;
    end

    radarTracks = localTracks([localTracks.SourceIndex]==1);
    lidarTracks = localTracks([localTracks.SourceIndex]==2);
```

```
        % Capture GOSPA and its components for all trackers
        [gospa(1,idx),~,~,~,missedTargets(1,idx),falseTracks(1,idx)] = gospaRadar(radarTracks, ground
        [gospa(2,idx),~,~,~,missedTargets(2,idx),falseTracks(2,idx)] = gospaLidar(lidarTracks, ground
        [gospa(3,idx),~,~,~,missedTargets(3,idx),falseTracks(3,idx)] = gospaCentral(fusedTracks, grou

        % Update the display
        display(scenario,[],[], radarTracks,...
            [],[],[],[], lidarTracks, fusedTracks);

        idx = idx + 1;
    end
```



Generate Code for the Track Fuser

To generate code, you must define the input types for both the radar and lidar tracks and the timestamp. In both the original script and in the previous section, the radar and lidar tracks are defined as arrays of `objectTrack` (Sensor Fusion and Tracking Toolbox) objects. In code generation, the entry-level function cannot use an array of objects. Instead, you define an array of structures.

You use the struct `oneLocalTrack` to define the inputs coming from radar and lidar tracks. In code generation, the specific data types of each field in the struct must be defined exactly the same as the types defined for the corresponding properties in the recorded tracks. Furthermore, the size of each field must be defined correctly. You use the `coder.typeof` (MATLAB Coder) function to specify fields that have variable size: `State`, `StateCovariance`, and `TrackLogicState`. You define the `localTracks` input using the `oneLocalTrack` struct and the `coder.typeof` function, because the

number of input tracks varies from zero to eight in each step. You use the function `codegen` (MATLAB Coder) to generate the code.

Notes:

1. If the input tracks use different types for the `State` and `StateCovariance` properties, you must decide which type to use, double or single. In this example, all tracks use double precision and there is no need for this step.
2. If the input tracks use different definitions of `StateParameters`, you must first create a superset of all `StateParameters` and use that superset in the `StateParameters` field. A similar process must be done for the `ObjectAttributes` field. In this example, all tracks use the same definition of `StateParameters` and `ObjectAttributes`.

```
% Define the inputs to fuserHeterogeneousInputs for code generation
oneLocalTrack = struct(...
    'TrackID', uint32(0), ...
    'BranchID', uint32(0), ...
    'SourceIndex', uint32(0), ...
    'UpdateTime', double(0), ...
    'Age', uint32(0), ...
    'State', coder.typeof(1, [10 1], [1 0]), ...
    'StateCovariance', coder.typeof(1, [10 10], [1 1]), ...
    'StateParameters', struct, ...
    'ObjectClassID', double(0), ...
    'TrackLogic', 'History', ...
    'TrackLogicState', coder.typeof(false, [1 10], [0 1]), ...
    'IsConfirmed', false, ...
    'IsCoasted', false, ...
    'IsSelfReported', false, ...
    'ObjectAttributes', struct);

localTracks = coder.typeof(oneLocalTrack, [8 1], [1 0]);
fuserInputArguments = {localTracks, time};

codegen heterogeneousInputsFuser -args fuserInputArguments;
```

Run the Example with the Generated Code

You run the generated code like you ran the MATLAB code, but first you must reinitialize the scenario, the GOSPA objects, and the display.

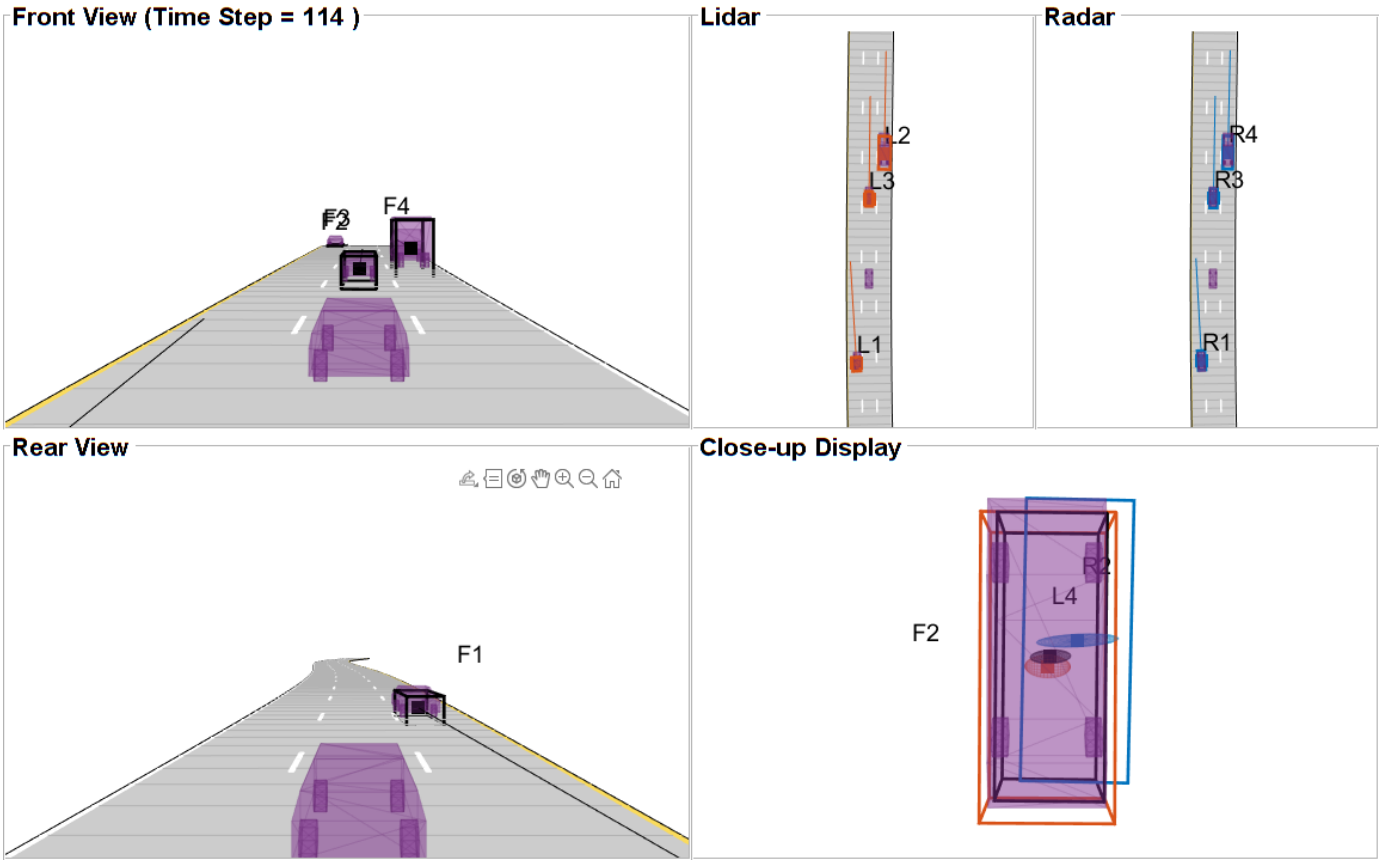
You use the `toStruct` (Sensor Fusion and Tracking Toolbox) object function to convert the input tracks to arrays of structures.

Notes:

1. If the input tracks use different data types for the `State` and `StateCovariance` properties, make sure to cast the `State` and `StateCovariance` of all the tracks to the data type you chose when you defined the `oneLocalTrack` structure above.
2. If the input tracks required a superset structure for the fields `StateParameters` or `ObjectAttributes`, make sure to populate these structures correctly before calling the `mex` file.

You use the `gospaCG` variable to keep the GOSPA metrics for this run so that you can compare them to the GOSPA values from the MATLAB run.

```
% Rerun the scenario with the generated code
fuserStepped = false;
fusedTracks = objectTrack.empty;
gospaCG = zeros(3,0);
missedTargetsCG = zeros(3,0);
falseTracksCG = zeros(3,0);

idx = 1;
clear heterogeneousInputsFuser_mex
reset(display);
reset(gospaRadar);
reset(gospaLidar);
reset(gospaCentral);
restart(scenario);
while advance(scenario)
    time = scenario.SimulationTime;
    localTracks = localTracksCollection{idx};

    if ~isempty(localTracks) || fuserStepped
        fusedTracks = heterogeneousInputsFuser_mex(toStruct(localTracks),time);
        fuserStepped = true;
    end

    radarTracks = localTracks([localTracks.SourceIndex]==1);
    lidarTracks = localTracks([localTracks.SourceIndex]==2);

    % Capture GOSPA and its components for all trackers
    [gospaCG(1,idx),~,~,~,missedTargetsCG(1,idx),falseTracksCG(1,idx)] = gospaRadar(radarTracks,
    [gospaCG(2,idx),~,~,~,missedTargetsCG(2,idx),falseTracksCG(2,idx)] = gospaLidar(lidarTracks,
    [gospaCG(3,idx),~,~,~,missedTargetsCG(3,idx),falseTracksCG(3,idx)] = gospaCentral(fusedTracks

    % Update the display
    display(scenario,[],[], radarTracks,...
        [],[],[],[], lidarTracks, fusedTracks);

    idx = idx + 1;
end
```



At the end of the run, you want to verify that the generated code provided the same results as the MATLAB code. Using the GOSPA metrics you collected in both runs, you can compare the results at the high level. Due to numerical roundoffs, there may be small differences in the results of the generated code relative to the MATLAB code. To compare the results, you use the absolute differences between GOSPA values and check if they are all smaller than 1e-10. The results show that the differences are very small.

```
% Compare the GOSPA values from MATLAB run and generated code
areGOSPAValuesEqual = all(abs(gospa-gospaCG)<1e-10,'all');
disp("Are GOSPA values equal up to the 10th decimal (true/false)? " + string(areGOSPAValuesEqual)
```

```
Are GOSPA values equal up to the 10th decimal (true/false)? true
```

Summary

In this example, you learned how to generate code for a track-level fusion algorithm when the input tracks are heterogeneous. You learned how to define the `trackFuser` and its `SourceConfigurations` property to support heterogeneous sources. You also learned how to define the input in compilation time and how to pass it to the mex file in runtime.

Supporting Functions

The following functions are used by the GOSPA metric.

`helperLidarDistance`

Function to calculate a normalized distance between the estimate of a track in radar state-space and the assigned ground truth.

```
function dist = helperLidarDistance(track, truth)

% Calculate the actual values of the states estimated by the tracker

% Center is different than origin and the trackers estimate the center
rOriginToCenter = -truth.OriginOffset(:) + [0;0;truth.Height/2];
rot = quaternion([truth.Yaw truth.Pitch truth.Roll],'eulerd','ZYX','frame');
actPos = truth.Position(:) + rotatepoint(rot,rOriginToCenter')';

% Actual speed and z-rate
actVel = [norm(truth.Velocity(1:2));truth.Velocity(3)];

% Actual yaw
actYaw = truth.Yaw;

% Actual dimensions.
actDim = [truth.Length;truth.Width;truth.Height];

% Actual yaw rate
actYawRate = truth.AngularVelocity(3);

% Calculate error in each estimate weighted by the "requirements" of the
% system. The distance specified using Mahalanobis distance in each aspect
% of the estimate, where covariance is defined by the "requirements". This
% helps to avoid skewed distances when tracks under/over report their
% uncertainty because of inaccuracies in state/measurement models.

% Positional error.
estPos = track.State([1 2 6]);
reqPosCov = 0.1*eye(3);
e = estPos - actPos;
d1 = sqrt(e'/reqPosCov*e);

% Velocity error
estVel = track.State([3 7]);
reqVelCov = 5*eye(2);
e = estVel - actVel;
d2 = sqrt(e'/reqVelCov*e);

% Yaw error
estYaw = track.State(4);
reqYawCov = 5;
e = estYaw - actYaw;
d3 = sqrt(e'/reqYawCov*e);

% Yaw-rate error
estYawRate = track.State(5);
reqYawRateCov = 1;
e = estYawRate - actYawRate;
d4 = sqrt(e'/reqYawRateCov*e);

% Dimension error
estDim = track.State([8 9 10]);
reqDimCov = eye(3);
e = estDim - actDim;
```

```
d5 = sqrt(e'/reqDimCov*e);

% Total distance
dist = d1 + d2 + d3 + d4 + d5;
end
```

helperRadarDistance

Function to calculate a normalized distance between the estimate of a track in radar state-space and the assigned ground truth.

```
function dist = helperRadarDistance(track, truth)
% Calculate the actual values of the states estimated by the tracker

% Center is different than origin and the trackers estimate the center
rOriginToCenter = -truth.OriginOffset(:) + [0;0;truth.Height/2];
rot = quaternion([truth.Yaw truth.Pitch truth.Roll],'eulerd','ZYX','frame');
actPos = truth.Position(:) + rotatepoint(rot,rOriginToCenter')';
actPos = actPos(1:2); % Only 2-D

% Actual speed
actVel = norm(truth.Velocity(1:2));

% Actual yaw
actYaw = truth.Yaw;

% Actual dimensions. Only 2-D for radar
actDim = [truth.Length;truth.Width];

% Actual yaw rate
actYawRate = truth.AngularVelocity(3);

% Calculate error in each estimate weighted by the "requirements" of the
% system. The distance specified using Mahalanobis distance in each aspect
% of the estimate, where covariance is defined by the "requirements". This
% helps to avoid skewed distances when tracks under/over report their
% uncertainty because of inaccuracies in state/measurement models.

% Positional error
estPos = track.State([1 2]);
reqPosCov = 0.1*eye(2);
e = estPos - actPos;
d1 = sqrt(e'/reqPosCov*e);

% Speed error
estVel = track.State(3);
reqVelCov = 5;
e = estVel - actVel;
d2 = sqrt(e'/reqVelCov*e);

% Yaw error
estYaw = track.State(4);
reqYawCov = 5;
e = estYaw - actYaw;
d3 = sqrt(e'/reqYawCov*e);

% Yaw-rate error
```

```
estYawRate = track.State(5);
reqYawRateCov = 1;
e = estYawRate - actYawRate;
d4 = sqrt(e'/reqYawRateCov*e);

% Dimension error
estDim = track.State([6 7]);
reqDimCov = eye(2);
e = estDim - actDim;
d5 = sqrt(e'/reqDimCov*e);

% Total distance
dist = d1 + d2 + d3 + d4 + d5;

% A constant penalty for not measuring 3-D state
dist = dist + 3;
end
```

See Also

`trackFuser`

More About

- “Track-to-Track Fusion for Automotive Safety Applications” on page 7-254
- “Code Generation for Tracking and Sensor Fusion” on page 7-117

Scenario Generation from Recorded Vehicle Data

This example shows how to generate a virtual driving scenario from recorded vehicle data. The scenario is generated from position information recorded from a GPS sensor and recorded object lists processed from a lidar sensor.

Overview

Virtual driving scenarios can be used to recreate a real scenario from recorded vehicle data. These virtual scenarios enable you to visualize and study the original scenario. Because you can programmatically modify virtual scenarios, you can also use them to synthesize scenario variations when designing and evaluating autonomous driving systems.

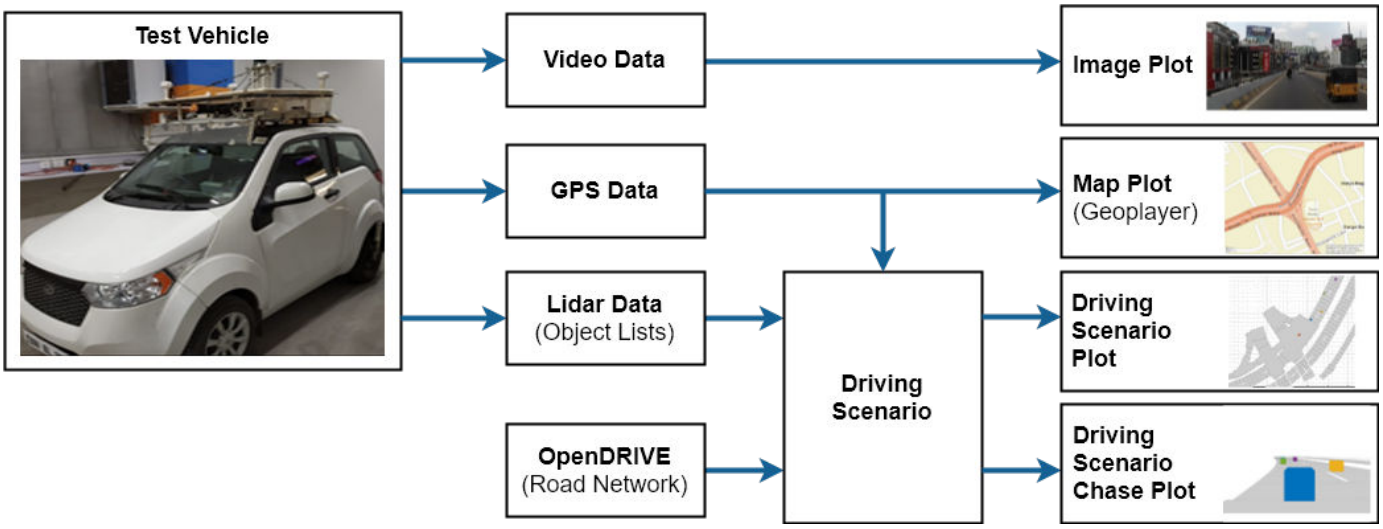
In this example, you create a virtual driving scenario by generating a `drivingScenario` object from data that was recorded from a test (ego) vehicle and an OpenDRIVE® file. The OpenDRIVE file describes the road network of the area where the data was recorded. The recorded vehicle data includes:

- **GPS data**: Text file containing the latitude and longitude coordinates of the ego vehicle at each timestamp.
- **Lidar object list data**: Text file containing the number of non-ego actors and the positions of their centers, relative to the ego vehicle, at each timestamp.
- **Video data**: MP4 file recorded from a forward-facing monocular camera mounted on the ego vehicle.

To generate and simulate the driving scenario, you follow these steps:

1. Explore recorded vehicle data.
2. Import OpenDRIVE road network into driving scenario.
3. Add ego vehicle data from GPS to driving scenario.
4. Add non-ego actors from lidar object list to driving scenario.
5. Simulate and visualize generated scenario.

The following diagram shows how you use the recorded data in this example. Notice that you create the driving scenario from the GPS, lidar object lists, and OpenDRIVE files. You use the camera data to visualize the original scenario and can compare this data with the scenario you generate. You also visualize the scenario route on a map using `geoplayer`.



Explore Recorded Vehicle Data

The positions of the ego vehicle were captured using a UBlox GPS NEO M8N sensor. The GPS sensor was placed on the center of the roof of the ego vehicle. This data is saved in the text file `EgoUrban.txt`.

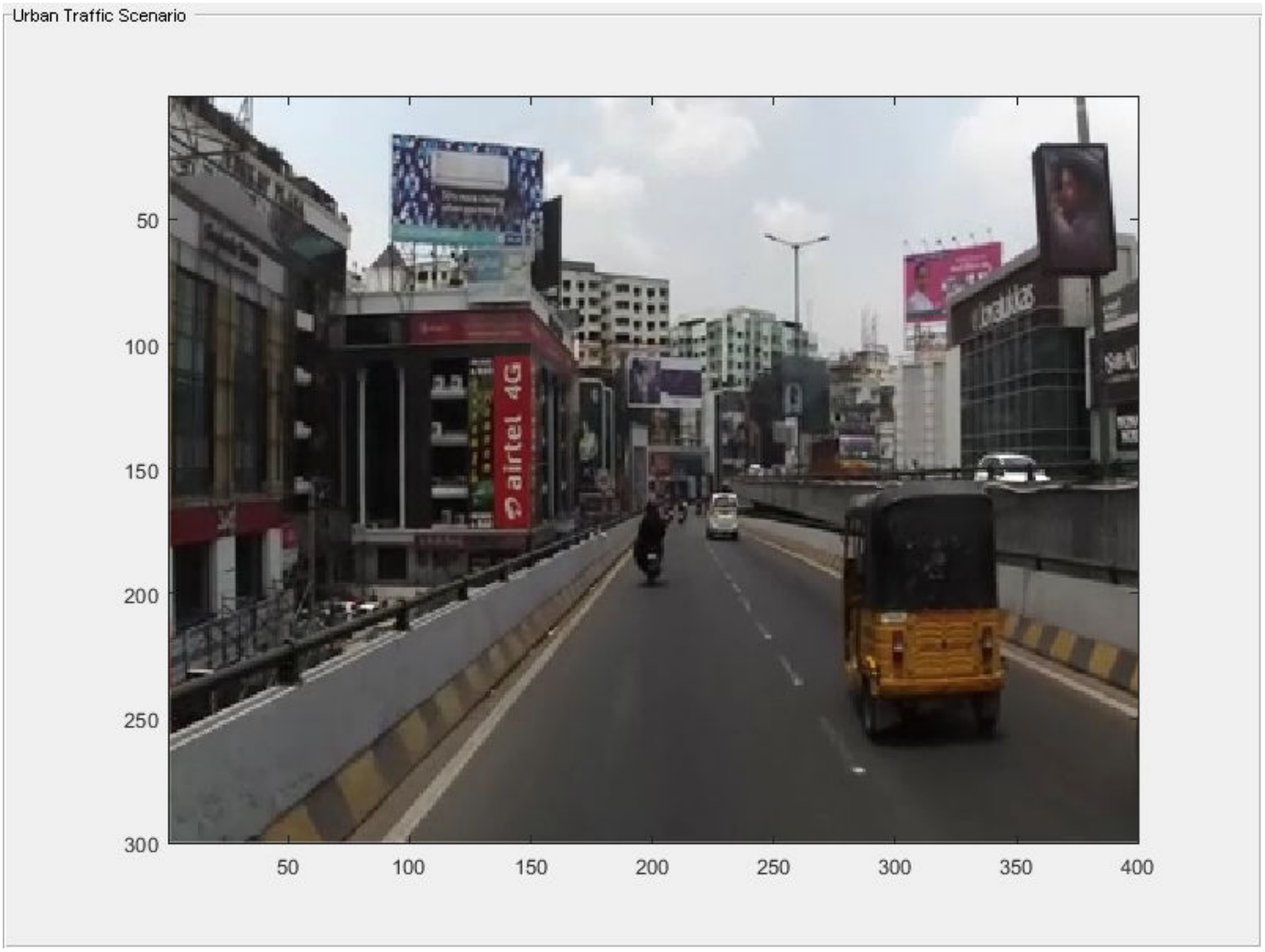
The positions of the non-ego actors were captured using a Velodyne® VLP-16 lidar sensor with a range of 30 meters. The VLP-16 sensor was placed on the roof of the ego vehicle at a position and height that avoids having the sensor collide with the body of the ego vehicle. The point cloud from the lidar sensor was processed on the vehicle to detect objects and their positions relative to the ego vehicle. This data is saved in the text file `NonEgoUrban.txt`.

To help understand the original scenario, video from a monocular camera was recorded as a reference. This video can also be used to visually compare the original and generated scenarios. A preview of this recorded video is saved in the video file `urbanpreview.mp4`. You can download the full recorded video file from here.

Generate a preview of the urban traffic scenario used in this example.

```
vidObj = VideoReader('urbanpreview.mp4');
fig = figure;
set(fig,'Position',[0, 0, 800, 600]);
movegui(fig,'center');
pnl = uipanel(fig,'Position',[0 0 1 1],'Title','Urban Traffic Scenario');
plt = axes(pnl);
while hasFrame(vidObj)
    vidFrame = readFrame(vidObj);
    image(vidFrame,'Parent',plt);
    currAxes.Visible = 'off';
    pause(1/vidObj.FrameRate);
end
```


Urban Traffic Scenario
50
100
150
200
250
300
50
100
150
200
250
300
350
400



Though the sensor coverage area can be defined around the entire ego vehicle, this example shows only the forward-looking scenario.

Import OpenDRIVE Road Network into Driving Scenario

The road network file for generating the virtual scenario was downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/. The OpenStreetMap data files are converted to OpenDRIVE files and saved with extension `.xodr`. Use the `roadNetwork` function to import this road network data to a driving scenario.

Create a driving scenario object and import the desired OpenDRIVE road network into the generated scenario.

```
scenario = drivingScenario;
openDRIVEFile = 'OpenDRIVEUrban.xodr';
roadNetwork(scenario,'OpenDRIVE',openDRIVEFile);
```

Add Ego Vehicle Data from GPS to Generated Scenario

The ego vehicle data is collected from the GPS sensor and stored as a text file. The text file consists of three columns that store the latitude, longitude, and timestamp values for the ego vehicle. Use the `helperGetEgoData` function to import the ego vehicle data from the text file into a structure in the MATLAB® workspace. The structure contains three fields specifying the latitude, longitude and timestamps.

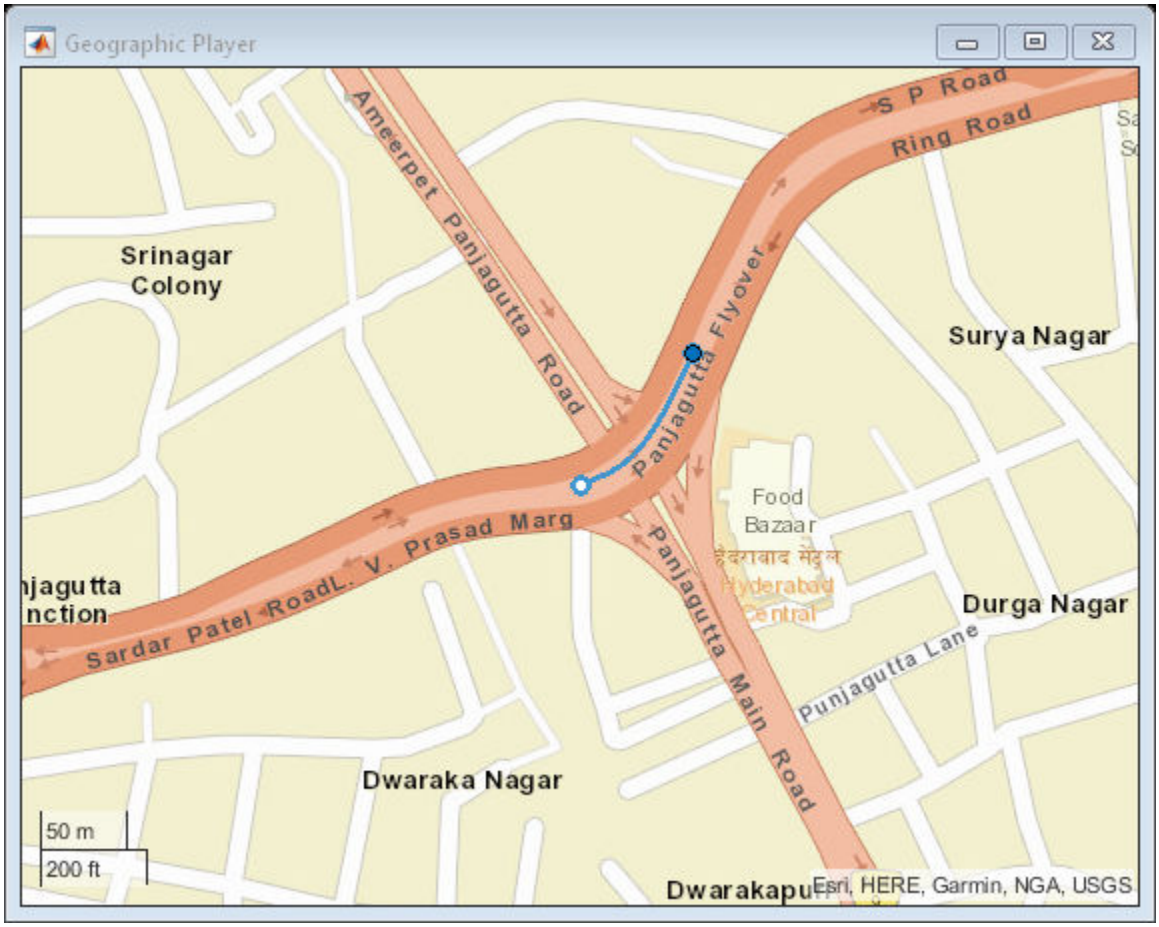
```
egoFile = 'EgoUrban.txt';
egoData = helperGetEgoData(egoFile);
```

Compute the trajectory waypoints of the ego vehicle from the recorded GPS coordinates. Use the `latlon2local` function to convert the raw GPS coordinates to the local east-north-up Cartesian coordinates. The transformed coordinates define the trajectory waypoints of the ego vehicle.

```
% Specify latitude and longitude at origin of data from OpenDRIVE file. This point will also defi
alt = 540.0; % Average altitude in Hyderabad, India
origin = [17.425853702697903, 78.44939480188313, alt]; % [lat, lon, altitude]
% Specify latitude and longitude of ego vehicle
lat = egoData.lat;
lon = egoData.lon;
% Compute waypoints of ego vehicle
[X,Y,~] = latlon2local(lat,lon,alt,origin);
egoWaypoints(:,1) =  X;
egoWaypoints(:,2) =  Y;
```

Visualize the GPS path of the ego vehicle using the `geoplayer` object.

```
zoomLevel = 17;
player = geoplayer(lat(1),lon(1),zoomLevel);
plotRoute(player,lat,lon);
for i = 1:length(lat)
    plotPosition(player,lat(i),lon(i));
end
```



Use `helperComputeEgoData` to compute the speed and the heading angle values of the ego vehicle at each sensor data timestamp.

```
[egoSpeed,egoAngle] = helperComputeEgoData(egoData,X,Y);
```

Add the ego vehicle to the driving scenario.

```
ego = vehicle(scenario,'ClassID',1,'Length',1,'Width',0.6,'Height',0.6);
```

Create a trajectory for the ego vehicle from the computed set of ego waypoints and the speed. The ego vehicle follows the trajectory at the specified speed.

```
trajectory(ego,egoWaypoints,egoSpeed);
```

Add Non-Ego Actors from Lidar Object Lists to Generated Scenario

The non-ego actor data is collected from the lidar sensor and stored as a text file. The text file consists of five columns that store the actor IDs, `x`-positions, `y`-positions, `z`-positions and timestamp values, respectively. Use the `helperGetNonEgoData` function to import the non-ego actor data from the text file into a structure in the MATLAB® workspace. The output is a structure with three fields:

1. `ActorID` - Scenario-defined actor identifier, specified as a positive integer.
2. `Position` - Position of actor, specified as an [*x y z*] real vector. Units are in meters.

3 `Time` - Timestamp of the sensor recording.

```
nonEgoFile = 'NonEgoUrban.txt';
nonEgoData = helperGetNonEgoData(nonEgoFile);
```

Use `helperComputeNonEgoData` to compute the trajectory waypoints and the speed of each non-ego actor at each timestamp. The trajectory waypoints are computed relative to the ego vehicle.

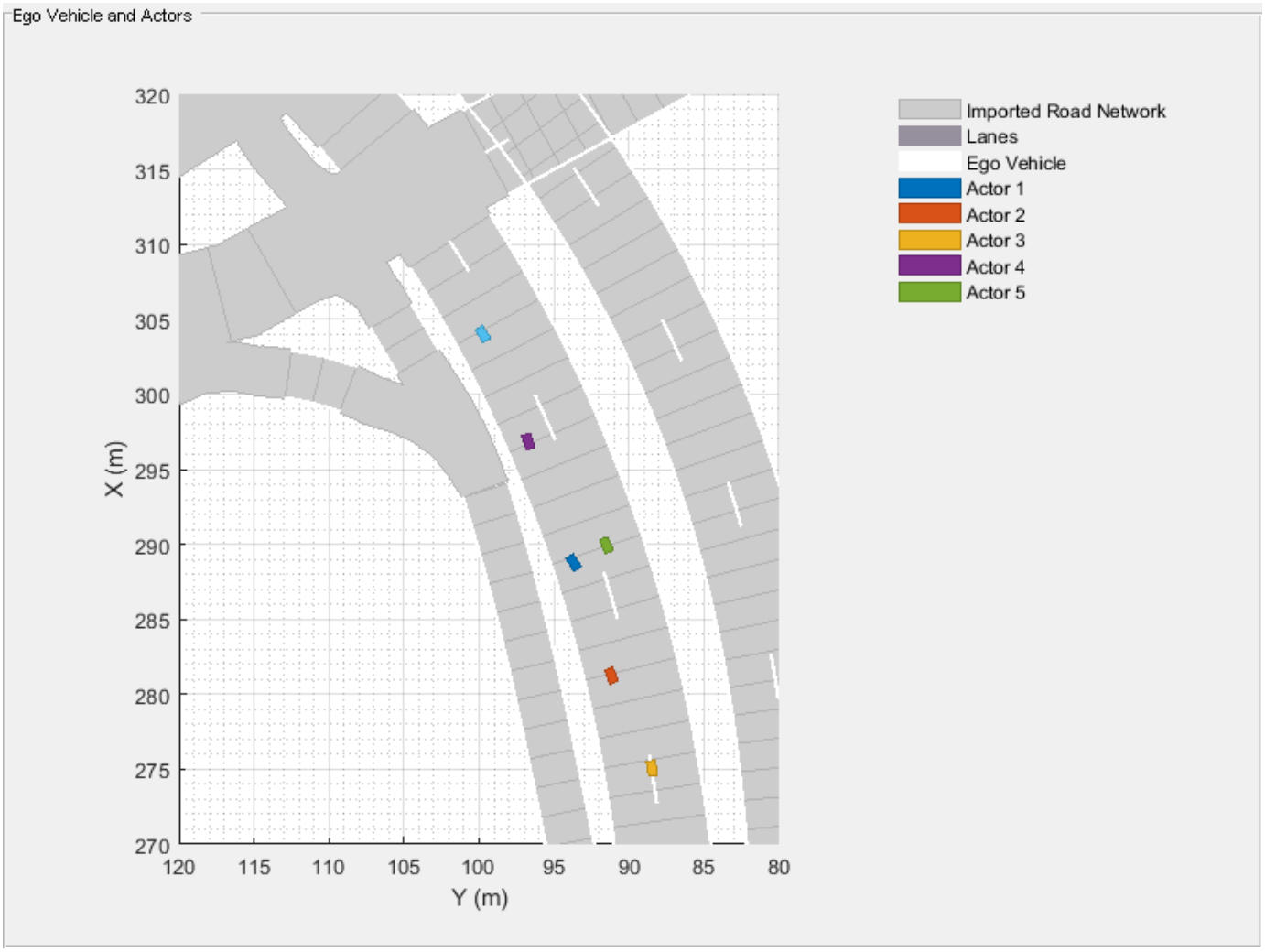
```
actors = unique(nonEgoData.ActorID);
[nonEgoSpeed, nonEgoWaypoints] = helperComputeNonEgoData(egoData,egoWaypoints,nonEgoData,egoAngle
```

Add the non-ego actors to the driving scenario. Create trajectories for the non-ego actors from the computed set of actor waypoints and the speed.

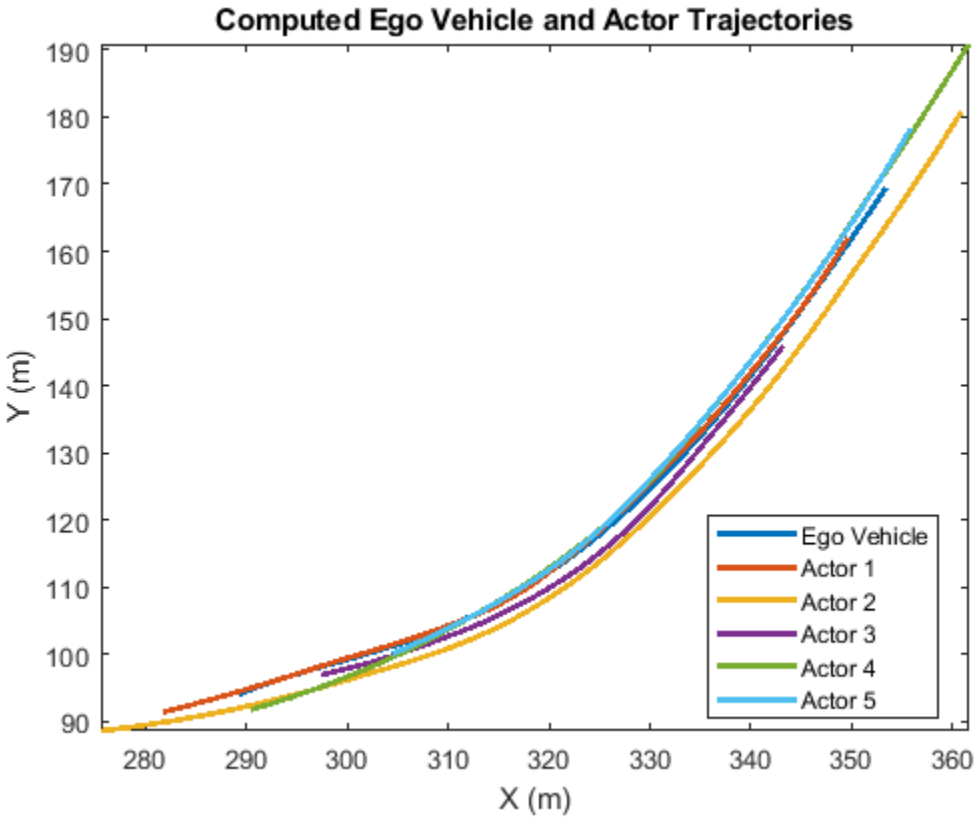
```
for i = 1:length(actors)
    actor = vehicle(scenario,'ClassID',1,'Length',1,'Width',0.6,'Height',0.6);
    trajectory(actor,nonEgoWaypoints{i},nonEgoSpeed{i});
end
```

Visualize the ego vehicle and non-ego actors that you imported into the generated scenario. Also visualize the corresponding trajectory waypoints of the ego vehicle and non-ego actors.

```
% Create a custom figure window and define an axes object
fig = figure;
set(fig,'Position',[0, 0, 800, 600]);
movegui(fig,'center');
hViewPnl = uipanel(fig,'Position',[0 0 1 1],'Title','Ego Vehicle and Actors');
hCarPlt = axes(hViewPnl);

% Plot the generated driving scenario.
plot(scenario,'Parent',hCarPlt);
axis([270 320 80 120]);
legend('Imported Road Network','Lanes','Ego Vehicle','Actor 1','Actor 2','Actor 3','Actor 4','Ac
legend(hCarPlt,'boxoff');
```




```
figure,
plot(egoWaypoints(:,1),egoWaypoints(:,2),'Color',[0 0.447 0.741],'LineWidth',2)
hold on
cMValues = [0.85 0.325 0.098;0.929 0.694 0.125;0.494 0.184 0.556;0.466 0.674 0.188;0.301 0.745 0
for i =1:length(actors)
    plot(nonEgoWaypoints{i}(:,1),nonEgoWaypoints{i}(:,2),'Color',cMValues(i,:),'LineWidth',2)
end
axis('tight')
xlabel('X (m)')
ylabel('Y (m)')
title('Computed Ego Vehicle and Actor Trajectories')
legend('Ego Vehicle', 'Actor 1', 'Actor 2', 'Actor 3','Actor 4','Actor 5','Location','Best')
hold off
```

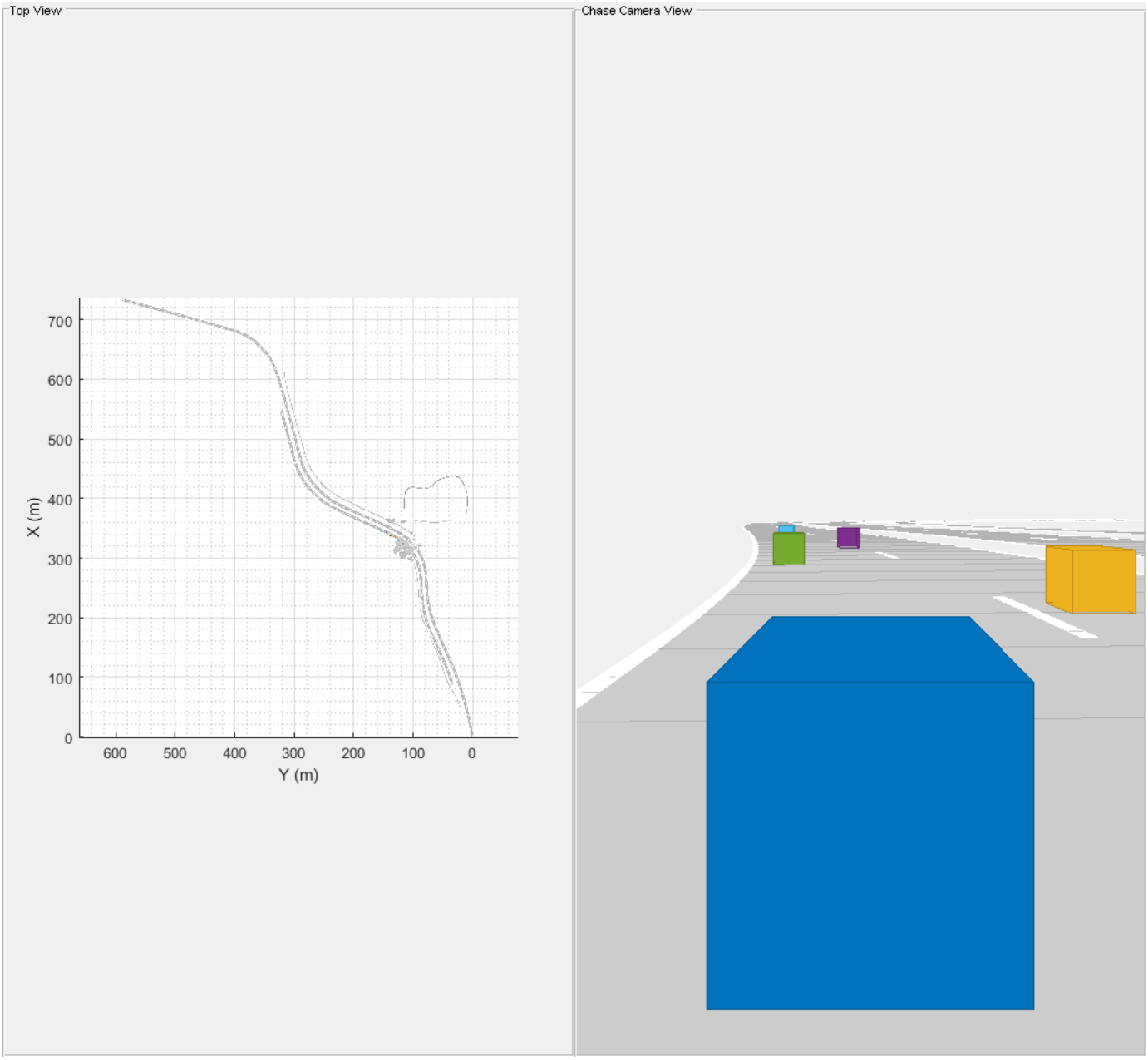


Simulate and Visualize Generated Scenario

Plot the scenario and the corresponding chase plot. Run the simulation to visualize the generated driving scenario. The ego vehicle and the non-ego actors follow their respective trajectories.

```
% Create a custom figure window to show the scenario and chase plot
close all;
figScene = figure('Name','Driving Scenario','Tag','ScenarioGenerationDemoDisplay');
set(figScene,'Position',[0, 0, 1032, 1032]);
movegui(figScene,'center');

% Add the chase plot
hCarViewPanel = uipanel(figScene,'Position',[0.5 0 0.5 1],'Title','Chase Camera View');
hCarPlot = axes(hCarViewPanel);
chasePlot(ego,'Parent',hCarPlot);

% Add the top view of the generated scenario
hViewPanel = uipanel(figScene,'Position',[0 0 0.5 1],'Title','Top View');
hCarPlot = axes(hViewPanel);
plot(scenario,'Parent',hCarPlot);
% Set the axis limits to display only the active area
xMin = min(egoWaypoints(:,1));
xMax = max(egoWaypoints(:,1));
yMin = min(egoWaypoints(:,2));
yMax = max(egoWaypoints(:,2));
limits = [xMin xMax yMin yMax];
axis(limits);
```

```
% Run the simulation
while advance(scenario)
    pause(0.01)
end
```



Summary

This example shows how to automatically generate a virtual driving scenario from vehicle data recorded using the GPS and lidar sensors.

Helper Functions

helperGetEgoData

This function reads the ego vehicle data from a text file and converts into a structure.

```
function [egoData] = helperGetEgoData(egoFile)
%Read the ego vehicle data from text file
fileID = fopen(egoFile);
content = textscan(fileID,'%f %f %f');
fields = {'lat','lon','Time'};
egoData = cell2struct(content,fields,2);
fclose(fileID);
end
```

helperGetNonEgoData

This function reads the processed lidar data from a text file and converts into a structure. The processed lidar data contains information about the non-ego actors.

```
function [nonEgoData] = helperGetNonEgoData(nonEgoFile)
% Read the processed lidar data of non-ego actors from text file.
fileID = fopen(nonEgoFile);
content = textscan(fileID,'%d %f %f %f %f');
newcontent{1} = content{1};
newcontent{2} = [content{2} content{3} content{4}];
newcontent{3} = content{5};
fields = {'ActorID','Position','Time'};
nonEgoData = cell2struct(newcontent,fields,2);
fclose(fileID);
end
```

helperComputeEgoData

This function calculates the speed and heading angle of the ego vehicle based on the trajectory waypoints and the timestamps.

```
function [egoSpeed, egoAngle] = helperComputeEgoData(egoData, X, Y)
egoTime = egoData.Time;
timeDiff = diff(egoTime);
points = [X Y];
difference  = diff(points, 1);
distance  = sqrt(sum(difference .* difference, 2));
egoSpeed = distance./timeDiff;
egoAngle = atan(diff(Y)./diff(X));
egoAngle(end+1) = egoAngle(end);
egoSpeed(end+1) = egoSpeed(end);
end
```

helperComputeNonEgoData

This function calculates the speed and heading angle of each non-ego actor based on the trajectory waypoints and timestamps. The speed and heading angle are calculated relative to the ego vehicle.

```
function [nonEgoSpeed, nonEgoWaypoints] = helperComputeNonEgoData(egoData, egoWaypoints, nonEgoDa

actors = unique(nonEgoData.ActorID);
numActors = length(actors);
```

```
nonEgoWaypoints = cell(numActors, 1);
nonEgoSpeed     = cell(numActors, 1);

for i = 1:numActors
    id = actors(i);
    idx = find([nonEgoData.ActorID] == id);
    actorXData = nonEgoData.Position(idx,1);
    actorYData = nonEgoData.Position(idx,2);
    actorTime = nonEgoData.Time(idx);
    actorWaypoints = [0 0];

    % Compute the trajectory waypoints of non-ego actor
    [sharedTimeStamps,nonEgoIdx,egoIdx] = intersect(actorTime,egoData.Time,'stable');
    tempX = actorXData(nonEgoIdx);
    tempY = actorYData(nonEgoIdx);
    relativeX = -tempX .* cos(egoAngle(egoIdx)) + tempY .* sin(egoAngle(egoIdx));
    relativeY = -tempX .* sin(egoAngle(egoIdx)) - tempY .* cos(egoAngle(egoIdx));
    actorWaypoints(nonEgoIdx,1) = egoWaypoints(egoIdx,1) + relativeX;
    actorWaypoints(nonEgoIdx,2) = egoWaypoints(egoIdx,2) + relativeY;

    % Compute the speed values of non-ego actor
    timeDiff = diff(sharedTimeStamps);
    difference = diff(actorWaypoints, 1);
    distance   = sqrt(sum(difference .* difference, 2));
    actorSpeed = distance./timeDiff;
    actorSpeed(end+1) = actorSpeed(end);

    % Smooth the trajectory waypoints of non-ego actor
    actorWaypoints = smoothdata(actorWaypoints,'sgolay');

    % Store the values of trajectory waypoints and speed computed of each non-ego actor
    nonEgoWaypoints(i) = {actorWaypoints};
    nonEgoSpeed(i) = {actorSpeed'};
end
end
```

See Also

Apps
Driving Scenario Designer

Functions
`roadNetwork` | `trajectory` | `vehicle`

Objects
`drivingScenario` | `geoplayer` | `visionDetectionGenerator`

More About

- “Create Driving Scenario Programmatically” on page 7-423
- “Create Actor and Vehicle Trajectories Programmatically” on page 7-442

External Websites

- ASAM OpenDRIVE

Lane Keeping Assist with Lane Detection

This example shows how to simulate and generate code for an automotive lane keeping assist (LKA) controller.

In this example, you:

1. Review a control algorithm that combines data processing from lane detections and a lane keeping controller from the Model Predictive Control Toolbox™.
2. Test the control system in a closed-loop Simulink model using synthetic data generated by the Automated Driving Toolbox™.
3. Configure the code generation settings for software-in-the-loop simulation and automatically generate code for the control algorithm.

Introduction

A lane keeping assist (LKA) system is a control system that aids a driver in maintaining safe travel within a marked lane of a highway. The LKA system detects when the vehicle deviates from a lane and automatically adjusts the steering to restore proper travel inside the lane without additional input from the driver. In this example, the LKA system switches between the driver steering command and lane keeping controller. This approach is sufficient to introduce a modeling architecture for an LKA system, however a real system would also provide haptic feedback to the steering wheel and enable the driver to override the LKA system by applying sufficient counter-torque.

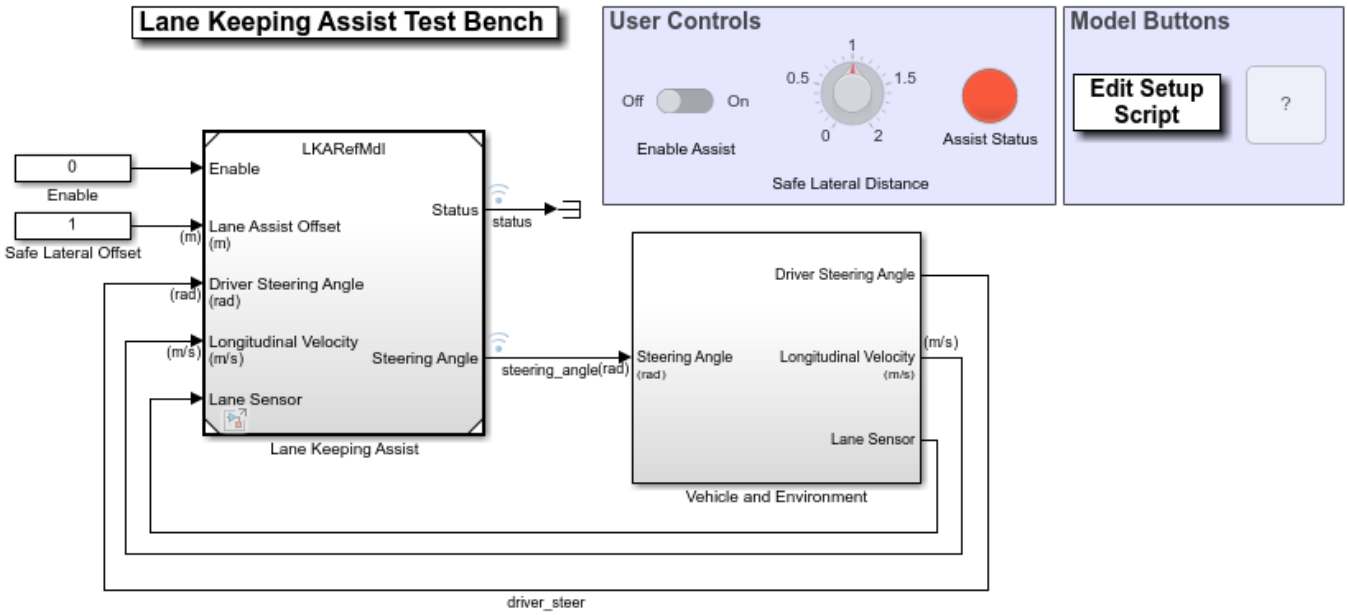
For the LKA to work correctly, the ego vehicle must determine the lane boundaries and how the lane in front of it curves. Idealized LKA designs rely mostly on the previewed curvature, the lateral deviation, and relative yaw angle between the centerline of the lane and the ego vehicle. An example of such a system is given in “Lane Keeping Assist System Using Model Predictive Control” (Model Predictive Control Toolbox). Moving from advanced drive-assistance system (ADAS) designs to more autonomous systems, the LKA must be robust to missing, incomplete, or inaccurate measurement readings from real-world lane detectors.

This example demonstrates a robust approach to the controller design when the data from lane detections may not be accurate. To do so, it uses data from a synthetic lane detector that simulates the impairments introduced by a wide-angle monocular vision camera. The controller makes decisions when the data from sensor is invalid or outside a range. This provides a safety guard when the sensor measurement is false due to conditions in the environment, such as a sharp curve on the road.

Open Test Bench Model

To open the Simulink test bench model, use the following command.

```
open_system('LKATestBenchExample')
```

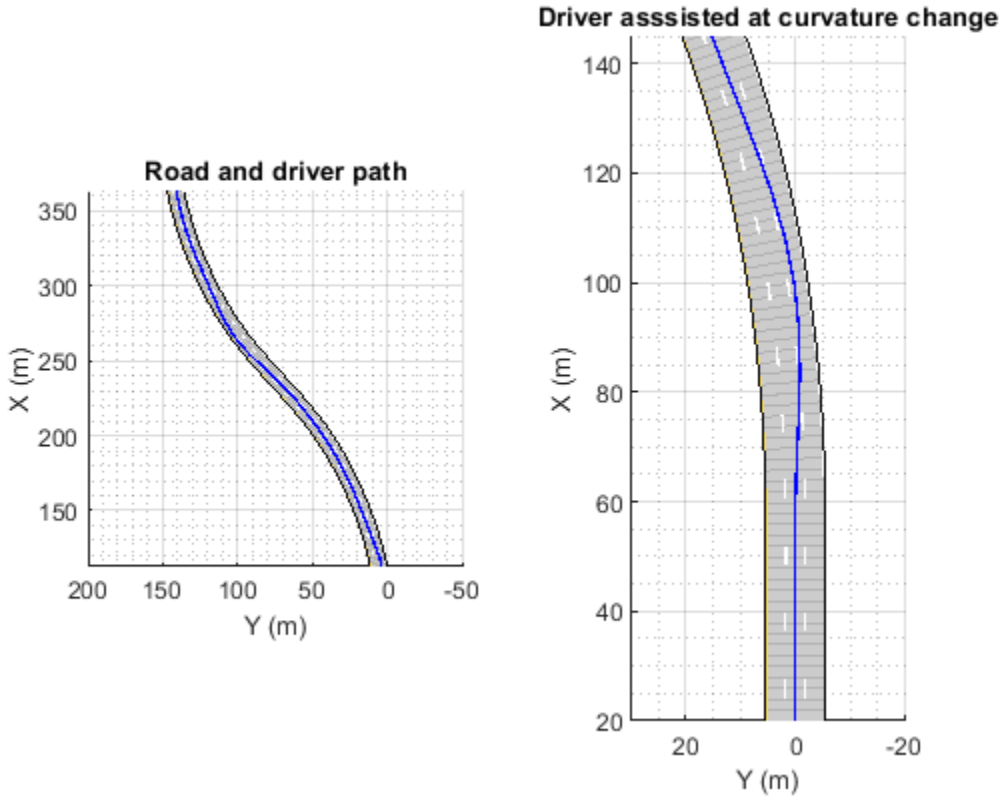


The model contains two main subsystems:

1. Lane Keeping Assist, which controls the front steering angle of the vehicle
2. Vehicle and Environment subsystem, which models the motion of the ego vehicle and models the environment

Opening this model also runs the `helperLKASetUp` script, which initializes data used by the model. The script loads certain constants needed by the Simulink model, such as the vehicle model parameters, controller design parameters, road scenario, and driver path. You can plot the road and the path that the driver model will follow.

```
plotLKAInputs(scenario,driverPath)
```



Simulate Assisting a Distracted Driver

You can explore the behavior of the algorithm by enabling lane-keeping assistance and setting the safe lateral distance. In the Simulink model, in the **User Controls** section, switch the toggle to **On**, and set the **Safe Lateral Distance** to 1 meter. Alternatively, enable the lane-keeping assist and set the safe lateral distance.

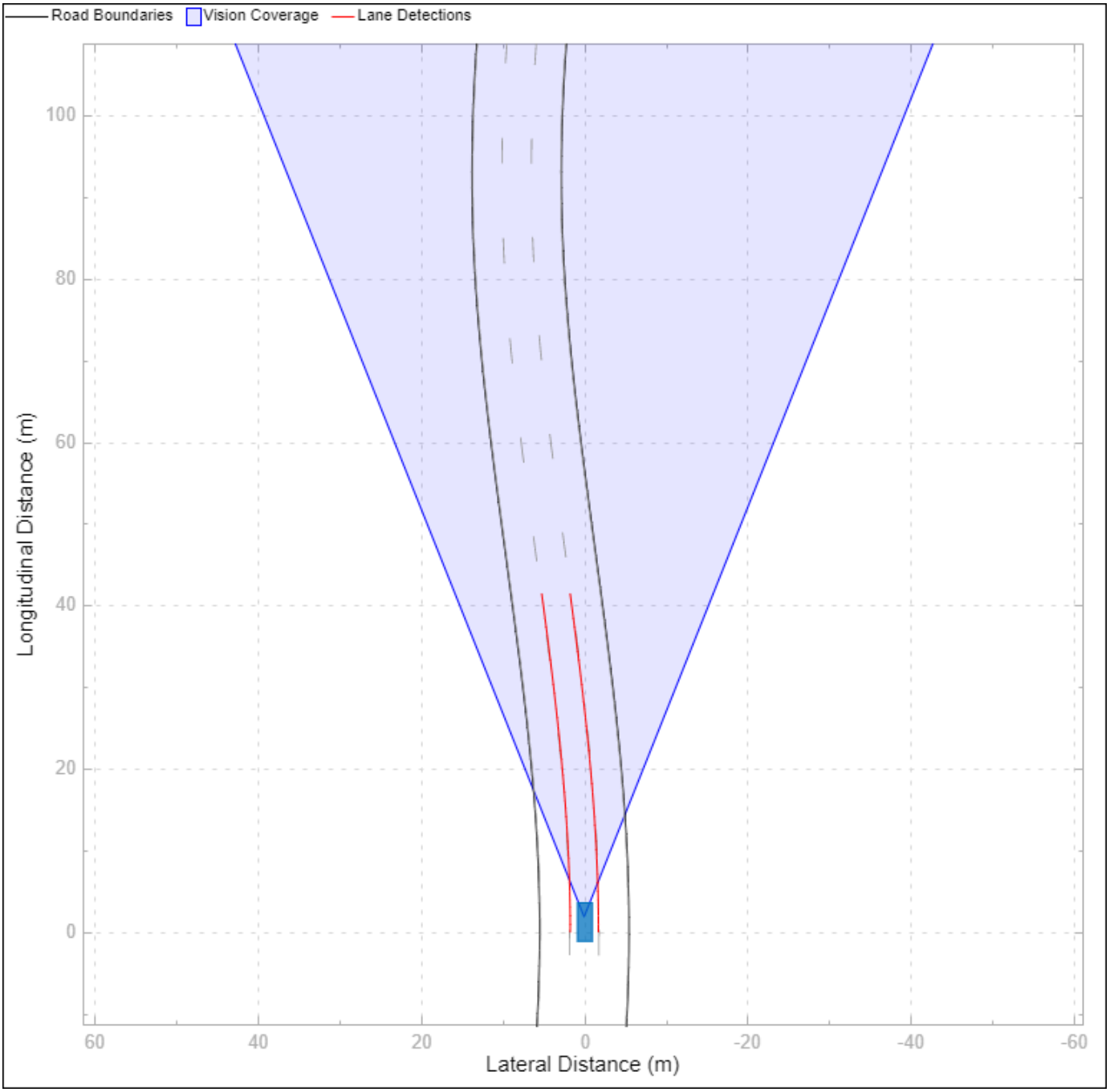
```
set_param('LKATestBenchExample/Enable','Value','1')
set_param('LKATestBenchExample/Safe Lateral Offset','Value','1')
```

To plot the results of the simulation, use the Bird's-Eye Scope. The Bird's-Eye Scope is a model-level visualization tool that you can open from the Simulink toolstrip. On the **Simulation** tab, under **Review Results**, click **Bird's-Eye Scope**. After opening the scope, click **Find Signals** to set up the signals. Then run the simulation for 15 seconds, and explore the contents of the Bird's-Eye Scope.

```
sim('LKATestBenchExample','StopTime','15') % Simulate 15 seconds
```

```
   Assuming no disturbance added to measured output channel #1.
-->Assuming output disturbance added to measured output channel #2 is integrated white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured

ans =

  Simulink.SimulationOutput:
                logsout: [1x1 Simulink.SimulationData.Dataset]
                   tout: [4681x1 double]
```

```
SimulationMetadata: [1x1 Simulink.SimulationMetadata]
      ErrorMessage: [0x0 char]
```

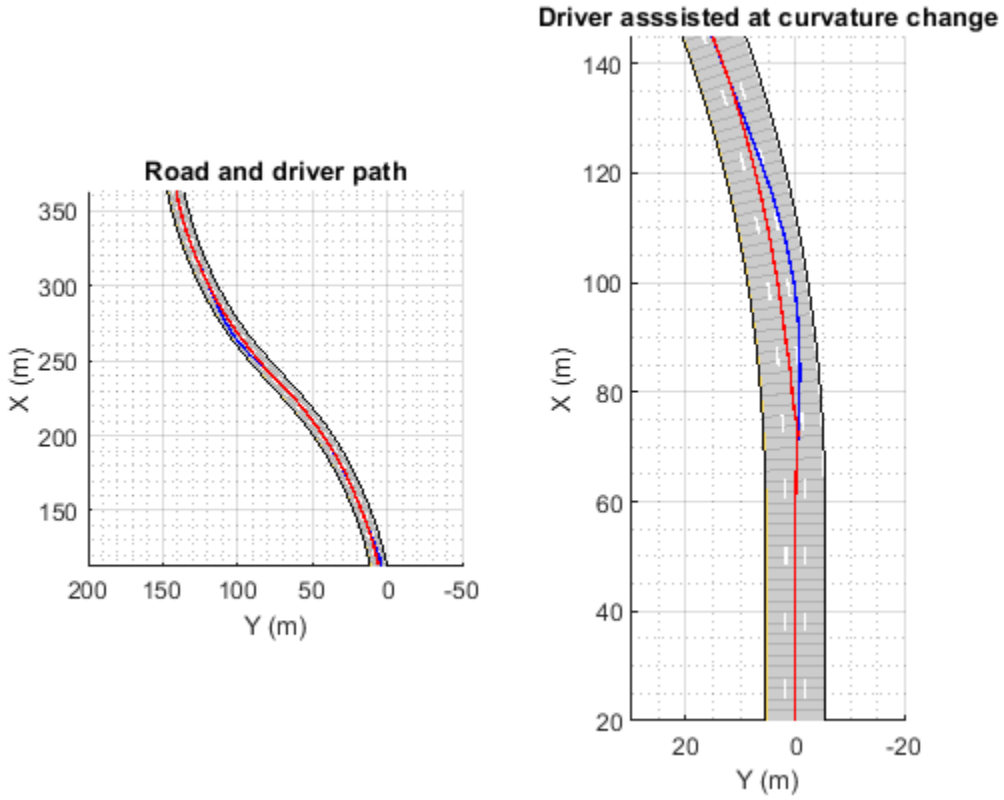


The Bird's-Eye Scope shows a symbolic representation of the road from the perspective of the ego vehicle. In this example, the Bird's-Eye Scope renders the coverage area of the synthetic vision detector as a shaded area. The ideal lane markings are additionally shown, as well as the synthetically detected left and right lane boundaries (shown here in red).

To run the full simulation and explore the results, use the following commands.

```
sim('LKATestBenchExample')                  % Simulate to end of scenario
plotLKAResults(scenario,logsout,driverPath)
```

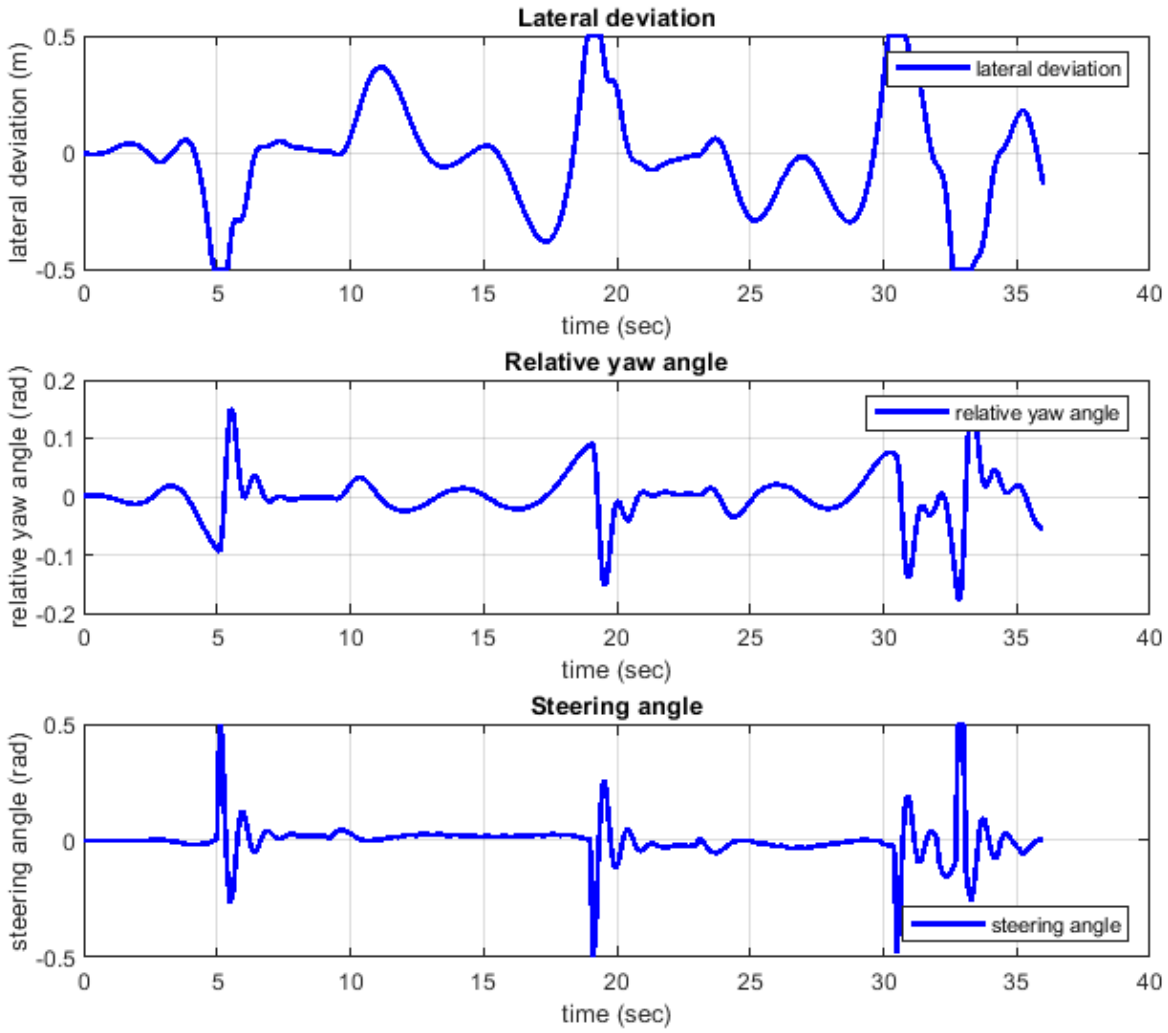
```
   Assuming no disturbance added to measured output channel #1.
-->Assuming output disturbance added to measured output channel #2 is integrated white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```



The blue curve for the driver path shows that the distracted driver may drive the ego vehicle to another lane when the road curvature changes. The red curve for the driver with Lane Keeping Assist shows that the ego vehicle remains in its lane when the road curvature changes.

To plot the controller performance, use the following command.

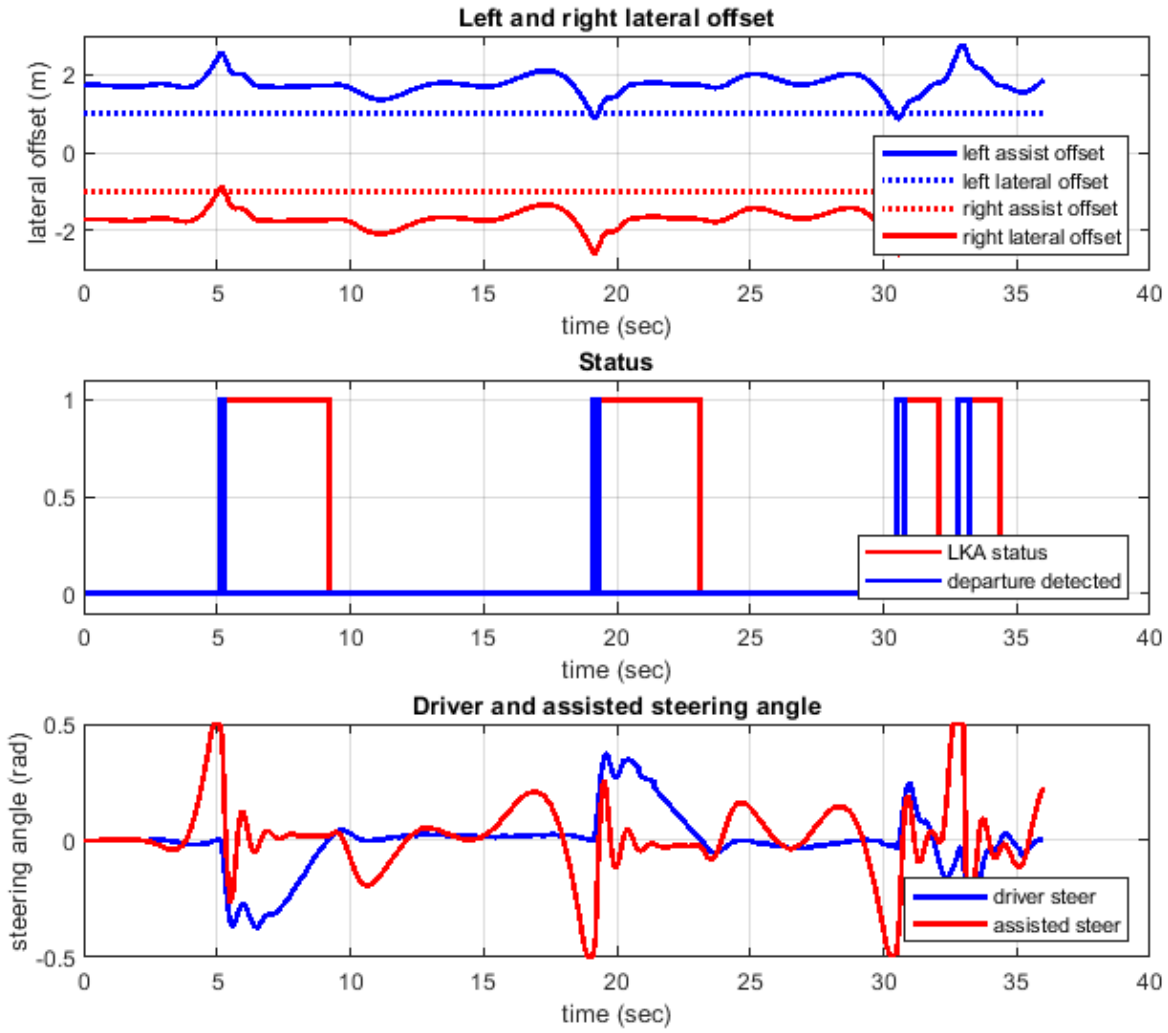
```
plotLKAPerformance(logsout)
```



- Top plot shows the lateral deviation relative to ego vehicle. The lateral deviation with LKA is within [-0.5,0.5] m.
- Middle plot shows the relative yaw angle. The relative yaw angle with LKA is within [-0.15,0.15] rad.
- Bottom plot shows the steering angle of the ego vehicle. The steering angle with LKA is within [-0.5,0.5] rad.

To view the controller status, use the following command.

```
plotLKAStatus(logsout)
```



- Top plot shows the left and right lane offset. Around 5.5 s, 19 s, 31 s, and 33 s, the lateral offset is within the distance set by the lane keeping assist. When this happens, the lane departure is detected.
- Middle plot shows the LKA status and the detection of lane departure. The departure detected status is consistent with the top plot. The LKA is turned on when the lane departure is detected, but the control is returned to the driver later when the driver can steer the ego vehicle correctly.
- Bottom plot shows the steering angle from driver and LKA. When the difference between the steering angle from driver and LKA is small, the LKA releases control to driver (for example, between 9 s to 17 s).

Simulate Lane Following

You can modify the value of Safe Lateral Offset for LKA to ignore the driver input, putting the controller into a pure lane following mode. By increasing this threshold, the lateral offset is always

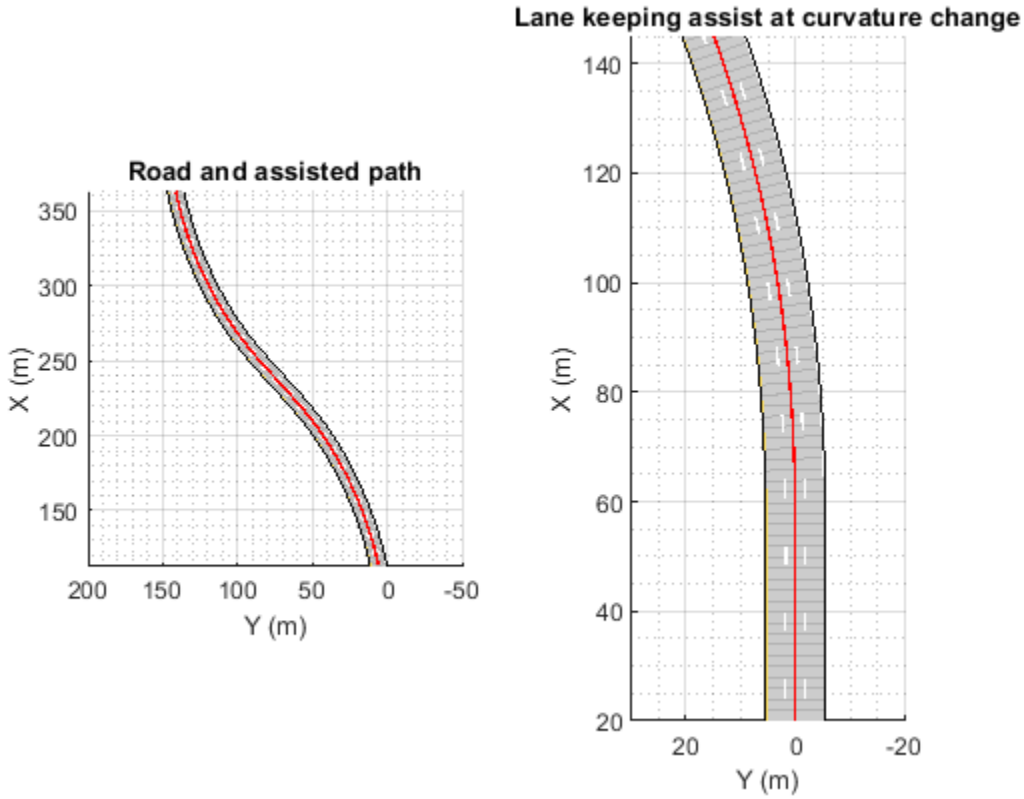
within the distance set by the lane keeping assist. Thus, the status for lane departure is on and the lane keeping assist takes control all the time.

```
set_param('LKATestBenchExample/Safe Lateral Offset','Value','2')
sim('LKATestBenchExample')                                    % Simulate to end of scenario
```

```
   Assuming no disturbance added to measured output channel #1.
-->Assuming output disturbance added to measured output channel #2 is integrated white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

You can explore the results of the simulation using the following commands.

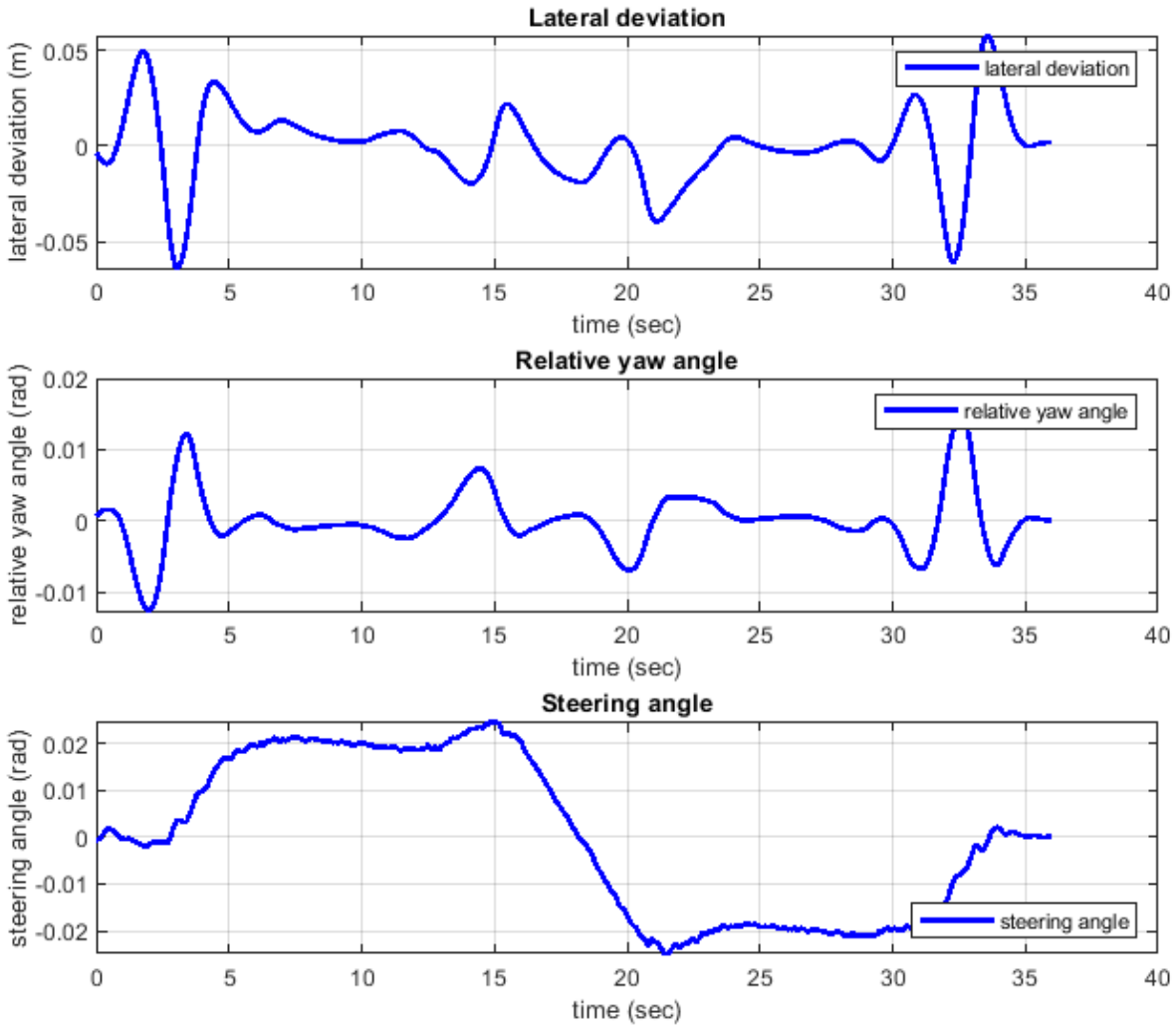
```
plotLKAResults(scenario,logsout)
```



The red curve shows that the Lane Keeping Assist on its own can keep the ego vehicle travelling along the centerline of its lane.

Use the following command to depict the controller performance.

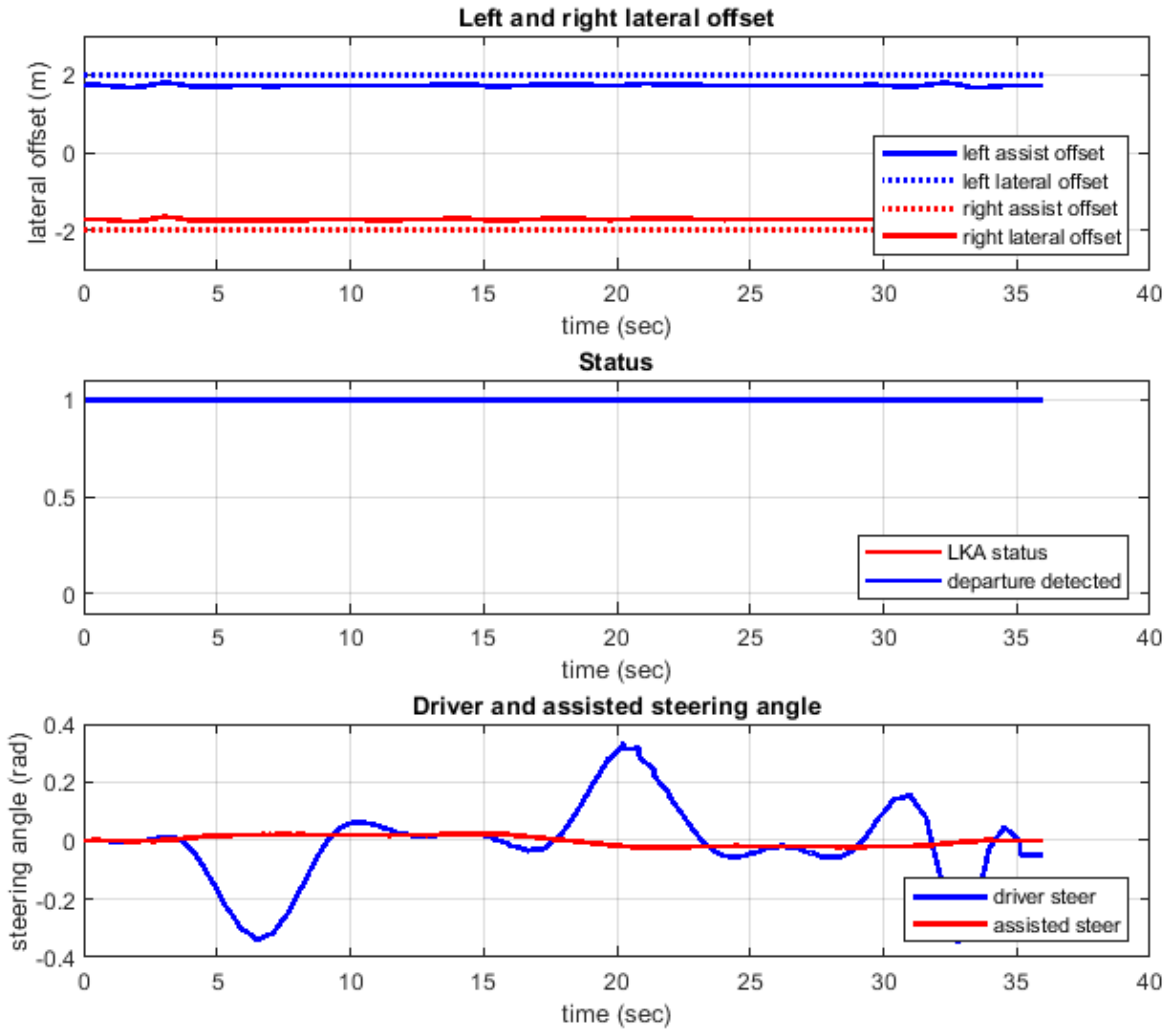
```
plotLKAPerformance(logsout)
```



- Top plot shows the lateral deviation relative to ego vehicle. The lateral deviation with LKA is within [-0.1,0.1] m.
- Middle plot shows the relative yaw angle. The relative yaw angle with LKA is within [-0.02,0.02] rad.
- Bottom plot shows the steering angle of the ego vehicle. The steering angle with LKA is within [-0.04,0.04] rad.

To view the controller status, use the following command.

```
plotLKAStatus(logsout)
```

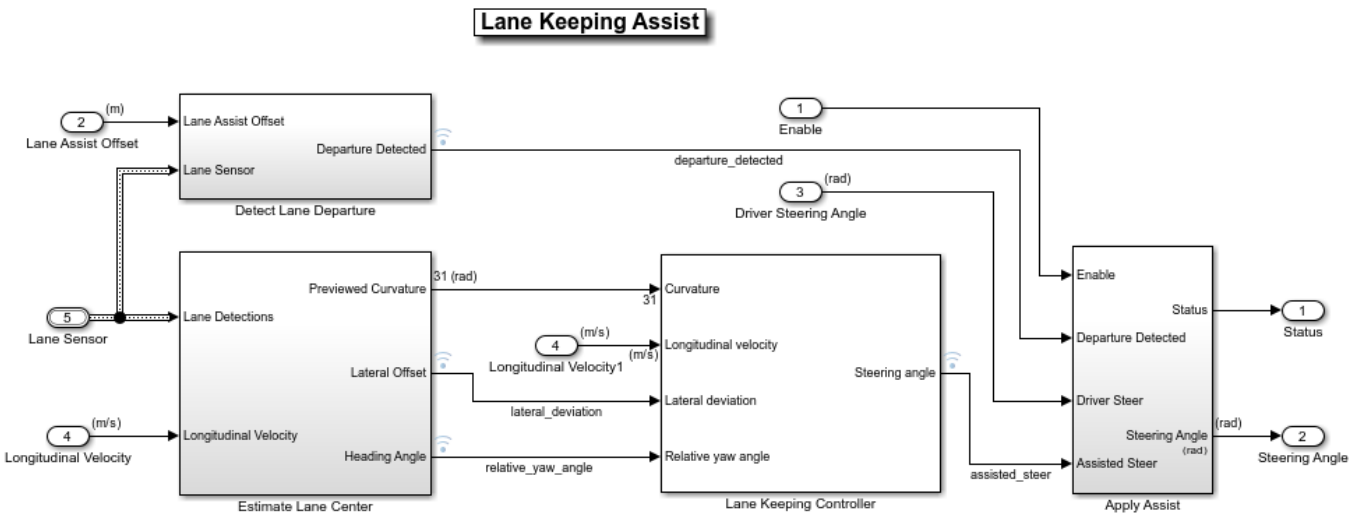


- Top plot shows the left and right lane offset. Since the lateral offset is never within the distance set by the lane keeping assist, the lane departure is not detected.
- Middle plot shows that the LKA status is always one, that is, the Lane Keeping Assist takes control all the time.
- Bottom plot shows the steering angle from driver and LKA. The steering angle from driver negotiating with the curved road is too aggressive. The small steering angle from LKA is sufficient for the curved road in this example.

Explore Lane Keeping Assist Algorithm

The Lane Keeping Assist model contains four main parts: 1) Estimate Lane Center 2) Lane Keeping Controller 3) Detect Lane Departure, and 4) Apply Assist.

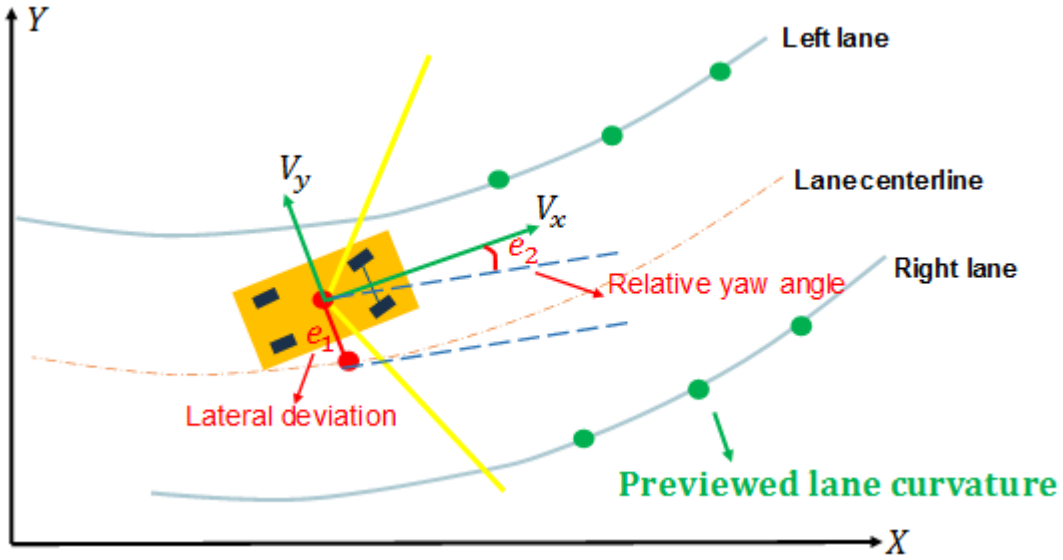
```
open_system('LKATestBenchExample/Lane Keeping Assist')
```



The Detect Lane Departure subsystem outputs a signal that is true when the vehicle is too close to a detected lane. You detect a departure when the offset between the vehicle and lane boundary from the Lane Sensor is less than the Lane Assist Offset input.

The Estimate Lane Center subsystem outputs the data from lane sensors to the lane keeping controller. The detector in this example is configured to report the left and right lane boundaries of the current lane in the current field-of-view of the camera. Each boundary is modeled as a length of a curve whose curvature varies linearly with distance (clothoid curve). To feed this data to a controller, offset both of the detected curves toward the center of the lane by the width of the car and a small margin (1.8 m total). Weight each of the resulting centered curves by the strength of the detection and pass the averaged result to the controller. Also, The Estimate Lane Center subsystem provides finite values for inputs to the Lane Keeping Controller subsystem. The previewed curvature provides the centerline of lane curvature ahead of the ego vehicle. In this example, the ego vehicle can look ahead for three seconds, which is the product of the prediction horizon and sample time. This look-ahead time enables the controller to use previewed information for calculating steering angle for the ego vehicle, which improves the MPC controller performance.

The goal for the Lane Keeping Controller block is to keep the vehicle in its lane and follow the curved road by controlling the front steering angle δ. This goal is achieved by driving the lateral deviation e_1 and the relative yaw angle e_2 to be small (see the following figure).



The LKA controller calculates a steering angle for the ego vehicle based on the following inputs:

- Previewed curvature (derived from Lane Detections)
- Ego vehicle longitudinal velocity
- Lateral deviation (derived from Lane Detections)
- Relative yaw angle (derived from Lane Detections)

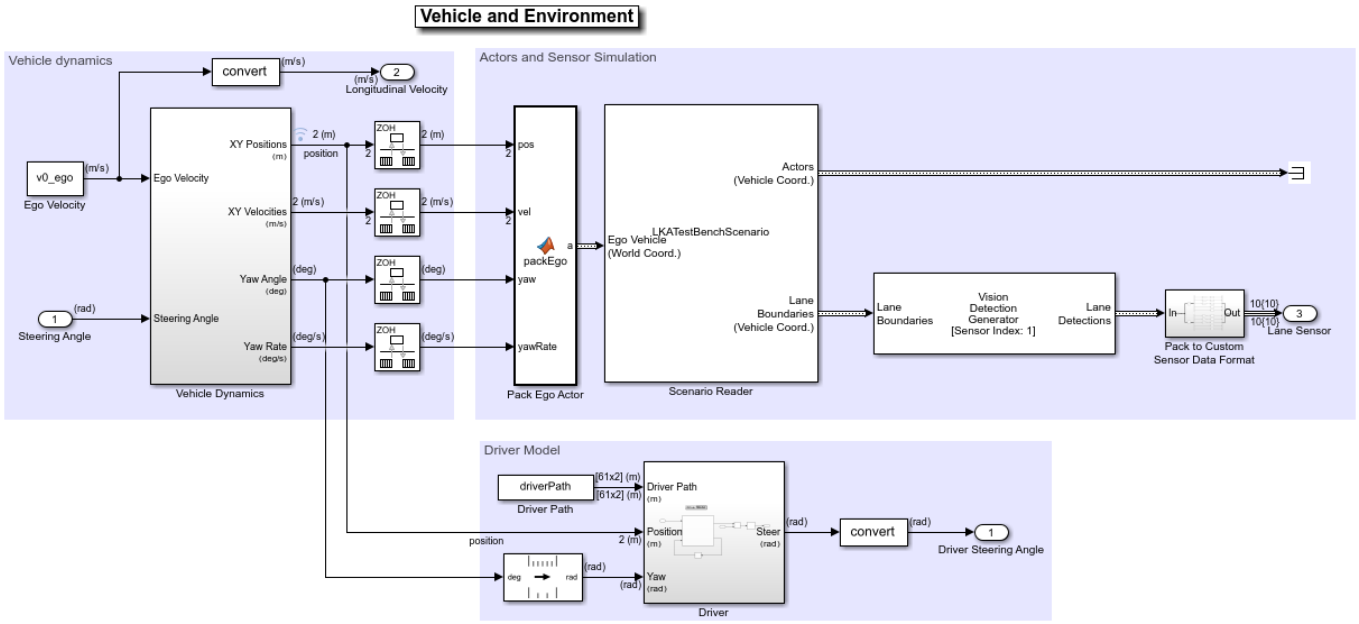
Considering physical limitations of the ego vehicle, the steering angle is constrained to be within [-0.5,0.5] rad. You can change the prediction horizon or move the **Controller Behavior** slider to adjust the performance of the controller.

The Apply Assist subsystem decides if the lane keeping controller or the driver takes control of the ego vehicle. The subsystem switches between the driver commanded steering and the assisted steering from the Lane Keeping Controller. The switch to assisted steering is initiated when a lane departure is detected. Control is returned to the driver when the driver begins steering within the lane again.

Explore Vehicle and Environment

The Vehicle and Environment subsystem enables closed loop simulation of the lane keeping assist controller.

```
open_system('LKATestBenchExample/Vehicle and Environment')
```



The Vehicle Dynamics subsystem models the vehicle dynamics with Vehicle Body 3DOF Single Track block from Vehicle Dynamics Blockset™.

The Scenario Reader block generates the ideal left and right lane boundaries based on the position of the vehicle with respect to the scenario read from scenario file `LKATestBenchScenario.mat`.

The Vision Detection Generator block takes the ideal lane boundaries from the Scenario Reader block. The detection generator models the field of view of a monocular camera and determines the heading angle, curvature, curvature derivative, and valid length of each road boundary, accounting for any other obstacles.

The Driver subsystem generates the driver steering angle based on the driver path which was created in `helperLKASetUp`.

Generate Code for the Control Algorithm

The `LKARefMdl` model is configured to support generating C code using Embedded Coder software. To check if you have access to Embedded Coder, run:

```
hasEmbeddedCoderLicense = license('checkout','RTW_Embedded_Coder')
```

You can generate a C function for the model and explore the code generation report by running:

```
if hasEmbeddedCoderLicense
    rtwbuild('LKARefMdl')
end
```

You can verify that the compiled C code behaves as expected using software-in-the-loop (SIL) simulation. To simulate the `LKARefMdl` referenced model in SIL mode, use:

```
if hasEmbeddedCoderLicense
    set_param('LKATestBenchExample/Lane Keeping Assist',...
        'SimulationMode','Software-in-the-loop (SIL)')
end
```

When you run the `LKATestBenchExample` model, code is generated, compiled, and executed for the `LKARefMdl` model. This enables you to test the behavior of the compiled code through simulation.

Conclusions

This example shows how to implement an integrated lane keeping assist (LKA) controller on a curved road with lane detection. It also shows how to test the controller in Simulink using synthetic data generated by the Automated Driving Toolbox, componentize it, and automatically generate code for it.

```
close all
bdclose all
```

See Also

Apps
Bird's-Eye Scope

Blocks
Vehicle Body 3DOF | Vision Detection Generator

More About

- “Lane Keeping Assist System Using Model Predictive Control” (Model Predictive Control Toolbox)
- “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” on page 7-205
- “Automated Driving Using Model Predictive Control” (Model Predictive Control Toolbox)

Model Radar Sensor Detections

This example shows how to model and simulate the output of an automotive radar sensor for different driving scenarios. Generating synthetic radar detections is important for testing and validating tracking and sensor fusion algorithms in corner cases or when sensor hardware is unavailable. This example analyzes the differences between radar measurements and the vehicle ground truth position and velocity for a forward collision warning (FCW) scenario, a passing vehicle scenario, and a scenario with closely spaced targets. It also includes a comparison of signal-to-noise ratio (SNR) values between pedestrian and vehicle targets at various ranges.

In this example, you generate radar detections programmatically. You can also generate detections by using the Driving Scenario Designer app. For an example, see “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2.

Introduction

Vehicles that contain advanced driver assistance system (ADAS) features or are designed to be fully autonomous typically rely on multiple types of sensors. These sensors include sonar, radar, lidar, and vision. A robust solution includes a sensor fusion algorithm to combine the strengths across the various types of sensors included in the system. For more information about sensor fusion of synthetic detections from a multisensor ADAS system, see “Sensor Fusion Using Synthetic Radar and Vision Data” on page 7-196.

When using synthetic detections for testing and validating tracking and sensor fusion algorithms, it is important to understand how the generated detections model the sensor's unique performance characteristics. Each kind of automotive sensor provides a specific set of strengths and weaknesses which contribute to the fused solution. This example presents some important performance characteristics of automotive radars and shows how the radar performance is modeled by using synthetic detections.

Radar Sensor Model

This example uses `radarDetectionGenerator` to generate synthetic radar detections. `radarDetectionGenerator` models the following performance characteristics of automotive radar:

Strengths

- Good range and range-rate accuracy over long detection ranges
- Long detection range for vehicles

Weaknesses

- Poor position and velocity accuracy along the cross-range dimension
- Shorter detection range for pedestrians and other nonmetallic objects
- Close range detection clusters pose a challenge to tracking algorithms
- Inability to resolve closely spaced targets at long ranges

FCW Driving Scenario

Create a forward collision warning (FCW) test scenario, which is used to illustrate how to measure a target's position with a typical long-range automotive radar. The scenario consists of a moving ego vehicle and a stationary target vehicle placed 150 meters down the road. The ego vehicle has an

initial speed of 50 kph before applying its brakes to achieve a constant deceleration of 3 m/s^2. The vehicle then comes to a complete stop 1 meter before the target vehicle's rear bumper.

```
addpath(fullfile(matlabroot,'toolbox','shared','tracking','fusionlib'));

rng default;
initialDist = 150; % m
initialSpeed = 50; % kph
brakeAccel = 3;    % m/s^2
finalDist = 1;     % m
[scenario, egoCar] = helperCreateSensorDemoScenario('FCW', initialDist, initialSpeed, brakeAccel
```

Forward-Facing Long-Range Radar

Create a forward-facing long-range radar sensor mounted on the ego vehicle's front bumper, 20 cm above the ground. The sensor generates measurements every 0.1 second and has an azimuthal field of view of 20 degrees and an angle resolution of 4 degrees. Its maximum range is 150 m and its range resolution is 2.5 m. The `ActorProfiles` property specifies the physical dimensions and radar cross-section (RCS) patterns of the vehicles seen by the radar in the simulation.

```
radarSensor = radarDetectionGenerator( ...
    'SensorIndex', 1, ...
    'UpdateInterval', 0.1, ...
    'SensorLocation', [egoCar.Wheelbase+egoCar.FrontOverhang 0], ...
    'Height', 0.2, ...
    'FieldOfView', [20 5], ...
    'MaxRange', 150, ...
    'AzimuthResolution', 4, ...
    'RangeResolution', 2.5, ...
    'ActorProfiles', actorProfiles(scenario))
```

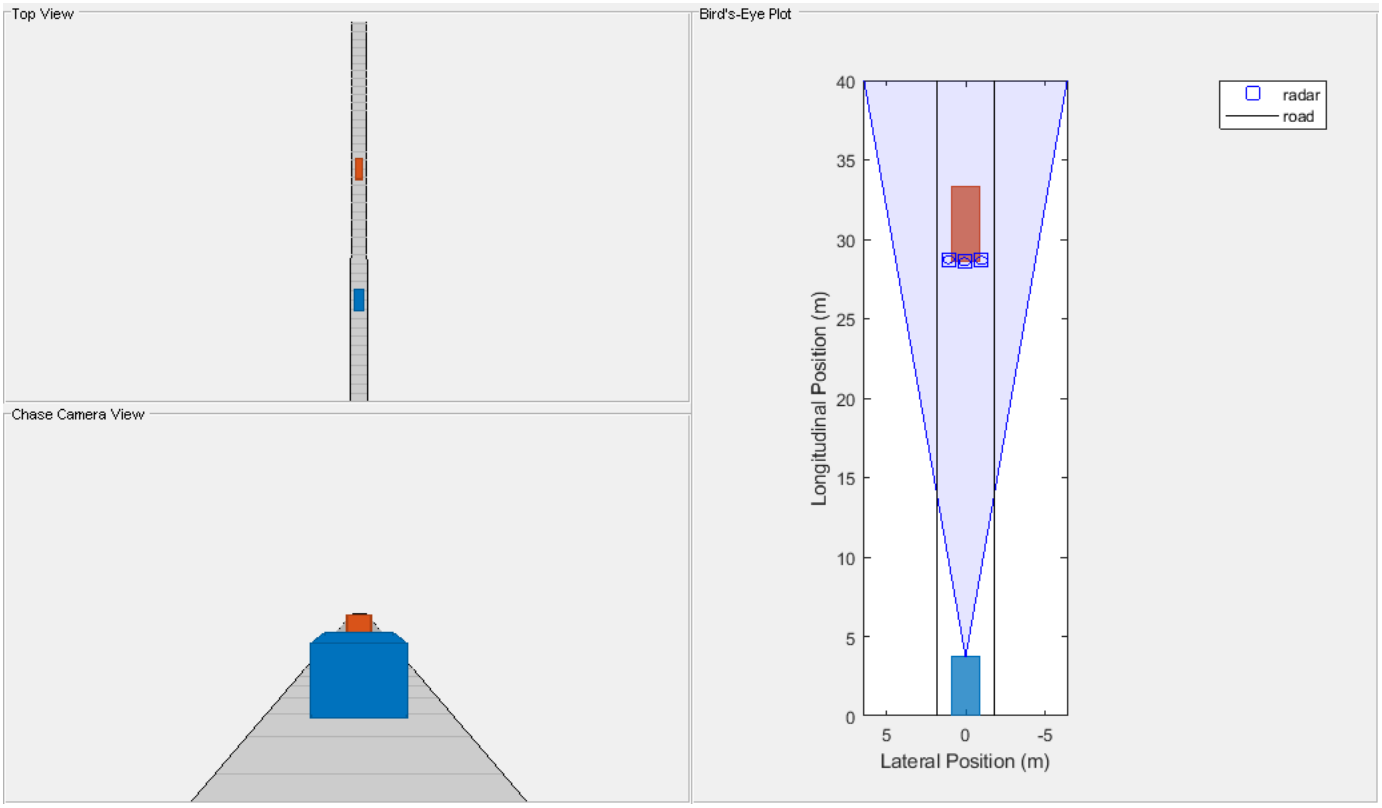
```
radarSensor =

  radarDetectionGenerator with properties:

             SensorIndex: 1
          UpdateInterval: 0.1000

          SensorLocation: [3.7000 0]
                  Height: 0.2000
                     Yaw: 0
                   Pitch: 0
                    Roll: 0

             FieldOfView: [20 5]
                MaxRange: 150
         RangeRateLimits: [-100 100]

    DetectionProbability: 0.9000
          FalseAlarmRate: 1.0000e-06

  Use get to show all properties
```

Simulation of Radar Detections

Simulate the radar measuring the position of the target vehicle by advancing the simulation time of the scenario. The radar sensor generates detections from the true target pose (position, velocity, and orientation) expressed in the ego vehicle's coordinate frame.

The radar is configured to generate detections at 0.1-second intervals, which is consistent with the update rate of typical automotive radars. However, to accurately model the motion of the vehicles, the scenario simulation advances every 0.01 seconds. The sensor returns a logical flag, `isValidTime`, that is true when the radar reaches its required update interval, indicating that this simulation time step will generate detections.

```
% Create display for FCW scenario
[bep, figScene] = helperCreateSensorDemoDisplay(scenario, egoCar, radarSensor);

metrics = struct;                % Initialize struct to collect scenario metrics
while advance(scenario)          % Update vehicle positions
    gTruth = targetPoses(egoCar); % Get target positions in ego vehicle coordinates

    % Generate time-stamped radar detections
    time = scenario.SimulationTime;
    [dets, ~, isValidTime] = radarSensor(gTruth, time);

    if isValidTime
        % Update Bird's-Eye Plot with detections and road boundaries
        helperUpdateSensorDemoDisplay(bep, egoCar, radarSensor, dets);

        % Collect radar detections and ground truth for offline analysis
        metrics = helperCollectScenarioMetrics(metrics, gTruth, dets);
    end

    % Take a snapshot for the published example
    helperPublishSnapshot(figScene, time>=9.1);
end
```

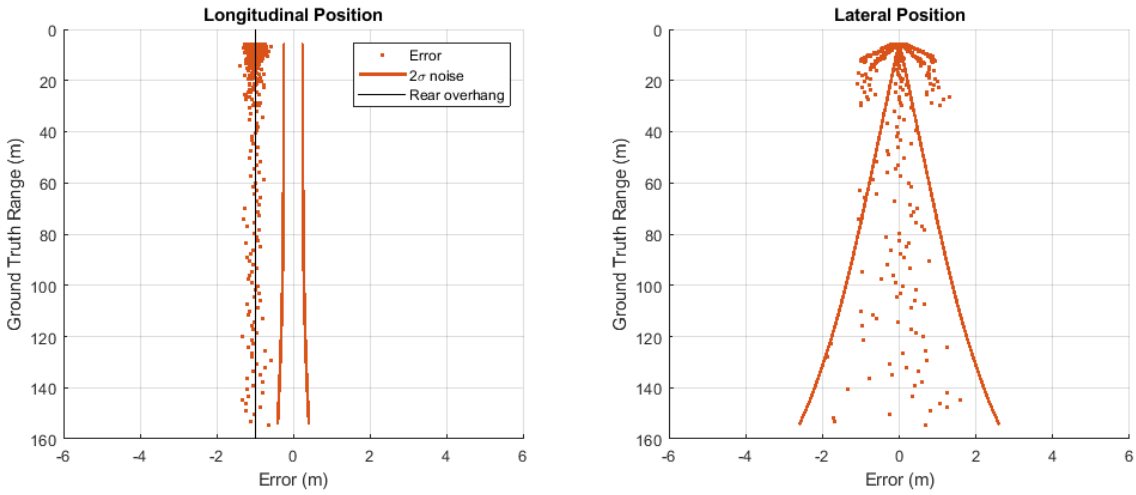


Position Measurements

Over the duration of the FCW test, the target vehicle's distance from the ego vehicle spans a wide range of values. By comparing the radar's measured longitudinal and lateral positions of the target vehicle to the vehicle's ground truth position, you can observe the accuracy of the radar's measured positions.

Use `helperPlotSensorDemoDetections` to plot the longitudinal and lateral position errors as the difference between the measured position reported by the radar and the target vehicle's ground truth. The ground truth reference for the target vehicle is the point on the ground directly below the center of the target vehicle's rear axle, which is 1 meter in front of the car's bumper.

```
helperPlotSensorDemoDetections(metrics, 'position', 'reverse range', [-6 6]);

% Show rear overhang of target vehicle
tgtCar = scenario.Actors(2);
rearOverhang = tgtCar.RearOverhang;

subplot(1,2,1);
hold on; plot(-rearOverhang*[1 1], ylim, 'k'); hold off;
legend('Error', '2\sigma noise', 'Rear overhang');
```



Longitudinal Position Measurements

For a forward-facing radar configuration, the radar's range measurements correspond to the longitudinal position of the target vehicle.

The longitudinal position errors in the preceding plot on the left show a -1 meter bias between the longitude measured by the radar and the target's ground truth position. This bias indicates that the radar consistently measures the target to be closer than the position reported by the ground truth. Instead of approximating the target as a single point in space, the radar models the physical dimensions of the vehicle's body. Detections are generated along the vehicle's rear side according to the radar's resolution in azimuth, range, and (when enabled) elevation. This -1 meter offset is then explained by the target vehicle's rear overhang, which defines the distance between the vehicle's rear side and its rear axle, where the ground truth reference is located.

The radar is modeled with a range resolution of 2.5 meters. However, the 2σ measurement noise is reported to be as small as 0.25 meter at the closest point and grows slightly to 0.41 meter at the farthest tested range. The realized sensor accuracy is much smaller than the radar's range resolution. Because the radar models the SNR dependence of the range errors using the Cramer-Rao lower bound, targets with a large radar cross-section (RCS) or targets that are close to the sensor will have better range accuracy than smaller or more distant targets.

This SNR dependence on the radar's measurement noise is modeled for each of the radar's measured dimensions: azimuth, elevation, range, and range rate.

Lateral Position Measurements

For a forward-facing radar configuration, the dimension orthogonal to the radar's range measurements (commonly referred to as the sensor's cross-range dimension) corresponds to the lateral position of the target vehicle.

The lateral position errors from the FCW test in the preceding plot on the right show a strong dependence on the target's ground truth range. The radar reports lateral position accuracies as small as 0.03 meters at close ranges and up to 2.6 meters when the target is far from the radar.

Additionally, multiple detections appear when the target is at ranges less than 30 meters. As previously mentioned, the target vehicle is not modeled as a single point in space, but the radar model compares the vehicle's dimensions with the radar's resolution. In this scenario, the radar views

the rear side of the target vehicle. When the vehicle's rear side spans more than one of the radar's azimuth resolution cells, the radar generates detections from each resolution cell that the target occupies.

Compute the azimuth spanned by the target vehicle in the FCW test when it is at 30 meters ground truth range from the ego vehicle.

```
% Range from radar to target vehicle's rear side
radarRange = 30-(radarSensor.SensorLocation(1)+tgtCar.RearOverhang);

% Azimuth spanned by vehicle's rear side at 30 meters ground truth range
width = tgtCar.Width;
azSpan = rad2deg(width/radarRange)
```

```
azSpan =

    4.0764
```

At a ground truth range of 30 meters, the vehicle's rear side begins to span an azimuth greater than the radar's azimuth resolution of 4 degrees. Because the azimuth spanned by the target's rear side exceeds the sensor's resolution, 3 resolved points along the vehicle's rear side are generated: one from the center of the rear side, one from the left edge of the rear side, and one from the right edge.

Velocity Measurements

Create a driving scenario with two target vehicles (a lead car and a passing car) to illustrate the accuracy of a radar's longitudinal and lateral velocity measurements. The lead car is placed 40 meters in front of the ego vehicle and is traveling with the same speed. The passing car starts in the left lane alongside the ego vehicle, passes the ego vehicle, and merges into the right lane just behind the lead car. This merging maneuver generates longitudinal and lateral velocity components, enabling you to compare the sensor's accuracy along these two dimensions.

Because the lead car is directly in front of the radar, it has a purely longitudinal velocity component. The passing car has a velocity profile with both longitudinal and lateral velocity components. These components change as the car passes the ego vehicle and moves into the right lane behind the lead car. Comparing the radar's measured longitudinal and lateral velocities of the target vehicles to their ground truth velocities illustrates the radar's ability to observe both of these velocity components.

```
% Create passing scenario
leadDist = 40;  % m
speed = 50;     % kph
passSpeed = 70; % kph
[scenario, egoCar] = helperCreateSensorDemoScenario('Passing', leadDist, speed, passSpeed);
```

Configuration of Radar Velocity Measurements

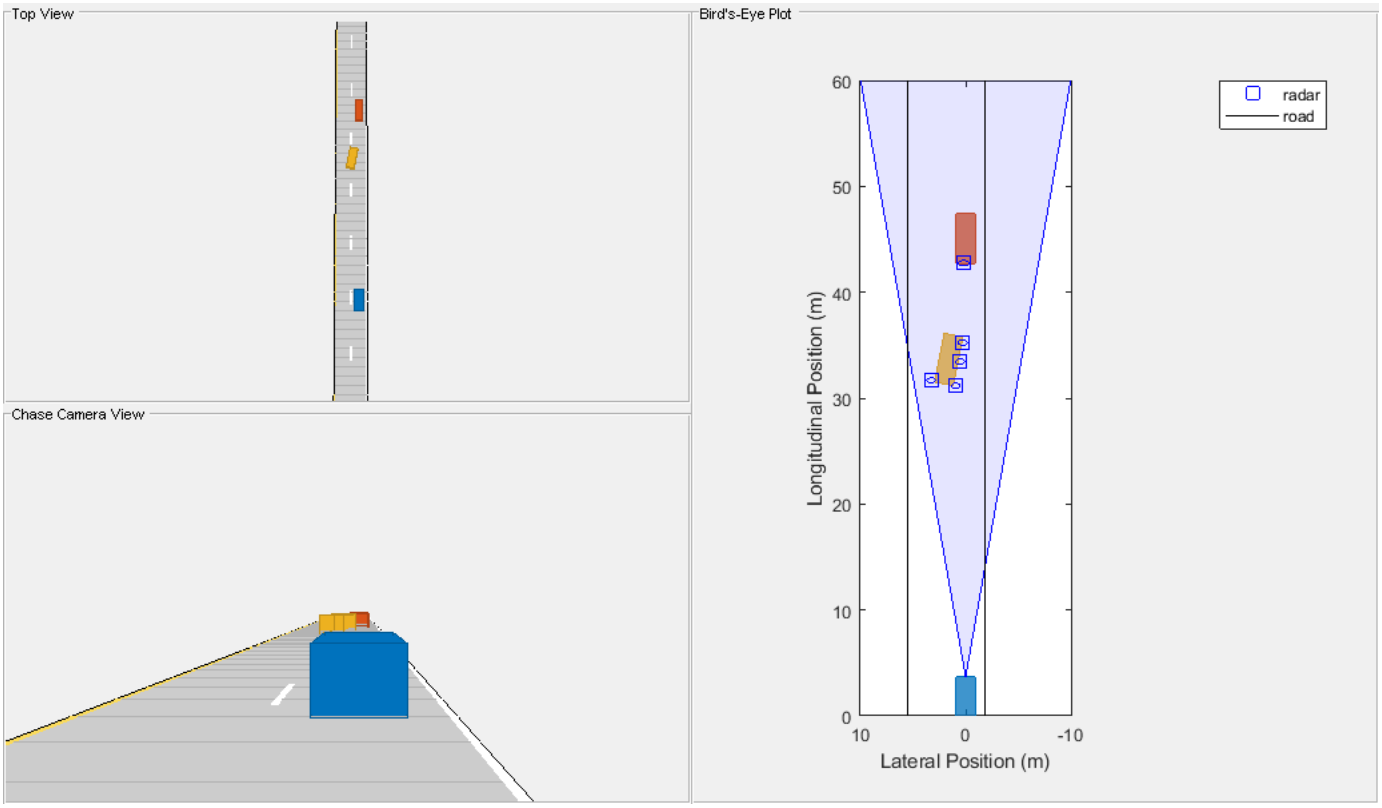
A radar generates velocity measurements by observing the Doppler frequency shift on the signal energy returned from each target. The rate at which the target's range is changing relative to the radar is derived directly from these Doppler frequencies. Take the radar sensor used in the previous section to measure position, and configure it to generate range-rate measurements. These measurements have a resolution of 0.5 m/s, which is a typical resolution for an automotive radar.

```
% Configure radar for range-rate measurements
release(radarSensor);
```

```
radarSensor.HasRangeRate = true;
radarSensor.RangeRateResolution = 0.5; % m/s

% Use actor profiles for the passing car scenario
radarSensor.ActorProfiles = actorProfiles(scenario);
```

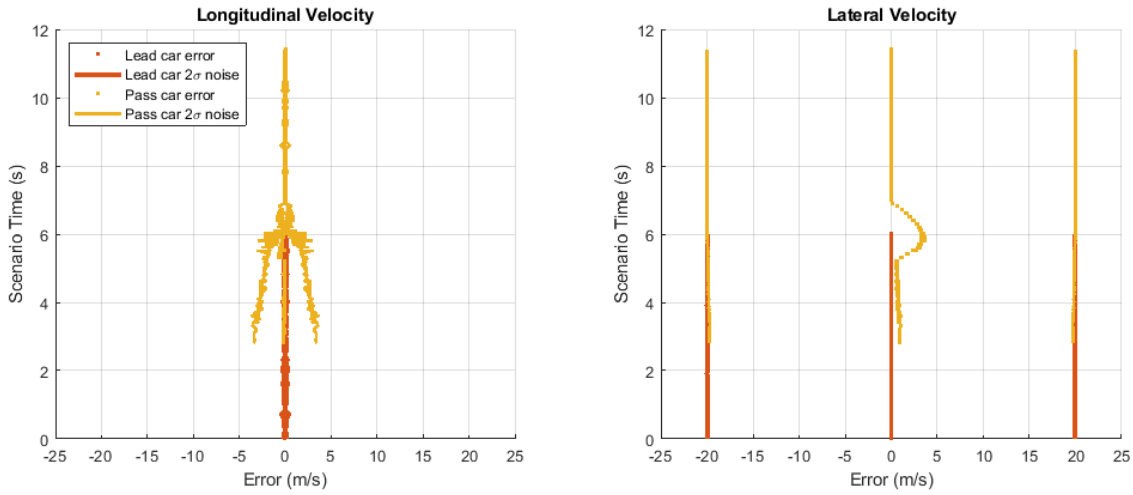
Use `helperRunSensorDemoScenario` to simulate the motion of the ego and target vehicles. This function also collects the simulated metrics, as was previously done for the FCW driving scenario.

```
snapTime = 6; % Simulation time to take snapshot for publishing
metrics = helperRunSensorDemoScenario(scenario, egoCar, radarSensor, snapTime);
```



Use `helperPlotSensorDemoDetections` to plot the radar's longitudinal and lateral velocity errors as the difference between the measured velocity reported by the radar and the target vehicle's ground truth.

```
helperPlotSensorDemoDetections(metrics, 'velocity', 'time', [-25 25]);
subplot(1,2,1);
legend('Lead car error', 'Lead car 2\sigma noise', ...
    'Pass car error', 'Pass car 2\sigma noise', 'Location', 'northwest');
```



Longitudinal Velocity Measurements

For a forward-facing radar, longitudinal velocity is closely aligned to the radar's range-rate measurements. The preceding plot on the left shows the radar's longitudinal velocity errors for the passing vehicle scenario. Because the radar can accurately measure longitudinal velocity from the Doppler frequency shift observed in the signal energy received from both cars, the velocity errors for both vehicles (shown as points) are small. However, when the passing car enters the radar's field of view at 3 seconds, the passing car's 2σ measurement noise (shown using solid yellow lines) is initially large. The noise then decreases until the car merges into the right lane behind the lead car at 7 seconds. As the car passes the ego vehicle, the longitudinal velocity of the passing car includes both radial and nonradial components. The radar inflates its reported 2σ longitudinal velocity noise to indicate its inability to observe the passing car's nonradial velocity components as it passes the ego vehicle.

Lateral Velocity Measurements

For a forward-facing radar, the measured lateral velocity corresponds to a target's nonradial velocity component. The preceding plot on the right shows the passing car's lateral velocity measurement errors, which display as yellow points. The radar's inability to measure lateral velocity produces a large error during the passing car's lane change maneuver between 5 and 7 seconds. However, the radar reports a large 2σ lateral velocity noise (shown as solid lines) to indicate that it is unable to observe velocity along the lateral dimension.

Pedestrian and Vehicle Detection

A radar "sees" not only an object's physical dimensions (length, width, and height) but also is sensitive to an object's *electrical* size. An object's electrical size is referred to as its radar cross-section (RCS) and is commonly given in units of decibel square meters (dBsm). An object's RCS defines how effectively it reflects the electromagnetic energy received from the radar back to the sensor. An object's RCS value depends on many properties, including the object's size, shape, and the kind of materials it contains. An object's RCS also depends on the transmit frequency of the radar. This value can be large for vehicles and other metallic objects. For typical automotive radar frequencies near 77 GHz, a car has a nominal RCS of approximately 10 square meters (10 dBsm). However, nonmetallic objects typically have much smaller values. -8 dBsm is a reasonable RCS to associate with a pedestrian. This value corresponds to an effective electrical size of only 0.16 square

meters. In an ADAS or autonomous driving system, a radar must be able to generate detections on both of these objects.

FCW Driving Scenario with a Pedestrian and a Vehicle

Revisit the FCW scenario from earlier by adding a pedestrian standing on the sidewalk beside the stopped vehicle. Over the duration of the FCW test, the distance from the radar to the target vehicle and pedestrian spans a wide range of values. Comparing the radar's measured signal-to-noise ratio (SNR) reported for the test vehicle and pedestrian detections across the tested ranges demonstrates how the radar's detection performance changes with both detection range and object type.

```
% Create FCW test scenario
initialDist = 150;  % m
finalDist = 1;      % m
initialSpeed = 50;  % kph
brakeAccel = 3;     % m/s^2
withPedestrian = true;
[scenario, egoCar] = helperCreateSensorDemoScenario('FCW', initialDist, initialSpeed, brakeAccel
```

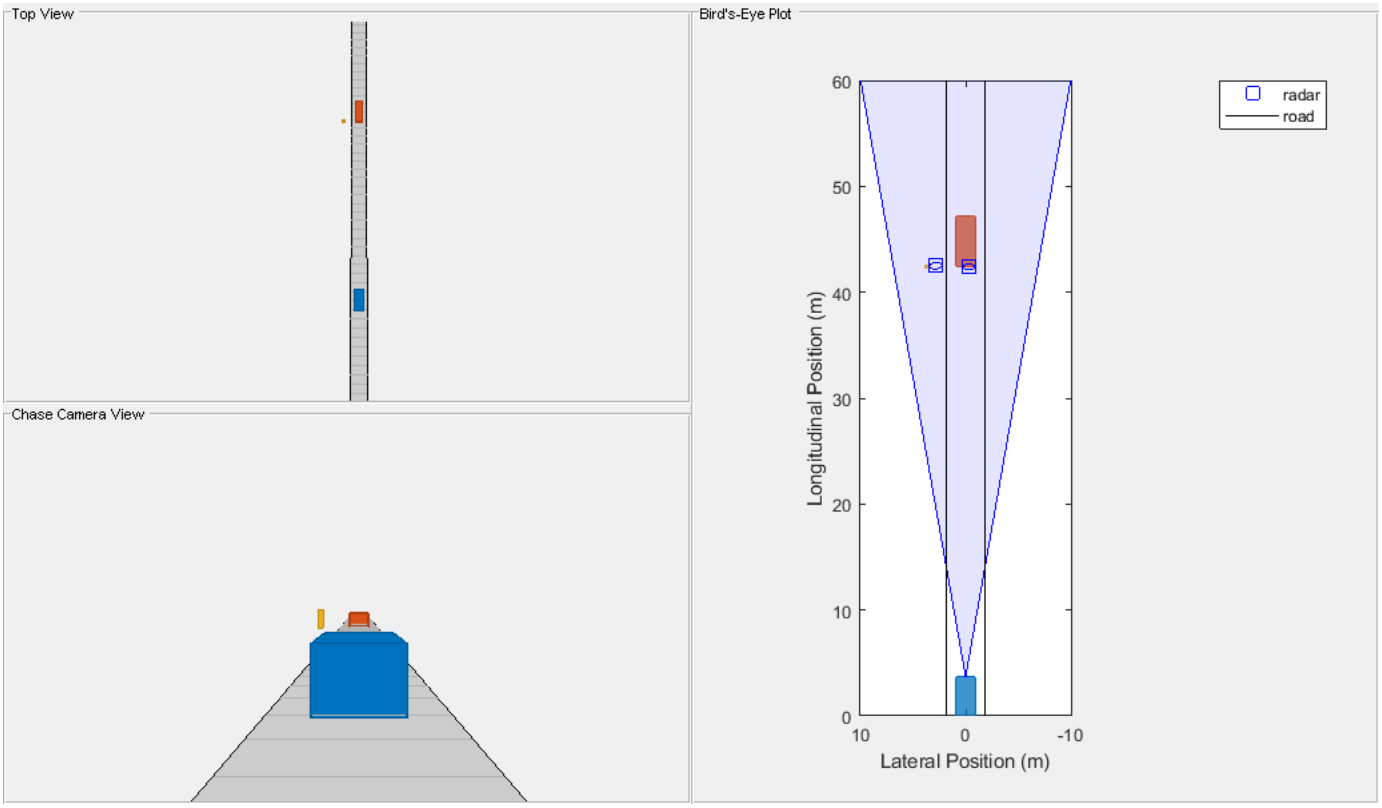
Configuration of Radar Detection Performance

A radar's detection performance is usually specified by the probability of detecting a reference target that has an RCS of 0 dBsm at a specific range. Create a long-range radar that detects a target with an RCS of 0 dBsm at a range of 100 meters, with a detection probability of 90%.

```
% Configure radar's long-range detection performance
release(radarSensor);
radarSensor.ReferenceRange = 100; % m
radarSensor.ReferenceRCS = 0;     % dBsm
radarSensor.DetectionProbability = 0.9;

% Use actor profiles for the passing car scenario
radarSensor.ActorProfiles = actorProfiles(scenario);
```

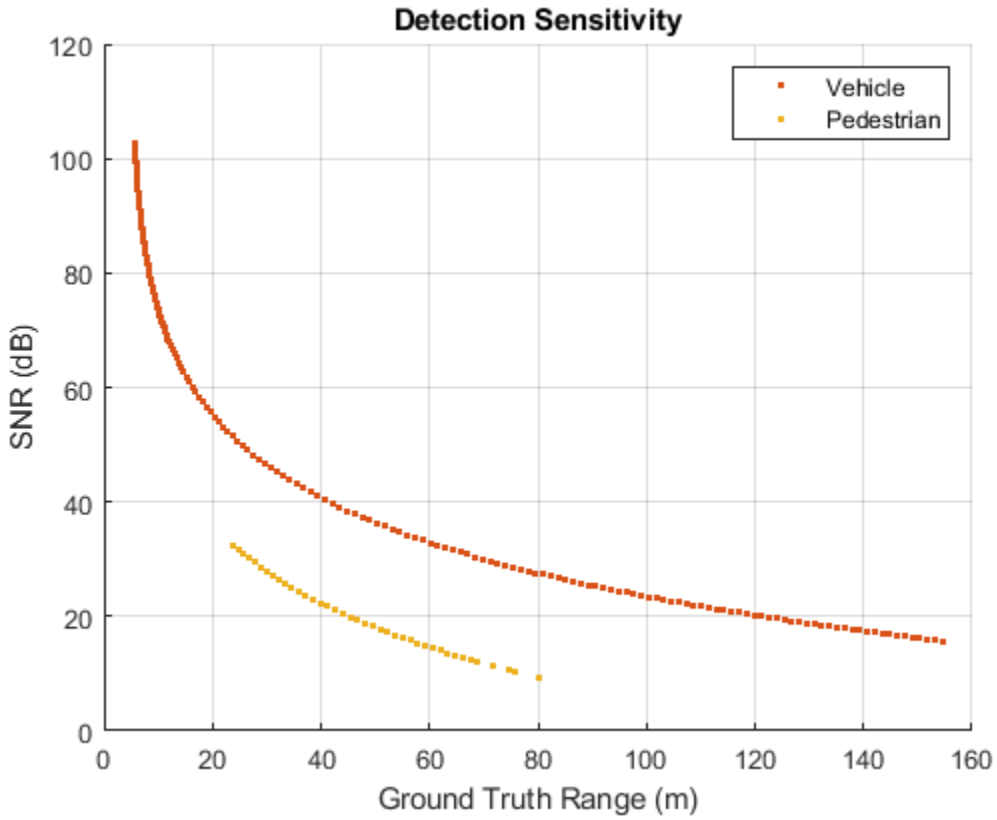
Run the scenario to collect radar detections and ground truth data. Store them for offline analysis.

```
snapTime = 8; % Simulation time to take snapshot for publishing
metrics = helperRunSensorDemoScenario(scenario, egoCar, radarSensor, snapTime);
```



Plot SNR of detections for both the target vehicle and the pedestrian.

```
helperPlotSensorDemoDetections(metrics, 'snr', 'range', [0 160]);
legend('Vehicle', 'Pedestrian');
```



This plot shows the effect of an object's RCS on the radar's ability to "see" it. Detections corresponding to the stationary test vehicle are shown in red. Detections from the pedestrian are shown in yellow.

The test vehicle is detected out to the farthest range included in this test, but detection of the pedestrian becomes less consistent near 70 meters. This difference between the detection range of the two objects occurs because the test vehicle has a much larger RCS (10 dBsm) than the pedestrian (-8 dBsm), which enables the radar to detect the vehicle at longer ranges than the pedestrian.

The test vehicle is also detected at the closest range included in this test, but the radar stops generating detections on the pedestrian near 20 meters. In this scenario, the target vehicle is placed directly in front of the radar, but the pedestrian is offset from the radar's line of sight. Near 20 meters, the pedestrian is no longer inside of the radar's field of view and cannot be detected by the radar.

Revisit this scenario for a mid-range automotive radar to illustrate how the radar's detection performance is affected. Model a mid-range radar to detect an object with an RCS of 0 dBsm at a reference range of 50 meters, with a detection probability of 90%.

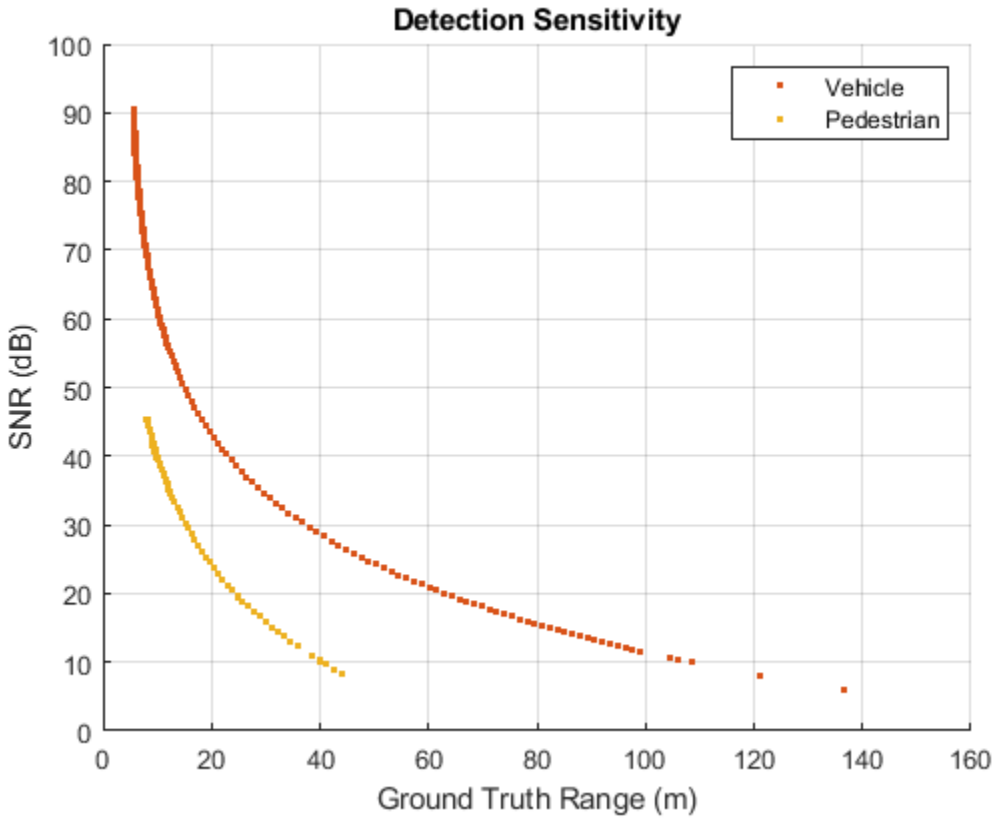
```
% Configure radar for a mid-range detection requirement
release(radarSensor);
radarSensor.ReferenceRange = 50; % m
radarSensor.ReferenceRCS = 0;    % dBsm
radarSensor.DetectionProbability = 0.9;
```

Additionally, to improve the detection of objects at close ranges that are offset from the radar's line of sight, the mid-range radar's azimuthal field of view is increased to 90 degrees. The radar's azimuth resolution is set to 10 degrees to search this large coverage area more quickly.

```
% Increase radar's field of view in azimuth and elevation to 90 and 10 degrees respectively
radarSensor.FieldOfView = [90 10];

% Increase radar's azimuth resolution
radarSensor.AzimuthResolution = 10;
```

Run the FCW test using the mid-range radar and the SNR for the detections from the target vehicle and pedestrian. Plot the SNR.

```
% Run simulation and collect detections and ground truth for offline analysis
metrics = helperRunSensorDemoScenario(scenario, egoCar, radarSensor);

% Plot SNR for vehicle and pedestrian detections
helperPlotSensorDemoDetections(metrics, 'snr', 'range', [0 160]);
legend('Vehicle', 'Pedestrian');
```



For the mid-range radar, the detections of both the vehicle and pedestrian are limited to shorter ranges. With the long-range radar, the vehicle is detected out to the full test range, but now vehicle detection becomes unreliable at 95 meters. Likewise, the pedestrian is detected reliably only out to 35 meters. However, the mid-range radar's extended field of view in azimuth enables detections on the pedestrian to a 10-meter ground truth range from the sensor, a significant improvement in coverage over the long-range radar.

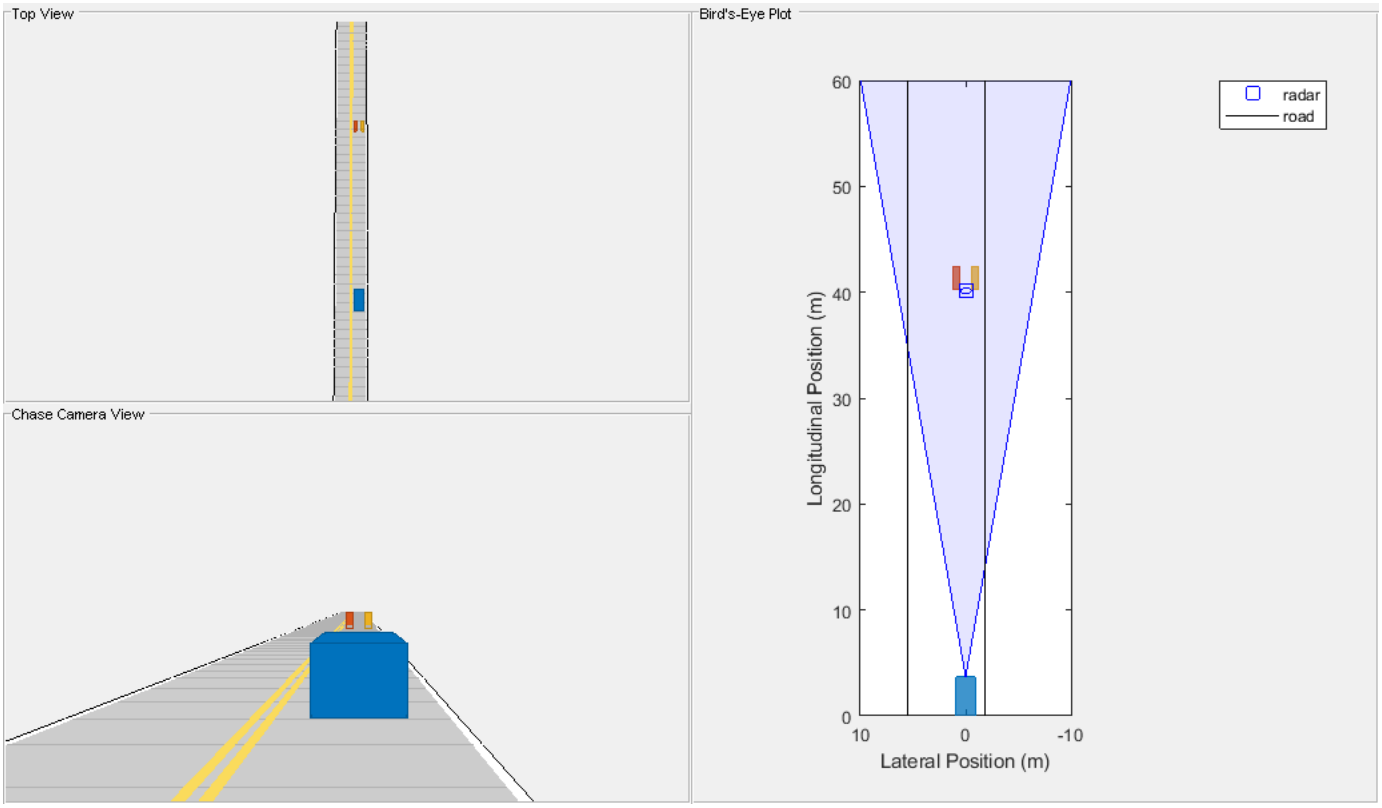
Detection of Closely Spaced Targets

When multiple targets occupy a radar's resolution cell, the group of closely spaced targets are reported as a single detection. The reported location is the centroid of the location of each contributing target. This merging of multiple targets into a single detection is common at long ranges, because the area covered by the radar's azimuth resolution grows with increasing distance from the sensor.

Create a scenario with two motorcycles traveling side-by-side in front of the ego vehicle. This scenario shows how the radar merges closely spaced targets. The motorcycles are 1.8 meters apart and are traveling 10 kph faster than the ego vehicle.

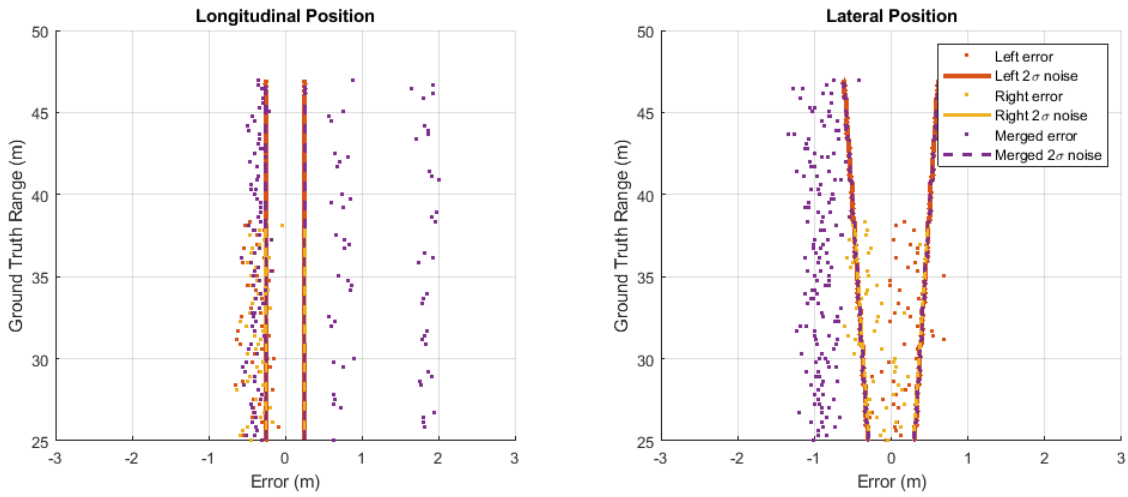
Over the duration of the scenario, the distance between the motorcycles and the ego vehicle increases. When the motorcycles are close to the radar, they occupy different radar resolution cells. By the end of the scenario, after the distance between the radar and the motorcycles has increased, both motorcycles occupy the same radar resolution cells and are merged. The radar's longitudinal and lateral position errors show when this transition occurs during the scenario.

```
duration = 8;          % s
speedEgo = 50;         % kph
speedMotorcycles = 60; % kph
distMotorcycles = 25;  % m
[scenario, egoCar] = helperCreateSensorDemoScenario('Side-by-Side', duration, speedEgo, speedMotc

% Create forward-facing long-range automotive radar sensor mounted on ego vehicle's front bumper
radarSensor = radarDetectionGenerator(...
    'SensorIndex', 1, ...
    'SensorLocation', [egoCar.Wheelbase+egoCar.FrontOverhang 0], ...
    'Height', 0.2, ...
    'ActorProfiles', actorProfiles(scenario));

% Run simulation and collect detections and ground truth for offline analysis
snapTime = 5.6; % Simulation time to take snapshot for publishing
metrics = helperRunSensorDemoScenario(scenario, egoCar, radarSensor, snapTime);
```



Plot the radar's longitudinal and lateral position errors. By analyzing the position errors reported for each motorcycle, you can identify the range where the radar no longer can distinguish the two motorcycles as unique objects.

```
helperPlotSensorDemoDetections(metrics, 'position', 'range', [-3 3], true);
subplot(1,2,2);
legend('Left error', 'Left 2\sigma noise', 'Right error', 'Right 2\sigma noise', 'Merged error',
```



Detections are generated from the rear and along the inner side of each motorcycle. The red errors are from the left motorcycle, the yellow errors are from the right motorcycle, and the purple points

show the detections that are merged between the two motorcycles. The motorcycles are separated by a distance of 1.8 meters. Each motorcycle is modeled to have a width of 0.6 meters and a length of 2.2 meters. The inner sides of the motorcycles are only 1.2 meters apart.

Inner Side Detections

Detections are generated from points along the inner side of each motorcycle. The detections start at the closest edge and are sampled in range according to the radar's range resolution of 2.5 meters and the motorcycle's position relative to the radar. When the motorcycle occupies multiple range resolution cells, the closest sample points in each unique range cell generate detections. The location of the range cell's boundary produces a detection that occurs either at the middle or far edge of the motorcycle's inner side. A detection from the motorcycle's closest edge is also generated. This movement through the radar's range resolution cell boundaries creates the 3 bands of longitudinal position errors seen in the preceding plot on the left. The total longitudinal extent covered by these 3 bands is 2.2 meters, which corresponds to the length of the motorcycles.

Because the inner sides of the motorcycles are separated by only 1.2 meters, these sampled points all fall within a common azimuthal resolution cell and are merged by the radar. The centroid of these merged points lies in the middle of the two motorcycles. The centroiding of the merged detections produces a lateral bias with a magnitude of 0.9 meters, corresponding to half of the distance between the motorcycles. In the lateral position error plot on the right, all of the merged detections (shown in purple) have this bias.

Rear Side Detections

Detections generated from the rear side of each motorcycle are further apart (1.8 m) than the sampled points along the inner sides (1.2 m).

At the beginning of the scenario, the motorcycles are at a ground truth range of 25 meters from the ego vehicle. At this close range, detections from the rear sides lie in different azimuthal resolution cells and the radar does not merge them. These distinct rear-side detections are shown as red points (left motorcycle) and yellow points (right motorcycle) in the preceding longitudinal and lateral position error plots. For these unmerged detections, the longitudinal position errors from the rear sides are offset by the rear overhang of the motorcycles (0.37 m). The lateral position errors from the rear sides do not exhibit any bias. This result is consistent with the position errors observed in the FCW scenario.

As the scenario proceeds, the distance between the motorcycles and the radar increases, and the area spanned by the radar's resolution cells grows. When the motorcycles move beyond a ground truth range of 39 meters, detections generated from the rear sides of the motorcycles merge. Compute the azimuthal separation of the outside edges of the rear sides.

```
% Range from radar to rear side of motorcycles
motorcycle = scenario.Actors(2);
radarRange = 39-(radarSensor.SensorLocation(1)+motorcycle.RearOverhang);

% Azimuth separation between outside edges of rear sides
lateralDist = 1.8+0.6; % Total lateral distance between outer sides
azSep = rad2deg(lateralDist/radarRange)
```

```
azSep =

    3.9367
```

At a ground truth range of 39 meters, the outer edges of the rear sides of the motorcycles now lie within the same 4 deg azimuth resolution cell and are merged. Beyond this range, the two motorcycles appear to the radar as a single object and only merged detections (shown in purple) are seen in the preceding plots.

Summary

This example demonstrated how to model the output of automotive radars using synthetic detections. In particular, it presented how the `radarDetectionGenerator` model:

- Provides accurate longitudinal position and velocity measurements over long ranges, but has limited lateral accuracy at long ranges
- Generates multiple detections from single target at close ranges, but merges detections from multiple closely spaced targets into a single detection at long ranges
- Sees vehicles and other targets with large radar cross-sections over long ranges, but has limited detection performance for nonmetallic objects such as pedestrians

See Also

Apps
Driving Scenario Designer

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

More About

- “Sensor Fusion Using Synthetic Radar and Vision Data” on page 7-196
- “Model Vision Sensor Detections” on page 7-393

Model Vision Sensor Detections

This example shows how to model and simulate the output of an automotive vision sensor for different driving scenarios. Generating synthetic vision detections is important for testing and validating tracking and sensor fusion algorithms in corner cases or when sensor hardware is unavailable. This example analyzes the differences between vision measurements and vehicle ground truth position and velocity for a forward collision warning (FCW) scenario, a passing vehicle scenario, and a hill descent scenario.

In this example, you generate vision detections programmatically. You can also generate detections by using the Driving Scenario Designer app. For an example, see “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2.

Introduction

Vehicles that contain advanced driver assistance system (ADAS) features or are designed to be fully autonomous typically rely on multiple types of sensors. These sensors include sonar, radar, lidar, and vision. A robust solution includes a sensor fusion algorithm to combine the strengths across the various types of sensors included in the system. For more information about sensor fusion of synthetic detections from a multisensor ADAS system, see “Sensor Fusion Using Synthetic Radar and Vision Data” on page 7-196.

When using synthetic detections for testing and validating tracking and sensor fusion algorithms, it is important to understand how the generated detections model the sensor's unique performance characteristics. Each kind of automotive sensor provides a specific set of strengths and weaknesses which contribute to the fused solution. This example presents some important performance characteristics of automotive vision sensors and shows how the sensor performance is modeled by using synthetic detections.

Vision Sensor Model

This example uses `visionDetectionGenerator` to generate synthetic vision sensor detections. `visionDetectionGenerator` models the following performance characteristics of automotive vision sensors:

Strengths

- Good lateral position and velocity accuracy
- One detection reported per target

Weaknesses

- Poor longitudinal position and velocity accuracy
- Inability to detect occluded targets
- Longitudinal biases for elevated targets

FCW Driving Scenario

Create a forward collision warning (FCW) test scenario, which is used to illustrate how to measure a target's position with an automotive vision sensor. The scenario consists of a moving ego vehicle and a stationary target vehicle placed 75 meters down the road. The ego vehicle has an initial speed of 50 kph before applying its brakes to achieve a constant deceleration of 3 m/s^2. The vehicle then comes to a complete stop 1 meter before the target vehicle's rear bumper.

```
rng default;
initialDist = 75;  % m
finalDist = 1;     % m
initialSpeed = 50; % kph
brakeAccel = 3;    % m/s^2
[scenario, egoCar] = helperCreateSensorDemoScenario('FCW', initialDist, initialSpeed, brakeAccel
```

Forward-Facing Vision Sensor

Create a forward-facing vision sensor mounted on the ego vehicle's front windshield, 1.1 m above the ground. The sensor is pitched down 1 degree toward the road and generates measurements every 0.1 second. The sensor's camera has a 480-by-640 pixel imaging array and a focal length of 800 pixels. The sensor can locate objects within a single image with an accuracy of 5 pixels and has a maximum detection range of 150 m. The `ActorProfiles` property specifies the physical dimensions of the vehicles seen by the vision sensor in the simulation.

```
visionSensor = visionDetectionGenerator(...
    'SensorIndex', 1, ...
    'UpdateInterval', 0.1, ...
    'SensorLocation', [0.75*egoCar.Wheelbase 0], ...
    'Height', 1.1, ...
    'Pitch', 1, ...
    'Intrinsics', cameraIntrinsics(800, [320 240], [480 640]), ...
    'BoundingBoxAccuracy', 5, ...
    'MaxRange', 150, ...
    'ActorProfiles', actorProfiles(scenario))
```

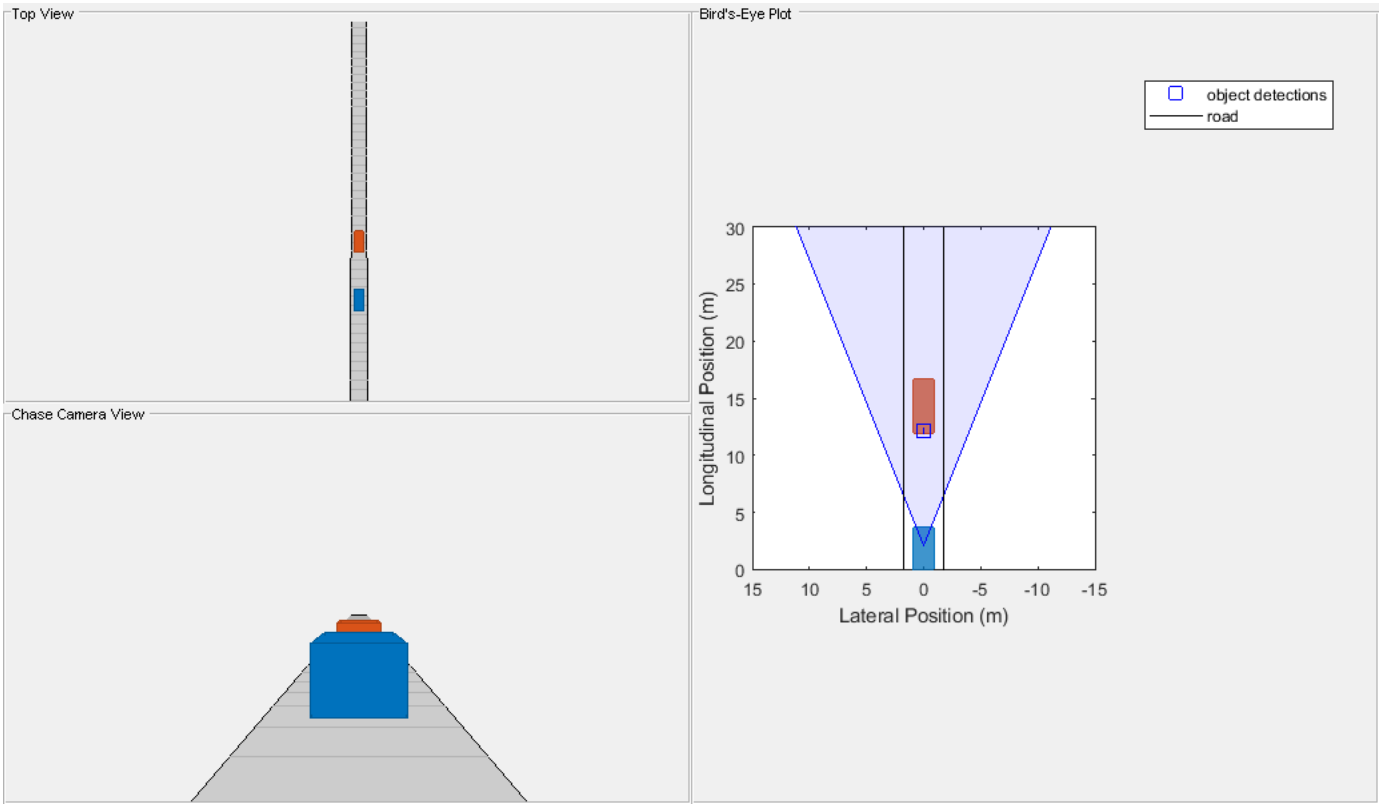
```
visionSensor =

  visionDetectionGenerator with properties:

              SensorIndex: 1
           UpdateInterval: 0.1000

           SensorLocation: [2.1000 0]
                   Height: 1.1000
                      Yaw: 0
                    Pitch: 1
                     Roll: 0
               Intrinsics: [1x1 cameraIntrinsics]

           DetectorOutput: 'Objects only'
              FieldOfView: [43.6028 33.3985]
                 MaxRange: 150
                 MaxSpeed: 100
      MaxAllowedOcclusion: 0.5000
       MinObjectImageSize: [15 15]

     DetectionProbability: 0.9000
   FalsePositivesPerImage: 0.1000

  Use get to show all properties
```

Simulation of Vision Detections

Simulate the vision sensor measuring the position of the target vehicle by advancing the simulation time of the scenario. The vision sensor generates detections from the true target pose (position, velocity, and orientation) expressed in the ego vehicle's coordinate frame.

The vision sensor is configured to generate detections at 0.1-second intervals, which is consistent with the update rate of typical automotive vision sensors. However, to accurately model the motion of the vehicles, the scenario simulation advances every 0.01 seconds. The sensor returns a logical flag, `isValidTime`, that is true when the vision sensor reaches its required update interval, indicating that this simulation time step will generate detections.

```
% Create display for FCW scenario
[bep, figScene] = helperCreateSensorDemoDisplay(scenario, egoCar, visionSensor);

metrics = struct;                % Initialize struct to collect scenario metrics
while advance(scenario)          % Update vehicle positions
    gTruth = targetPoses(egoCar); % Get target positions in ego vehicle coordinates

    % Generate time-stamped vision detections
    time = scenario.SimulationTime;
    [dets, ~, isValidTime] = visionSensor(gTruth, time);

    if isValidTime
        % Update Bird's-Eye Plot with detections and road boundaries
        helperUpdateSensorDemoDisplay(bep, egoCar, visionSensor, dets);

        % Collect vision detections and ground truth for offline analysis
        metrics = helperCollectScenarioMetrics(metrics, gTruth, dets);
    end

    % Take a snapshot for the published example
    helperPublishSnapshot(figScene, time>=6);
end
```

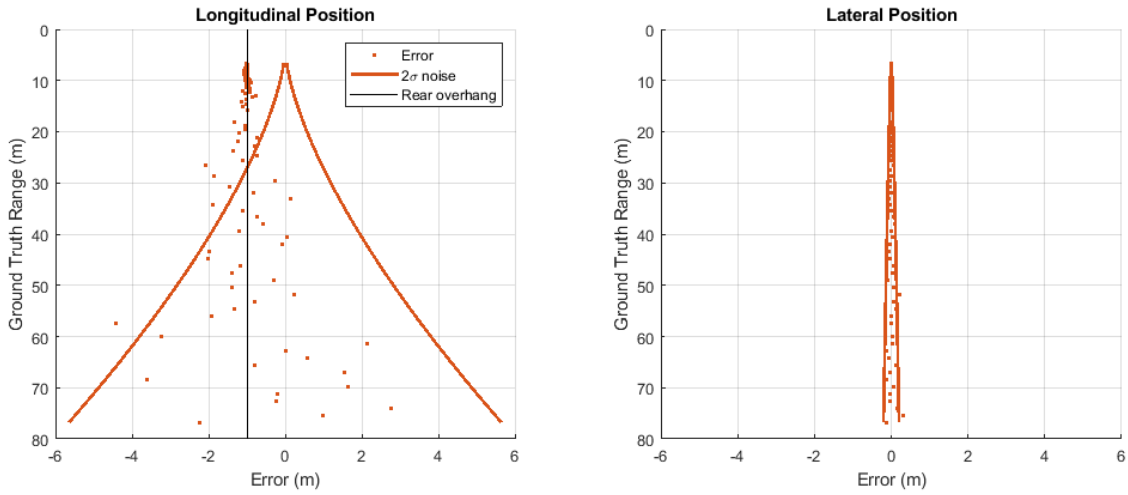


Position Measurements

Over the duration of the FCW test, the target vehicle's distance from the ego vehicle spans a wide range of values. By comparing the vision sensor's measured longitudinal and lateral positions of the target vehicle to the target vehicle's ground truth position, you can observe the accuracy of the sensor's measured positions.

Use `helperPlotSensorDemoDetections` to plot the longitudinal and lateral position errors as the difference between the measured position reported by the vision sensor and the target vehicle's ground truth. The ground truth reference for the target vehicle is the point on the ground directly below the center of the target vehicle's rear axle, which is 1 meter in front of the car's bumper.

```
helperPlotSensorDemoDetections(metrics, 'position', 'reverse range', [-6 6]);

% Show rear overhang of target vehicle
tgtCar = scenario.Actors(2);
rearOverhang = tgtCar.RearOverhang;

subplot(1,2,1);
hold on; plot(-rearOverhang*[1 1], ylim, 'k'); hold off;
legend('Error', '2\sigma noise', 'Rear overhang');
```



The vision sensor converts the target's position in the camera's image to longitudinal and lateral positions in the ego vehicle's coordinate system. The sensor does this conversion by assuming that the detected points in the image lie on a flat road that is at the same elevation as the ego vehicle.

Longitudinal Position Measurements

For a forward-facing vision sensor configuration, longitudinal position measurements are derived primarily from the target's vertical placement in the camera's image.

An object's vertical position in the image is strongly correlated to the object's height above the road, but it is weakly correlated to the object's distance from the camera. This weak correlation causes a monocular vision sensor's longitudinal position errors to become large as an object moves away from the sensor. The longitudinal position errors in the preceding plot on the left show how the sensor's longitudinal errors quickly increase when the target vehicle is far from the sensor. The sensor's longitudinal 2σ measurement noise is less than 1 meter when the ground truth range to the target vehicle is less than 30 meters, but grows to more than 5 meters at ranges beyond 70 meters from the ego vehicle.

The longitudinal position errors also show a -1 meter bias between the longitude measured by the vision sensor and the target's ground truth position. The -1 meter bias indicates that the sensor consistently measures the target to be closer to the ego vehicle than the target vehicle's ground truth position. Instead of approximating the target as a single point in space, the vision sensor models the physical dimensions of the vehicle's body. For the FCW scenario, the vision sensor views the target vehicle's rear side. The -1 meter bias in the detections generated from this side corresponds to the vehicle's rear overhang. A vehicle's rear overhang defines the distance between the vehicle's rear side and its rear axle, which is where the ground truth reference is located.

Lateral Position Measurements

For a forward-facing vision sensor configuration, lateral position is derived from the target's horizontal location in the camera's image.

Unlike longitudinal position, an object's lateral position is strongly correlated to its horizontal position in the vision sensor's image. This strong correlation produces accurate lateral position measurements that do not degrade quickly with an object's distance from the sensor. The lateral position errors in the preceding plot on the right grow slowly with range. The 2σ measurement noise reported by the sensor remains below 0.2 meters out to a ground truth range of 70 meters.

Velocity Measurements and Target Occlusion

Create a driving scenario with two target vehicles (a lead car and a passing car) to illustrate the accuracy of a vision sensor's longitudinal and lateral velocity measurements. The lead car is placed 40 meters in front of the ego vehicle and is traveling with the same speed. The passing car starts in the left lane alongside the ego vehicle, passes the ego vehicle, and merges into the right lane just behind the lead car. This merging maneuver generates longitudinal and lateral velocity components, enabling you to compare the sensor's accuracy along these two dimensions.

Because the lead car is directly in front of the sensor, it has a purely longitudinal velocity component. The passing car has a velocity profile with both longitudinal and lateral velocity components. These components change as the car passes the ego vehicle and moves into the right lane behind the lead car. Comparing the sensor's measured longitudinal and lateral velocities of the target vehicles to their ground truth velocities illustrates the vision sensor's ability to observe both of these velocity components.

```
% Create passing scenario
leadDist = 40;     % m
speed = 50;        % kph
passSpeed = 70;    % kph
mergeFract = 0.55; % Merge 55% into right lane
[scenario, egoCar] = helperCreateSensorDemoScenario('Passing', leadDist, speed, passSpeed, mergeF
```

Configuration of Vision Sensor Velocity Measurements

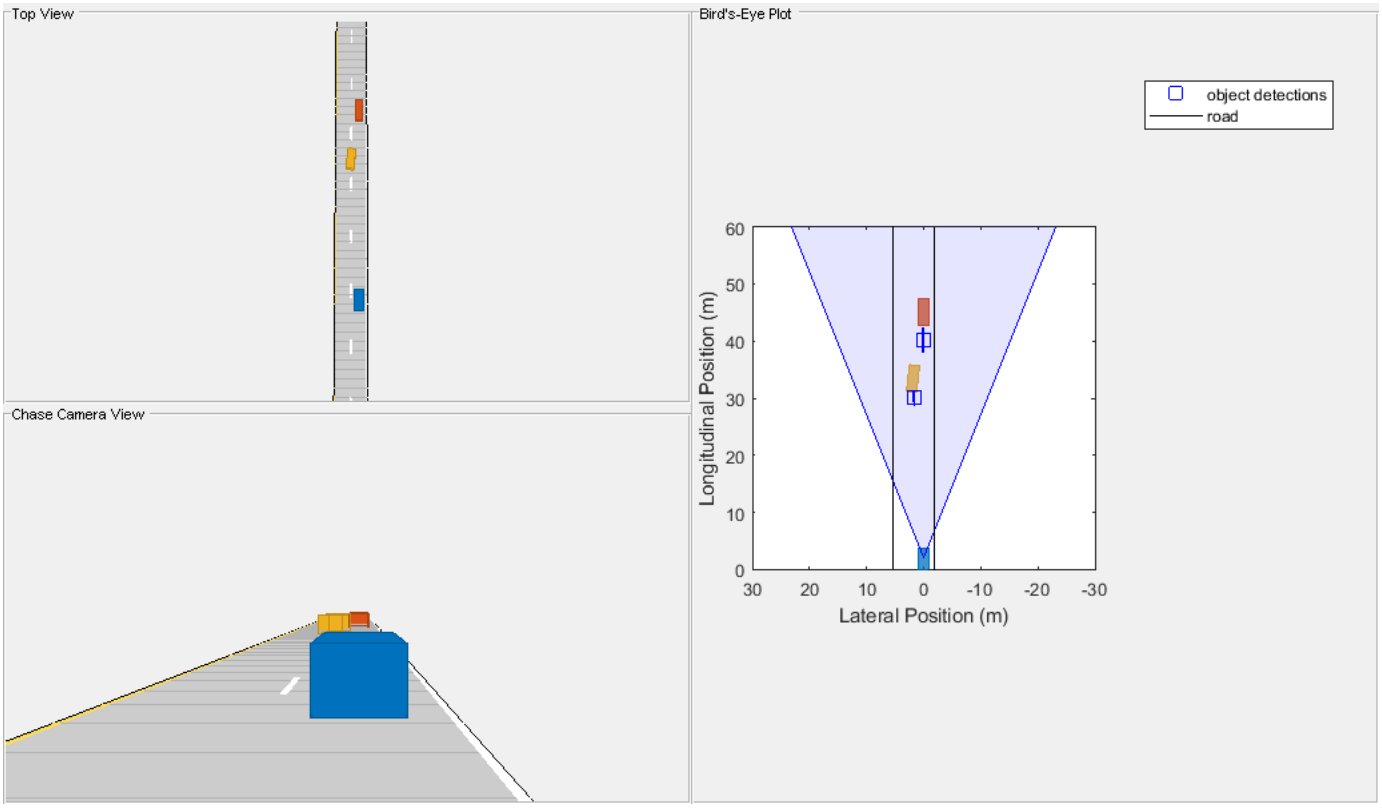
The vision sensor cannot determine an object's velocity from a single image. To estimate velocity, the vision sensor compares the object's movement between multiple images. The extracted target positions from multiple images are processed by using a smoothing filter. In addition to estimating velocity, this filter produces a smoothed position estimate. To adjust the amount of smoothing that the filter applies, you can set the sensor's process noise intensity. The sensor's process noise should be set to be of the order of the maximum acceleration magnitude expected from a target that must be detected by the sensor.

Take the vision sensor used in the previous section, and configure it to generate position and velocity estimates from a smoothing filter with a process noise intensity of 5 m/s^2.

```
% Configure vision sensor's noise intensity used by smoothing filter
release(visionSensor);
visionSensor.ProcessNoiseIntensity = 5; % m/s^2

% Use actor profiles for the passing car scenario
visionSensor.ActorProfiles = actorProfiles(scenario);
```

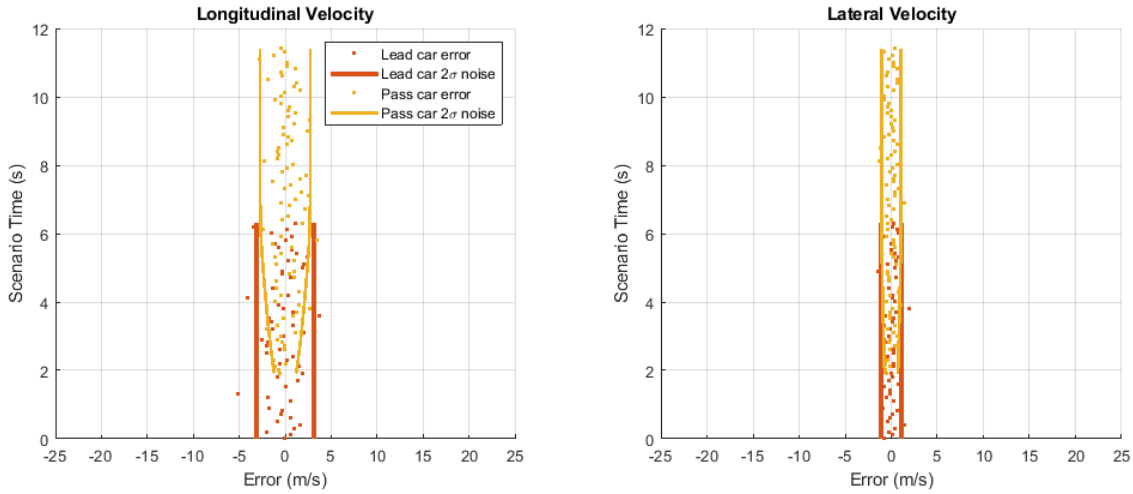
Use `helperRunSensorDemoScenario` to simulate the motion of the ego and target vehicles. This function also collects the simulated metrics, as was previously done for the FCW driving scenario.

```
snapTime = 5.9; % Simulation time to take snapshot for publishing
metrics = helperRunSensorDemoScenario(scenario, egoCar, visionSensor, snapTime);
```



Use `helperPlotSensorDemoDetections` to plot the vision sensor's longitudinal and lateral velocity errors as the difference between the measured velocity reported by the sensor and the target vehicle's ground truth.

```
helperPlotSensorDemoDetections(metrics, 'velocity', 'time', [-25 25]);
subplot(1,2,1);
legend('Lead car error', 'Lead car 2\sigma noise', 'Pass car error', 'Pass car 2\sigma noise');
```



Longitudinal Velocity Measurements

Forward-facing vision sensors measure longitudinal velocity by comparing how the sensor's longitudinal position measurements change between sensor update intervals. Because the sensor's longitudinal position errors grow with range, the longitudinal velocity errors will also grow with increasing target range.

The longitudinal velocity errors from the passing vehicle scenario are shown in the preceding plot on the left. Because the lead car maintains a constant distance from the vision sensor, its errors (displayed as red points) show the same 2σ measurement noise throughout the scenario. However, the passing car's distance from the sensor is not constant, but this distance increases as the car passes the sensor and moves toward the lead car. The passing car's longitudinal velocity errors (displayed as yellow points) are small when it first enters the sensor's field of view at 2 seconds. The passing car is close to the vision sensor at this point in the scenario. From 2 seconds to 6 seconds, the passing car is moving away from the ego vehicle and approaching the lead car. Its longitudinal velocity errors grow as its distance from the sensor increases. Once the passing car merges into the right lane behind the lead car, it maintains a constant distance from the sensor, and its 2σ measurement noise remains constant.

Lateral Velocity Measurements

Forward-facing vision sensors measure lateral velocity by comparing how the sensor's lateral position measurements change between sensor update intervals. Because the sensor's lateral position errors are not strongly correlated with the target's range from the sensor, the lateral velocity errors will also show little dependence on target range.

The lateral velocity errors from the passing vehicle scenario are shown in the preceding plot on the right. The errors from the lead car (red points) and the passing car (yellow points) have nearly the same measurement noise for the entire scenario. The passing car's reported lateral velocity errors show little change as it moves away from the sensor.

Detection of Targets with Partial Occlusion

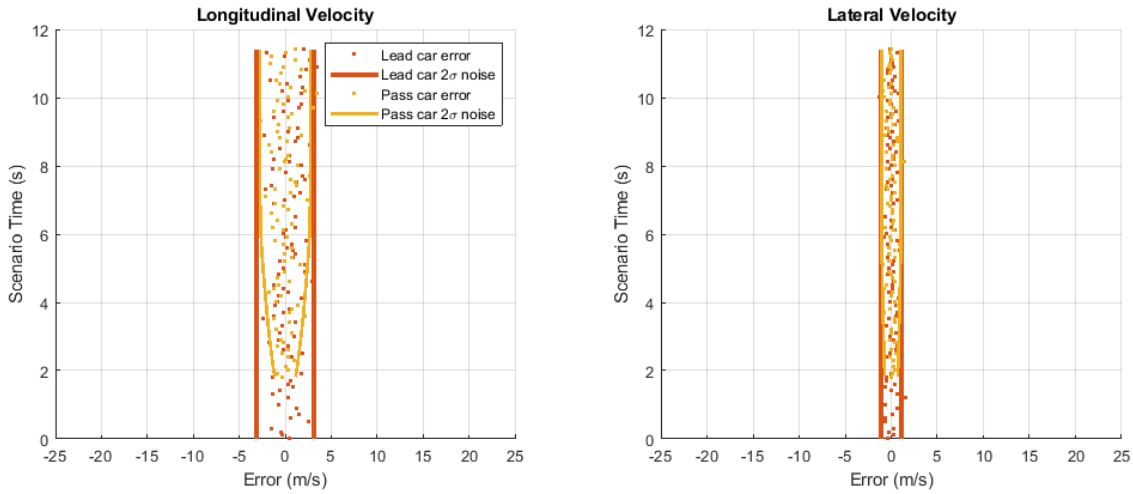
In the preceding velocity error plots, the lead car (red points) is reliably detected during the first 6 seconds of the scenario. The passing car (yellow points) is detected at 2 seconds when it first enters the camera's field of view. Detections are then generated on both of the target vehicles until 6 seconds. At 6 seconds, the passing car merges into the right lane and moves between the ego vehicle and the lead car. For the remainder of the scenario, the passing car partially occludes the vision sensor's view of the lead car. 55% of the lead car's rear side is occluded, leaving only 45% visible to the sensor for detection. This occluded view of the lead car prevents the sensor from finding the car in the camera's image and generating detections.

A vision sensor's ability to provide reliable detections is strongly dependent on an unobstructed view of the object it is detecting. In dense traffic, the visibility of vehicles in the scenario can change rapidly as distances between vehicles change and vehicles move in and out of lanes. This inability to maintain detection on obstructed targets poses a challenge to tracking algorithms processing the vision sensor's detections.

Rerun the passing vehicle scenario with a vision sensor that can detect targets with as much as 60% of the target's viewable area occluded.

```
% Configure vision sensor to support maximum occlusion of 60%
release(visionSensor);
visionSensor.MaxAllowedOcclusion = 0.6;

% Run simulation and collect detections and ground truth for offline analysis
```

```
metrics = helperRunSensorDemoScenario(scenario, egoCar, visionSensor);

% Plot longitudinal and lateral velocity errors
helperPlotSensorDemoDetections(metrics, 'velocity', 'time', [-25 25]);
subplot(1,2,1);
legend('Lead car error', 'Lead car 2\sigma noise', 'Pass car error', 'Pass car 2\sigma noise');
```



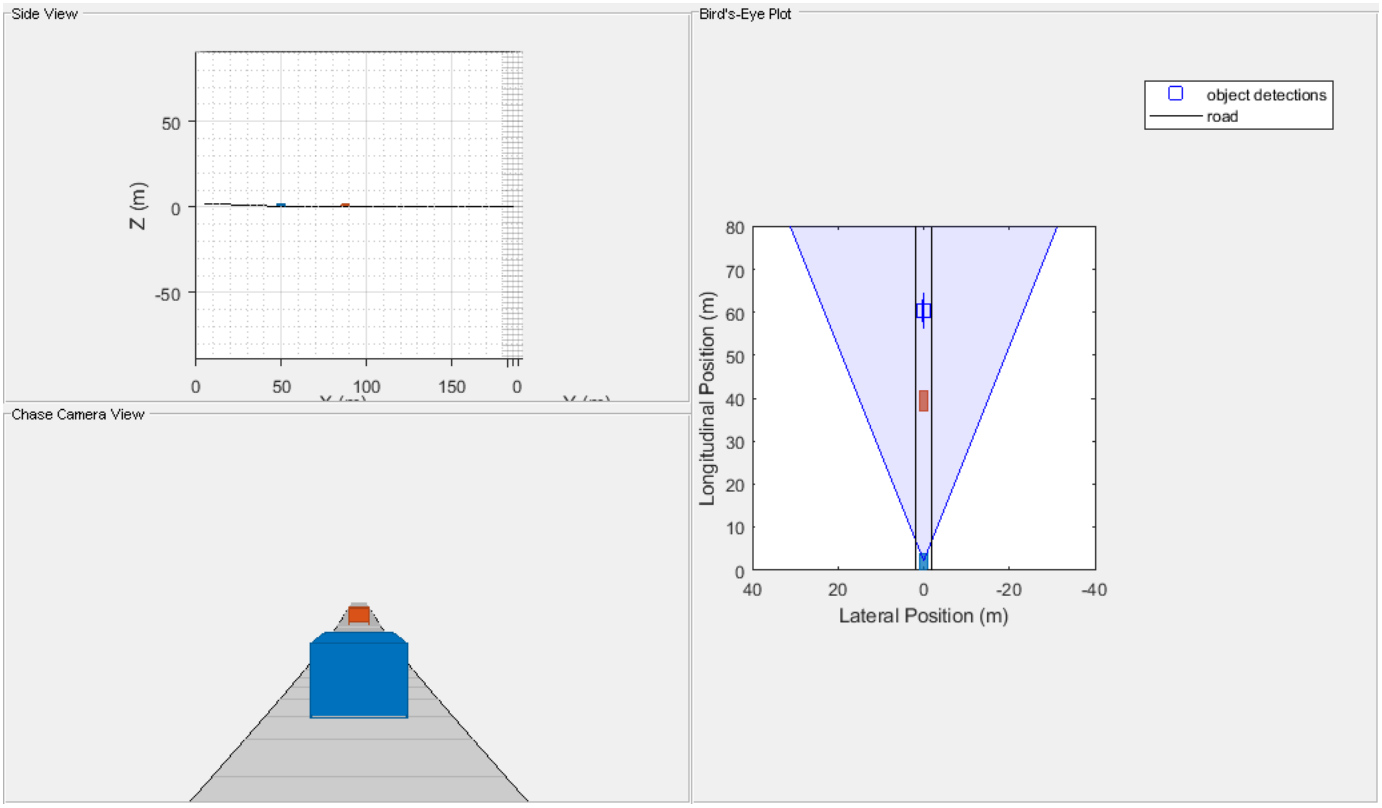
In the preceding plot, velocity errors are shown for both the lead car (red points) and the passing car (yellow points). The same error performance is observed as before, but now detections on the lead car are maintained after the passing car has merged behind it at 6 seconds. By adjusting the maximum allowed occlusion, you can model a vision sensor's sensitivity to target occlusion.

Longitudinal Position Bias from Target Elevation

An object's vertical location in the camera's image is strongly correlated to its height above the road. Because a monocular vision sensor generates longitudinal position measurements from the vertical location of objects in its camera's image, large errors can arise for targets at different elevations from the ego vehicle. When an object changes elevation, the sensor incorrectly interprets the vertical displacement in the camera's image as a change in the object's longitudinal position.

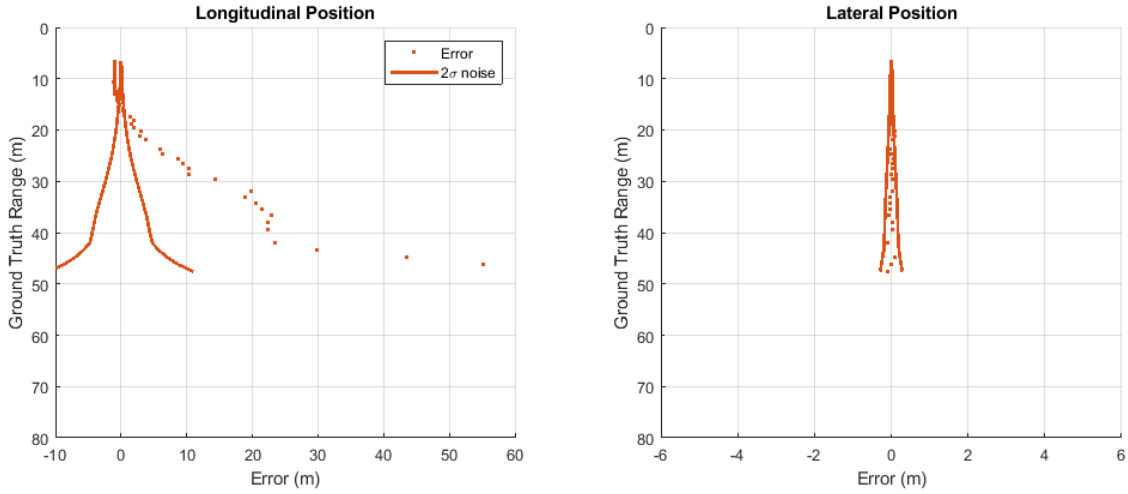
Run the FCW scenario again with a stationary target vehicle placed at a location 2 meters lower than the initial position of the ego vehicle. The ego vehicle descends a small hill as it approaches the target vehicle. As the ego vehicle descends the hill, the target vehicle's vertical location in the camera's image changes, introducing a bias in the sensor's measured longitudinal position.

```
% Create FCW hill descent scenario
initialDist = 75;  % m
finalDist = 1;     % m
initialSpeed = 50; % kph
brakeAccel = 3;    % m/s^2
[scenario, egoCar] = helperCreateSensorDemoScenario('FCW', initialDist, initialSpeed, brakeAccel

% Use actor profiles for the FCW hill descent scenario
release(visionSensor);
visionSensor.ActorProfiles = actorProfiles(scenario);

% Run simulation and collect detections and ground truth for offline analysis
snapTime = 3; % Simulation time to take snapshot for publishing
metrics = helperRunSensorDemoScenario(scenario, egoCar, visionSensor, snapTime, true);
```



Plot the position errors generated for the target vehicle as the ego vehicle descends the small hill.

```
helperPlotSensorDemoDetections(metrics, 'position', 'reverse range', [-6 6;0 80]);
subplot(1,2,1); xlim([-10 60]); ylim([0 80]);
legend('Error', '2\sigma noise');
```



The preceding plots show the longitudinal position errors (on the left) and the lateral position errors (on the right) for the hill descent scenario. Note that in the longitudinal position error plot, the limits of the error axis have been increased to accommodate the large bias induced by the target's elevation relative to the camera sensor as the ego vehicle descends the hill.

The ego vehicle begins its descent when it is 75 meters away from the target vehicle. Because the ego vehicle is pitched down as it descends the hill, the target appears at an elevated position near the top of the camera's image. As the ego vehicle descends the hill, the target vehicle's location in the camera's image moves from the top of the image and crosses the horizon line. For monocular vision sensors, targets located near the horizon line in the camera's image are mapped to positions that are very far from the sensor. (By definition, points on the horizon line are located at infinity.) The vision sensor does not generate detections for objects appearing above the horizon in the camera's image, because these points do not map to locations on the road's surface.

Large changes in the vehicle's longitudinal position as its location in the image moves away from the horizon also cause the sensor's smoothing filter to generate large longitudinal velocity estimates. The sensor rejects detections with speeds exceeding its `MaxSpeed` property. These large longitudinal velocities produced by the target's elevation also prevent the sensor from generating detections when the target vehicle is near the camera's horizon.

When the ego vehicle is approximately 40 meters from the target vehicle, the target vehicle's image location has crossed the horizon line and the sensor's velocity estimates satisfy its max speed constraint. At this distance, the vision sensor begins to generate detections from the target vehicle. The mapping of target locations near the camera's horizon to points on the road far from the sensor explains the large longitudinal errors modeled by the monocular vision sensor when it begins detecting the target vehicle. The longitudinal bias continues to decrease as the ego vehicle approaches the bottom of the hill and the target's location moves away from the horizon line in the camera's image. At the end of the ego vehicle's descent, the target is at the same elevation as the ego vehicle. Only the -1 meter bias corresponding to the target vehicle's rear overhang is present. The sensor's lateral position errors show no bias, because the pitch of the ego vehicle as it descends the hill does not change the target's horizontal location in the camera's image.

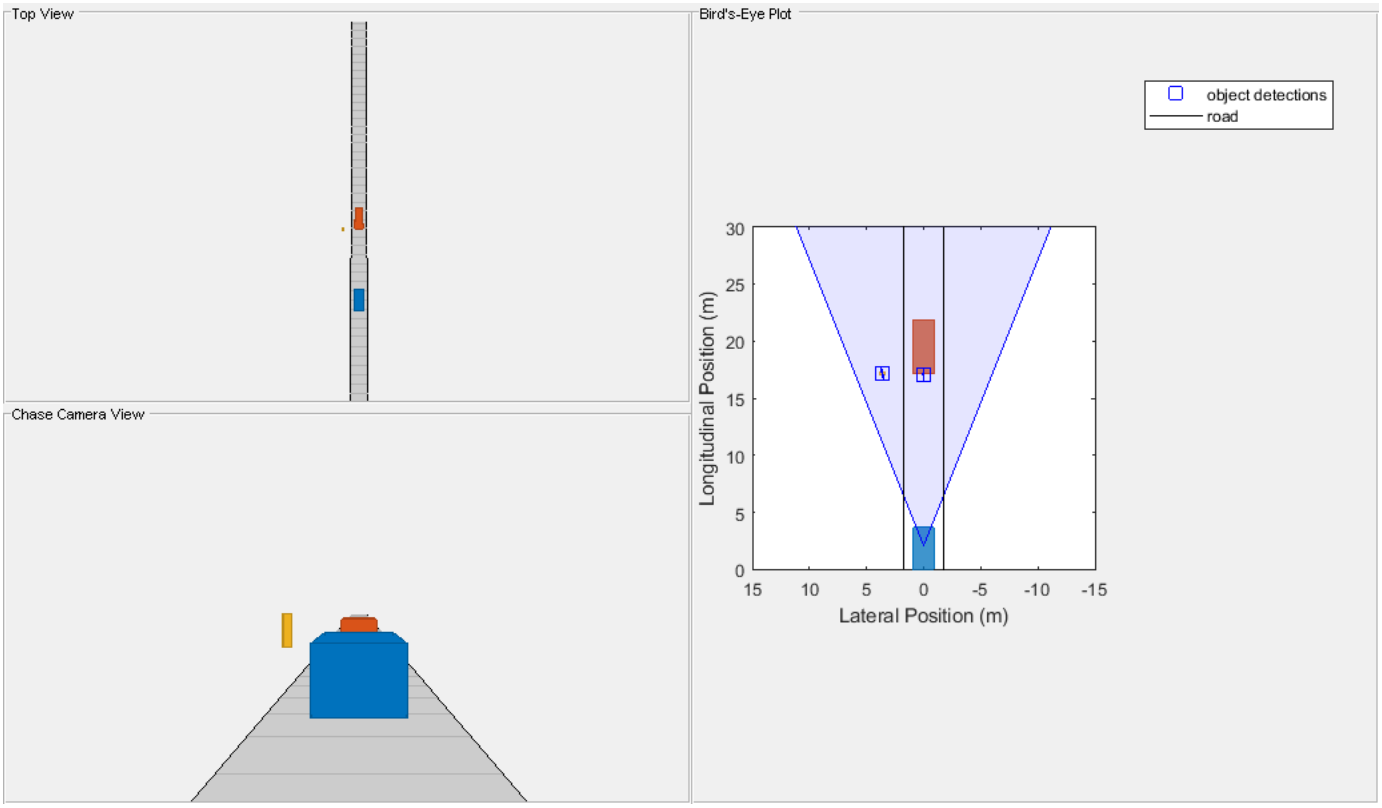
Pedestrian and Vehicle Detection

A vision sensor's ability to detect an object in its camera's image depends on the number of pixels the object occupies in the image. When an object's size in the image is large (hundreds of pixels), the sensor can easily identify the object and generate a detection. However when an object's size in the image is small (tens of pixels) the sensor might not find it and will not generate a detection. An object's projected size on the camera's imaging array is a function of both the object's physical size and its distance from the camera. Therefore, when a vehicle is positioned farther from the camera than a pedestrian, both the vehicle and the pedestrian might have similar sizes in the camera's image. This means that a vision sensor will detect large objects (vehicles) at longer ranges than smaller objects (pedestrians).

Run the FCW scenario again with both a stationary car and pedestrian 75 meters in front of the sensor. This scenario illustrates the difference in the sensor's detection range for these two objects. The ego vehicle, stationary car, and pedestrian are all placed at the same elevation.

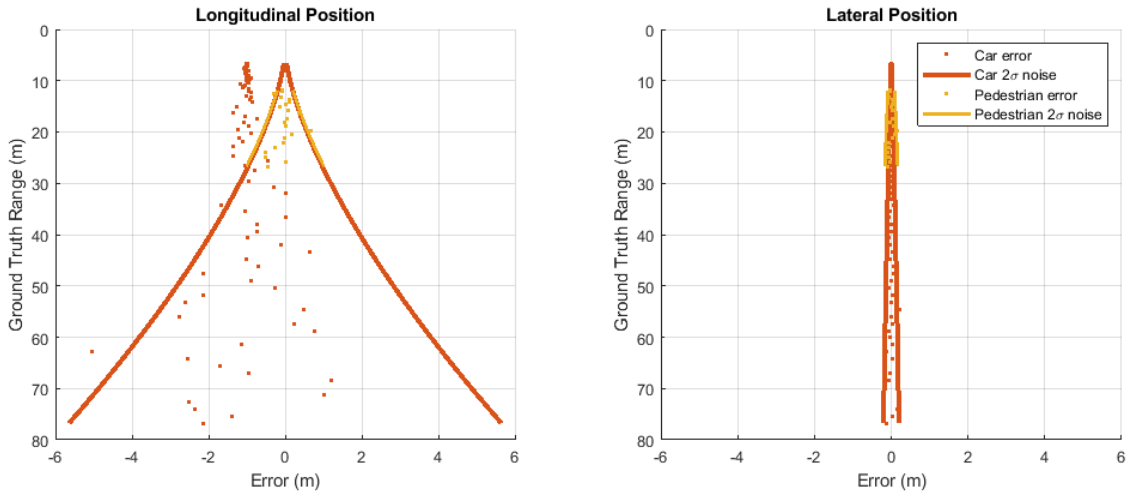
```
% Create FCW test scenario
initialDist = 75;  % m
finalDist = 1;     % m
initialSpeed = 50; % kph
brakeAccel = 3;    % m/s^2
[scenario, egoCar] = helperCreateSensorDemoScenario('FCW', initialDist, initialSpeed, brakeAccel

% Use actor profiles for the FCW hill descent scenario
release(visionSensor);
visionSensor.ActorProfiles = actorProfiles(scenario);

% Run simulation and collect detections and ground truth for offline analysis
```

```
snapTime = 5; % Simulation time to take snapshot for publishing
metrics = helperRunSensorDemoScenario(scenario, egoCar, visionSensor, snapTime);
```



Plot the position errors generated for the target vehicle and pedestrian from the FCW scenario.

```
helperPlotSensorDemoDetections(metrics, 'position', 'reverse range', [-6 6]);
legend('Car error','Car 2\sigma noise', 'Pedestrian error', 'Pedestrian 2\sigma noise');
```



The preceding plots show the longitudinal position errors (on the left) and lateral position errors (on the right) generated by the vision sensor's detections of the target vehicle and pedestrian. Errors

from detections for the target vehicle (shown in red) are generated out to the farthest range included in this test (75 m), but errors for the pedestrian (shown in yellow) do not appear until the ego vehicle has reached a distance of approximately 30 m. This difference in the detection ranges is due to the difference in the sizes of these two objects.

The sensor stops generating detections from the pedestrian at ranges less than 12 m. At this range, the offset of the pedestrian from the camera's optical axis moves the pedestrian outside of the camera's horizontal field of view. Because the target vehicle is directly in front of the camera, it remains centered within the camera's image for the entire FCW test.

Some vision sensors can detect objects with smaller image sizes, enabling the sensors to detect objects at longer ranges. In the previous scenario, the sensor's detection of the pedestrian is limited by the pedestrian's width (0.45 m), which is much narrower than the width of the car (1.8 m). To increase the sensor's detection range for pedestrians to 40 m, compute the width of the pedestrian in the camera's image when it is at 40 m.

Find physical width modeled for a pedestrian

```
profiles = actorProfiles(scenario);
pedWidth = profiles(3).Width

% Compute width of pedestrian in camera's image in pixels at 40 meters from ego vehicle
cameraRange = 40-visionSensor.SensorLocation(1);
focalLength = visionSensor.Intrinsics.FocalLength(1);
pedImageWidth = focalLength*pedWidth/cameraRange
```

```
pedWidth =

    0.4500


pedImageWidth =

    9.4987
```

At 40 m, the pedestrian has a width of 9.5 pixels in the camera's image. Set the vision sensor's minimum object width to match the pedestrian's width at 40 m.

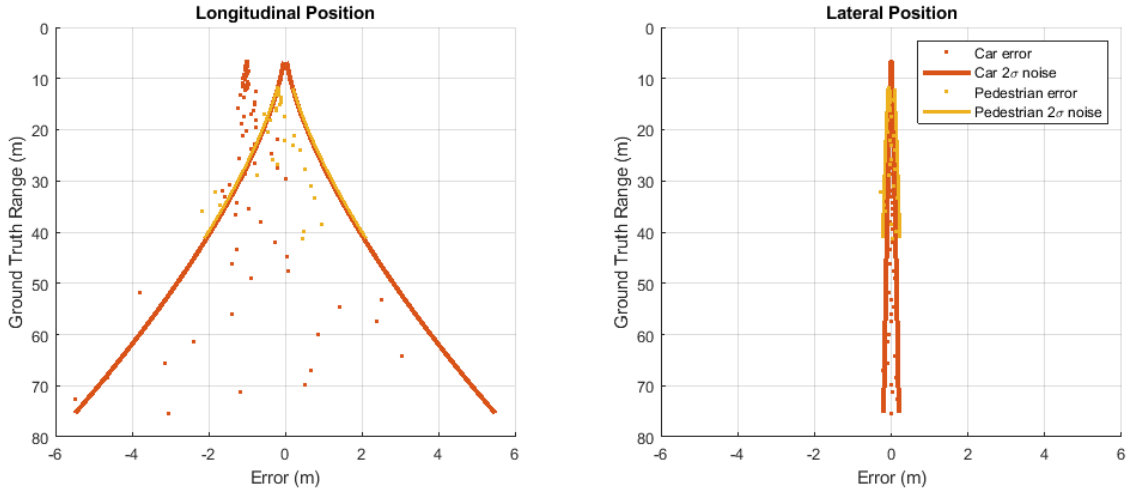
```
% Configure sensor to detect pedestrians out to a range of 40 m
release(visionSensor);
visionSensor.MinObjectImageSize(2) = pedImageWidth
```

```
visionSensor =

  visionDetectionGenerator with properties:

             SensorIndex: 1
          UpdateInterval: 0.1000

          SensorLocation: [2.1000 0]
                  Height: 1.1000
                     Yaw: 0
                   Pitch: 1
                    Roll: 0
```

```
                Intrinsics: [1x1 cameraIntrinsics]

            DetectorOutput: 'Objects only'
               FieldOfView: [43.6028 33.3985]
                  MaxRange: 150
                  MaxSpeed: 100
        MaxAllowedOcclusion: 0.6000
         MinObjectImageSize: [15 9.4987]

       DetectionProbability: 0.9000
     FalsePositivesPerImage: 0.1000

  Use get to show all properties
```

Run the scenario again and plot the position errors to show the revised detection ranges for the vehicle and pedestrian.

```
% Run simulation and collect detections and ground truth for offline analysis
metrics = helperRunSensorDemoScenario(scenario, egoCar, visionSensor);

% Plot position errors for the target vehicle and pedestrian
helperPlotSensorDemoDetections(metrics, 'position', 'reverse range', [-6 6]);
legend('Car error','Car 2\sigma noise', 'Pedestrian error', 'Pedestrian 2\sigma noise');
```



The preceding plots show the longitudinal position errors (on the left) and lateral position errors (on the right) for a vision sensor configured to support pedestrian detection out to a range of 40 m. The vehicle (shown in red) is still detected out to the farthest test range, but now detections on the pedestrian (shown in yellow) are generated out to 40 m from the sensor.

Lane Boundary Measurements and Lane Occlusion

The vision detection generator can also be configured to detect lanes. Recreate the two-lane driving scenario with the lead car and passing car to illustrate the accuracy of a vision sensor's lane boundary measurements. This same merging maneuver is used to occlusion of the lane markings.

```
% Create passing scenario
leadDist = 40;     % m
speed = 50;        % kph
```

```
passSpeed = 70;    % kph
mergeFract = 0.55; % Merge 55% into right lane
[scenario, egoCar] = helperCreateSensorDemoScenario('Passing', leadDist, speed, passSpeed, mergeF
```

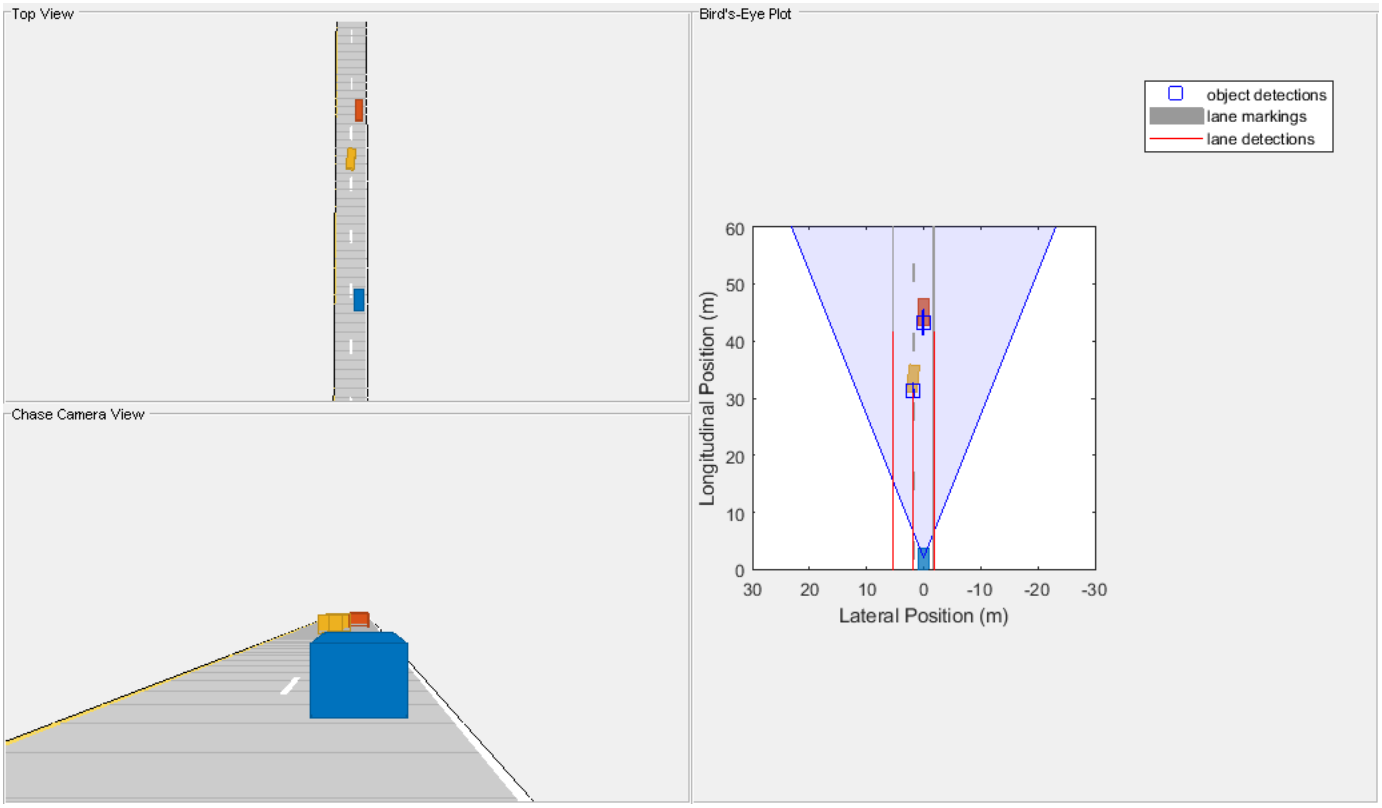
Configuration of Vision Sensor Lane Boundary Measurements

Configure the vision sensor used in the previous section, and configure it to generate position and velocity estimates from a smoothing filter with a process noise intensity of 5 m/s^2.

```
% Configure vision sensor to detect both lanes and objects
release(visionSensor);
visionSensor.DetectorOutput = 'lanes and objects';

% Use actor profiles for the passing car scenario
visionSensor.ActorProfiles = actorProfiles(scenario);
```

Use `helperRunSensorDemoScenario` to simulate the motion of the ego and target vehicles.

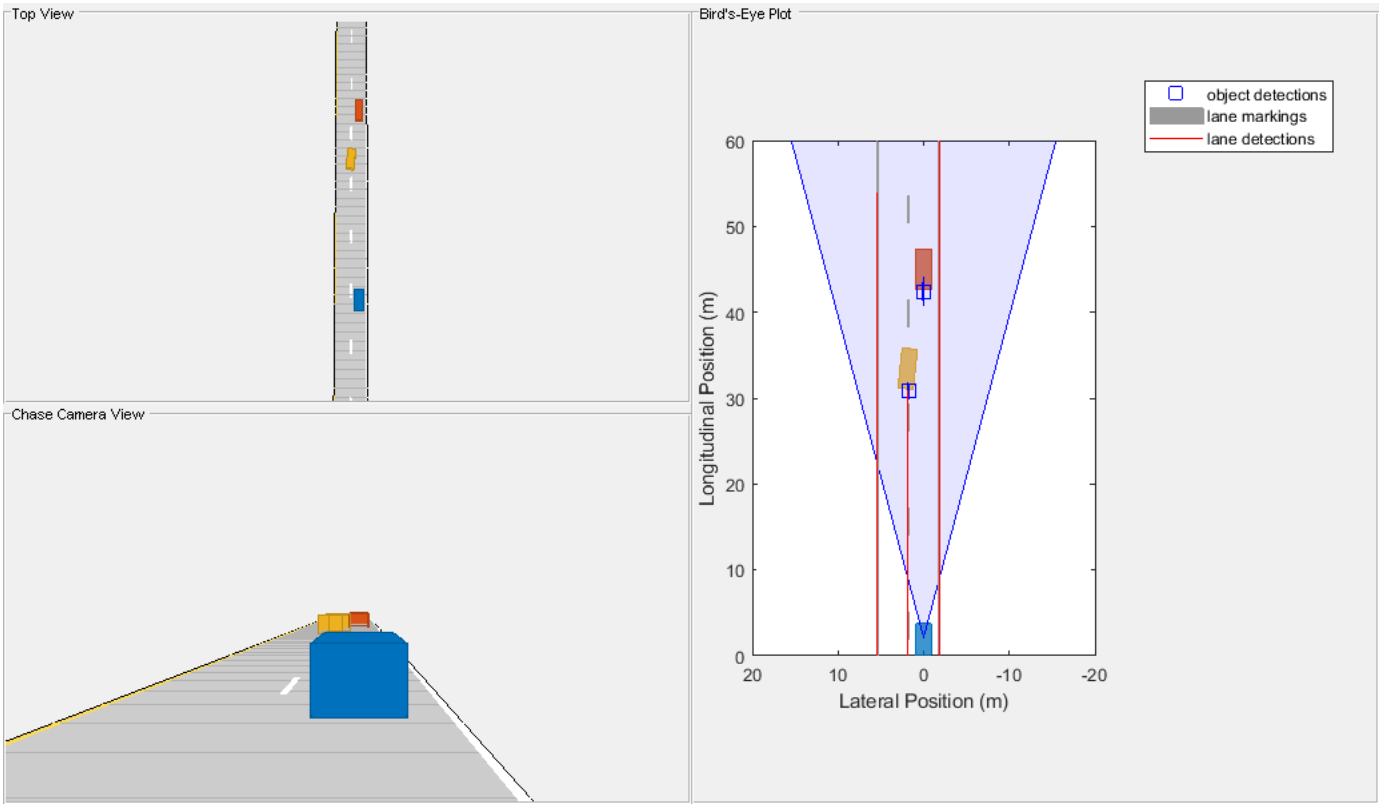
```
snapTime = 5.9; % Simulation time to take snapshot for publishing
helperRunSensorDemoScenario(scenario, egoCar, visionSensor, snapTime);
```



As can be seen above, the default detector can see lane boundaries out to 45 meters or so when presented with an unoccluded view. You can change the intrinsics of the detector to observe its effect.

```
% show camera intrinsics
visionSensor.Intrinsics


ans =
```

```
  cameraIntrinsics with properties:

           FocalLength: [800 800]
        PrincipalPoint: [320 240]
             ImageSize: [480 640]
      RadialDistortion: [0 0]
    TangentialDistortion: [0 0]
                 Skew: 0
       IntrinsicMatrix: [3x3 double]


% increase the focal length and observe its effect.
release(visionSensor);
visionSensor.Intrinsics = cameraIntrinsics([1200 1200],[320 240],[480 640])

helperRunSensorDemoScenario(scenario, egoCar, visionSensor, snapTime);


visionSensor =

  visionDetectionGenerator with properties:

             SensorIndex: 1
          UpdateInterval: 0.1000

          SensorLocation: [2.1000 0]
                  Height: 1.1000
                     Yaw: 0
                   Pitch: 1
                    Roll: 0
              Intrinsics: [1x1 cameraIntrinsics]

          DetectorOutput: 'Lanes and objects'
             FieldOfView: [29.8628 22.6199]
                MaxRange: 150
                MaxSpeed: 100
      MaxAllowedOcclusion: 0.6000
       MinObjectImageSize: [15 9.4987]
         MinLaneImageSize: [20 3]

    DetectionProbability: 0.9000
  FalsePositivesPerImage: 0.1000

  Use get to show all properties
```



Changing the focal length from 800 pixels to 1200 in both x- and y-directions zooms the camera, enabling it to detect out to further ranges.

Summary

This example demonstrated how to model the output of automotive vision sensors using synthetic detections. In particular, it presented how the `visionDetectionGenerator` model:

- Provides accurate lateral position and velocity measurements over long ranges, but has limited longitudinal accuracy at long ranges
- Limits detection according to a target's physical dimensions and a target's occlusion by other objects in the scenario
- Includes longitudinal biases for targets located at different elevations than the ego vehicle
- Adjusts object and lane detections due to monocular camera intrinsics.

See Also

Apps
Driving Scenario Designer

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

More About

- “Sensor Fusion Using Synthetic Radar and Vision Data” on page 7-196
- “Model Radar Sensor Detections” on page 7-377

Radar Signal Simulation and Processing for Automated Driving

This example shows how to model a radar's hardware, signal processing, and propagation environment for a driving scenario. First you develop a model of the radar transmit and receive hardware, signal processing, detection, and estimation using Phased Array System Toolbox™. Then you model the vehicle motion and track the synthetic vehicle detections using Automated Driving Toolbox™. In this example, you use this radar model to track detections in a highway driving scenario.

This example requires Phased Array System Toolbox.

Introduction

You can model vehicle motion by using the `drivingScenario` object from Automated Driving Toolbox. The vehicle ground truth can then be used to generate synthetic sensor detections, which you can track by using the `multiObjectTracker` object. For an example of this workflow, see “Sensor Fusion Using Synthetic Radar and Vision Data” on page 7-196. The automotive radar used in this example uses a statistical model that is parameterized according to high-level radar specifications. The generic radar architecture modeled in this example does not include specific antenna configurations, waveforms, or unique channel propagation characteristics. When designing an automotive radar, or when a radar's specific architecture is known, using a radar model that includes this additional information is recommended.

Phased Array System Toolbox enables you to evaluate different radar architectures. You can explore different transmit and receive array configurations, waveforms, and signal processing chains. You can also evaluate your designs against different channel models to assess their robustness to different environmental conditions. This modeling helps you to identify the specific design that best fits your application requirements.

In this example, you learn how to define a radar model from a set of system requirements for a long-range radar. You then simulate a driving scenario to generate detections from your radar model. A tracker is used to process these detections to generate precise estimates of the position and velocity of the vehicles detected by your automotive radar.

Calculate Radar Parameters from Long-Range Radar Requirements

The radar parameters are defined for the frequency-modulated continuous wave (FMCW) waveform, as described in the example “Automotive Adaptive Cruise Control Using FMCW Technology” (Phased Array System Toolbox). The radar operates at a center frequency of 77 GHz. This frequency is commonly used by automotive radars. For long-range operation, the radar must detect vehicles at a maximum range of 100 meters in front of the ego vehicle. The radar is required to resolve objects in range that are at least 1 meter apart. Because this is a forward-facing radar application, the radar also needs to handle targets with large closing speeds, as high as 230 km/hr.

The radar is designed to use an FMCW waveform. These waveforms are common in automotive applications because they enable range and Doppler estimation through computationally efficient FFT operations.

```
% Set random number generator for repeatable results
rng(2017);

% Compute hardware parameters from specified long-range requirements
fc = 77e9;                                  % Center frequency (Hz)
c = physconst('LightSpeed');                % Speed of light in air (m/s)
```

```
lambda = c/fc;                                  % Wavelength (m)

% Set the chirp duration to be 5 times the max range requirement
rangeMax = 100;                                 % Maximum range (m)
tm = 5*range2time(rangeMax,c);                  % Chirp duration (s)

% Determine the waveform bandwidth from the required range resolution
rangeRes = 1;                                   % Desired range resolution (m)
bw = range2bw(rangeRes,c);                      % Corresponding bandwidth (Hz)

% Set the sampling rate to satisfy both the range and velocity requirements
% for the radar
sweepSlope = bw/tm;                             % FMCW sweep slope (Hz/s)
fbeatMax = range2beat(rangeMax,sweepSlope,c);   % Maximum beat frequency (Hz)

vMax = 230*1000/3600;                           % Maximum Velocity of cars (m/s)
fdopMax = speed2dop(2*vMax,lambda);             % Maximum Doppler shift (Hz)

fifMax = fbeatMax+fdopMax;                      % Maximum received intermediate frequency (IF) (H
fs = max(2*fifMax,bw);                          % Sampling rate (Hz)

% Configure the FMCW waveform using the waveform parameters derived from
% the long-range requirements
waveform = phased.FMCWWaveform('SweepTime',tm,'SweepBandwidth',bw,'SampleRate',fs);
Nsweep = 192;
```

Model Automotive Radar Hardware

The radar uses a uniform linear array (ULA) to transmit and receive the radar waveforms. Using a linear array enables the radar to estimate the azimuthal direction of the reflected energy received from the target vehicles. The long-range radar needs to detect targets across a coverage area that spans 15 degrees in front of the ego vehicle. A 6-element receive array satisfies this requirement by providing a 16 degree half-power beamwidth. On transmit, the radar uses only a single array element, enabling it to cover a larger area than on receive.

```
% Model the antenna element
antElmnt = phased.IsotropicAntennaElement('BackBaffled',true);

% Construct the receive array
Ne = 6;
rxArray = phased.ULA('Element',antElmnt,'NumElements',Ne,'ElementSpacing',lambda/2);

% Form forward-facing beam to detect objects in front of the ego vehicle
beamformer = phased.PhaseShiftBeamformer('SensorArray',rxArray,...
    'PropagationSpeed',c,'OperatingFrequency',fc,'Direction',[0;0]);

% Half-power beamwidth of the receive array
helperAutoDrivingRadarSigProc('Array Beamwidth',rxArray,c,fc)
```

```
ans = 16.3636
```

Estimate the direction-of-arrival of the received signals using a root MUSIC estimator. A beamscan is also used for illustrative purposes to help spatially visualize the distribution of the received signal energy.

```
% Direction-of-arrival estimator for linear phased array signals
doaest = phased.RootMUSICEstimator(...
    'SensorArray',rxArray,...
```

```
    'PropagationSpeed',c,'OperatingFrequency',fc,...
    'NumSignalsSource','Property','NumSignals',1);

% Scan beams in front of ego vehicle for range-angle image display
angscan = -80:80;
beamscan = phased.PhaseShiftBeamformer('Direction',[angscan;0*angscan],...
    'SensorArray',rxArray,'OperatingFrequency',fc);
```

Model the radar transmitter for a single transmit channel, and model a receiver preamplifier for each receive channel, using the parameters defined in the example “Automotive Adaptive Cruise Control Using FMCW Technology” (Phased Array System Toolbox).

```
antAperture = 6.06e-4;                          % Antenna aperture (m^2)
antGain = aperture2gain(antAperture,lambda);    % Antenna gain (dB)

txPkPower = db2pow(5)*1e-3;                     % Tx peak power (W)
txGain = antGain;                               % Tx antenna gain (dB)

rxGain = antGain;                               % Rx antenna gain (dB)
rxNF = 4.5;                                     % Receiver noise figure (dB)

% Waveform transmitter
transmitter = phased.Transmitter('PeakPower',txPkPower,'Gain',txGain);

% Radiator for single transmit element
radiator = phased.Radiator('Sensor',antElmnt,'OperatingFrequency',fc);

% Collector for receive array
collector = phased.Collector('Sensor',rxArray,'OperatingFrequency',fc);

% Receiver preamplifier
receiver = phased.ReceiverPreamp('Gain',rxGain,'NoiseFigure',rxNF,'SampleRate',fs);
```

Define Radar Signal Processing Chain

The radar collects multiple sweeps of the waveform on each of the linear phased array antenna elements. These collected sweeps form a data cube, which is defined in “Radar Data Cube” (Phased Array System Toolbox). These sweeps are coherently processed along the fast- and slow-time dimensions of the data cube to estimate the range and Doppler of the vehicles.

Use the `phased.RangeDopplerResponse` object to perform the range and Doppler processing on the radar data cubes. Use a Hanning window to suppress the large sidelobes produced by the vehicles when they are close to the radar.

```
Nft = waveform.SweepTime*waveform.SampleRate; % Number of fast-time samples
Nst = Nsweep;                                 % Number of slow-time samples
Nr = 2^nextpow2(Nft);                         % Number of range samples after processing
Nd = 2^nextpow2(Nst);                         % Number of Doppler samples after processing
rngdopresp = phased.RangeDopplerResponse('RangeMethod','FFT','DopplerOutput','Speed',...
    'SweepSlope',sweepSlope,...
    'RangeFFTLengthSource','Property','RangeFFTLength',Nr,...
    'RangeWindow','Hann',...
    'DopplerFFTLengthSource','Property','DopplerFFTLength',Nd,...
    'DopplerWindow','Hann',...
    'PropagationSpeed',c,'OperatingFrequency',fc,'SampleRate',fs);
```

Identify detections in the processed range and Doppler data by using a constant false alarm rate (CFAR) detector. CFAR detectors estimate the background noise level of the received radar data.

Detections are found at locations where the signal power exceeds the estimated noise floor by a certain threshold. Low threshold values result in a higher number of reported false detections due to environmental noise. Increasing the threshold produces fewer false detections, but also reduces the probability of detection of an actual target in the scenario. For more information on CFAR detection, see the example “Constant False Alarm Rate (CFAR) Detection” (Phased Array System Toolbox).

```
% Guard cell and training regions for range dimension
nGuardRng = 4;
nTrainRng = 4;
nCUTRng = 1+nGuardRng+nTrainRng;

% Guard cell and training regions for Doppler dimension
dopOver = round(Nd/Nsweep);
nGuardDop = 4*dopOver;
nTrainDop = 4*dopOver;
nCUTDop = 1+nGuardDop+nTrainDop;

cfar = phased.CFARDetector2D('GuardBandSize',[nGuardRng nGuardDop],...
    'TrainingBandSize',[nTrainRng nTrainDop],...
    'ThresholdFactor','Custom','CustomThresholdFactor',db2pow(13),...
    'NoisePowerOutputPort',true,'OutputFormat','Detection index');

% Perform CFAR processing over all of the range and Doppler cells
freqs = ((0:Nr-1)'/Nr-0.5)*fs;
rnggrid = beat2range(freqs,sweepSlope);
iRngCUT = find(rnggrid>0);
iRngCUT = iRngCUT(iRngCUT<=Nr-nCUTRng+1);
iDopCUT = nCUTDop:(Nd-nCUTDop+1);
[iRng,iDop] = meshgrid(iRngCUT,iDopCUT);
idxCFAR = [iRng(:) iDop(:)]';
```

The `phased.RangeEstimator` and `phased.DopplerEstimator` objects convert the locations of the detections found in the range-Doppler data into measurements and their corresponding measurement variances. These estimators fit quadratic curves to the range-Doppler data to estimate the peak location of each detection. The resulting measurement resolutions are a fraction of the range and Doppler sampling of the data.

The root-mean-square (RMS) range resolution of the transmitted waveform is needed to compute the variance of the range measurements. The Rayleigh range resolution for the long-range radar was defined previously as 1 meter. The Rayleigh resolution is the minimum separation at which two unique targets can be resolved. This value defines the distance between range resolution cells for the radar. However, the variance of the target within a resolution cell is determined by the waveform's RMS resolution. For an LFM chirp waveform, the relationship between the Rayleigh resolution and the RMS resolution is given by [1].

$$\sigma_{RMS} = \sqrt{12}\Delta_{Rayleigh}$$

where σ_{RMS} is the RMS range resolution and $\Delta_{Rayleigh}$ is the Rayleigh range resolution.

The variance of the Doppler measurements depends on the number of sweeps processed.

Now, create the range and Doppler estimation objects using the parameters previously defined.

```
rmsRng = sqrt(12)*rangeRes;
rngestimator = phased.RangeEstimator('ClusterInputPort',true,...
    'VarianceOutputPort',true,'NoisePowerSource','Input port','RMSResolution',rmsRng);
```

```
dopestimator = phased.DopplerEstimator('ClusterInputPort',true,...
    'VarianceOutputPort',true,'NoisePowerSource','Input port','NumPulses',Nsweep);
```

To further improve the precision of the estimated vehicle locations, pass the radar's detections to a tracker. Configure the track to use an extended Kalman filter (EKF), which converts the spherical radar measurements into the ego vehicle's Cartesian coordinate frame. Also configure the tracker to use constant velocity dynamics for the detected vehicles. By comparing vehicle detections over multiple measurement time intervals, the tracker further improves the accuracy of the vehicle positions and provides vehicle velocity estimates.

```
tracker = multiObjectTracker('FilterInitializationFcn',@initcvekf,'AssignmentThreshold',50);
```

Model Free Space Propagation Channel

Use the free space channel to model the propagation of the transmitted and received radar signals.

```
channel = phased.FreeSpace('PropagationSpeed',c,'OperatingFrequency',fc,...
    'SampleRate',fs,'TwoWayPropagation',true);
```

In a free space model, the radar energy propagates along a direct line-of-sight between the radar and the target vehicles, as shown in the following illustration.

Simulate the Driving Scenario

Create a highway driving scenario with three vehicles traveling in the vicinity of the ego vehicle. The vehicles are modeled as point targets, and have different velocities and positions defined in the driving scenario. The ego vehicle is moving with a velocity of 80 km/hr and the other three cars are moving at 110 km/hr, 100 km/hr, and 130 km/hr respectively. For details on modeling a driving scenario, see the example “Create Actor and Vehicle Trajectories Programmatically” on page 7-442. The radar sensor is mounted on the front of the ego vehicle.

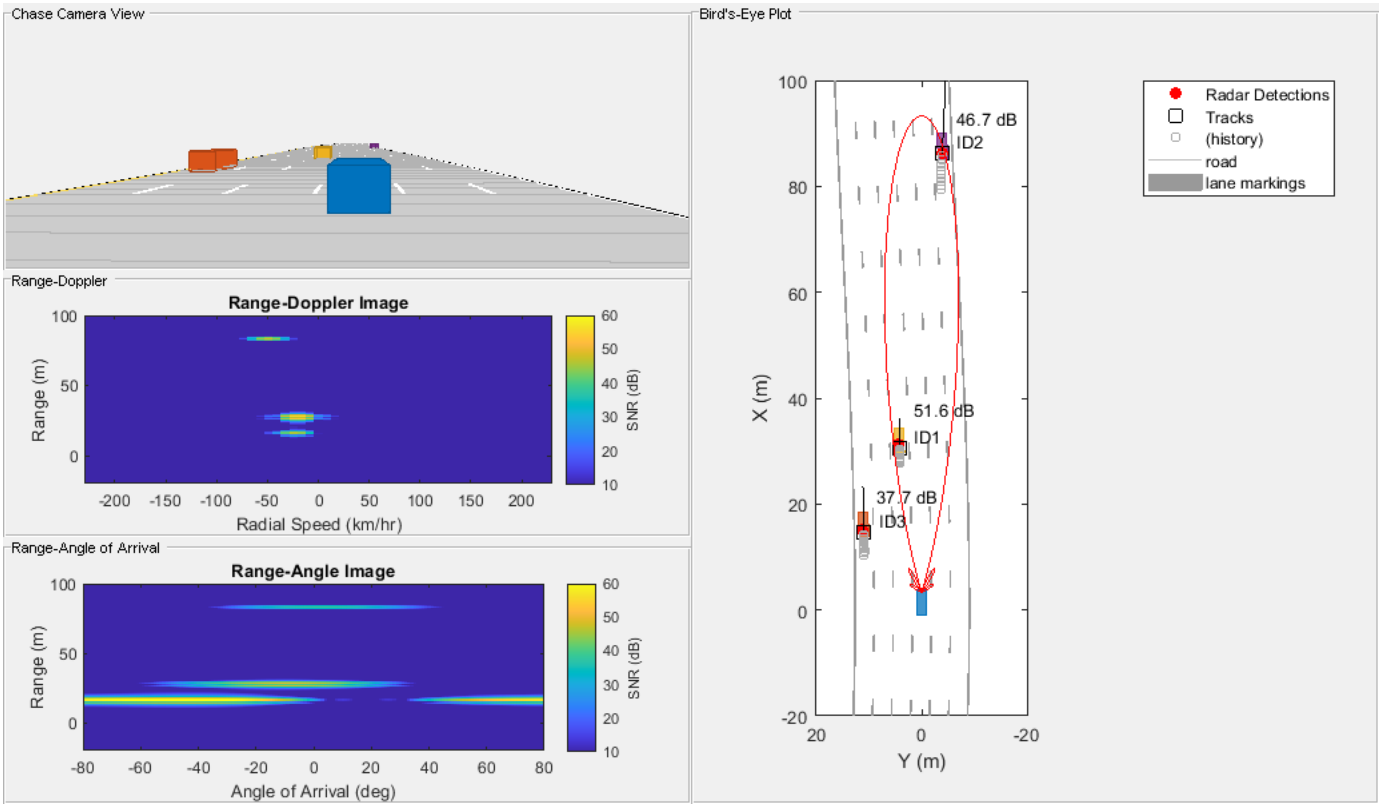
To create the driving scenario, use the `helperAutoDrivingRadarSigProc` function. To examine the contents of this function, use the `edit('helperAutoDrivingRadarSigProc')` command.

```
% Create driving scenario
[scenario,egoCar,radarParams,pointTgts] = helperAutoDrivingRadarSigProc('Setup Scenario',c,fc);
```

The following loop uses the `drivingScenario` object to advance the vehicles in the scenario. At every simulation time step, a radar data cube is assembled by collecting 192 sweeps of the radar waveform. The assembled data cube is then processed in range and Doppler. The range and Doppler processed data is then beamformed, and CFAR detection is performed on the beamformed data. Range, radial speed, and direction of arrival measurements are estimated for the CFAR detections. These detections are then assembled into `objectDetection` objects, which are then processed by the `multiObjectTracker` object.

```
% Initialize display for driving scenario example
helperAutoDrivingRadarSigProc('Initialize Display',egoCar,radarParams,rxArray,fc,vMax,rangeMax);

tgtProfiles = actorProfiles(scenario);
tgtProfiles = tgtProfiles(2:end);
tgtHeight = [tgtProfiles.Height];

% Run the simulation loop
sweepTime = waveform.SweepTime;
while advance(scenario)

    % Get the current scenario time
    time = scenario.SimulationTime;

    % Get current target poses in ego vehicle's reference frame
    tgtPoses = targetPoses(egoCar);
    tgtPos = reshape([tgtPoses.Position],3,[]);
    tgtPos(3,:) = tgtPos(3,:)+0.5*tgtHeight; % Position point targets at half of each target's he
    tgtVel = reshape([tgtPoses.Velocity],3,[]);

    % Assemble data cube at current scenario time
    Xcube = zeros(Nft,Ne,Nsweep);
    for m = 1:Nsweep

        % Calculate angles of the targets viewed by the radar
        tgtAngs = NaN(2,numel(tgtPoses));
        for iTgt = 1:numel(tgtPoses)
            tgtAxes = rotz(tgtPoses(iTgt).Yaw)*roty(tgtPoses(iTgt).Pitch)*rotx(tgtPoses(iTgt).Ro
            [~,tgtAngs(:,iTgt)] = rangeangle(radarParams.OriginPosition,tgtPos(:,iTgt),tgtAxes);
        end

        % Transmit FMCW waveform
        sig = waveform();
        [~,txang] = rangeangle(tgtPos,radarParams.OriginPosition,radarParams.Orientation);
        txsig = transmitter(sig);
        txsig = radiator(txsig,txang);
        txsig = channel(txsig,radarParams.OriginPosition,tgtPos,radarParams.OriginVelocity,tgtVe

        % Propagate the signal and reflect off the target
        tgtsig = pointTgts(txsig,tgtAngs);

        % Collect received target echos
        rxsig = collector(tgtsig,txang);
        rxsig = receiver(rxsig);

        % Dechirp the received signal
        rxsig = dechirp(rxsig,sig);

        % Save sweep to data cube
        Xcube(:,:,m) = rxsig;

        % Move targets forward in time for next sweep
        tgtPos = tgtPos+tgtVel*sweepTime;
    end

    % Calculate the range-Doppler response
    [Xrngdop,rnggrid,dopgrid] = rngdopresp(Xcube);
```

```
        % Beamform received data
        Xbf = permute(Xrngdop,[1 3 2]);
        Xbf = reshape(Xbf,Nr*Nd,Ne);
        Xbf = beamformer(Xbf);
        Xbf = reshape(Xbf,Nr,Nd);

        % Detect targets
        Xpow = abs(Xbf).^2;
        [detidx,noisepwr] = cfar(Xpow,idxCFAR);

        % Cluster detections
        clusterIDs = helperAutoDrivingRadarSigProc('Cluster Detections',detidx);

        % Estimate azimuth, range, and radial speed measurements
        [azest,azvar,snrdB] = helperAutoDrivingRadarSigProc('Estimate Angle',doaest,conj(Xrngdop),Xb
        azvar = azvar+radarParams.RMSBias(1)^2;

        [rngest,rngvar] = rngestimator(Xbf,rnggrid,detidx,noisepwr,clusterIDs);
        rngvar = rngvar+radarParams.RMSBias(2)^2;

        [rsest,rsvar] = dopestimator(Xbf,dopgrid,detidx,noisepwr,clusterIDs);

        % Convert radial speed to range rate for use by the tracker
        rrest = -rsest;
        rrvar = rsvar;
        rrvar = rrvar+radarParams.RMSBias(3)^2;

        % Assemble object detections for use by tracker
        numDets = numel(rngest);
        dets = cell(numDets,1);
        for iDet = 1:numDets
            dets{iDet} = objectDetection(time,[azest(iDet) rngest(iDet) rrest(iDet)]',...
                'MeasurementNoise',diag([azvar(iDet) rngvar(iDet) rrvar(iDet)]),...
                'MeasurementParameters',{radarParams},...
                'ObjectAttributes',{struct('SNR',snrdB(iDet))});
        end

        % Track detections
        tracks = tracker(dets,time);

        % Update displays
        helperAutoDrivingRadarSigProc('Update Display',egoCar,dets,tracks,...
            dopgrid,rnggrid,Xbf,beamscan,Xrngdop);

        % Publish snapshot
        helperAutoDrivingRadarSigProc('Publish Snapshot',time>=1.1);

        % Collect free space channel metrics
        metricsFS = helperAutoDrivingRadarSigProc('Collect Metrics',radarParams,tgtPos,tgtVel,dets);
    end
```



The previous figure shows the radar detections and tracks for the 3 target vehicles at 1.1 seconds of simulation time. The plot on the upper-left side shows the chase camera view of the driving scenario from the perspective of the ego vehicle (shown in blue). For reference, the ego vehicle is traveling at 80 km/hr and the other three cars are traveling at 110 km/hr (orange car), 100 km/hr (yellow car), and 130 km/hr (purple car).

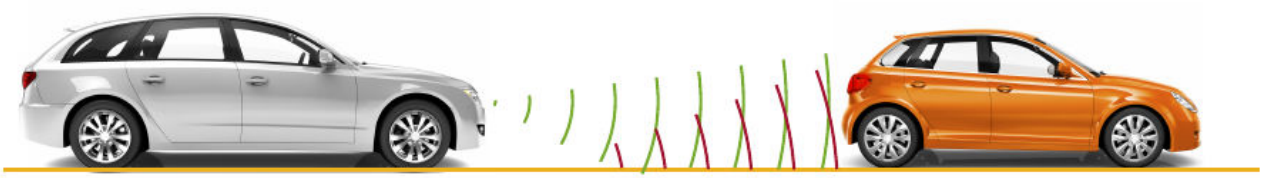
The right side of the figure shows the bird's-eye plot, which presents a "top down" perspective of the scenario. All of the vehicles, detections, and tracks are shown in the ego vehicle's coordinate reference frame. The estimated signal-to-noise ratio (SNR) for each radar measurement is printed next to each detection. The vehicle location estimated by the tracker is shown in the plot using black squares with text next to them indicating each track's ID. The tracker's estimated velocity for each vehicle is shown as a black line pointing in the direction of the vehicle's velocity. The length of the line corresponds to the estimated speed, with longer lines denoting vehicles with higher speeds relative to the ego vehicle. The purple car's track (ID2) has the longest line while the yellow car's track (ID1) has the shortest line. The tracked speeds are consistent with the modeled vehicle speeds previously listed.

The two plots on the lower-left side show the radar images generated by the signal processing. The upper plot shows how the received radar echos from the target vehicles are distributed in range and radial speed. Here, all three vehicles are observed. The measured radial speeds correspond to the velocities estimated by the tracker, as shown in the bird's-eye plot. The lower plot shows how the received target echos are spatially distributed in range and angle. Again, all three targets are present, and their locations match what is shown in the bird's-eye plot.

Due to its close proximity to the radar, the orange car can still be detected despite the large beamforming losses due to its position well outside of the beam's 3 dB beamwidth. These detections have generated a track (ID3) for the orange car.

Model a Multipath Channel

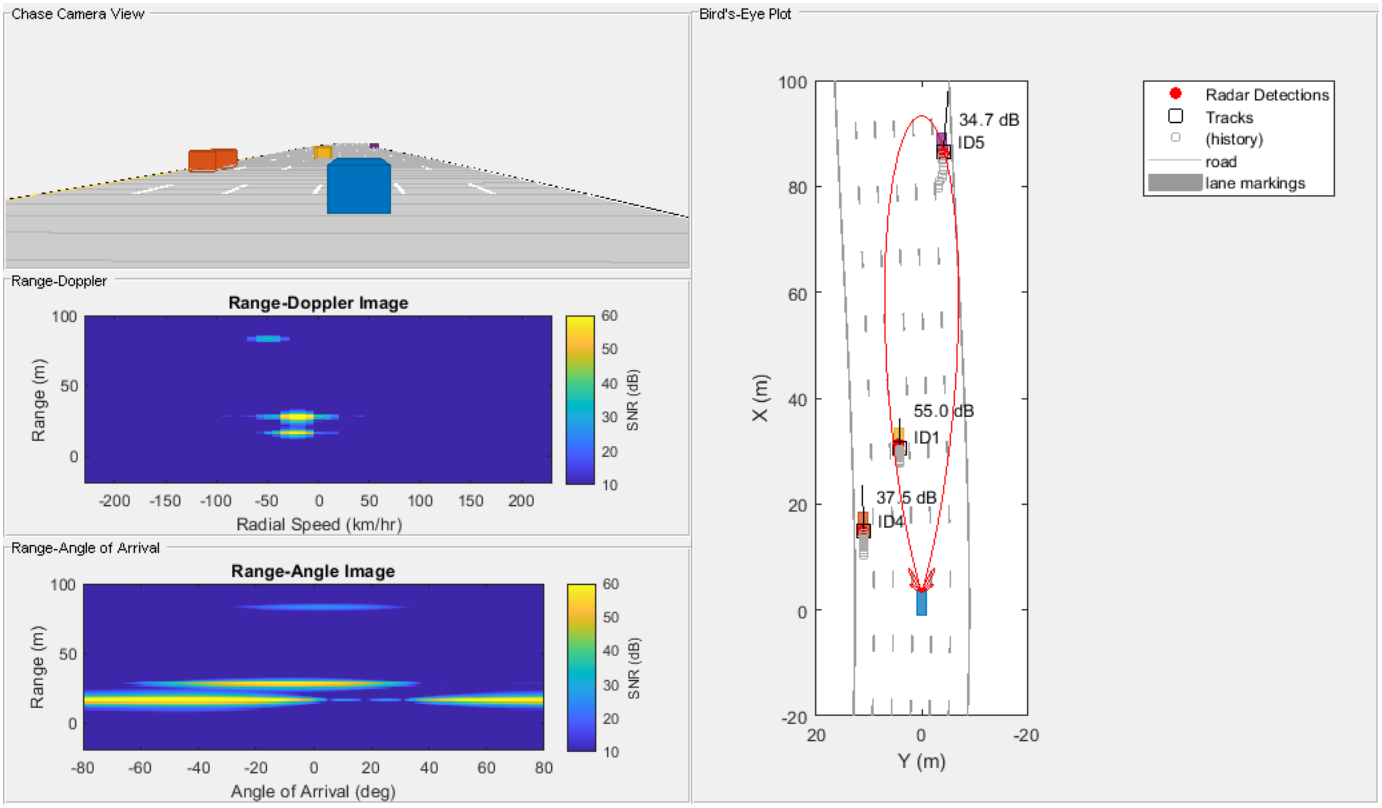
The previous driving scenario simulation used free space propagation. This is a simple model that models only direct line-of-sight propagation between the radar and each of the targets. In reality, the radar signal propagation is much more complex, involving reflections from multiple obstacles before reaching each target and returning back to the radar. This phenomenon is known as *multipath propagation*. The following illustration shows one such case of multipath propagation, where the signal impinging the target is coming from two directions: line-of-sight and a single bounce from the road surface.



The overall effect of multipath propagation is that the received radar echoes can interfere constructively and destructively. This constructive and destructive interference results from path length differences between the various signal propagation paths. As the distance between the radar and the vehicles changes, these path length differences also change. When the differences between these paths result in echos received by the radar that are almost 180 degrees out of phase, the echos destructively combine, and the radar makes no detection for that range.

Replace the free space channel model with a two-ray channel model to demonstrate the propagation environment shown in the previous illustration. Reuse the remaining parameters in the driving scenario and radar model, and run the simulation again.

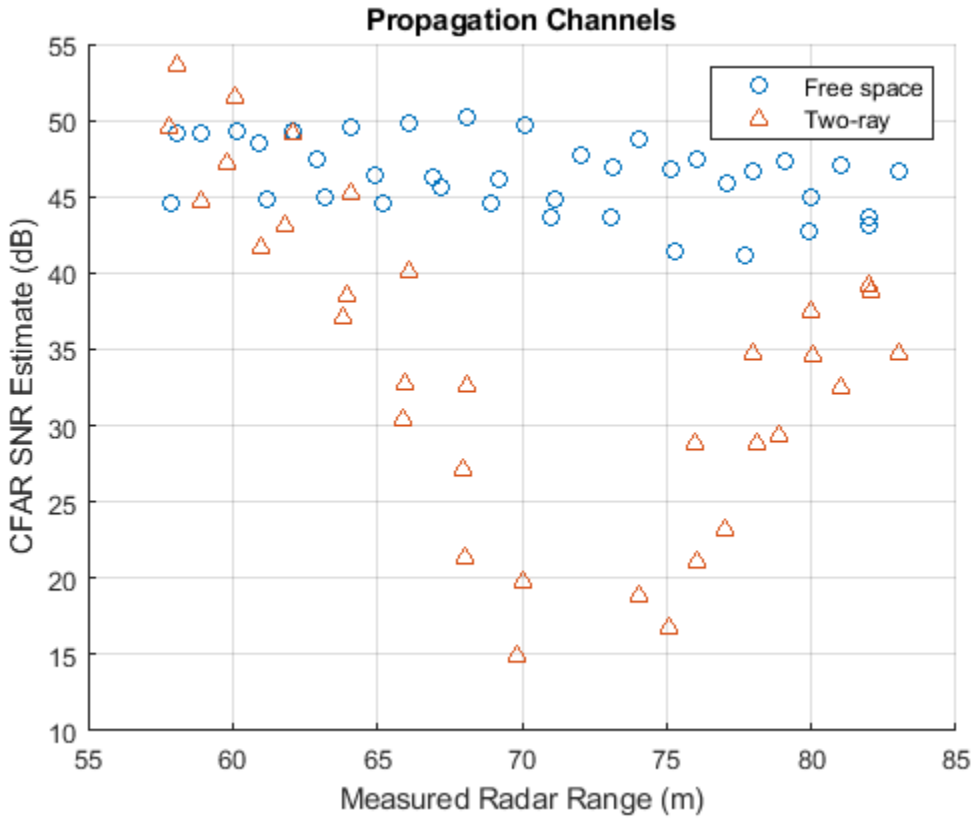
```
% Reset the driving scenario
[scenario,egoCar,radarParams,pointTgts] = helperAutoDrivingRadarSigProc('Setup Scenario',c,fc);

% Create two-ray propagation channels. One channel is used for the transmit
% path and a different channel is used for the receive path.
txchannel = phased.TwoRayChannel('PropagationSpeed',c,'OperatingFrequency',fc,'SampleRate',fs);
rxchannel = phased.TwoRayChannel('PropagationSpeed',c,'OperatingFrequency',fc,'SampleRate',fs);

% Run the simulation again, now using the two-ray channel model
metrics2Ray = helperAutoDrivingRadarSigProc('Run Simulation',...
    c,fc,rangeMax,vMax,waveform,Nsweep,...                  % Waveform parameters
    transmitter,radiator,collector,receiver,...            % Hardware models
    rngdopresp,beamformer,cfar,idxCFAR,...                 % Signal processing
    rngestimator,dopestimator,doaest,beamscan,tracker,...  % Estimation
    txchannel,rxchannel);                                  % Propagation channel models
```



The previous figure shows the chase plot, bird's-eye plot, and radar images at 1.1 seconds of simulation time, just as was shown for the free space channel propagation scenario. Comparing these two figures, observe that for the two-ray channel, no detection is present for the purple car at this simulation time. This detection loss is because the path length differences for this car are destructively interfering at this range, resulting in a total loss of detection.

Plot the SNR estimates generated from the CFAR processing against the purple car's range estimates from the free space and two-ray channel simulations.

```
helperAutoDrivingRadarSigProc('Plot Channels',metricsFS,metrics2Ray);
```



As the car approaches a range of 72 meters from the radar, a large loss in the estimated SNR from the two-ray channel is observed with respect to the free space channel. It is near this range that the multipath interference combines destructively, resulting in a loss in signal detections. However, observe that the tracker is able to coast the track during these times of signal loss and provide a predicted position and velocity for the purple car.

Summary

This example demonstrated how to model an automotive radar's hardware and signal processing using Phased Array System Toolbox. You also learned how to integrate this radar model with the Automated Driving Toolbox driving scenario simulation. First you generated synthetic radar detections. Then you processed these detections further by using a tracker to generate precise position and velocity estimates in the ego vehicle's coordinate frame. Finally, you learned how to simulate multipath propagation effects by using the `phased.TwoRayChannel` model provided in Phased Array System Toolbox.

The presented workflow enables you to understand how your radar architecture design decisions impact higher-level system requirements. Using this workflow enables you to select a radar design that satisfies your unique application requirements.

Reference

[1] Richards, Mark. *Fundamentals of Radar Signal Processing*. New York: McGraw Hill, 2005.

See Also

Objects

`multiObjectTracker` | `phased.DopplerEstimator` | `phased.FMCWWaveform` | `phased.FreeSpace` | `phased.RootMUSICEstimator` | `phased.TwoRayChannel` | `phased.ULA`

More About

- “Sensor Fusion Using Synthetic Radar and Vision Data” on page 7-196
- “Automotive Adaptive Cruise Control Using FMCW Technology” (Phased Array System Toolbox)
- “Constant False Alarm Rate (CFAR) Detection” (Phased Array System Toolbox)
- “Radar Data Cube” (Phased Array System Toolbox)
- “Create Actor and Vehicle Trajectories Programmatically” on page 7-442

Create Driving Scenario Programmatically

This example shows how to generate ground truth for synthetic sensor data and tracking algorithms. It also shows how to update actor poses in open-loop and closed-loop simulations. Finally, it shows how to use the driving scenario to perform coordinate conversion and incorporate them into the bird's-eye plot.

In this example, you programmatically create the driving scenario from the MATLAB® command line. Alternatively, you can create scenarios interactively by using the Driving Scenario Designer app. For an example, see “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2.

Introduction

One of the goals of a driving scenario is to generate "ground truth" test cases for use with sensor detection and tracking algorithms used on a specific vehicle.

This ground truth is typically defined in a global coordinate system; but, because sensors are typically mounted on a moving vehicle, this data needs to be converted to a reference frame that moves along with the vehicle. The driving scenario facilitates this conversion automatically, allowing you to specify roads and trajectories of objects in global coordinates and provides tools to convert and visualize this information in the reference frame of any actor in the scenario.

Convert Pose Information to an Actor's Reference Frame

A `drivingScenario` consists of a model of roads and movable objects, called actors. You can use actors to model pedestrians, parking meters, fire hydrants, and other objects within the scenario. Actors consist of cuboids with a length, width, height, and a radar cross-section (RCS). An actor is positioned and oriented about a single point in the center of its bottom face.

A special kind of actor that moves on wheels is a vehicle, which is positioned and oriented on the ground directly beneath the center of the rear axle, which is a more natural center of rotation.

All actors (including vehicles) may be placed anywhere within the scenario by specifying their respective `Position`, `Roll`, `Pitch`, `Yaw`, `Velocity`, and `AngularVelocity` properties.

Here is an example of a scenario consisting of two vehicles 10 meters apart and driving towards the origin at a speed of 3 and 4 meters per second, respectively:

```
scenario = drivingScenario;
v1 = vehicle(scenario,'ClassID',1,'Position',[6 0 0],'Velocity',[-3 0 0],'Yaw',180)
```

```
v1 =

  Vehicle with properties:

         FrontOverhang: 0.9000
          RearOverhang: 1
             Wheelbase: 2.8000
             EntryTime: 0
              ExitTime: Inf
               ActorID: 1
               ClassID: 1
                  Name: ""
```

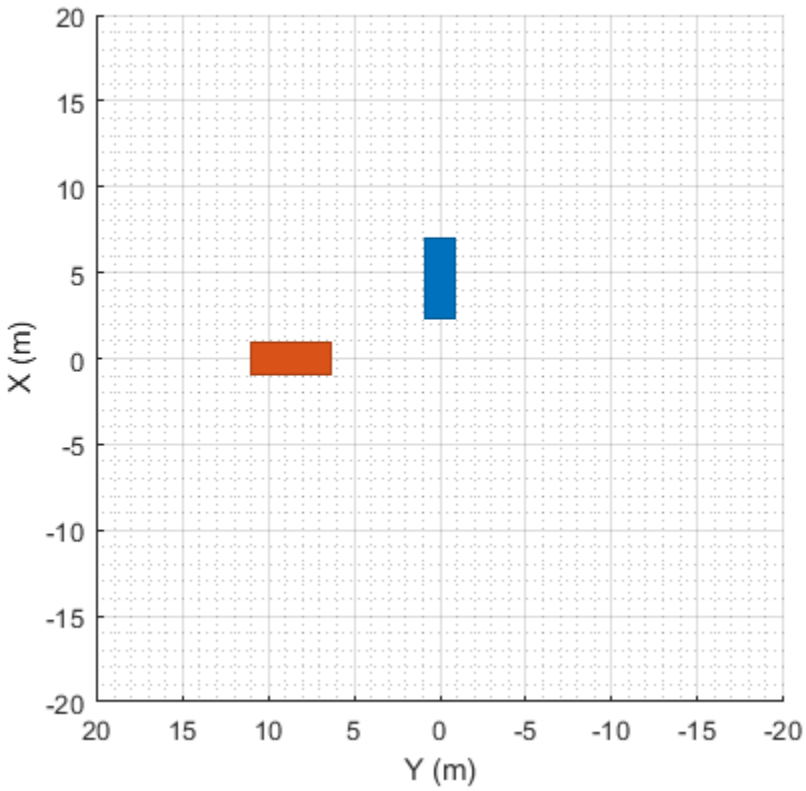
```
             PlotColor: [0 0.4470 0.7410]
              Position: [6 0 0]
              Velocity: [-3 0 0]
                   Yaw: 180
                 Pitch: 0
                  Roll: 0
       AngularVelocity: [0 0 0]
                Length: 4.7000
                 Width: 1.8000
                Height: 1.4000
                  Mesh: [1x1 extendedObjectMesh]
           RCSPattern: [2x2 double]
     RCSAzimuthAngles: [-180 180]
   RCSElevationAngles: [-90 90]
```

```
v2 = vehicle(scenario,'ClassID',1,'Position',[0 10 0],'Velocity',[0 -4 0],'Yaw',-90)
```

```
v2 =

  Vehicle with properties:

         FrontOverhang: 0.9000
          RearOverhang: 1
             Wheelbase: 2.8000
             EntryTime: 0
              ExitTime: Inf
               ActorID: 2
               ClassID: 1
                  Name: ""
             PlotColor: [0.8500 0.3250 0.0980]
              Position: [0 10 0]
              Velocity: [0 -4 0]
                   Yaw: -90
                 Pitch: 0
                  Roll: 0
       AngularVelocity: [0 0 0]
                Length: 4.7000
                 Width: 1.8000
                Height: 1.4000
                  Mesh: [1x1 extendedObjectMesh]
           RCSPattern: [2x2 double]
     RCSAzimuthAngles: [-180 180]
   RCSElevationAngles: [-90 90]
```

To visualize a scenario, call the `plot` function on it:

```
plot(scenario);
set(gcf,'Name','Scenario Plot')
xlim([-20 20]);
ylim([-20 20]);
```



Once all the actors in a scenario have been created, you can inspect the pose information of all the actors in the coordinates of the scenario by inspecting the `Position`, `Roll`, `Pitch`, `Yaw`, `Velocity`, and `AngularVelocity` properties of each actor, or you may obtain all of them in a convenient structure by calling the `actorPoses` function on the scenario:

```
ap = actorPoses(scenario)
```

```
ap =

  2x1 struct array with fields:

    ActorID
    Position
    Velocity
    Roll
    Pitch
    Yaw
    AngularVelocity
```

To obtain the pose information of all other objects (or targets) seen by a specific actor in its own reference frame, you can call the `targetPoses` function on the actor itself:

```
v2TargetPoses = targetPoses(v2)
```

```
v2TargetPoses =
```

```
  struct with fields:

           ActorID: 1
           ClassID: 1
          Position: [10 6.0000 0]
          Velocity: [-4 -3.0000 0]
              Roll: 0
             Pitch: 0
               Yaw: -90.0000
    AngularVelocity: [0 0 0]
```

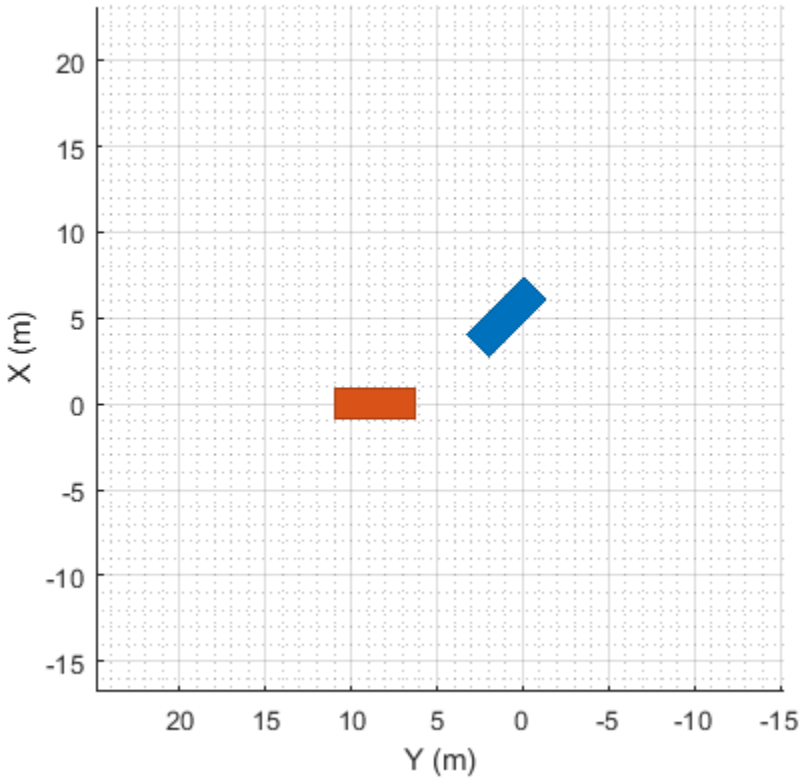
We can qualitatively confirm the relative vehicle placement by adding a chase plot for a vehicle. By default, a chase plot displays a projective-perspective view from a fixed distance behind the vehicle.

Here we show the perspective seen just behind the second vehicle (red). The target poses seen by the second vehicle show that the location of the other vehicle (in blue) is 6 m forward and 10 m to the left of the second vehicle. We can see this qualitatively in the chase plot:

```
chasePlot(v2)
set(gcf,'Name','Chase Plot')
```

Normally all plots associated with a driving scenario are updated in the course of simulation when calling the `advance` function. If you update a position property of another actor manually, you can call `updatePlots` to see the results immediately:

```
v1.Yaw = 135;
updatePlots(scenario);
```

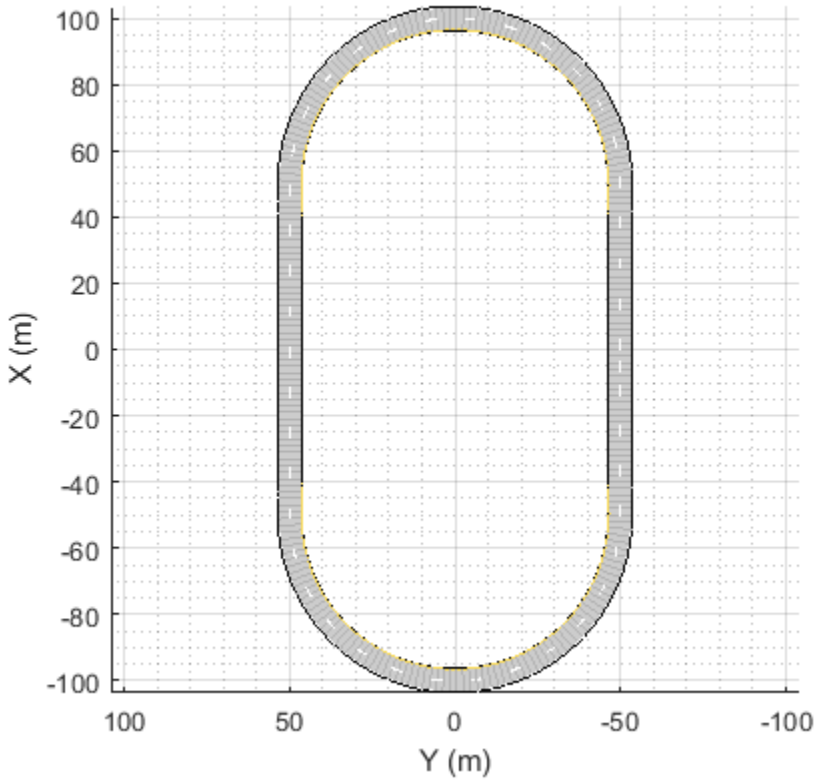


Convert Road Boundaries to an Actor's Reference Frame

The driving scenario can also be used to retrieve the boundaries of roads defined in the scenario.

Here we make use of the simple oval track described in “Define Road Layouts Programmatically” on page 7-453, which covers an area roughly 200 meters long and 100 meters wide and whose curves have a bank angle of nine degrees:

```
scenario = drivingScenario;
roadCenters = ...
    [  0  40  49  50 100  50  49 40 -40 -49 -50 -100  -50  -49  -40    0
     -50 -50 -50 -50   0  50  50 50  50  50  50    0  -50  -50  -50  -50
       0   0 .45 .45 .45 .45 .45  0   0 .45 .45  .45  .45  .45    0    0]';
bankAngles = ...
    [  0   0   9   9   9   9   9  0   0   9   9    9    9    9    0    0];

road(scenario, roadCenters, bankAngles, 'lanes', lanespec(2));
plot(scenario);
```



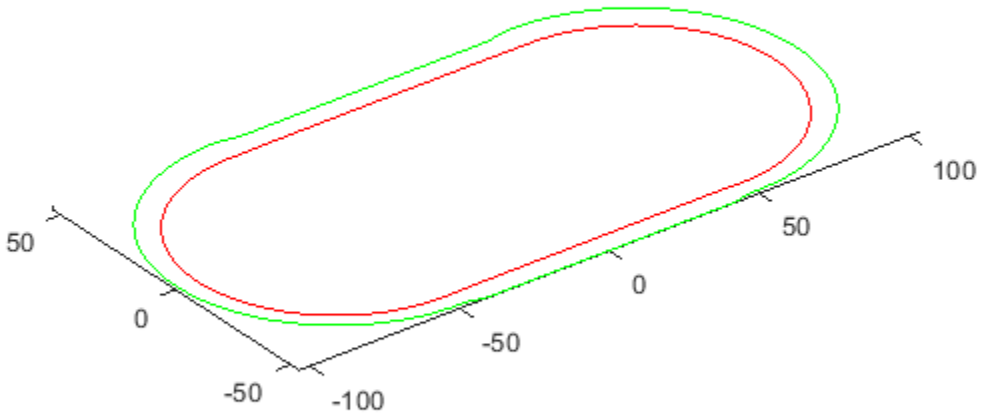
To obtain the lines that define the borders of the road, use the `roadBoundaries` function on the driving scenario. It returns a cell array that contains the road borders (shown in the scenario plot above as the solid black lines).

```
rb = roadBoundaries(scenario)
```

```
rb =

  1x2 cell array

    {258x3 double}    {258x3 double}
```

In the example above, there are two road boundaries (an outer and an inner boundary). You can plot them yourself as follows:

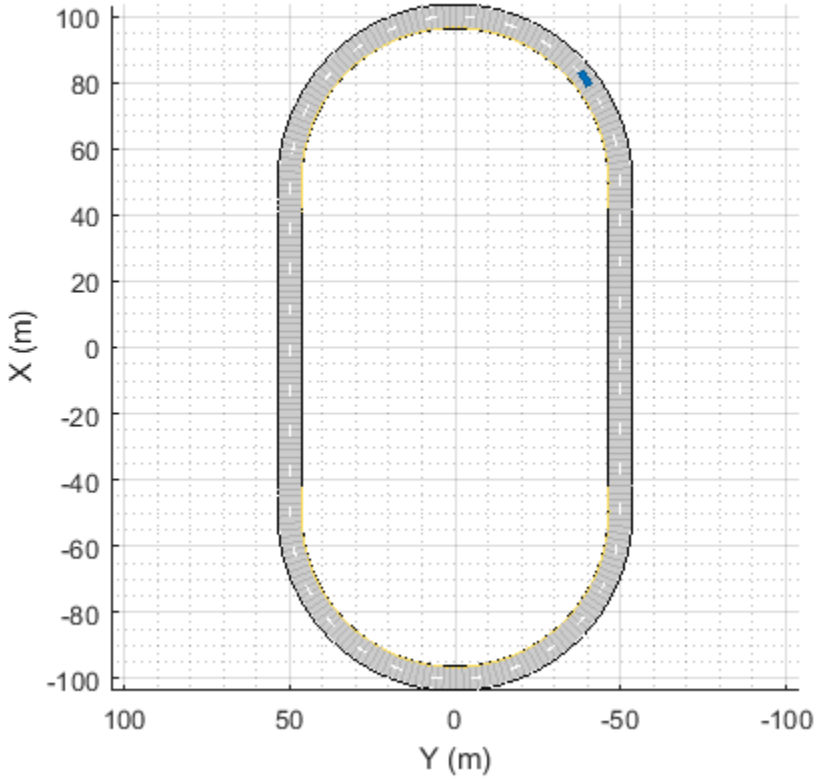
```
figure

outerBoundary = rb{1};
innerBoundary = rb{2};

plot3(innerBoundary(:,1),innerBoundary(:,2),innerBoundary(:,3),'r', ...
      outerBoundary(:,1),outerBoundary(:,2),outerBoundary(:,3),'g')
axis equal
```



You can use the `roadBoundaries` function on an actor to obtain the road boundaries in the coordinates of the actor. To do that, simply pass the actor as the first argument, instead of the scenario.

To see this, add an "ego vehicle" and place it on the track:

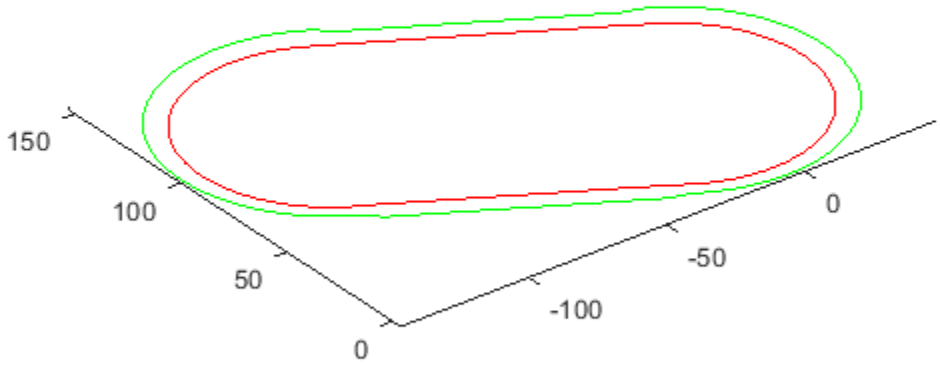
```
egoCar = vehicle(scenario,'ClassID',1,'Position',[80 -40 0.45],'Yaw',30);
```



Next, call the `roadBoundaries` function on the vehicle and plot it as before. It will be rendered relative to the vehicle's coordinates:

```
figure

rb = roadBoundaries(egoCar)
outerBoundary = rb{1};
innerBoundary = rb{2};

plot3(innerBoundary(:,1),innerBoundary(:,2),innerBoundary(:,3),'r', ...
      outerBoundary(:,1),outerBoundary(:,2),outerBoundary(:,3),'g')
axis equal
```

```
rb =

  1x2 cell array

    {258x3 double}    {258x3 double}
```

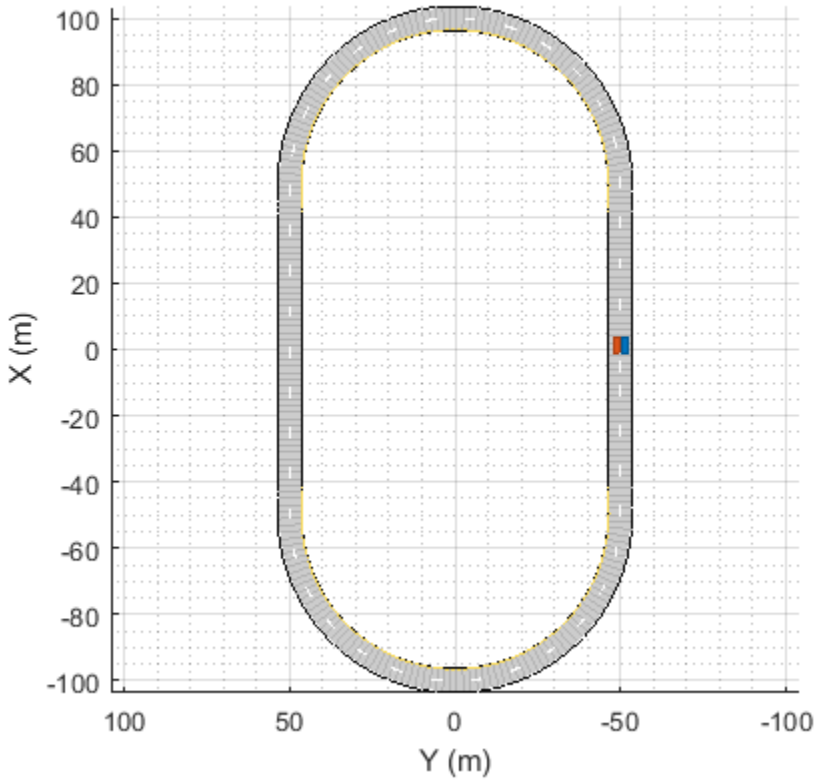


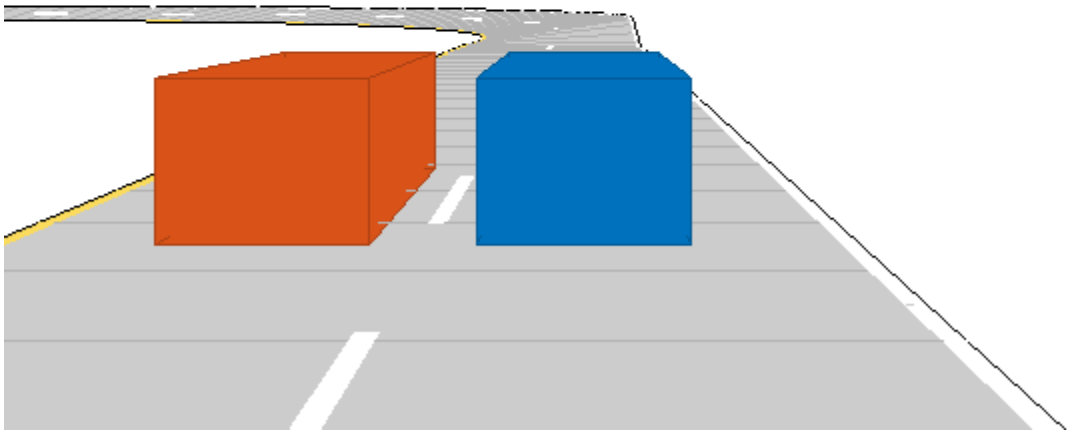
Specify Actor Trajectory

You can position and plot any specific actor along a predefined three-dimensional path.

Here is an example for two vehicles that follow the racetrack at 30 m/s and 50 m/s respectively, each in its own respective lane. We offset the cars from the center of the road by setting the offset position by half a lane width of 2.7 meters, and, for the banked angle sections of the track, half the vertical height on each side:

```
chasePlot(egoCar);
fastCar = vehicle(scenario,'ClassID',1);

d = 2.7/2;
h = .45/2;
roadOffset = [ 0  0  0  0  d  0  0  0  0  0  0 -d  0  0  0  0
              -d -d -d -d  0  d  d  d  d  d  d  0 -d -d -d -d
               0  0  h  h  h  h  h  0  0  h  h  h  h  h  0  0]';

rWayPoints = roadCenters + roadOffset;
lWayPoints = roadCenters - roadOffset;

% loop around the track four times
rWayPoints = [repmat(rWayPoints(1:end-1,:),5,1); rWayPoints(1,:)];
lWayPoints = [repmat(lWayPoints(1:end-1,:),5,1); lWayPoints(1,:)];

trajectory(egoCar,rWayPoints(:,:), 30);
trajectory(fastCar,lWayPoints(:,:), 50);
```


100
80
60
40
20
0
-20
-40
-60
-80
-100
X (m)
100
50
0
-50
-100
Y (m)



Advance the Simulation

Actors that follow a trajectory are updated by calling `advance` on the driving scenario. When `advance` is called, each actor that is following a trajectory will move forward, and the corresponding plots will be updated. Only actors that have defined trajectories actually update. This is so you can provide your own logic while simulation is running.

The `SampleTime` property in the scenario governs the interval of time between updates. By default it is 10 milliseconds, but you may specify it with arbitrary resolution:

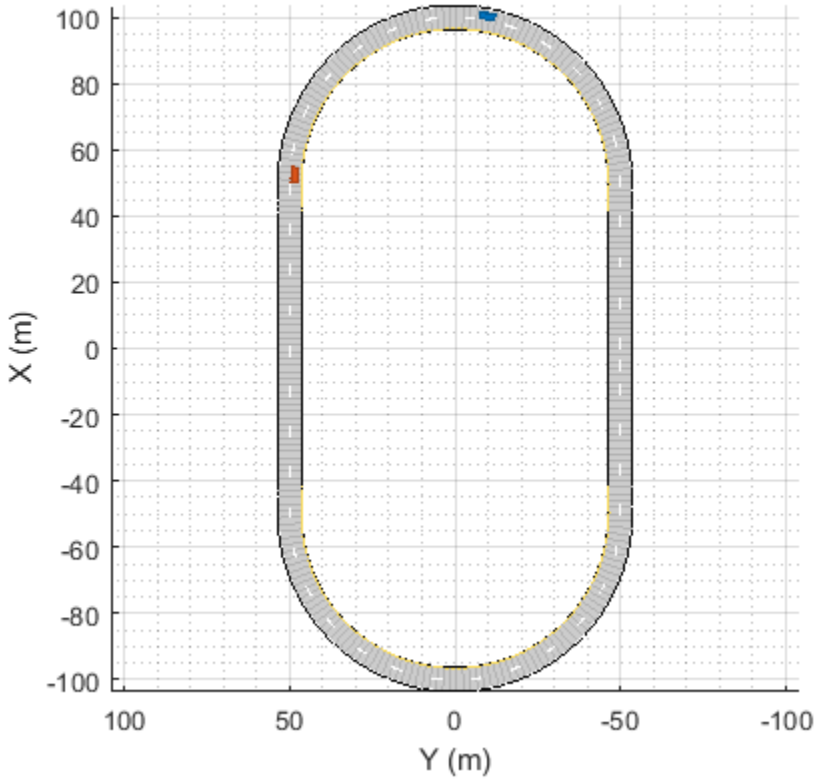
```
scenario.SampleTime = 0.02
```

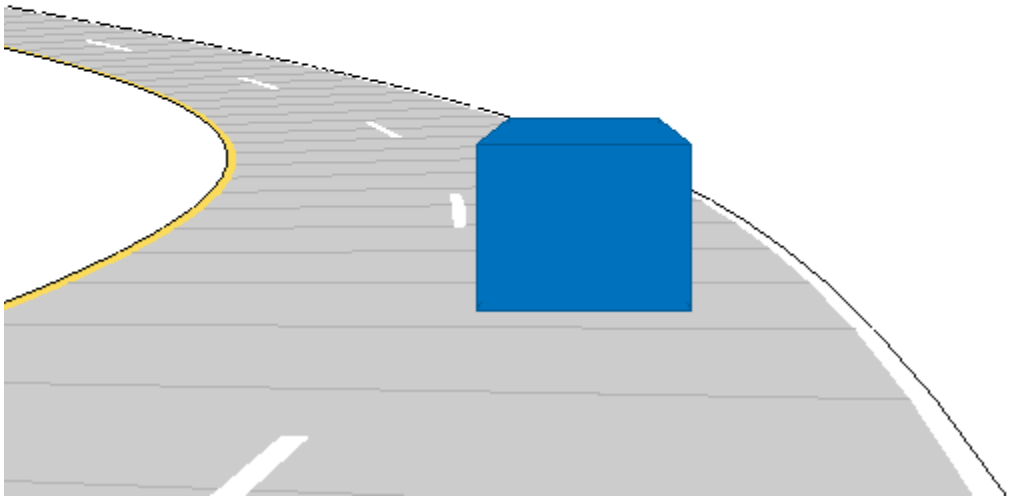
```
scenario =

  drivingScenario with properties:

       SampleTime: 0.0200
         StopTime: Inf
   SimulationTime: 0
        IsRunning: 1
           Actors: [1x2 driving.scenario.Vehicle]
```

You can run the simulation by calling `advance` in the conditional of a while loop and placing commands to inspect or modify the scenario within the body of the loop.

The while loop will automatically terminate when the trajectory for any vehicle has finished or an optional `StopTime` has been reached.

```
scenario.StopTime = 4;
while advance(scenario)
  pause(0.001)
end
```





Record a Scenario

As a convenience when the trajectories of all actors are known in advance, you can call the `record` function on the scenario to return a structure that contains the pose information of each actor at each time-step.

For example, you can inspect the pose information of each actor for the first 100 milliseconds of the simulation, and inspect the fifth recorded sample:

```
close all

scenario.StopTime = 0.100;
poseRecord = record(scenario)

r = poseRecord(5)
r.ActorPoses(1)
r.ActorPoses(2)
```

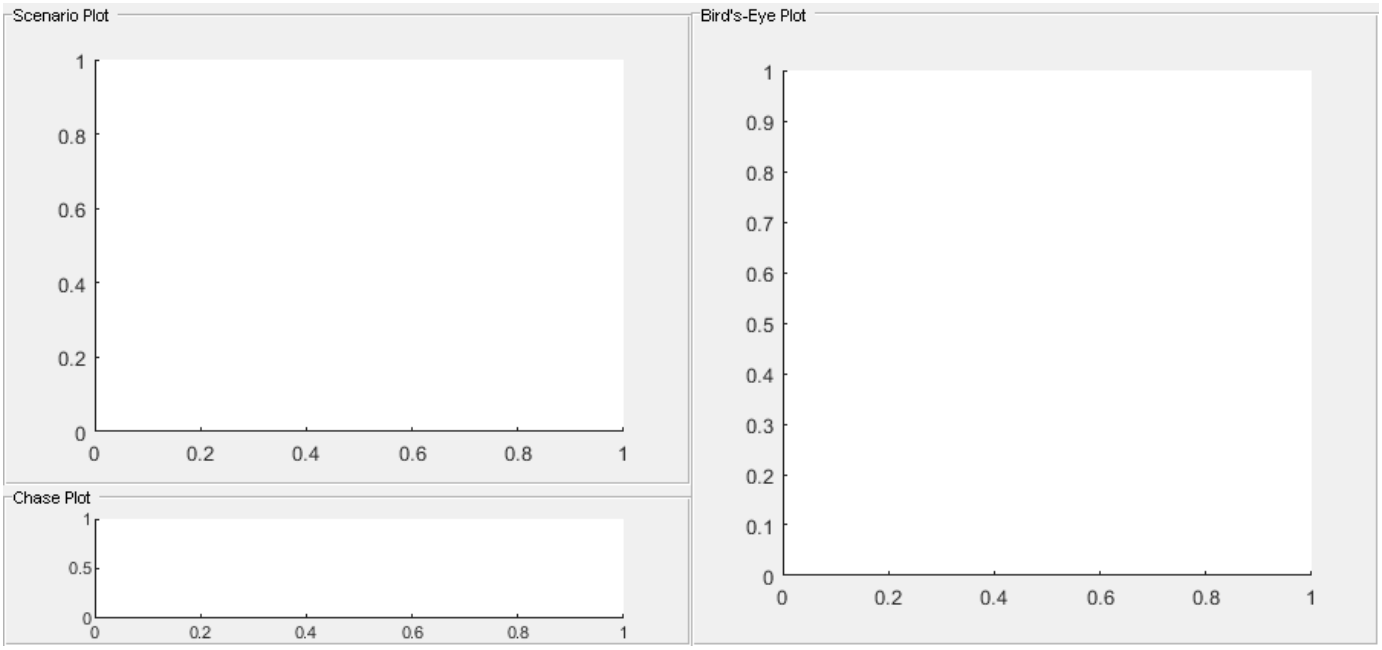
```
poseRecord =

  1x5 struct array with fields:

    SimulationTime
    ActorPoses
```

```
r =

  struct with fields:

    SimulationTime: 0.0800
        ActorPoses: [2x1 struct]


ans =

  struct with fields:

           ActorID: 1
          Position: [2.4000 -51.3502 0]
          Velocity: [30.0000 -0.0038 0]
              Roll: 0
             Pitch: 0
               Yaw: -0.0073
    AngularVelocity: [0 0 -0.0823]


ans =

  struct with fields:

           ActorID: 2
          Position: [4.0000 -48.6504 0]
          Velocity: [50.0000 -0.0105 0]
              Roll: 0
             Pitch: 0
               Yaw: -0.0120
    AngularVelocity: [0 0 -0.1235]
```

Incorporating Multiple Views with the Bird's Eye Plot

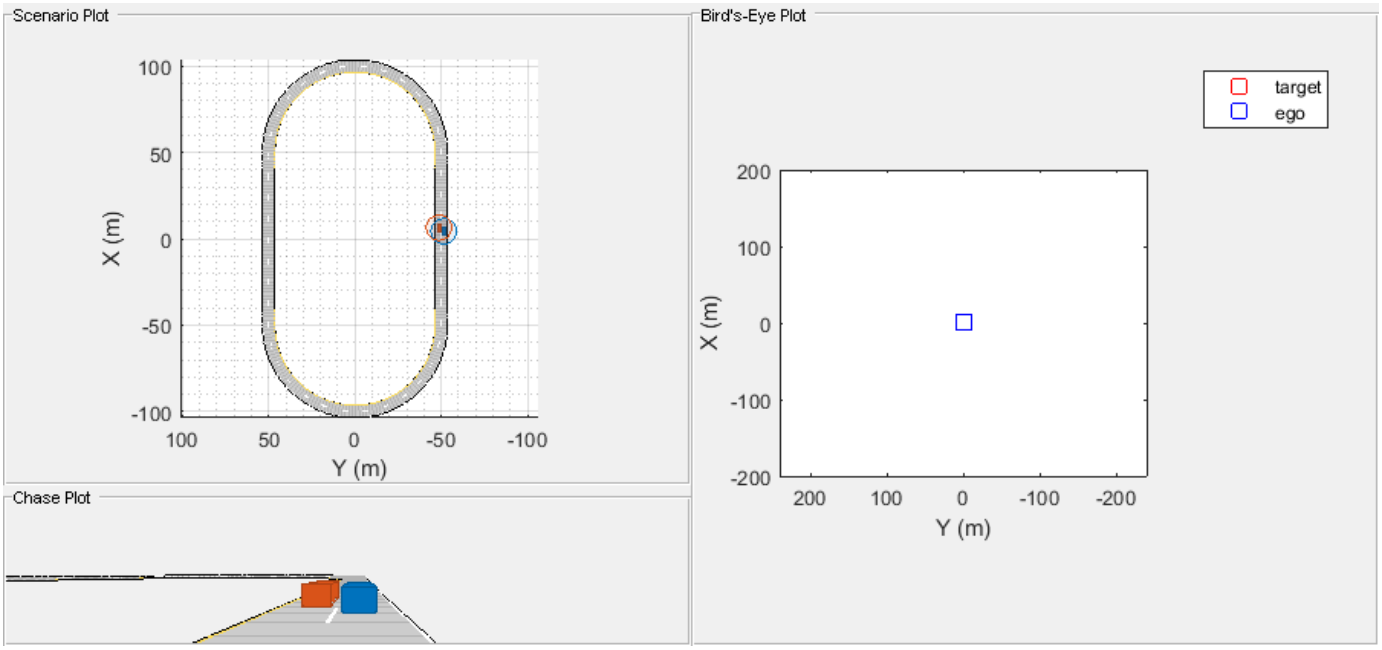
When debugging the simulation, you may wish to report the "ground truth" data in the bird's-eye plot of a specific actor while simultaneously viewing the plots generated by the scenario. To do this, you can first create a figure with axes placed in a custom arrangement:

```
close all;
hFigure = figure;
hFigure.Position(3) = 900;

hPanel1 = uipanel(hFigure,'Units','Normalized','Position',[0 1/4 1/2 3/4],'Title','Scenario Plot
hPanel2 = uipanel(hFigure,'Units','Normalized','Position',[0 0 1/2 1/4],'Title','Chase Plot');
hPanel3 = uipanel(hFigure,'Units','Normalized','Position',[1/2 0 1/2 1],'Title','Bird''s-Eye Plo

hAxes1 = axes('Parent',hPanel1);
hAxes2 = axes('Parent',hPanel2);
hAxes3 = axes('Parent',hPanel3);
```



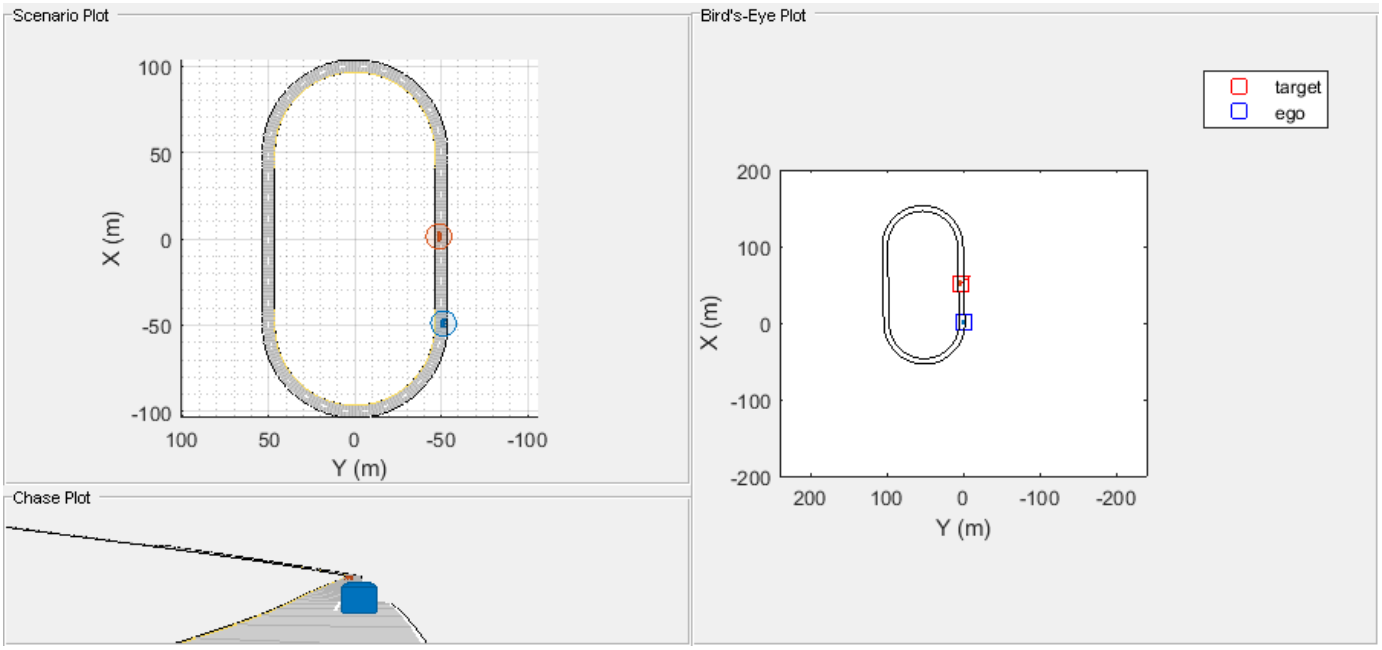
Once you have the axes defined, you specify them via the `Parent` property when creating the plots:

```
% assign scenario plot to first axes and add indicators for ActorIDs 1 and 2
plot(scenario, 'Parent', hAxes1,'ActorIndicators',[1 2]);

% assign chase plot to second axes
chasePlot(egoCar, 'Parent', hAxes2);

% assign bird's-eye plot to third axes
egoCarBEP = birdsEyePlot('Parent',hAxes3,'XLimits',[-200 200],'YLimits',[-240 240]);
fastTrackPlotter = trackPlotter(egoCarBEP,'MarkerEdgeColor','red','DisplayName','target','Velocit
egoTrackPlotter = trackPlotter(egoCarBEP,'MarkerEdgeColor','blue','DisplayName','ego','VelocityS
egoLanePlotter = laneBoundaryPlotter(egoCarBEP);
plotTrack(egoTrackPlotter, [0 0]);
egoOutlinePlotter = outlinePlotter(egoCarBEP);
```



You now can restart the simulation and run it to completion, this time extracting the positional information of the target car via `targetPoses` and display it in the bird's-eye plot. Similarly, you can also call `roadBoundaries` and `targetOutlines` directly from the ego vehicle to extract the road boundaries and the outlines of the actors. The bird's-eye plot is capable of displaying the results of these functions directly:

```
restart(scenario)
scenario.StopTime = Inf;

while advance(scenario)
    t = targetPoses(egoCar);
    plotTrack(fastTrackPlotter, t.Position, t.Velocity);
    rbs = roadBoundaries(egoCar);
    plotLaneBoundary(egoLanePlotter, rbs);
    [position, yaw, length, width, originOffset, color] = targetOutlines(egoCar);
    plotOutline(egoOutlinePlotter, position, yaw, length, width, 'OriginOffset', originOffset, '
end
```



Next Steps

This example showed how to generate and visualize ground truth for synthetic sensor data and tracking algorithms using a `drivingScenario` object. To simulate, visualize, or modify this driving scenario in an interactive environment, try importing the `drivingScenario` object into the Driving Scenario Designer app:

```
drivingScenarioDesigner(scenario)
```

Further Information

For more information on how to define actors and roads, see “Create Actor and Vehicle Trajectories Programmatically” on page 7-442 and “Define Road Layouts Programmatically” on page 7-453.

For a more in-depth example on how to use the bird's-eye plot with detections and tracks, see “Visualize Sensor Coverage, Detections, and Tracks” on page 7-226.

For examples that use the driving scenario to assist in generating synthetic data, see “Model Radar Sensor Detections” on page 7-377, “Model Vision Sensor Detections” on page 7-393, and “Sensor Fusion Using Synthetic Radar and Vision Data” on page 7-196.

See Also

Apps
Driving Scenario Designer

Objects
`birdsEyePlot` | `drivingScenario`

Functions
`actorPoses` | `chasePlot` | `record` | `road` | `roadBoundaries` | `targetPoses` | `updatePlots` | `vehicle`

More About

- “Create Driving Scenario Variations Programmatically” on page 5-107
- “Create Actor and Vehicle Trajectories Programmatically” on page 7-442
- “Define Road Layouts Programmatically” on page 7-453
- “Scenario Generation from Recorded Vehicle Data” on page 7-350
- “Automatic Scenario Generation” on page 7-722

Create Actor and Vehicle Trajectories Programmatically

This example shows how to programmatically create actor and vehicle trajectories for a driving scenario using Automated Driving Toolbox™ functions. To create actor and vehicle trajectories interactively, use the Driving Scenario Designer app.

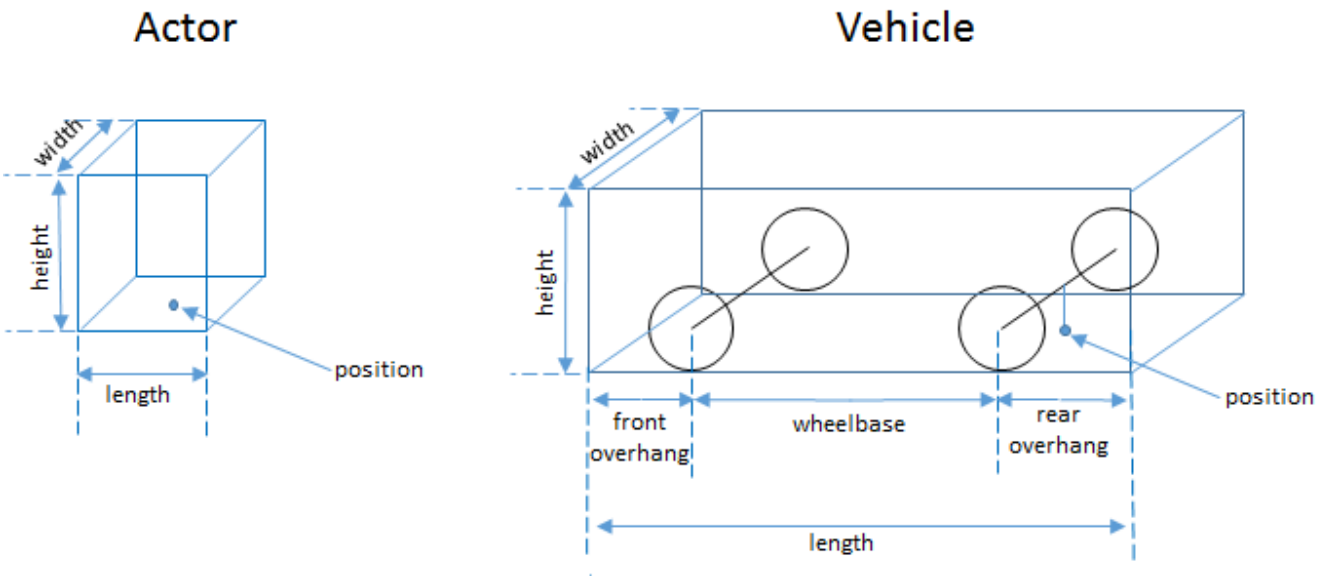
Actors and Vehicles

Actors in a driving scenario are defined as cuboid objects with a specific length, width, and height. Actors also have a radar cross section (specified in dBsm) which you can refine by defining angular coordinates (azimuth and elevation). Cuboid driving scenarios define the position of an actor as the center of its bottom face. Driving scenarios use this point as the point of contact of the actor with the ground. This point is also the rotational center of the actor.

A vehicle is a special kind of actor that moves on wheels. Vehicles possess three extra properties that govern the placement of the front and rear axle.

- The *wheelbase* is the distance between the front and rear axles.
- The *front overhang* is the amount of distance between the front axle and the front of the vehicle.
- The *rear overhang* is the distance between the rear axle and the rear of the vehicle.

Unlike actors, the position of a vehicle is on the ground at the center of the rear axle. This position corresponds to the natural center of rotation of the vehicle.



This table shows a typical list of actors and their corresponding dimensions:

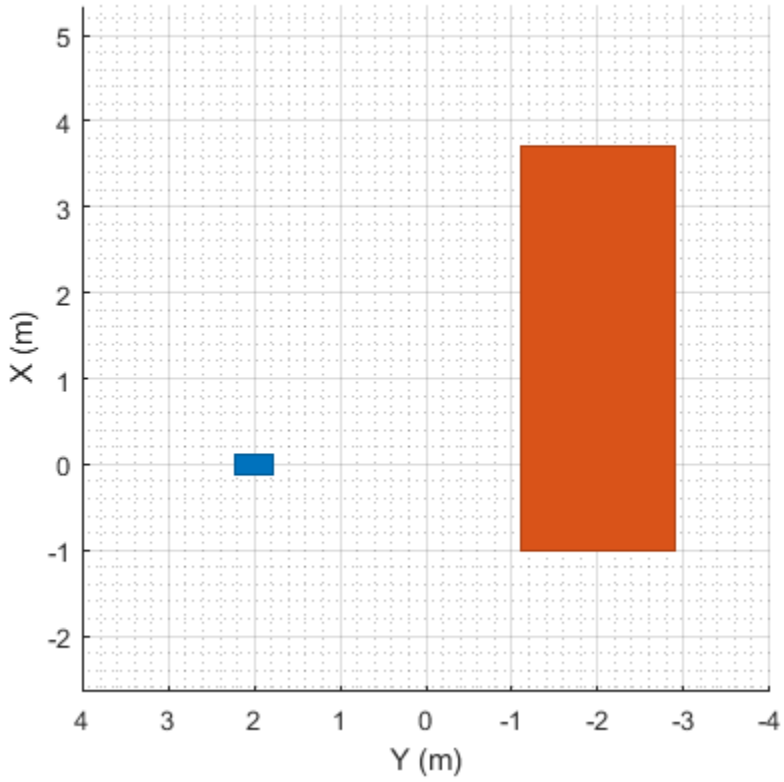
actor classification	length	width	height	wheelbase	front overhang	rear overhang	radar cross-section
pedestrian	0.24 m	0.45 m	1.7 m	N/A	N/A	N/A	-8 dBsm
automobile	4.7 m	1.8 m	1.4 m	2.8 m	0.9 m	1.0 m	10 dBsm
motorcycle	2.2 m	0.6 m	1.5 m	1.51 m	0.37 m	0.32 m	0 dBsm

This code plots the position of an actor, with the dimensions of a typical human, and a vehicle in a driving scenario. The actor and vehicle are located at positions (0, 2) and (0, –2), respectively.

```
scenario = drivingScenario;
a = actor(scenario,'ClassID',1,'Length',0.24,'Width',0.45,'Height',1.7);
passingCar = vehicle(scenario,'ClassID',1);
a.Position = [0 2 0]
passingCar.Position = [0 -2 0]
plot(scenario)
ylim([-4 4])
```

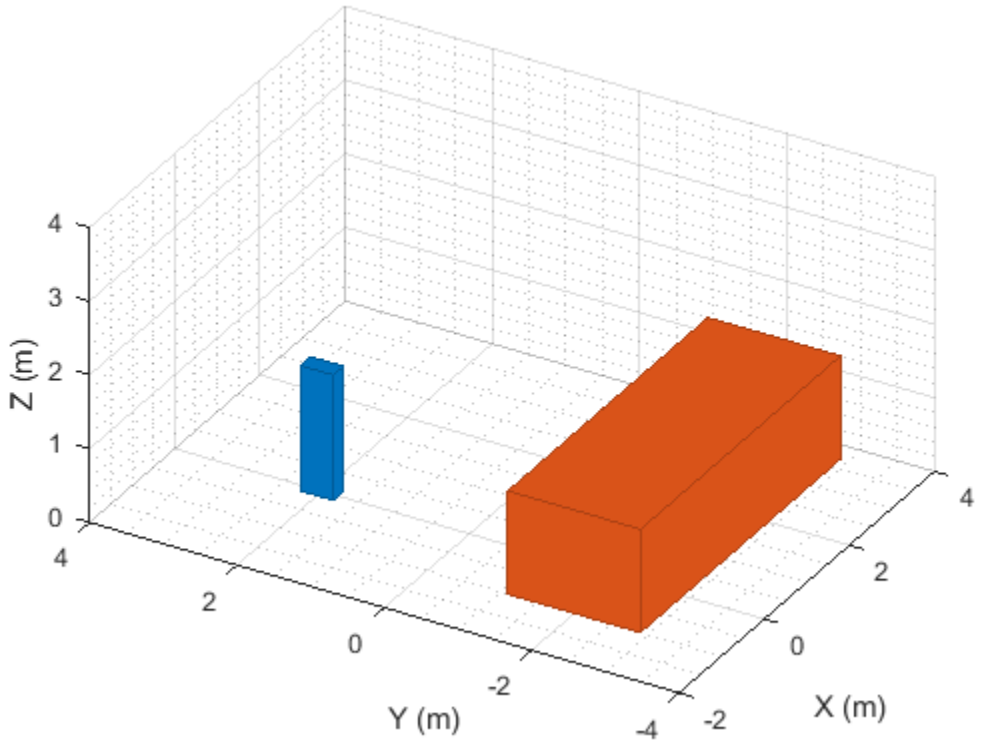
```
a =

  Actor with properties:

            EntryTime: 0
             ExitTime: Inf
              ActorID: 1
              ClassID: 1
                 Name: ""
            PlotColor: [0 0.4470 0.7410]
             Position: [0 2 0]
             Velocity: [0 0 0]
                  Yaw: 0
                Pitch: 0
                 Roll: 0
      AngularVelocity: [0 0 0]
               Length: 0.2400
                Width: 0.4500
               Height: 1.7000
                 Mesh: [1x1 extendedObjectMesh]
          RCSPattern: [2x2 double]
    RCSAzimuthAngles: [-180 180]
  RCSElevationAngles: [-90 90]


passingCar =

  Vehicle with properties:

        FrontOverhang: 0.9000
         RearOverhang: 1
            Wheelbase: 2.8000
            EntryTime: 0
             ExitTime: Inf
              ActorID: 2
              ClassID: 1
                 Name: ""
            PlotColor: [0.8500 0.3250 0.0980]
             Position: [0 -2 0]
             Velocity: [0 0 0]
                  Yaw: 0
                Pitch: 0
                 Roll: 0
      AngularVelocity: [0 0 0]
               Length: 4.7000
                Width: 1.8000
```

```
              Height: 1.4000
                Mesh: [1x1 extendedObjectMesh]
          RCSPattern: [2x2 double]
    RCSAzimuthAngles: [-180 180]
  RCSElevationAngles: [-90 90]
```



By default, the scenario plot shows an overhead view of the actors. To change this view, you can manipulate the scenario plot interactively by selecting the **Camera Toolbar** available in the **View** menu of the plot. Alternatively, you can programmatically manipulate the plot by using functions such as `xlim`, `ylim`, `zlim`, and `view`. These functions enable you to compare the relative heights of the actors.

```
zlim([0 4])
view(-60,30)
```



Define Trajectories

By using the `trajectory` function, you can specify actors and vehicles to follow a path along a set of waypoints at a set of given speeds. When you specify the waypoints, the `trajectory` function fits a piecewise clothoid curve to each segment between waypoints, preserving curvature between points. Clothoid curves have a curvature that varies linearly with distance traveled, which creates a very simple trajectory for drivers to navigate when traveling at constant velocity.

By default, actor trajectories have no curvature at the endpoints. To complete a loop, specify identical first and last waypoints.

To follow the entire trajectory at a constant speed, specify the speed as a scalar value.

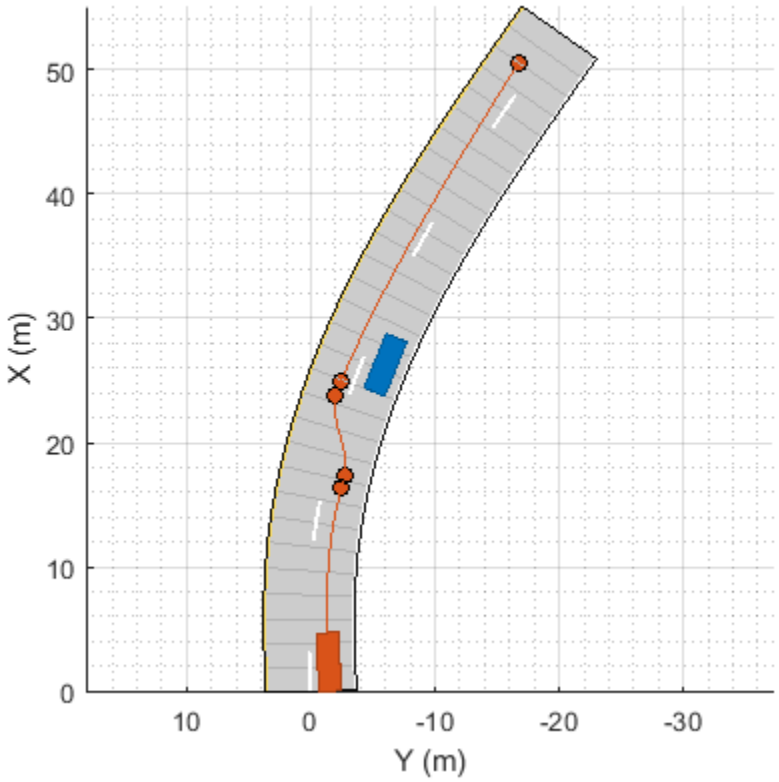
Vehicles pass through the curve between waypoints at their rotational centers. Therefore, to accommodate the length of the vehicle in front of and behind the rear axle during simulation, you can offset the beginning and ending waypoints. Offsetting these waypoints fits the vehicle completely within the road at its endpoints.

If the vehicle needs to turn quickly to avoid an obstacle, place two points close together in the intended direction of travel. This example shows a vehicle turning quickly at two places, but otherwise steering normally.

```
scenario = drivingScenario;
road(scenario, [0 0; 10 0; 53 -20],'lanes',lanespec(2));
plot(scenario,'Waypoints','on')
idleCar = vehicle(scenario,'ClassID',1,'Position',[25 -5.5 0],'Yaw',-22);
```

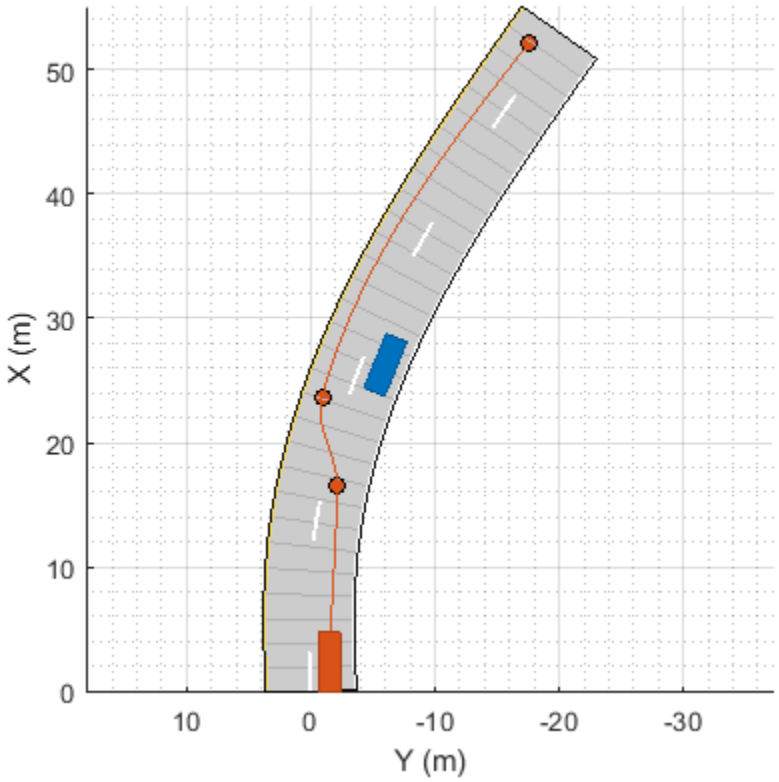
```
passingCar = vehicle(scenario,'ClassID',1)

waypoints = [1 -1.5; 16.36 -2.5; 17.35 -2.765; 23.83 -2.01; 24.9 -2.4; 50.5 -16.7];
speed = 15;
trajectory(passingCar,waypoints,speed)
```

```
passingCar =

  Vehicle with properties:

         FrontOverhang: 0.9000
          RearOverhang: 1
             Wheelbase: 2.8000
             EntryTime: 0
              ExitTime: Inf
               ActorID: 2
               ClassID: 1
                  Name: ""
             PlotColor: [0.8500 0.3250 0.0980]
              Position: [0 0 0]
              Velocity: [0 0 0]
                   Yaw: 0
                 Pitch: 0
                  Roll: 0
       AngularVelocity: [0 0 0]
                Length: 4.7000
                 Width: 1.8000
                Height: 1.4000
                  Mesh: [1x1 extendedObjectMesh]
            RCSPattern: [2x2 double]
      RCSAzimuthAngles: [-180 180]
    RCSElevationAngles: [-90 90]
```



Alternatively, you can use fewer waypoints at such turns by explicitly setting the yaw orientation angle of the vehicle at each waypoint. Yaw is positive in the counterclockwise direction, and its units are in degrees. In this variation of the previous example, the trajectory is constrained such that the vehicle is at a –15 degree angle after moving into the left lane. By setting a waypoint to `NaN`, the `trajectory` function defaults to fitting a clothoid curve to the segment leading up that waypoint. In this case, that segment is the final one in the trajectory.

```
waypoints = [1 -1.5; 16.6 -2.1; 23.7 -0.9; 52.2 -17.6];
yaw =  [0; 0; -15; NaN];
trajectory(passingCar,waypoints,speed,'Yaw',yaw)
```



Turning and Braking at Intersections

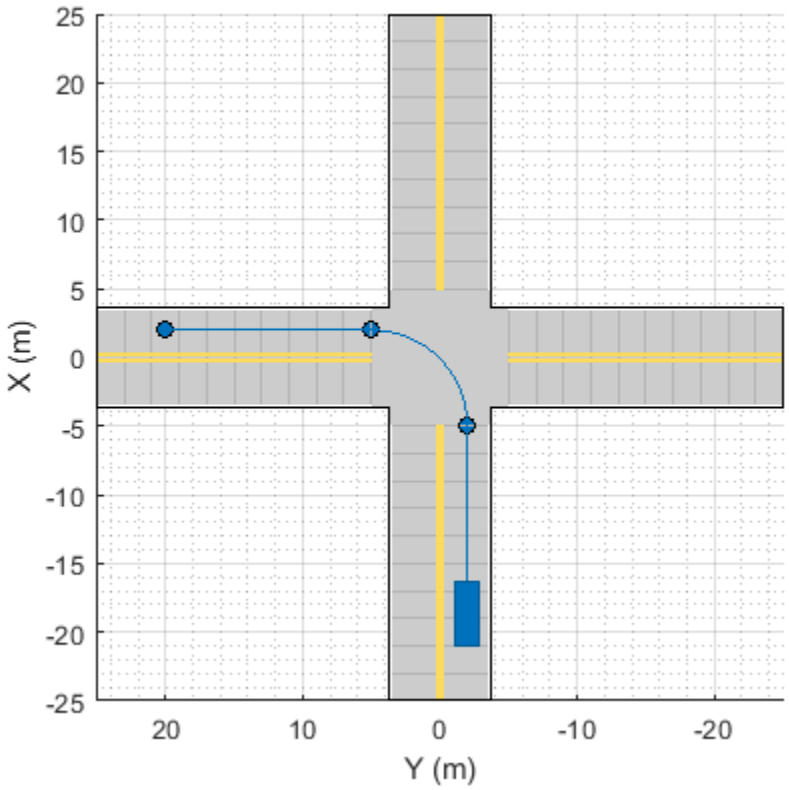
For sharp turns, either define waypoints close together at the start and end of the turn or explicitly set the yaw of the vehicle at each waypoint. This setting faithfully renders the sudden change in steering.

In this example, a vehicle makes a sharp left turn at an intersection using explicitly set yaw values. At the first waypoint and the waypoint before the turn, the vehicle has a yaw of 0 degrees. At the waypoint after the turn and the final waypoint, the vehicle has a yaw of 90 degrees, which is the orientation of the vehicle after completing the turn. By constraining the trajectory such that the vehicle achieves these yaw orientations, the vehicle turn is much sharper than if you had used the default yaw orientations.

To specify for vehicles to follow curves of piecewise constant acceleration, specify the vehicle speed at each waypoint. In this example, the vehicle decelerates from a speed of 20 m/s and comes to a brief complete stop at location (–5, –2). After completing the turn, the vehicle gradually accelerates back to its original speed.

```
scenario = drivingScenario;
road(scenario,[0 -25; 0 25],'lanes',lanespec([1 1]));
road(scenario,[-25 0; 25 0],'lanes',lanespec([1 1]));

turningCar = vehicle(scenario,'ClassID',1);

waypoints = [-20 -2; -5 -2; 2 5; 2 20];
speed = [15 5 5 15];
```

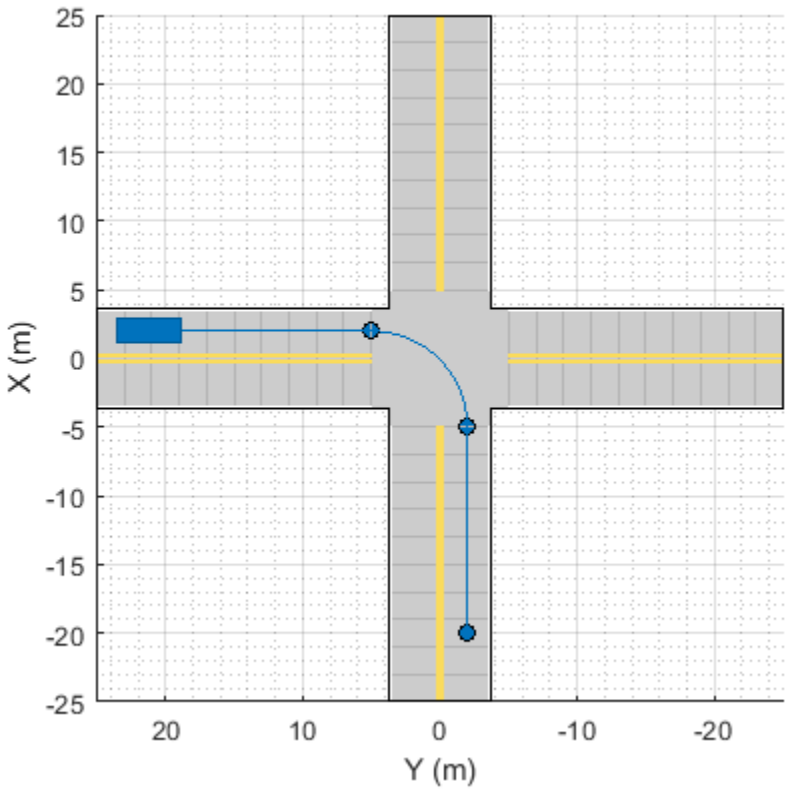
```
yaw = [0 0 90 90];
trajectory(turningCar,waypoints,speed,'Yaw',yaw)

plot(scenario,'Waypoints','on')
```



Move Vehicles

After you define all the roads, actors, and actor trajectories, you can increment the position of each actor by using the `advance` function on the driving scenario in a loop.

```
while advance(scenario)
    pause(0.01)
end
```

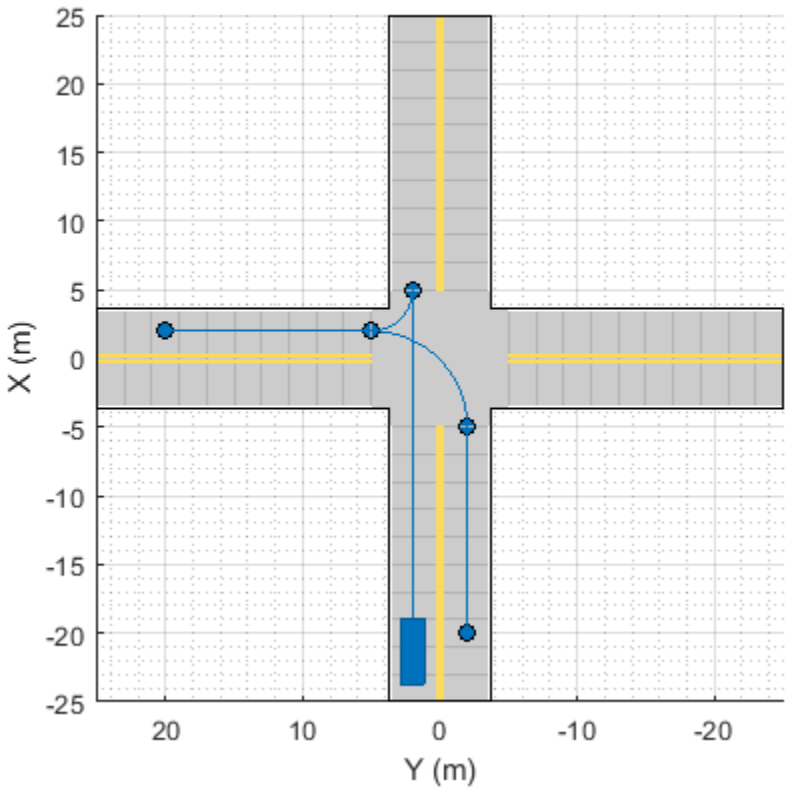


Move Vehicles in Reverse

To specify reverse driving motions, specify a trajectory with negative speeds. When switching between forward and reverse motions, you must specify a waypoint between these motions that has a speed of `0`. At this waypoint, the vehicle decelerates until it comes to a complete stop, and then changes driving directions.

This example expands on the previous example. This time, after completing the left turn, the vehicle backs up and reverses at the intersection. Then, the vehicle switches direction again and drives forward until it stops in the opposite lane from where it started.

```
scenario = drivingScenario;
road(scenario,[0 -25; 0 25],'lanes',lanespec([1 1]));
road(scenario,[-25 0; 25 0],'lanes',lanespec([1 1]));

turningCar = vehicle(scenario,'ClassID',1);

waypoints = [-20 -2; -5 -2; 2 5; 2 20; 2 5; 5 2; -20 2];
speed = [15 5 5 0 -15 0 15];
yaw =  [0 0 90 90 -90 0 -180];
trajectory(turningCar,waypoints,speed,'Yaw',yaw)

plot(scenario,'Waypoints','on')

while advance(scenario)
    pause(0.01)
end
```



Next Steps

This example showed how to create actor and vehicle trajectories for a driving scenario using a `drivingScenario` object. To simulate, visualize, or modify this driving scenario in an interactive environment, try importing the `drivingScenario` object into the **Driving Scenario Designer** app by using this command:

```
drivingScenarioDesigner(scenario)
```

See Also

Apps
Driving Scenario Designer

Objects
`birdsEyePlot` | `drivingScenario` | `lanespec`

Functions
`actor` | `road` | `trajectory` | `vehicle`

More About

- “Create Driving Scenario Programmatically” on page 7-423
- “Create Driving Scenario Variations Programmatically” on page 5-107

- “Scenario Generation from Recorded Vehicle Data” on page 7-350

Define Road Layouts Programmatically

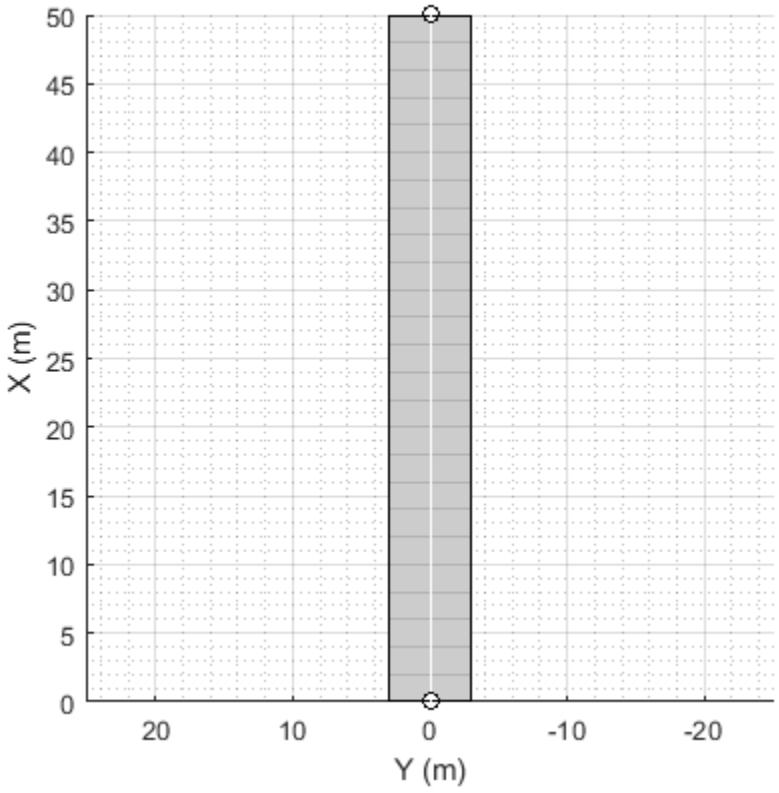
This example shows how to programmatically create a variety of road junctions with Automated Driving Toolbox™ functions. You can combine these junctions with other junctions to create complicated road networks. You can view the code for each plot and use it in your own project.

Alternatively, you can create road junctions interactively by using the Driving Scenario Designer app.

Straight Roads

Roads of a fixed width can be defined by a series of points that define the locations of the center of the road. A straight road is very simple to describe by specifying its starting and stopping location. Here is an example of a road which starts at (0,0) and ends at (50,0) and has a width of 6 (meters).

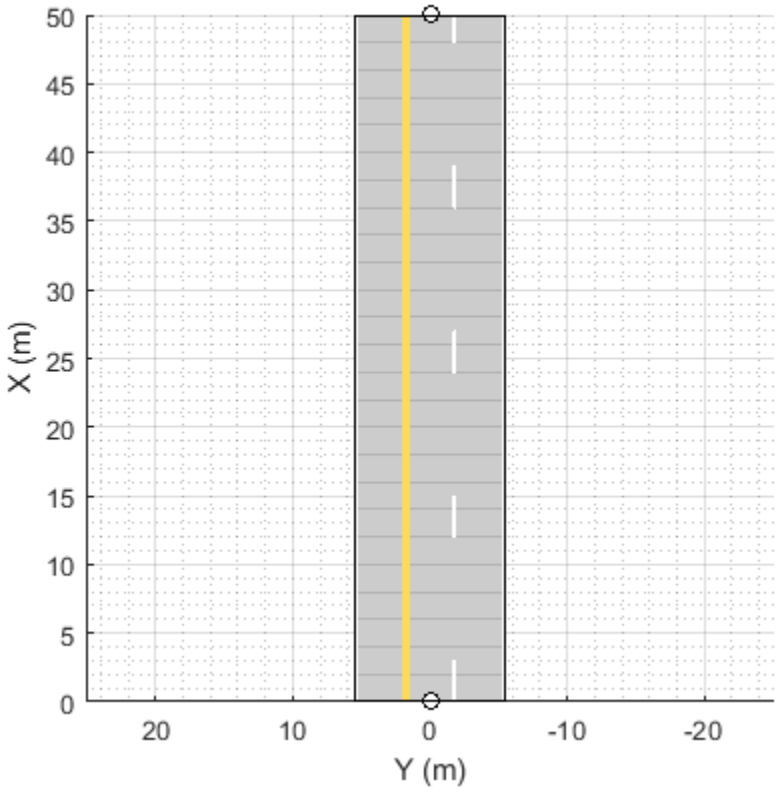
```
scenario = drivingScenario;

roadCenters = [0 0; 50 0];
roadWidth = 6;

road(scenario, roadCenters, roadWidth);
plot(scenario,'RoadCenters','on','Centerline','on');
```



Laned Roads

As an alternative to specifying road widths, you can specify lanes by providing a lane specification. Here is an example of specifying a road with one lane on the left and two on the right.

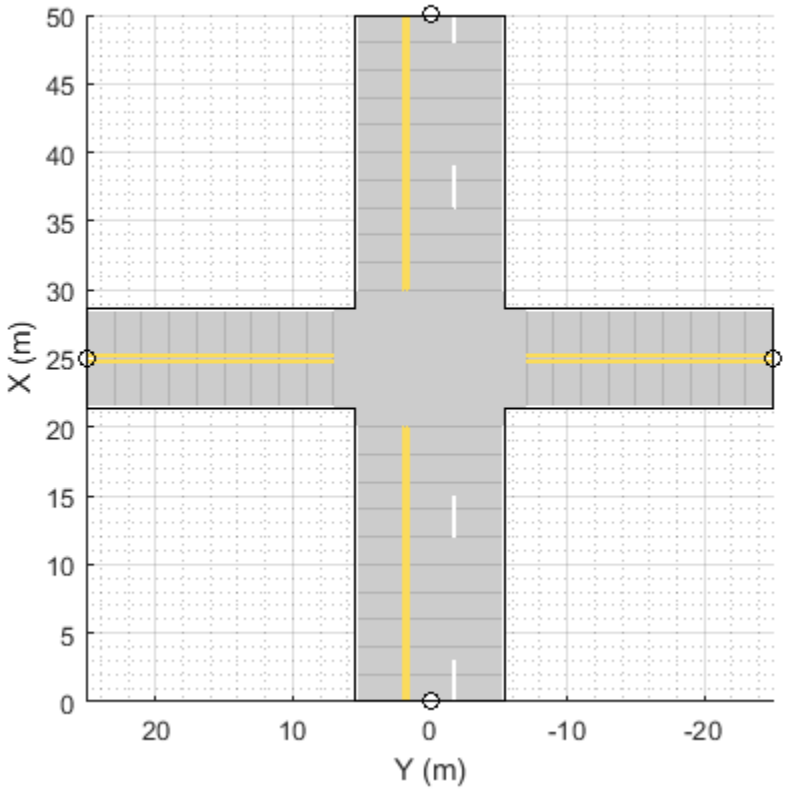
```
scenario = drivingScenario;

roadCenters = [0 0; 50 0];

road(scenario, roadCenters, 'lanes', lanespec([1 2]));
plot(scenario,'RoadCenters','on');
```



Intersections

Intersections are automatically generated wherever two roads meet. In this example, we add another 50 m section of road.

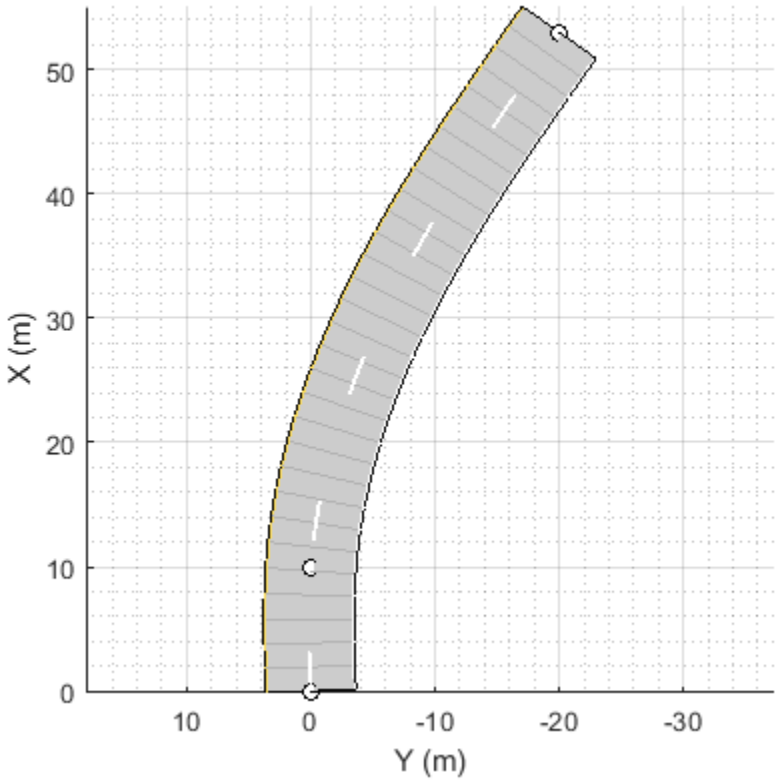
```
roadCenters = [25 -25; 25  25];
road(scenario, roadCenters, 'lanes', lanespec([1 1]));
```



Curved Roads

Curved roads can be described by using three or more points. The more points you use, the more complex the curve you can create. In this example, we have a curve passing through three points:

```
scenario = drivingScenario;

roadCenters = [0 0; 10 0; 53 -20];
roadWidth = 6;

road(scenario, roadCenters, roadWidth,'lanes',lanespec(2));
plot(scenario,'RoadCenters','on');
```



Roundabouts

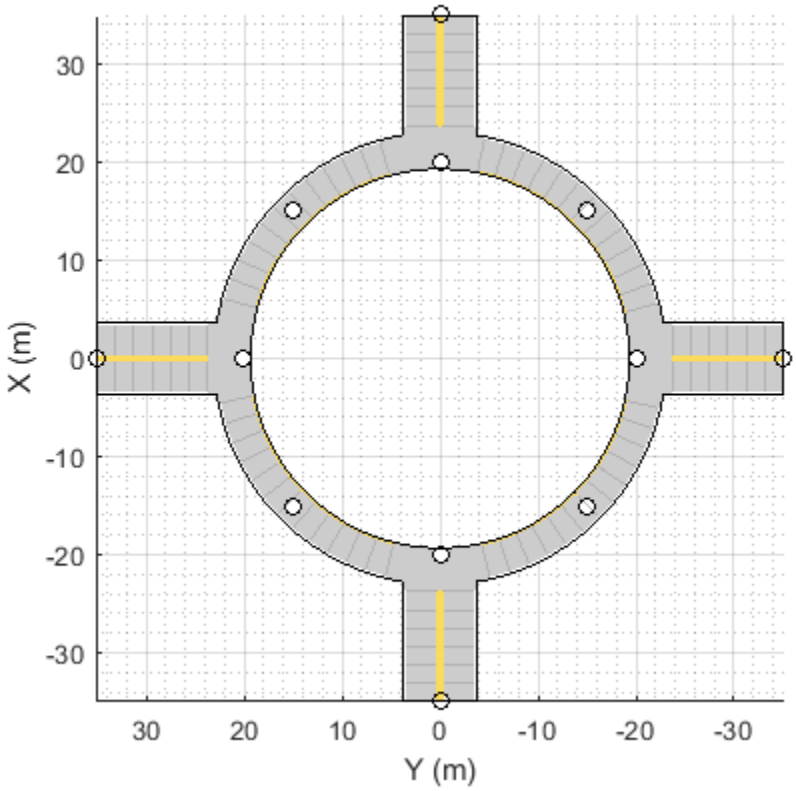
When you specify the road centers, a piecewise clothoid curve is fit in between each segment, where curvature is preserved in between points. Clothoid curves are used extensively when designing roads, because they have a curvature that varies linearly with distance traveled along the road, which is very simple for drivers to navigate.

By default, roads built by the scenario will have no curvature at the endpoints. To make a road loop, repeat the first and last point.

In this example, we show a 4m wide circular road segment circumscribed about a 30 m square area. Adding roads that feed into the roundabout is a matter of specifying other straight or curved road segments

```
scenario = drivingScenario;

roadCenters = [-15 -15
                15 -15
                15  15
               -15  15
               -15 -15];

road(scenario, roadCenters, 'lanes', lanespec(1));

% Define roundabout exits with two lanes each
road(scenario, [-35   0; -20   0],'lanes', lanespec([1 1]));
road(scenario, [ 20   0;  35   0],'lanes', lanespec([1 1]));
```

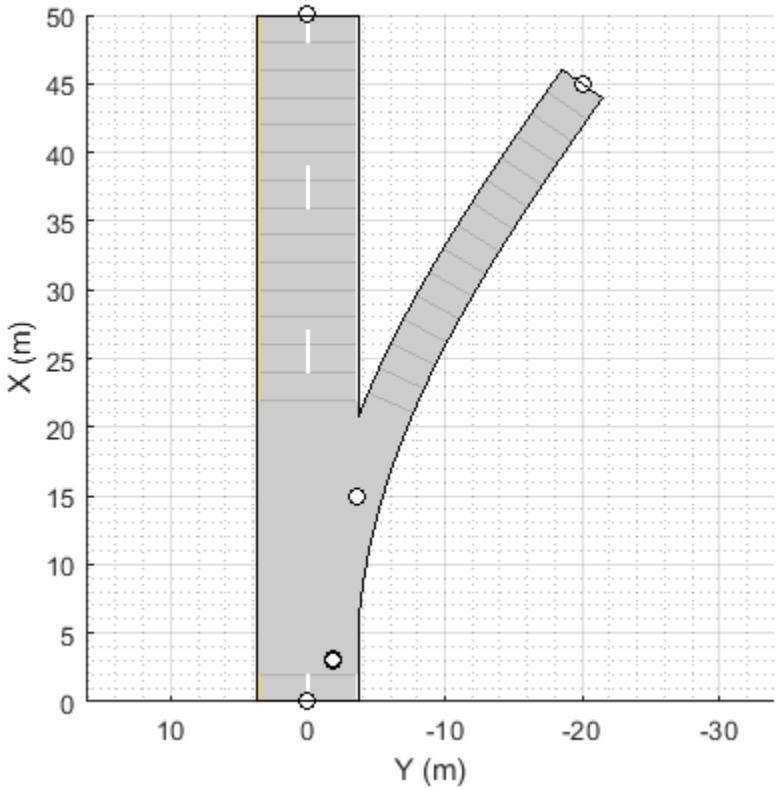
```
road(scenario, [  0  35;   0  20],'lanes', lanespec([1 1]));
road(scenario, [  0 -20;  0  -35],'lanes', lanespec([1 1]));

plot(scenario,'RoadCenters','on');
```



Exit Lane

This example simulates a simple exit lane. We start with a simple straight road and then overlay a few points of another road so that it overlaps the original straight road:

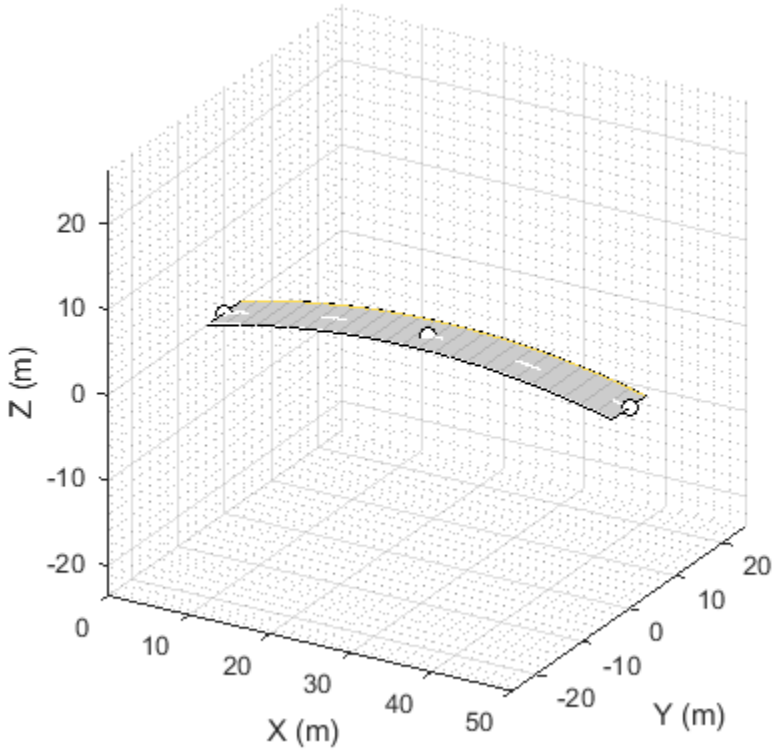
```
scenario = drivingScenario;

ls = lanespec(2);
laneWidth = ls.Width(1);

% add straight road segment
road(scenario, [0 0 0; 50 0 0], 'lanes', lanespec(2));

% define waypoints of lane exit
roadCenters =  [3.0 -laneWidth/2
                3.1 -laneWidth/2
               15.0 -laneWidth
               45.0 -20];

% add the exit lane
road(scenario, roadCenters, laneWidth);

plot(scenario,'RoadCenters','on')
```



Adding Elevation

Roads can optionally have elevation information. This can be accomplished by including a third column in the waypoints.

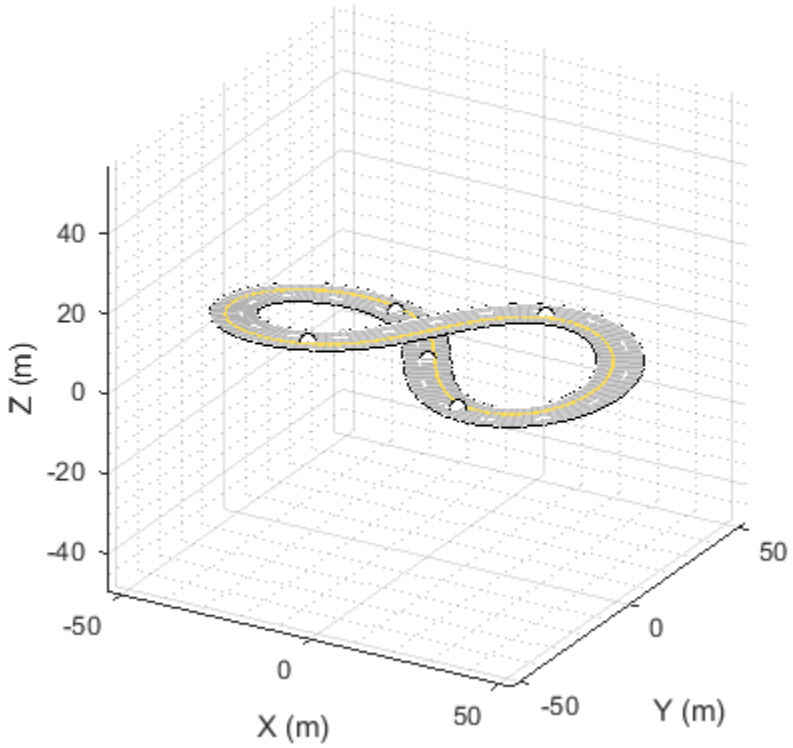
```
scenario = drivingScenario;

roadCenters = [ 0 0 0
                25 0 3
                50 0 0];

road(scenario, roadCenters, 'lanes', lanespec(2));

plot(scenario,'RoadCenters','on');
view(30,24);
```



Overpasses

Roads can cross each other without intersecting if they have differing elevation. The road surface of an overpass is typically 6 to 8 meters above the road.

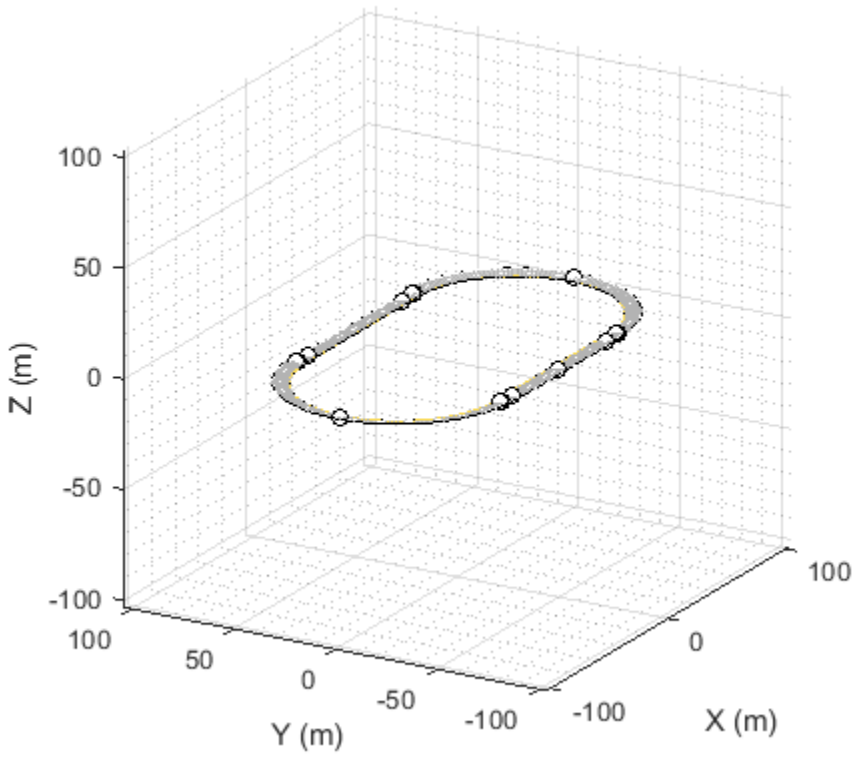
```
scenario = drivingScenario;
roadCenters = [  0   0  0
                20 -20  0
                20  20  8
               -20 -20  8
               -20  20  0
                 0   0  0];

             road(scenario, roadCenters, 'lanes',lanespec([1 2]));
plot(scenario,'RoadCenters','on');
view(30,24)
```



Road Banking

Roads can be banked; where bank angles can be defined for each waypoint. The following is an oval racetrack with 9 degree banked curves.

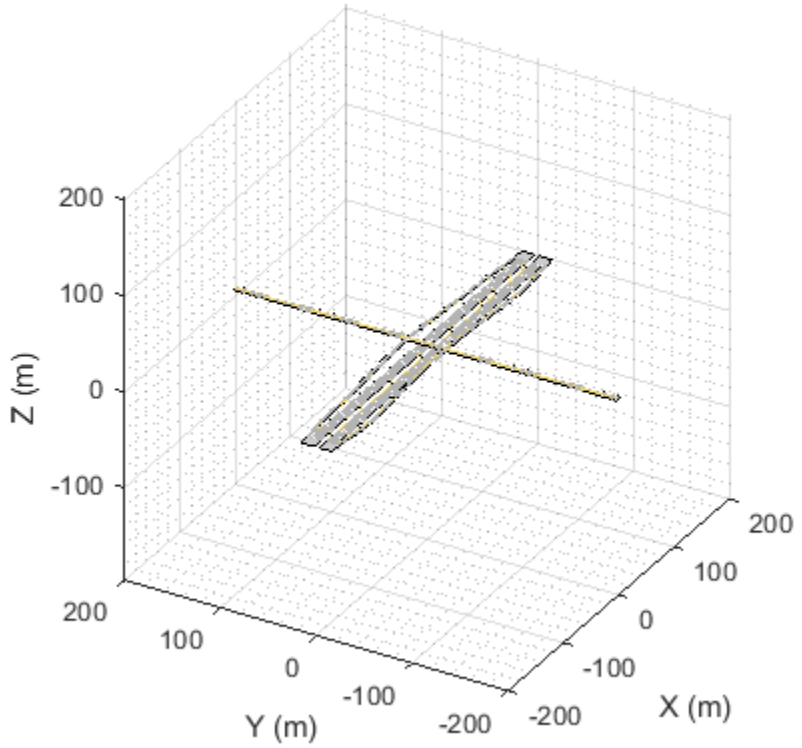
```
scenario = drivingScenario;

% transpose waypoints so they visually align with bank angles below
roadCenters = ...
    [  0  40  49  50 100  50  49 40 -40 -49 -50 -100  -50  -49  -40    0
     -50 -50 -50 -50   0  50  50 50  50  50  50    0  -50  -50  -50  -50
       0   0 .45 .45 .45 .45 .45  0   0 .45 .45  .45  .45  .45    0    0]';
bankAngles = ...
    [  0   0   9   9   9   9   9  0   0   9   9    9    9    9    0    0];

road(scenario, roadCenters, bankAngles, 'lanes', lanespec(2));
plot(scenario,'RoadCenters','on');
view(-60,20)
```



Diamond Interchange

Highways and expressways typically are comprised of two parallel roads, each going in the opposing direction. An economical interchange between a highway and a local road is a diamond interchange, which typically consists of a local road overpass and four ramps.

```
scenario = drivingScenario;

% Highways
road(scenario, [-200 -8 0; 200 -8 0], 'lanes',lanespec(3)); % north
road(scenario, [ 200  8 0;-200  8 0], 'lanes',lanespec(3)); % south

% Local Road
road(scenario, [-0 -200 8; 0  200 8], 'lanes',lanespec([1 1]));

% Access ramps
rampNE = [ 3 -20 8;  10 -20 7.8; 126 -20 .2; 130 -20 0; 200 -13.5 0];
road(scenario, [ 1  1  1] .* rampNE, 'lanes',lanespec(1));
road(scenario, [ 1 -1  1] .* flipud(rampNE), 'lanes',lanespec(1));
road(scenario, [-1 -1  1] .* rampNE, 'lanes',lanespec(1));
road(scenario, [-1  1  1] .* flipud(rampNE), 'lanes',lanespec(1));

plot(scenario);
view(-60,30)
```



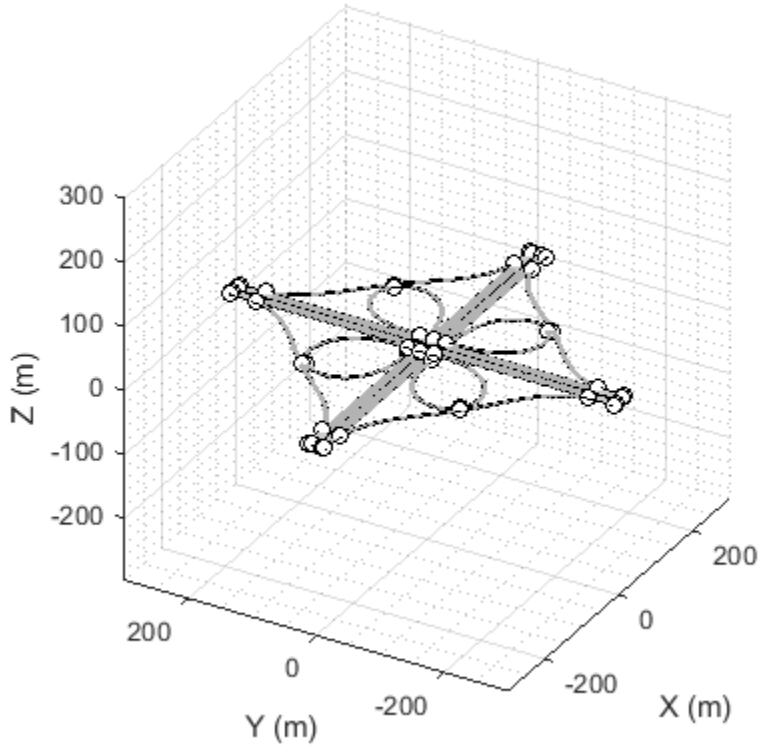
Cloverleaf Interchange

A popular interchange between two highways is the cloverleaf interchange. The cloverleaf interchange consists of four inner and four outer ramps.

A limitation of the driving scenario is that road information is removed in the vicinity of a road junction

```
scenario = drivingScenario;

% Highways
road(scenario, [-300 -8 0; 300 -8 0], 15); % north
road(scenario, [-300  8 0; 300  8 0], 15); % south
road(scenario, [-8 -300 8; -8 300 8], 15); % east
road(scenario, [ 8 -300 8;  8 300 8], 15); % west

% Inner ramps
rampNE = [0 -18 0; 20 -18  0; 120 -120  4; 18  -20  8; 18 0  8];
rampNW = [ 1 -1 1] .* rampNE(end:-1:1,:);
rampSW = [-1 -1 1] .* rampNE;
rampSE = [ 1 -1 1] .* rampSW(end:-1:1,:);
innerRamps = [rampNE(1:end-1,:)
              rampNW(1:end-1,:)
              rampSW(1:end-1,:)
              rampSE];
road(scenario, innerRamps, 5.4);
```

```
% Outer ramps
roadCenters = [13.5 -300 8; 15 -260 8; 125 -125 4; 260 -15 0; 300 -13.5 0];
road(scenario, [ 1  1  1] .* roadCenters, 5.4);
road(scenario, [ 1 -1  1] .* roadCenters, 5.4);
road(scenario, [-1 -1  1] .* roadCenters, 5.4);
road(scenario, [-1  1  1] .* roadCenters, 5.4);

plot(scenario,'RoadCenters','on');
view(-60,30);
```



Next Steps

This example showed how to create a variety of road junctions using a `drivingScenario` object. To add actors and trajectories to these roads using Automated Driving Toolbox functions, see “Create Actor and Vehicle Trajectories Programmatically” on page 7-442. Alternatively, you can add actors and trajectories interactively by loading the `drivingScenario` object into the Driving Scenario Designer app:

```
drivingScenarioDesigner(scenario)
```

See Also

Apps
Driving Scenario Designer

Objects
`drivingScenario` | `lanespec`

Functions
`laneBoundaries` | `road` | `roadBoundaries`

More About

- “Create Driving Scenario Programmatically” on page 7-423
- “Create Driving Scenario Variations Programmatically” on page 5-107
- “Create Actor and Vehicle Trajectories Programmatically” on page 7-442

Automated Parking Valet

This example shows how to construct an automated parking valet system. In this example, you learn about tools and techniques in support of path planning, trajectory generation, and vehicle control. While this example focuses on a MATLAB®-oriented workflow, these tools are also available in Simulink®. For a Simulink® version of this example, see “Automated Parking Valet in Simulink” on page 7-493.

Overview

Automatically parking a car that is left in front of a parking lot is a challenging problem. The vehicle's automated systems are expected to take over control and steer the vehicle to an available parking spot. Such a function makes use of multiple on-board sensors. For example:

- Front and side cameras for detecting lane markings, road signs (stop signs, exit markings, etc.), other vehicles, and pedestrians
- Lidar and ultrasound sensors for detecting obstacles and calculating accurate distance measurements
- Ultrasound sensors for obstacle detection
- IMU and wheel encoders for dead reckoning

On-board sensors are used to perceive the environment around the vehicle. The perceived environment includes an understanding of road markings to interpret road rules and infer drivable regions, recognition of obstacles, and detection of available parking spots.

As the vehicle sensors perceive the world, the vehicle must plan a path through the environment towards a free parking spot and execute a sequence of control actions needed to drive to it. While doing so, it must respond to dynamic changes in the environment, such as pedestrians crossing its path, and readjust its plan.

This example implements a subset of features required to implement such a system. It focuses on planning a feasible path through the environment, and executing the actions needed to traverse the path. Map creation and dynamic obstacle avoidance are excluded from this example.

Environment Model

The environment model represents a map of the environment. For a parking valet system, this map includes available and occupied parking spots, road markings, and obstacles such as pedestrians or other vehicles. Occupancy maps are a common representation for this form of environment model. Such a map is typically built using Simultaneous Localization and Mapping (SLAM) by integrating observations from lidar and camera sensors. This example concentrates on a simpler scenario, where a map is already provided, for example, by a vehicle-to-infrastructure (V2X) system or a camera overlooking the entire parking space. It uses a static map of a parking lot and assumes that the self-localization of the vehicle is accurate.

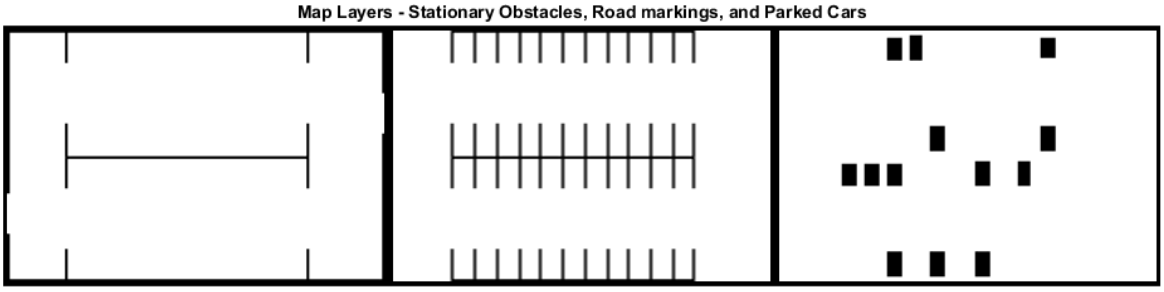
The parking lot example used in this example is composed of three occupancy grid layers.

- Stationary obstacles: This layer contains stationary obstacles like walls, barriers, and bounds of the parking lot.
- Road markings: This layer contains occupancy information pertaining to road markings, including road markings for parking spaces.
- Parked cars: This layer contains information about which parking spots are already occupied.

Each map layer contains different kinds of obstacles that represent different levels of danger for a car navigating through it. With this structure, each layer can be handled, updated, and maintained independently.

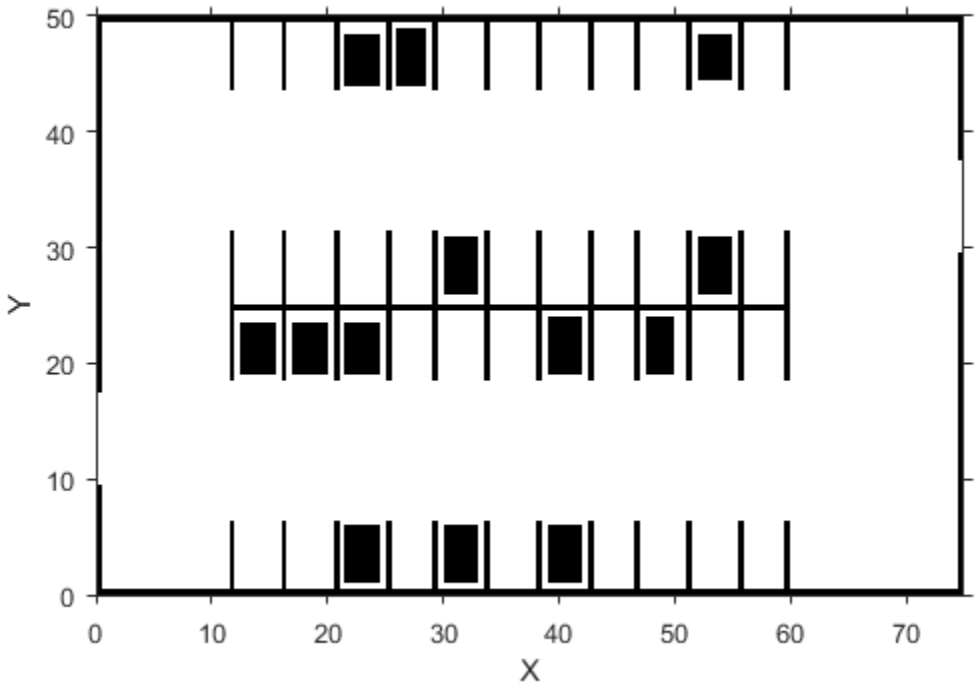
Load and display the three map layers. In each layer, dark cells represent occupied cells, and light cells represent free cells.

```
mapLayers = loadParkingLotMapLayers;
plotMapLayers(mapLayers)
```



For simplicity, combine the three layers into a single costmap.

```
costmap = combineMapLayers(mapLayers);

figure
plot(costmap, 'Inflation', 'off')
legend off
```



The combined `costmap` is a `vehicleCostmap` object, which represents the vehicle environment as a 2-D occupancy grid. Each grid in the cell has values between 0 and 1, representing the cost of navigating through the cell. Obstacles have a higher cost, while free space has a lower cost. A cell is considered an obstacle if its cost is higher than the `OccupiedThreshold` property, and free if its cost is lower than the `FreeThreshold` property.

The `costmap` covers the entire 75m-by-50m parking lot area, divided into 0.5m-by-0.5m square cells.

```
costmap.MapExtent % [x, width, y, height] in meters
```

```
costmap.CellSize  % cell size in meters
```

```
ans =

     0    75     0    50
```

```
ans =

    0.5000
```

Create a `vehicleDimensions` object for storing the dimensions of the vehicle that will park automatically. Also define the maximum steering angle of the vehicle. This value determines the limits on the turning radius during motion planning and control.

```
vehicleDims      = vehicleDimensions;
maxSteeringAngle = 35; % in degrees
```

Update the `VehicleDimensions` property of the costmap collision checker with the dimensions of the vehicle to park. This setting adjusts the extent of the inflation in the map around obstacles to correspond to the size of the vehicle being parked, ensuring that collision-free paths can be found through the parking lot.

```
costmap.CollisionChecker.VehicleDimensions = vehicleDims;
```

Define the starting pose of the vehicle. The pose is obtained through localization, which is left out of this example for simplicity. The vehicle pose is specified as $[x, y, \theta]$, in world coordinates. (x, y) represents the position of the center of the vehicle's rear axle in world coordinate system. θ represents the orientation of the vehicle with respect to world X axis. For more details, see “Coordinate Systems in Automated Driving Toolbox” on page 1-2.

```
currentPose = [4 12 0]; % [x, y, theta]
```

Behavioral Layer

Planning involves organizing all pertinent information into hierarchical layers. Each successive layer is responsible for a more fine-grained task. The behavioral layer [1] sits at the top of this stack. It is responsible for activating and managing the different parts of the mission by supplying a sequence of navigation tasks. The behavioral layer assembles information from all relevant parts of the system, including:

- Localization: The behavioral layer inspects the localization module for an estimate of the current location of the vehicle.
- Environment model: Perception and sensor fusion systems report a map of the environment around the vehicle.
- Determining a parking spot: The behavioral layer analyzes the map to determine the closest available parking spot.
- Finding a global route: A routing module calculates a global route through the road network obtained either from a mapping service or from a V2X infrastructure. Decomposing the global route as a series of road links allows the trajectory for each link to be planned differently. For example, the final parking maneuver requires a different speed profile than the approach to the parking spot. In a more general setting, this becomes crucial for navigating through streets that involve different speed limits, numbers of lanes, and road signs.

Rather than rely on vehicle sensors to build a map of the environment, this example uses a map that comes from a smart parking lot via V2X communication. For simplicity, assume that the map is in the form of an occupancy grid, with road links and locations of available parking spots provided by V2X.

The `HelperBehavioralPlanner` class mimics an interface of a behavioral planning layer. The `HelperBehavioralPlanner` is created using the map and the global route plan. This example uses a static global route plan stored in a MATLAB table, but typically a routing algorithm provided by the local parking infrastructure or a mapping service determines this plan. The global route plan is described as a sequence of lane segments to traverse to reach a parking spot.

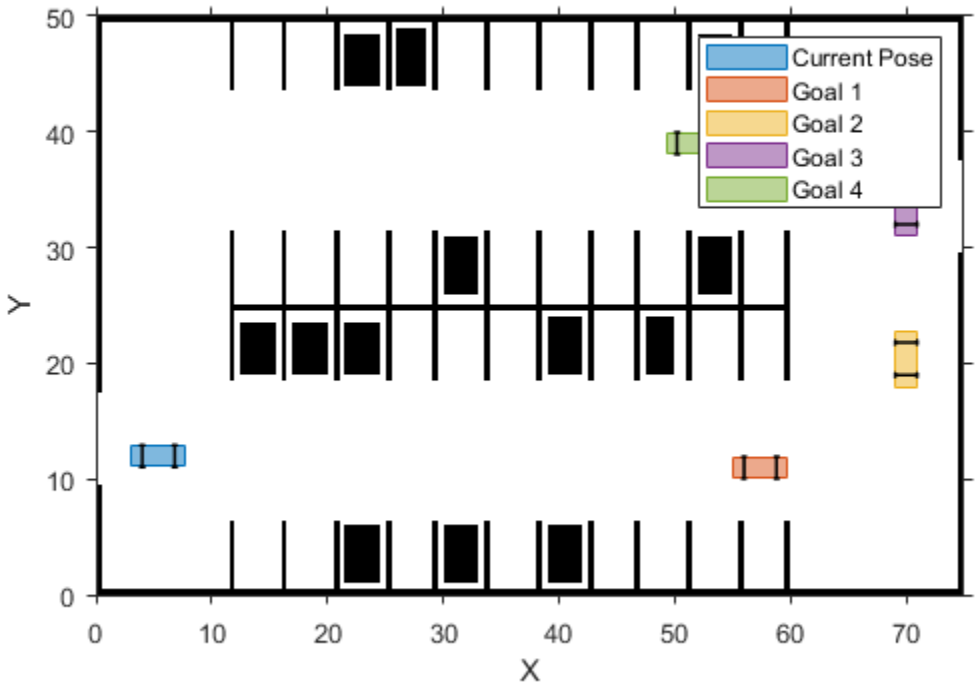
Load the MAT-file containing a route plan that is stored in a table. The table has three variables: `StartPose`, `EndPose`, and `Attributes`. `StartPose` and `EndPose` specify the start and end poses of the segment, expressed as $[x, y, \theta]$. `Attributes` specifies properties of the segment such as the speed limit.

```
data = load('routePlan.mat');
routePlan = data.routePlan %#ok<NOPTS>
```

```
routePlan =

  4×3 table

      StartPose           EndPose          Attributes
    ______________    _______________    ____________

     4    12     0    56     11      0    [1×1 struct]
    56    11     0    70     19     90    [1×1 struct]
    70    19    90    70     32     90    [1×1 struct]
    70    32    90    53     39    180    [1×1 struct]
```

Plot a vehicle at the current pose, and along each goal in the route plan.

```
% Plot vehicle at current pose
hold on
helperPlotVehicle(currentPose, vehicleDims, 'DisplayName', 'Current Pose')
legend

for n = 1 : height(routePlan)
    % Extract the goal waypoint
    vehiclePose = routePlan{n, 'EndPose'};

    % Plot the pose
    legendEntry = sprintf('Goal %i', n);
    helperPlotVehicle(vehiclePose, vehicleDims, 'DisplayName', legendEntry);
end
hold off
```



Create the behavioral planner helper object. The `requestManeuver` method requests a stream of navigation tasks from the behavioral planner until the destination is reached.

```
behavioralPlanner = HelperBehavioralPlanner(routePlan, maxSteeringAngle);
```

The vehicle navigates each path segment using these steps:

1. **Motion Planning:** Plan a feasible path through the environment map using the optimal rapidly exploring random tree (RRT*) algorithm (`pathPlannerRRT`).
2. **Path Smoothing:** Smooth the reference path by fitting splines to it using `smoothPathSpline`.
3. **Trajectory Generation:** Convert the smoothed path into a trajectory by generating a speed profile using `helperGenerateVelocityProfile`.
4. **Vehicle Control:** Given the smoothed reference path, `HelperPathAnalyzer` calculates the reference pose and velocity based on the current pose and velocity of the vehicle. Provided with the reference values, `lateralControllerStanley` computes the steering angle to control the heading of the vehicle. `HelperLongitudinalController` computes the acceleration and deceleration commands to maintain the desired vehicle velocity.
5. **Goal Checking:** Check if the vehicle has reached the final pose of the segment using `helperGoalChecker`.

The rest of this example describes these steps in detail, before assembling them into a complete solution.

Motion Planning

Given a global route, motion planning can be used to plan a path through the environment to reach each intermediate waypoint, until the vehicle reaches the final destination. The planned path for each link must be feasible and collision-free. A feasible path is one that can be realized by the vehicle given the motion and dynamic constraints imposed on it. A parking valet system involves low velocities and low accelerations. This allows us to safely ignore dynamic constraints arising from inertial effects.

Create a `pathPlannerRRT` object to configure a path planner using an optimal rapidly exploring random tree (RRT*) approach. The RRT family of planning algorithms find a path by constructing a tree of connected, collision-free vehicle poses. Poses are connected using Dubins or Reeds-Shepp steering, ensuring that the generated path is kinematically feasible.

```
motionPlanner = pathPlannerRRT(costmap, 'MinIterations', 1000, ...
    'ConnectionDistance', 10, 'MinTurningRadius', 20);
```

Plan a path from the current pose to the first goal by using the `plan` function. The returned `driving.Path` object, `refPath`, is a feasible and collision-free reference path.

```
goalPose = routePlan{1, 'EndPose'};
refPath = plan(motionPlanner, currentPose, goalPose);
```

The reference path consists of a sequence of path segments. Each path segment describes the set of Dubins or Reeds-Shepp maneuvers used to connect to the next segment. Inspect the path segments.

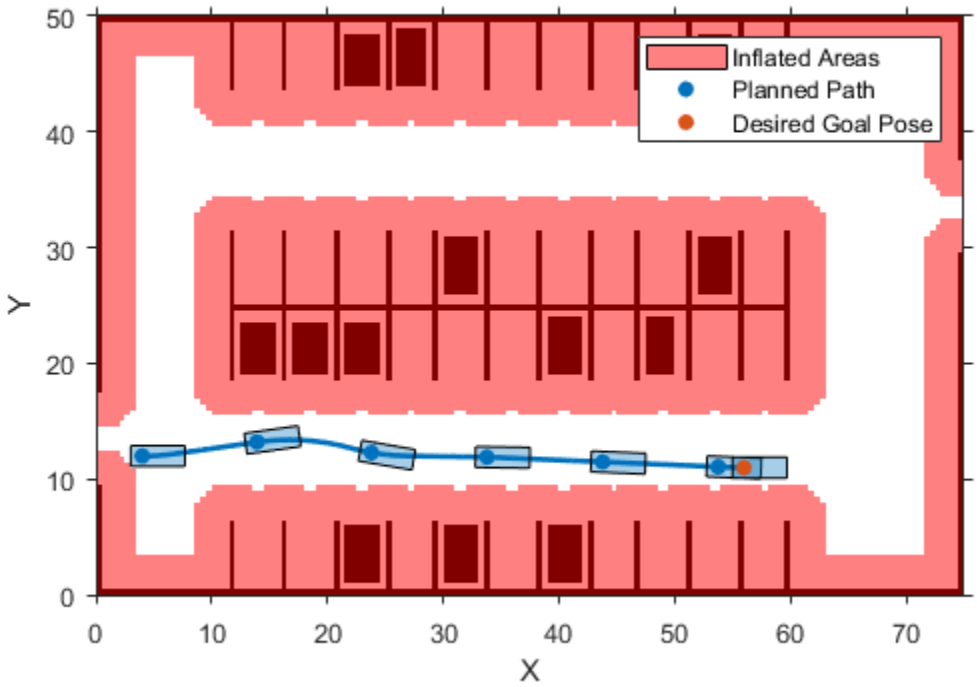
```
refPath.PathSegments
```

```
ans =

  1×6 DubinsPathSegment array with properties:

    StartPose
    GoalPose
    MinTurningRadius
    MotionLengths
    MotionTypes
    Length
```

The reference path contains transition poses along the way, representing points along the path corresponding to a transition from one maneuver to the next. They can also represent changes in direction, for example, from forward to reverse motion along a Reeds-Shepp path.

Retrieve transition poses and directions from the planned path.

```
[transitionPoses, directions] = interpolate(refPath);

% Visualize the planned path
plot(motionPlanner)
```



In addition to the planned reference path, notice the red areas on the plot. These areas represent areas of the costmap where the origin of the vehicle (center of the rear axle) must not cross in order to avoid hitting any obstacles. `pathPlannerRRT` finds paths that avoid obstacles by checking to ensure that vehicle poses generated do not lie on these areas.

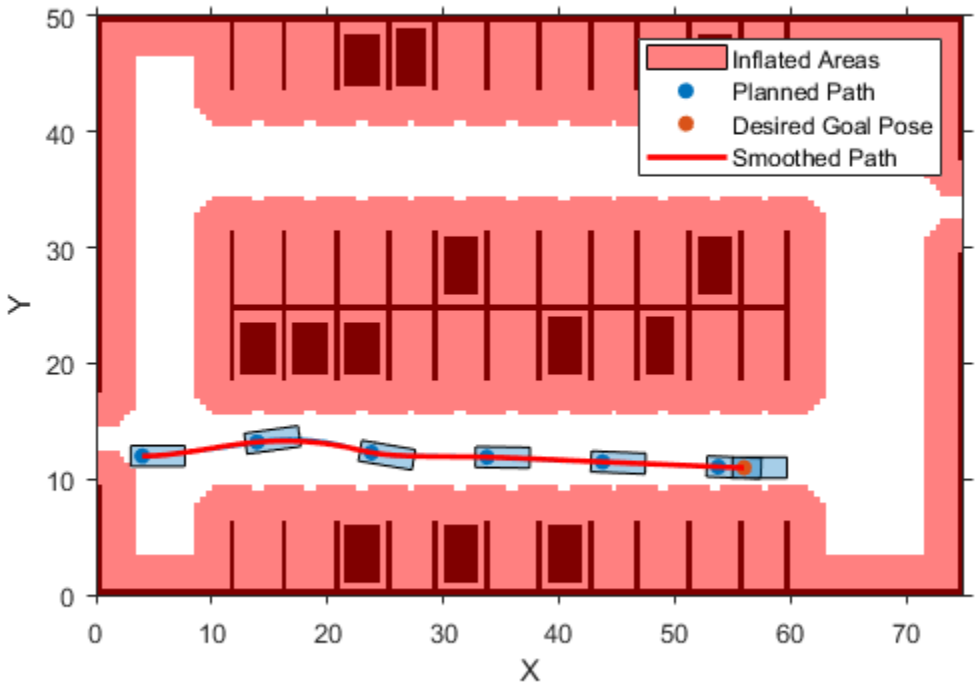
Path Smoothing and Trajectory Generation

The reference path generated by the path planner is composed either of Dubins or Reeds-Shepp segments. The curvature at the junctions of two such segments is not continuous and can result in abrupt changes to the steering angle. To avoid such unnatural motion and to ensure passenger comfort, the path needs to be continuously differentiable and therefore smooth [2]. One approach to smoothing a path involves fitting a parametric cubic spline. Spline fitting enables you to generate a smooth path that a controller can execute.

Use `smoothPathSpline` to fit a parametric cubic spline that passes through all the transition points in the reference path. The spline approximately matches the starting and ending directions with the starting and ending heading angle of the vehicle.

```
% Specify number of poses to return using a separation of approximately 0.1 m
approxSeparation = 0.1; % meters
numSmoothPoses   = round(refPath.Length / approxSeparation);

% Return discretized poses along the smooth path
[refPoses, directions, cumLengths, curvatures] = smoothPathSpline(transitionPoses, directions, nu

% Plot the smoothed path
```

```
hold on
hSmoothPath = plot(refPoses(:, 1), refPoses(:, 2), 'r', 'LineWidth', 2, ...
    'DisplayName', 'Smoothed Path');
hold off
```



Next, convert the generated smooth path to a trajectory that can be executed using a speed profile. Compute a speed profile for each path as a sequence of three phases: accelerating to a set maximum speed, maintaining the maximum speed and decelerating to a terminal speed. The `helperGenerateVelocityProfile` function generates such a speed profile.

Specify initial, maximum, and terminal speeds so that the vehicle starts stationary, accelerates to a speed of 5 meters/second, and comes to a stop.

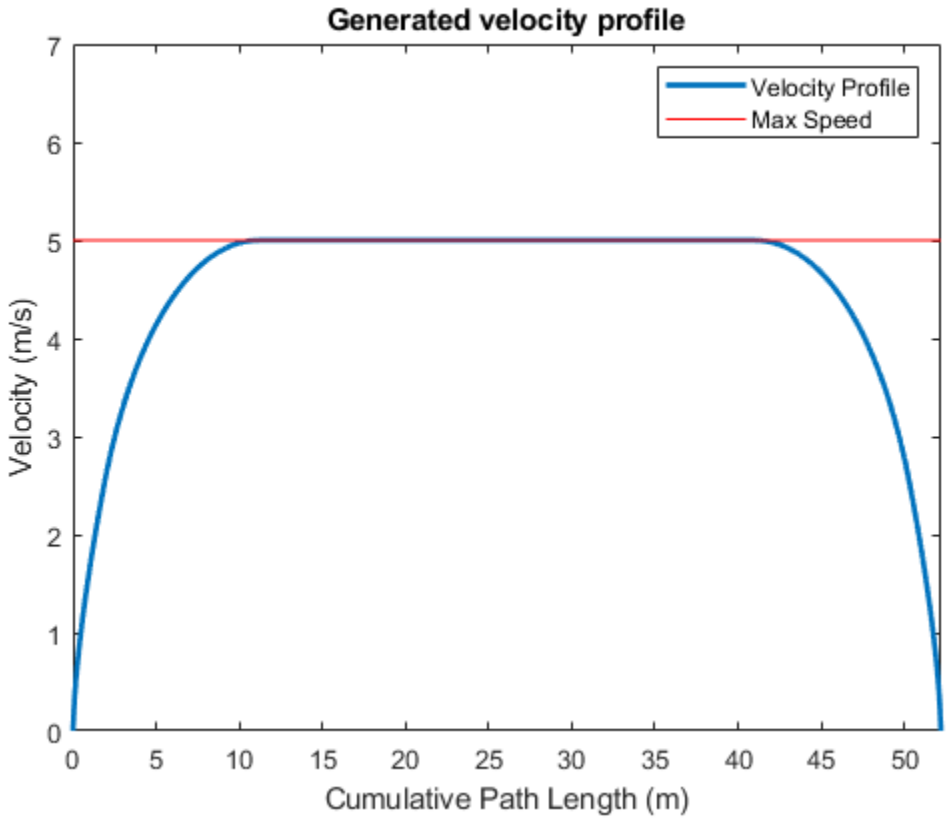
```
maxSpeed   = 5; % in meters/second
startSpeed = 0; % in meters/second
endSpeed   = 0; % in meters/second
```

Generate a velocity profile

```
refVelocities = helperGenerateVelocityProfile(directions, cumLengths, curvatures, startSpeed, end
```

`refVelocities` contains reference velocities for each point along the smoothed path. Plot the generated velocity profile.

```
plotVelocityProfile(cumLengths, refVelocities, maxSpeed)
```



Vehicle Control and Simulation

The reference speeds, together with the smoothed path, comprise a feasible trajectory that the vehicle can follow. A feedback controller is used to follow this trajectory. The controller corrects errors in tracking the trajectory that arise from tire slippage and other sources of noise, such as inaccuracies in localization. In particular, the controller consists of two components:

- **Lateral control**: Adjust the steering angle such that the vehicle follows the reference path.
- **Longitudinal control**: While following the reference path, maintain the desired speed by controlling the throttle and the brake.

Since this scenario involves slow speeds, you can simplify the controller to take into account only a kinematic model. In this example, lateral control is realized by the `lateralControllerStanley` function. The longitudinal control is realized by a helper System object™ `HelperLongitudinalController`, that computes acceleration and deceleration commands based on the Proportional-Integral law.

The feedback controller requires a simulator that can execute the desired controller commands using a suitable vehicle model. The `HelperVehicleSimulator` class simulates such a vehicle using the following kinematic bicycle model:

$$\dot{x}_r = v_r * \cos(\theta)$$

$$\dot{y}_r = v_r * \sin(\theta)$$

$$\dot{\theta} = \frac{v_r}{l} * \tan(\delta)$$

$$\dot{v}_r = a_r$$

In the above equations, (x_r, y_r, θ) represents the vehicle pose in world coordinates. v_r, a_r, l, and δ represent the rear-wheel speed, rear-wheel acceleration, wheelbase, and steering angle, respectively. The position and speed of the front wheel can be obtained by:

$$x_f = x_r + lcos(\theta)$$

$$y_f = y_r + lsin(\theta)$$

$$v_f = \frac{v_r}{cos(\delta)}$$

```
% Close all the figures
closeFigures;

% Create the vehicle simulator
vehicleSim = HelperVehicleSimulator(costmap, vehicleDims);

% Set the vehicle pose and velocity
vehicleSim.setVehiclePose(currentPose);
currentVel = 0;
vehicleSim.setVehicleVelocity(currentVel);

% Configure the simulator to show the trajectory
vehicleSim.showTrajectory(true);

% Hide vehicle simulation figure
hideFigure(vehicleSim);
```

Create a `HelperPathAnalyzer` object to compute reference pose, reference velocity and driving direction for the controller.

```
pathAnalyzer = HelperPathAnalyzer(refPoses, refVelocities, directions, ...
    'Wheelbase', vehicleDims.Wheelbase);
```

Create a `HelperLongitudinalController` object to control the velocity of the vehicle and specify the sample time.

```
sampleTime = 0.05;
lonController = HelperLongitudinalController('SampleTime', sampleTime);
```

Use the `HelperFixedRate` object to ensure fixed-rate execution of the feedback controller. Use a control rate to be consistent with the longitudinal controller.

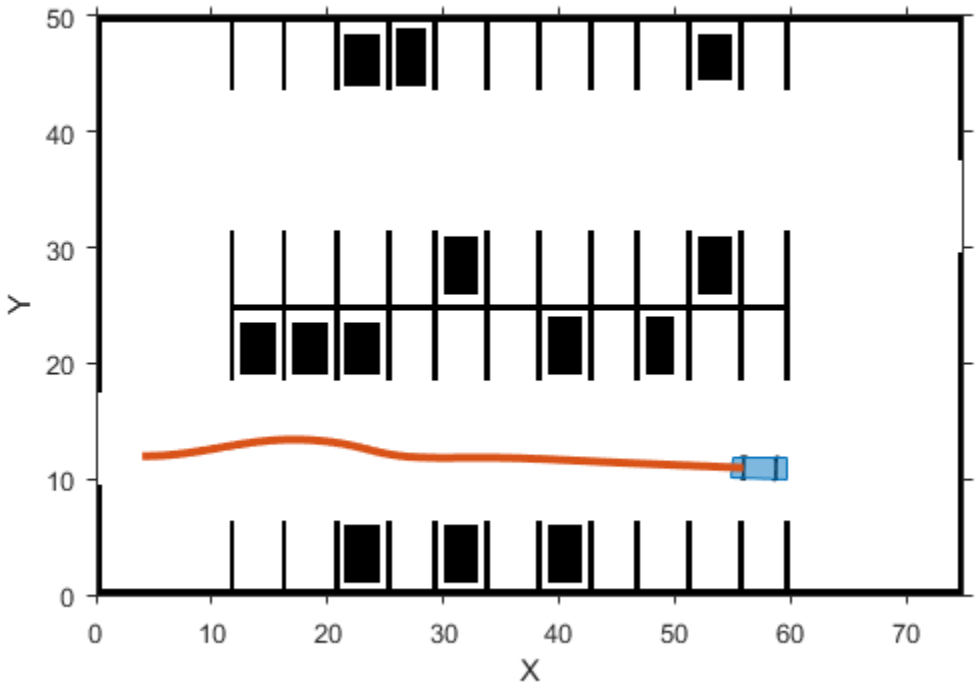
```
controlRate = HelperFixedRate(1/sampleTime); % in Hertz
```

Until the goal is reached, do the following:

- Compute steering and acceleration/deceleration commands required to track the planned trajectory.
- Feed control commands to the simulator.

- Record the returned vehicle pose and velocity to feed into the controller in the next iteration.

```
reachGoal = false;

while ~reachGoal
    % Find the reference pose on the path and the corresponding velocity
    [refPose, refVel, direction] = pathAnalyzer(currentPose, currentVel);

    % Update driving direction for the simulator
    updateDrivingDirection(vehicleSim, direction);

    % Compute steering command
    steeringAngle = lateralControllerStanley(refPose, currentPose, currentVel, ...
        'Direction', direction, 'Wheelbase', vehicleDims.Wheelbase);

    % Compute acceleration and deceleration commands
    lonController.Direction = direction;
    [accelCmd, decelCmd] = lonController(refVel, currentVel);

    % Simulate the vehicle using the controller outputs
    drive(vehicleSim, accelCmd, decelCmd, steeringAngle);

    % Check if the vehicle reaches the goal
    reachGoal = helperGoalChecker(goalPose, currentPose, currentVel, endSpeed, direction);

    % Wait for fixed-rate execution
    waitfor(controlRate);

    % Get current pose and velocity of the vehicle
    currentPose  = getVehiclePose(vehicleSim);
    currentVel   = getVehicleVelocity(vehicleSim);
end

% Show vehicle simulation figure
showFigure(vehicleSim);
```



This completes the first leg of the route plan and demonstrates each step of the process. The next sections run the simulator for the entire route, which takes the vehicle close to the parking spot, and finally executes a parking maneuver to place the vehicle into the parking spot.

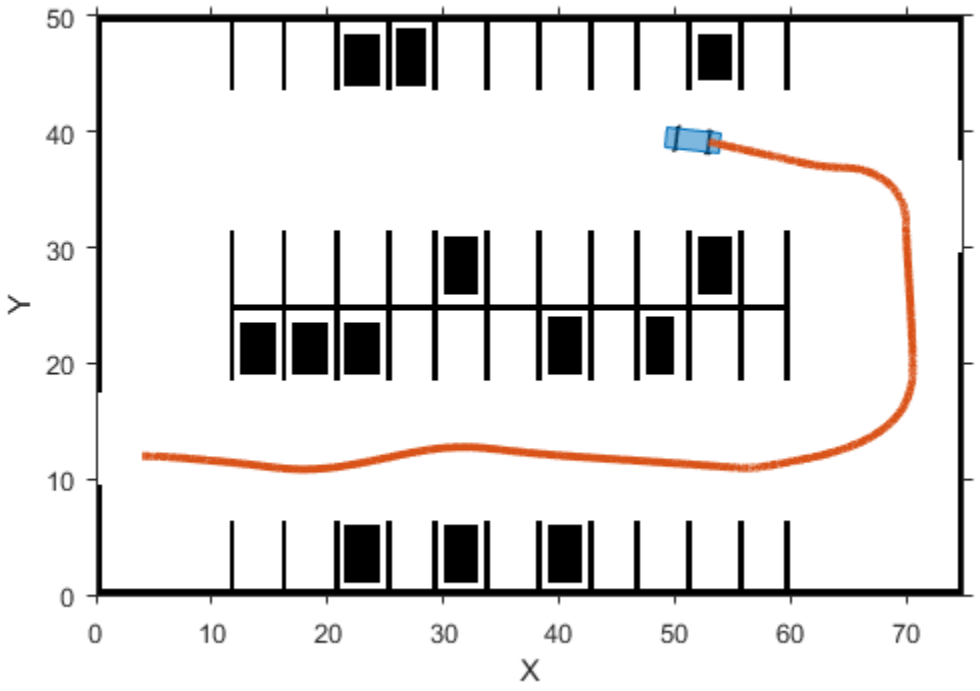
Execute a Complete Plan

Now combine all the previous steps in the planning process and run the simulation for the complete route plan. This process involves incorporating the behavioral planner.

```
% Set the vehicle pose back to the initial starting point
currentPose = [4 12 0]; % [x, y, theta]
vehicleSim.setVehiclePose(currentPose);

% Reset velocity
currentVel  = 0; % meters/second
vehicleSim.setVehicleVelocity(currentVel);

while ~reachedDestination(behavioralPlanner)

    % Request next maneuver from behavioral layer
    [nextGoal, plannerConfig, speedConfig] = requestManeuver(behavioralPlanner, ...
        currentPose, currentVel);

    % Configure the motion planner
    configurePlanner(motionPlanner, plannerConfig);

    % Plan a reference path using RRT* planner to the next goal pose
```

```
refPath = plan(motionPlanner, currentPose, nextGoal);

% Check if the path is valid. If the planner fails to compute a path,
% or the path is not collision-free because of updates to the map, the
% system needs to re-plan. This scenario uses a static map, so the path
% will always be collision-free.
isReplanNeeded = ~checkPathValidity(refPath, costmap);
if isReplanNeeded
    warning('Unable to find a valid path. Attempting to re-plan.')

    % Request behavioral planner to re-plan
    replanNeeded(behavioralPlanner);
    continue;
end

% Retrieve transition poses and directions from the planned path
[transitionPoses, directions] = interpolate(refPath);

% Smooth the path
numSmoothPoses   = round(refPath.Length / approxSeparation);
[refPoses, directions, cumLengths, curvatures] = smoothPathSpline(transitionPoses, directions

% Generate a velocity profile
refVelocities = helperGenerateVelocityProfile(directions, cumLengths, curvatures, startSpeed

% Configure path analyzer
pathAnalyzer.RefPoses     = refPoses;
pathAnalyzer.Directions   = directions;
pathAnalyzer.VelocityProfile = refVelocities;

% Reset longitudinal controller
reset(lonController);

reachGoal = false;

% Execute control loop
while ~reachGoal
    % Find the reference pose on the path and the corresponding velocity
    [refPose, refVel, direction] = pathAnalyzer(currentPose, currentVel);

    % Update driving direction for the simulator
    updateDrivingDirection(vehicleSim, direction);

    % Compute steering command
    steeringAngle = lateralControllerStanley(refPose, currentPose, currentVel, ...
        'Direction', direction, 'Wheelbase', vehicleDims.Wheelbase);

    % Compute acceleration and deceleration commands
    lonController.Direction = direction;
    [accelCmd, decelCmd] = lonController(refVel, currentVel);

    % Simulate the vehicle using the controller outputs
    drive(vehicleSim, accelCmd, decelCmd, steeringAngle);

    % Check if the vehicle reaches the goal
    reachGoal = helperGoalChecker(nextGoal, currentPose, currentVel, speedConfig.EndSpeed, di

    % Wait for fixed-rate execution
```

```
        waitfor(controlRate);

        % Get current pose and velocity of the vehicle
        currentPose  = getVehiclePose(vehicleSim);
        currentVel   = getVehicleVelocity(vehicleSim);
    end
end

% Show vehicle simulation figure
showFigure(vehicleSim);
```



Parking Maneuver

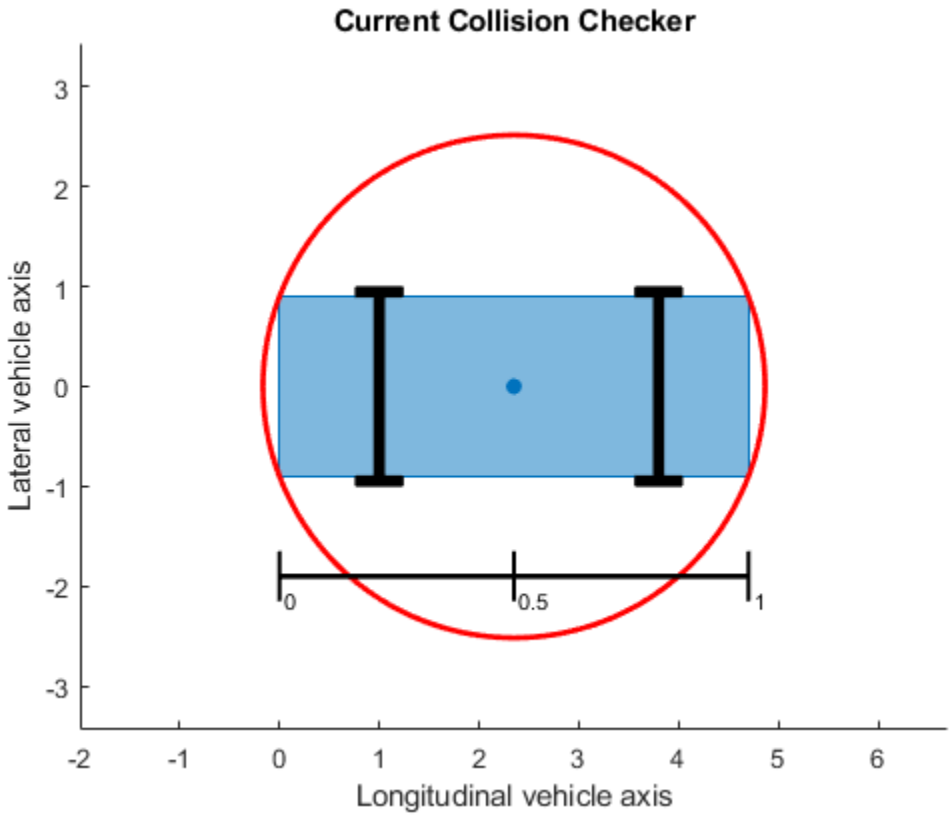
Now that the vehicle is near the parking spot, a specialized parking maneuver is used to park the vehicle in the final parking spot. This maneuver requires passing through a narrow corridor flanked by the edges of the parking spot on both ends. Such a maneuver is typically accompanied with ultrasound sensors or laser scanners continuously checking for obstacles.

```
% Hide vehicle simulation figure
hideFigure(vehicleSim);
```

The `vehicleCostmap` uses inflation-based collision checking. First, visually inspect the current collision checker in use.

```
ccConfig = costmap.CollisionChecker;

figure
```

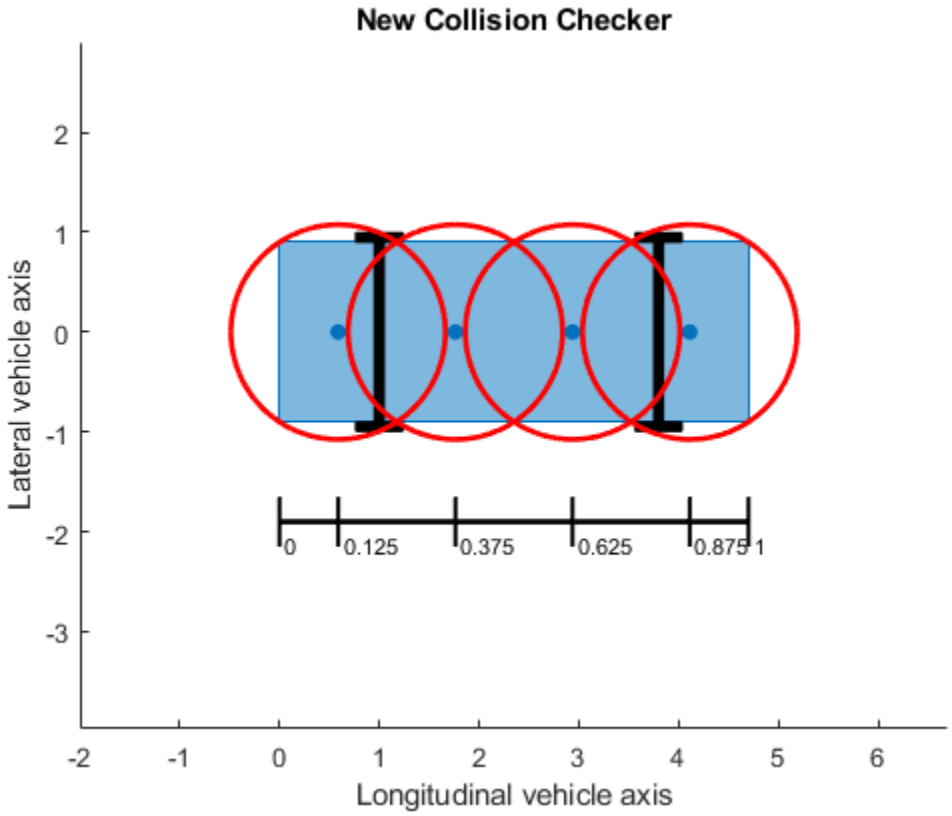
```
plot(ccConfig)
title('Current Collision Checker')
```



Collision checking is performed by inflating obstacles in the costmap by the inflation radius, and checking whether the center of the circle shown above lies on an inflated grid cell. The final parking maneuver requires a more precise, less conservative collision-checking mechanism. This is commonly solved by representing the shape of the vehicle using multiple (3-5) overlapping circles instead of a single circle.

Use a larger number of circles in the collision checker and visually inspect the collision checker. This allows planning through narrow passages.

```
ccConfig.NumCircles = 4;

figure
plot(ccConfig)
title('New Collision Checker')
```

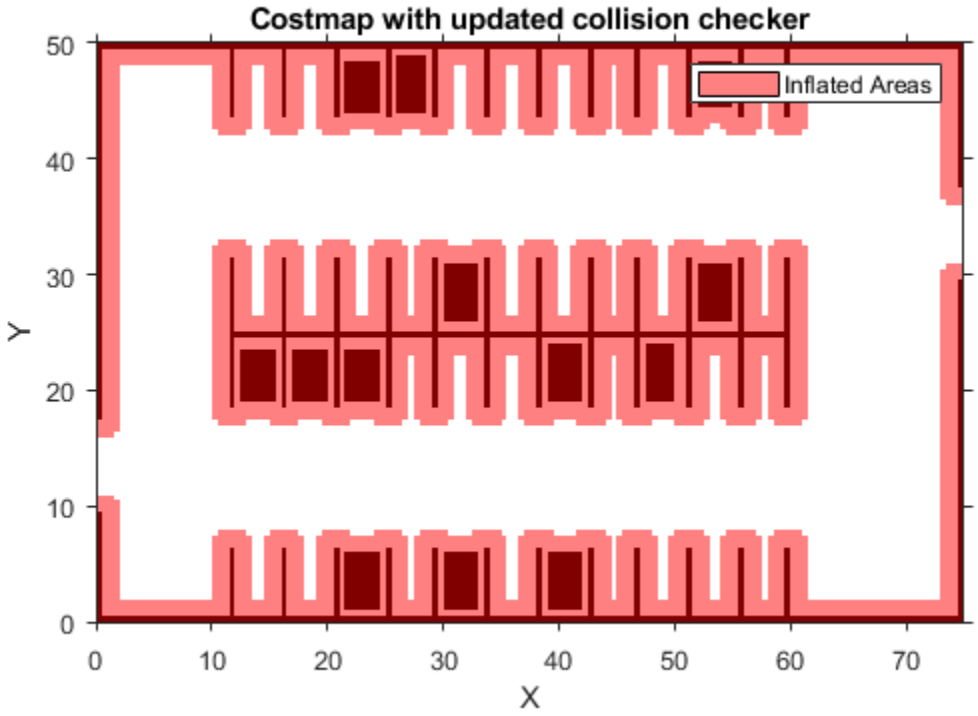


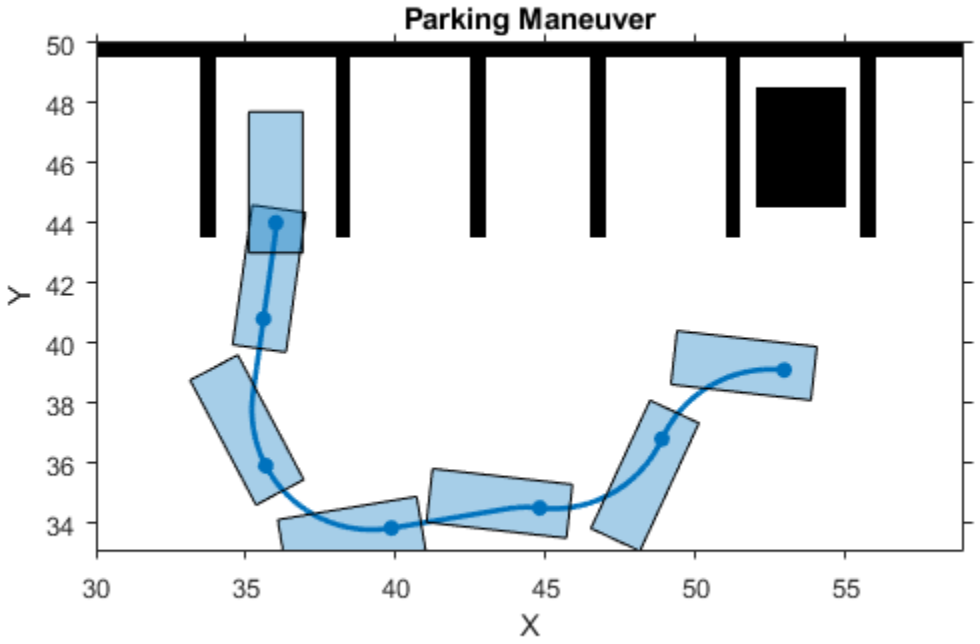
Update the costmap to use this collision checker.

```
costmap.CollisionChecker = ccConfig;
```

Notice that the inflation radius has reduced, allowing the planner to find an unobstructed path to the parking spot.

```
figure
plot(costmap)
title('Costmap with updated collision checker')

% Set up the pathPlannerRRT to use the updated costmap
parkMotionPlanner = pathPlannerRRT(costmap, 'MinIterations', 1000);

% Define desired pose for the parking spot, returned by the V2X system
parkPose = [36 44 90];
preParkPose = currentPose;

% Compute the required parking maneuver
refPath = plan(parkMotionPlanner, preParkPose, parkPose);

% Plot the resulting parking maneuver
figure
plotParkingManeuver(costmap, refPath, preParkPose, parkPose)
```

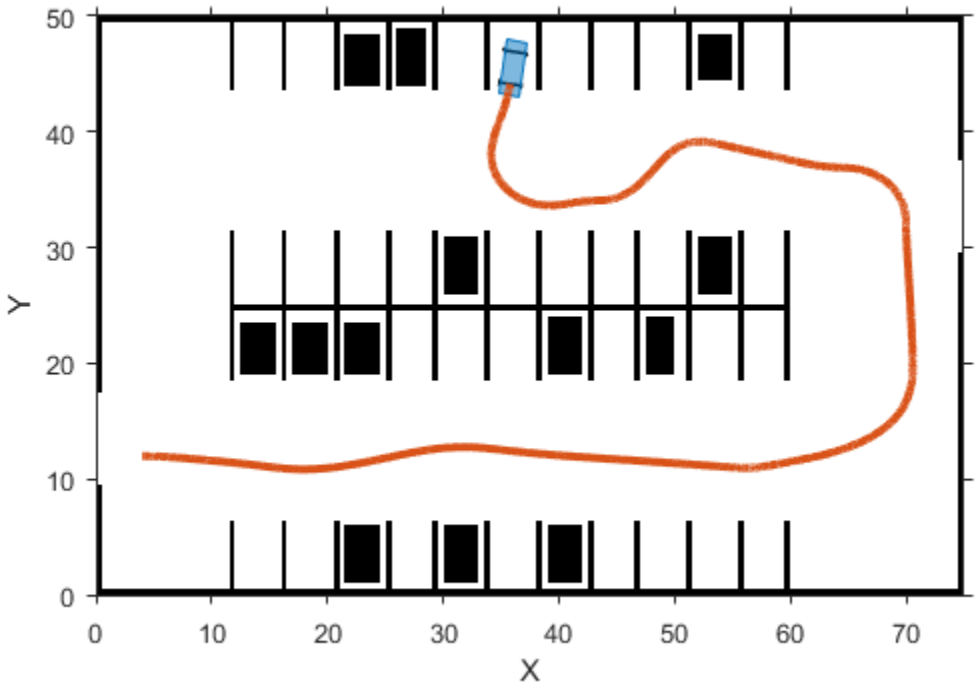

Costmap with updated collision checker
Inflated Areas
Y
X
0
10
20
30
40
50
60
70



Once the maneuver is found, repeat the previous process to determine a complete plan: smooth the path, generate a speed profile and follow the trajectory using the feedback controller.

```
% Retrieve transition poses and directions from the planned path
[transitionPoses, directions] = interpolate(refPath);

% Smooth the path
numSmoothPoses   = round(refPath.Length / approxSeparation);
[refPoses, directions, cumLengths, curvatures] = smoothPathSpline(transitionPoses, directions, nu

% Set up the velocity profile generator to stop at the end of the trajectory,
% with a speed limit of 5 mph
refVelocities = helperGenerateVelocityProfile(directions, cumLengths, curvatures, currentVel, 0,

pathAnalyzer.RefPoses     = refPoses;
pathAnalyzer.Directions   = directions;
pathAnalyzer.VelocityProfile = refVelocities;

% Reset longitudinal controller
reset(lonController);

reachGoal = false;

while ~reachGoal
    % Find the reference pose on the path and the corresponding velocity
    [refPose, refVel, direction] = pathAnalyzer(currentPose, currentVel);
```

```
        % Update driving direction for the simulator
        updateDrivingDirection(vehicleSim, direction);

        % Compute steering command
        steeringAngle = lateralControllerStanley(refPose, currentPose, currentVel, ...
            'Direction', direction, 'Wheelbase', vehicleDims.Wheelbase);

        % Compute acceleration and deceleration commands
        lonController.Direction = direction;
        [accelCmd, decelCmd] = lonController(refVel, currentVel);

        % Simulate the vehicle using the controller outputs
        drive(vehicleSim, accelCmd, decelCmd, steeringAngle);

        % Check if the vehicle reaches the goal
        reachGoal = helperGoalChecker(parkPose, currentPose, currentVel, 0, direction);

        % Wait for fixed-rate execution
        waitfor(controlRate);

        % Get current pose and velocity of the vehicle
        currentPose  = getVehiclePose(vehicleSim);
        currentVel   = getVehicleVelocity(vehicleSim);
    end

    % Show vehicle simulation figure
    closeFigures;
    showFigure(vehicleSim);
```



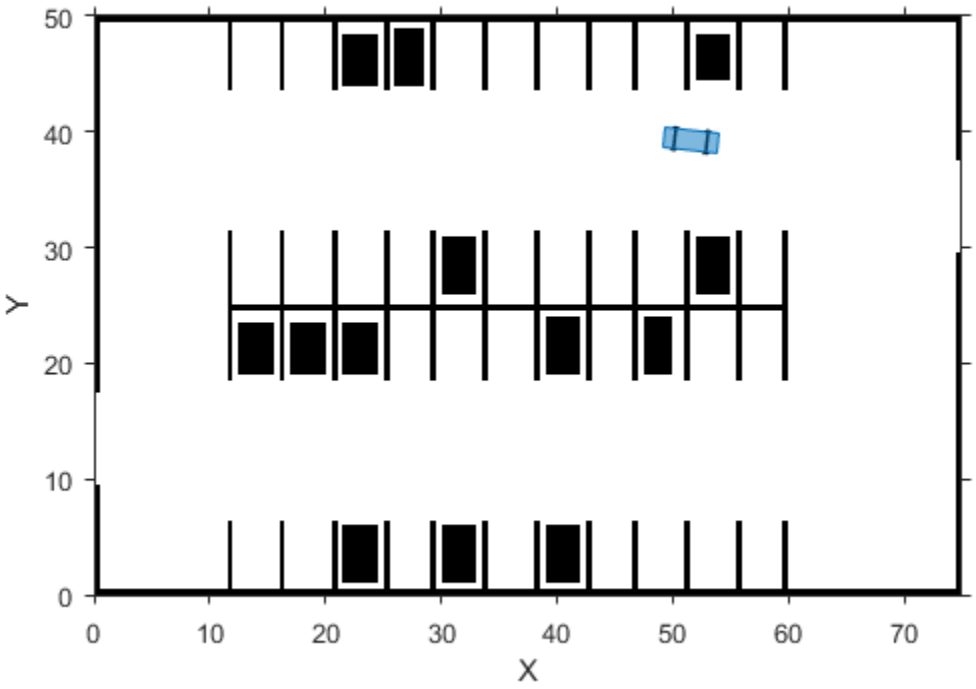
An alternative way to park the vehicle is to back into the parking spot. When the vehicle needs to back up into a spot, the motion planner needs to use the Reeds-Shepp connection method to search for a feasible path. The Reeds-Shepp connection allows for reverse motions during planning.

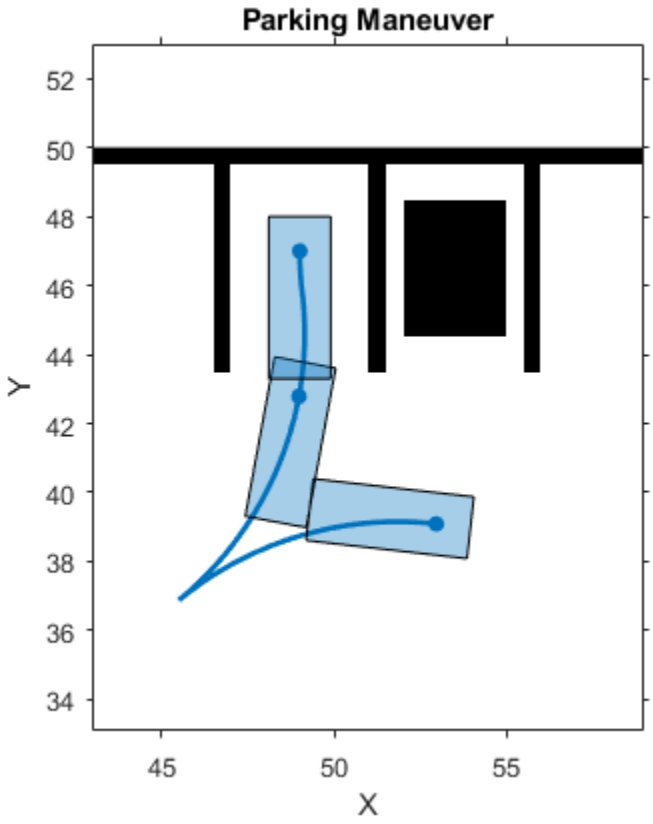
```
% Specify a parking pose corresponding to a back-in parking maneuver
parkPose = [49 47 -90];

% Change the connection method to allow for reverse motions
parkMotionPlanner.ConnectionMethod = 'Reeds-Shepp';
```

To find a feasible path, the motion planner needs to be adjusted. Use a larger turning radius and connection distance to allow for a smooth back-in.

```
parkMotionPlanner.MinTurningRadius   = 10; % meters
parkMotionPlanner.ConnectionDistance = 15;

% Reset vehicle pose and velocity
currentVel = 0;
vehicleSim.setVehiclePose(preParkPose);
vehicleSim.setVehicleVelocity(currentVel);

% Compute the parking maneuver
replan = true;
while replan
    refPath = plan(parkMotionPlanner, preParkPose, parkPose);

    % The path corresponding to the parking maneuver is small and requires
```

```
        % precise maneuvering. Instead of interpolating only at transition poses,
        % interpolate more finely along the length of the path.

        numSamples = 10;
        stepSize   = refPath.Length / numSamples;
        lengths    = 0 : stepSize : refPath.Length;

        [transitionPoses, directions] = interpolate(refPath, lengths);

        % Replan if the path contains more than one direction switching poses
        % or if the path is too long
        replan = sum(abs(diff(directions)))~=2 || refPath.Length > 20;
    end

    % Visualize the parking maneuver
    figure
    plotParkingManeuver(costmap, refPath, preParkPose, parkPose)
```

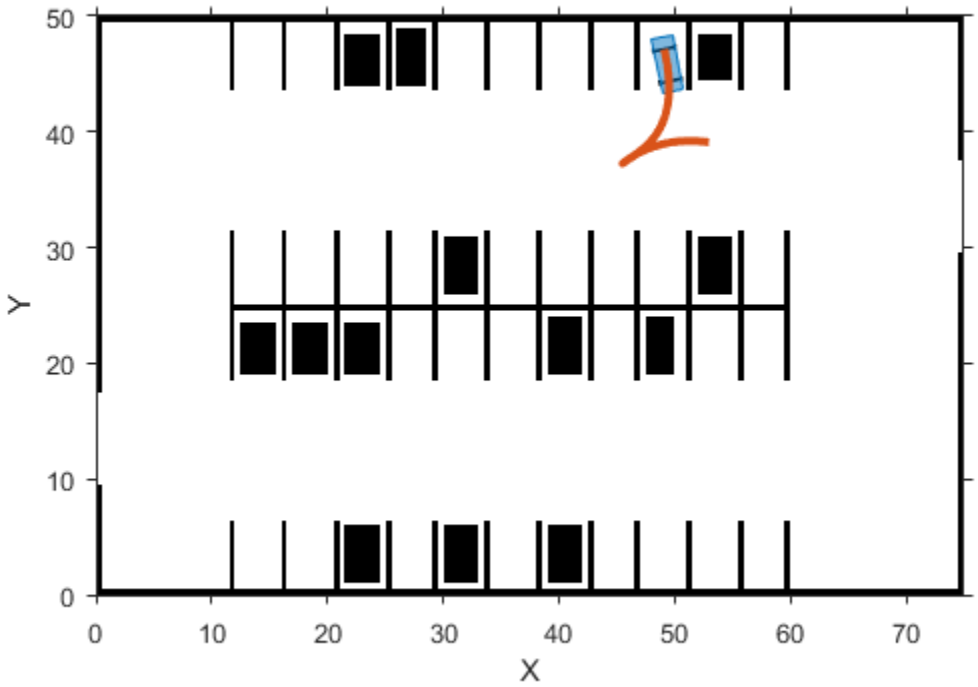




Smooth the path

```
numSmoothPoses   = round(refPath.Length / approxSeparation);
[refPoses, directions, cumLengths, curvatures] = smoothPathSpline(transitionPoses, directions, nu

% Generate velocity profile
refVelocities = helperGenerateVelocityProfile(directions, cumLengths, curvatures, currentVel, 0,

pathAnalyzer.RefPoses     = refPoses;
pathAnalyzer.Directions   = directions;
pathAnalyzer.VelocityProfile = refVelocities;

% Reset longitudinal controller
reset(lonController);

reachGoal = false;

while ~reachGoal
    % Get current driving direction
    currentDir = getDrivingDirection(vehicleSim);

    % Find the reference pose on the path and the corresponding velocity.
    [refPose, refVel, direction] = pathAnalyzer(currentPose, currentVel);

    % If the vehicle changes driving direction, reset vehicle velocity in
    % the simulator and reset longitudinal controller
    if currentDir ~= direction
```

```
            currentVel = 0;
            setVehicleVelocity(vehicleSim, currentVel);
            reset(lonController);
        end

        % Update driving direction for the simulator. If the vehicle changes
        % driving direction, reset and return the current vehicle velocity as zero.
        currentVel = updateDrivingDirection(vehicleSim, direction, currentDir);

        % Compute steering command
        steeringAngle = lateralControllerStanley(refPose, currentPose, currentVel, ...
            'Direction', direction, 'Wheelbase', vehicleDims.Wheelbase);

        % Compute acceleration and deceleration commands
        lonController.Direction = direction;
        [accelCmd, decelCmd] = lonController(refVel, currentVel);

        % Simulate the vehicle using the controller outputs
        drive(vehicleSim, accelCmd, decelCmd, steeringAngle);

        % Check if the vehicle reaches the goal
        reachGoal = helperGoalChecker(parkPose, currentPose, currentVel, 0, direction);

        % Wait for fixed-rate execution
        waitfor(controlRate);

        % Get current pose and velocity of the vehicle
        currentPose  = getVehiclePose(vehicleSim);
        currentVel   = getVehicleVelocity(vehicleSim);
    end

    % Take a snapshot for the example
    closeFigures;
    snapnow;

    % Delete the simulator
    delete(vehicleSim);
```



Conclusion

This example showed how to:

1. Plan a feasible path in a semi-structured environment, such as a parking lot, using an RRT* path planning algorithm.
2. Smooth the path using splines and generate a speed profile along the smoothed path.
3. Control the vehicle to follow the reference path at the desired speed.
4. Realize different parking behaviors by using different motion planner settings.

References

[1] Buehler, Martin, Karl Iagnemma, and Sanjiv Singh. *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic* (1st ed.). Springer Publishing Company, Incorporated, 2009.

[2] Lepetic, Marko, Gregor Klancar, Igor Skrjanc, Drago Matko, Bostjan Potocnik, "Time Optimal Path Planning Considering Acceleration Limits." *Robotics and Autonomous Systems*. Volume 45, Issues 3-4, 2003, pp. 199-210.

Supporting Functions

loadParkingLotMapLayers Load environment map layers for parking lot

```
function mapLayers = loadParkingLotMapLayers()
%loadParkingLotMapLayers
%   Load occupancy maps corresponding to 3 layers - obstacles, road
```

```
%   markings, and used spots.

mapLayers.StationaryObstacles = imread('stationary.bmp');
mapLayers.RoadMarkings        = imread('road_markings.bmp');
mapLayers.ParkedCars          = imread('parked_cars.bmp');
end
```

plotMapLayers Plot struct containing map layers

```
function plotMapLayers(mapLayers)
%plotMapLayers
%   Plot the multiple map layers on a figure window.

figure
cellOfMaps = cellfun(@imcomplement, struct2cell(mapLayers), 'UniformOutput', false);
montage( cellOfMaps, 'Size', [1 numel(cellOfMaps)], 'Border', [5 5], 'ThumbnailSize', [300 NaN] )
title('Map Layers - Stationary Obstacles, Road markings, and Parked Cars')
end
```

combineMapLayers Combine map layers into a single costmap

```
function costmap = combineMapLayers(mapLayers)
%combineMapLayers
%   Combine map layers struct into a single vehicleCostmap.

combinedMap = mapLayers.StationaryObstacles + mapLayers.RoadMarkings + ...
    mapLayers.ParkedCars;
combinedMap = im2single(combinedMap);

res = 0.5; % meters
costmap = vehicleCostmap(combinedMap, 'CellSize', res);
end
```

configurePlanner Configure path planner with specified settings

```
function configurePlanner(pathPlanner, config)
%configurePlanner
% Configure the path planner object, pathPlanner, with settings specified
% in struct config.

fieldNames = fields(config);
for n = 1 : numel(fieldNames)
    if ~strcmpi(fieldNames{n}, 'IsParkManeuver')
        pathPlanner.(fieldNames{n}) = config.(fieldNames{n});
    end
end
end
```

plotVelocityProfile Plot speed profile

```
function plotVelocityProfile(cumPathLength, refVelocities, maxSpeed)
%plotVelocityProfile
% Plot the generated velocity profile

% Plot reference velocity along length of the path
plot(cumPathLength, refVelocities, 'LineWidth', 2);

% Plot a line to display maximum speed
hold on
```

```
line([0;cumPathLength(end)], [maxSpeed;maxSpeed], 'Color', 'r')
hold off

% Set axes limits
buffer = 2;
xlim([0 cumPathLength(end)]);
ylim([0 maxSpeed + buffer])

% Add labels
xlabel('Cumulative Path Length (m)');
ylabel('Velocity (m/s)');

% Add legend and title
legend('Velocity Profile', 'Max Speed')
title('Generated velocity profile')
end
```

closeFigures

```
function closeFigures()
% Close all the figures except the simulator visualization

% Find all the figure objects
figHandles = findobj('Type', 'figure');
for i = 1: length(figHandles)
    if ~strcmp(figHandles(i).Name, 'Automated Valet Parking')
        close(figHandles(i));
    end
end
end
```

plotParkingManeuver Display the generated parking maneuver on a costmap

```
function plotParkingManeuver(costmap, refPath, currentPose, parkPose)
%plotParkingManeuver
% Plot the generated parking maneuver on a costmap.

% Plot the costmap, without inflated areas
plot(costmap, 'Inflation', 'off')

% Plot reference parking maneuver on the costmap
hold on
plot(refPath, 'DisplayName', 'Parking Maneuver')

title('Parking Maneuver')

% Zoom into parking maneuver by setting axes limits
lo = min([currentPose(1:2); parkPose(1:2)]);
hi = max([currentPose(1:2); parkPose(1:2)]);

buffer = 6; % meters

xlim([lo(1)-buffer hi(1)+buffer])
```

```
ylim([lo(2)-buffer hi(2)+buffer])
end
```

See Also

Functions
`inflationCollisionChecker` | `interpolate` | `lateralControllerStanley` | `smoothPathSpline`

Objects
`driving.Path` | `pathPlannerRRT` | `vehicleCostmap` | `vehicleDimensions`

More About

- “Automated Parking Valet in Simulink” on page 7-493
- “Automated Parking Valet with ROS in MATLAB” (ROS Toolbox)
- “Automated Parking Valet with ROS 2 in MATLAB” (ROS Toolbox)
- “Coordinate Systems in Automated Driving Toolbox” on page 1-2

Automated Parking Valet in Simulink

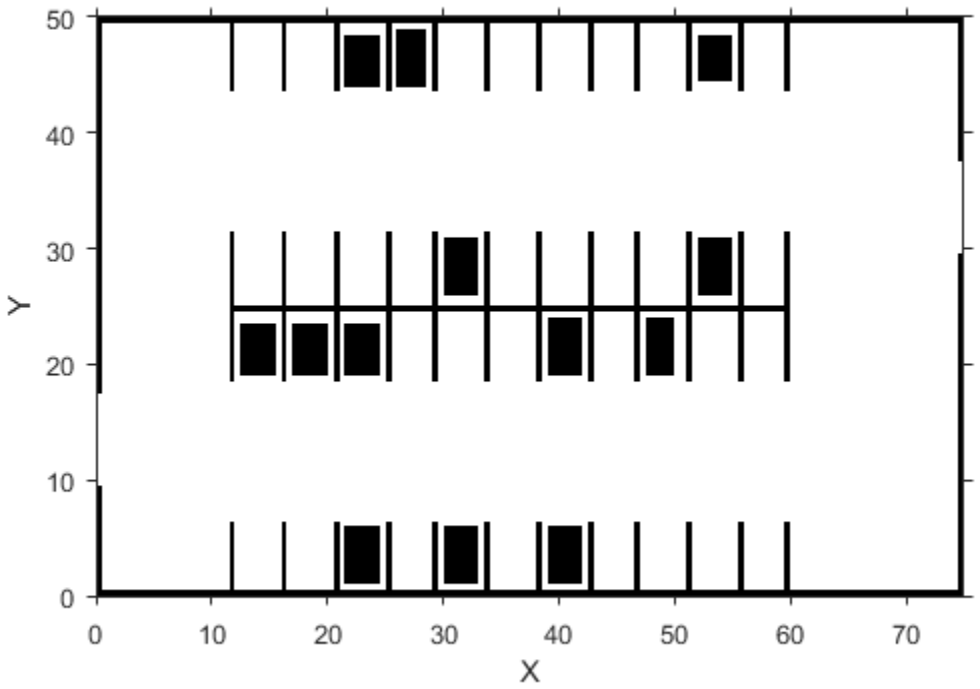
This example shows how to construct an automated parking valet system in Simulink® with Automated Driving Toolbox™. It closely follows the “Automated Parking Valet” on page 7-465 MATLAB® example.

Introduction

Automatically parking a car that is left in front of a parking lot is a challenging problem. The vehicle's automated systems are expected to take over and steer the vehicle to an available parking spot. This example focuses on planning a feasible path through the environment, generating a trajectory from this path, and using a feasible controller to execute the trajectory. Map creation and dynamic obstacle avoidance are excluded from this example.

Before simulation, the `helperSLCreateCostmap` function is called within the `PreLoadFcn` callback function of the model. For details on using callback functions, see “Model Callbacks” (Simulink). The `helperSLCreateCostmap` function creates a static map of the parking lot that contains information about stationary obstacles, road markings, and parked cars. The map is represented as a `vehicleCostmap` object.

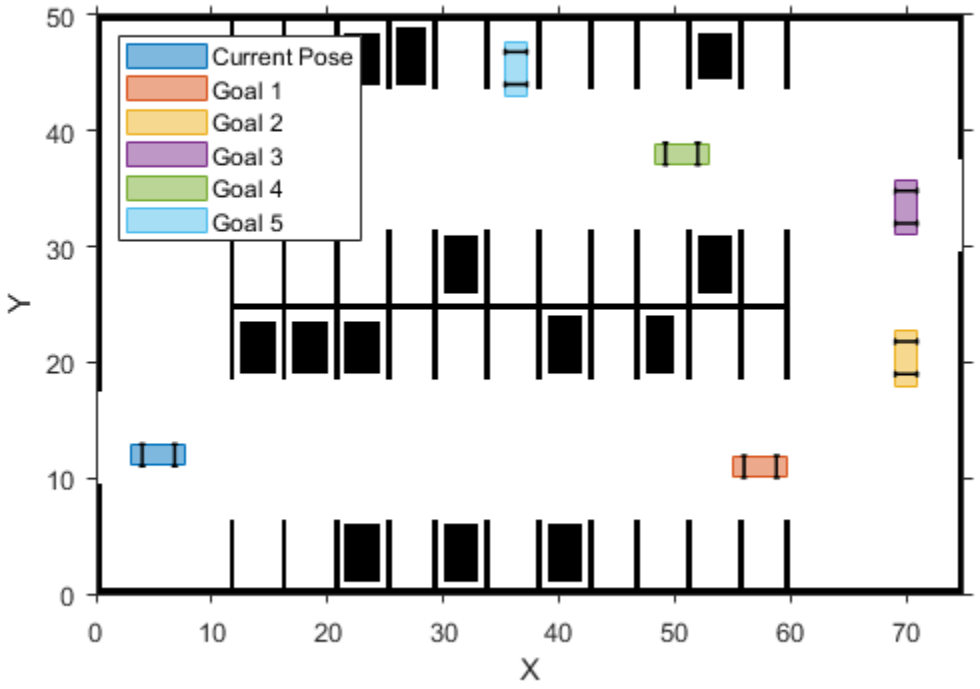
To use the `vehicleCostmap` object in Simulink®, the `helperSLCreateUtilityStruct` function converts the `vehicleCostmap` into a struct array in the block's mask initialization. For more details, see “Initialize Mask” (Simulink).



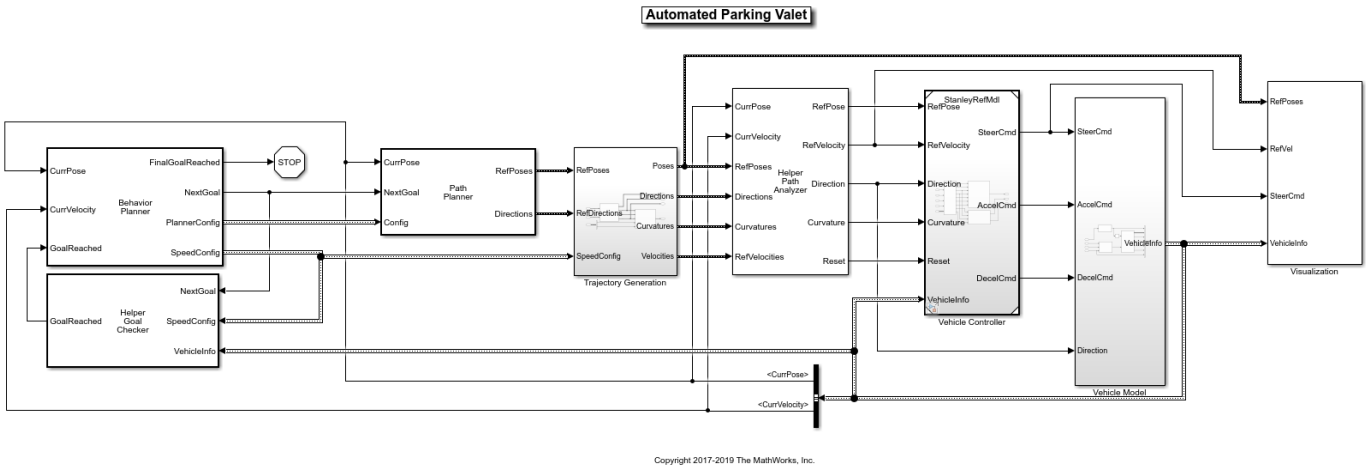
The global route plan is described as a sequence of lane segments to traverse to reach a parking spot. Before simulation, the `PreLoadFcn` callback function of the model loads a route plan, which is stored

as a table. The table specifies the start and end poses of the segment, as well as properties of the segment, such as the speed limit.

```
routePlan =

  5×3 table

      StartPose              EndPose             Attributes
    _______________    ___________________    _____________

     4     12      0      56      11       0    [1×1 struct]
    56     11      0      70      19      90    [1×1 struct]
    70     19     90      70      32      90    [1×1 struct]
    70     32     90      52      38     180    [1×1 struct]
    53     38    180    36.3      44      90    [1×1 struct]
```



The inputs and outputs of many blocks in this example are Simulink buses (`Simulink.Bus (Simulink)` classes). In the `PreLoadFcn` callback function of the model, the `helperSLCreateUtilityBus` function creates these buses.

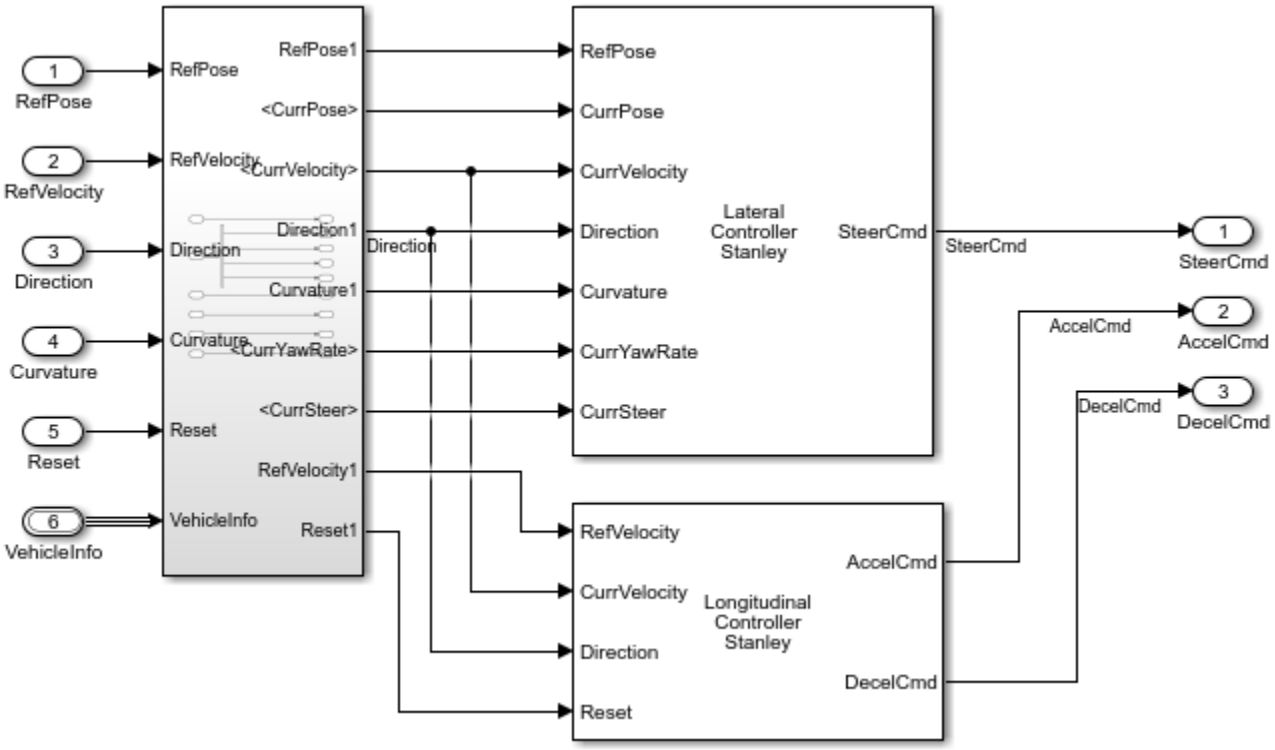


Planning is a hierarchical process, with each successive layer responsible for a more fine-grained task. The behavior layer [1] sits at the top of this stack. The **Behavior Planner** block triggers a sequence of navigation tasks based on the global route plan by providing an intermediate goal and configuration for the **Motion Planning** and **Trajectory Generation** blocks. Each path segment is navigated using these steps:

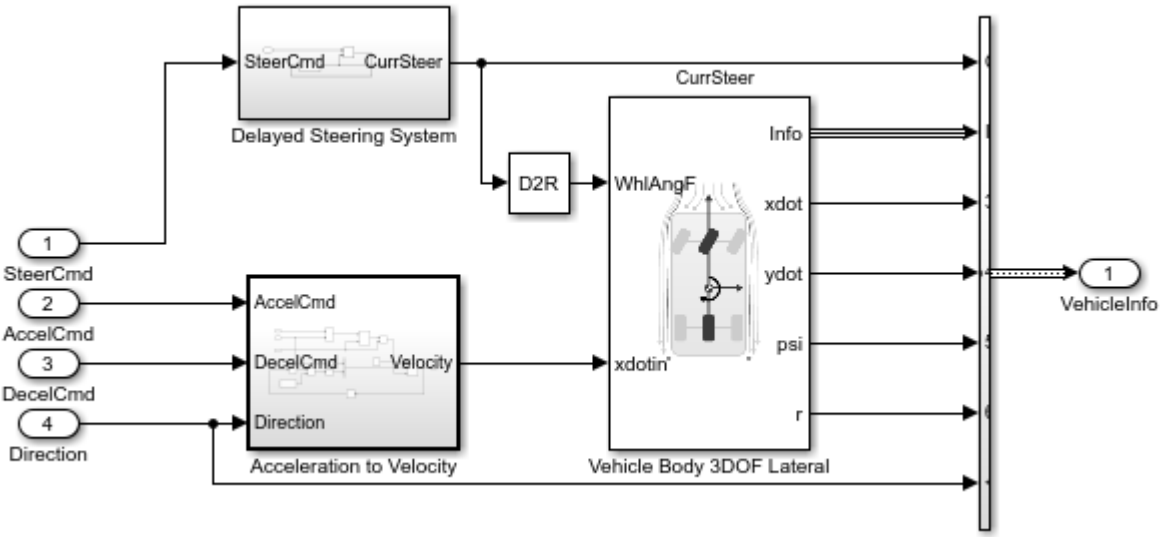
1. **Motion Planning**: Plan a feasible path through the environment map using the optimal rapidly exploring random tree (RRT*) algorithm (`pathPlannerRRT`).
2. **Trajectory Generation**: Smooth the reference path by fitting splines [2] to it using the Path Smoother Spline block. Then convert the smoothed path into a trajectory by generating a speed profile using the Velocity Profiler block.
3. **Vehicle Control**: The `HelperPathAnalyzer` provides the reference signal for the Vehicle Controller subsystem that controls the steering and the velocity of the vehicle.
4. **Goal Checking**: Check if the vehicle has reached the final pose of the segment using `helperGoalChecker`.

Explore the Subsystems

The Vehicle Controller subsystem contains a Lateral Controller Stanley block and a Longitudinal Controller Stanley block to regulate the pose and the velocity of the vehicle, respectively. To handle realistic vehicle dynamics [3], the **Vehicle model** parameter in the Lateral Controller Stanley block is set to `Dynamic bicycle model`. With this configuration, additional inputs, such as the path curvature, the current yaw rate of the vehicle, and the current steering angle are required to compute the steering command. The Longitudinal Controller Stanley block uses a switching Proportional-Integral controller to calculate the acceleration and the deceleration commands that actuate the brake and throttle in the vehicle.

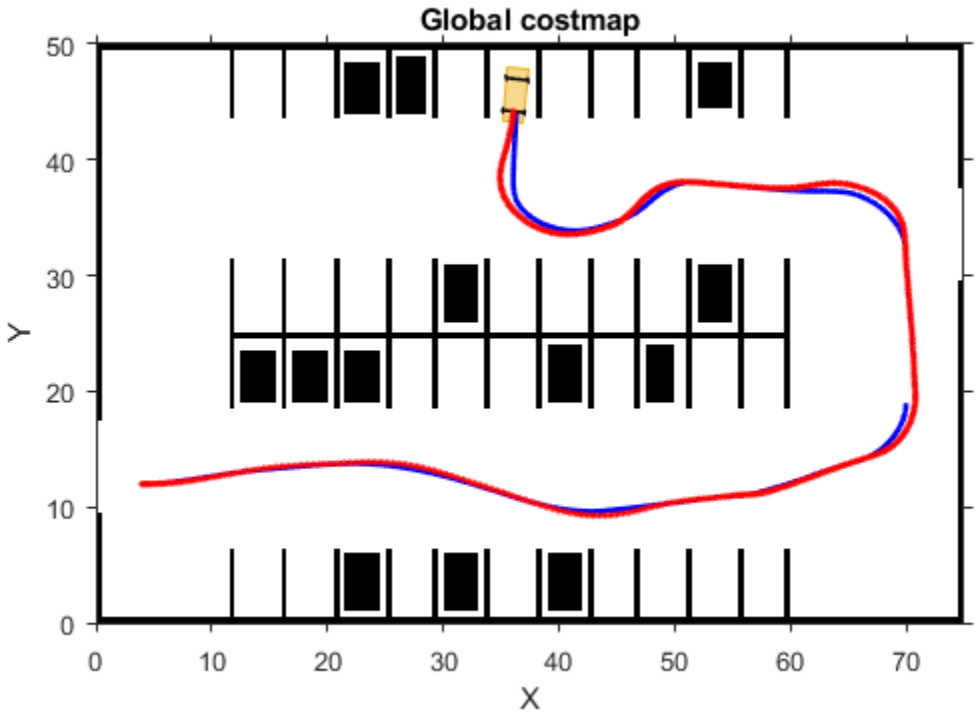


To demonstrate the performance, the vehicle controller is applied to the Vehicle Model block, which contains a simplified steering system [3] that is modeled as a first-order system and a Vehicle Body 3DOF (Vehicle Dynamics Blockset) block shared between Automated Driving Toolbox™ and Vehicle Dynamics Blockset™. Compared with the kinematic bicycle model used in the “Automated Parking Valet” on page 7-465 MATLAB® example, this Vehicle Model block is more accurate because it considers the inertial effects, such as tire slip and steering servo actuation.

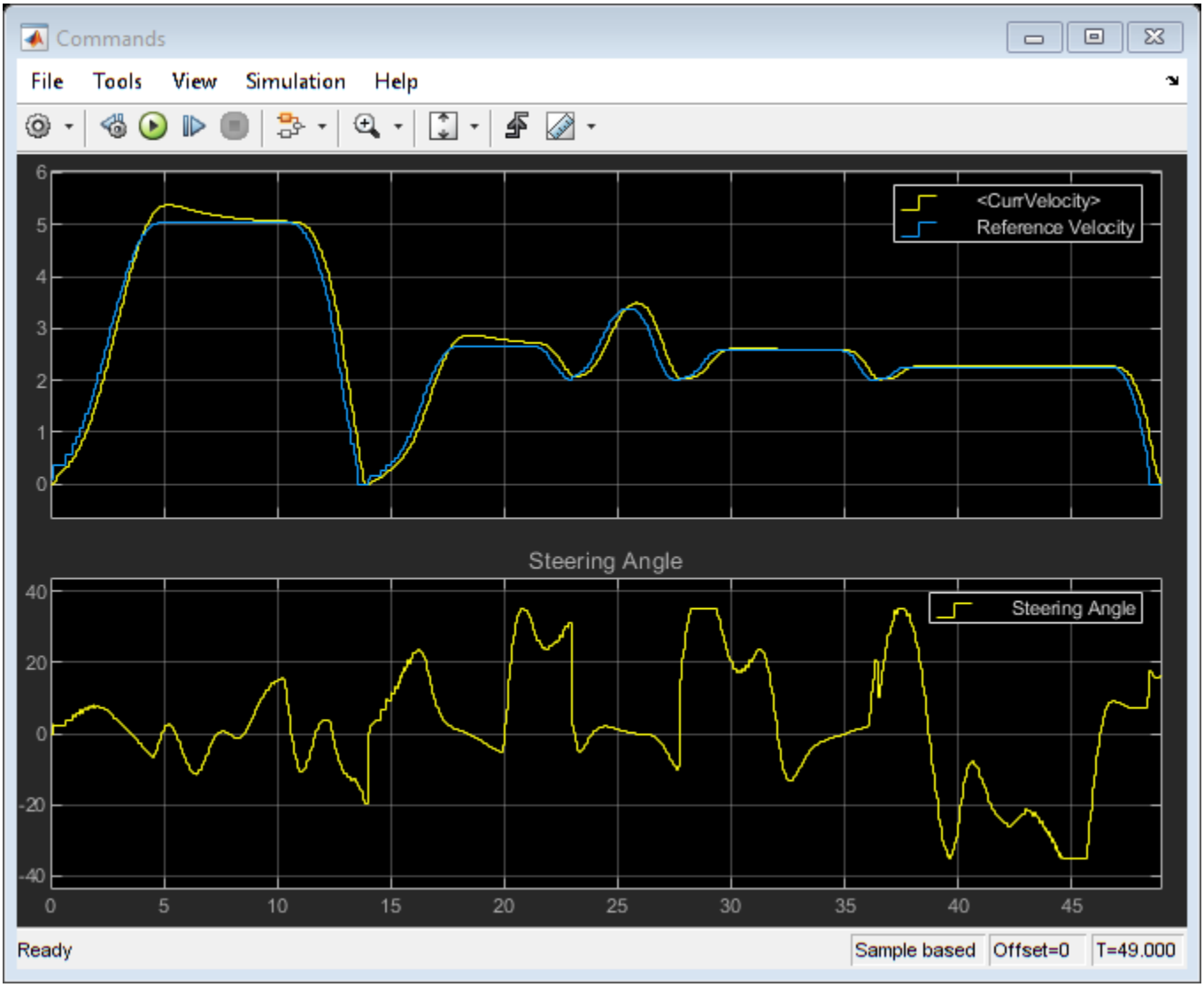


Simulation Results

The Visualization block shows how the vehicle tracks the reference path. It also displays vehicle speed and steering command in a scope. The following images are the simulation results for this example:



Simulation stops at about 45 seconds, which is when the vehicle reaches the destination.



Conclusions

This example shows how to implement an automated parking valet in Simulink.

References

[1] Buehler, Martin, Karl Iagnemma, and Sanjiv Singh. *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic* (1st ed.). Springer Publishing Company, Incorporated, 2009.

[2] Lepetic, Marko, Gregor Klancar, Igor Skrjanc, Drago Matko, and Bostjan Potocnik, "Time Optimal Path Planning Considering Acceleration Limits." *Robotics and Autonomous Systems*, Volume 45, Issues 3-4, 2003, pp. 199-210.

[3] Hoffmann, Gabriel M., Claire J. Tomlin, Michael Montemerlo, and Sebastian Thrun. "Autonomous Automobile Trajectory Tracking for Off-Road Driving: Controller Design, Experimental Validation and Racing." *American Control Conference*, 2007, pp. 2296-2301.

See Also

Blocks

Lateral Controller Stanley | Longitudinal Controller Stanley | Path Smoother Spline | Vehicle Body 3DOF | Velocity Profiler

Objects

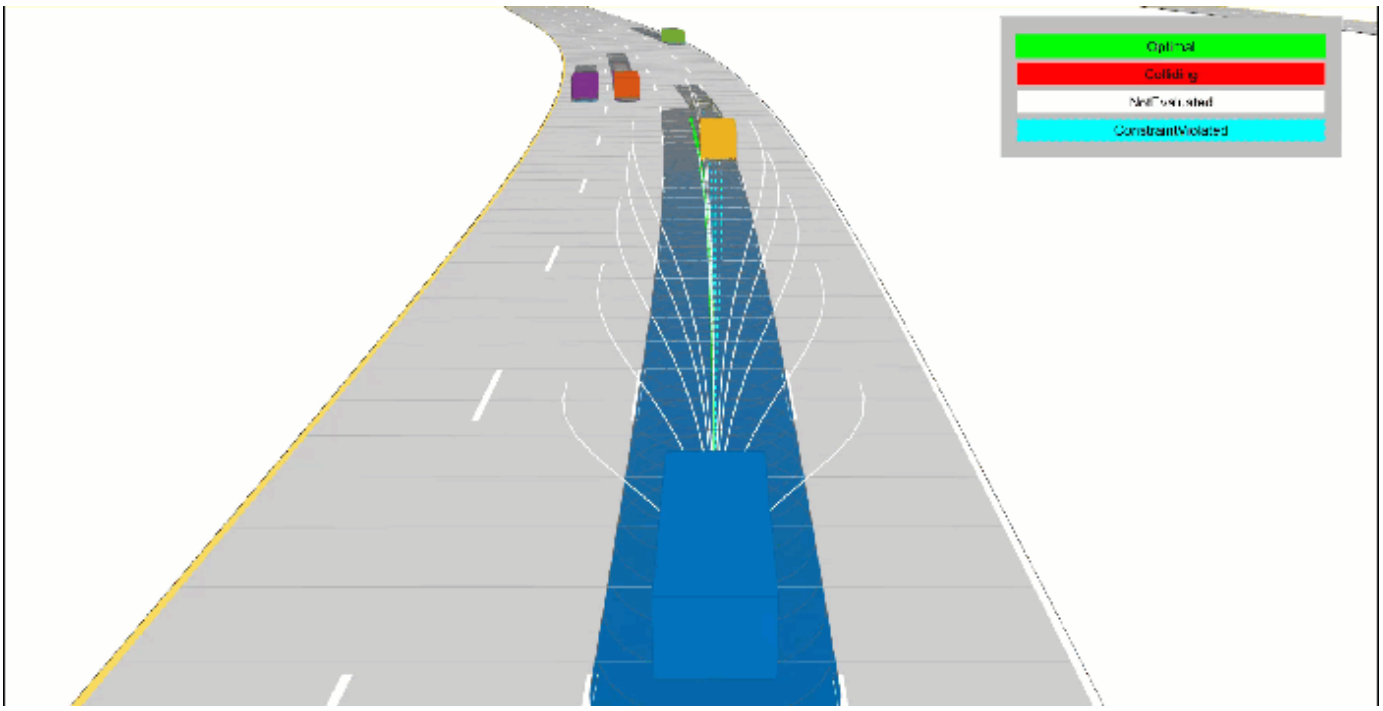
`pathPlannerRRT` | `vehicleCostmap` | `vehicleDimensions`

More About

- “Automated Parking Valet” on page 7-465
- “Automated Parking Valet with ROS in Simulink” (ROS Toolbox)
- “Automated Parking Valet with ROS 2 in Simulink” (ROS Toolbox)
- “Lateral Control Tutorial” on page 7-589
- “Code Generation for Path Planning and Vehicle Control” on page 7-515

Highway Trajectory Planning Using Frenet Reference Path

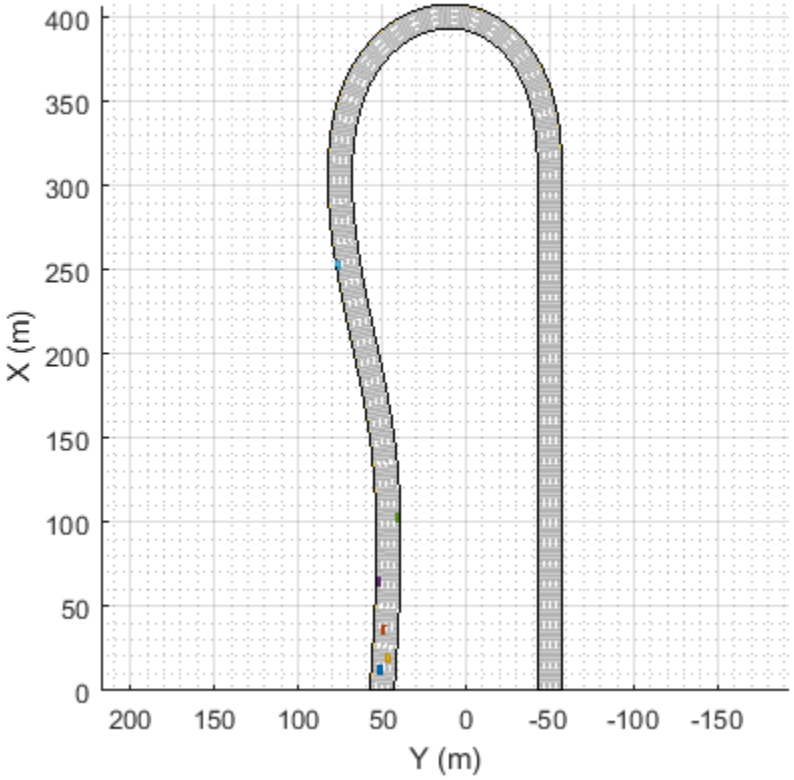
This example demonstrates how to plan a local trajectory in a highway driving scenario. This example uses a reference path and dynamic list of obstacles to generate alternative trajectories for an ego vehicle. The ego vehicle navigates through traffic defined in a provided driving scenario from a `drivingScenario` object. The vehicle alternates between adaptive cruise control, lane changing, and vehicle following maneuvers based on cost, feasibility, and collision-free motion.



Load Driving Scenario

Begin by loading the provided `drivingScenario` object, which defines the vehicle and road properties in the current workspace. This scenario was generated using the Driving Scenario Designer app and exported to a MATLAB® function, `drivingScenarioTrafficExample`. For more information, see “Create Driving Scenario Variations Programmatically” on page 5-107.

```
scenario = drivingScenarioTrafficExample;
% Default car properties
carLen   = 4.7; % in meters
carWidth = 1.8; % in meters
rearAxleRatio = .25;

% Define road dimensions
laneWidth   = carWidth*2; % in meters

plot(scenario);
```

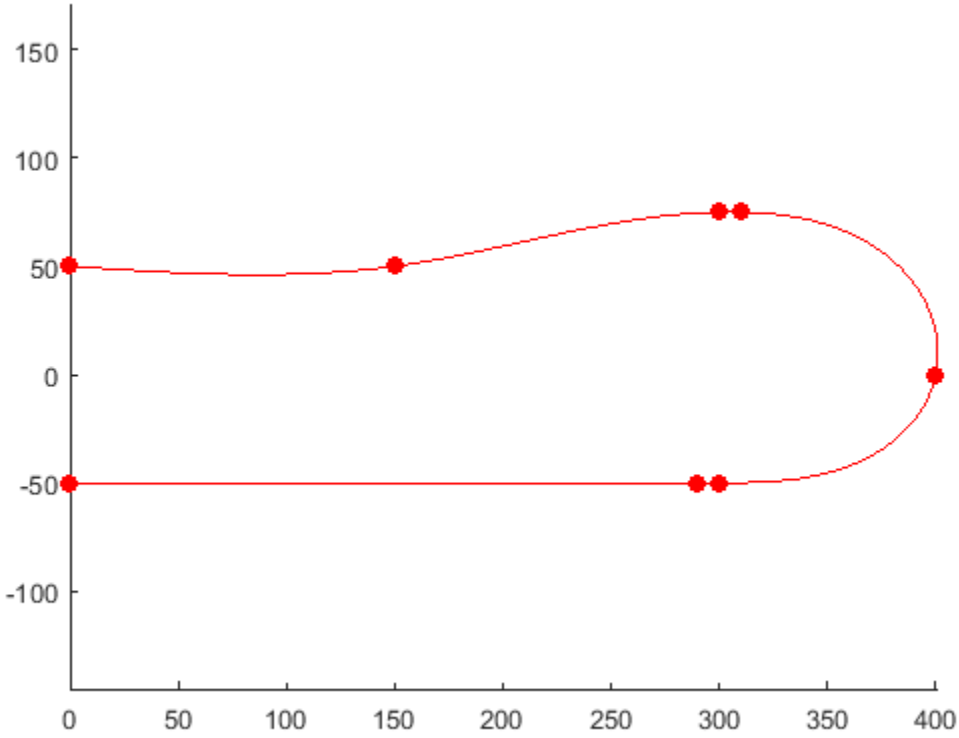


Construct Reference Path

All of the local planning in this example is performed with respect to a reference path, represented by a `referencePathFrenet` (Navigation Toolbox) object. This object can return the state of the curve at given lengths along the path, find the closest point along the path to some global *xy*-location, and facilitates the coordinate transformations between global and Frenet reference frames.

For this example, the reference path is treated as the center of a four-lane highway, and the waypoints match the road defined in the provided `drivingScenario` object.

```
waypoints = [0 50; 150 50; 300 75; 310 75; 400 0; 300 -50; 290 -50; 0 -50]; % in meters
refPath = referencePathFrenet(waypoints);
ax = show(refPath);
axis(ax,'equal');
```



Construct Trajectory Generator

For a local planner, the goal is typically to sample a variety of possible motions that move towards a final objective while satisfying the current kinematic and dynamic conditions. The `trajectoryGeneratorFrenet` (Navigation Toolbox) object accomplishes this by connecting the initial state with a set of terminal states using 4th- or 5th-order polynomial trajectories. Initial and terminal states are defined in the Frenet coordinate system, and each polynomial solution satisfies the lateral and longitudinal position, velocity, and acceleration boundary conditions while minimizing jerk.

Terminal states are often calculated using custom behaviors. These behaviors leverage information found in the local environment, such as the lane information, speed limit, and current or future predictions of actors in the ego vehicle's vicinity.

Construct a `trajectoryGeneratorFrenet` object using the reference path

```
connector = trajectoryGeneratorFrenet(refPath);
```

Construct Dynamic Collision Checker

The `dynamicCapsuleList` (Navigation Toolbox) object is a data structure that represents the state of a dynamic environment over a discrete set of timesteps. This environment can then be used to efficiently validate multiple potential trajectories for the ego vehicle. Each object in the scene is represented by:

- Unique integer-valued identifier

- Properties for a capsule geometry used for efficient collision checking
- Sequence of SE2 states, where each state represents a discrete snapshot in time.

In this example, the trajectories generated by the `trajectoryGeneratorFrenet` object occur over some span of time, known as the time horizon. To ensure that collision checking covers all possible trajectories, the `dynamicCapsuleList` object should contain predicted trajectories of all actors spanning the maximum expected time horizon.

```
capList = dynamicCapsuleList;
```

Create a geometry structure for the ego vehicle with the given parameters.

```
egoID = 1;
[egoID, egoGeom] = egoGeometry(capList,egoID);

egoGeom.Geometry.Length = carLen; % in meters
egoGeom.Geometry.Radius = carWidth/2; % in meters
egoGeom.Geometry.FixedTransform(1,end) = -carLen*rearAxleRatio; % in meters
```

Add the ego vehicle to the dynamic capsule list.

```
updateEgoGeometry(capList,egoID,egoGeom);
```

Add the `drivingScenario` actors to the `dynamicCapsuleList` object. The geometry is set here, and the states are defined during the planning loop. You can see that the `dynamicCapsuleList` now contains one ego vehicle and five obstacles.

```
actorID = (1:5)';
actorGeom = repelem(egoGeom,5,1);
updateObstacleGeometry(capList,actorID,actorGeom)
```

```
ans =
  dynamicCapsuleList with properties:

     MaxNumSteps: 31
          EgoIDs: 1
     ObstacleIDs: [5x1 double]
    NumObstacles: 5
         NumEgos: 1
```

Planning Adaptive Routes Through Traffic

The planning utilities support a local planning strategy that samples a set of local trajectories based on the current and foreseen state of the environment before choosing the most optimal trajectory. The simulation loop has been organized into the following sections:

1. Advance the Ground Truth Scenario on page 7-0
2. Generate Terminal States on page 7-0
3. Evaluate Cost of Terminal States on page 7-0
4. Generate Trajectories and Check for Kinematic Feasibility on page 7-0
5. Check Trajectories for Collision and Select the Optimal Trajectory on page 7-0
6. Display the Sampled Trajectories and Animate Scene on page 7-0

Click the titles of each section to navigate to the relevant code in the simulation loop.

Advance Ground Truth Scenario on page 7-0

When planning in a dynamic environment, it is often necessary to estimate the state of the environment or predict its state in the near future. For simplicity, this example uses the `drivingScenario` as a ground truth source of trajectories for each actor over the planning horizon. To test a custom prediction algorithm, you can replace or modify the `exampleHelperRetrieveActorGroundTruth` function with custom code.

Generate Terminal States on page 7-0

A common goal in automated driving is to ensure that planned trajectories are not just feasible but also natural. Typical highway driving involves elements of lane keeping, maintaining the speed limit, changing lanes, adapting speed to traffic, and so on. Each custom behavior might require different environment information. This example demonstrates how to generate terminal states that implement three such behaviors: cruise control, lane changes, and follow lead vehicle.

Cruise control

The `exampleHelperBasicCruiseControl` function generates terminal states that carry out a cruise control behavior. This function uses the ego vehicle's lateral velocity and a time horizon to predict the ego vehicle's expected lane N-seconds in the future. The lane-center is calculated and becomes the terminal state's lateral deviation, and the lateral velocity and acceleration are set to zero.

For longitudinal boundary conditions, the terminal velocity is set to the road speed limit and the terminal acceleration is set to zero. The longitudinal position is unrestricted, which is specified as `NaN`. By dropping the longitude constraint, `trajectoryGeneratorFrenet` can use a lower 4th-order polynomial to solve the longitudinal boundary-value problem, resulting in a trajectory that smoothly matches the road speed over the given time horizon:

$$cruiseControlState = \left[\text{NaN}\ \dot{s}_{des}\ 0\ l_{expLane}\ 0\ 0\right]$$

Lane change

The `exampleHelperBasicLaneChange` function generates terminal states that transition the vehicle from the current lane to either adjacent lane. The function first determines the ego vehicle's current lane, and then checks whether the left and right lanes exist. For each existing lane, the terminal state is defined in the same manner as the cruise control behavior, with the exception that the terminal velocity is set to the current velocity rather than the road's speed limit:

$$laneChangeState = [\text{NaN}\ \dot{s}_{cur}\ \ 0\ l_{desLane}\ 0\ 0]$$

Follow lead vehicle

The `exampleHelperBasicLeadVehicleFollow` function generates terminal states that attempt to trail a vehicle found ahead of the ego vehicle. The function first determines the ego vehicle's current lane. For each provided `timeHorizon`, the function predicts the future state of each actor, finds all actors that occupy the same lane as the ego vehicle, and determines which is the closest *lead* vehicle (if no lead vehicles are found, the function does not return anything).

The ego vehicle's terminal state is calculated by taking the lead vehicle's position and velocity and reducing the terminal longitudinal position by some safety distance:

$$closestLeadVehicle\text{State} = \left[s_{lead}\ \dot{s}_{lead}\ 0\ l_{lead}\ \dot{l}_{lead}\ 0\right]$$

$$followState = \left[\left(s_{lead} - d_{safety}\right)\ \dot{s}_{lead}\ 0\ l_{lead}\ \dot{l}_{lead}\ 0\right]$$

Evaluate Cost of Terminal States on page 7-0

After the terminal states have been generated, their cost can be evaluated. Trajectory evaluation and the ways to prioritize potential solutions is highly subjective. For the sake of simplicity, the `exampleHelperEvaluateTSCost` function defines cost as the combination of three weighted sums.

- **Lateral Deviation Cost (C_{latDev})** — A positive weight that penalizes states that deviate from the center of a lane.

  $$C_{latDev} = w_{\Delta L}*\Delta \mathrm{L}$$

  $$\Delta L = \mathrm{argmin}_i\left(\left|L_{\mathrm{termState}} - L_{\mathrm{lane}_i}\right|\right)$$

- **Time Cost (C_{time})** — A negative weight that prioritizes motions that occur over a longer interval, resulting in smoother trajectories.

  $$C_{time} = w_{\Delta t}*\Delta t$$

- **Terminal Velocity Cost (C_{speed})** — A positive weight that prioritizes motions that maintain the speed limit, resulting in less dynamic maneuvers.

  $$C_{speed} = w_{\Delta v}*\Delta v$$

Generate Trajectories and Check for Kinematic Feasibility on page 7-0

In addition to having terminal states with minimal cost, an optimal trajectory must often satisfy additional constraints related to kinematic feasibility and ride comfort. Trajectory constraints are one way of enforcing a desired ride quality, but they do so at the expense of a reduced driving envelope.

In this example, the `exampleHelperEvaluateTrajectory` function verifies that each trajectory satisfies the following constraints:

- **`MaxAcceleration`**: The absolute acceleration throughout the trajectory must be below a specified value. Smaller values reduce driving aggressiveness and eliminate kinematically infeasible trajectories. This restriction may eliminate maneuvers that could otherwise be performed by the vehicle.
- **`MaxCurvature`**: The minimum turning radius that is allowed throughout a trajectory. As with `MaxAcceleration`, reducing this value results in a smoother driving experience but may eliminate otherwise feasible trajectories.
- **`MinVelocity`**: This example constrains the ego vehicle to forward-only motion by setting a minimum velocity. This restriction is desired in highway driving scenarios and eliminates trajectories that fit overconstrained or poorly conditioned boundary values.

Check Trajectories for Collision and Select Optimal Trajectory on page 7-0

The final step in the planning process is choosing the best trajectory that also results in a collision-free path. Collision checking is often deferred until the end because it is an expensive operation, so by evaluating cost and analyzing constraints first, invalid trajectories can be removed from consideration. Remaining trajectories can then be checked for collision in optimal order until a collision free path has been found or all trajectories have been evaluated.

Define Simulator and Planning Parameters

This section defines the properties required to run the simulator and parameters that are used by the planner and behavior utilities. Properties such as `scenario.SampleTime and connector.TimeResolution` are synced so that states in the ground truth actor trajectories and planned ego trajectories occur at the same timesteps. Similarly, `replanRate`, `timeHorizons`, and `maxHorizon` are chosen such that they are integer multiples of the simulation rate.

As mentioned in the previous section, weights and constraints are selected to promote smooth driving trajectories while adhering to the rules of the road.

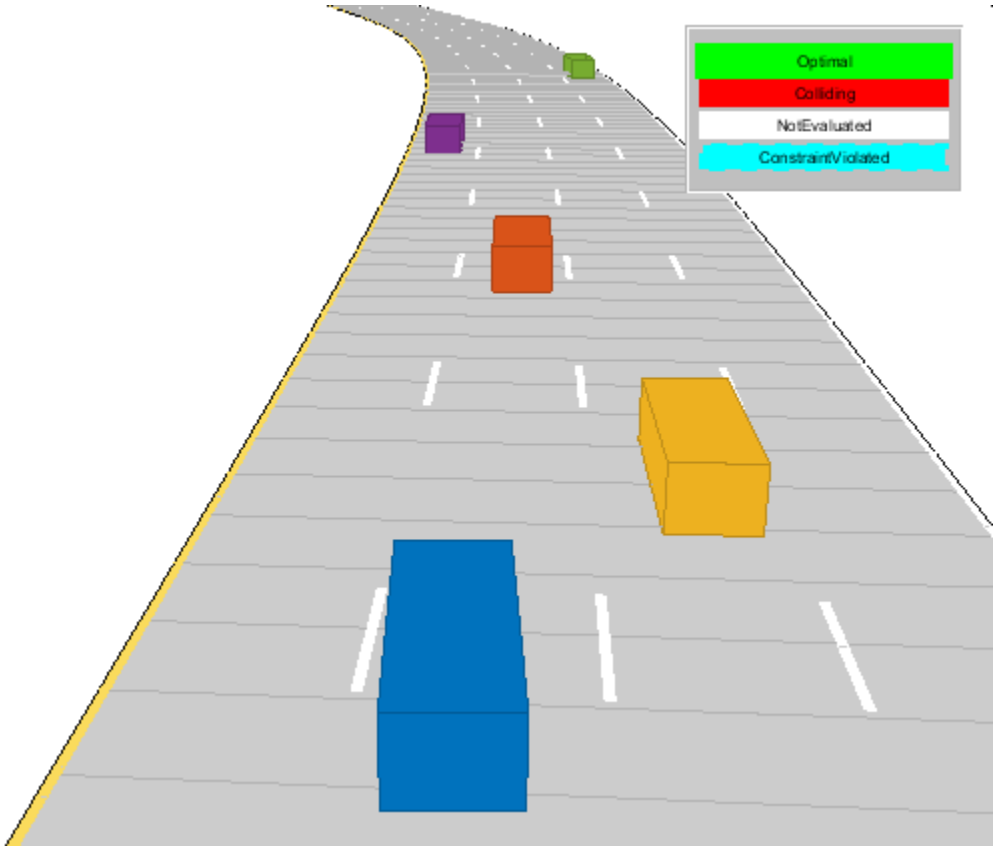
Lastly, define the `speedLimit` and `safetyGap` parameters, which are used to generate terminal states for the planner.

```
% Synchronize the simulator's update rate to match the trajectory generator's
% discretization interval.
scenario.SampleTime = connector.TimeResolution; % in seconds

% Define planning parameters.
replanRate = 10; % Hz

% Define the time intervals between current and planned states.
timeHorizons = 1:3; % in seconds
maxHorizon = max(timeHorizons); % in seconds

% Define cost parameters.
latDevWeight    =  1;
timeWeight      = -1;
speedWeight     =  1;

% Reject trajectories that violate the following constraints.
maxAcceleration =  15; % in meters/second^2
maxCurvature    =   1; % 1/meters, or radians/meter
minVelocity     =   0; % in meters/second

% Desired velocity setpoint, used by the cruise control behavior and when
% evaluating the cost of trajectories.
speedLimit = 11; % in meters/second

% Minimum distance the planner should target for following behavior.
safetyGap = 10; % in meters
```

Initialize Simulator

Initialize the simulator and create a `chasePlot` viewer.

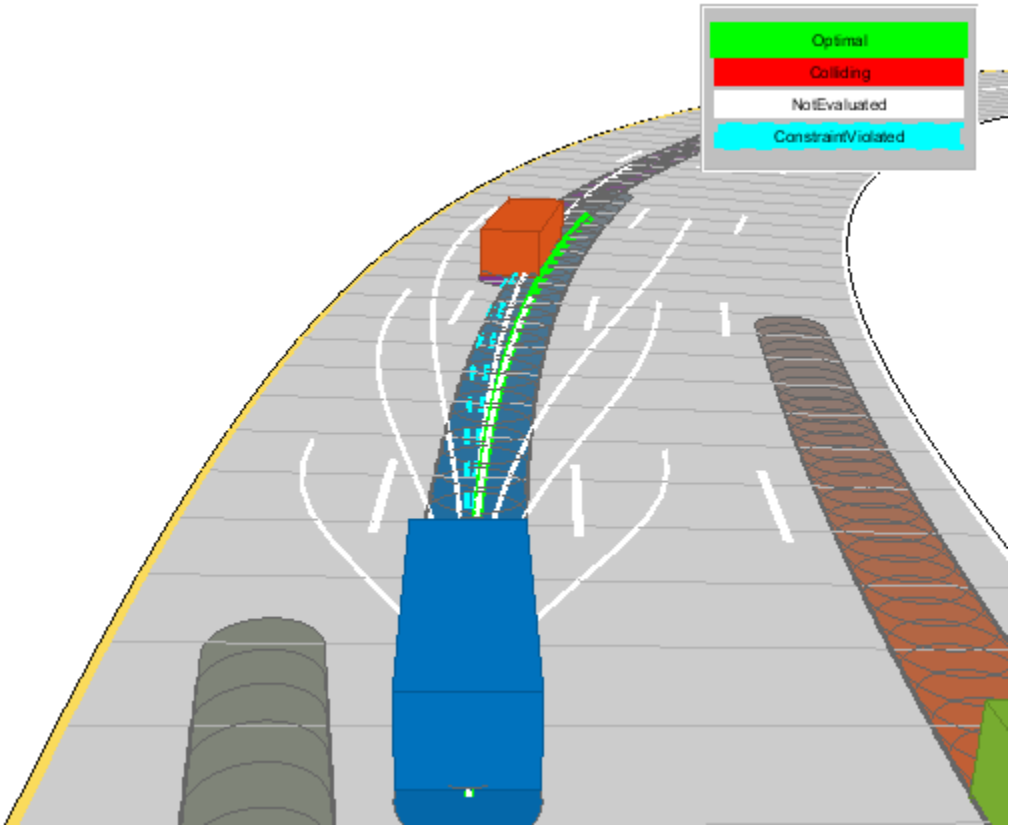
```
[scenarioViewer,futureTrajectory,actorID,actorPoses,egoID,egoPoses,stepPerUpdate,egoState,isRunni
    exampleHelperInitializeSimulator(scenario,capList,refPath,laneWidth,replanRate,carLen);
```



Run Driving Simulation

```
tic
while isRunning
    % Retrieve the current state of actor vehicles and their trajectory over
    % the planning horizon.
    [curActorState,futureTrajectory,isRunning] = ...
        exampleHelperRetrieveActorGroundTruth(scenario,futureTrajectory,replanRate,maxHorizon);

    % Generate cruise control states.
    [termStatesCC,timesCC] = exampleHelperBasicCruiseControl(...
        refPath,laneWidth,egoState,speedLimit,timeHorizons);

    % Generate lane change states.
    [termStatesLC,timesLC] = exampleHelperBasicLaneChange(...
        refPath,laneWidth,egoState,timeHorizons);

    % Generate vehicle following states.
    [termStatesF,timesF] = exampleHelperBasicLeadVehicleFollow(...
        refPath,laneWidth,safetyGap,egoState,curActorState,timeHorizons);

    % Combine the terminal states and times.
    allTS = [termStatesCC; termStatesLC; termStatesF];
    allDT = [timesCC; timesLC; timesF];
    numTS = [numel(timesCC); numel(timesLC); numel(timesF)];

    % Evaluate cost of all terminal states.
```

```
costTS = exampleHelperEvaluateTSCost(allTS,allDT,laneWidth,speedLimit,...
    speedWeight, latDevWeight, timeWeight);

% Generate trajectories.
egoFrenetState = global2frenet(refPath,egoState);
[frenetTraj,globalTraj] = connect(connector,egoFrenetState,allTS,allDT);

% Eliminate trajectories that violate constraints.
isValid = exampleHelperEvaluateTrajectory(globalTraj,maxAcceleration,maxCurvature,minVelocity

% Update the collision checker with the predicted trajectories
% of all actors in the scene.
for i = 1:numel(actorPoses)
    actorPoses(i).States = futureTrajectory(i).Trajectory(:,1:3);
end
updateObstaclePose(capList,actorID,actorPoses);

% Determine evaluation order.
[cost, idx] = sort(costTS);
optimalTrajectory = [];

trajectoryEvaluation = nan(numel(isValid),1);

% Check each trajectory for collisions starting with least cost.
for i = 1:numel(idx)
    if isValid(idx(i))
        % Update capsule list with the ego object's candidate trajectory.
        egoPoses.States = globalTraj(idx(i)).Trajectory(:,1:3);
        updateEgoPose(capList,egoID,egoPoses);

        % Check for collisions.
        isColliding = checkCollision(capList);

        if all(~isColliding)
            % If no collisions are found, this is the optimal.
            % trajectory.
            trajectoryEvaluation(idx(i)) = 1;
            optimalTrajectory = globalTraj(idx(i)).Trajectory;
            break;
        else
            trajectoryEvaluation(idx(i)) = 0;
        end
    end
end

% Display the sampled trajectories.
lineHandles = exampleHelperVisualizeScene(lineHandles,globalTraj,isValid,trajectoryEvaluatio
```




```
hold on;
show(capList,'TimeStep',1:capList.MaxNumSteps,'FastUpdate',1);
hold off;

if isempty(optimalTrajectory)
    % All trajectories either violated a constraint or resulted in collision.
    %
    %   If planning failed immediately, revisit the simulator, planner,
    %   and behavior properties.
    %
    %   If the planner failed midway through a simulation, additional
    %   behaviors can be introduced to handle more complicated planning conditions.
    error('No valid trajectory has been found.');
else
    % Visualize the scene between replanning.
    for i = (2+(0:(stepPerUpdate-1)))
        % Approximate realtime visualization.
        dt = toc;
        if scenario.SampleTime-dt > 0
            pause(scenario.SampleTime-dt);
        end

        egoState = optimalTrajectory(i,:);
        scenarioViewer.Actors(1).Position(1:2) = egoState(1:2);
        scenarioViewer.Actors(1).Velocity(1:2) = [cos(egoState(3)) sin(egoState(3))]*egoState
        scenarioViewer.Actors(1).Yaw = egoState(3)*180/pi;
```

```
            scenarioViewer.Actors(1).AngularVelocity(3) = egoState(4)*egoState(5);

            % Update capsule visualization.
            hold on;
            show(capList,'TimeStep',i:capList.MaxNumSteps,'FastUpdate',1);
            hold off;

            % Update driving scenario.
            advance(scenarioViewer);
            ax = gca;
            ax.ZLim = [-100 100];
            tic;
        end
    end
end
```

Planner Customizations and Additional Considerations

Custom solutions often involve many tunable parameters, each capable of changing the final behavior in ways that are difficult to predict. This section highlights some of the feature-specific properties and their effect on the above planner. Then, suggestions provide ways to tune or augment the custom logic.

Dynamic Capsule List

As mentioned previously, the `dynamicCapsuleList` object acts as a temporal database, which caches predicted trajectories of obstacles. You can perform collision checking with one or more ego bodies over some span of time. The `MaxNumSteps` property determines the total number of time-steps that are checked by the object. In the above simulation loop, the property was set to 31. This value means the planner checks the entire 1-3 second span of any trajectories (sampled at every 0.1 second). Now, increase the maximum value in `timeHorizons`:

```
timeHorizons   = 1:5; % in seconds
maxTimeHorizon = max(timeHorizons); % in seconds
```

There are now two options:

1. The `MaxNumSteps` property is left unchanged.
2. The `MaxNumSteps` property is updated to accommodate the new max timespan.

If the property is left unchanged, then the capsule list only validates the first 3 seconds of any trajectory, which may be preferable if computational efficiency is paramount or the prediction certainty drops off quickly.

Alternatively, one may be working with ground truth data (as is shown above), or the future state of the environment is well known (e.g. a fully automated environment with centralized control). Since this example uses ground truth data for the actors, update the property.

```
capList.MaxNumSteps = 1+floor(maxTimeHorizon/scenario.SampleTime);
```

Another, indirectly tunable, property of the list is the capsule geometry. The geometry of the ego vehicle or actors can be inflated by increasing the `Radius`, and buffer regions can be added to vehicles by modifying the `Length` and `FixedTransform` properties.

Inflate the ego vehicle's entire footprint by increasing the radius.

```
egoGeom.Geometry.Radius = laneWidth/2; % in meters
updateEgoGeometry(capList,egoID,egoGeom);
```

Add a front and rear buffer region to all actors.

```
actorGeom(1).Geometry.Length = carLen*1.5; % in meters
actorGeom(1).Geometry.FixedTransform(1,end) = -actorGeom(1).Geometry.Length*rearAxleRatio; % in m
actorGeom = repmat(actorGeom(1),5,1);
updateObstacleGeometry(capList,actorID,actorGeom);
```

Rerun Simulation With Updated Properties

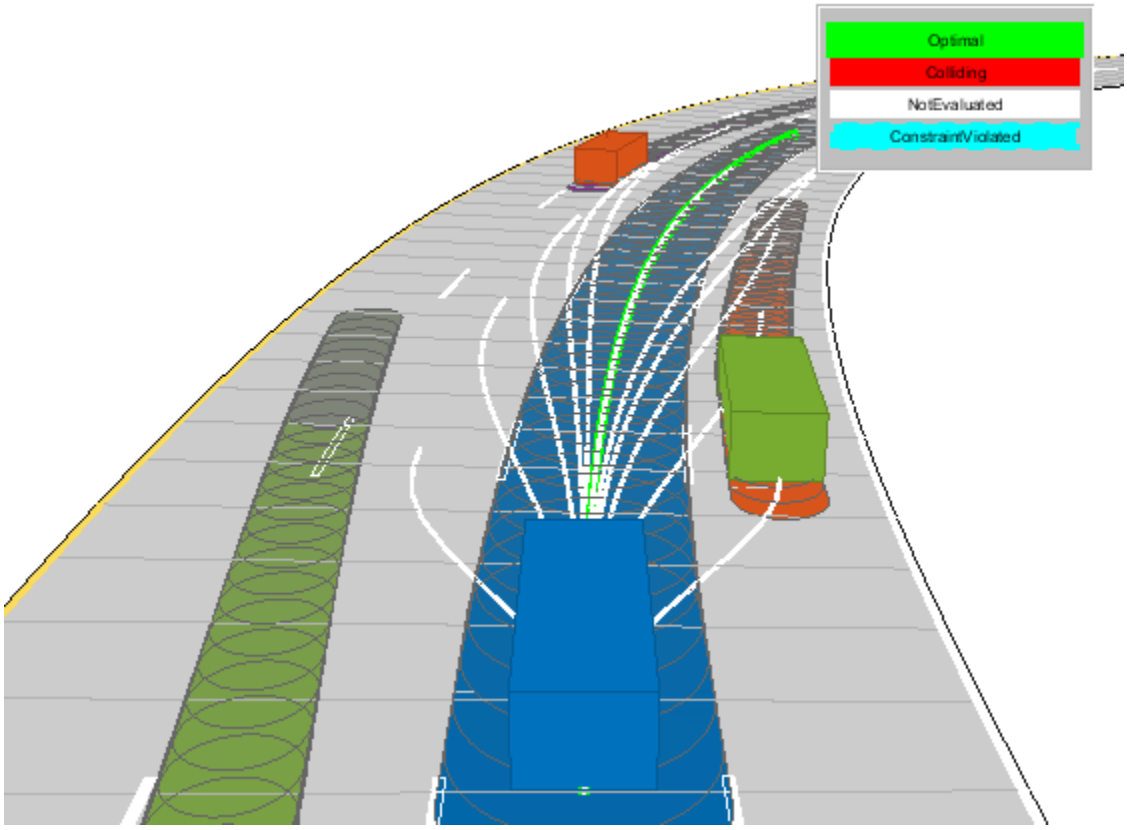
Rerun the simulation. The resulting simulation has a few interesting developments:

- The longer five-second time horizon results in a much smoother driving experience. The planner still prioritizes the longer trajectories due to the negative `timeWeight`.
- The updated `MaxNumSteps` property has enabled collision checking over the full trajectory. When paired with the longer planning horizon, the planner identifies and discards the previously optimal left-lane change and returns to the original lane.
- The inflated capsules find a collision earlier and reject a trajectory, which results in more conservative driving behavior. One potential drawback to this is a reduced planning envelope, which runs the risk of the planner not being able to find a valid trajectory.

```
% Initialize the simulator and create a chasePlot viewer.
[scenarioViewer,futureTrajectory,actorID,actorPoses,egoID,egoPoses,stepPerUpdate,egoState,isRunni
    exampleHelperInitializeSimulator(scenario,capList,refPath,laneWidth,replanRate,carLen);
tic;
while isRunning
    % Retrieve the current state of actor vehicles and their trajectory over
    % the planning horizon.
    [curActorState,futureTrajectory,isRunning] = exampleHelperRetrieveActorGroundTruth(...
        scenario,futureTrajectory,replanRate,maxHorizon);

    % Generate cruise control states.
    [termStatesCC,timesCC] = exampleHelperBasicCruiseControl(...
        refPath,laneWidth,egoState,speedLimit,timeHorizons);

    % Generate lane change states.
    [termStatesLC,timesLC] = exampleHelperBasicLaneChange(...
        refPath,laneWidth,egoState,timeHorizons);

    % Generate vehicle following states.
    [termStatesF,timesF] = exampleHelperBasicLeadVehicleFollow(...
        refPath,laneWidth,safetyGap,egoState,curActorState,timeHorizons);

    % Combine the terminal states and times.
    allTS = [termStatesCC; termStatesLC; termStatesF];
    allDT = [timesCC; timesLC; timesF];
    numTS = [numel(timesCC); numel(timesLC); numel(timesF)];

    % Evaluate cost of all terminal states.
    costTS = exampleHelperEvaluateTSCost(allTS,allDT,laneWidth,speedLimit, ...
        speedWeight,latDevWeight,timeWeight);

    % Generate trajectories.
    egoFrenetState = global2frenet(refPath,egoState);
    [frenetTraj,globalTraj] = connect(connector,egoFrenetState,allTS,allDT);
```

```
% Eliminate trajectories that violate constraints.
isValid = exampleHelperEvaluateTrajectory(...
    globalTraj, maxAcceleration, maxCurvature, minVelocity);

% Update the collision checker with the predicted trajectories
% of all actors in the scene.
for i = 1:numel(actorPoses)
    actorPoses(i).States = futureTrajectory(i).Trajectory(:,1:3);
end
updateObstaclePose(capList, actorID, actorPoses);

% Determine evaluation order.
[cost, idx] = sort(costTS);
optimalTrajectory = [];

trajectoryEvaluation = nan(numel(isValid),1);

% Check each trajectory for collisions starting with least cost.
for i = 1:numel(idx)
    if isValid(idx(i))
        % Update capsule list with the ego object's candidate trajectory.
        egoPoses.States = globalTraj(idx(i)).Trajectory(:,1:3);
        updateEgoPose(capList, egoID, egoPoses);

        % Check for collisions.
        isColliding = checkCollision(capList);

        if all(~isColliding)
            % If no collisions are found, this is the optimal
            % trajectory.
            trajectoryEvaluation(idx(i)) = 1;
            optimalTrajectory = globalTraj(idx(i)).Trajectory;
            break;
        else
            trajectoryEvaluation(idx(i)) = 0;
        end
    end
end

% Display the sampled trajectories.
lineHandles = exampleHelperVisualizeScene(lineHandles, globalTraj, isValid, trajectoryEvaluat

if isempty(optimalTrajectory)
    % All trajectories either violated a constraint or resulted in collision.
    %
    %   If planning failed immediately, revisit the simulator, planner,
    %   and behavior properties.
    %
    %   If the planner failed midway through a simulation, additional
    %   behaviors can be introduced to handle more complicated planning conditions.
    error('No valid trajectory has been found.');
else
    % Visualize the scene between replanning.
    for i = (2+(0:(stepPerUpdate-1)))
        % Approximate realtime visualization.
        dt = toc;
        if scenario.SampleTime-dt > 0
```

```
                pause(scenario.SampleTime-dt);
            end

            egoState = optimalTrajectory(i,:);
            scenarioViewer.Actors(1).Position(1:2) = egoState(1:2);
            scenarioViewer.Actors(1).Velocity(1:2) = [cos(egoState(3)) sin(egoState(3))]*egoState
            scenarioViewer.Actors(1).Yaw = egoState(3)*180/pi;
            scenarioViewer.Actors(1).AngularVelocity(3) = egoState(4)*egoState(5);

            % Update capsule visualization.
            hold on;
            show(capList,'TimeStep',i:capList.MaxNumSteps,'FastUpdate',1);
            hold off;

            % Update driving scenario.
            advance(scenarioViewer);
            ax = gca;
            ax.ZLim = [-100 100];
            tic;
        end
    end
end
```



See Also

`drivingScenario` | `referencePathFrenet` | `trajectoryGeneratorFrenet`

More About

- “Optimal Trajectory Generation for Urban Driving” (Navigation Toolbox)
- “Highway Lane Change” on page 7-598

Code Generation for Path Planning and Vehicle Control

This example shows how to modify a Simulink® model of a path planning and vehicle control algorithm, generate C++ code, and verify the generated code using software-in-the-loop (SIL) simulation.

Introduction

Developing a path planning and vehicle control algorithm often involves designing and simulating an algorithm model in Simulink, implementing the algorithm in C++ code, and integrating the algorithm code into an external software environment for deployment into a vehicle. Automatically generating and verifying code from the algorithm model ensures functional equivalence between the simulation and implementation.

The “Automated Parking Valet in Simulink” on page 7-493 example showed how to design a path planning and vehicle control algorithm. This example shows how to modify the design for implementation in C++. This steps in this workflow are:

1. Partition the design into algorithm and test bench models.
2. Modify the algorithm model to support code generation.
3. Generate C++ code from the algorithm model.
4. Verify the behavior of the generated code using SIL simulation.

You can then integrate the generated code into an external software project for further testing in a vehicle.

Partition the Algorithm and Test Bench

The original model from the “Automated Parking Valet in Simulink” on page 7-493 example has already been partitioned into separate algorithm and test bench models.

- **Algorithm Model**: `AutomatedParkingValetAlgorithm` specifies the path planning and vehicle control functionality to be implemented in C++.
- **Test Bench Model**: `AutomatedParkingValetTestBench` specifies the stimulus and environment to test the algorithm model.

Simulate Test Bench Model

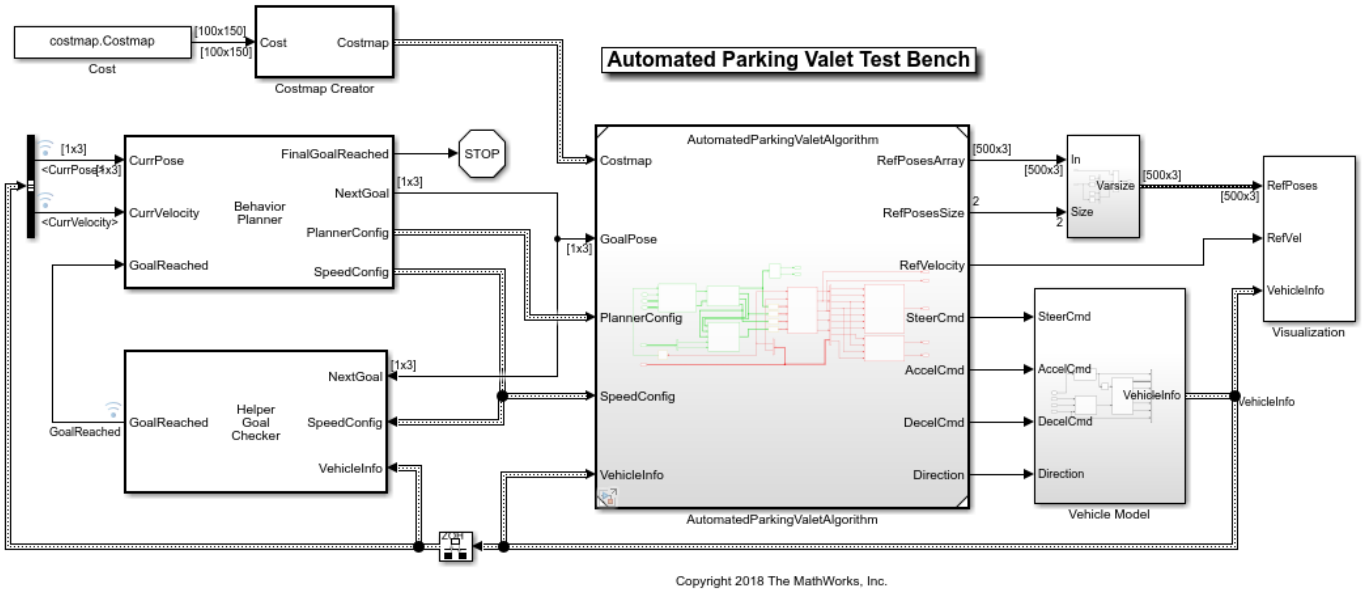
The `AutomatedParkingValetTestBench` model specifies the stimulus and environment to test the `AutomatedParkingValetAlgorithm` model. The main components of the `AutomatedParkingValetTestBench` include:

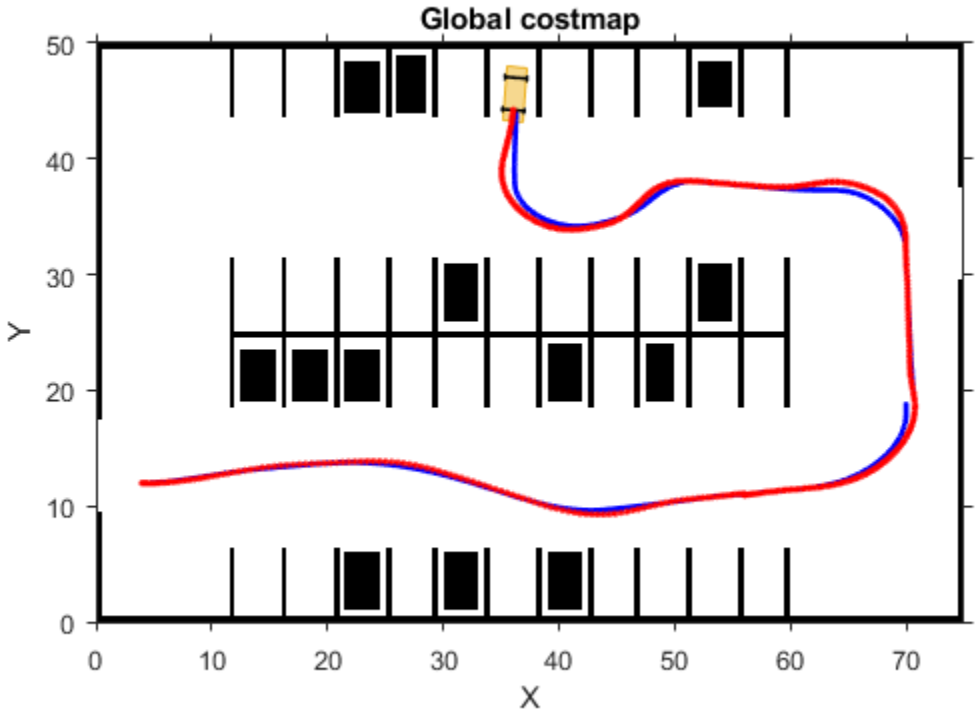
- **Algorithm Model Reference**: The `AutomatedParkingValetAlgorithm` model block is referenced by a Model block. The Model block and supports simulating the referenced model in different modes of simulation including normal and SIL modes. To learn more about the Model block, refer to “Reference Existing Models” (Simulink).
- **Costmap**: The Costmap Creator block creates the costmap of the environment and outputs it as a bus signal.
- **Behavior Planner**: The Behavior Planner block triggers a sequence of navigation tasks based on the global route plan by providing an intermediate goal and configuration.
- **Vehicle Model**: To demonstrate the performance of the algorithm, the parking valet controller is applied to the Vehicle Model block, which contains a Vehicle Body 3DOF block.

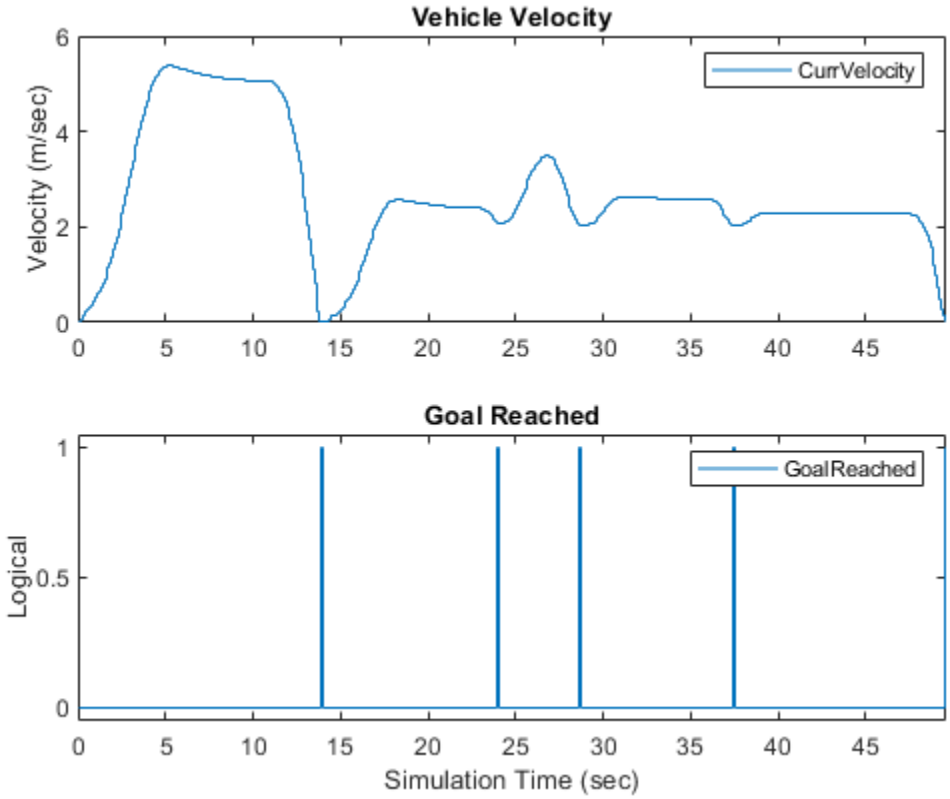
The `AutomatedParkingValetTestBench` model is also configured to log the pose(`CurrPose`) and longitudinal velocity (`CurrVelocity`) of the vehicle and the status of whether the goal from the behavioral planner was reached (`GoalReached`). These signals are logged to the workspace variable `logsout`.

Simulate the test bench model with the algorithm in normal mode.

```
open_system('AutomatedParkingValetTestBench')
set_param('AutomatedParkingValetTestBench/AutomatedParkingValetAlgorithm','SimulationMode','Norma
sim('AutomatedParkingValetTestBench')
helperPlotSimulationSignals(logsout)
```




Global costmap
50
40
30
20
10
0
Y
0
10
20
30
40
50
60
70
X



The first figure displays the path that the vehicle traversed from the parking lot input to the final parking space. The second figure plots the velocity and goal-reached signals. Notice that the vehicle velocity is smooth and continuous when transitioning between goals.

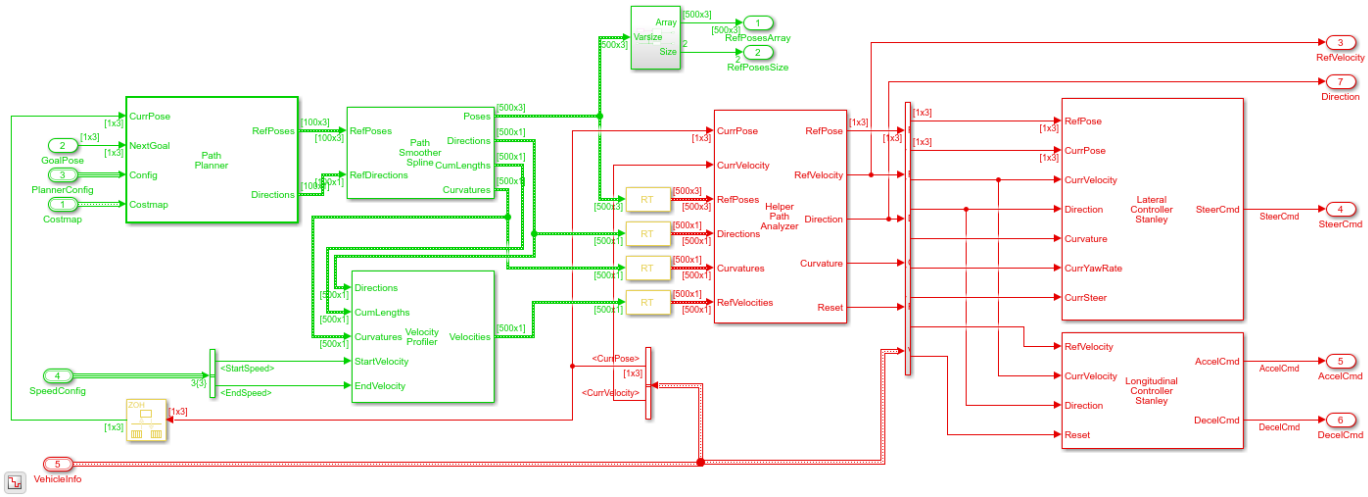
Modify Algorithm Model to Support Code Generation

The `AutomatedParkingValetAlgorithm` model specifies the functionality to be implemented in C ++. The main components of the `AutomatedParkingValetAlgorithm` model are:

- **Path Planner**: Plans a feasible path through the environment map using a `pathPlannerRRT` object.
- **Trajectory Generator**: Smooths the reference path by fitting splines and converts the smoothed path into a trajectory by generating a speed profile.
- **Vehicle Controller**: Controls the steering and velocity of the vehicle to follow the generated path and speed profile.

Open and update the algorithm model.

```
open_system('AutomatedParkingValetAlgorithm')
set_param('AutomatedParkingValetAlgorithm','SimulationCommand','Update');
```



The `AutomatedValetParking` model includes several modifications from the "Automated Parking Valet in Simulink" on page 7-493 example to support code generation. The most significant modifications are specifying fixed-size component interfaces and explicit rate transitions.

Variable-size component interfaces have been replaced with fixed-size interface to enable generating and verifying C++ code with SIL simulation.

- The variable-size `Poses` signal has been split into a fixed-size outport (`RefPosesArray`) with an additional outport specifying the size (`RefPosesSize`).
- The `costmapBus` bus associated with the `Costmap` input port contains only fixed-size elements, since the costmap does not change size in this example.

The `AutomatedValetParking` model contains multiple rates. The color of the blocks represents different sample times. Path planning and trajectory generation is performed at a 0.1s sample time and is colored green. Vehicle control is performed at a 0.05s sample time and is colored red. To learn more about displaying sample time colors, refer to "View Sample Time Information" (Simulink).

Explicit rate transition blocks have been inserted into the model to treat each rate as a separate task.

- A Rate Transition block has been inserted to the fixed-size `CurrPose` signal.
- A helper Varsize Rows Rate Transition block (named RT) has been inserted to variable-size signals that connect blocks of different rates.

Treating each rate as a specific task enables generating a C++ class with separate method entry points for each rate. Generating separate methods for each rate simplifies integration into multi-tasking software schedulers or operating systems in the vehicle. To learn more about treating rates as separate tasks, refer to "Modeling for Multitasking Execution" (Embedded Coder).

Configure and Generate Code from Algorithm Model

Configuring the `AutomatedParkingValetAlgorithm` model to generate code includes setting parameters to:

- Generate C++ code with entry points for each rate.
- Apply common optimizations.

- Generate a report to facilitate exploring the generated code.

Set and view model parameters to enable C++ code generation.

```
helperSetModelParametersForCodeGeneration('AutomatedParkingValetAlgorithm')
```

```
Set AutomatedParkingValetAlgorithm configuration parameters:

                Parameter                        Value
    ____________________________________    _____________    ____________________________________

    {'SystemTargetFile'                }    {'ert.tlc'    }    {'Code Generation>System target fi
    {'TargetLang'                      }    {'C++'        }    {'Code Generation>Language'
    {'SolverType'                      }    {'Fixed-step' }    {'Solver>Type'
    {'FixedStep'                       }    {'auto'       }    {'Solver>Fixed-step size (fundamen
    {'EnableMultiTasking'              }    {'on'         }    {'Solver>Treat each discrete rate a
    {'ProdLongLongMode'                }    {'on'         }    {'Hardware Implementation>Support
    {'BlockReduction'                  }    {'on'         }    {'Simulation Target>Block reductio
    {'MATLABDynamicMemAlloc'           }    {'on'         }    {'Simulation Target>Simulation Tar
    {'OptimizeBlockIOStorage'          }    {'on'         }    {'Simulation Target>Signal storage
    {'InlineInvariantSignals'          }    {'on'         }    {'Simulation Target>Inline invaria
    {'BuildConfiguration'              }    {'Faster Runs'}    {'Code Generation>Build configurat
    {'RTWVerbose'                      }    {'of'         }    {'Code Generation>Verbose build'
    {'CombineSignalStateStructs'       }    {'on'         }    {'Code Generation>Interface>Combine
    {'GenerateExternalIOAccessMethods'}    {'Method'     }    {'Code Generation>Interface>Externa
    {'SupportVariableSizeSignals'      }    {'on'         }    {'Code Generation>Interface>Suppor
    {'EfficientFloat2IntCast'          }    {'on'         }    {'Code Generation>Optimization>Remo
    {'ZeroExternalMemoryAtStartup'     }    {'off'        }    {'Code Generation>Optimization>Remo
    {'CustomSymbolStrGlobalVar'        }    {'$N$M'       }    {'Code Generation>Symbols>Global va
    {'CustomSymbolStrType'             }    {'$N$M_T'     }    {'Code Generation>Symbols>Global ty
    {'CustomSymbolStrField'            }    {'$N$M'       }    {'Code Generation>Symbols>Field nam
    {'CustomSymbolStrFcn'              }    {'APV_$N$M$F' }    {'Code Generation>Symbols>Subsyster
    {'CustomSymbolStrTmpVar'           }    {'$N$M'       }    {'Code Generation>Symbols>Local te
    {'CustomSymbolStrMacro'            }    {'$N$M'       }    {'Code Generation>Symbols>Constant
```

Generate code and the code generation report from the algorithm model.

```
rtwbuild('AutomatedParkingValetAlgorithm');
```

```
### Starting build procedure for: AutomatedParkingValetAlgorithm
### Successful completion of build procedure for: AutomatedParkingValetAlgorithm
```

Use the Code Generation Report to explore the generated code. To learn more about the Code Generation Report, refer to “Reports for Code Generation” (Simulink Coder). Use the Code Interface Report link in the Code Generation Report to explore these generated methods:

- `initialize`: Call once on initialization.
- `step0`: Call periodically every 0.05s to execute trajectory generation and vehicle control.
- `step1`: Call periodically every 0.1s seconds to execute path planning.
- `terminate`: Call once on termination.

Additional get and set methods for signal interface are declared in `AutomatedParkingValetAlgorithm.h` and defined in `AutomatedParkingValetAlgorithm.c`.

Verify Implementation with SIL Simulation

Software-in-the-loop (SIL) simulation provides early insight into the behavior of a deployed application. To learn more about SIL simulation, refer to “SIL and PIL Simulations” (Embedded Coder).

SIL simulation enables you to: * Verify that the compiled generated code on the host is functionally equivalent to the normal mode. * Log execution times of generated code on the host computer. These times can be an early indicator of performance of the generated code. For accurate execution time measurements, profile the generated code when it is integrated into the external environment or when using with processor-in-the-loop(PIL) simulation. To learn more about SIL profiling, refer to “Code Execution Profiling with SIL and PIL” (Embedded Coder).

Configure algorithm and test bench model parameters to support SIL simulation and log execution profiling information.

```
helperSetModelParametersForSIL('AutomatedParkingValetAlgorithm');
helperSetModelParametersForSIL('AutomatedParkingValetTestBench');
```

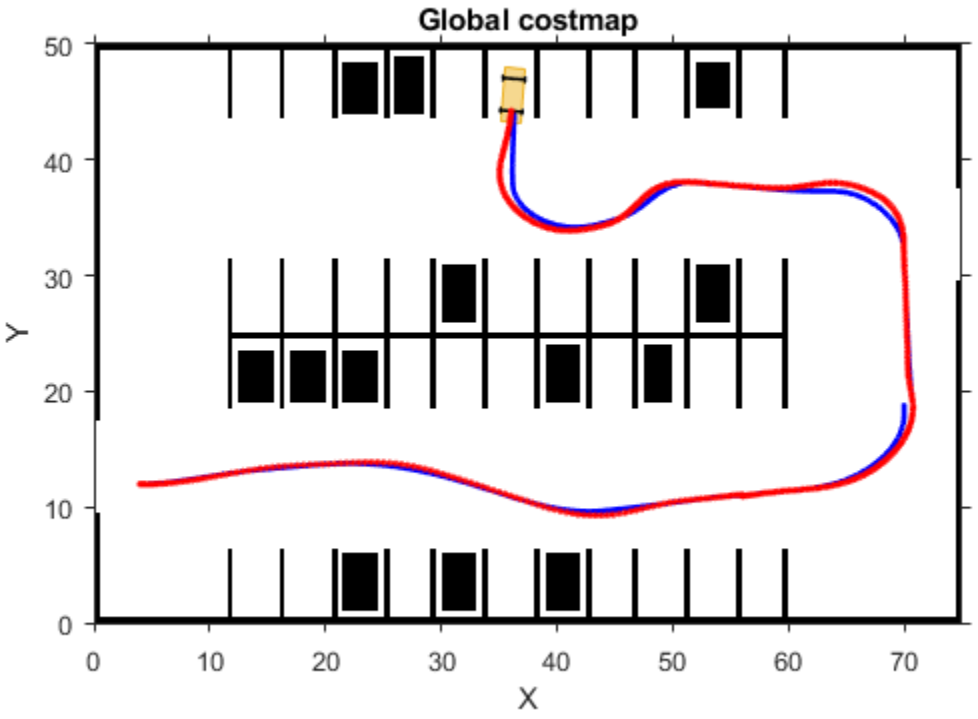
```
Set AutomatedParkingValetAlgorithm configuration parameters:

              Parameter                      Value                                Descripti
    ____________________________________    ____________________    ____________________________________

    {'SystemTargetFile'            }    {'ert.tlc'         }    {'Code Generation>System target
    {'TargetLang'                  }    {'C++'             }    {'Code Generation>Language'
    {'CodeExecutionProfiling'      }    {'on'              }    {'Code Generation>Verification>Me
    {'CodeProfilingSaveOptions'    }    {'AllData'         }    {'Code Generation>Verification>Sa
    {'CodeExecutionProfileVariable'}    {'executionProfile'}    {'Code Generation>Verification>Wo


Set AutomatedParkingValetTestBench configuration parameters:

              Parameter                      Value                                Descripti
    ____________________________________    ____________________    ____________________________________

    {'SystemTargetFile'            }    {'ert.tlc'         }    {'Code Generation>System target
    {'TargetLang'                  }    {'C++'             }    {'Code Generation>Language'
    {'CodeExecutionProfiling'      }    {'on'              }    {'Code Generation>Verification>Me
    {'CodeProfilingSaveOptions'    }    {'AllData'         }    {'Code Generation>Verification>Sa
    {'CodeExecutionProfileVariable'}    {'executionProfile'}    {'Code Generation>Verification>Wo
```

Simulate the test bench model with the algorithm in SIL mode and plot the results.

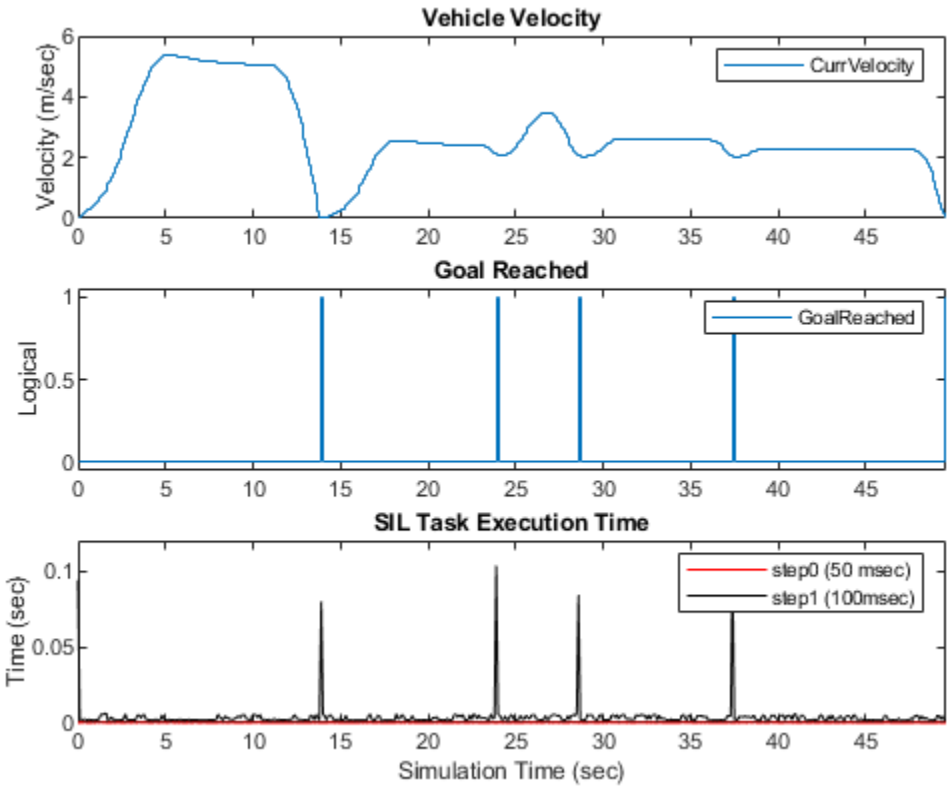
```
open_system('AutomatedParkingValetTestBench')
set_param('AutomatedParkingValetTestBench/AutomatedParkingValetAlgorithm','SimulationMode','Softw
sim('AutomatedParkingValetTestBench');
```

```
### Starting build procedure for: AutomatedParkingValetAlgorithm
### Generated code for 'AutomatedParkingValetAlgorithm' is up to date because no structural, para
### Successful completion of build procedure for: AutomatedParkingValetAlgorithm
### Preparing to start SIL simulation ...
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Updating code generation report with SIL files ...
```

```
### Starting SIL simulation for component: AutomatedParkingValetAlgorithm
### Stopping SIL simulation for component: AutomatedParkingValetAlgorithm
```




```
helperPlotSimulationSignals(logsout, executionProfile)
```



The execution time for the `step0` and `step1` methods are shown in the lower plot. The plots indicate that the maximum execution time is required at the lower rate (`step1`) after a goal pose is achieved. This lower rate is expected because it corresponds to the time when a new path is planned.

Conclusion

This example demonstrated a workflow to generate and verify C++ code for a path planner and vehicle control algorithm. Compiling and verifying the code with SIL simulation established confidence that the generated code is functionally correct before integrating into an external software environment. The workflow was demonstrated as an extension of the “Automated Parking Valet in Simulink” on page 7-493 example and is generally applicable to designing and implementing path planning applications.

See Also

More About

- “Automated Parking Valet in Simulink” on page 7-493

Use HERE HD Live Map Data to Verify Lane Configurations

This example shows how to read and visualize lane configurations for a recorded driving route from the HERE HD Live Map (HERE HDLM) service. This visualization can be used to verify the lane configurations detected by the perception system of an onboard sensor, such as a monocular camera.

In this example, you learn how to access the tiled layers from the HDLM service and identify relevant road-level and lane-level topology, geometry, and attributes.

To read the data, you use a `hereHDLMReader` object. Use of the HERE HD Live Map service requires valid HERE HDLM credentials. You need to enter into a separate agreement with HERE in order to gain access to the HDLM services and to get the required credentials (access_key_id and access_key_secret) for using the HERE Service.

Overview

High-definition (HD) maps refer to mapping services developed specifically for automated driving applications. The precise geometry of these maps (with up to 1 cm resolution near the equator) make them suitable for automated driving workflows beyond the route planning applications of traditional road maps. Such workflows include lane-level verification, localization, and path planning. This example shows you how to verify the performance of a lane detection system using lane-level information from HD mapping data.

The accuracy of HD mapping data enables its use as a source of ground truth data for verification of onboard sensor perception systems. This high accuracy enables faster and more accurate verification of existing deployed algorithms.

HERE HD Live Map (HERE HDLM) is a cloud-based HD map service developed by HERE Technologies to support highly automated driving. The data is composed of tiled mapping layers that provide access to accurate geometry and robust attributes of a road network. The layers are grouped into the following models:

- **Road Centerline Model:** Provides road topology (specified as nodes and links in a graph), shape geometry, and other road-level attributes.
- **HD Lane Model:** Provides lane topology (as lane groups and lane group connectors), highly accurate geometry, and lane-level attributes.
- **HD Localization Model:** Provides features to support vehicle localization strategies.

For an overview of HERE HDLM layers, see “HERE HD Live Map Layers” on page 4-15.

Cameras are used in automated driving to gather semantic information about the road topology around the vehicle. Lane boundary detection, lane type classification, and road sign detection algorithms form the core of such a camera processing pipeline. You can use the HERE HDLM service, along with a high-precision GPS mounted on the vehicle, to evaluate the accuracy of such algorithms and verify their performance.

In this example, you learn how to:

1. Read road and lane information from the HERE HDLM service for a recorded GPS sequence.
2. Apply a heuristic route matching approach to the recorded GPS data. Because GPS data is often imprecise, it is necessary to solve the problem of matching recorded geographic coordinates to a representation of the road network.

3 Identify environmental attributes relevant to the vehicle. Once a vehicle is successfully located within the context of the map, you can use road and lane attributes relevant to the vehicle to verify data recorded by the vehicle's onboard camera sensor.

Load and Display Camera and GPS Data

Start by loading data from the recorded drive. The recorded data in this example is from a driving dataset collected by the Udacity® Self-Driving Car team. This data includes a video captured by a front-facing monocular camera and vehicle positions and velocities logged by a GPS.

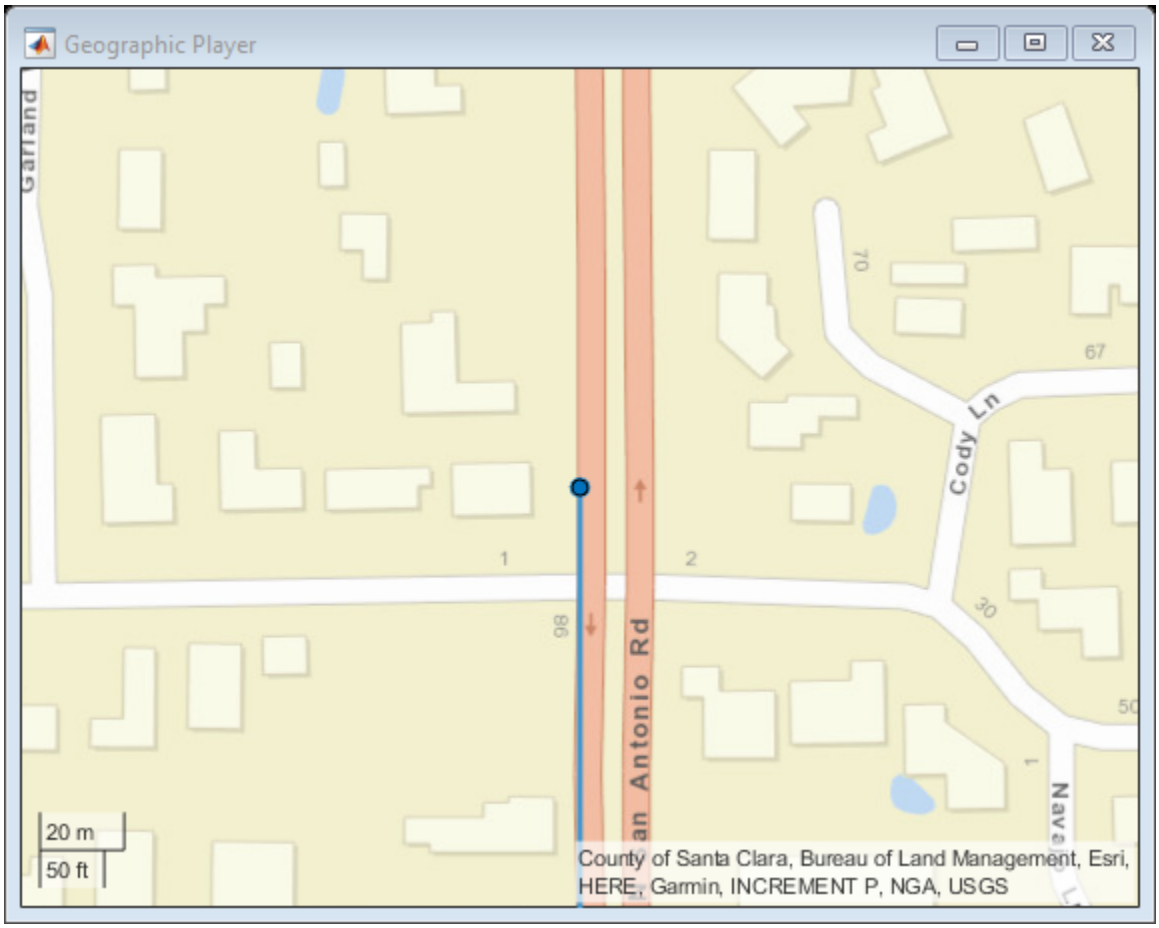
Load the `centerCamera.avi` camera video data and corresponding video timestamps.

```
recordedData = fullfile(toolboxdir('driving'), 'drivingdata', ...
    'udacity', 'drive_segment_09_29_16');
[videoReader, videoTime] = helperLoadCameraData(fullfile(recordedData));

% Show the first frame of the camera data
imageFrame = readFrame(videoReader);
imshow(imageFrame, 'Border', 'tight');
```

Load the GPS data from the `gpsSequence.mat` MAT-file.

```
data = load(fullfile(recordedData, 'gpsSequence.mat'));
gpsData = data.gpsTT;
```

```
% Plot the full route and the first position recorded from the GPS
gpsPlayer = geoplayer(gpsData.Latitude(1), gpsData.Longitude(1), 18);
plotRoute(gpsPlayer, gpsData.Latitude, gpsData.Longitude);
plotPosition(gpsPlayer, gpsData.Latitude(1), gpsData.Longitude(1));
```



Match Recorded Vehicle Position to a Road

Create a reader for reading the HERE HD Live Map tiles that cover all recorded GPS locations in the drive. If you have not previously set up HERE HDLM credentials, a dialog box prompts you to enter them. Enter the **Access Key ID** and **Access Key Secret** that you obtained from HERE Technologies, and click **OK**.

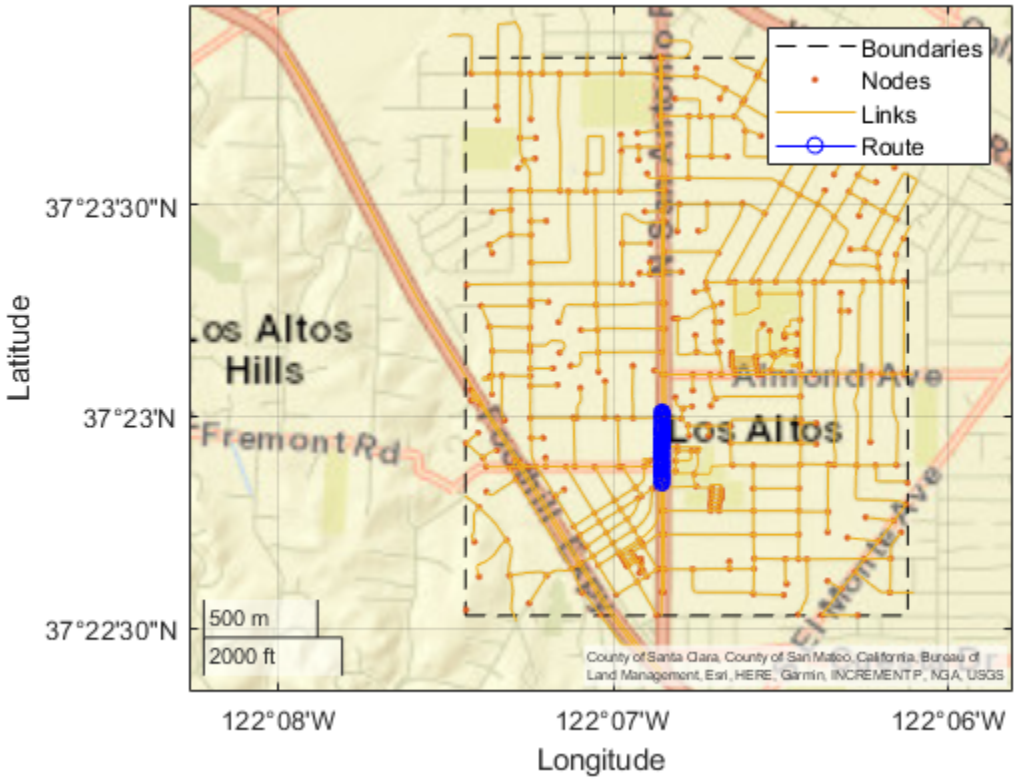
```
reader = hereHDLMReader(gpsData.Latitude, gpsData.Longitude);
```

Read and plot road topology data from the `TopologyGeometry` layer. This layer represents the configuration of the road network. Nodes of the network correspond to intersections and dead-ends. Links between nodes represent the shape of the connecting streets as polylines. The connectivity and geometry for these features are contained in the `LinksStartingInTile` and `NodesInTile` fields.

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

figure('Name', 'TopologyGeometry');
topologyAxes = plot(topologyLayer);
hold(topologyAxes, 'on');
```

```
geoplot(topologyAxes, gpsData.Latitude, gpsData.Longitude, ...
    'bo-', 'DisplayName', 'Route');
```

```
topologyLayer =

  TopologyGeometry with properties:

   Data:
                  HereTileId: 309106790
        IntersectingLinkRefs: [60×1 struct]
         LinksStartingInTile: [611×1 struct]
                NodesInTile: [448×1 struct]
    TileCenterHere2dCoordinate: [37.3865 -122.1130]

   Metadata:
                     Catalog: 'hrn:here:data::olp-here-had:here-hdlm-protobuf-na-2'
              CatalogVersion: 3377

  Use plot to visualize TopologyGeometry data.
```



The plotting functionality is captured within the `helperPlotLayer` function, which visualizes available data from a HD Live Map layer with the recorded drive on the same geographic axes. This function, defined at the end of the example, will be used to plot subsequent layers.

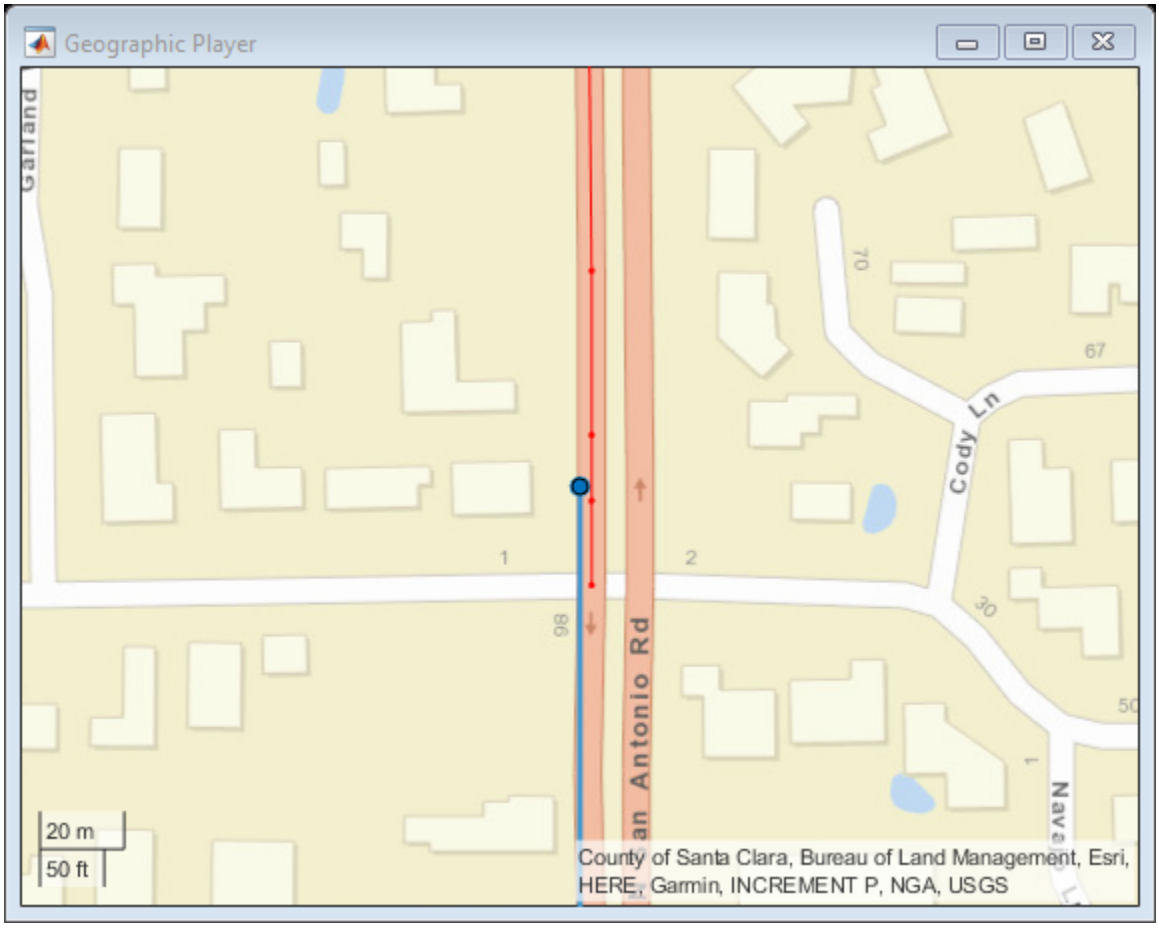
Given a GPS location recorded along a drive, you can use a route matching algorithm to determine which road on the network the recorded position corresponds to. This example uses a heuristic route matching algorithm that considers the nearest links spatially to the recorded geographic point. The algorithm applies the vehicle's direction of travel to determine the most probable link. This route matching approach does not consider road connectivity, vehicle access, or GPS data with high positional error. Therefore, this approach might not apply to all scenarios.

The `helperGetGeometry` function extracts geometry information from the given topology layer and returns this information in a table with the corresponding links.

```
topologyTable = helperGetGeometry(topologyLayer.LinksStartingInTile, ...
    {'LinkId', 'Geometry.Here2dCoordinateDiffs'});
topologyTable.Properties.VariableNames = {'LinkId', 'Geometry'};
```

The `HelperLinkMatcher` class creates a link matcher, which contains the shape geometry for each link in the desired map tile. This class uses a basic spatial analysis to match the recorded position to the shape coordinates of the road links.

```
linkMatcher = HelperLinkMatcher(topologyTable);

% Match first point of the recorded route to the most probable link
[linkId, linkLat, linkLon] = match(linkMatcher, gpsData.Latitude(1), ...
    gpsData.Longitude(1), gpsData.Velocity(1,1:2));

% Plot the shape geometry of the link
geoplot(gpsPlayer.Axes, linkLat, linkLon, 'r.-');
```



Retrieve Speed Limit Along a Matched Road

Characteristics common to all lanes along a given road are attributed to the link element that describes that road. One such attribute, speed limit, describes the maximum legal speed for vehicles traveling on the link. Once a given geographic coordinate is matched to a link, you can identify the speed limit along that link. Because features like speed limits often change along the length of a link, these attributes are identified for specific ranges of the link.

The `SpeedAttributes` layer contains information about the expected vehicle speed on a link, including the posted speed limit.

```
speedLayer = read(reader, 'SpeedAttributes');
```

The `helperGetSpeedLimits` function extracts speed limit data for the relevant length and direction of a link. As with extracting the geometry information for a link, specifically capturing the speed limit data requires specialized code.

```
speedTable = helperGetSpeedLimits(speedLayer);

% Find the speed limit entry for the matched link
speed = speedTable(speedTable.LinkId == linkId, :);
```

Match Recorded Position to a Lane Group

The HD Lane Model contains the lane-level geometry and attributes of the road, providing the detail necessary to support automated driving applications. Much like the Road Centerline Model, the HD Lane Model also follows a pattern of using topology to describe the road network at the lane level. Then features of the lane groups are attributed to the elements of this topology. In the HD Lane Model, the primary topological element is the lane group.

Read and plot lane topology data from the `LaneTopology` layer. This layer represents lane topology as lane groups and lane group connectors. Lane groups represent a group of lanes within a link (road segment). Lane group connectors connect individual lane groups to each other. The connectivity and geometry for these features are contained in the `LaneGroupsStartingInTile` and `LaneGroupConnectorsInTile` fields, for lane groups and lane group connectors, respectively.

```
laneTopologyLayer = read(reader, 'LaneTopology')
laneAxes = helperPlotLayer(laneTopologyLayer, ...
    gpsData.Latitude, gpsData.Longitude);
geolimits(laneAxes, [37.3823, 37.3838], [-122.1151, -122.1128]);
```

```
laneTopologyLayer =

  LaneTopology with properties:

   Data:
                   HereTileId: 309106790
     IntersectingLaneGroupRefs: [56×1 struct]
     LaneGroupConnectorsInTile: [1222×1 struct]
      LaneGroupsStartingInTile: [1863×1 struct]
    TileCenterHere2dCoordinate: [37.3865 -122.1130]

   Metadata:
                      Catalog: 'hrn:here:data::olp-here-had:here-hdlm-protobuf-na-2'
                CatalogVersion: 3377

  Use plot to visualize LaneTopology data.
```

The lane group represents multiple lanes. Therefore, the geometry of this element is given by the polygonal shape the group takes, as expressed by the left and right boundaries of the lane group. Obtain this lane geometry from the lane boundaries by using the `helperGetGeometry` function.

```
laneGroupFields = {'LaneGroupId', ...
    'BoundaryGeometry.LeftBoundary.Here2dCoordinateDiffs', ...
    'BoundaryGeometry.RightBoundary.Here2dCoordinateDiffs'};
laneTopologyTable = helperGetGeometry(laneTopologyLayer.LaneGroupsStartingInTile, ...
    laneGroupFields);
laneTopologyTable.Properties.VariableNames = {'LaneGroupId', ...
    'LeftGeometry', 'RightGeometry'};
```

As with developing a matching algorithm to identify the most probable travel link, matching the given GPS data to the most probable lane group can follow multiple approaches. The approach described here uses two layers: `LaneRoadReferences` and `LaneTopology`.

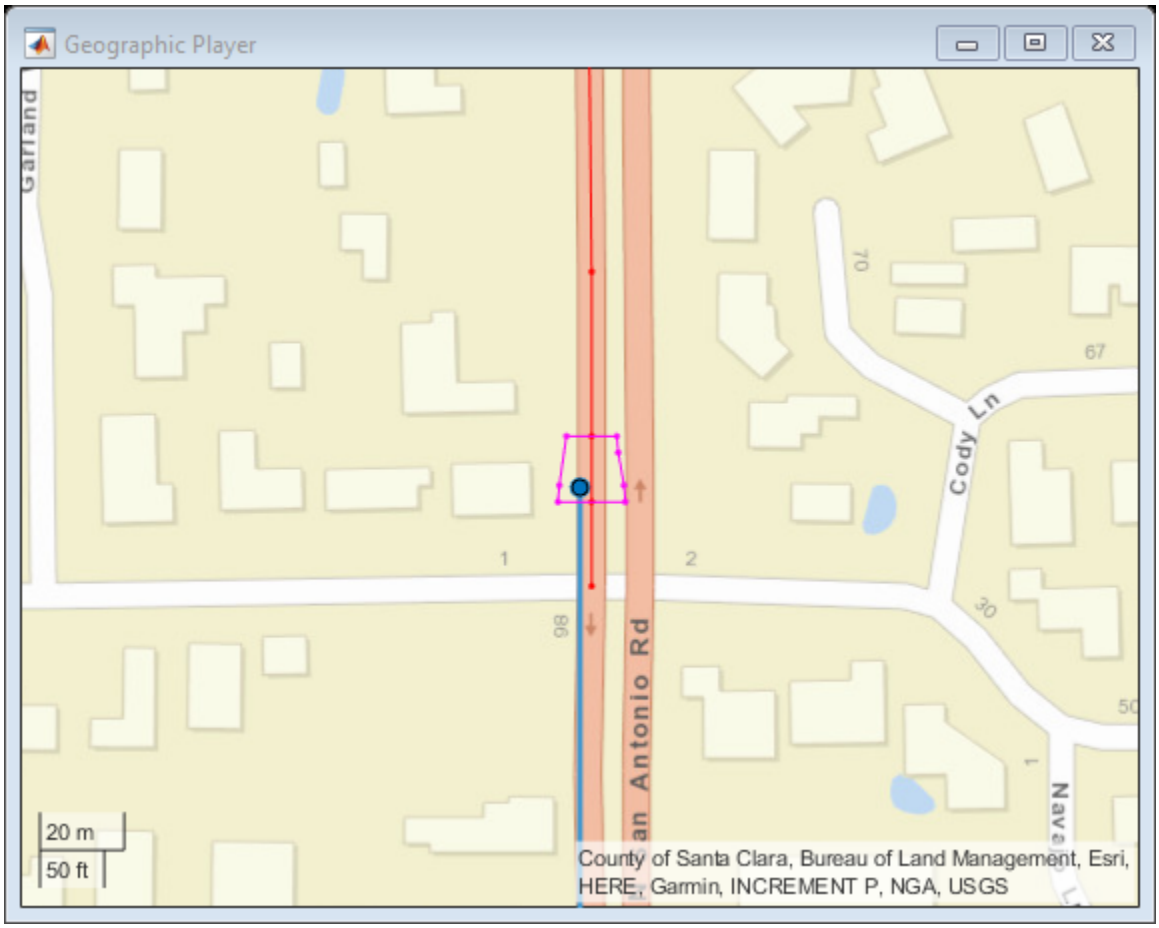
- The `LaneRoadReferences` layer enables you to translate a position on the Road Centerline Model (given by a link) to a corresponding position on the HD Lane Model (given by a lane group). Since the link was previously identified, you can filter the candidates for a lane group match to a smaller subset of all the lane groups available in the tile.
- The `LaneTopology` layer gives geometry data, which can be used to consider data that exists spatially within the boundaries of each candidate lane group. As with spatially matching GPS data to links, such an approach is prone to error and subject to the accuracy of the recorded GPS data. In addition to matching the lane group, you also need to match the direction vector of the vehicle relative to the orientation of the lane group. This step is necessary because the attributes of the lanes are defined with respect to the topology orientation.

Use the `helperGetReferences` function to generate a table of all lane groups that exist for at least some length of a link.

```
referenceLayer = read(reader, 'LaneRoadReferences');
referenceTable = helperGetReferences(referenceLayer);
```

Create a lane group matcher that contains the boundary geometry for each lane group in the desired map tile. The `HelperLaneGroupMatcher` class creates a lane group matcher that contains the boundary shape geometry for each lane group in the desired map tile. It also contains a reference table of links to lane groups. As with the `HelperLinkMatcher` class, this class uses a simple spatial analysis approach to determine if a given recorded coordinate exists within the boundaries of a lane group.

```
laneGroupMatcher = HelperLaneGroupMatcher(referenceTable, laneTopologyTable);

% Match the lane group and relative direction
[laneGroupId, isForward, boundGeometry] = match(laneGroupMatcher, linkId, ...
    gpsData.Latitude(1), gpsData.Longitude(1), gpsData.Velocity(1,1:2));

% Plot the boundary geometry of the lane group
geoplot(gpsPlayer.Axes, boundGeometry(:,1), boundGeometry(:,2), 'm.-');
```



Retrieve Lane Configurations for a Matched Lane Group

As with the speed attributes that are mapped to links by an identifier, lane attributes also assign features to lane groups by using a lane group ID. The `LaneAttributes` layer contains information about the lane groups, including the types of each lane in the group and the characteristics of the lane boundaries.

Use the `helperGetLaneAttributes` function to extract the different lane types and lane boundary markings for each lane group in the tile.

```
laneAttributesLayer = read(reader, 'LaneAttributes');
laneAttributesTable = helperGetLaneAttributes(laneAttributesLayer);

% Find the lane attribute entry for the matched lane group
laneAttribute = laneAttributesTable.LaneGroupId == laneGroupId;
```

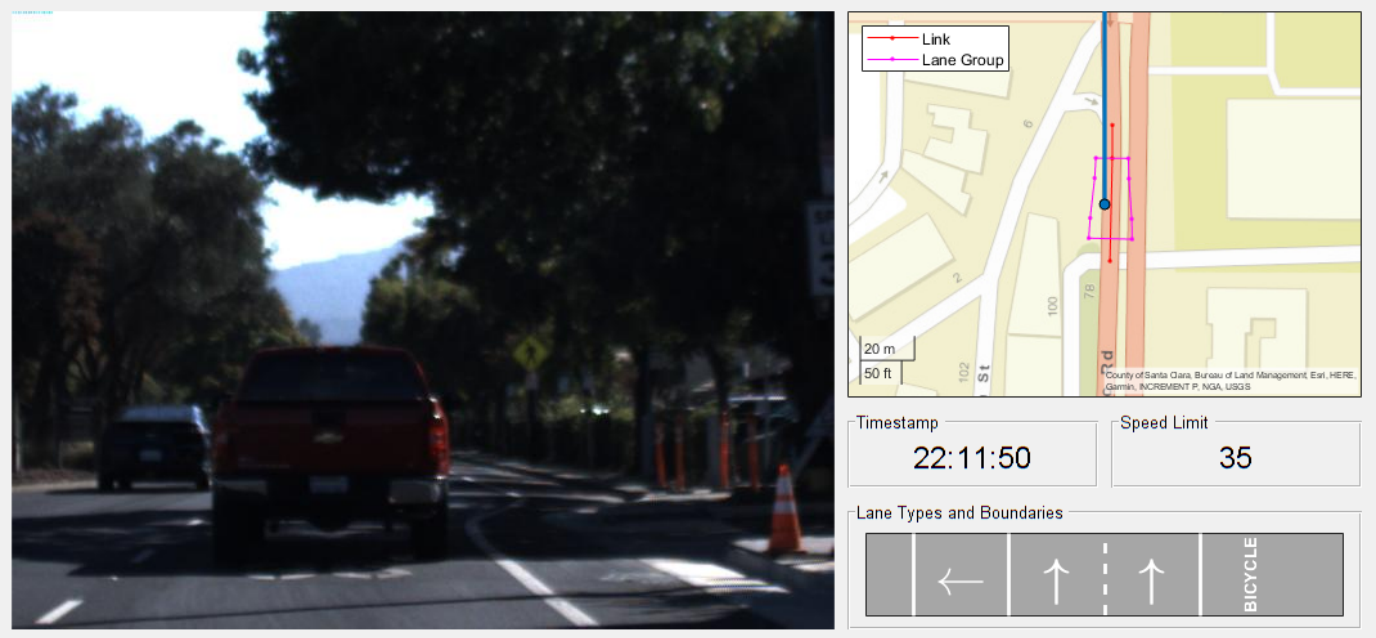
Visualize and Verify Recorded Drive with HERE HDLM Data

The matching algorithms and tables generated to identify the road and lane properties can be extended to a sequence of recorded GPS coordinates. For each time step, the vehicle's position is matched to a link and lane group on the road. In addition, the speed limit and lane configurations are displayed along with the corresponding camera images.

The `HelperHDLMUI` class creates a tool for streaming video and GPS data from a recorded drive and displaying relevant information from selected HERE HD Live Map layers at each recorded vehicle position.

```
hdlmUI = HelperHDLMUI(gpsData.Latitude(1), gpsData.Longitude(1));

% Synchronize the camera and GPS data into a common timetable
synchronizedData = synchronize(videoTime, gpsData);
videoReader.CurrentTime = 0;
maxDisplayRate = videoReader.FrameRate * 5;

% Initialize some variables to maintain history
prevLinkId      = 0;
prevLaneGroupId = 0;

for idx = 1 : height(synchronizedData)

    timeStamp = synchronizedData.Time(idx);

    % Check if the current timestamp has GPS data
    hasGPSFrame = ~(ismissing(synchronizedData.Latitude(idx)) || ...
        ismissing(synchronizedData.Longitude(idx)));

    if hasGPSFrame
        latitude  = synchronizedData.Latitude(idx);
        longitude = synchronizedData.Longitude(idx);
        velocity  = synchronizedData.Velocity(idx, 1:2);

        % Match GPS position to link
        [linkId, linkLat, linkLon] = match(linkMatcher, ...
            latitude, longitude, velocity);

        if linkId ~= prevLinkId
            % Update link
```

```
            updateLink(hdlmUI, linkLat, linkLon);
            prevLinkId = linkId;

            % Update speed limit
            speed = speedTable(speedTable.LinkId == linkId, :);
            updateSpeed(hdlmUI, speed.Value);
        end

        % Match GPS position to lane group
        [laneGroupId, isForward, boundGeometry] = match(laneGroupMatcher, linkId, ...
            latitude, longitude, velocity);

        if laneGroupId ~= prevLaneGroupId
            % Update lane group
            updateLaneGroup(hdlmUI, boundGeometry);
            prevLaneGroupId = laneGroupId;

            % Update lane types and boundary markings
            laneAttribute = laneAttributesTable.LaneGroupId == laneGroupId;
            plotLanes(hdlmUI, laneAttributesTable.Lanes{laneAttribute}, ...
                laneAttributesTable.LaneBoundaries{laneAttribute}, isForward);
        end

        updatePosition(hdlmUI, latitude, longitude);
    else
        % Read frame of the video
        imageFrame = readFrame(videoReader);
    end

    updateImage(hdlmUI, imageFrame);
    updateTime(hdlmUI, timeStamp);
    pause(1/maxDisplayRate);
end
```



Conclusion

In this example, you explored how to:

1. Access HD mapping data for a given GPS sequence from the HERE HD Live Map service and import that data into MATLAB.
2. Match recorded GPS data against the imported road network data to find the relevant link and lane group for each geographic coordinate.
3. Query attributes of the matched link and lane group, such as speed limit and lane types, to develop a tool for visually verifying features of the road against the recorded camera data.

The techniques discussed in this example can be further extended to support automated verification of perception algorithms.

Supporting Functions

`helperPlotLayer` plots layer data and route on a geographic plot.

```
function gx = helperPlotLayer(layer, latitude, longitude)
%helperPlotLayer Create geographic plot with layer data and route
%   gx = helperPlotLayer(layer, latitude, longitude) creates a geographic
%   axes plot with the plottable HDLM layer and the route given by latitude
%   and longitude on a new figure.

figure;

% Plot layer
gx = plot(layer);

% Enable adding data to the plot
hold(gx, 'on');

% Plot latitude, longitude data
geoplot(gx, latitude, longitude, 'bo-', 'DisplayName', 'Route');
hold(gx, 'off');
end
```

`helperGetGeometry` extracts geometry for topology elements from a layer in the form of a table.

```
function geometryTable = helperGetGeometry(layer, fields)
%helperGetGeometry Create a table with geometry for topology elements
%   geometryTable = helperGetGeometry(layer, fields) returns a table
%   formatted with the specified geometry fields of the TopologyGeometry
%   and LaneTopology layers.

% Pre-allocate struct
S = repmat(struct, size(layer));

for field = fields
    C = strsplit(field{:}, '.');
    for idx = 1:numel(layer)
        fieldname = strjoin(C, '');
        S(idx).(fieldname) = getfield(layer, {idx}, C{:});
    end
end
```

```
geometryTable = struct2table(S);
end
```

`helperGetSpeedLimits` extracts speed limit data from a layer in the form of a table.

```
function speedTable = helperGetSpeedLimits(layer)
%helperGetSpeedLimits Create a data table with speed limits
%   speedTable = helperGetSpeedLimits(layer) returns a table formatted with
%   the speed limit start, end, direction, and value along a specified link
%   as given by the SpeedAttributes layer object specified by layer.

speed = struct(...
    'LinkId',       {}, ...
    'Start',        {}, ...
    'End',          {}, ...
    'Direction',    {}, ...
    'Value',        {});

for idx = 1 : numel(layer.LinkAttribution)

    % Assign the link ID
    link            = layer.LinkAttribution(idx);
    attributions    = link.ParametricAttribution;

    % Examine each attribute assigned to the link
    for attrIndex = 1 : numel(attributions)

        linkAttr = vertcat(attributions.LinkParametricAttribution);

        % For each attribute, check if the speed limit information is
        % listed. If speed limit is provided, make an entry.
        for linkAttrIndex = 1 : numel(linkAttr)

            if ~isempty(linkAttr(linkAttrIndex).SpeedLimit)

                % Assign speed limit to specified link
                speedLimit = struct;
                speedLimit.LinkId       = link.LinkLocalRef;
                speedLimit.Start        = attributions(attrIndex).AppliesToRange.RangeOffsetFrom
                speedLimit.End          = attributions(attrIndex).AppliesToRange.RangeOffsetFrom
                speedLimit.Direction    = attributions(attrIndex).AppliesToDirection;
                speedLimit.Value        = linkAttr(linkAttrIndex).SpeedLimit.Value;

                % Convert KPH to MPH
                if strcmpi(linkAttr(linkAttrIndex).SpeedLimit.Unit, 'KILOMETERS_PER_HOUR')
                    speedLimit.Value = speedLimit.Value / 1.609;
                end

                if strcmpi(speedLimit.Direction, 'BOTH')
                    speed = [speed; speedLimit]; %#ok<AGROW>
                end
            end
        end

    end
end
```

```
speedTable = struct2table(speed);
end
```

`helperGetReferences` extracts lane road references from a layer object in the form of a table.

```
function laneRoadReferenceTable = helperGetReferences(layer)
%helperGetReferences Create a data table with lane road references
%   laneRoadReferenceTable = helperGetReferences(layer) returns a table
%   formatted with a list of all lane groups existing on a specified link
%   as given by the LaneRoadReferences layer object specified by layer.

numLinks = numel(layer.LinkLaneGroupReferences);
reference = repmat(struct('LinkId', {}, 'LaneGroupId', {}), numLinks, 1);

% Get references from links to lane groups
for idx = 1 : numLinks
    link = layer.LinkLaneGroupReferences(idx);
    laneGroups = vertcat(link.LaneGroupReferences.LaneGroupRef);

    reference(idx).LinkId = link.LinkLocalRef;
    reference(idx).LaneGroupId = [laneGroups(:).LaneGroupId]';
end

laneRoadReferenceTable = struct2table(reference);
end
```

`helperGetLaneAttributes` extracts lane attributes from a layer object in the form of a table.

```
function laneAttributesTable = helperGetLaneAttributes(layer)
%helperGetLaneAttributes Create a table with lane and boundary types
%   laneAttributesTable = helperGetLaneAttributes(layer) returns a table
%   formatted with the lane types and the lane boundary markings for each
%   lane group in the LaneAttributes layer object specified by layer.

for laneGroupAttrIndex = 1 : numel(layer.LaneGroupAttribution)
    laneGroup = layer.LaneGroupAttribution(laneGroupAttrIndex);
    attributes(laneGroupAttrIndex).LaneGroupId = laneGroup.LaneGroupRef; %#ok

    % Get lane types for each lane group
    for laneAttrIndex = 1 : numel(laneGroup.LaneAttribution)

        lane = laneGroup.LaneAttribution(laneAttrIndex);

        laneAttr = vertcat(lane.ParametricAttribution);
        laneAttr = vertcat(laneAttr.LaneParametricAttribution);

        for idx = 1 : numel(laneAttr)
            if ~isempty(laneAttr(idx).LaneType)
                attributes(laneGroupAttrIndex).Lanes{lane.LaneNumber} = ...
                    laneAttr(idx).LaneType;
            end
        end
    end

    % Get lane boundaries for each lane group
    for laneBoundaryIndex = 1 : numel(laneGroup.LaneBoundaryAttribution)
        laneBoundary = laneGroup.LaneBoundaryAttribution(laneBoundaryIndex);
        boundaries   = vertcat(laneBoundary.ParametricAttribution.LaneBoundaryParametricAttributi
```

```
            attributes(laneGroupAttrIndex).LaneBoundaries{laneBoundary.LaneBoundaryNumber} = ...
                boundaries.LaneBoundaryMarking;
        end

    end

    laneAttributesTable = struct2table(attributes);
    end
```

`helperLoadCameraData` loads a video reader and timestamps from folder.

```
function [videoReader, videoTime] = helperLoadCameraData(dirName)
%helperLoadCameraData Load camera images from folder in a timetable
%   [videoReader, videoTime] = helperLoadCameraData(dirName) loads a video
%   from the folder dirName. Timestamps for the video are read from a
%   MAT-file in the folder named timeStamps.mat.

if ~isfolder(dirName)
    error('Expected dirName to be a path to a folder.')
end

matFileName = fullfile(dirName, 'centerCameraTime.mat');
if exist(matFileName, 'file') ~= 2
    error('Expected dirName to have a MAT-file named centerCameraTime.mat containing timestamps.
end

% Load the MAT-file with timestamps
ts = load(matFileName);
fieldNames = fields(ts);
Time = ts.(fieldNames{1});

videoFileName = fullfile(dirName, 'centerCamera.avi');
if exist(matFileName, 'file') ~= 2
    error('Expected dirName to have a video file named centerCamera.avi.')
end

% Load the video file
videoTime = timetable(Time);
videoReader = VideoReader(videoFileName);
end
```

See Also

`hereHDLMReader` | `plot` | `read`

More About

- “HERE HD Live Map Layers” on page 4-15
- “Read and Visualize HERE HD Live Map Data” on page 4-7
- “Import HERE HD Live Map Roads into Driving Scenario” on page 5-93

Build a Map from Lidar Data

This example shows how to process 3-D lidar data from a sensor mounted on a vehicle to progressively build a map, with assistance from inertial measurement unit (IMU) readings. Such a map can facilitate path planning for vehicle navigation or can be used for localization. For evaluating the generated map, this example also shows how to compare the trajectory of the vehicle against global positioning system (GPS) recording.

Overview

High Definition (HD) maps are mapping services that provide precise geometry of roads up to a few centimeters in accuracy. This level of accuracy makes HD maps suitable for automated driving workflows such as localization and navigation. Such HD maps are generated by building a map from 3-D lidar scans, in conjunction with high-precision GPS and or IMU sensors and can be used to localize a vehicle within a few centimeters. This example implements a subset of features required to build such a system.

In this example, you learn how to:

- Load, explore and visualize recorded driving data
- Build a map using lidar scans
- Improve the map using IMU readings

Load and Explore Recorded Driving Data

The data used in this example is from this GitHub® repository, and represents approximately 100 seconds of lidar, GPS and IMU data. The data is saved in the form of MAT-files, each containing a `timetable`. Download the MAT-files from the repository and load them into the MATLAB® workspace.

Note: This download can take a few minutes.

```
baseDownloadURL = 'https://github.com/mathworks/udacity-self-driving-data-subset/raw/master/drive
dataFolder      = fullfile(tempdir, 'drive_segment_11_18_16', filesep);
options         = weboptions('Timeout', Inf);

lidarFileName = dataFolder + "lidarPointClouds.mat";
imuFileName   = dataFolder + "imuOrientations.mat";
gpsFileName   = dataFolder + "gpsSequence.mat";

folderExists  = exist(dataFolder, 'dir');
matfilesExist = exist(lidarFileName, 'file') && exist(imuFileName, 'file') ...
    && exist(gpsFileName, 'file');

if ~folderExists
    mkdir(dataFolder);
end

if ~matfilesExist
    disp('Downloading lidarPointClouds.mat (613 MB)...')
    websave(lidarFileName, baseDownloadURL + "lidarPointClouds.mat", options);

    disp('Downloading imuOrientations.mat (1.2 MB)...')
    websave(imuFileName, baseDownloadURL + "imuOrientations.mat", options);
```

```
    disp('Downloading gpsSequence.mat (3 KB)...')
    websave(gpsFileName, baseDownloadURL + "gpsSequence.mat", options);
end
```

```
Downloading lidarPointClouds.mat (613 MB)...
Downloading imuOrientations.mat (1.2 MB)...
Downloading gpsSequence.mat (3 KB)...
```

First, load the point cloud data saved from a Velodyne® HDL32E lidar. Each scan of lidar data is stored as a 3-D point cloud using the `pointCloud (Computer Vision Toolbox)` object. This object internally organizes the data using a K-d tree data structure for faster search. The timestamp associated with each lidar scan is recorded in the `Time` variable of the timetable.

```
% Load lidar data from MAT-file
data = load(lidarFileName);
lidarPointClouds = data.lidarPointClouds;

% Display first few rows of lidar data
head(lidarPointClouds)
```

```
ans =

  8×1 timetable

        Time            PointCloud
    ________________    ________________

    23:46:10.5115    [1×1 pointCloud]
    23:46:10.6115    [1×1 pointCloud]
    23:46:10.7116    [1×1 pointCloud]
    23:46:10.8117    [1×1 pointCloud]
    23:46:10.9118    [1×1 pointCloud]
    23:46:11.0119    [1×1 pointCloud]
    23:46:11.1120    [1×1 pointCloud]
    23:46:11.2120    [1×1 pointCloud]
```

Load the GPS data from the MAT-file. The `Latitude`, `Longitude`, and `Altitude` variables of the `timetable` are used to store the geographic coordinates recorded by the GPS device on the vehicle.

```
% Load GPS sequence from MAT-file
data = load(gpsFileName);
gpsSequence = data.gpsSequence;

% Display first few rows of GPS data
head(gpsSequence)
```

```
ans =

  8×3 timetable

        Time         Latitude    Longitude    Altitude
    ________________    ________    _________    ________

    23:46:11.4563      37.4       -122.11       -42.5
    23:46:12.4563      37.4       -122.11       -42.5
```

```
    23:46:13.4565      37.4       -122.11      -42.5
    23:46:14.4455      37.4       -122.11      -42.5
    23:46:15.4455      37.4       -122.11      -42.5
    23:46:16.4567      37.4       -122.11      -42.5
    23:46:17.4573      37.4       -122.11      -42.5
    23:46:18.4656      37.4       -122.11      -42.5
```

Load the IMU data from the MAT-file. An IMU typically consists of individual sensors that report information about the motion of the vehicle. They combine multiple sensors, including accelerometers, gyroscopes and magnetometers. The `Orientation` variable stores the reported orientation of the IMU sensor. These readings are reported as quaternions. Each reading is specified as a 1-by-4 vector containing the four quaternion parts. Convert the 1-by-4 vector to a `quaternion` object.

```
% Load IMU recordings from MAT-file
data = load(imuFileName);
imuOrientations = data.imuOrientations;

% Convert IMU recordings to quaternion type
imuOrientations = convertvars(imuOrientations, 'Orientation', 'quaternion');

% Display first few rows of IMU data
head(imuOrientations)
```

```
ans =

  8×1 timetable

        Time           Orientation
    _____________    _______________

    23:46:11.4570    [1×1 quaternion]
    23:46:11.4605    [1×1 quaternion]
    23:46:11.4620    [1×1 quaternion]
    23:46:11.4655    [1×1 quaternion]
    23:46:11.4670    [1×1 quaternion]
    23:46:11.4705    [1×1 quaternion]
    23:46:11.4720    [1×1 quaternion]
    23:46:11.4755    [1×1 quaternion]
```

To understand how the sensor readings come in, for each sensor, compute the approximate frame `duration`.

```
lidarFrameDuration = median(diff(lidarPointClouds.Time));
gpsFrameDuration   = median(diff(gpsSequence.Time));
imuFrameDuration   = median(diff(imuOrientations.Time));

% Adjust display format to seconds
lidarFrameDuration.Format = 's';
gpsFrameDuration.Format   = 's';
imuFrameDuration.Format   = 's';

% Compute frame rates
lidarRate = 1/seconds(lidarFrameDuration);
gpsRate   = 1/seconds(gpsFrameDuration);
```

```
imuRate   = 1/seconds(imuFrameDuration);

% Display frame durations and rates
fprintf('Lidar: %s, %3.1f Hz\n', char(lidarFrameDuration), lidarRate);
fprintf('GPS  : %s, %3.1f Hz\n', char(gpsFrameDuration), gpsRate);
fprintf('IMU  : %s, %3.1f Hz\n', char(imuFrameDuration), imuRate);
```

```
Lidar: 0.10008 sec, 10.0 Hz
GPS  : 1.0001 sec, 1.0 Hz
IMU  : 0.002493 sec, 401.1 Hz
```

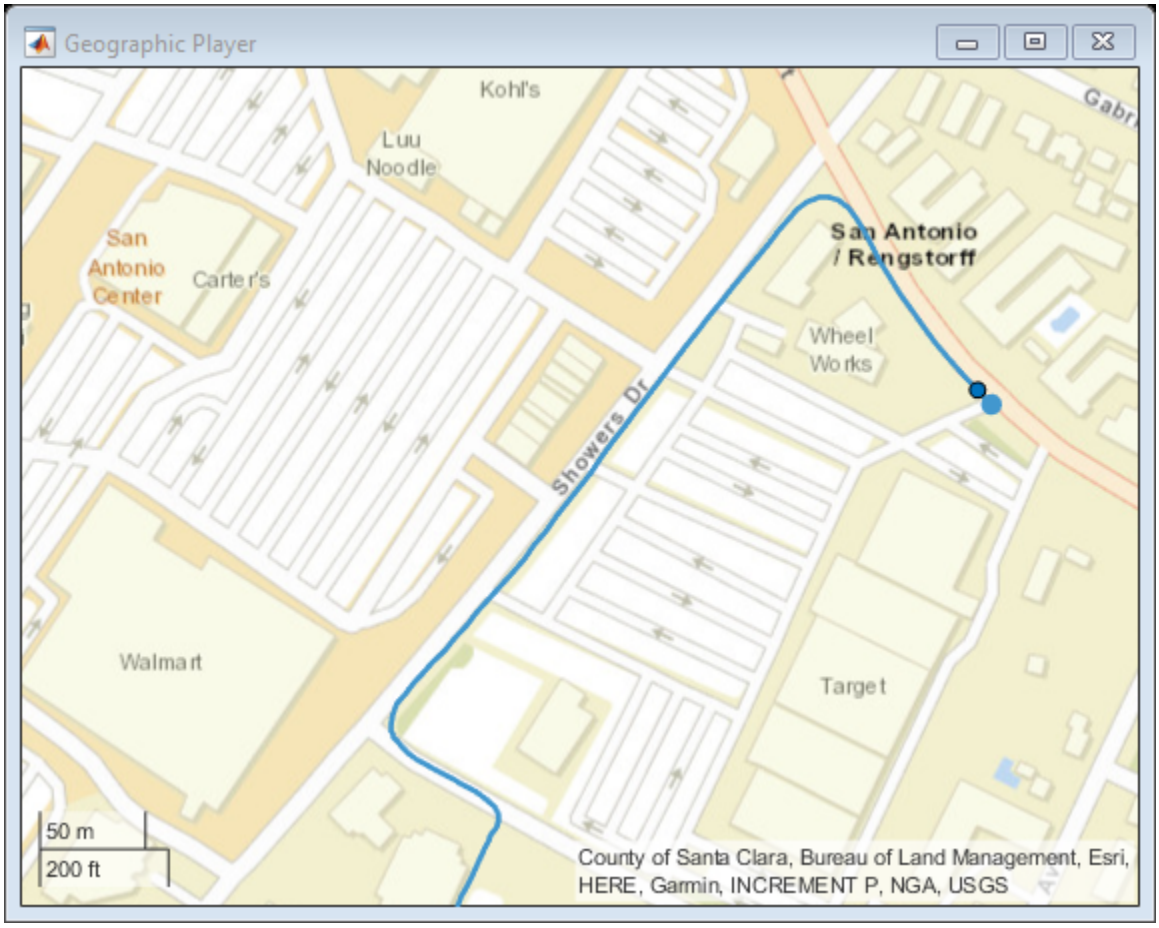
The GPS sensor is the slowest, running at a rate close to 1 Hz. The lidar is next slowest, running at a rate close to 10 Hz, followed by the IMU at a rate of almost 400 Hz.

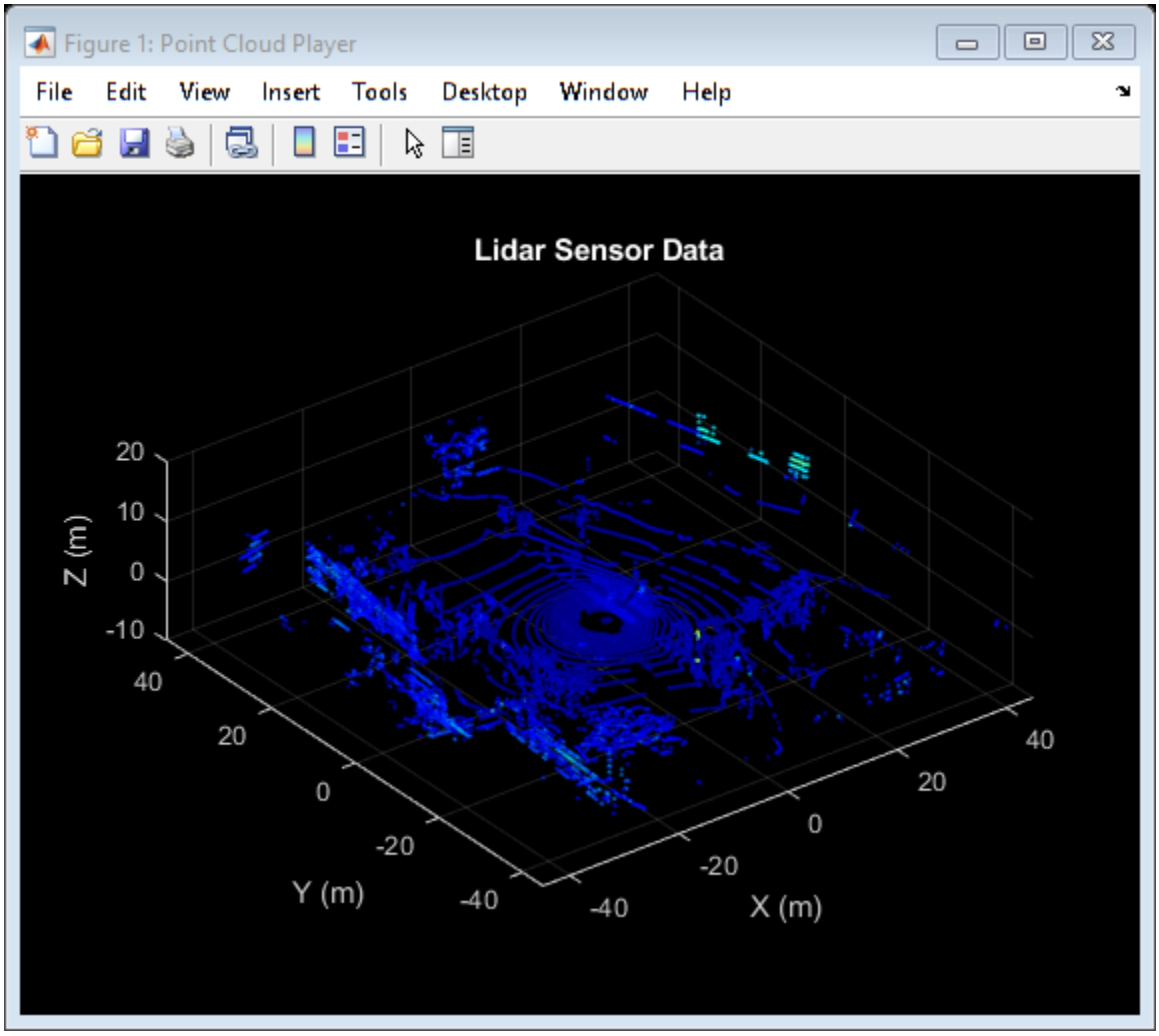
Visualize Driving Data

To understand what the scene contains, visualize the recorded data using streaming players. To visualize the GPS readings, use `geoplayer`. To visualize lidar readings using `pcplayer (Computer Vision Toolbox)`.

```
% Create a geoplayer to visualize streaming geographic coordinates
latCenter = gpsSequence.Latitude(1);
lonCenter = gpsSequence.Longitude(1);
zoomLevel = 17;

gpsPlayer = geoplayer(latCenter, lonCenter, zoomLevel);

% Plot the full route
plotRoute(gpsPlayer, gpsSequence.Latitude, gpsSequence.Longitude);

% Determine limits for the player
xlimits = [-45 45]; % meters
ylimits = [-45 45];
zlimits = [-10 20];

% Create a pcplayer to visualize streaming point clouds from lidar sensor
lidarPlayer = pcplayer(xlimits, ylimits, zlimits);

% Customize player axes labels
xlabel(lidarPlayer.Axes, 'X (m)')
ylabel(lidarPlayer.Axes, 'Y (m)')
zlabel(lidarPlayer.Axes, 'Z (m)')

title(lidarPlayer.Axes, 'Lidar Sensor Data')

% Align players on screen
helperAlignPlayers({gpsPlayer, lidarPlayer});

% Outer loop over GPS readings (slower signal)
for g = 1 : height(gpsSequence)-1

    % Extract geographic coordinates from timetable
    latitude  = gpsSequence.Latitude(g);
    longitude = gpsSequence.Longitude(g);

    % Update current position in GPS display
    plotPosition(gpsPlayer, latitude, longitude);
```

```
    % Compute the time span between the current and next GPS reading
    timeSpan = timerange(gpsSequence.Time(g), gpsSequence.Time(g+1));

    % Extract the lidar frames recorded during this time span
    lidarFrames = lidarPointClouds(timeSpan, :);

    % Inner loop over lidar readings (faster signal)
    for l = 1 : height(lidarFrames)

        % Extract point cloud
        ptCloud = lidarFrames.PointCloud(l);

        % Update lidar display
        view(lidarPlayer, ptCloud);

        % Pause to slow down the display
        pause(0.01)
    end
end
```

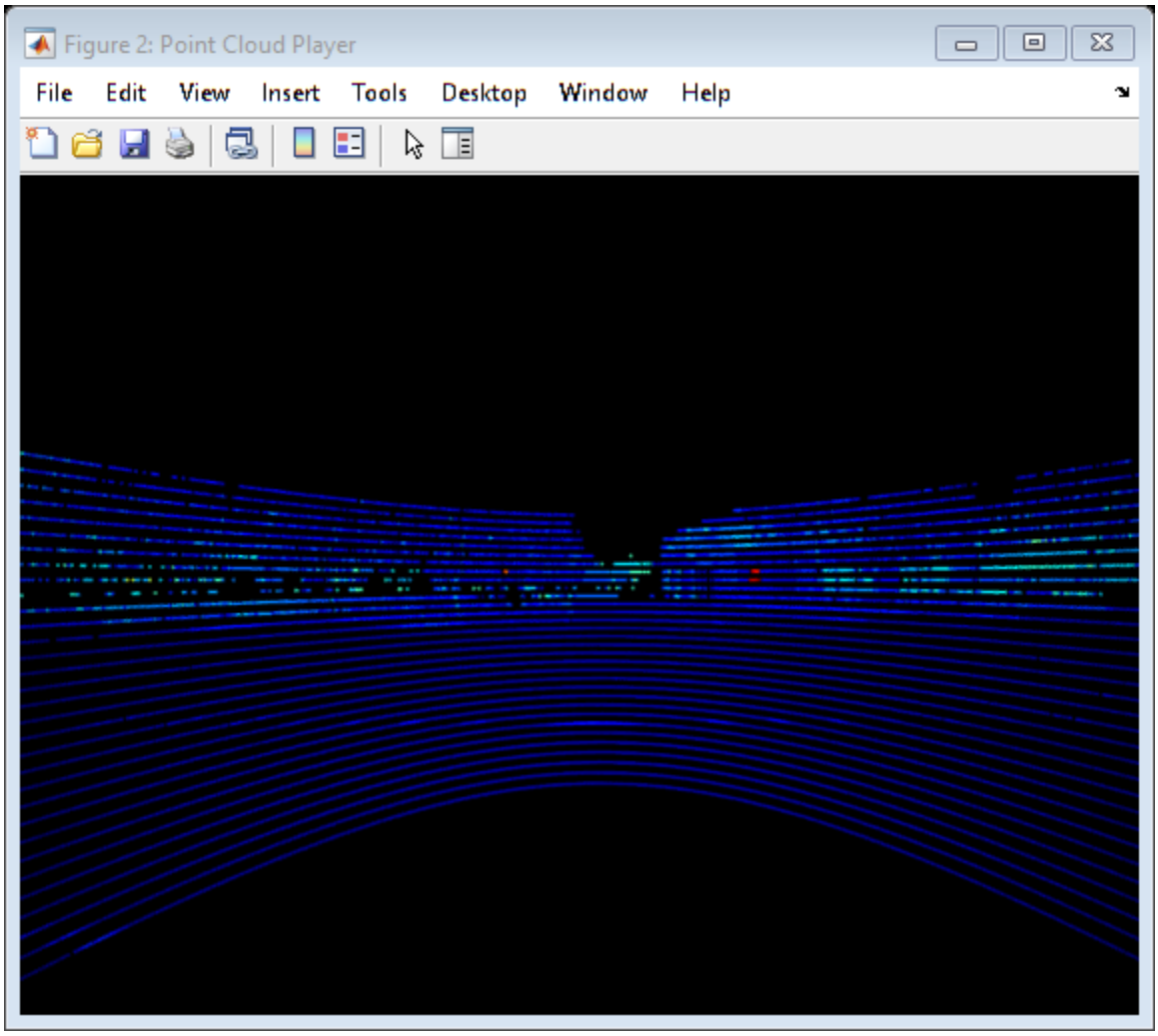




Use Recorded Lidar Data to Build a Map

Lidars are powerful sensors that can be used for perception in challenging environments where other sensors are not useful. They provide a detailed, full 360 degree view of the environment of the vehicle.

```
% Hide players
hide(gpsPlayer)
hide(lidarPlayer)

% Select a frame of lidar data to demonstrate registration workflow
frameNum = 600;
ptCloud = lidarPointClouds.PointCloud(frameNum);

% Display and rotate ego view to show lidar data
helperVisualizeEgoView(ptCloud);
```



Lidars can be used to build centimeter-accurate HD maps, including HD maps of entire cities. These maps can later be used for in-vehicle localization. A typical approach to build such a map is to align successive lidar scans obtained from the moving vehicle and combine them into a single large point cloud. The rest of this example explores this approach to building a map.

1. **Align lidar scans:** Align successive lidar scans using a point cloud registration technique like the iterative closest point (ICP) algorithm or the normal-distributions transform (NDT) algorithm. See `pcregistericp (Computer Vision Toolbox)` and `pcregisterndt (Computer Vision Toolbox)` for more details about each algorithm. This example uses NDT, because it is typically more accurate, especially when considering rotations. The `pcregisterndt` function returns the rigid transformation that aligns the moving point cloud with respect to the reference point cloud. By successively composing these transformations, each point cloud is transformed back to the reference frame of the first point cloud.
2. **Combine aligned scans:** Once a new point cloud scan is registered and transformed back to the reference frame of the first point cloud, the point cloud can be merged with the first point cloud using `pcmerge (Computer Vision Toolbox)`.

Start by taking two point clouds corresponding to nearby lidar scans. To speed up processing, and accumulate enough motion between scans, use every tenth scan.

```
skipFrames = 10;
frameNum   = 100;

fixed  = lidarPointClouds.PointCloud(frameNum);
moving = lidarPointClouds.PointCloud(frameNum + skipFrames);
```

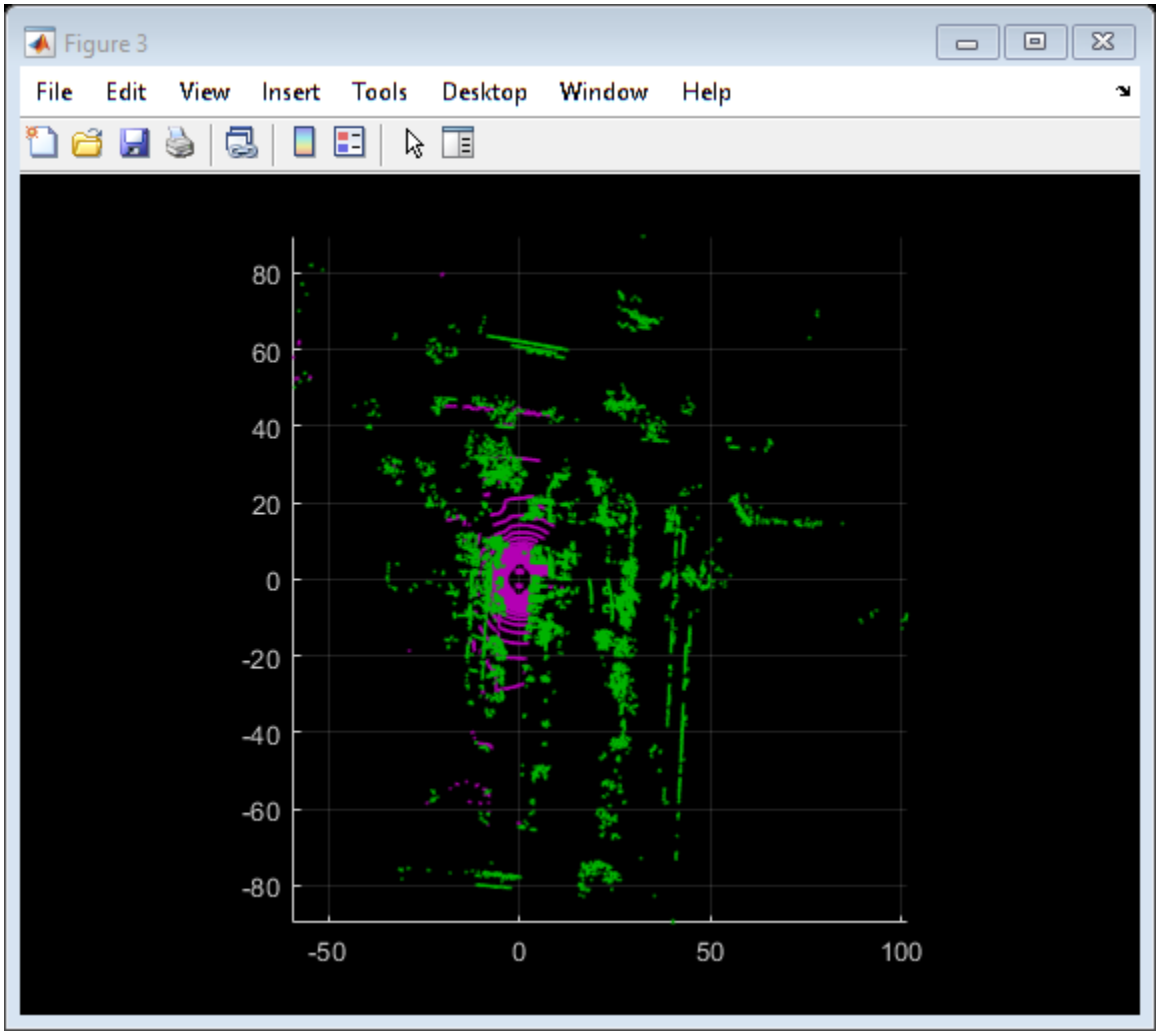
Prior to registration, process the point cloud so as to retain structures in the point cloud that are distinctive. These pre-processing steps include the following: * Detect and remove the ground plane * Detect and remove ego-vehicle

These steps are described in more detail in the “Ground Plane and Obstacle Detection Using Lidar” on page 7-107 example. In this example, the `helperProcessPointCloud` helper function accomplishes these steps.

```
fixedProcessed  = helperProcessPointCloud(fixed);
movingProcessed = helperProcessPointCloud(moving);
```

Display the raw and processed point clouds in top-view. Magenta points were removed during processing. These points correspond to the ground plane and ego vehicle.

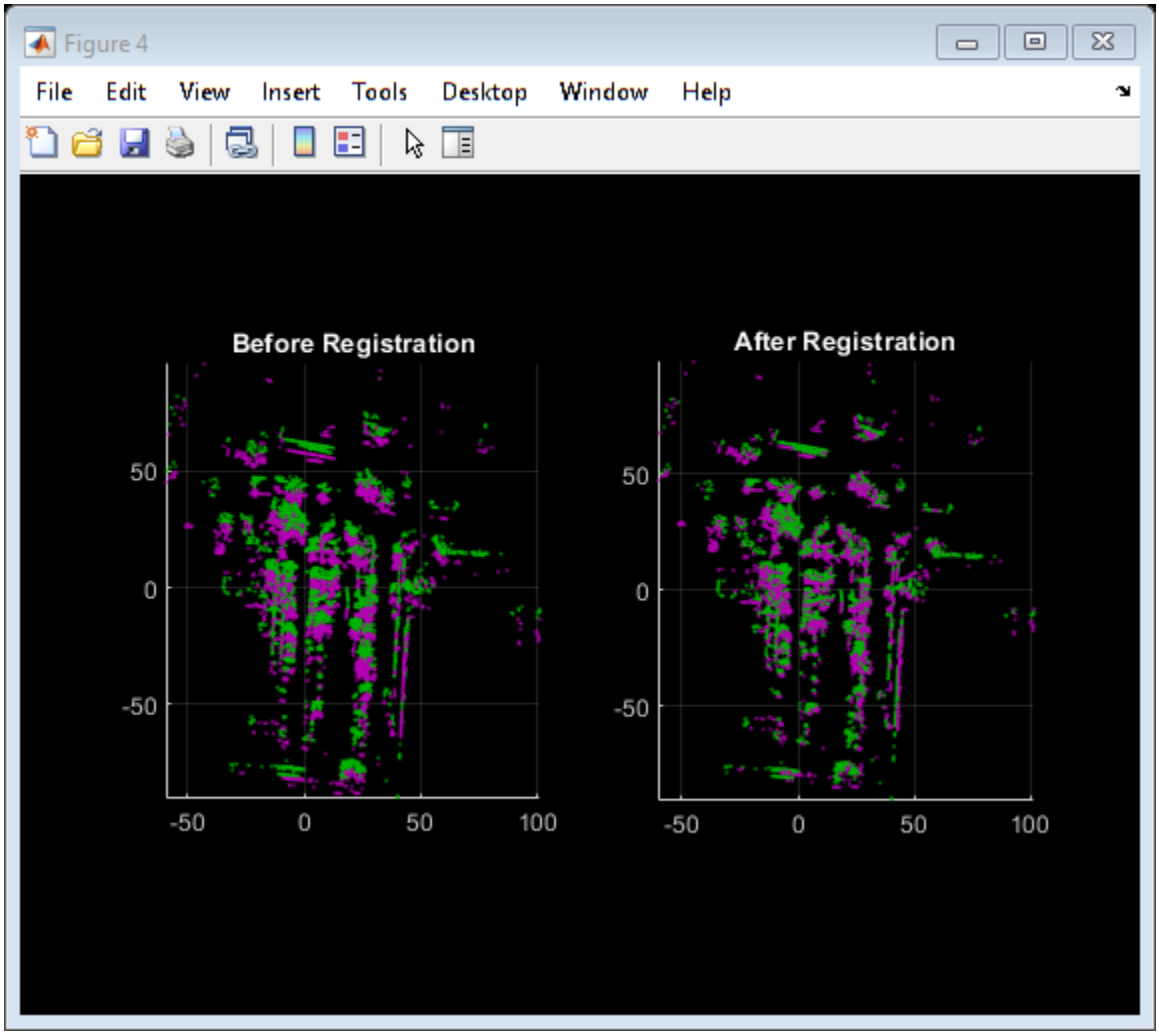
```
hFigFixed = figure;
pcshowpair(fixed, fixedProcessed)
view(2);                            % Adjust view to show top-view

helperMakeFigurePublishFriendly(hFigFixed);

% Downsample the point clouds prior to registration. Downsampling improves
% both registration accuracy and algorithm speed.
downsamplePercent = 0.1;
fixedDownsampled  = pcdownsample(fixedProcessed, 'random', downsamplePercent);
movingDownsampled = pcdownsample(movingProcessed, 'random', downsamplePercent);
```



After preprocessing the point clouds, register them using NDT. Visualize the alignment before and after registration.

```
regGridStep = 5;
tform = pcregisterndt(movingDownsampled, fixedDownsampled, regGridStep);

movingReg = pctransform(movingProcessed, tform);

% Visualize alignment in top-view before and after registration
hFigAlign = figure;

subplot(121)
pcshowpair(movingProcessed, fixedProcessed)
title('Before Registration')
view(2)

subplot(122)
pcshowpair(movingReg, fixedProcessed)
title('After Registration')
view(2)
```

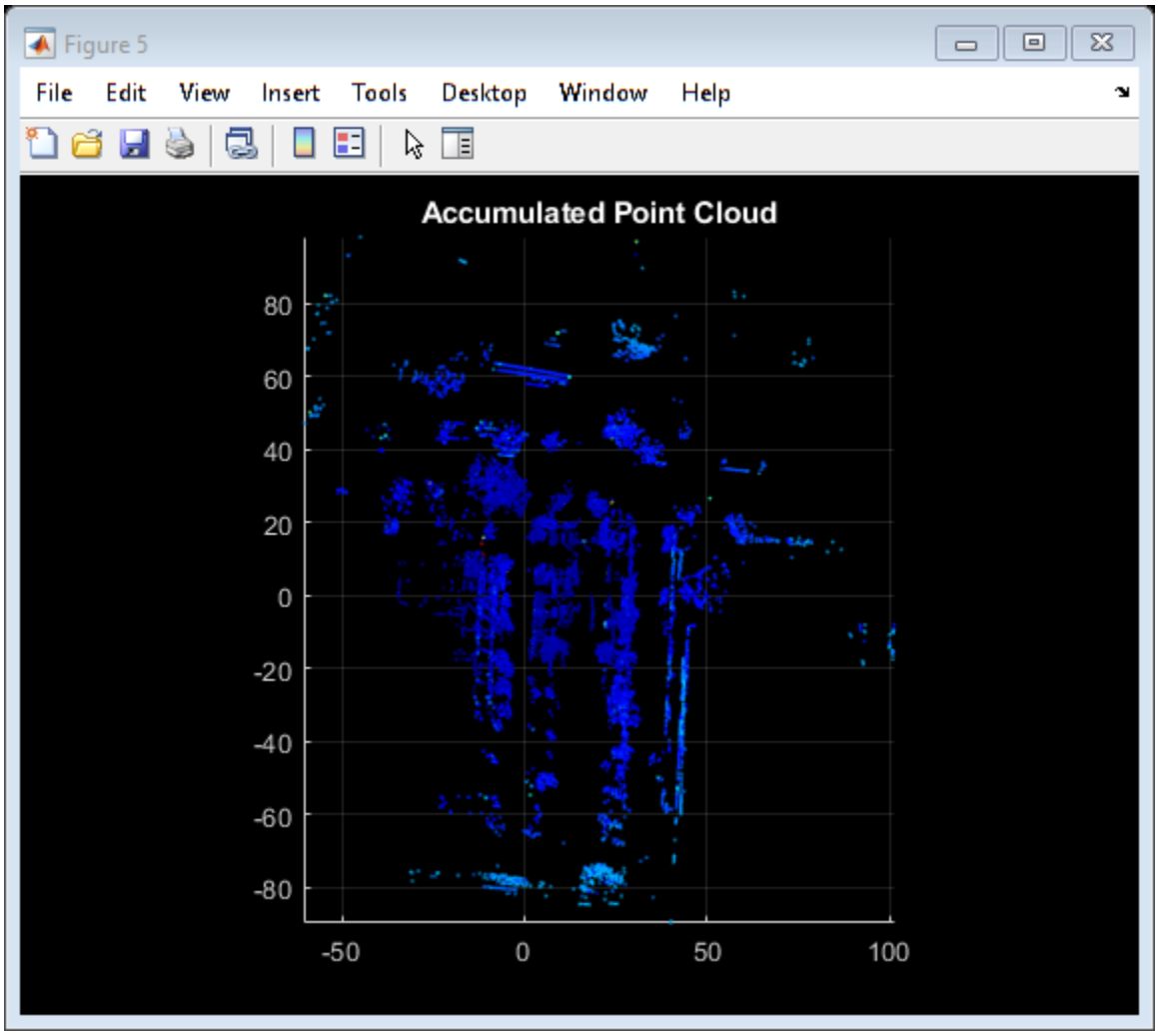
```
helperMakeFigurePublishFriendly(hFigAlign);
```



Notice that the point clouds are well-aligned after registration. Even though the point clouds are closely aligned, the alignment is still not perfect.

Next, merge the point clouds using `pcmerge`.

```
mergeGridStep = 0.5;
ptCloudAccum = pcmerge(fixedProcessed, movingReg, mergeGridStep);

hFigAccum = figure;
pcshow(ptCloudAccum)
title('Accumulated Point Cloud')
view(2)

helperMakeFigurePublishFriendly(hFigAccum);
```



Now that the processing pipeline for a single pair of point clouds is well-understood, put this together in a loop over the entire sequence of recorded data. The `helperLidarMapBuilder` class puts all this together. The `updateMap` method of the class takes in a new point cloud and goes through the steps detailed previously:

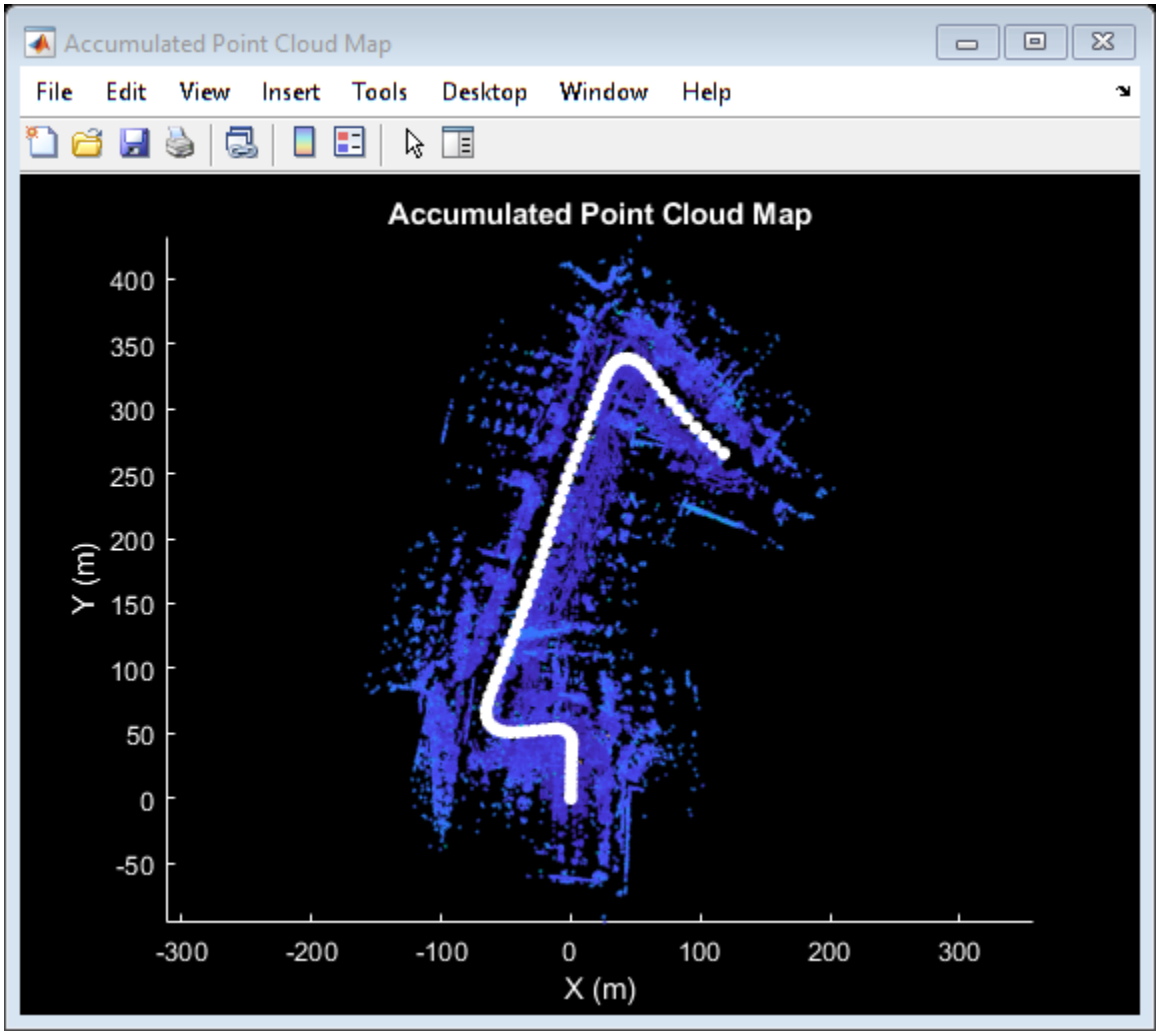
- Processing the point cloud by removing the ground plane and ego vehicle, using the `processPointCloud` method.
- Downsampling the point cloud.
- Estimating the rigid transformation required to merge the previous point cloud with the current point cloud.
- Transforming the point cloud back to the first frame.
- Merging the point cloud with the accumulated point cloud map.

Additionally, the `updateMap` method also accepts an initial transformation estimate, which is used to initialize the registration. A good initialization can significantly improve results of registration. Conversely, a poor initialization can adversely affect registration. Providing a good initialization can also improve the execution time of the algorithm.

A common approach to providing an initial estimate for registration is to use a constant velocity assumption. Use the transformation from the previous iteration as the initial estimate.

The `updateDisplay` method additionally creates and updates a 2-D top-view streaming point cloud display.

```
% Create a map builder object
mapBuilder = helperLidarMapBuilder('DownsamplePercent', downsamplePercent);

% Set random number seed
rng(0);

closeDisplay = false;
numFrames    = height(lidarPointClouds);

tform = rigid3d;
for n = 1 : skipFrames : numFrames - skipFrames

    % Get the nth point cloud
    ptCloud = lidarPointClouds.PointCloud(n);

    % Use transformation from previous iteration as initial estimate for
    % current iteration of point cloud registration. (constant velocity)
    initTform = tform;

    % Update map using the point cloud
    tform = updateMap(mapBuilder, ptCloud, initTform);

    % Update map display
    updateDisplay(mapBuilder, closeDisplay);
end
```



Point cloud registration alone builds a map of the environment traversed by the vehicle. While the map may appear locally consistent, it might have developed significant drift over the entire sequence.

Use the recorded GPS readings as a ground truth trajectory, to visually evaluate the quality of the built map. First convert the GPS readings (latitude, longitude, altitude) to a local coordinate system. Select a local coordinate system that coincides with the origin of the first point cloud in the sequence. This conversion is computed using two transformations:

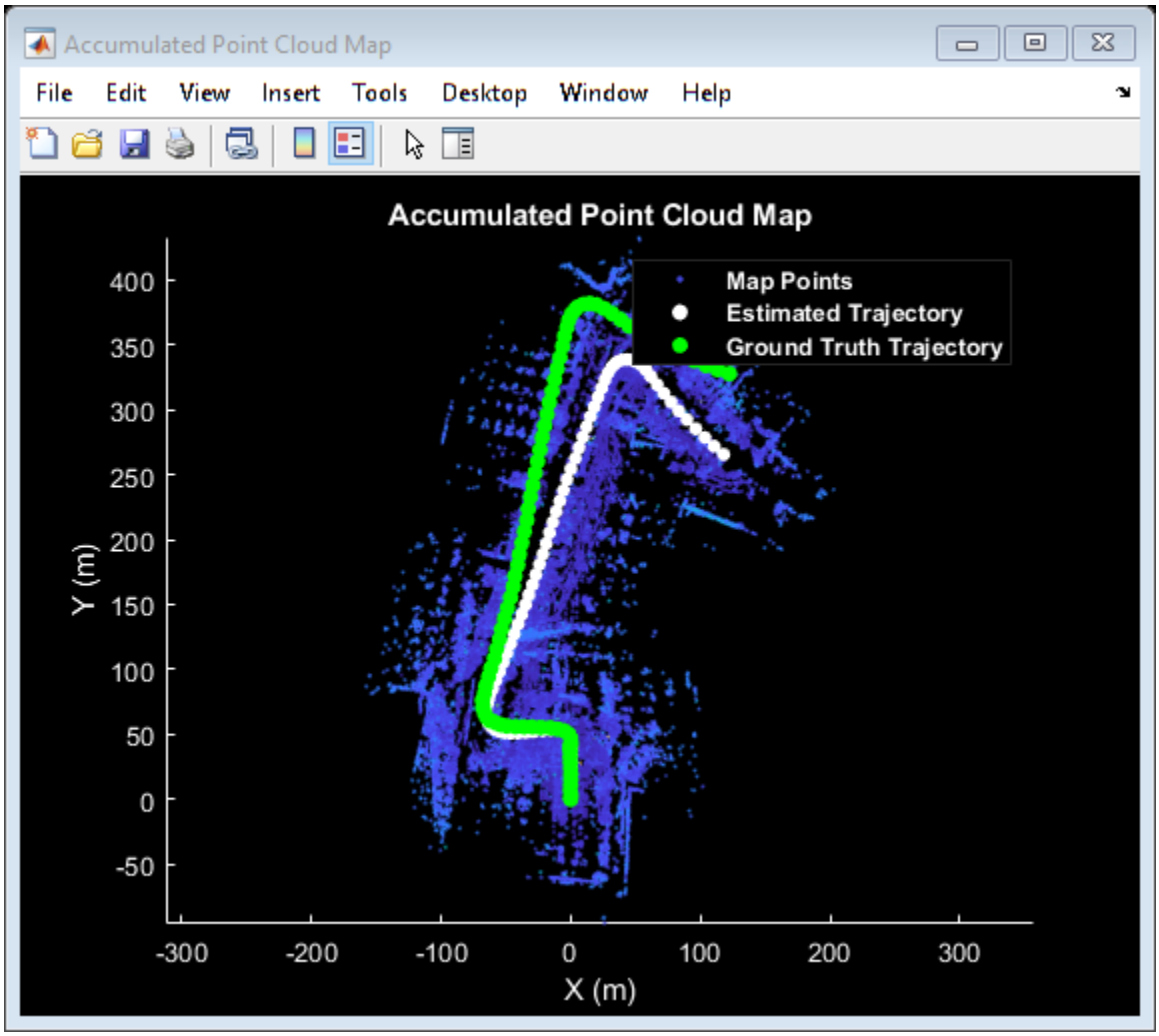
1. Convert the GPS coordinates to local Cartesian East-North-Up coordinates using the `latlon2local` function. The GPS location from the start of the trajectory is used as the reference point and defines the origin of the local x,y,z coordinate system.
2. Rotate the Cartesian coordinates so that the local coordinate system is aligned with the first lidar sensor coordinates. Since the exact mounting configuration of the lidar and GPS on the vehicle are not known, they are estimated.

```
% Select reference point as first GPS reading
origin = [gpsSequence.Latitude(1), gpsSequence.Longitude(1), gpsSequence.Altitude(1)];

% Convert GPS readings to a local East-North-Up coordinate system
[xEast, yNorth, zUp] = latlon2local(gpsSequence.Latitude, gpsSequence.Longitude, ...
```

```
    gpsSequence.Altitude, origin);

% Estimate rough orientation at start of trajectory to align local ENU
% system with lidar coordinate system
theta = median(atan2d(yNorth(1:15), xEast(1:15)));

R = [ cosd(90-theta) sind(90-theta) 0;
     -sind(90-theta) cosd(90-theta) 0;
     0               0              1];

% Rotate ENU coordinates to align with lidar coordinate system
groundTruthTrajectory = [xEast, yNorth, zUp] * R;
```

Superimpose the ground truth trajectory on the built map.

```
hold(mapBuilder.Axes, 'on')
scatter(mapBuilder.Axes, groundTruthTrajectory(:,1), groundTruthTrajectory(:,2), ...
    'green','filled');

helperAddLegend(mapBuilder.Axes, ...
    {'Map Points', 'Estimated Trajectory', 'Ground Truth Trajectory'});
```



After the initial turn, the estimated trajectory veers off the ground truth trajectory significantly. The trajectory estimated using point cloud registration alone can drift for a number of reasons:

- Noisy scans from the sensor without sufficient overlap
- Absence of strong enough features, for example, near long roads
- Inaccurate initial transformation, especially when rotation is significant.

```
% Close map display
updateDisplay(mapBuilder, true);
```

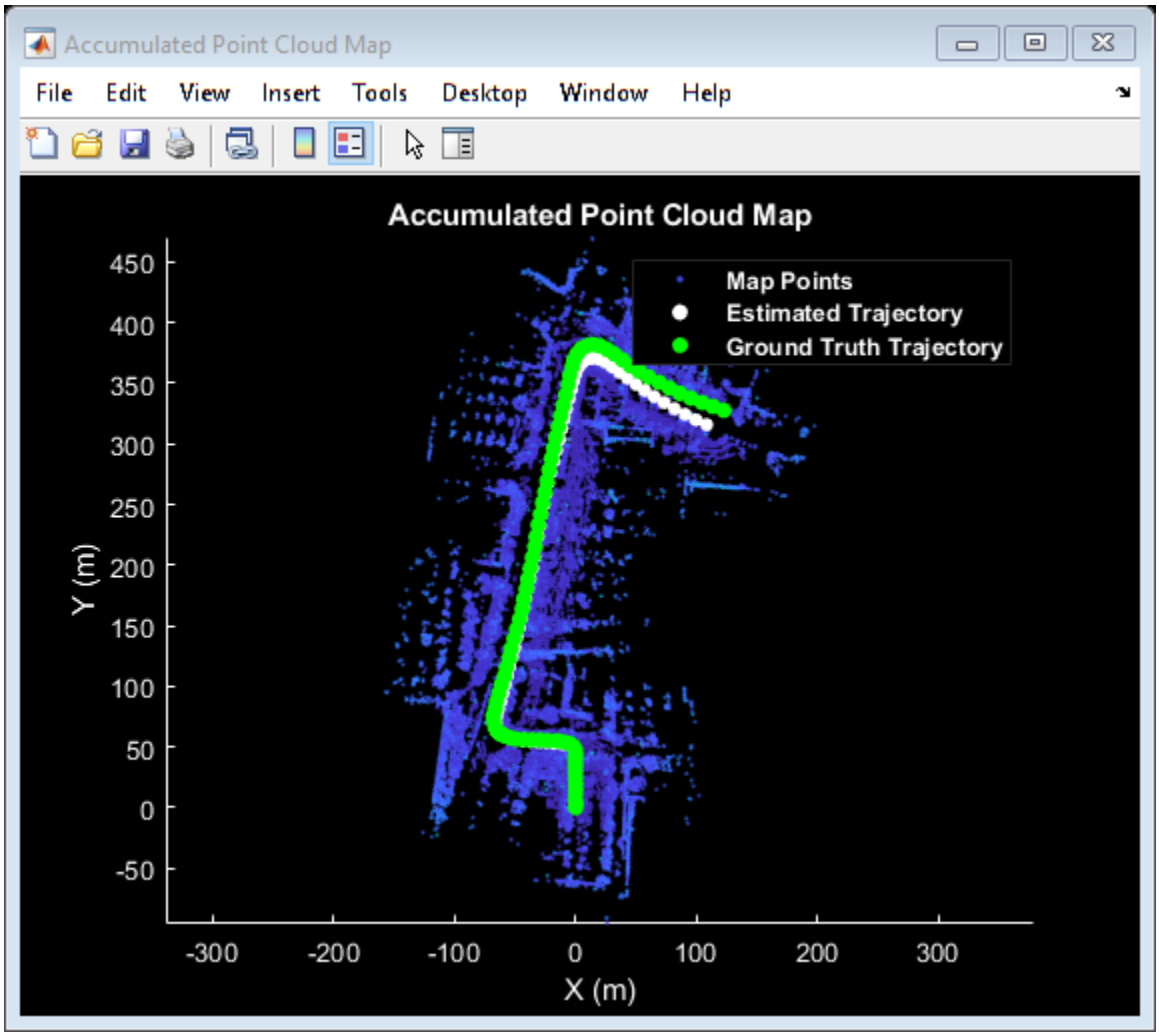
Use IMU Orientation to Improve Built Map

An IMU is an electronic device mounted on a platform. IMUs contain multiple sensors that report various information about the motion of the vehicle. Typical IMUs incorporate accelerometers, gyroscopes, and magnetometers. An IMU can provide a reliable measure of orientation.

Use the IMU readings to provide a better initial estimate for registration. The IMU-reported sensor readings used in this example have already been filtered on the device.

```
% Reset the map builder to clear previously built map
reset(mapBuilder);

% Set random number seed
rng(0);

initTform = rigid3d;
for n = 1 : skipFrames : numFrames - skipFrames

    % Get the nth point cloud
    ptCloud = lidarPointClouds.PointCloud(n);

    if n > 1
        % Since IMU sensor reports readings at a much faster rate, gather
        % IMU readings reported since the last lidar scan.
        prevTime = lidarPointClouds.Time(n - skipFrames);
        currTime = lidarPointClouds.Time(n);
        timeSinceScan = timerange(prevTime, currTime);

        imuReadings = imuOrientations(timeSinceScan, 'Orientation');

        % Form an initial estimate using IMU readings
        initTform = helperComputeInitialEstimateFromIMU(imuReadings, tform);
    end

    % Update map using the point cloud
    tform = updateMap(mapBuilder, ptCloud, initTform);

    % Update map display
    updateDisplay(mapBuilder, closeDisplay);
end


% Superimpose ground truth trajectory on new map
hold(mapBuilder.Axes, 'on')
scatter(mapBuilder.Axes, groundTruthTrajectory(:,1), groundTruthTrajectory(:,2), ...
    'green','filled');
```

```
helperAddLegend(mapBuilder.Axes, ...
    {'Map Points', 'Estimated Trajectory', 'Ground Truth Trajectory'});

% Capture snapshot for publishing
snapnow;

% Close open figures
close([hFigFixed, hFigAlign, hFigAccum]);
updateDisplay(mapBuilder, true);
```



Using the orientation estimate from IMU significantly improved registration, leading to a much closer trajectory with smaller drift.

Supporting Functions

`helperAlignPlayers` aligns a cell array of streaming players so they are arranged from left to right on the screen.

```
function helperAlignPlayers(players)

validateattributes(players, {'cell'}, {'vector'});
```

```
hasAxes = cellfun(@(p)isprop(p,'Axes'),players,'UniformOutput', true);
if ~all(hasAxes)
    error('Expected all viewers to have an Axes property');
end

screenSize   = get(groot, 'ScreenSize');
screenMargin = [50, 100];

playerSizes = cellfun(@getPlayerSize, players, 'UniformOutput', false);
playerSizes = cell2mat(playerSizes);

maxHeightInSet = max(playerSizes(1:3:end));

% Arrange players vertically so that the tallest player is 100 pixels from
% the top.
location = round([screenMargin(1), screenSize(4)-screenMargin(2)-maxHeightInSet]);
for n = 1 : numel(players)
    player = players{n};

    hFig = ancestor(player.Axes, 'figure');
    hFig.OuterPosition(1:2) = location;

    % Set up next location by going right
    location = location + [50+hFig.OuterPosition(3), 0];
end

    function sz = getPlayerSize(viewer)

        % Get the parent figure container
        h = ancestor(viewer.Axes, 'figure');

        sz = h.OuterPosition(3:4);
    end
end
```

`helperVisualizeEgoView` visualizes point cloud data in the ego perspective by rotating about the center.

```
function player = helperVisualizeEgoView(ptCloud)

% Create a pcplayer object
xlimits = ptCloud.XLimits;
ylimits = ptCloud.YLimits;
zlimits = ptCloud.ZLimits;

player = pcplayer(xlimits, ylimits, zlimits);

% Turn off axes lines
axis(player.Axes, 'off');

% Set up camera to show ego view
camproj(player.Axes, 'perspective');
camva(player.Axes, 90);
campos(player.Axes, [0 0 0]);
camtarget(player.Axes, [-1 0 0]);

% Set up a transformation to rotate by 5 degrees
```

```
theta = 5;
R = [ cosd(theta) sind(theta) 0 0
     -sind(theta) cosd(theta) 0 0
      0           0           1 0
      0           0           0 1];
rotateByTheta = rigid3d(R);

for n = 0 : theta : 359
    % Rotate point cloud by theta
    ptCloud = pctransform(ptCloud, rotateByTheta);

    % Display point cloud
    view(player, ptCloud);

    pause(0.05)
end
end
```

`helperProcessPointCloud` processes a point cloud by removing points belonging to the ground plane or ego vehicle.

```
function ptCloudProcessed = helperProcessPointCloud(ptCloud)

% Check if the point cloud is organized
isOrganized = ~ismatrix(ptCloud.Location);

% If the point cloud is organized, use range-based flood fill algorithm
% (segmentGroundFromLidarData). Otherwise, use plane fitting.
groundSegmentationMethods = ["planefit", "rangefloodfill"];
method = groundSegmentationMethods(isOrganized+1);

if method == "planefit"
    % Segment ground as the dominant plane, with reference normal vector
    % pointing in positive z-direction, using pcfitplane. For organized
    % point clouds, consider using segmentGroundFromLidarData instead.
    maxDistance    = 0.4;       % meters
    maxAngDistance = 5;         % degrees
    refVector      = [0, 0, 1]; % z-direction

    [~,groundIndices] = pcfitplane(ptCloud, maxDistance, refVector, maxAngDistance);
elseif method == "rangefloodfill"
    % Segment ground using range-based flood fill.
    groundIndices = segmentGroundFromLidarData(ptCloud);
else
    error("Expected method to be 'planefit' or 'rangefloodfill'")
end

% Segment ego vehicle as points within a given radius of sensor
sensorLocation = [0, 0, 0];
radius         = 3.5;

egoIndices  = findNeighborsInRadius(ptCloud, sensorLocation, radius);

% Remove points belonging to ground or ego vehicle
ptsToKeep = true(ptCloud.Count, 1);
ptsToKeep(groundIndices) = false;
ptsToKeep(egoIndices)    = false;
```

```
% If the point cloud is organized, retain organized structure
if isOrganized
    ptCloudProcessed = select(ptCloud, find(ptsToKeep), 'OutputSize', 'full');
else
    ptCloudProcessed = select(ptCloud, find(ptsToKeep));
end
end
```

helperComputeInitialEstimateFromIMU estimates an initial transformation for NDT using IMU orientation readings and previously estimated transformation.

```
function tform = helperComputeInitialEstimateFromIMU(imuReadings, prevTform)

% Initialize transformation using previously estimated transform
tform = prevTform;

% If no IMU readings are available, return
if height(imuReadings) <= 1
    return;
end

% IMU orientation readings are reported as quaternions representing the
% rotational offset to the body frame. Compute the orientation change
% between the first and last reported IMU orientations during the interval
% of the lidar scan.
q1 = imuReadings.Orientation(1);
q2 = imuReadings.Orientation(end);

% Compute rotational offset between first and last IMU reading by
%   - Rotating from q2 frame to body frame
%   - Rotating from body frame to q1 frame
q = q1 * conj(q2);

% Convert to Euler angles
yawPitchRoll = euler(q, 'ZYX', 'point');

% Discard pitch and roll angle estimates. Use only heading angle estimate
% from IMU orientation.
yawPitchRoll(2:3) = 0;

% Convert back to rotation matrix
q = quaternion(yawPitchRoll, 'euler', 'ZYX', 'point');
R = rotmat(q, 'point');

% Use computed rotation
tform.T(1:3, 1:3) = R';
end
```

helperAddLegend adds a legend to the axes.

```
function helperAddLegend(hAx, labels)

% Add a legend to the axes
hLegend = legend(hAx, labels{:});

% Set text color and font weight
hLegend.TextColor  = [1 1 1];
```

```
hLegend.FontWeight = 'bold';
end
```

helperMakeFigurePublishFriendly adjusts figures so that screenshot captured by publish is correct.

```
function helperMakeFigurePublishFriendly(hFig)

if ~isempty(hFig) && isvalid(hFig)
    hFig.HandleVisibility = 'callback';
end

end
```

See Also

Functions
`pcmerge` | `pcregistericp` | `pcregisterndt`

Objects
`geoplayer` | `pcplayer` | `pointCloud`

More About

- “Build a Map from Lidar Data Using SLAM” on page 7-559
- “Lidar Localization with Unreal Engine Simulation” on page 7-701
- “Ground Plane and Obstacle Detection Using Lidar” on page 7-107

External Websites

- Udacity Self-Driving Car Data Subset (MathWorks GitHub repository)

Build a Map from Lidar Data Using SLAM

This example shows how to process 3-D lidar data from a sensor mounted on a vehicle to progressively build a map and estimate the trajectory of a vehicle using simultaneous localization and mapping (SLAM). In addition to 3-D lidar data, an inertial navigation sensor (INS) is also used to help build the map. Maps built this way can facilitate path planning for vehicle navigation or can be used for localization.

Overview

The “Build a Map from Lidar Data” on page 7-539 example uses 3-D lidar data and IMU readings to progressively build a map of the environment traversed by a vehicle. While this approach builds a locally consistent map, it is suitable only for mapping small areas. Over longer sequences, the drift accumulates into a significant error. To overcome this limitation, this example recognizes previously visited places and tries to correct for the accumulated drift using the graph SLAM approach.

Load and Explore Recorded Data

The data used in this example is part of the Velodyne SLAM Dataset, and represents close to 6 minutes of recorded data. Download the data to a temporary directory.

Note: This download can take a few minutes.

```
baseDownloadURL = 'https://www.mrt.kit.edu/z/publ/download/velodyneslam/data/scenario1.zip';
dataFolder      = fullfile(tempdir, 'kit_velodyneslam_data_scenario1', filesep);
options         = weboptions('Timeout', Inf);

zipFileName  = dataFolder + "scenario1.zip";

folderExists = exist(dataFolder, 'dir');
if ~folderExists
    % Create a folder in a temporary directory to save the downloaded zip
    % file.
    mkdir(dataFolder);

    disp('Downloading scenario1.zip (153 MB) ...')
    websave(zipFileName, baseDownloadURL, options);

    % Unzip downloaded file
    unzip(zipFileName, dataFolder);
end
```

```
Downloading scenario1.zip (153 MB) ...
```

Use the `helperReadDataset` function to read data from the created folder in the form of a `timetable`. The point clouds captured by the lidar are stored in the form of PNG image files. Extract the list of point cloud file names in the `pointCloudTable` variable. To read the point cloud data from the image file, use the `helperReadPointCloudFromFile` function. This function takes an image file name and returns a `pointCloud` (Computer Vision Toolbox) object. The INS readings are read directly from a configuration file and stored in the `insDataTable` variable.

```
datasetTable = helperReadDataset(dataFolder);

pointCloudTable = datasetTable(:, 1);
insDataTable    = datasetTable(:, 2:end);
```

Read the first point cloud and display it at the MATLAB® command prompt

```
ptCloud = helperReadPointCloudFromFile(pointCloudTable.PointCloudFileName{1});
disp(ptCloud)
```

```
  pointCloud with properties:

     Location: [64×870×3 single]
        Count: 55680
      XLimits: [-78.4980 77.7062]
      YLimits: [-76.8795 74.7502]
      ZLimits: [-2.4839 2.6836]
        Color: []
       Normal: []
    Intensity: []
```

Display the first INS reading. The `timetable` holds `Heading`, `Pitch`, `Roll`, `X`, `Y`, and `Z` information from the INS.

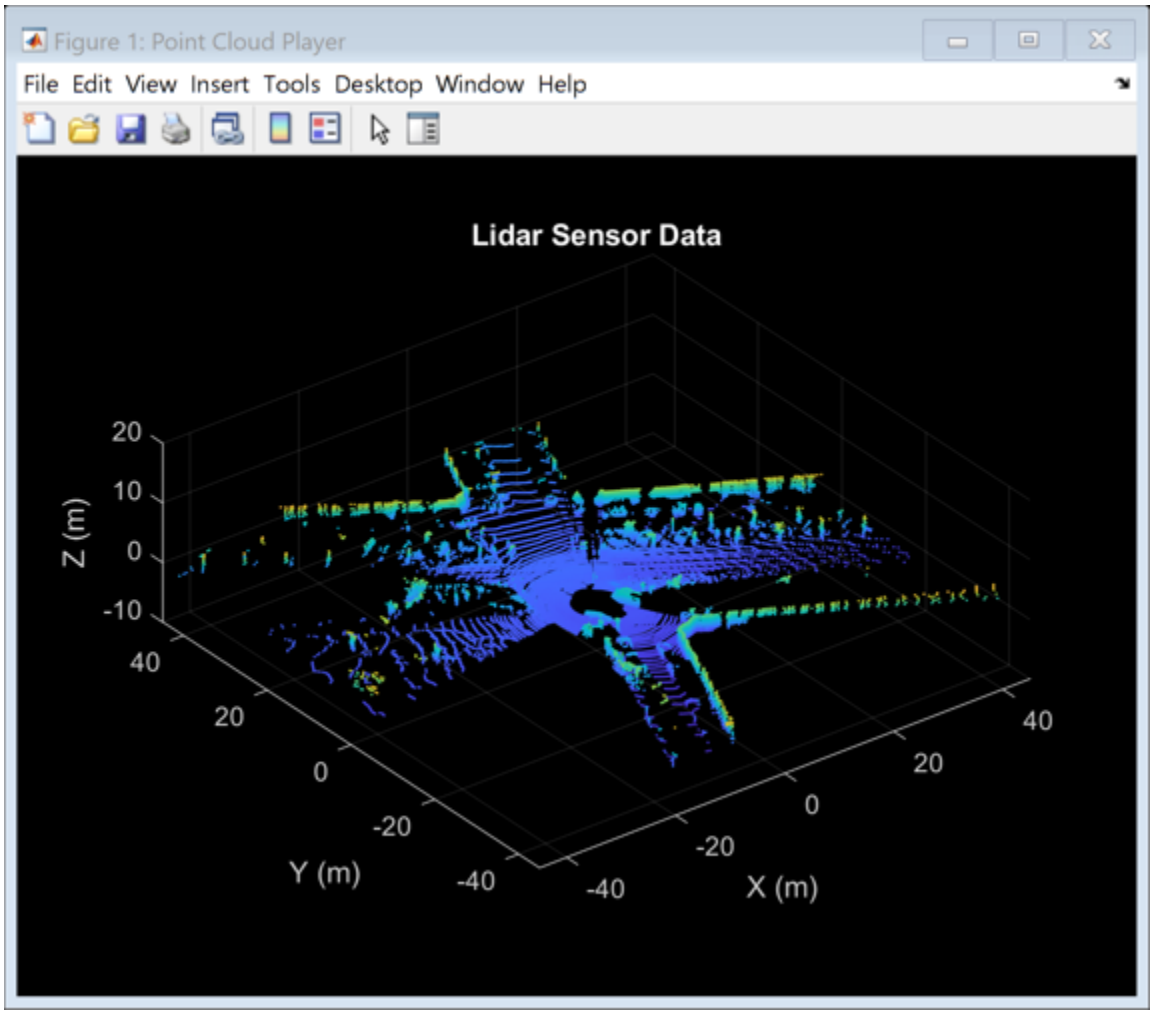
```
disp(insDataTable(1, :))
```

Timestamps	Heading	Pitch	Roll	X	Y	Z
13-Jun-2010 06:27:31	1.9154	0.007438	-0.019888	-2889.1	-2183.9	116.47

Visualize the point clouds using `pcplayer` (Computer Vision Toolbox), a streaming point cloud display. The vehicle traverses a path consisting of two loops. In the first loop, the vehicle makes a series of turns and returns to the starting point. In the second loop, the vehicle makes a series of turns along another route and again returns to the starting point.

```
% Specify limits of the player
xlimits = [-45 45]; % meters
ylimits = [-45 45];
zlimits = [-10 20];

% Create a streaming point cloud display object
lidarPlayer = pcplayer(xlimits, ylimits, zlimits);

% Customize player axes labels
xlabel(lidarPlayer.Axes, 'X (m)')
ylabel(lidarPlayer.Axes, 'Y (m)')
zlabel(lidarPlayer.Axes, 'Z (m)')

title(lidarPlayer.Axes, 'Lidar Sensor Data')

% Skip evey other frame since this is a long sequence
skipFrames  = 2;
numFrames   = height(pointCloudTable);
for n = 1 : skipFrames : numFrames

    % Read a point cloud
    fileName = pointCloudTable.PointCloudFileName{n};
    ptCloud = helperReadPointCloudFromFile(fileName);

    % Visualize point cloud
    view(lidarPlayer, ptCloud);
```

```
    pause(0.01)
end
```



Build a Map Using Odometry

First, use the approach explained in the “Build a Map from Lidar Data” on page 7-539 example to build a map. The approach consists of the following steps:

- **Align lidar scans:** Align successive lidar scans using a point cloud registration technique. This example uses `pcregisterndt` (Computer Vision Toolbox) for registering scans. By successively composing these transformations, each point cloud is transformed back to the reference frame of the first point cloud.
- **Combine aligned scans:** Generate a map by combining all the transformed point clouds.

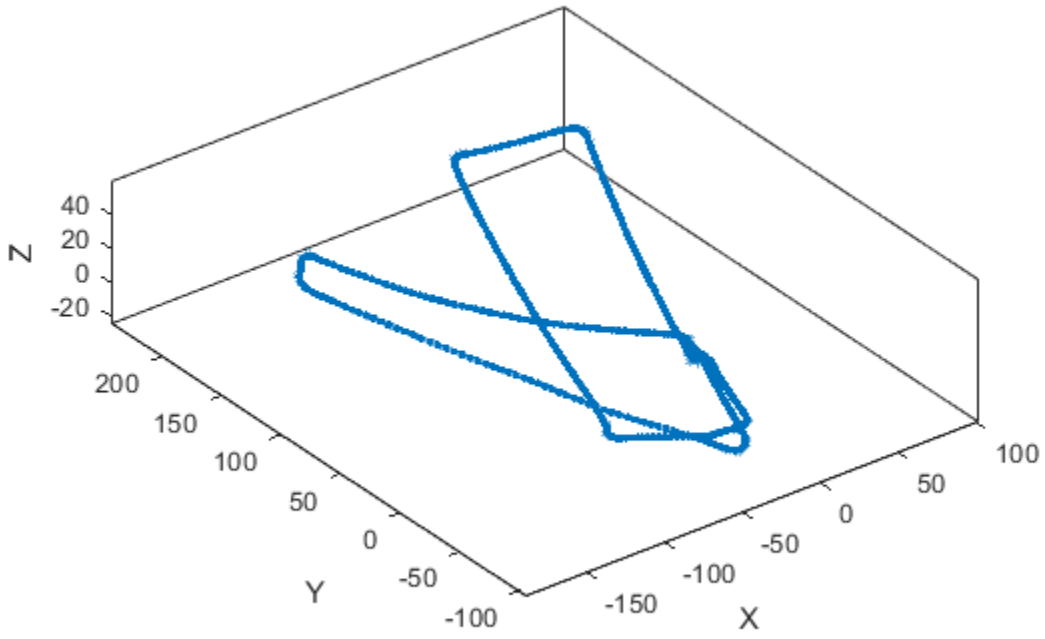
This approach of incrementally building a map and estimating the trajectory of the vehicle is called *odometry*.

Use a `pcviewset` (Computer Vision Toolbox) object to store and manage data across multiple views. A view set consists of a set of connected views.

- Each view stores information associated with a single view. This information includes the absolute pose of the view, the point cloud sensor data captured at that view, and a unique identifier for the view. Add views to the view set using `addView` (Computer Vision Toolbox).
- To establish a connection between views use `addConnection` (Computer Vision Toolbox). A connection stores information like the relative transformation between the connecting views, the uncertainty involved in computing this measurement (represented as an information matrix) and the associated view identifiers.
- Use the `plot` (Computer Vision Toolbox) method to visualize the connections established by the view set. These connections can be used to visualize the path traversed by the vehicle.

```
hide(lidarPlayer)

% Set random seed to ensure reproducibility
rng(0);

% Create an empty view set
vSet = pcviewset;

% Create a figure for view set display
hFigBefore = figure('Name', 'View Set Display');
hAxBefore = axes(hFigBefore);

% Initialize point cloud processing parameters
downsamplePercent = 0.1;
regGridSize       = 3;

% Initialize transformations
absTform   = rigid3d;  % Absolute transformation to reference frame
relTform   = rigid3d;  % Relative transformation between successive scans

viewId = 1;
skipFrames  = 5;
numFrames   = height(pointCloudTable);
displayRate = 100;      % Update display every 100 frames
for n = 1 : skipFrames : numFrames

    % Read point cloud
    fileName = pointCloudTable.PointCloudFileName{n};
    ptCloudOrig = helperReadPointCloudFromFile(fileName);

    % Process point cloud
    %   - Segment and remove ground plane
    %   - Segment and remove ego vehicle
    ptCloud = helperProcessPointCloud(ptCloudOrig);

    % Downsample the processed point cloud
    ptCloud = pcdownsample(ptCloud, "random", downsamplePercent);

    firstFrame = (n==1);
    if firstFrame
        % Add first point cloud scan as a view to the view set
        vSet = addView(vSet, viewId, absTform, "PointCloud", ptCloudOrig);

        viewId = viewId + 1;
        ptCloudPrev = ptCloud;
        continue;
```

```
    end

    % Use INS to estimate an initial transformation for registration
    initTform = helperComputeInitialEstimateFromINS(relTform, ...
        insDataTable(n-skipFrames:n, :));

    % Compute rigid transformation that registers current point cloud with
    % previous point cloud
    relTform = pcregisterndt(ptCloud, ptCloudPrev, regGridSize, ...
        "InitialTransform", initTform);

    % Update absolute transformation to reference frame (first point cloud)
    absTform = rigid3d( relTform.T * absTform.T );

    % Add current point cloud scan as a view to the view set
    vSet = addView(vSet, viewId, absTform, "PointCloud", ptCloudOrig);

    % Add a connection from the previous view to the current view, representing
    % the relative transformation between them
    vSet = addConnection(vSet, viewId-1, viewId, relTform);

    viewId = viewId + 1;

    ptCloudPrev = ptCloud;
    initTform   = relTform;

    if n>1 && mod(n, displayRate) == 1
        plot(vSet, "Parent", hAxBefore);
        drawnow update
    end
end
```



The view set object `vSet`, now holds views and connections. In the Views table of vSet, the `AbsolutePose` variable specifies the absolute pose of each view with respect to the first view. In the `Connections` table of `vSet`, the `RelativePose` variable specifies relative constraints between the connected views, the `InformationMatrix` variable specifies, for each edge, the uncertainty associated with a connection.

```
% Display the first few views and connections
head(vSet.Views)
head(vSet.Connections)
```

```
ans =

  8×3 table

    ViewId    AbsolutePose        PointCloud
    ______    ____________    _________________

      1       [1×1 rigid3d]    [1×1 pointCloud]
      2       [1×1 rigid3d]    [1×1 pointCloud]
      3       [1×1 rigid3d]    [1×1 pointCloud]
      4       [1×1 rigid3d]    [1×1 pointCloud]
      5       [1×1 rigid3d]    [1×1 pointCloud]
      6       [1×1 rigid3d]    [1×1 pointCloud]
      7       [1×1 rigid3d]    [1×1 pointCloud]
      8       [1×1 rigid3d]    [1×1 pointCloud]
```

```
ans =

  8×4 table

    ViewId1    ViewId2    RelativePose     InformationMatrix
    _______    _______    ____________    __________________

       1          2       [1×1 rigid3d]      {6×6 double}
       2          3       [1×1 rigid3d]      {6×6 double}
       3          4       [1×1 rigid3d]      {6×6 double}
       4          5       [1×1 rigid3d]      {6×6 double}
       5          6       [1×1 rigid3d]      {6×6 double}
       6          7       [1×1 rigid3d]      {6×6 double}
       7          8       [1×1 rigid3d]      {6×6 double}
       8          9       [1×1 rigid3d]      {6×6 double}
```

Now, build a point cloud map using the created view set. Align the view absolute poses with the point clouds in the view set using `pcalign`. Specify a grid size to control the resolution of the map. The map is returned as a `pointCloud` object.

```
ptClouds = vSet.Views.PointCloud;
absPoses = vSet.Views.AbsolutePose;
mapGridSize = 0.2;
ptCloudMap = pcalign(ptClouds, absPoses, mapGridSize);
```

Notice that the path traversed using this approach drifts over time. While the path along the first loop back to the starting point seems reasonable, the second loop drifts significantly from the starting point. The accumulated drift results in the second loop terminating several meters away from the starting point.

A map built using odometry alone is inaccurate. Display the built point cloud map with the traversed path. Notice that the map and traversed path for the second loop are not consistent with the first loop.

```
hold(hAxBefore, 'on');
pcshow(ptCloudMap);
hold(hAxBefore, 'off');

close(hAxBefore.Parent)
```

Correct Drift Using Pose Graph Optimization

Graph SLAM is a widely used technique for resolving the drift in odometry. The graph SLAM approach incrementally creates a graph, where nodes correspond to vehicle poses and edges represent sensor measurements constraining the connected poses. Such a graph is called a *pose graph*. The pose graph contains edges that encode contradictory information, due to noise or inaccuracies in measurement. The nodes in the constructed graph are then optimized to find the set of vehicle poses that optimally explain the measurements. This technique is called *pose graph optimization*.

To create a pose graph from a view set, you can use the `createPoseGraph` (Computer Vision Toolbox) function. This function creates a node for each view, and an edge for each connection in the view set. To optimize the pose graph, you can use the `optimizePoseGraph` (Navigation Toolbox) function.

A key aspect contributing to the effectiveness of graph SLAM in correcting drift is the accurate detection of loops, that is, places that have been previously visited. This is called *loop closure detection* or *place recognition*. Adding edges to the pose graph corresponding to loop closures provides a contradictory measurement for the connected node poses, which can be resolved during pose graph optimization.

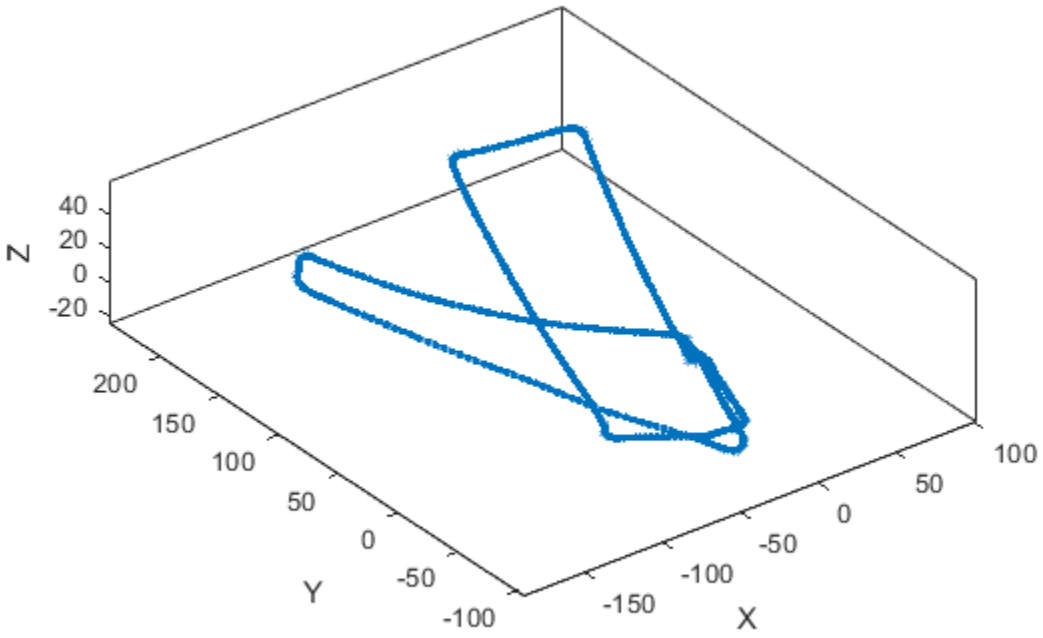
Loop closures can be detected using descriptors that characterize the local environment visible to the Lidar sensor. The *Scan Context* descriptor [1] is one such descriptor that can be computed from a point cloud using the `scanContextDescriptor` (Computer Vision Toolbox) function. This example uses the `helperLoopClosureDetector` class to detect loop closures. This class uses scan context descriptors for describing individual point clouds, and a two-phase descriptor search algorithm.

- **Descriptor Computation:** A scan context descriptor and a ring key are extracted from each point cloud using the `scanContextDescriptor` function.
- **Descriptor Matching:** In the first phase, the ring key is used to find potential loop candidates. In the second phase, a nearest neighbor search is performed using the scan context feature descriptors among the potential loop candidates. The `scanContextDistance` (Computer Vision Toolbox) function is used to compute the distance between two scan context descriptors. See the `helperFeatureSearcher` class for more details.

```
% Set random seed to ensure reproducibility
rng(0);

% Create an empty view set
vSet = pcviewset;

% Create a loop closure detector
matchThresh = 0.08;
loopDetector = helperLoopClosureDetector('MatchThreshold', matchThresh);

% Create a figure for view set display
hFigBefore = figure('Name', 'View Set Display');
hAxBefore = axes(hFigBefore);

% Initialize transformations
absTform   = rigid3d;  % Absolute transformation to reference frame
relTform   = rigid3d;  % Relative transformation between successive scans

maxTolerableRMSE  = 3; % Maximum allowed RMSE for a loop closure candidate to be accepted

viewId = 1;
for n = 1 : skipFrames : numFrames

    % Read point cloud
    fileName = pointCloudTable.PointCloudFileName{n};
    ptCloudOrig = helperReadPointCloudFromFile(fileName);

    % Process point cloud
    %   - Segment and remove ground plane
    %   - Segment and remove ego vehicle
    ptCloud = helperProcessPointCloud(ptCloudOrig);

    % Downsample the processed point cloud
    ptCloud = pcdownsample(ptCloud, "random", downsamplePercent);

    firstFrame = (n==1);
```

```
if firstFrame
    % Add first point cloud scan as a view to the view set
    vSet = addView(vSet, viewId, absTform, "PointCloud", ptCloudOrig);

    viewId = viewId + 1;
    ptCloudPrev = ptCloud;
    continue;
end

% Use INS to estimate an initial transformation for registration
initTform = helperComputeInitialEstimateFromINS(relTform, ...
    insDataTable(n-skipFrames:n, :));

% Compute rigid transformation that registers current point cloud with
% previous point cloud
relTform = pcregisterndt(ptCloud, ptCloudPrev, regGridSize, ...
    "InitialTransform", initTform);

% Update absolute transformation to reference frame (first point cloud)
absTform = rigid3d( relTform.T * absTform.T );

% Add current point cloud scan as a view to the view set
vSet = addView(vSet, viewId, absTform, "PointCloud", ptCloudOrig);

% Add a connection from the previous view to the current view representing
% the relative transformation between them
vSet = addConnection(vSet, viewId-1, viewId, relTform);

% Detect loop closure candidates
[loopFound, loopViewId] = detectLoop(loopDetector, ptCloudOrig);

% A loop candidate was found
if loopFound
    loopViewId = loopViewId(1);

    % Retrieve point cloud from view set
    ptCloudOrig = vSet.Views.PointCloud( find(vSet.Views.ViewId == loopViewId, 1) );

    % Process point cloud
    ptCloudOld = helperProcessPointCloud(ptCloudOrig);

    % Downsample point cloud
    ptCloudOld = pcdownsample(ptCloudOld, "random", downsamplePercent);

    % Use registration to estimate the relative pose
    [relTform, ~, rmse] = pcregisterndt(ptCloud, ptCloudOld, ...
        regGridSize, "MaxIterations", 50);

    acceptLoopClosure = rmse <= maxTolerableRMSE;
    if acceptLoopClosure
        % For simplicity, use a constant, small information matrix for
        % loop closure edges
        infoMat = 0.01 * eye(6);

        % Add a connection corresponding to a loop closure
        vSet = addConnection(vSet, loopViewId, viewId, relTform, infoMat);
    end
end
```

```
    viewId = viewId + 1;

    ptCloudPrev = ptCloud;
    initTform   = relTform;

    if n>1 && mod(n, displayRate) == 1
        hG = plot(vSet, "Parent", hAxBefore);
        drawnow update
    end
end
```



Create a pose graph from the view set by using the `createPoseGraph` (Computer Vision Toolbox) method. The pose graph is a `digraph` object with:

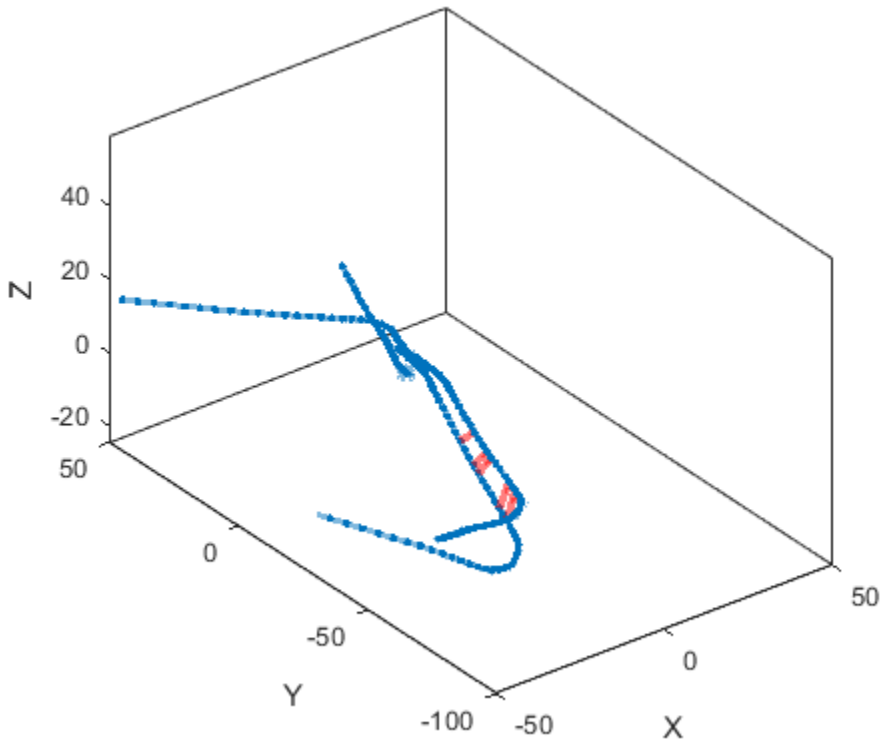
- Nodes containing the absolute pose of each view
- Edges containing the relative pose constraints of each connection

```
G = createPoseGraph(vSet);
disp(G)
```

```
  digraph with properties:

    Edges: [507×3 table]
    Nodes: [503×2 table]
```

In addition to the odometry connections between successive views, the view set now includes loop closure connections. For example, notice the new connections between the second loop traversal and the first loop traversal. These are loop closure connections. These can be identified as edges in the graph whose end nodes are not consecutive.

```
% Update axes limits to focus on loop closure connections
xlim(hAxBefore, [-50 50]);
ylim(hAxBefore, [-100 50]);

% Find and highlight loop closure connections
loopEdgeIds = find(abs(diff(G.Edges.EndNodes, 1, 2)) > 1);
highlight(hG, 'Edges', loopEdgeIds, 'EdgeColor', 'red', 'LineWidth', 3)
```

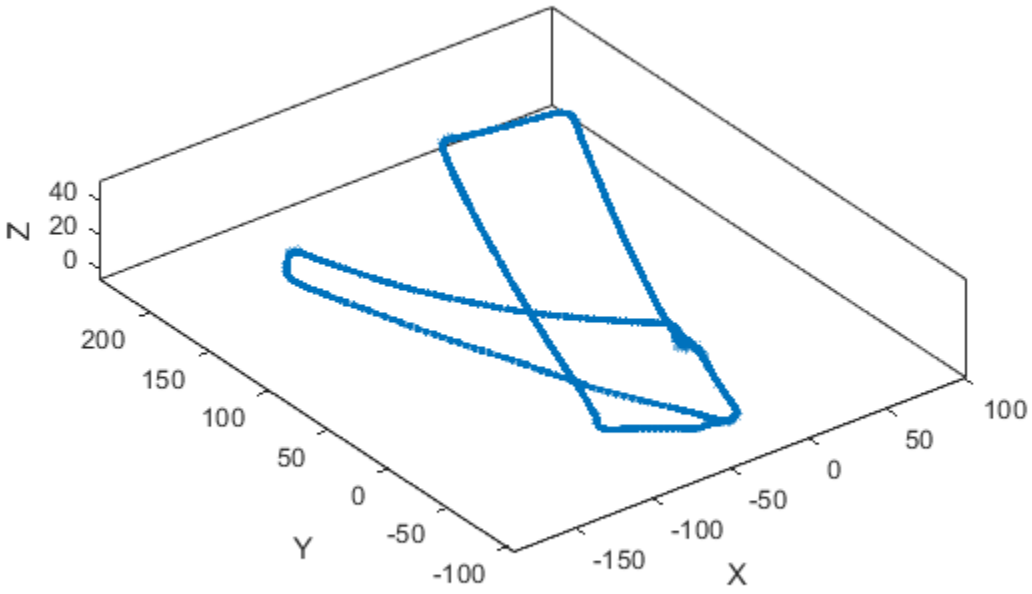


Optimize the pose graph using `optimizePoseGraph`.

```
optimG = optimizePoseGraph(G, 'g2o-levenberg-marquardt');

vSetOptim = updateView(vSet, optimG.Nodes);
```

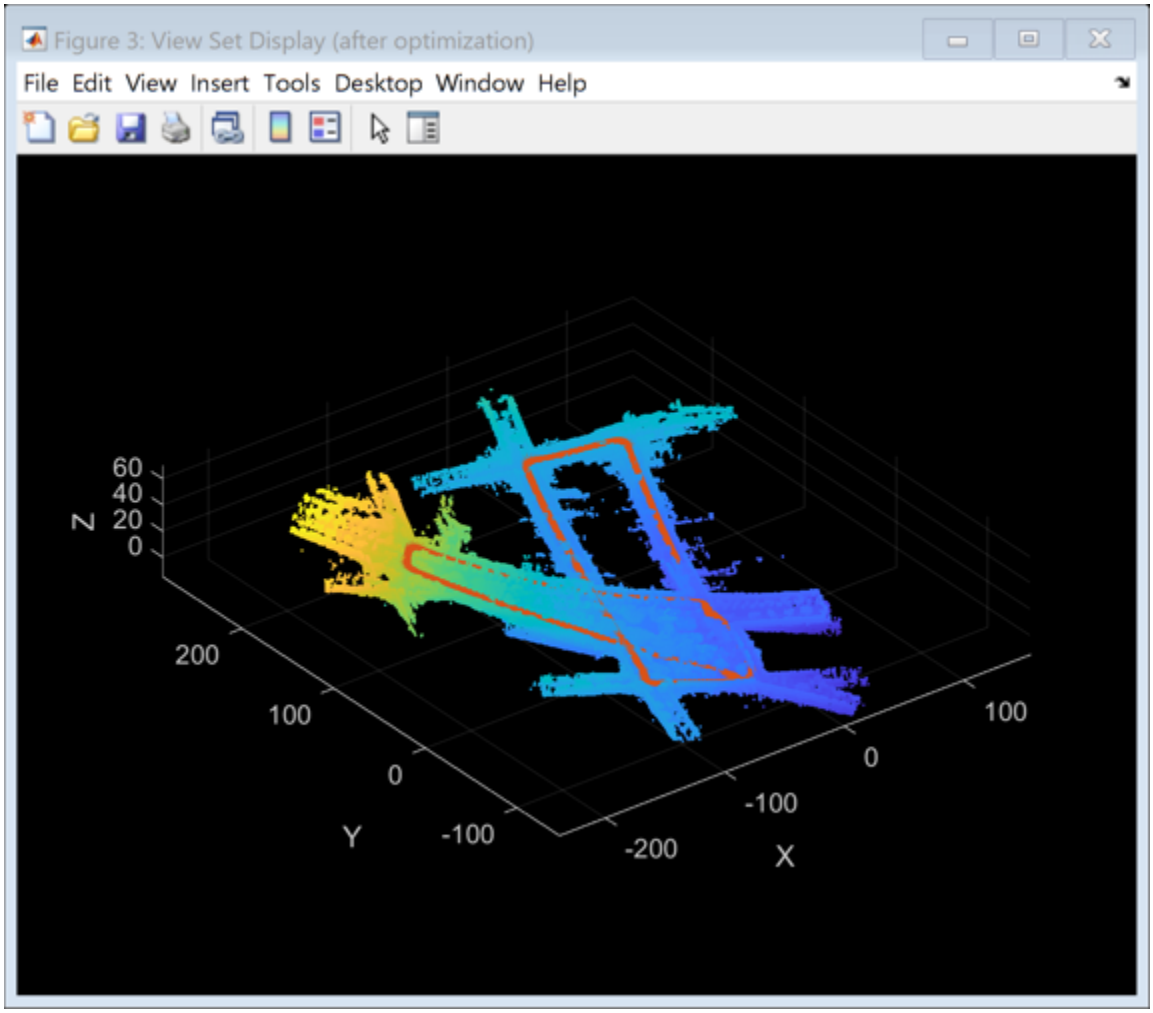
Display the view set with optimized poses. Notice that the detected loops are now merged, resulting in a more accurate trajectory.

```
plot(vSetOptim, 'Parent', hAxBefore)
```



The absolute poses in the optimized view set can now be used to build a more accurate map. Use the `pcalign` (Computer Vision Toolbox) function to align the view set point clouds with the optimized view set absolute poses into a single point cloud map. Specify a grid size to control the resolution of the created point cloud map.

```
mapGridSize = 0.2;
ptClouds = vSetOptim.Views.PointCloud;
absPoses = vSetOptim.Views.AbsolutePose;
ptCloudMap = pcalign(ptClouds, absPoses, mapGridSize);

hFigAfter = figure('Name', 'View Set Display (after optimization)');
hAxAfter = axes(hFigAfter);
pcshow(ptCloudMap, 'Parent', hAxAfter);

% Overlay view set display
hold on
plot(vSetOptim, 'Parent', hAxAfter);

helperMakeFigurePublishFriendly(hFigAfter);
```



While accuracy can still be improved, this point cloud map is significantly more accurate.

References

1 G. Kim and A. Kim, "Scan Context: Egocentric Spatial Descriptor for Place Recognition Within 3D Point Cloud Map," *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Madrid, 2018, pp. 4802-4809.

Supporting Functions and Classes

`helperReadDataset` reads data from specified folder into a timetable.

```
function datasetTable = helperReadDataset(dataFolder)
%helperReadDataset Read Velodyne SLAM Dataset data into a timetable
%   datasetTable = helperReadDataset(dataFolder) reads data from the
%   folder specified in dataFolder into a timetable. The function
%   expects data from the Velodyne SLAM Dataset.
%
%   See also fileDatastore, helperReadINSConfigFile.

% Point clouds are stored as PNG files in the scenario1 folder
pointCloudFilePattern = fullfile(dataFolder, 'scenario1', 'scan*.png');
```

```
% Create a file datastore to read in files in the right order
fileDS = fileDatastore(pointCloudFilePattern, 'ReadFcn', ...
    @helperReadPointCloudFromFile);

% Extract the file list from the datastore
pointCloudFiles = fileDS.Files;

imuConfigFile = fullfile(dataFolder, 'scenario1', 'imu.cfg');
insDataTable = helperReadINSConfigFile(imuConfigFile);

% Delete the bad row from the INS config file
insDataTable(1447, :) = [];

% Remove columns that will not be used
datasetTable = removevars(insDataTable, ...
    {'Num_Satellites', 'Latitude', 'Longitude', 'Altitude', 'Omega_Heading', ...
     'Omega_Pitch', 'Omega_Roll', 'V_X', 'V_Y', 'V_ZDown'});

datasetTable = addvars(datasetTable, pointCloudFiles, 'Before', 1, ...
    'NewVariableNames', "PointCloudFileName");
end
```

`helperProcessPointCloud` processes a point cloud by removing points belonging to the ground plane and the ego vehicle.

```
function ptCloud = helperProcessPointCloud(ptCloudIn, method)
%helperProcessPointCloud Process pointCloud to remove ground and ego vehicle
%   ptCloud = helperProcessPointCloud(ptCloudIn, method) processes
%   ptCloudIn by removing the ground plane and the ego vehicle.
%   method can be "planefit" or "rangefloodfill".
%
%   See also pcfitplane, pointCloud/findNeighborsInRadius.

arguments
    ptCloudIn (1,1) pointCloud
    method          string      {mustBeMember(method, ["planefit","rangefloodfill"])} = "rangeflo
end

isOrganized = ~ismatrix(ptCloudIn.Location);

if (method=="rangefloodfill" && isOrganized)
    % Segment ground using floodfill on range image
    groundFixedIdx = segmentGroundFromLidarData(ptCloudIn, ...
        "ElevationAngleDelta", 11);
else
    % Segment ground as the dominant plane with reference normal
    % vector pointing in positive z-direction
    maxDistance         = 0.4;
    maxAngularDistance  = 5;
    referenceVector     = [0 0 1];

    [~, groundFixedIdx] = pcfitplane(ptCloudIn, maxDistance, ...
    referenceVector, maxAngularDistance);
end

if isOrganized
    groundFixed = false(size(ptCloudIn.Location,1),size(ptCloudIn.Location,2));
```

```
else
    groundFixed = false(ptCloudIn.Count, 1);
end
groundFixed(groundFixedIdx) = true;

% Segment ego vehicle as points within a given radius of sensor
sensorLocation = [0 0 0];
radius = 3.5;
egoFixedIdx = findNeighborsInRadius(ptCloudIn, sensorLocation, radius);

if isOrganized
    egoFixed = false(size(ptCloudIn.Location,1),size(ptCloudIn.Location,2));
else
    egoFixed = false(ptCloudIn.Count, 1);
end
egoFixed(egoFixedIdx) = true;

% Retain subset of point cloud without ground and ego vehicle
if isOrganized
    indices = ~groundFixed & ~egoFixed;
else
    indices = find(~groundFixed & ~egoFixed);
end

ptCloud = select(ptCloudIn, indices);
end
```

helperComputeInitialEstimateFromINS estimates an initial transformation for registration from INS readings.

```
function initTform = helperComputeInitialEstimateFromINS(initTform, insData)

% If no INS readings are available, return
if isempty(insData)
    return;
end

% The INS readings are provided with X pointing to the front, Y to the left
% and Z up. Translation below accounts for transformation into the lidar
% frame.
insToLidarOffset = [0 -0.79 -1.73]; % See DATAFORMAT.txt
Tnow = [-insData.Y(end), insData.X(end), insData.Z(end)].' + insToLidarOffset';
Tbef = [-insData.Y(1)  , insData.X(1)  , insData.Z(1)].' + insToLidarOffset';

% Since the vehicle is expected to move along the ground, changes in roll
% and pitch are minimal. Ignore changes in roll and pitch, use heading only.
Rnow = rotmat(quaternion([insData.Heading(end) 0 0], 'euler', 'ZYX', 'point'), 'point');
Rbef = rotmat(quaternion([insData.Heading(1)   0 0], 'euler', 'ZYX', 'point'), 'point');

T = [Rbef Tbef;0 0 0 1] \ [Rnow Tnow;0 0 0 1];

initTform = rigid3d(T.');
end
```

helperMakeFigurePublishFriendly adjusts figures so that screenshot captured by publish is correct.

```
function helperMakeFigurePublishFriendly(hFig)

if ~isempty(hFig) && isvalid(hFig)
    hFig.HandleVisibility = 'callback';
end
end
```

`helperFeatureSearcher` creates an object that can be used to search for closest feature matches.

`helperLoopClosureDetector` creates an object that can be used to detect loop closures using scan context feature descriptors.

See Also

Functions
`createPoseGraph` | `optimizePoses` | `pcregisterndt` | `pcshow`

Objects
`pcviewset` | `pointCloud` | `rigid3d`

More About

- “Build a Map from Lidar Data” on page 7-539
- “Ground Plane and Obstacle Detection Using Lidar” on page 7-107
- “Lidar Localization with Unreal Engine Simulation” on page 7-701
- “Design Lidar SLAM Algorithm Using Unreal Engine Simulation Environment” on page 7-691

External Websites

- Velodyne SLAM Dataset

Create Occupancy Grid Using Monocular Camera and Semantic Segmentation

This example shows how to estimate free space around a vehicle and create an occupancy grid using semantic segmentation and deep learning. You then use this occupancy grid to create a vehicle costmap, which can be used to plan a path.

About Free Space Estimation

Free space estimation identifies areas in the environment where the ego vehicle can drive without hitting any obstacles such as pedestrians, curbs, or other vehicles. A vehicle can use a variety of sensors to estimate free space such as radar, lidar, or cameras. This example focuses on estimating free space from an image sensor using semantic segmentation.

In this example, you learn how to:

- Use semantic image segmentation to estimate free space.
- Create an occupancy grid using the free space estimate.
- Visualize the occupancy grid on a bird's-eye plot.
- Create a vehicle costmap using the occupancy grid.
- Check whether locations in the world are occupied or free.

Download Pretrained Network

This example uses a pretrained semantic segmentation network, which can classify pixels into 11 different classes, including `Road`, `Pedestrian`, `Car`, and `Sky`. The free space in an image can be estimated by defining image pixels classified as `Road` as free space. All other classes are defined as non-free space or obstacles.

The complete procedure for training this network is shown in the “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox) example. Download the pretrained network.

```
% Download the pretrained network.
pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/segnetVGG16CamVid.mat';
pretrainedFolder = fullfile(tempdir,'pretrainedSegNet');
pretrainedSegNet = fullfile(pretrainedFolder,'segnetVGG16CamVid.mat');
if ~exist(pretrainedFolder,'dir')
    mkdir(pretrainedFolder);
    disp('Downloading pretrained SegNet (107 MB)...');
    websave(pretrainedSegNet,pretrainedURL);
    disp('Download complete.');
end

% Load the network.
data = load(pretrainedSegNet);
net = data.net;
```

Note: Download time of the data depends on your Internet connection. The commands used above block MATLAB until the download is complete. Alternatively, you can use your web browser to first download the data set to your local disk. In this case, to use the file you downloaded from the web, change the `pretrainedFolder` variable above to the location of the downloaded file.

Estimate Free Space

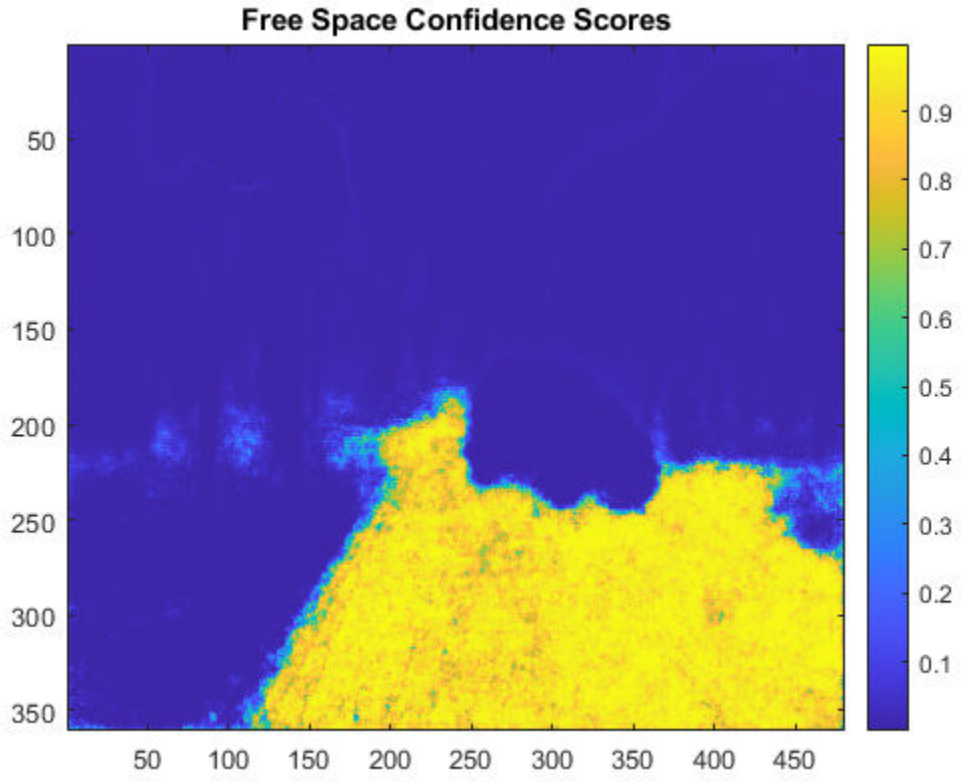
Estimate free space by processing the image using downloaded semantic segmentation network. The network returns classifications for each image pixel in the image. The free space is identified as image pixels that have been classified as `Road`.

The image used in this example is a single frame from an image sequence in the CamVid data set[1]. The procedure shown in this example can be applied to a sequence of frames to estimate free space as a vehicle drives along. However, because a very deep convolutional neural network architecture is used in this example (SegNet with a VGG-16 encoder), it takes about 1 second to process each frame. Therefore, for expediency, process a single frame.

```
% Read the image.
I = imread('seq05vd_snap_shot.jpg');

% Segment the image.
[C,scores,allScores] = semanticseg(I,net);

% Overlay free space onto the image.
B = labeloverlay(I,C,'IncludedLabels',"Road");

% Display free space and image.
figure
imshow(B)
```

To understand the confidence in the free space estimate, display the output score for the Road class for every pixel. The confidence values can be used to inform downstream algorithms of the estimate's validity. For example, even if the network classifies a pixel as Road, the confidence score may be low enough to disregard that classification for safety reasons.

```
% Use the network's output score for Road as the free space confidence.
roadClassIdx = 4;
freeSpaceConfidence = allScores(:,:,roadClassIdx);

% Display the free space confidence.
figure
imagesc(freeSpaceConfidence)
title('Free Space Confidence Scores')
colorbar
```



Although the initial segmentation result for Road pixels showed most pixels on the road were classified correctly, visualizing the scores provides richer detail on the classifier's confidence in those classifications. For example, the confidence decreases as you get closer to the boundary of the car.

Create Bird's-Eye-View Image

The free space estimate is generated in the image space. To facilitate generation of an occupancy grid that is useful for navigation, the free space estimate needs to be transformed into the vehicle coordinate system. This can be done by transforming the free space estimate to a bird's-eye-view image.

To create the bird's-eye-view image, first define the camera sensor configuration. The supporting function listed at the end of this example, `camvidMonoCameraSensor`, returns a `monoCamera` object representing the monocular camera used to collect the CamVid[1] data. Configuring the `monoCamera` requires the camera intrinsics and extrinsics, which were estimated using data provided in the CamVid data set. To estimate the camera intrinsics, the function used CamVid checkerboard calibration images and the Camera Calibrator (Computer Vision Toolbox) app. Estimates of the camera extrinsics, such as height and pitch, were derived from the extrinsic data estimated by the authors of the CamVid data set.

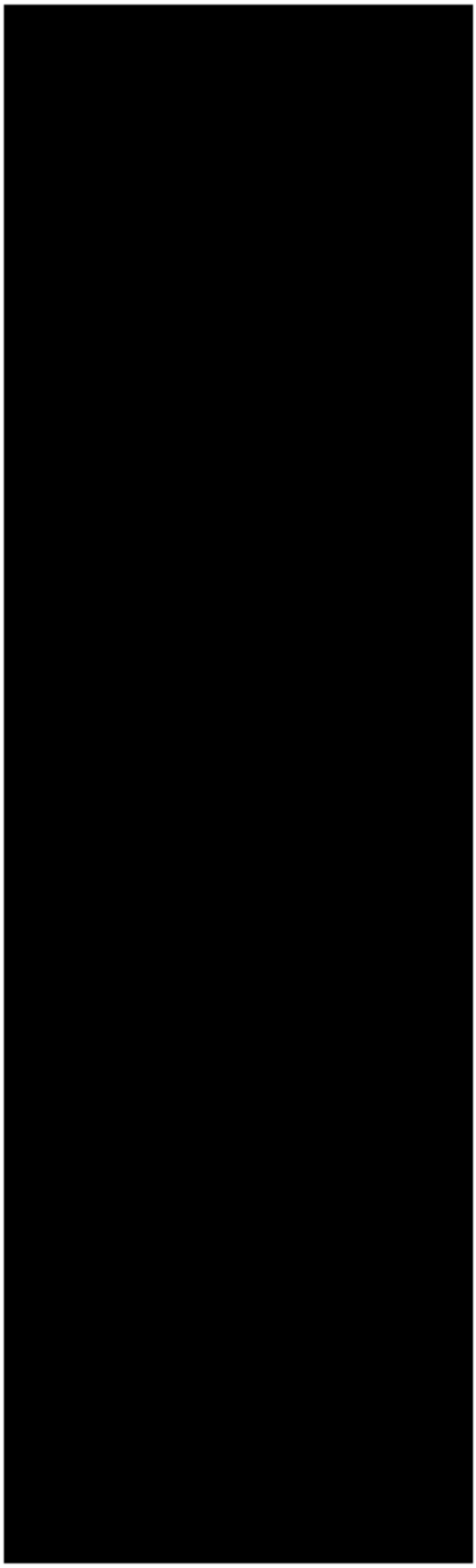
```
% Create monoCamera for CamVid data.
sensor = camvidMonoCameraSensor();
```

Given the camera setup, the `birdsEyeView` object transforms the original image to the bird's-eye view. This object lets you specify the area that you want transformed using vehicle coordinates. Note that the vehicle coordinate units were established by the `monoCamera` object, when the camera mounting height was specified in meters. For example, if the height was specified in millimeters, the rest of the simulation would use millimeters.

```
% Define bird's-eye-view transformation parameters.
distAheadOfSensor = 20; % in meters, as previously specified in monoCamera height input
spaceToOneSide    = 3;  % look 3 meters to the right and left
bottomOffset      = 0;
outView = [bottomOffset, distAheadOfSensor, -spaceToOneSide, spaceToOneSide];

outImageSize = [NaN, 256]; % output image width in pixels; height is chosen automatically to pres

birdsEyeConfig = birdsEyeView(sensor,outView,outImageSize);
```

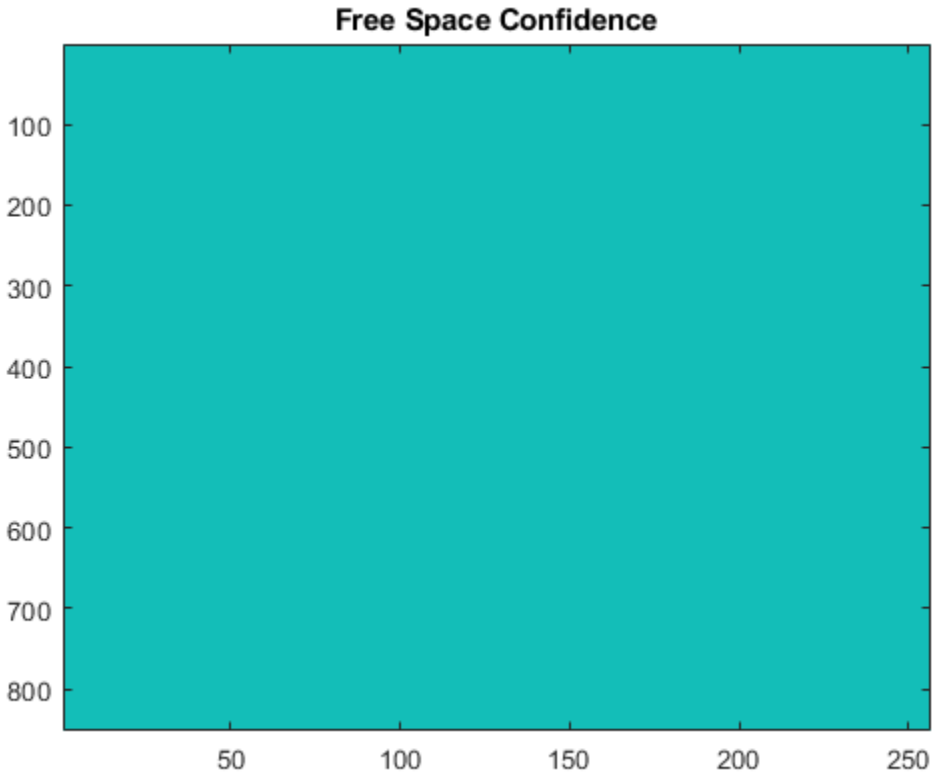
Generate bird's-eye-view image for the image and free space confidence.

```
% Resize image and free space estimate to size of CamVid sensor.
imageSize = sensor.Intrinsics.ImageSize;
I = imresize(I,imageSize);
freeSpaceConfidence = imresize(freeSpaceConfidence,imageSize);

% Transform image and free space confidence scores into bird's-eye view.
imageBEV = transformImage(birdsEyeConfig,I);
freeSpaceBEV = transformImage(birdsEyeConfig,freeSpaceConfidence);

% Display image frame in bird's-eye view.
figure
imshow(imageBEV)
```



Transform the image into a bird's-eye view and generate the free space confidence.

```
figure
imagesc(freeSpaceBEV)
title('Free Space Confidence')
```



The areas farther away from the sensor are more blurry, due to having fewer pixels and thus requiring greater amount of interpolation.

Create Occupancy Grid Based on Free Space Estimation

Occupancy grids are used to represent a vehicle's surroundings as a discrete grid in vehicle coordinates and are used for path planning. Each cell in the occupancy grid has a value representing the probability of the occupancy of that cell. The estimated free space can be used to fill in values of occupancy grid.

The procedure to fill the occupancy grid using the free space estimate is as follows:

1. Define the dimensions of the occupancy grid in vehicle coordinates.
2. Generate a set of (X,Y) points for each grid cell. These points are in the vehicle's coordinate system.
3. Transform the points from the vehicle coordinate space (X,Y) into the bird's-eye-view image coordinate space (x,y) using the `vehicleToImage` transform.
4. Sample the free space confidence values at (x,y) locations using `griddedInterpolant` to interpolate free space confidence values that are not exactly at pixel centers in the image.

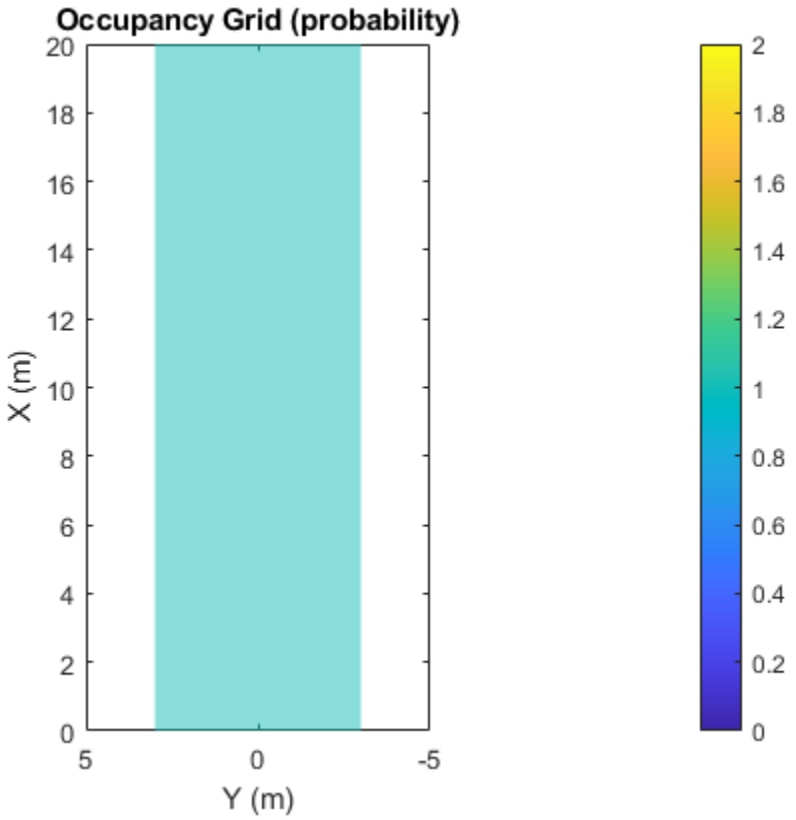
5 Fill the occupancy grid cell with the average free space confidence value for all (x,y) points that correspond to that grid cell.

For brevity, the procedure shown above is implemented in the supporting function, `createOccupancyGridFromFreeSpaceEstimate`, which is listed at the end of this example. Define the dimensions of the occupancy grid in terms of the bird's-eye-view configuration and create the occupancy grid by calling `createOccupancyGridFromFreeSpaceEstimate`.

```
% Define dimensions and resolution of the occupancy grid.
gridX = distAheadOfSensor;
gridY = 2 * spaceToOneSide;
cellSize = 0.25; % in meters to match units used by CamVid sensor

% Create the occupancy grid from the free space estimate.
occupancyGrid = createOccupancyGridFromFreeSpaceEstimate(...
    freeSpaceBEV, birdsEyeConfig, gridX, gridY, cellSize);
```

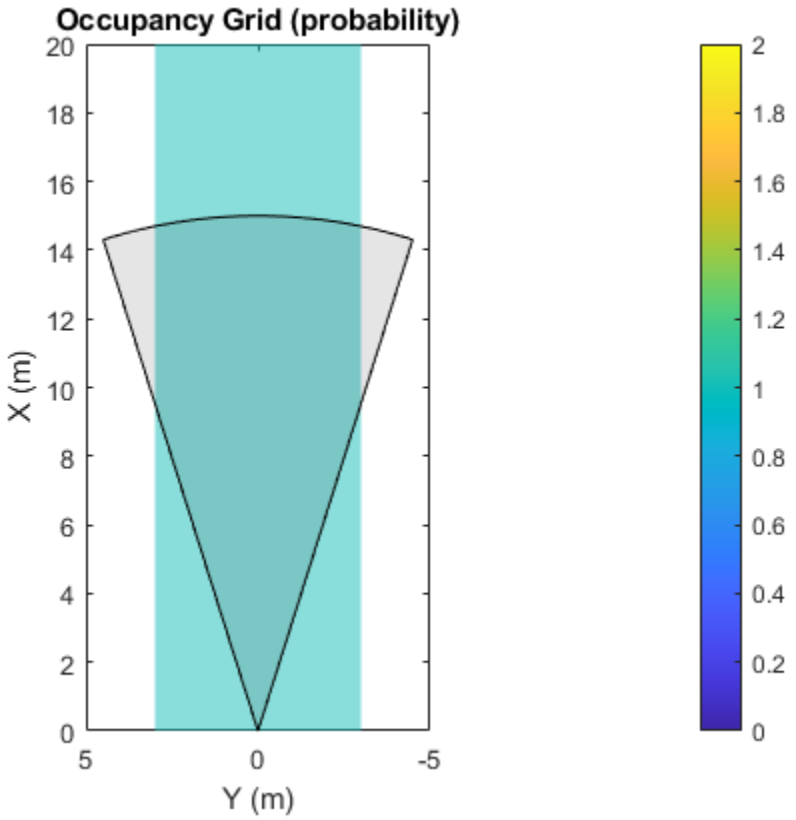
Visualize the occupancy grid using `birdsEyePlot`. Create a `birdsEyePlot` and add the occupancy grid on top using `pcolor`.

```
% Create bird's-eye plot.
bep = birdsEyePlot('XLimits',[0 distAheadOfSensor],'YLimits', [-5 5]);

% Add occupancy grid to bird's-eye plot.
hold on
[numCellsY,numCellsX] = size(occupancyGrid);
X = linspace(0, gridX, numCellsX);
Y = linspace(-gridY/2, gridY/2, numCellsY);
h = pcolor(X,Y,occupancyGrid);
title('Occupancy Grid (probability)')
colorbar
delete(legend)

% Make the occupancy grid visualization transparent and remove grid lines.
h.FaceAlpha = 0.5;
h.LineStyle = 'none';
```



The bird's-eye plot can also display data from multiple sensors. For example, add the radar coverage area using `coverageAreaPlotter`.

```
% Add coverage area to plot.
caPlotter = coverageAreaPlotter(bep, 'DisplayName', 'Coverage Area');

% Update it with a field of view of 35 degrees and a range of 60 meters
mountPosition = [0 0];
range = 15;
orientation = 0;
fieldOfView = 35;
plotCoverageArea(caPlotter, mountPosition, range, orientation, fieldOfView);
hold off
```

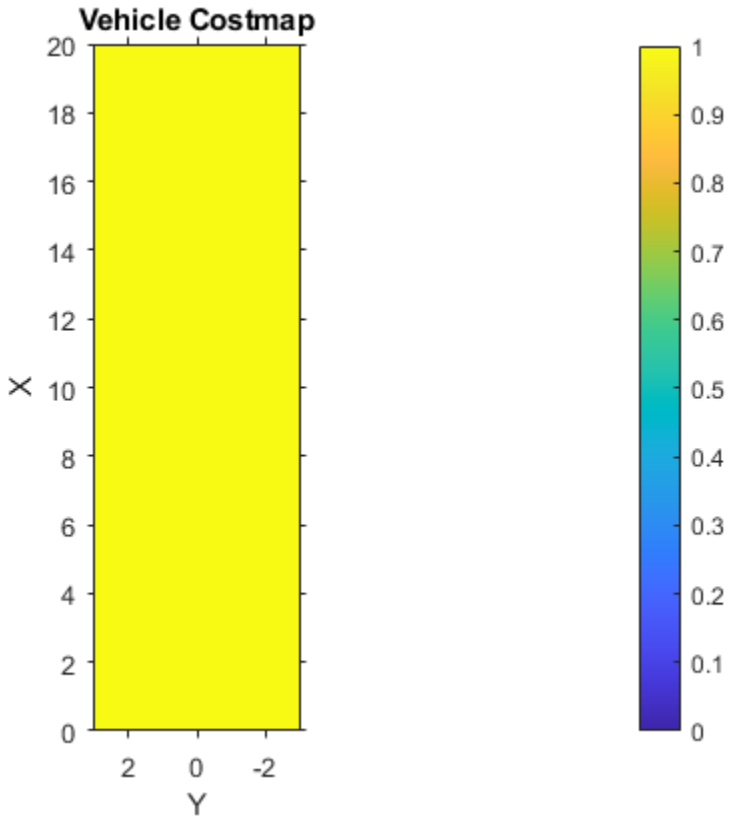


Displaying data from multiple sensors is useful for diagnosing and debugging decisions made by autonomous vehicles.

Create Vehicle Costmap Using the Occupancy Grid

The `vehicleCostmap` provides functionality to check if locations, in vehicle or world coordinates, are occupied or free. This check is required for any path-planning or decision-making algorithm. Create the `vehicleCostmap` using the generated `occupancyGrid`.

```
% Create the costmap.
costmap = vehicleCostmap(flipud(occupancyGrid), ...
    'CellSize',cellSize, ...
    'MapLocation',[0,-spaceToOneSide]);
costmap.CollisionChecker.InflationRadius = 0;

% Display the costmap.
figure
plot(costmap,'Inflation','off')
colormap(parula)
colorbar
title('Vehicle Costmap')

% Orient the costmap so that it lines up with the vehicle coordinate
% system, where the X-axis points in front of the ego vehicle and the
% Y-axis points to the left.
view(gca,-90,90)
```



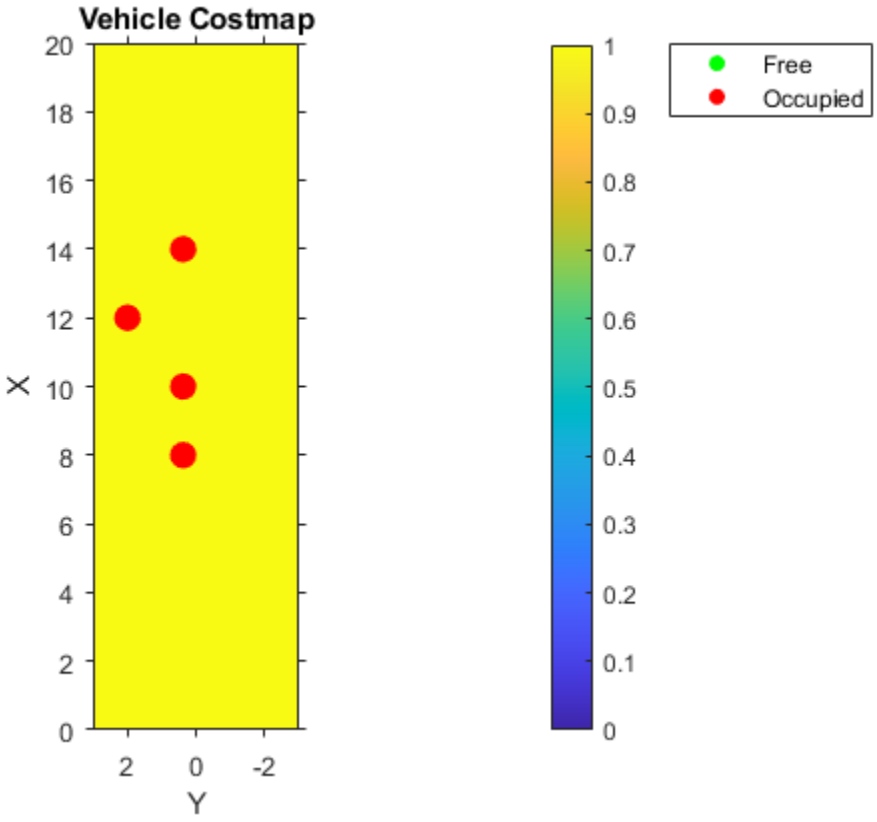
To illustrate how to use the `vehicleCostmap`, create a set of locations in world coordinates. These locations represent a path the vehicle could traverse.

```
% Create a set of locations in vehicle coordinates.
candidateLocations = [
    8 0.375
    10 0.375
    12 2
    14 0.375
    ];
```

Use `checkOccupied` to check whether each location is occupied or free. Based on the results, a potential path might be impossible to follow because it collides with obstacles defined in the `costmap`.

```
% Check if locations are occupied.
isOccupied = checkOccupied(costmap,candidateLocations);

% Partition locations into free and occupied for visualization purposes.
occupiedLocations = candidateLocations(isOccupied,:);
freeLocations = candidateLocations(~isOccupied,:);

% Display free and occupied points on top of costmap.
hold on
markerSize = 100;
scatter(freeLocations(:,1),freeLocations(:,2),markerSize,'g','filled')
scatter(occupiedLocations(:,1),occupiedLocations(:,2),markerSize,'r','filled');
```

```
legend(["Free" "Occupied"])
hold off
```



The use of `occupancyGrid`, `vehicleCostmap`, and `checkOccupied` shown above illustrate the basic operations used by path planners such as `pathPlannerRRT`. Learn more about path planning in the “Automated Parking Valet” on page 7-465 example.

References

[1] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. "Semantic Object Classes in Video: A high-definition ground truth database." *Pattern Recognition Letters*. Vol. 30, Issue 2, 2009, pp. 88-97.

Supporting Functions

```
function sensor = camvidMonoCameraSensor()
% Return a monoCamera camera configuration based on data from the CamVid
% data set[1].
%
% The cameraCalibrator app was used to calibrate the camera using the
% calibration images provided in CamVid:
%
% http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/CalibrationSeq_and_Files_0010YU
%
% Calibration pattern grid size is 28 mm.
%
% Camera pitch is computed from camera pose matrices [R t] stored here:
%
```

```
% http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/EgoBoost_trax_matFiles.zip

% References
% ----------
% [1] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. "Semantic Object
% Classes in Video: A high-definition ground truth database." _Pattern Recognition
% Letters_. Vol. 30, Issue 2, 2009, pp. 88-97.

calibrationData = load('camera_params_camvid.mat');

% Describe camera configuration.
focalLength    = calibrationData.cameraParams.FocalLength;
principalPoint = calibrationData.cameraParams.PrincipalPoint;
imageSize      = calibrationData.cameraParams.ImageSize;

% Camera height estimated based on camera setup pictured in [1].
height = 0.5;  % height in meters from the ground

% Camera pitch was computed using camera extrinsics provided in data set.
pitch = 0;  % pitch of the camera, towards the ground, in degrees

camIntrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);
sensor = monoCamera(camIntrinsics,height,'Pitch',pitch);
end

function occupancyGrid = createOccupancyGridFromFreeSpaceEstimate(...
    freeSpaceBEV,birdsEyeConfig,gridX,gridY,cellSize)
% Return an occupancy grid that contains the occupancy probability over
% a uniform 2-D grid.

% Number of cells in occupancy grid.
numCellsX = ceil(gridX / cellSize);
numCellsY = ceil(gridY / cellSize);

% Generate a set of (X,Y) points for each grid cell. These points are in
% the vehicle's coordinate system. Start by defining the edges of each grid
% cell.

% Define the edges of each grid cell in vehicle coordinates.
XEdges = linspace(0,gridX,numCellsX);
YEdges = linspace(-gridY/2,gridY/2,numCellsY);

% Next, specify the number of sample points to generate along each
% dimension within a grid cell. Use these to compute the step size in the
% X and Y direction. The step size will be used to shift the edge values of
% each grid to produce points that cover the entire area of a grid cell at
% the desired resolution.

% Sample 20 points from each grid cell. Sampling more points may produce
% smoother estimates at the cost of additional computation.
numSamplePoints = 20;

% Step size needed to sample number of desired points.
XStep = (XEdges(2)-XEdges(1)) / (numSamplePoints-1);
YStep = (YEdges(2)-YEdges(1)) / (numSamplePoints-1);

% Finally, slide the set of points across both dimensions of the grid
% cells. Sample the occupancy probability along the way using
```

```
% griddedInterpolant.

% Create griddedInterpolant for sampling occupancy probability. Use 1
% minus the free space confidence to represent the probability of occupancy.
occupancyProb = 1 - freeSpaceBEV;
sz = size(occupancyProb);
[y,x] = ndgrid(1:sz(1),1:sz(2));
F = griddedInterpolant(y,x,occupancyProb);

% Initialize the occupancy grid to zero.
occupancyGrid = zeros(numCellsY*numCellsX,1);

% Slide the set of points XEdges and YEdges across both dimensions of the
% grid cell.
for j = 1:numSamplePoints

    % Increment sample points in the X-direction
    X = XEdges + (j-1)*XStep;

    for i = 1:numSamplePoints

        % Increment sample points in the Y-direction
        Y = YEdges + (i-1)*YStep;

        % Generate a grid of sample points in bird's-eye-view vehicle coordinates
        [XGrid,YGrid] = meshgrid(X,Y);

        % Transform grid of sample points to image coordinates
        xy = vehicleToImage(birdsEyeConfig,[XGrid(:) YGrid(:)]);

        % Clip sample points to lie within image boundaries
        xy = max(xy,1);
        xq = min(xy(:,1),sz(2));
        yq = min(xy(:,2),sz(1));

        % Sample occupancy probabilities using griddedInterpolant and keep
        % a running sum.
        occupancyGrid = occupancyGrid + F(yq,xq);
    end

end

% Determine mean occupancy probability.
occupancyGrid = occupancyGrid / numSamplePoints^2;
occupancyGrid = reshape(occupancyGrid,numCellsY,numCellsX);
end
```

See Also

Apps
Camera Calibrator

Functions
checkOccupied | semanticseg

Objects
birdsEyePlot | birdsEyeView | monoCamera | pathPlannerRRT | vehicleCostmap

More About

- “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)
- “Automated Parking Valet” on page 7-465
- “Automate Ground Truth Labeling for Semantic Segmentation” on page 7-29
- “Train Deep Learning Semantic Segmentation Network Using 3-D Simulation Data” (Deep Learning Toolbox)

Lateral Control Tutorial

This example shows how to control the steering angle of a vehicle that is following a planned path while changing lanes, using the Lateral Controller Stanley block.

Overview

Vehicle control is the final step in a navigation system and is typically accomplished using two independent controllers:

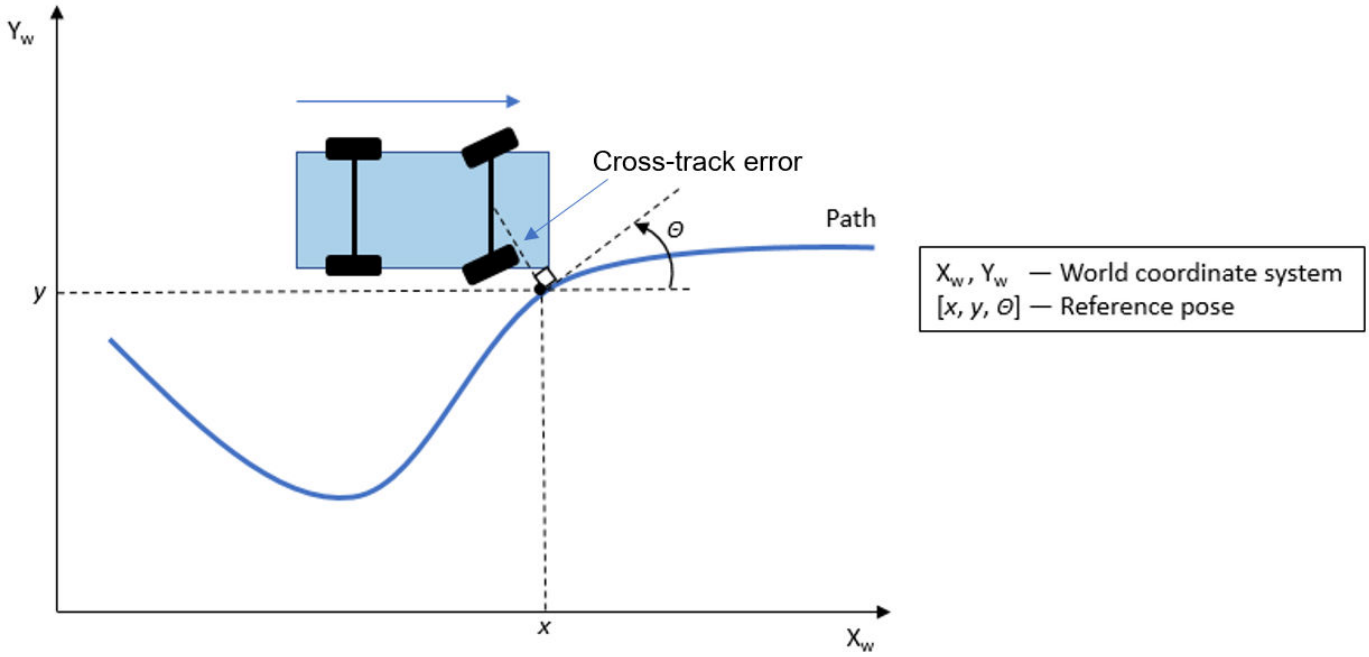
- **Lateral Controller**: Adjust the steering angle such that the vehicle follows the reference path. The controller minimizes the distance between the current vehicle position and the reference path.
- **Longitudinal Controller**: While following the reference path, maintain the desired speed by controlling the throttle and the brake. The controller minimizes the difference between the heading angle of the vehicle and the orientation of the reference path.

This example focuses on lateral control in the context of path following in a constant longitudinal velocity scenario. In the example, you will:

1. Learn about the algorithm behind the Lateral Controller Stanley block.
2. Create a driving scenario using the Driving Scenario Designer app and generate a reference path for the vehicle to follow.
3. Test the lateral controller in the scenario using a closed-loop Simulink® model.
4. Visualize the scenario and the associated simulation results using the Bird's-Eye Scope.

Lateral Controller

The Stanley lateral controller [1] uses a nonlinear control law to minimize the cross-track error and the heading angle of the front wheel relative to the reference path. The Lateral Controller Stanley block computes the steering angle command that adjusts a vehicle's current pose to match a reference pose.



Depending on the vehicle model used in deriving the control law, the Lateral Controller Stanley block has two configurations [1]:

- **Kinematic bicycle model**: The kinematic model assumes that the vehicle has negligible inertia. This configuration is mainly suitable for low-speed environments, where inertial effects are minimal. The steering command is computed based on the reference pose, the current pose, and the velocity of the vehicle.
- **Dynamic bicycle model**: The dynamic model includes inertia effects: tire slip and steering servo actuation. This more complicated, but more accurate, model allows the controller to handle realistic dynamics. In this configuration, the controller also requires the path curvature, the current yaw rate of the vehicle, and the current steering angle to compute the steering command.

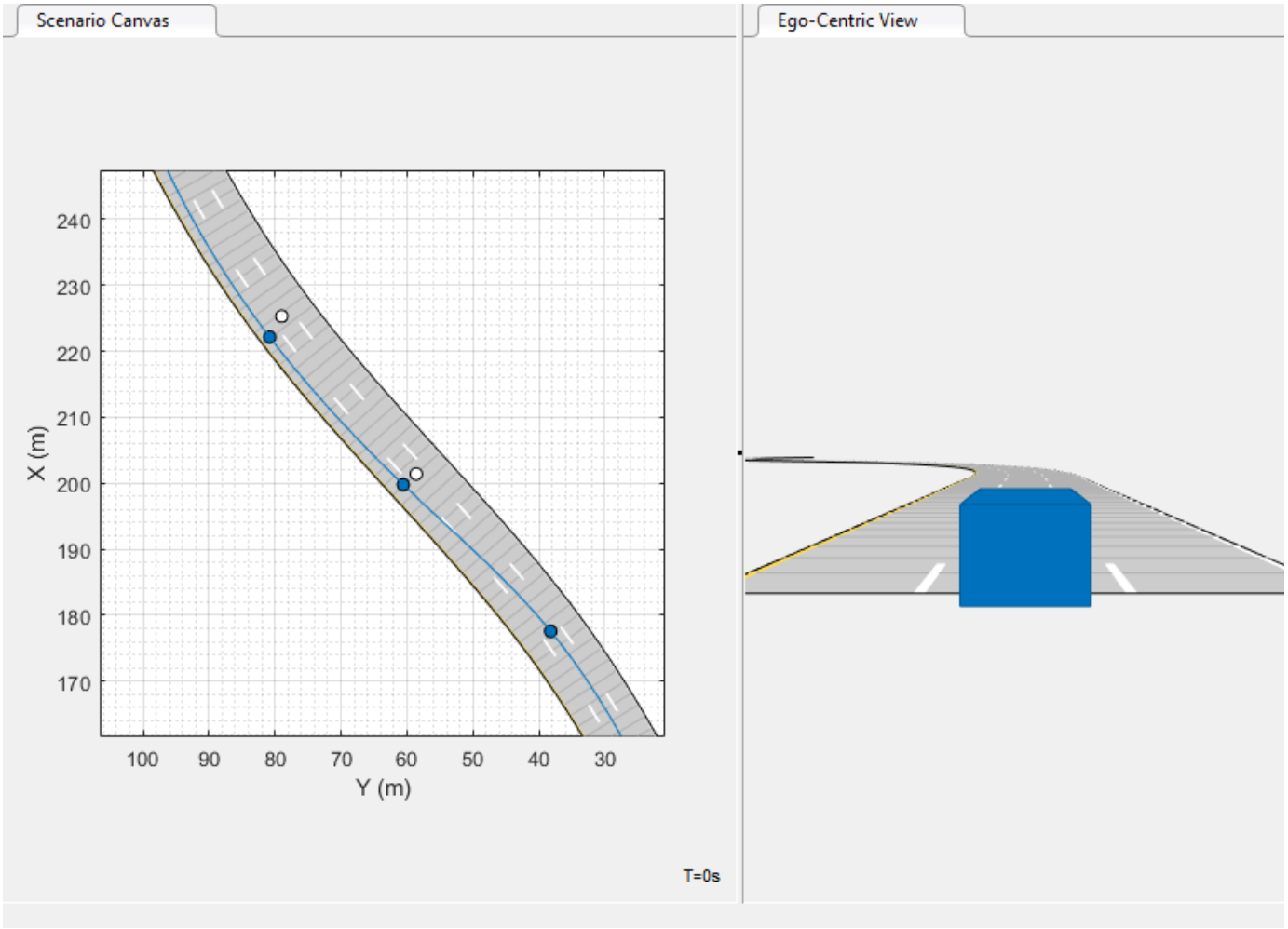
You can set the configuration through the **Vehicle model** parameter in the block dialog box.

Scenario Creation

The scenario was created using the Driving Scenario Designer app. This scenario includes a single, three-lane road and the ego vehicle. For detailed steps on adding roads, lanes, and vehicles, see “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2. In this scenario, the vehicle:

1. Starts in the middle lane.
2. Switches to the left lane after entering the curved part of the road.
3. Changes back to the middle lane.

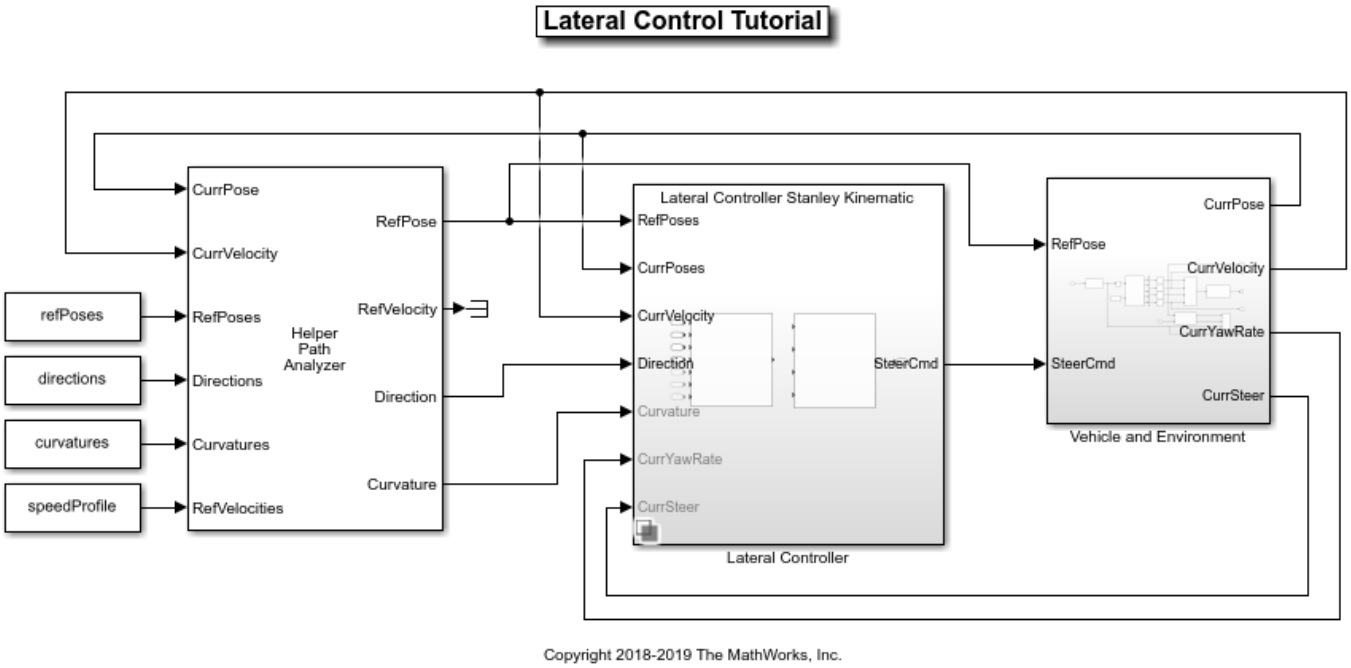
Throughout the simulation, the vehicle runs at a constant velocity of 10 meters/second. This scenario was exported from the app as a MATLAB® function using the **Export > Export MATLAB Function** button. The exported function is named `helperCreateDrivingScenario`. The roads and actors from this scenario were saved to the scenario file `LateralControl.mat`.



Model Setup

Open the Simulink tutorial model.

```
open_system('LateralControlTutorial')
```



The model contains the following main components:

- A **Lateral Controller** Variant Subsystem, Variant Model (Simulink) which contains two Lateral Controller Stanley blocks, one configured with a kinematic bicycle model and the other one with a dynamic bicycle model. They both can control the steering angle of the vehicle. You can specify the active one from the command line. For example, to select the Lateral Controller Stanley Kinematic block, use the following command:

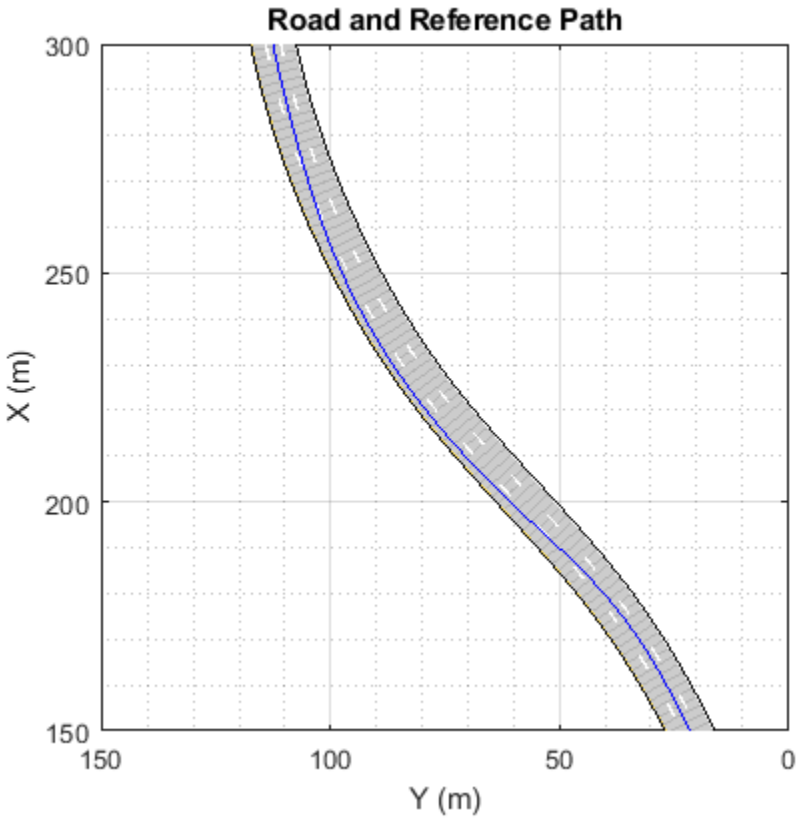
```
variant = 'LateralControlTutorial/Lateral Controller';
set_param(variant, 'LabelModeActivechoice', 'Kinematic');
```

- A `HelperPathAnalyzer` block, which provides the reference signal for the lateral controller. Given the current pose of the vehicle, it determines the reference pose by searching for the closest point to the vehicle on the reference path.
- A **Vehicle and Environment** subsystem, which models the motion of the vehicle using a Vehicle Body 3DOF (Vehicle Dynamics Blockset) block. The subsystem also models the environment by using a Scenario Reader block to read the roads and actors from the LateralControl.mat scenario file.

Opening the model also runs the `helperLateralControlTutorialSetup` script, which initializes data used by the model. The script loads certain constants needed by the Simulink model, such as vehicle parameters, controller parameters, the road scenario, and reference poses. In particular, the script calls the previously exported function `helperCreateDrivingScenario` to build the scenario. The script also sets up the buses required for the model by calling `helperCreateLaneSensorBuses`.

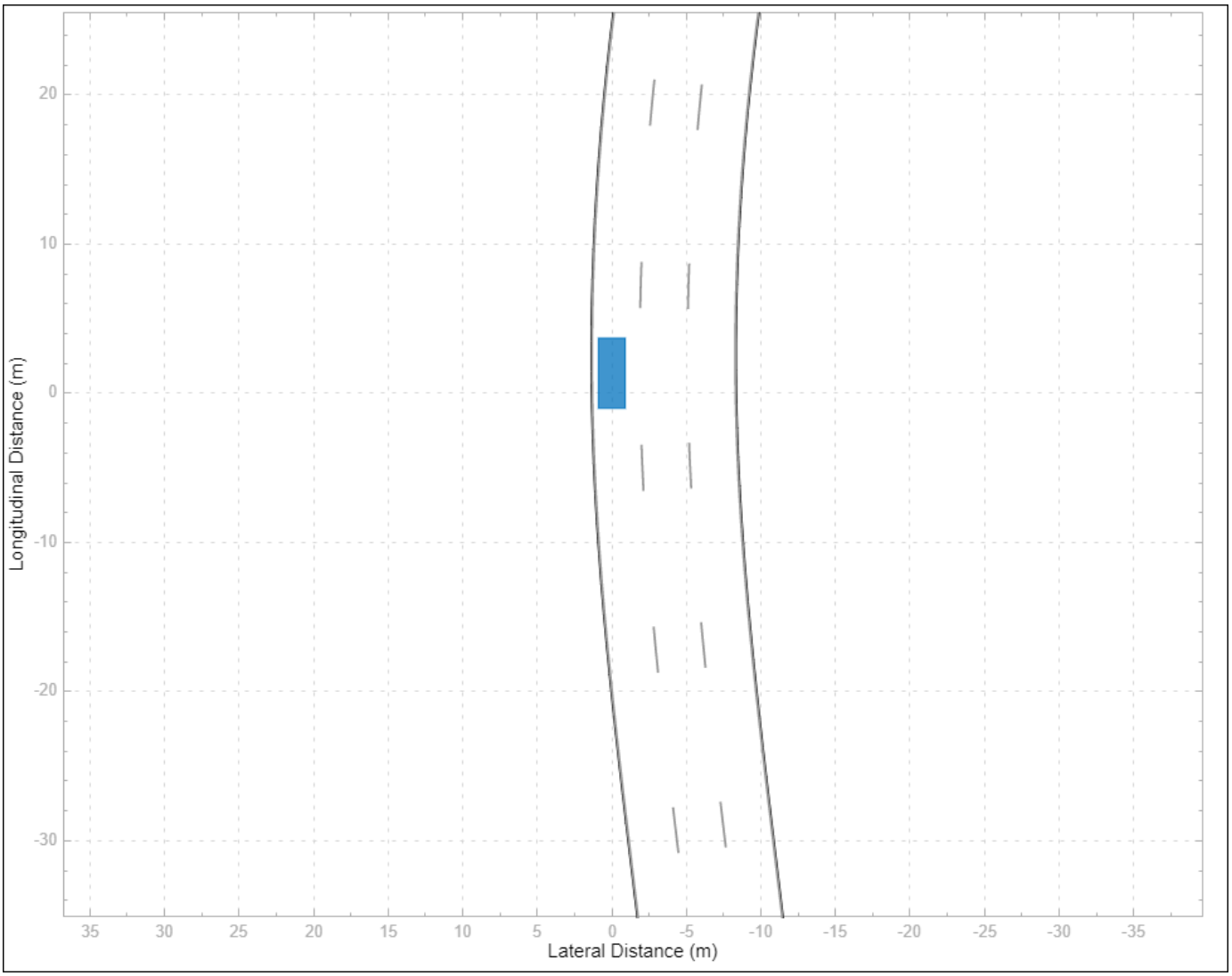
You can plot the road and the planned path using:

```
helperPlotRoadAndPath(scenario, refPoses)
```



Simulate Scenario

When simulating the model, you can open the Bird's-Eye Scope to analyze the simulation. After opening the scope, click **Find Signals** to set up the signals. Then run the simulation to display the vehicle, the road boundaries, and the lane markings. The image below shows the Bird's-Eye Scope for this example at 25 seconds. At this instant, the vehicle has switched to the left lane.

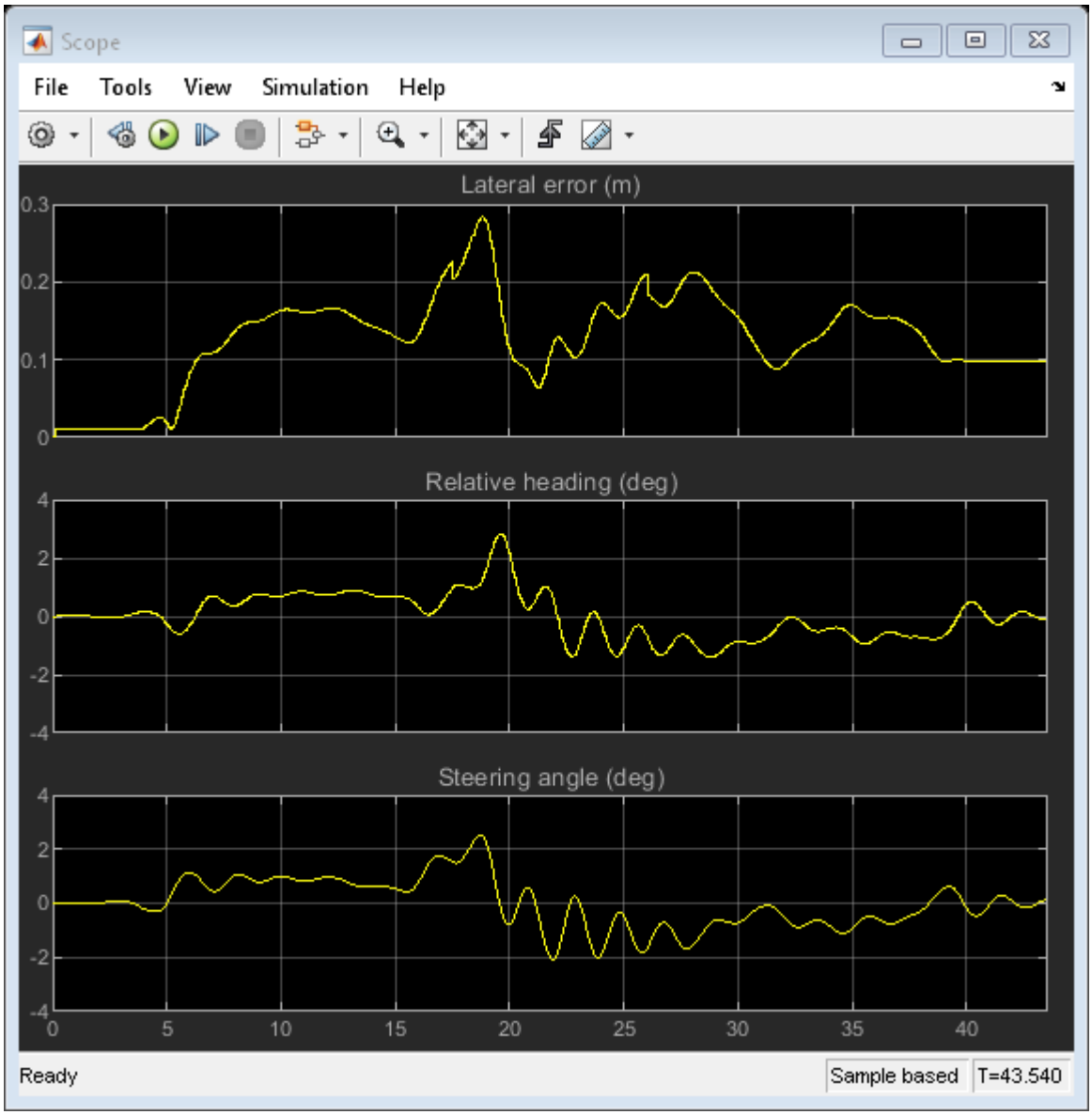


You can run the full simulation and explore the results using the following command:

```
sim('LateralControlTutorial');
```

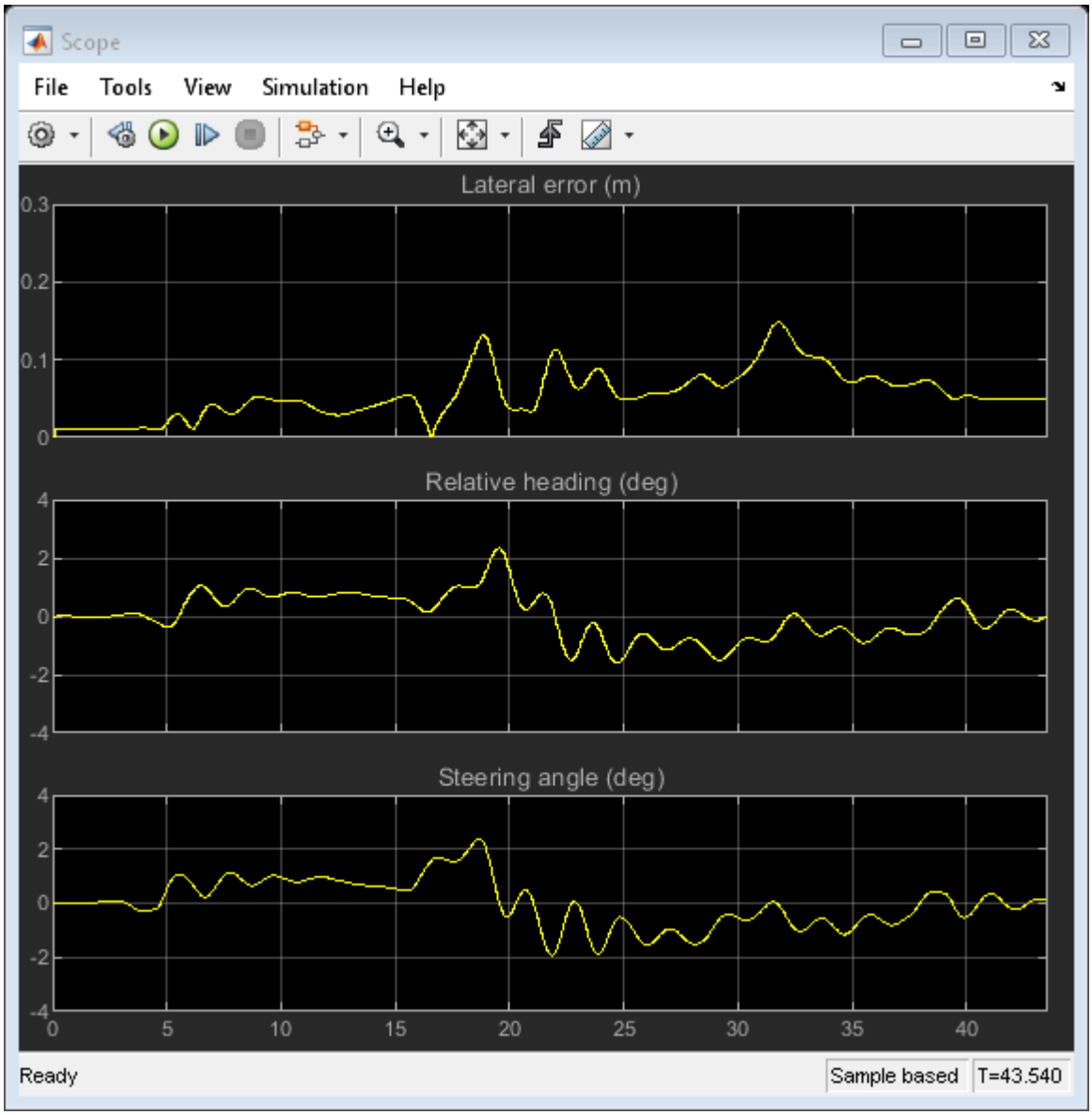
You can also use the Simulink® Scope (Simulink) in the **Vehicle and Environment** subsystem to inspect the performance of the controller as the vehicle follows the planned path. The scope shows the maximum deviation from the path is less than 0.3 meters and the largest steering angle magnitude is less than 3 degrees.

```
scope = 'LateralControlTutorial/Vehicle and Environment/Scope';
open_system(scope)
```



To reduce the lateral deviation and oscillation in the steering command, use the Lateral Controller Stanley Dynamic block and simulate the model again:

```
set_param(variant, 'LabelModeActivechoice', 'Dynamic');
sim('LateralControlTutorial');
```



Conclusions

This example showed how to simulate lateral control of a vehicle in a lane changing scenario using Simulink. Compared with the Lateral Controller Stanley Kinematic block, the Lateral Controller Stanley Dynamic block provides improved performance in path following with smaller lateral deviation from the reference path.

References

[1] Hoffmann, Gabriel M., Claire J. Tomlin, Michael Montemerlo, and Sebastian Thrun. "Autonomous Automobile Trajectory Tracking for Off-Road Driving: Controller Design, Experimental Validation and Racing." *American Control Conference*. 2007, pp. 2296-2301.

Supporting Functions

helperPlotRoadAndPath Plot the road and the reference path

```
function helperPlotRoadAndPath(scenario,refPoses)
%helperPlotRoadAndPath Plot the road and the reference path
h = figure('Color','white');
ax1 = axes(h, 'Box','on');

plot(scenario,'Parent',ax1)
hold on
plot(ax1,refPoses(:,1),refPoses(:,2),'b')
xlim([150, 300])
ylim([0 150])
ax1.Title = text(0.5,0.5,'Road and Reference Path');
end
```

See Also

Apps

Bird's-Eye Scope | Driving Scenario Designer

Blocks

Lane Keeping Assist System | Lateral Controller Stanley | Vehicle Body 3DOF

More About

- “Automated Parking Valet in Simulink” on page 7-493
- “Create Driving Scenario Interactively and Generate Synthetic Sensor Data” on page 5-2

Highway Lane Change

This example shows how to simulate an automated lane change maneuver system for highway driving scenario.

Introduction

An automated lane change maneuver (LCM) system enables the ego vehicle to automatically move from one lane to another lane. The LCM system models the longitudinal and lateral control dynamics for automated lane change. An LCM system senses the environment for most important objects (MIOs) using on-board sensors, identifies an optimal trajectory that avoids these objects, and steers ego vehicle along this trajectory.

This example shows how to design and test the planner and controller components of an LCM system. In this example, the lane change planner uses ground truth information from the scenario to detect MIOs. It then generates a feasible trajectory to negotiate a lane change that is executed by the lane change controller. In this example, you:

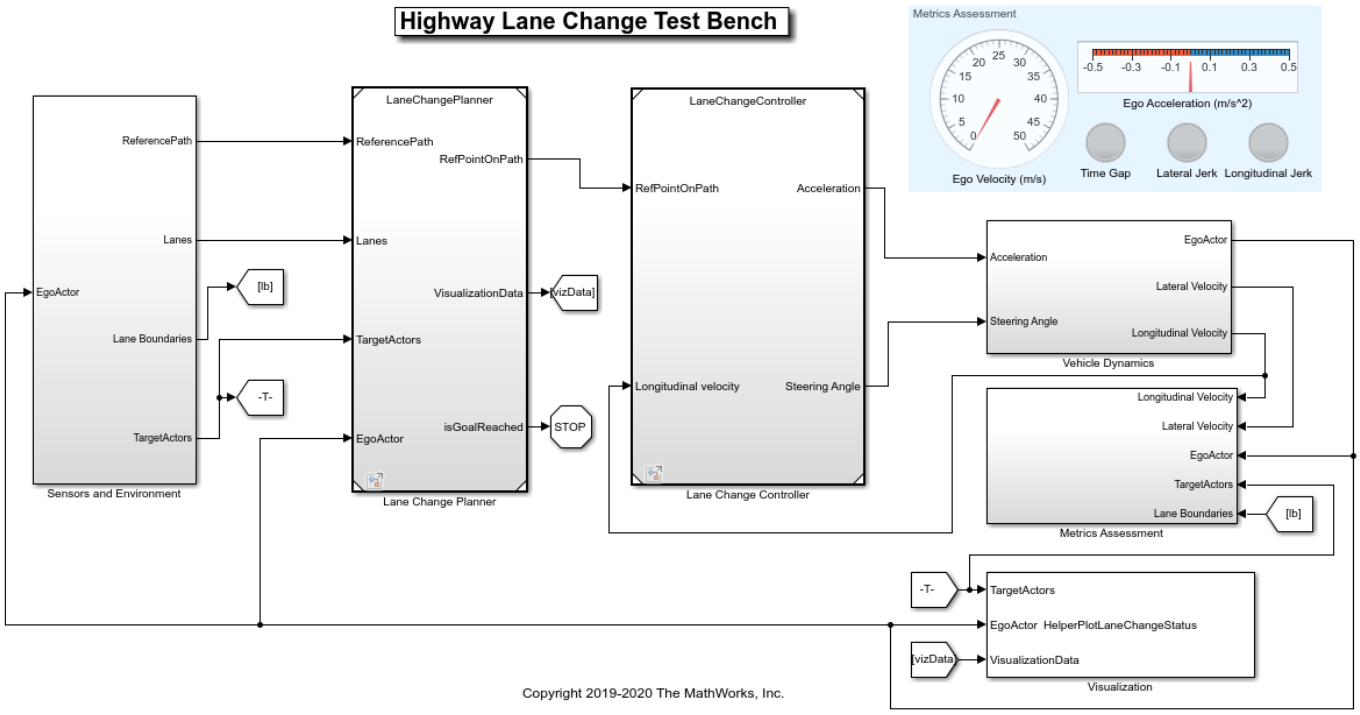
1. **Explore the test bench model:** The model contains planning, controls, vehicle dynamics, sensors, and metrics to assess functionality.
2. **Model the lane change planner:** The reference model contains a simple behavior layer and a motion planner. The behavior layer configures the motion planner to generate an optimal trajectory by considering MIO information.
3. **Model the lane change controller:** The reference model generates control commands for the ego vehicle based on the generated trajectory.
4. **Simulate and visualize system behavior:** The test bench model is configured to test the integration of planning and controls to perform lane change maneuver on a straight road with multiple vehicles.
5. **Explore other scenarios:** These scenarios test the system under additional conditions.

You can apply the modeling patterns used in this example to test your own planner and controller components of an LCM system.

Explore Test Bench Model

In this example, you use a system-level simulation test bench model to explore the behavior of the planner and controller components for a lane change maneuver system. Open the system-level simulation test bench model.

```
open_system('HighwayLaneChangeTestBench')
```



Opening this model runs the `helperSLHighwayLaneChangeSetup` script that initializes the road scenario using the `drivingScenario` object in the base workspace. It also configures the controller design parameters, vehicle model parameters, and the Simulink® bus signals required for defining the inputs and outputs for the `HighwayLaneChangeTestBench` model.

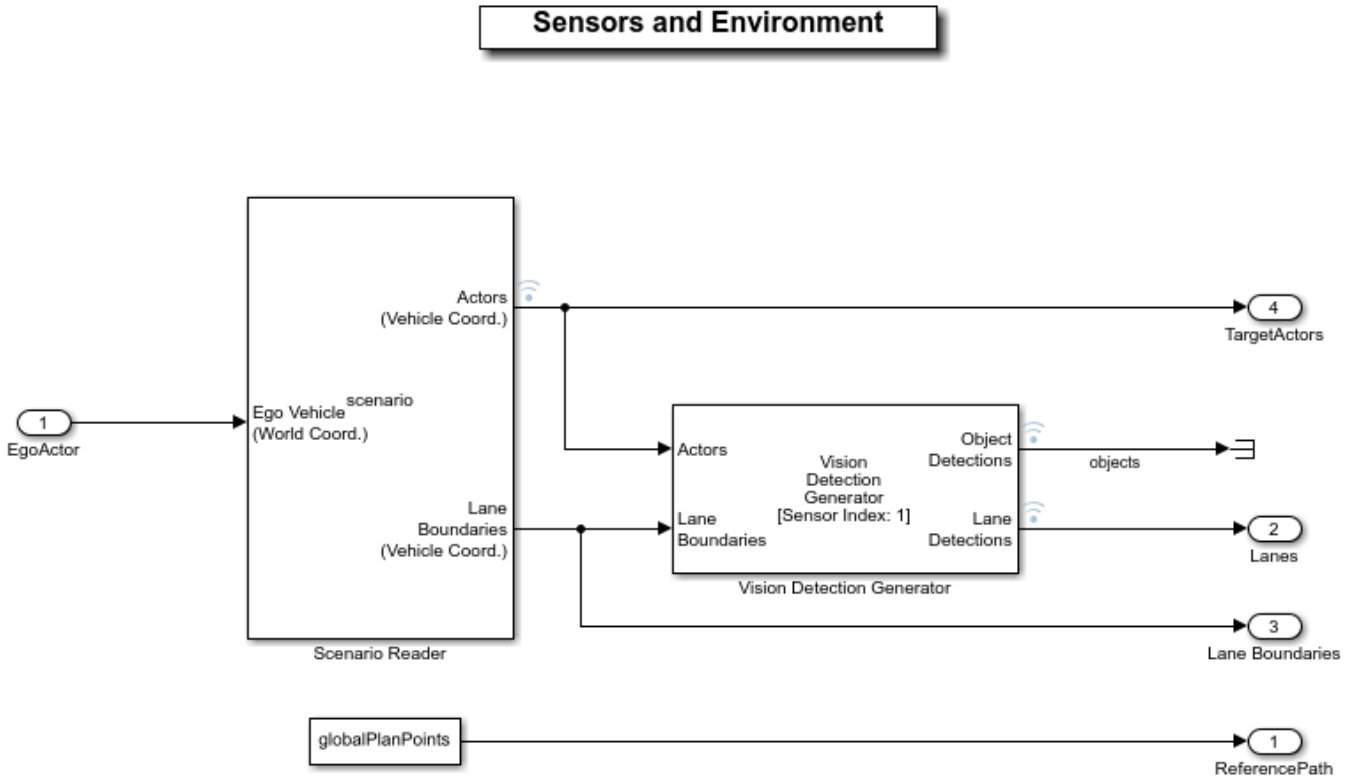
The test bench model contains the following subsystems.

1. **Sensors and Environment** specifies the road, vehicles, and sensors used for simulation.
2. **Lane Change Planner** specifies behavior and trajectory planning for the ego vehicle.
3. **Lane Change Controller** specifies the path following controller that generates control commands to steer the ego vehicle along the generated trajectory.
4. **Vehicle Dynamics** models the ego-vehicle using a Bicycle Model and updates its state using commands received from the **Lane Change Controller**.
5. **Metrics Assessment** specifies metrics to assess system level behavior.

The **Vehicle Dynamics** subsystem is based on the subsystem used in the “Highway Lane Following” on page 7-653 example. This example focuses on the **Lane Change Planner** and **Lane Change Controller** reference models.

The **Sensors and Environment** subsystem uses Scenario Reader and Vision Detection Generator blocks to provide the road network, vehicle ground truth positions, and lane detections required for the lane change maneuver system. Open the **Sensors and Environment** subsystem. This subsystem also outputs the reference path required by the lane change planner.

```
open_system('HighwayLaneChangeTestBench/Sensors and Environment')
```



The Scenario Reader block is configured to read the `drivingScenario` object from the base workspace. It uses this object to read the actor data and lane information. It takes in ego vehicle information to perform a closed-loop simulation. This block outputs ground truth information of lanes and actors in ego vehicle coordinates.

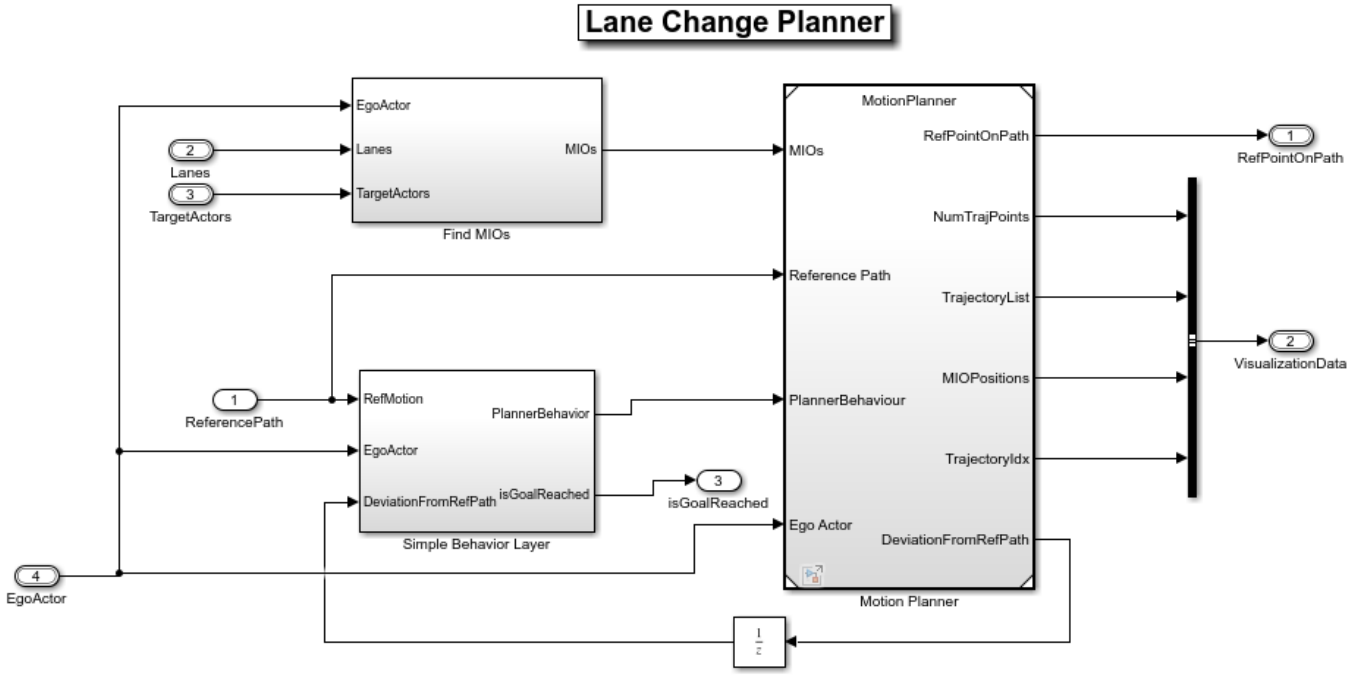
The Vision Detection Generator block provides lane detections with respect to the ego vehicle which help in identifying vehicles present in the ego lane and adjacent lanes.

The **Lane Change Planner** reference model uses lane detections from the Vision Detection Generator block and ground-truth actor or vehicle poses from the Scenario Reader to perform trajectory planning for the automated lane change maneuver.

Model Lane Change Planner

The **Lane Change Planner** reference model uses a **Simple Behavior Layer** to configure the **Motion Planner**. **Motion Planner** is responsible for generating the trajectory for a lane change maneuver. Open the **Lane Change Planner** reference model.

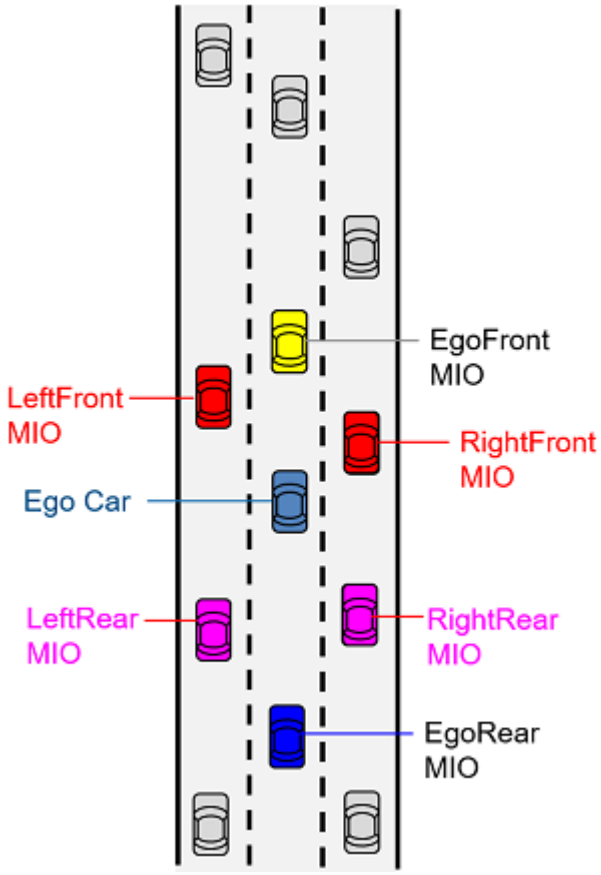
```
open_system('LaneChangePlanner')
```



Copyright 2019-2020 The MathWorks, Inc.

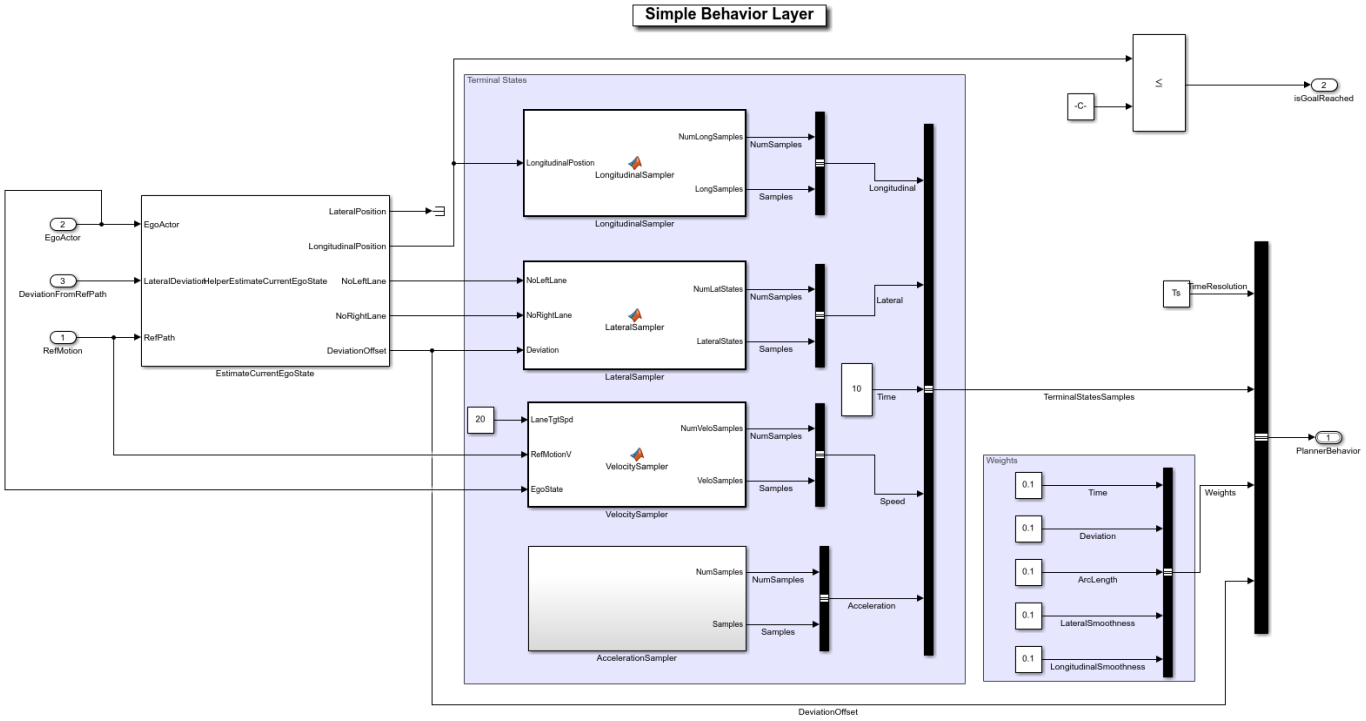
1. The **Find MIOs** subsystem finds the most important objects with respect to the current state of the ego vehicle.
2. The **Simple Behavior Layer** subsystem specifies the planner behavior for the Motion planner.
3. The **Motion Planner** reference model uses `trajectoryOptimalFrenet` (Navigation Toolbox) and MIO information to perform trajectory planning.

The **Find MIOs** subsystem uses ground-truth vehicle poses to compute MIO information with respect to the ego vehicle. The vehicles present in the front or rear of the ego vehicle are considered as MIOs. The MIOs can also be in adjacent lanes as shown in the following figure.

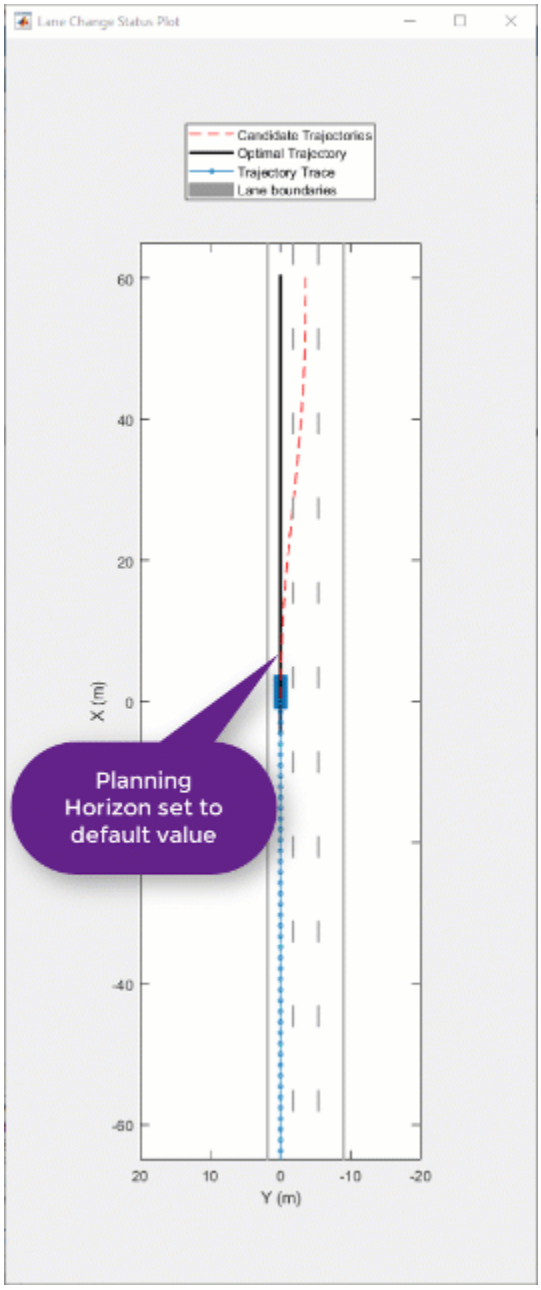


The **Simple Behavior Layer** subsystem configures planner behavior by computing the terminal states, weights, and deviation values using the current ego vehicle state information. Open the **Simple Behavior Layer** subsystem.

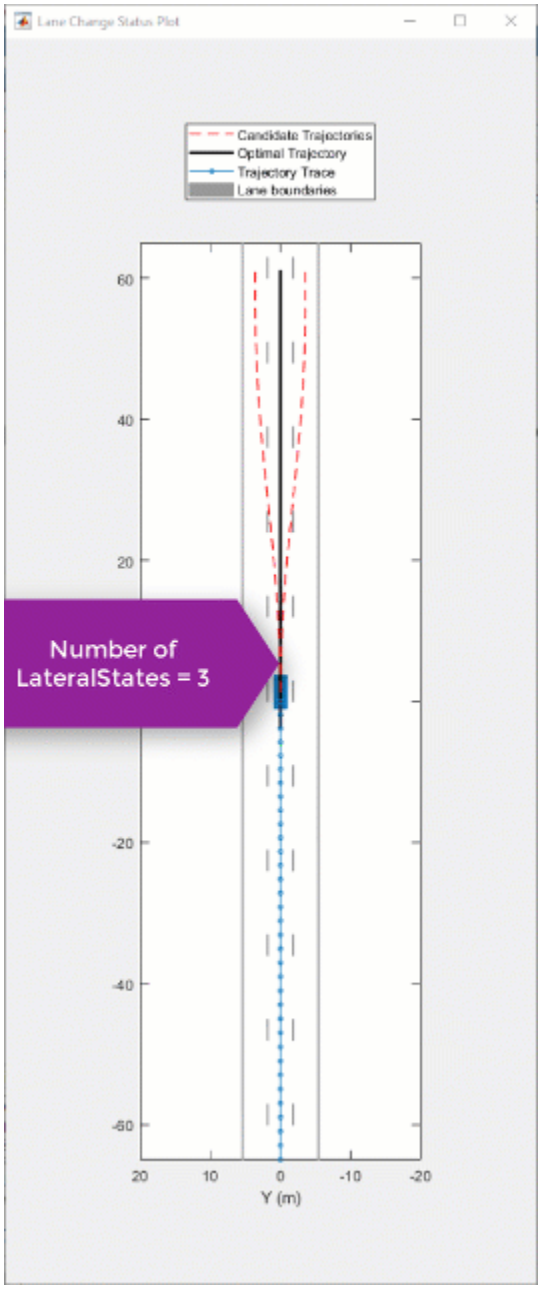
```
open_system('LaneChangePlanner/Simple Behavior Layer')
```



- The **EstimateCurrentEgoState** block uses a System object™, `HelperEstimateCurrentEgoState`, to compute the lateral and longitudinal positions of the ego vehicle. This computation is done in Frenet space using the current state of the ego vehicle and lane detections. It also computes information about adjacent lane availability (NoLeftLane, NoRightLane) with respect to ego vehicle position and deviation offset from the reference path.
- The **LongitudinalSampler** block defines the longitudinal planning horizon, as shown in the following figure. It configures the possible longitudinal terminal state for defining the planner behavior based on the distance to the input longitudinal position. The input longitudinal position is calculated based on the distance to the goal point.



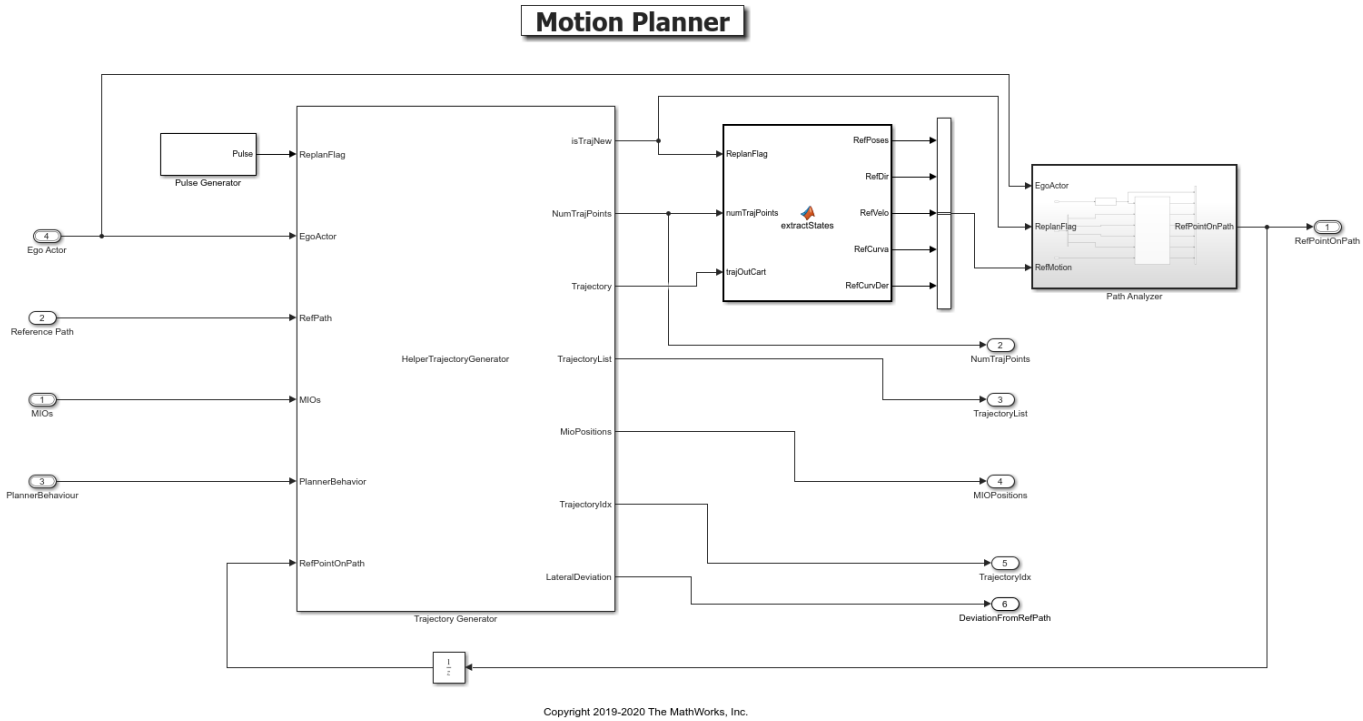
- The **LateralSampler** block defines the lateral planning horizon, as shown in the following figure. It configures the possible lateral terminal states for defining the planner behavior based on adjacent lane availability (NoLeftLane, NoRightLane).



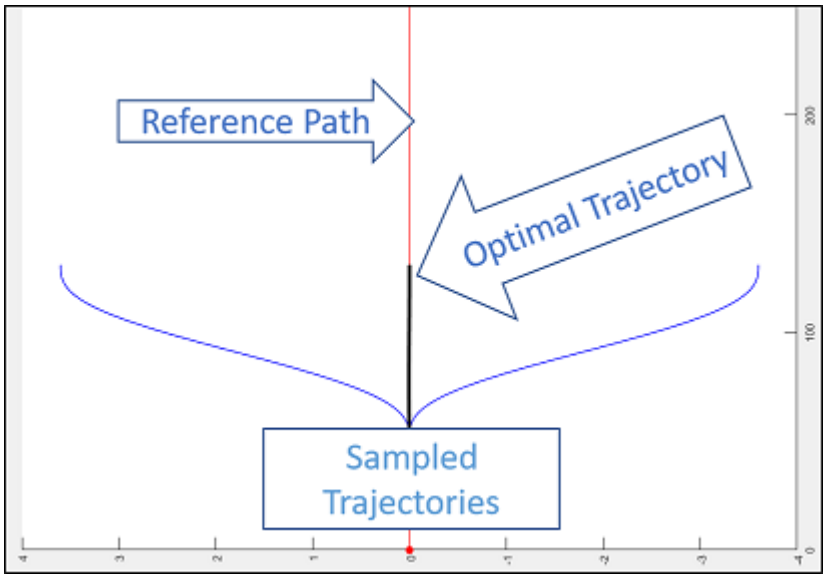
- The **VelocitySampler** block configures the possible velocity terminal states for defining the planner behavior. It is set to a default configuration in this model.
- The **AccelerationSampler** block configures the possible acceleration terminal states for defining the planner behavior. It is set to a default configuration in this model.
- The **Weights** bus defines the attributes that can be configured to vary the trajectory profile generated by the planner. These are set to default values in this model.

The **Motion Planner** reference model generates a trajectory by using the planner behavior, scenario information, and MIO information. Open the **Motion Planner** reference model.

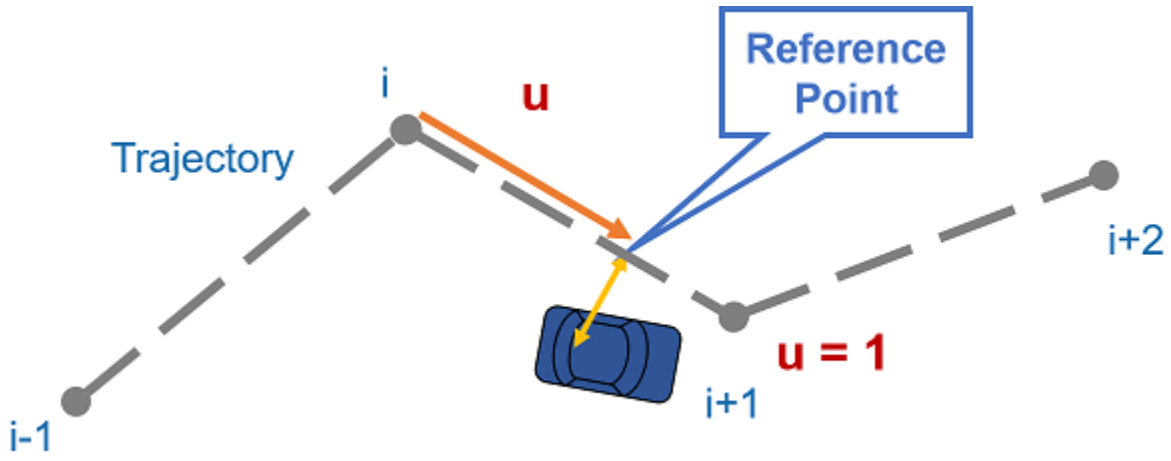
```
open_system('MotionPlanner')
```



- The **Pulse Generator** block defines a re-plan period for the **Trajectory Generator** block. The default value is set to 1 second.
- The **Trajectory Generator** block generates an optimal trajectory based on MIOs, the reference path, and the specified planner behavior. This block uses a system object, `HelperTrajectoryGenerator`, to generate the required trajectory. This System object is implemented using `trajectoryOptimalFrenet` (Navigation Toolbox). It uses MIO information updated using a state validator. Multiple trajectory samples are validated for dynamic collision using the state validator and the optimal trajectory is identified.



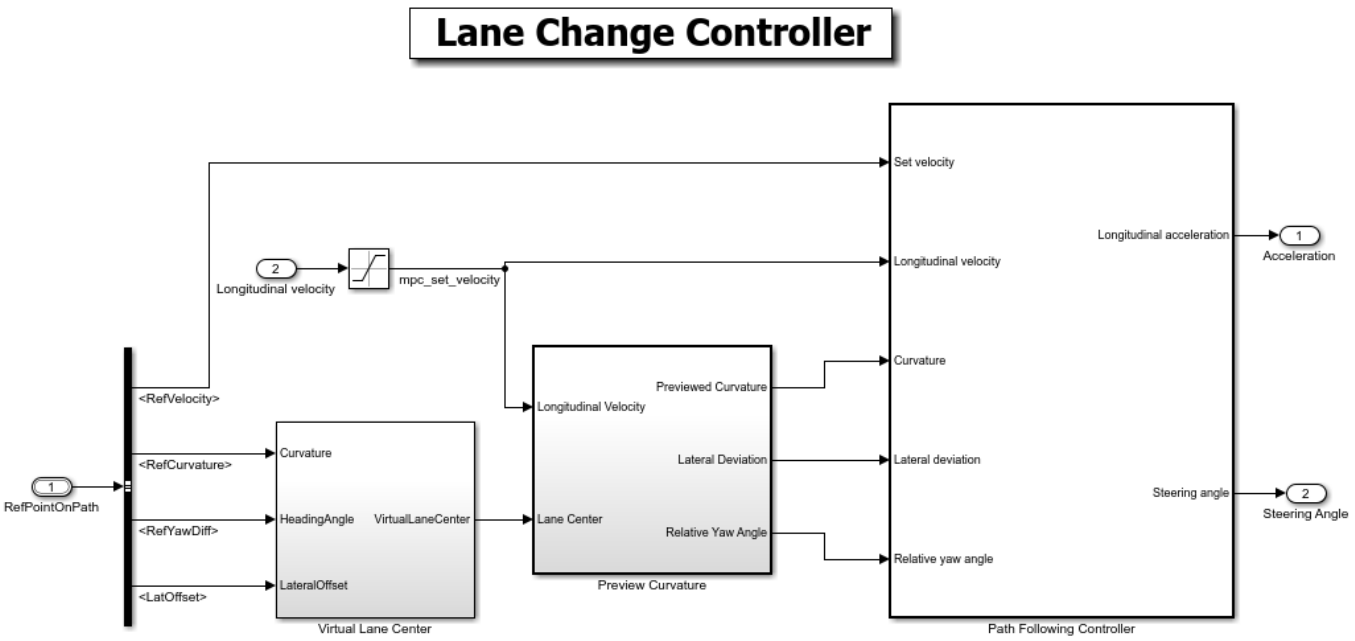
- The **Extract States** block extracts path information from the generated trajectory.
- The **Path Analyzer** block estimates the heading angle and finds the appropriate point on the path to follow. The generated path must conform to the road shape. This reference point on the path is used by the **Lane Change Controller** reference model.



Model Lane Change Controller

The **Lane Change Controller** reference model simulates a path following control mechanism that keeps the ego vehicle traveling along the generated trajectory while tracking a set velocity. To do so, the controller adjusts both the longitudinal acceleration and front steering angle of the ego vehicle. The controller computes optimal control actions while satisfying velocity, acceleration, and steering angle constraints using adaptive model predictive control (MPC). Open the **Lane Change Controller** reference model.

```
open_system('LaneChangeController')
```



Copyright 2019-2020 The MathWorks, Inc.

- The **Virtual Lane Center** subsystem creates a virtual lane from the path point. The virtual lane matches the format required by the **Path Following Controller** block.
- The **Preview Curvature** subsystem converts trajectory to curvature input required by **Path Following Controller**.
- The **Path Following Controller** block uses the Path Following Control System (Model Predictive Control Toolbox) block from the Model Predictive Control Toolbox™.

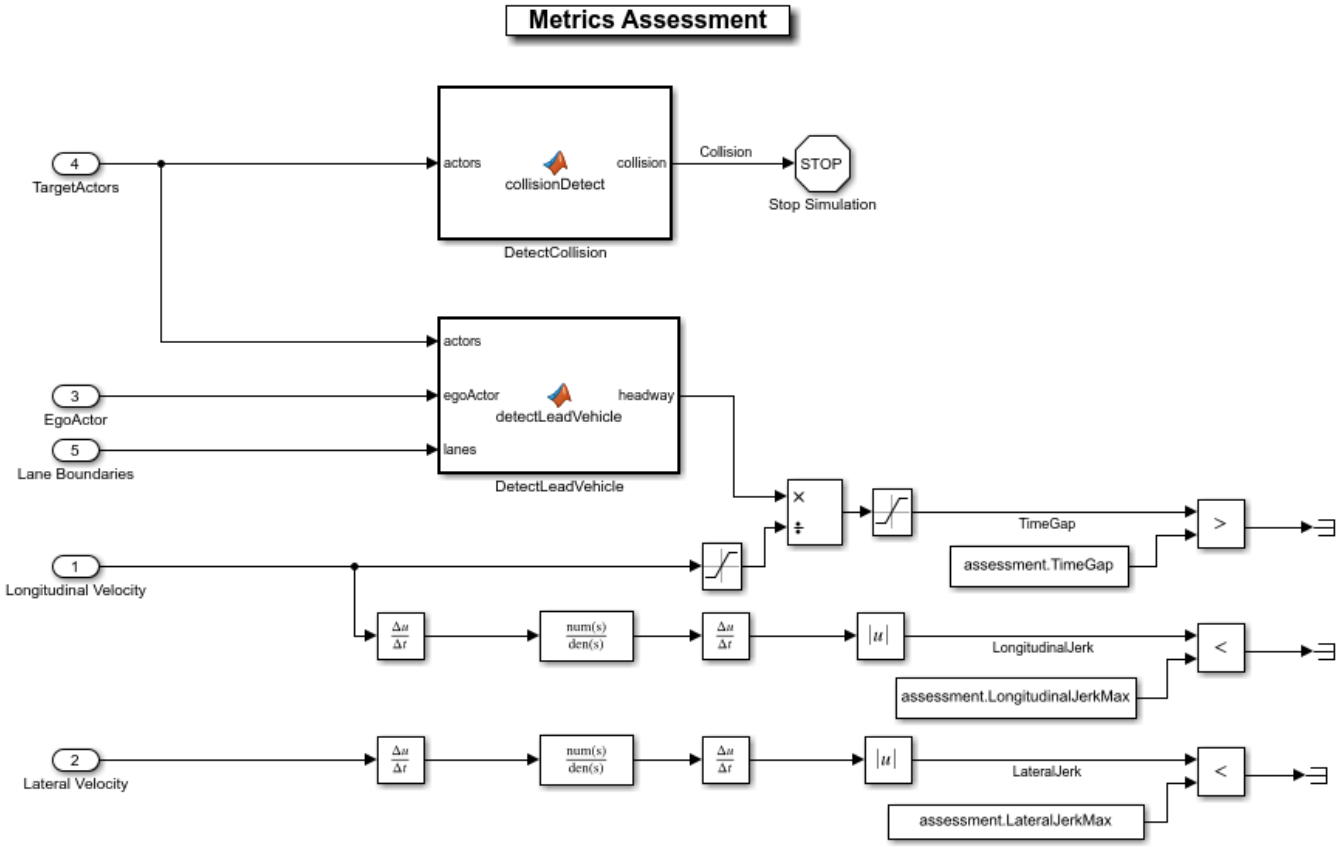
The **Path Following Controller** block keeps the vehicle traveling within a marked lane of a highway while maintaining a user-set velocity. This controller includes combined longitudinal and lateral control of the ego vehicle:

- Longitudinal control maintains a user-set velocity of the ego vehicle.
- Lateral control keeps the ego vehicle traveling along the center line of its lane by adjusting the steering of the ego vehicle.

Explore Metrics Assessment

The **Metrics Assessment** subsystem assesses system level behavior of the LCM system using the metrics mentioned below. Open the **Metrics Assessment** subsystem.

```
open_system('HighwayLaneChangeTestBench/Metrics Assessment')
```



- The **DetectCollision** block detects the collision of the ego vehicle with other vehicles and halts the simulation if a collision is detected.
- The **DetectLeadVehicle** block computes the headway between the ego and lead vehicles, which is used for computing the **TimeGap**.
- The **TimeGap** is calculated using the distance to the lead vehicle (headway) and the longitudinal velocity of the ego vehicle, and it is evaluated against prescribed limits.
- The **LongitudinalJerk** is calculated using the longitudinal velocity and evaluated against prescribed limits.
- The **LateralJerk** value is calculated using the lateral velocity evaluated against prescribed limits.

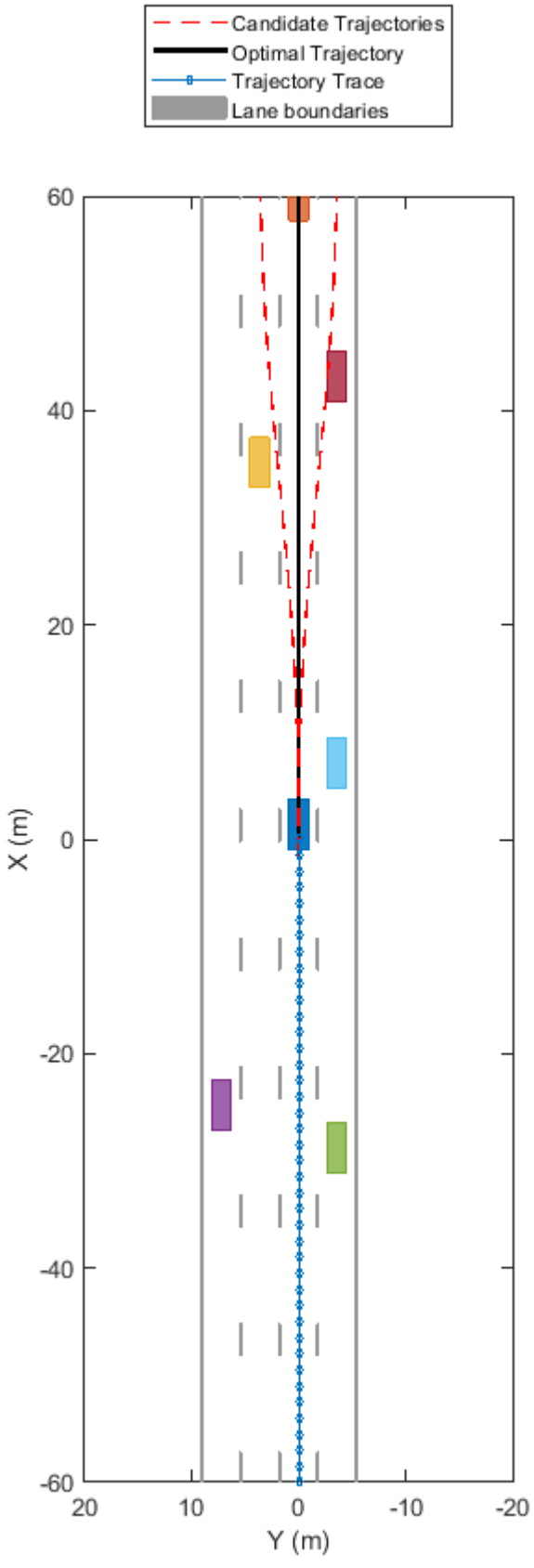
Simulate and Visualize System Behavior

Set up and run the `HighwayLaneChangeTestBench` simulation model to visualize the behavior of the system during a lane change. The **Visualization** block in the model creates a bird's eye plot that displays the lane information, ego vehicle, ego trajectory, and other vehicles in the scenario. Configure the `HighwayLaneChangeTestBench` model to use `scenario_LC_06_DoubleLaneChange` scenario.

```
helperSLHighwayLaneChangeSetup("scenario_LC_06_DoubleLaneChange")
```

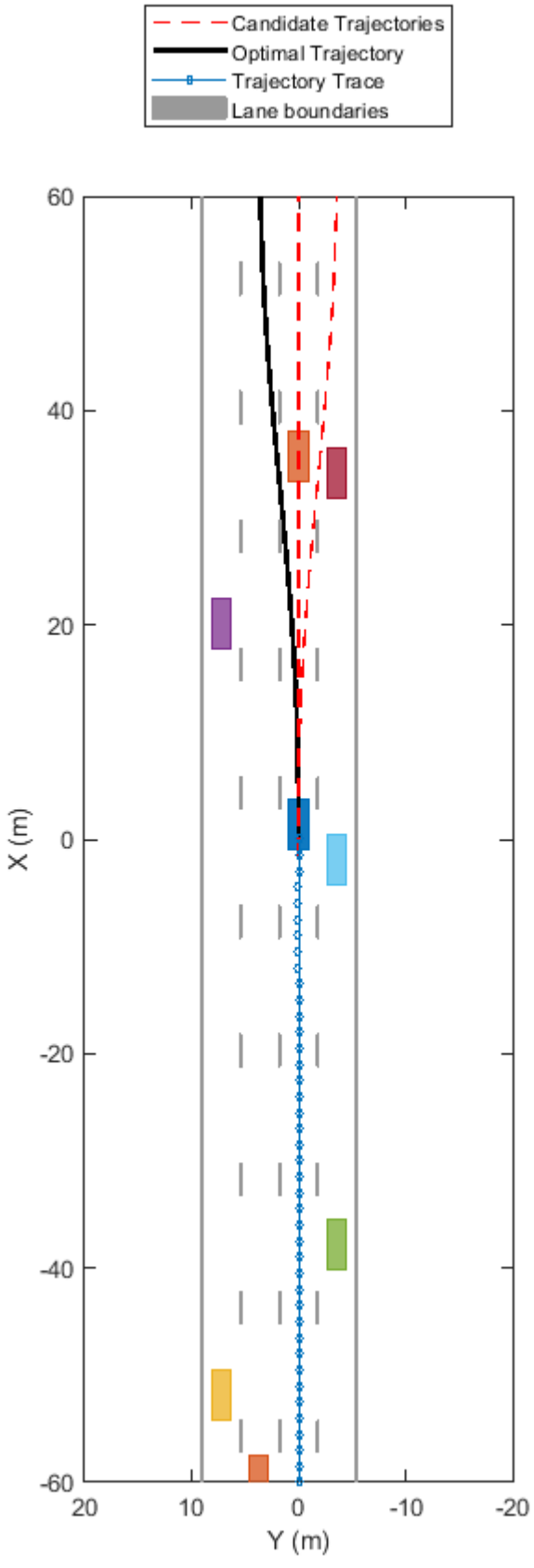
Simulate the model for 5 seconds. To reduce command-window output, first turn off the MPC update messages.

```
mpcverbosity('off');
sim("HighwayLaneChangeTestBench","StopTime","5");
```


Candidate Trajectories
Optimal Trajectory
Trajectory Trace
Lane boundaries
60
40
20
0
-20
-40
-60
X (m)
20
10
0
-10
-20
Y (m)

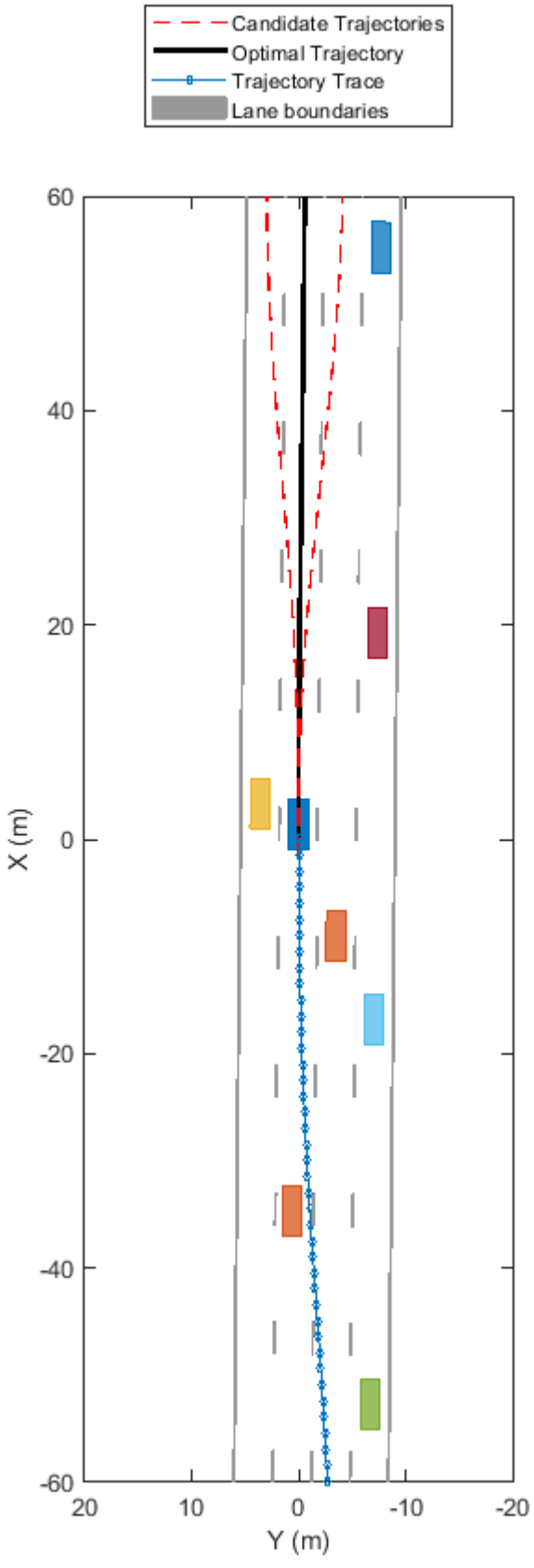
Run the simulation for 8 seconds. A trajectory is calculated to navigate around a slower lead vehicle.

```
sim("HighwayLaneChangeTestBench","StopTime","8");
```


Candidate Trajectories
Optimal Trajectory
Trajectory Trace
Lane boundaries
60
40
20
0
-20
-40
-60
X (m)
20
10
0
-10
-20
Y (m)

Run the simulation for 13 seconds. The vehicle continues straight ahead in the left lane.

```
sim("HighwayLaneChangeTestBench","StopTime","13");
```


Candidate Trajectories
Optimal Trajectory
Trajectory Trace
Lane boundaries
60
40
20
0
-20
-40
-60
X (m)
20
10
0
-10
-20
Y (m)

Explore Other Scenarios

In the previous section, you explored the system behavior for the `scenario_LC_06_DoubleLaneChange` scenario. Below is a list of scenarios that are compatible with the `HighwayLaneChangeTestBench` model.

```
scenario_LC_01_SlowMoving
scenario_LC_02_SlowMovingWithPassingCar
scenario_LC_03_DisabledCar
scenario_LC_04_CutInWithBrake
scenario_LC_05_SingleLaneChange
scenario_LC_06_DoubleLaneChange [Default]
scenario_LC_07_RightLaneChange
scenario_LC_08_SlowmovingCar_Curved
scenario_LC_09_CutInWithBreak_Curved
scenario_LC_10_SingleLaneChange_Curved
```

These scenarios are created using the Driving Scenario Designer and are exported to a scenario file. Examine the comments in each file for more details on the road and vehicles in each scenario. You can configure the `HighwayLaneChangeTestBench` and workspace to simulate these scenarios using the `helperSLHighwayLaneChangeSetup` function. For example, you can configure the simulation for a curved road scenario.

```
helperSLHighwayLaneChangeSetup("scenario_LC_10_SingleLaneChange_Curved")
```

Conclusion

This example shows how to simulate a highway lane change maneuver using ideal vehicle positions and lane detections.

Enable the MPC update messages again.

```
mpcverbosity('on');
```

See Also

`trajectoryOptimalFrenet`

More About

- “Highway Lane Following” on page 7-653

Design Lane Marker Detector Using Unreal Engine Simulation Environment

This example shows how to use a 3D simulation environment to record synthetic sensor data, develop a lane marker detection system, and test that system under different scenarios. This simulation environment is rendered using the Unreal Engine® from Epic Games®.

Overview

Developing a reliable perception system can be very challenging. A visual perception system must be reliable under a variety of conditions, especially when it is used in a fully automated system that controls a vehicle. This example uses a lane detection algorithm to illustrate the process of using the 3D simulation environment to strengthen the design of the algorithm. The main focus of the example is the effective use of the 3D simulation tools rather than the algorithm itself. Therefore, this example reuses the perception algorithms from the “Visual Perception Using Monocular Camera” on page 7-78 example.

The Visual Perception Using Monocular Camera example uses recorded video data to develop a visual perception system that contains lane marker detection and classification, vehicle detection, and distance estimation. Use of the recorded video is a great start, but it is inadequate for exploring many other cases that can be more easily synthesized in a virtual environment. More complex scenarios can include complex lane change maneuvers, occlusion of lane markers due to other vehicles, and so on. Most importantly, closed-loop simulation involves both perception and control of the vehicle, both of which require either a virtual environment or a real vehicle. Additionally, testing up front with a real vehicle can be expensive, thus making the use of a 3D simulation environment very attractive.

This example takes the following steps to familiarize you with an approach to designing a visual perception algorithm:

1. Introduces you to the 3D simulation environment in Simulink®
2. Guides you through the setup of a virtual vehicle and camera sensor
3. Shows you how to effectively set up a debugging environment for your visual perception algorithm
4. Presents how to increase scene complexity in preparation for closed-loop simulation

Introduction to the 3D Simulation Environment

Automated Driving Toolbox™ integrates a 3D simulation environment in Simulink. The 3D simulation environment uses the Unreal Engine by Epic Games. Simulink blocks related to the 3D simulation environment provide the ability to:

- Select different scenes in the 3D visualization engine
- Place and move vehicles in the scene
- Attach and configure sensors on the vehicles
- Simulate sensor data based on the environment around the vehicle

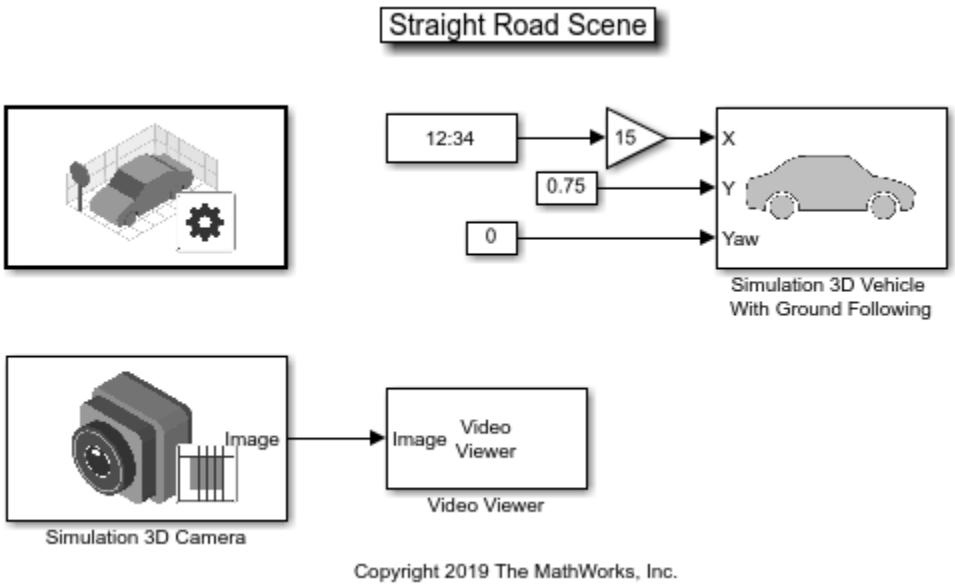
The Simulink blocks for 3D simulation can be accessed by opening `drivingsim3d` library.

To aid in the design of visual perception algorithms in this example, you use a block that defines a scene, a block that controls a virtual vehicle, and a block that defines a virtual camera. The example focuses on detecting lane markers using a monocular camera system.

Create a Simple Straight Road Scene in 3D Simulation

Start by defining a simple scenario involving a straight highway road on which to exercise the lane marker detection algorithm.

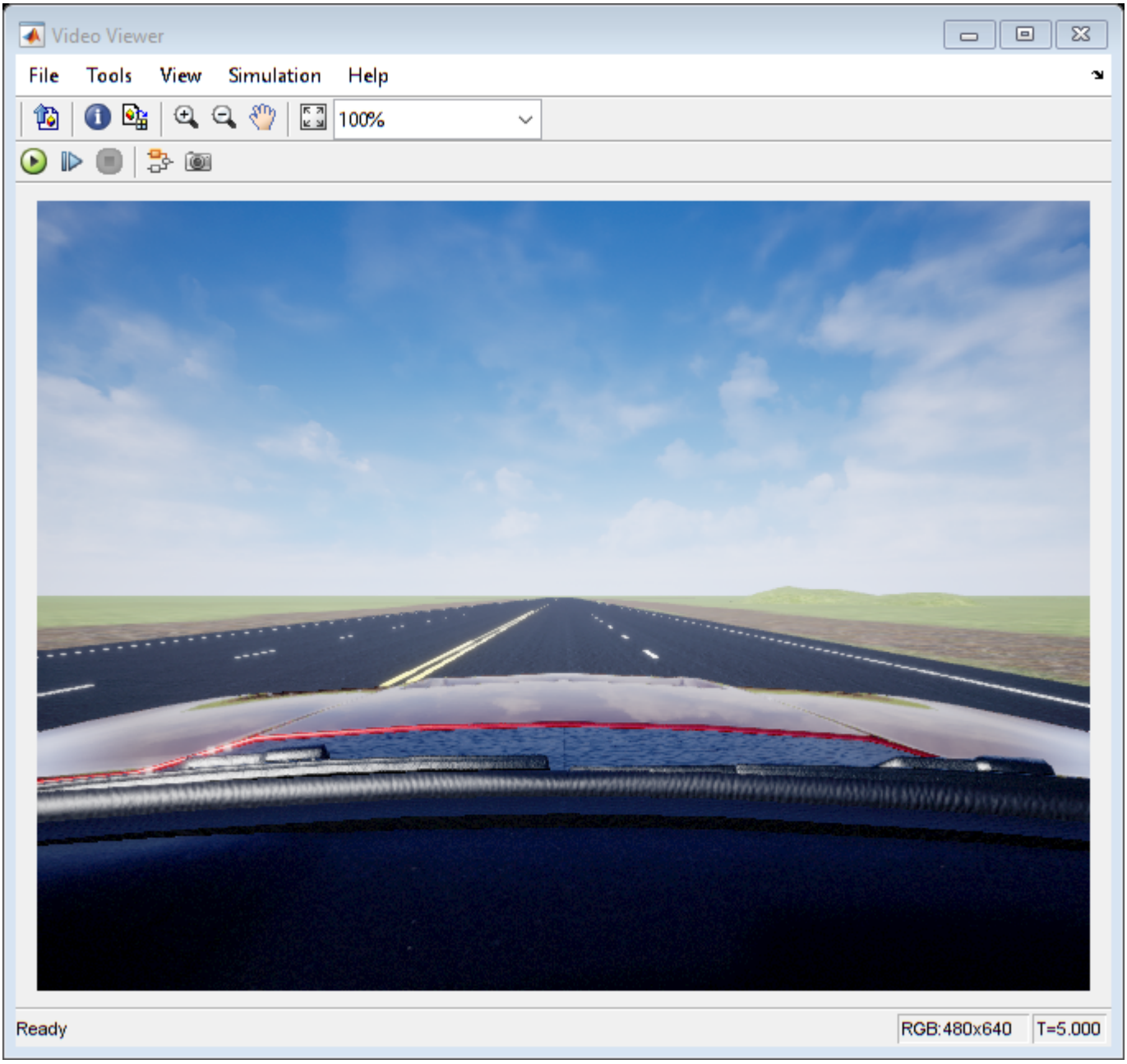
```
open_system('straightRoadSim3D');
```



The Simulation 3D Scene Configuration block lets you choose one of the predefined scenes, in this case Straight Road. When the model is invoked, it launches the Unreal Engine®. The Simulation 3D Vehicle with Ground Following block creates a virtual vehicle within the gaming engine and lets Simulink take control of its position by supplying **X** and **Y** in meters, and **Yaw** in degrees. **X**, **Y**, and **Yaw** are specified with respect to a world coordinate system, with an origin in the middle of the scene. In this case, since the road is straight, an offset of 0.75 meters in the *Y*-direction and a series of increasing *X* values move the vehicle forward. Later sections of this example show how to define more complex maneuvers without resorting to **X**, **Y**, and **Yaw** settings based on trial and error.

The model also contains a Simulation 3D Camera block, which extracts video frames from a virtual camera attached at the rearview mirror within the virtual vehicle. The camera parameters let you simulate typical parameters of a camera that can be described by a pinhole camera model, including focal length, camera optical center, radial distortion, and output image size. When the model is invoked, the resulting scene is shown from a perspective of a camera that automatically follows the vehicle.

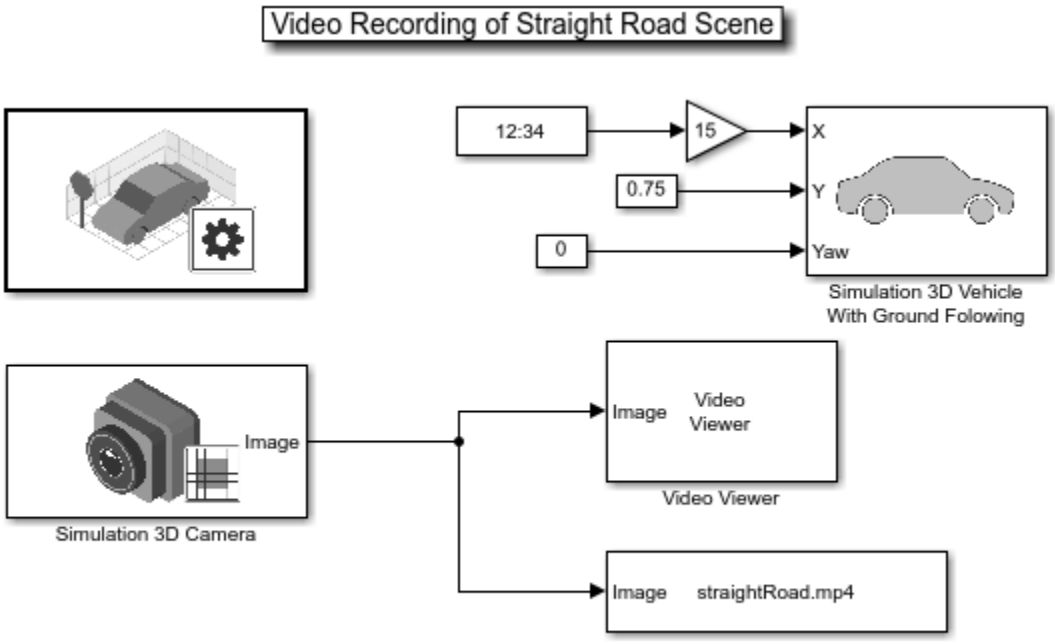
```
sim('straightRoadSim3D');
```



Design and Debugging of Visual Perception Module

Visual perception is generally complex, whether it involves classic computer vision or deep learning. Developing such a system often requires rapid iterations with incremental refinements. Although Simulink is a powerful environment for system-level engineering and closed-loop simulations, perception-based algorithms are typically developed in textual programming languages like MATLAB or C++. Additionally, the startup time for a model that needs to establish communication between Simulink and the Unreal Engine® is significant. For these reasons, it is convenient to record the image data generated by the virtual camera into a video and develop the perception algorithm in MATLAB. The following model records the camera into an MP4 file on disk.

```
open_system('straightRoadVideoRecording');
```

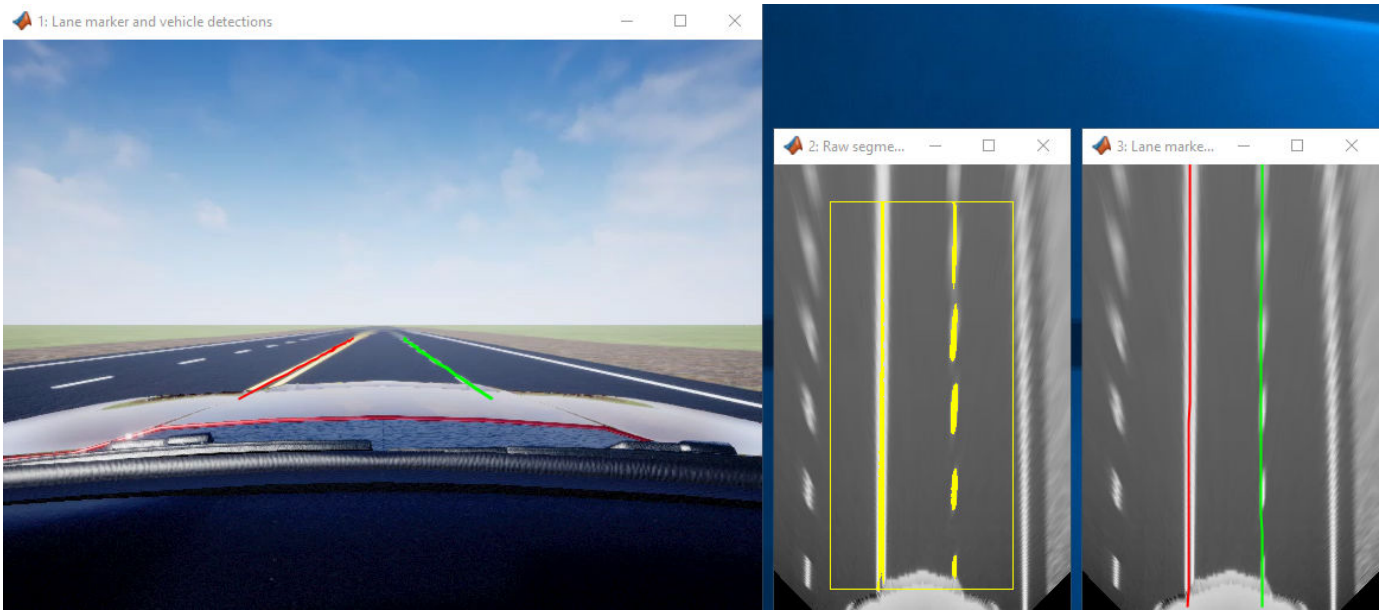


The video is recorded using the To Multimedia File (Computer Vision Toolbox) block. The resulting `straightRoad.mp4` file can now be used to develop the perception module, without incurring the startup-time penalty of the 3D simulation environment.

To design the lane marker detector, you use a module from the “Visual Perception Using Monocular Camera” on page 7-78 example. However, if you simply transplant the existing `helperMonoSensor.m` routine from that example, even the simplest straight road scene does not produce good results. Immediately, you can see how powerful the virtual environment can be. You can choose any trajectory or environment for your vehicle, thus letting you explore many what-if scenarios prior to placing the perception module on an actual vehicle.

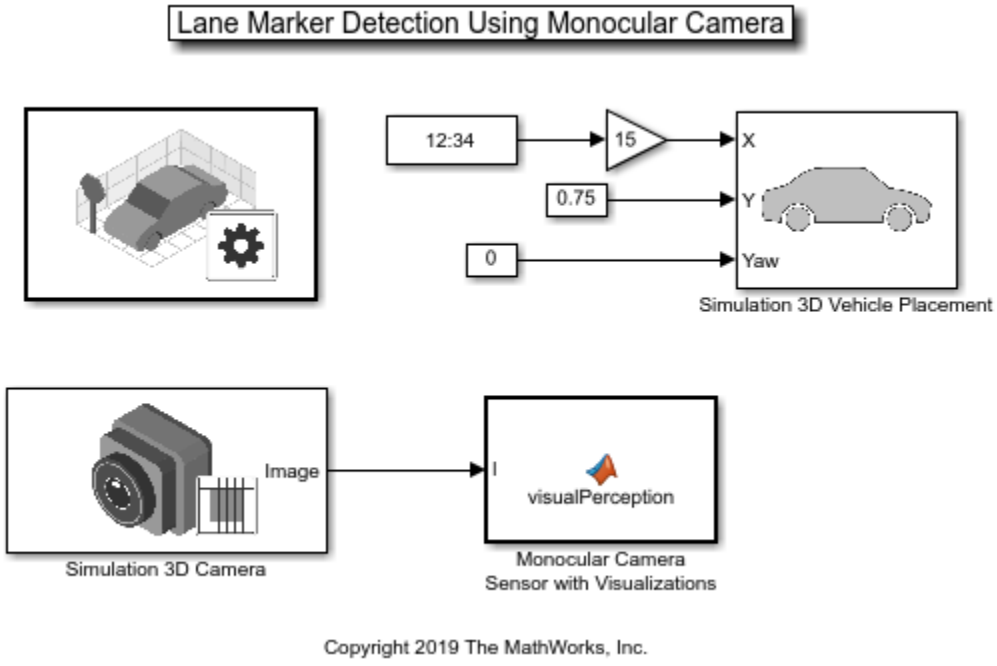
To aid in the design of the algorithm, use the provided `helperMonoSensorWrapper.m` function. This function works in MATLAB and when placed inside the MATLAB Function (Simulink) block in Simulink. The following script, `helperStraightRoadMLTest`, invokes the wrapper from the MATLAB command prompt. This approach permits quick iterations of the design without continuous invocation of the 3D simulation environment.

```
helperStraightRoadMLTest
```



Once the algorithm begins to work well, you can place it back into a model as shown below. You can attempt to change the car's trajectory, as demonstrated in the “Select Waypoints for Unreal Engine Simulation” on page 7-626 example. That way, you can look for ways to move the car such that the algorithm fails. The entire process is meant to be iterative.

```
open_system('straightRoadMonoCamera');
```



Navigate Through a More Complex Scene to Improve the Perception Algorithm

While developing your algorithm, you can increase the level of scene complexity to continue adapting your system to conditions resembling reality. In this section, switch the scene to Virtual Mcity, which provides stretches of the road with curved lanes, no lane markers, or merging lane markers.

Before you begin, you need to define a trajectory through a suitable stretch of the virtual Mcity, which is a representation of actual testing grounds that belong to the University of Michigan. To see the details of how to obtain a series of **X**, **Y**, and **Yaw** values suitable for moving a car through a complex environment, refer to the “Select Waypoints for Unreal Engine Simulation” on page 7-626 example. The key steps are summarized below for your convenience.

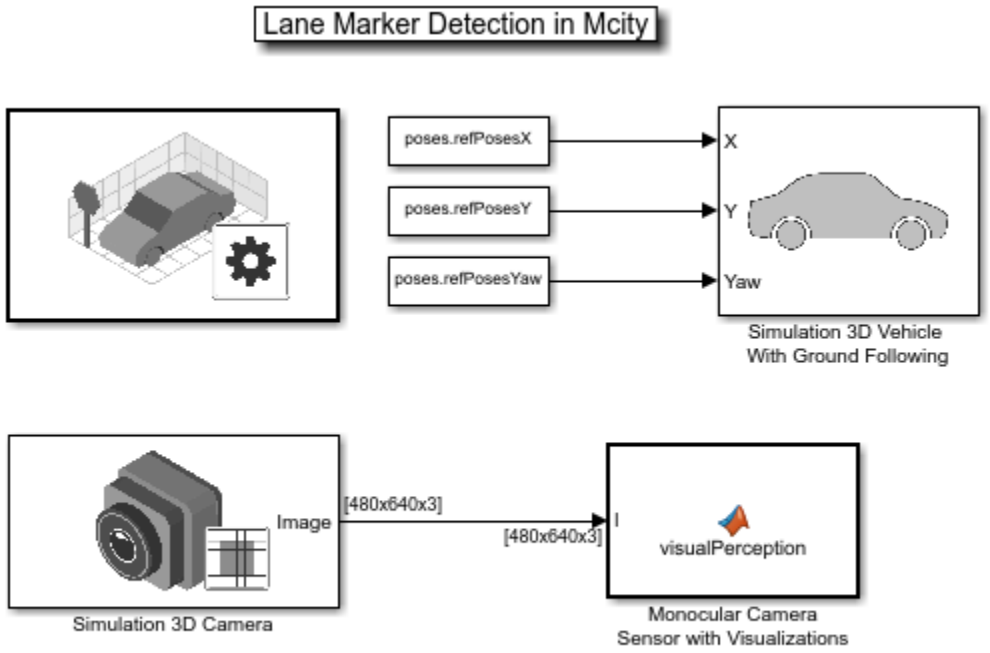
```
% Extract scene image location based on scene's name
sceneName = 'VirtualMCity';
[sceneImage, sceneRef] = helperGetSceneImage(sceneName);

% Interactively select waypoints through Mcity
helperSelectSceneWaypoints(sceneImage, sceneRef)

% Convert the sparse waypoints into a denser trajectory that a car can
% follow
numPoses = size(refPoses, 1);
refDirections  = ones(numPoses,1);   % Forward-only motion
numSmoothPoses = 20 * numPoses;      % Increase this to increase the number of returned poses
[newRefPoses,~,cumLengths] = smoothPathSpline(refPoses, refDirections, numSmoothPoses);

% Create a constant velocity profile by generating a time vector
% proportional to the cumulative path length
simStopTime = 10;
timeVector = normalize(cumLengths, 'range', [0, simStopTime]);

refPosesX   = [timeVector, newRefPoses(:,1)];
refPosesY   = [timeVector, newRefPoses(:,2)];
refPosesYaw = [timeVector, newRefPoses(:,3)];
```

Load the preconfigured vehicle poses created using the method shown above.

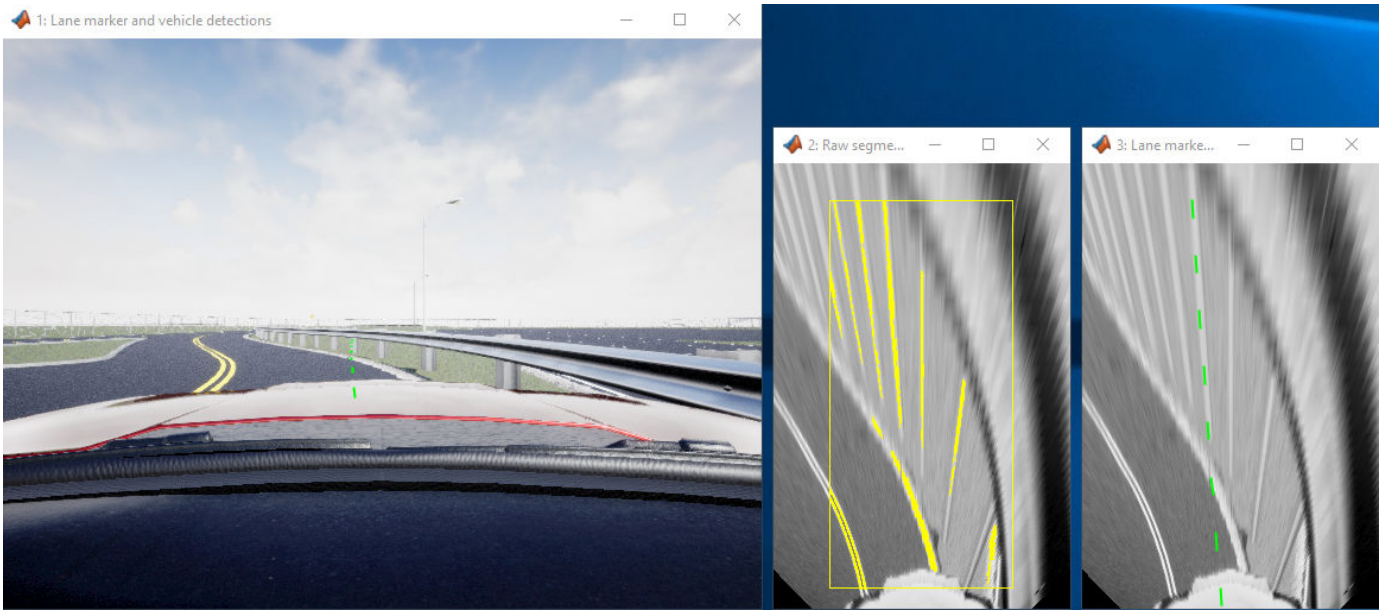
```
poses = load('mcityPoses');
```

With the predefined trajectory, you can now virtually drive the vehicle through a longer stretch of a complex virtual environment.

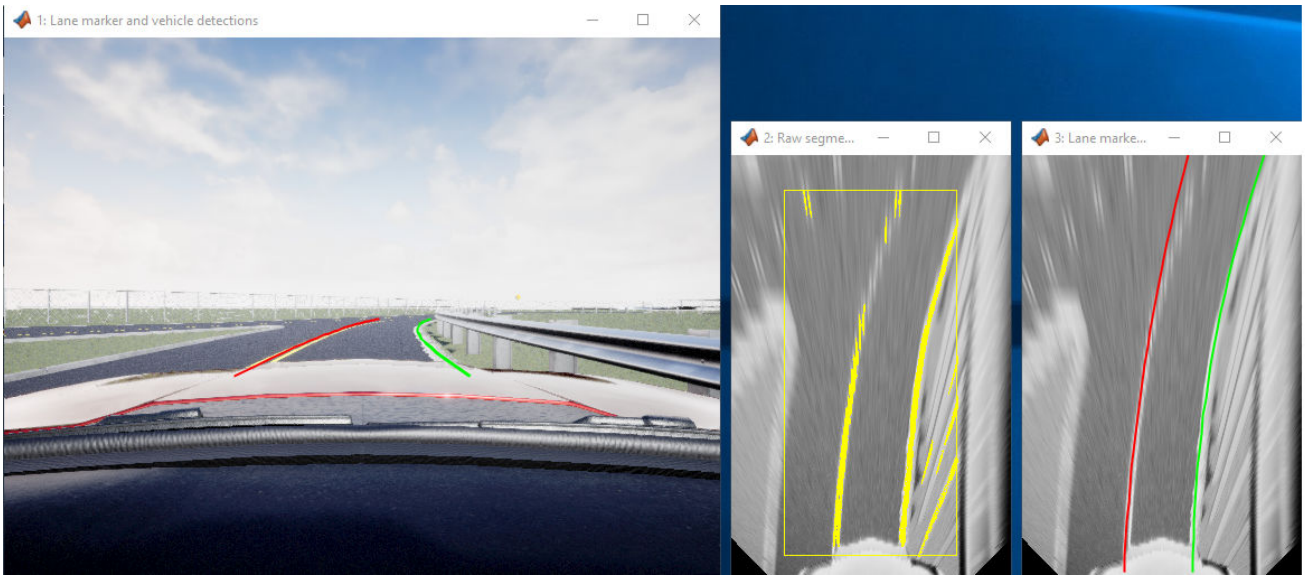
```
open_system('mcityMonoCamera');
sim('mcityMonoCamera');
clear poses;
```



Many times, the results are less than desirable. For example, notice where the barriers are confused with lane markers and when the region of interest selected for analysis is too narrow to pick up the left lane.



However, the detector performs well in other areas of the scene.



The main point is that the virtual environment lets you stress-test your design and helps you realize what kind of conditions you may encounter on real roads. Running your algorithm in a virtual environment also saves you time. If your design does not run successfully in the virtual environment, then there is no point of running it in a real vehicle on a road, which is far more time-consuming and expensive.

Closed-Loop Testing

One of the most powerful features of a 3D simulation environment is that it can facilitate closed-loop testing of a complex system. Lane keep assist, for example, involves both perception and control of the vehicle. Once a perception system is perfected on very complex scenes and performs well, it can then be used to drive a control system that actually steers the car. In this case, rather than manually set up a trajectory, the vehicle uses the perception system to drive itself. It is beyond the scope of this example to show the entire process. However, the steps described here should provide you with ideas on how to design and debug your perception system so it can later be used in a more complex closed-loop simulation.

```
bdclose all;
```

See Also

Blocks

Simulation 3D Camera | Simulation 3D Fisheye Camera | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following

More About

- “Select Waypoints for Unreal Engine Simulation” on page 7-626
- “Choose a Sensor for Unreal Engine Simulation” on page 6-16
- “Simulate Simple Driving Scenario and Sensor in Unreal Engine Environment” on page 6-22
- “Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation” on page 6-31
- “Unreal Engine Simulation for Automated Driving” on page 6-2

- “Highway Lane Following” on page 7-653

Select Waypoints for Unreal Engine Simulation

This example shows how to select a sequence of waypoints from a scene and visualize the path of a vehicle following these waypoints in a 3D simulation environment. This environment uses the Unreal Engine® by Epic Games®.

Introduction

Automated Driving Toolbox™ integrates an Unreal Engine simulation environment in Simulink®. Simulink blocks related to the simulation environment can be found in the `drivingsim3d` library. These blocks provide the ability to:

- Select different scenes in the simulation environment.
- Place and move vehicles in the scene.
- Attach and configure sensors on the vehicles.
- Simulate sensor data based on the environment around the vehicle.

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

The first step in using this environment is understanding the scene and selecting waypoints along a desired vehicle path. This step is useful especially in scenarios where the localization algorithm is not under test. This example focuses on this first step.

In this example, you will:

- Visualize the scene in MATLAB®.
- Interactively select waypoints along a path in the scene.
- Set up the simulation environment.
- Move the vehicle along the path.

Visualize Scene

First, visualize the scene in MATLAB. Each scene can be visualized using a 2D top-view projection of the scene onto an image. Each scene image has a corresponding 2D spatial referencing object of class `imref2d (Image Processing Toolbox)` describing the relationship between the pixels in the image and the world coordinates of the scene. Use the `helperGetSceneImage` function to retrieve the scene image and associated spatial reference. This example uses a prebuilt scene of a large parking lot. To use generate a scene image and spatial reference for a custom scene, follow the process described in “Create Top-Down Map of Unreal Engine Scene” on page 6-63 instead.

```
sceneName = 'LargeParkingLot';
[sceneImage, sceneRef] = helperGetSceneImage(sceneName);
```

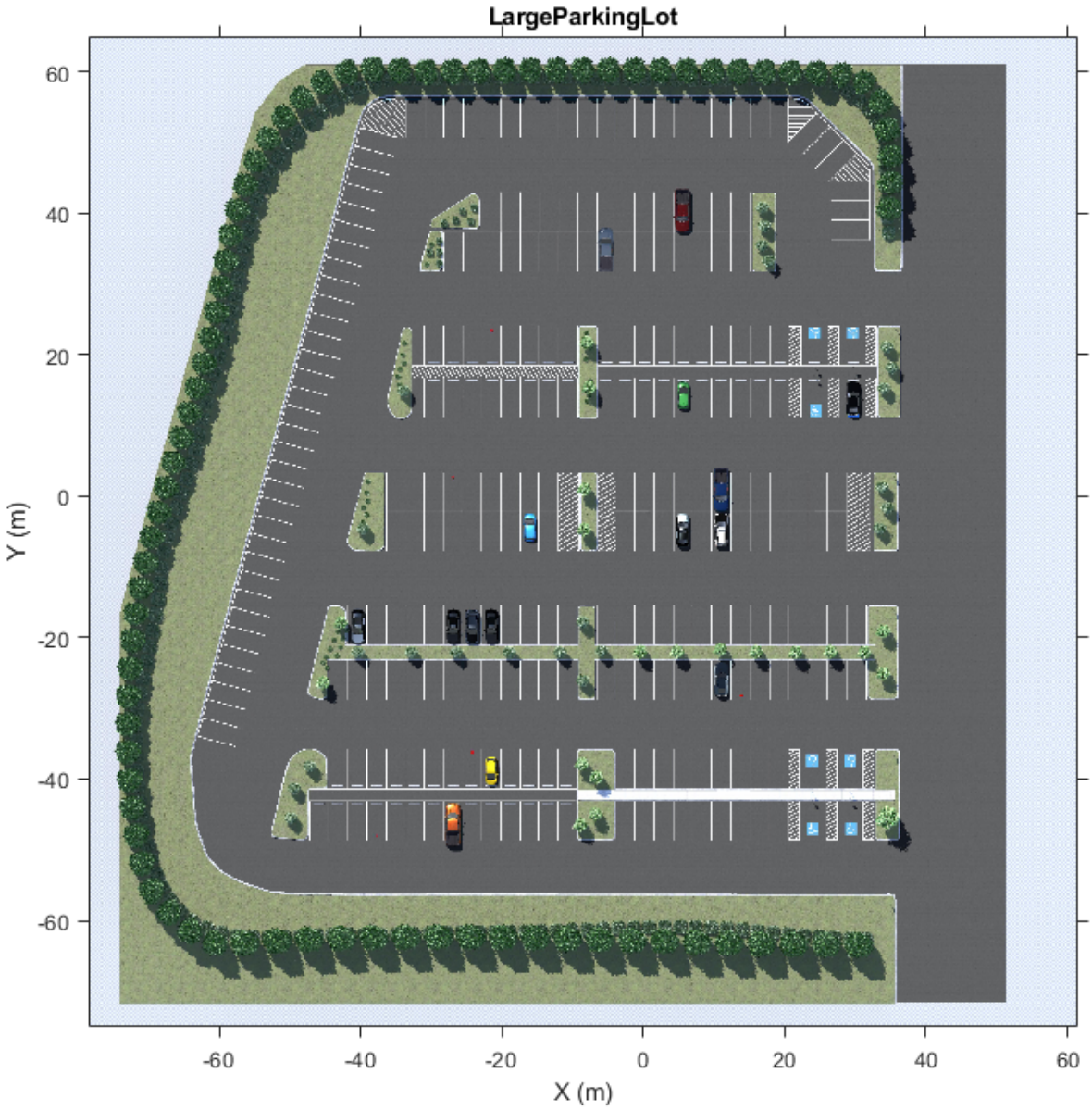
To better understand the physical dimensions of the scene, inspect the `sceneRef` variable. The `XWorldLimits` and `YWorldLimits` properties specify the limits of the world in the X and Y directions.

```
sceneRef.XWorldLimits   % (in meters)
sceneRef.YWorldLimits   % (in meters)
```

```
ans =

  -78.5000   61.5000


ans =

   -75    65
```

Visualize the scene image by using the `helperShowSceneImage` function. This helper function displays the scene image in a figure window. Use the pan and zoom tools to explore the scene.

```
hScene = figure;
helperShowSceneImage(sceneImage, sceneRef)
title(sceneName)
```



Interactively Select Waypoints

After exploring the scene, select a set of waypoints to define a path for a vehicle to follow. This path can be used to move the vehicle in the scene. Use the helper function `helperSelectSceneWaypoints` to interactively select waypoints in the scene.

```
hFig = helperSelectSceneWaypoints(sceneImage, sceneRef);
```

This helper function opens a figure window with the selected scene.

- Explore the scene by zooming and panning through the scene image. Use the mouse scrollwheel or the axes toolbar to zoom. Hover over the edge of the axes to pan in that direction.
- Begin drawing a path by clicking on the scene. A path is created as a polyline consisting of multiple points. Finish drawing the path by double-clicking or right-clicking.
- Once you are done drawing a path, click **Export to Workspace** to export the variables to the MATLAB workspace. In the dialog box that opens, click **OK** to export a set of variables to the workspace.

The following data is exported to the workspace as MATLAB variables:

- **Waypoints:** A cell array with each element containing an *M*-by-2 matrix of (x, y) waypoints in world coordinates. Each element of the cell array corresponds to the waypoints from a different path.
- **Path Poses:** A cell array with each element containing *M*-by-3 matrices of (x, y, θ) poses containing the pose of each waypoint. x and y are specified in meters. θ is specified in degrees.

```
% Load variables to workspace if they do not exist
if exist('refPoses', 'var')==0 || exist('wayPoints', 'var')==0

    % Load MAT-file containing preselected waypoints
    data = load('waypointsLargeParkingLot');
    data = data.waypointsLargeParkingLot;

    % Assign to caller workspace
    assignin('caller', 'wayPoints', {data.waypoints});
    assignin('caller', 'refPoses', {data.refPoses});
end
```

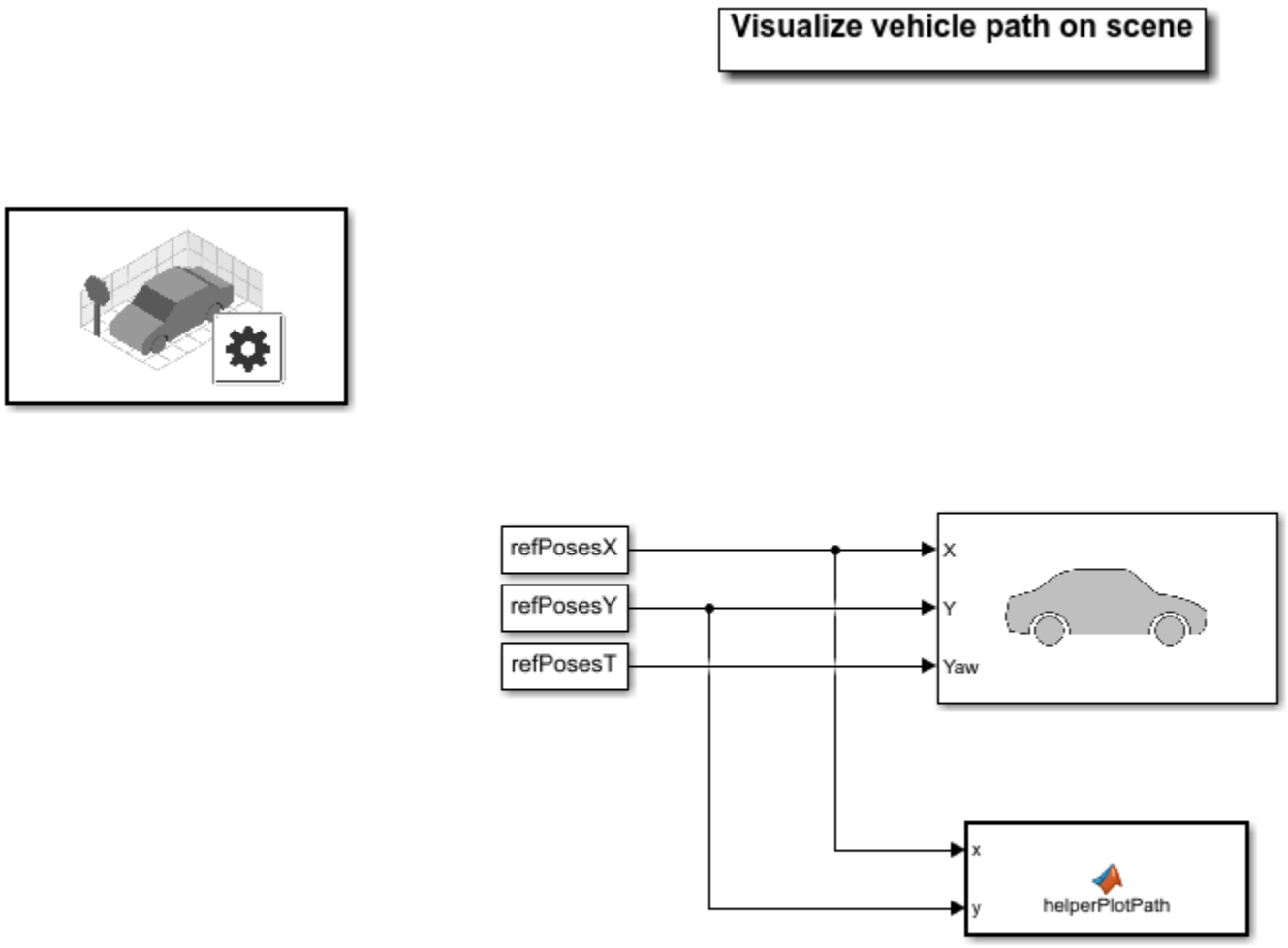
The exported variables now contain a sequence of waypoints (`wayPoints`) and a sequence of poses (`refPoses`). Use the `smoothPathSpline` function to transform the sequence of poses to a C^2 continuous path.

```
numPoses = size(refPoses{1}, 1);

refDirections  = ones(numPoses,1);   % Forward-only motion
numSmoothPoses = 20 * numPoses;      % Increase this to increase the number of returned poses

[smoothRefPoses,~,cumLengths] = smoothPathSpline(refPoses{1}, refDirections, numSmoothPoses);
```

Set Up Model and Simulation Environment

Open the `VisualizeVehiclePathIn3DSimulation` Simulink model. This model uses the Simulation 3D Scene Configuration block to select a desired scene. This example uses the Large Parking Lot scene. The Simulation 3D Scene Configuration block sets up the environment and establishes communication between Simulink and the simulation environment.

```
if ~ispc
    error(['3D Simulation is only supported on Microsoft', char(174), ' Windows', char(174), '.'])
end

modelName = 'VisualizeVehiclePathIn3DSimulation';
open_system(modelName);
snapnow;
```



Move Vehicle Along Path

Use the Simulation 3D Vehicle with Ground Following block to place and move vehicles in the scene. The model is set up to accept variables `refPosesX`, `refPosesY`, and `refPosesT` from the workspace using the From Workspace (Simulink) block. Separate x, y and θ from `newRefPoses` into separate time series. The model reads these workspace variables to update the position of the vehicle.

```
% Configure the model to stop simulation at 5 seconds.
simStopTime = 5;
set_param(gcs, 'StopTime', num2str(simStopTime));

% Create a constant velocity profile by generating a time vector
% proportional to the cumulative path length.
timeVector = normalize(cumLengths, 'range', [0, simStopTime]);
```

```
% Create variables required by the Simulink model
refPosesX = [timeVector, smoothRefPoses(:,1)];
refPosesY = [timeVector, smoothRefPoses(:,2)];
refPosesT = [timeVector, smoothRefPoses(:,3)];
```

When you simulate the model, a few seconds are needed to initialize the simulation environment. Once this initialization is complete, a separate window opens for the simulation environment visualization. The image below is a snapshot of the simulation environment window.

Run the simulation. The figure window plot shows the path that the vehicle traverses through the simulation environment.

```
sim(modelName);
```


```
% Close the model and figure windows
close(hFig)
close_system(modelName)
close(hScene)
```

See Also

Blocks

Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following

Functions

`imref2d`

More About

- “Create Top-Down Map of Unreal Engine Scene” on page 6-63
- “Design Lane Marker Detector Using Unreal Engine Simulation Environment” on page 7-617
- “Visualize Automated Parking Valet Using Unreal Engine Simulation” on page 7-635
- “Simulate Vision and Radar Sensors in Unreal Engine Environment” on page 7-647
- “Unreal Engine Simulation for Automated Driving” on page 6-2
- “Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox” on page 6-10

Visualize Automated Parking Valet Using Unreal Engine Simulation

This example shows how to visualize vehicle motion in a 3D simulation environment rendered using the Unreal Engine® from Epic Games®. It closely follows the “Automated Parking Valet in Simulink” on page 7-493 example.

Introduction

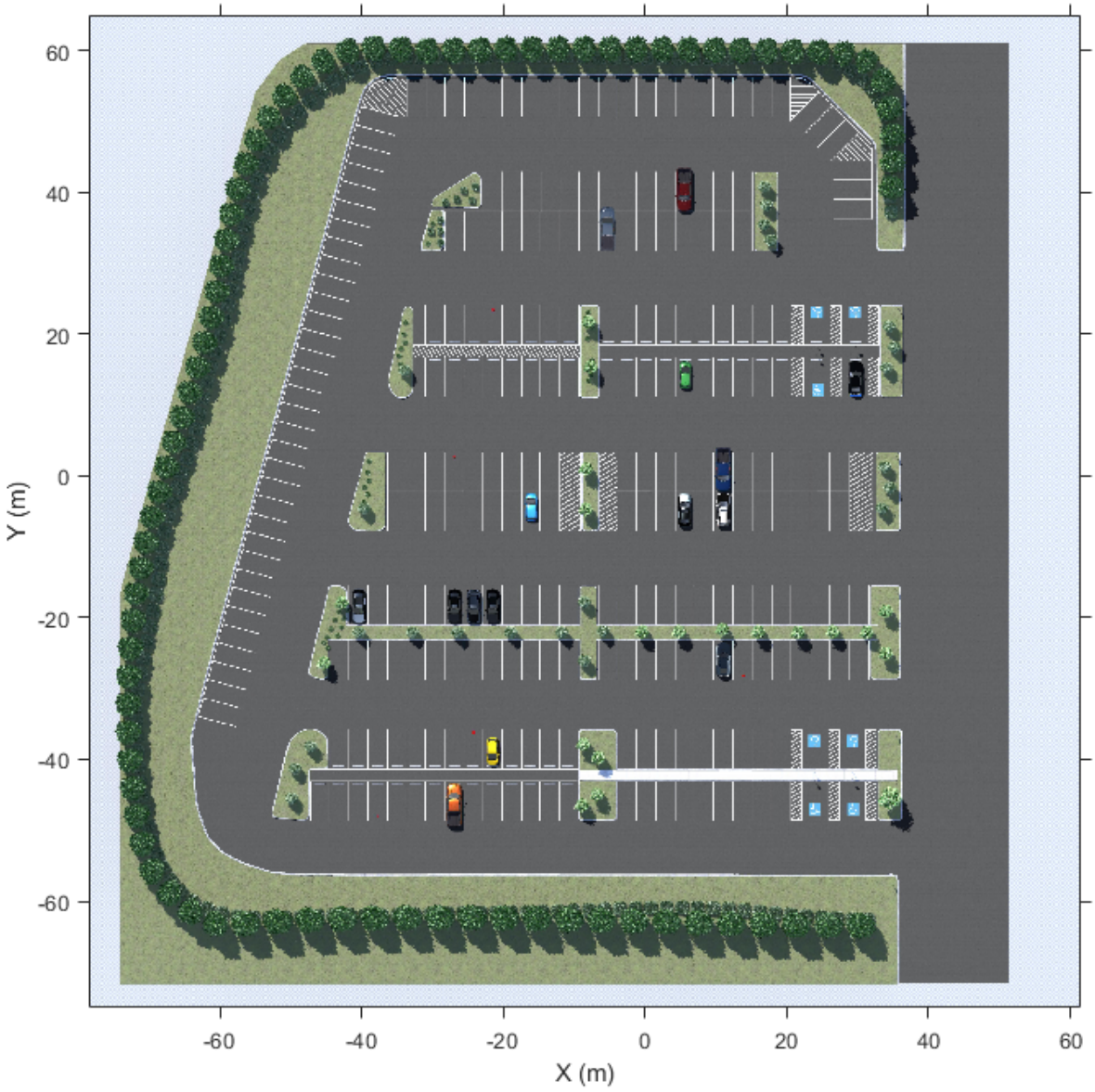
Automated Driving Toolbox™ integrates an Unreal Engine simulation environment in Simulink®. You can use this environment to visualize the motion of a vehicle in a prebuilt scene. This environment provides an intuitive way to analyze the performance of path planning and vehicle control algorithms. The “Automated Parking Valet in Simulink” on page 7-493 example shows how to design a path planning and vehicle control algorithm for an automated parking valet system in Simulink. This example shows how to augment the model to visualize the vehicle motion in a scene using the visualization engine. The steps in this workflow are:

1. Create a costmap from a 3D scene.
2. Create a route plan from the scene.
3. Configure the 3D scene and ego vehicle in Simulink.
4. Simulate and visualize the vehicle's motion in the 3D scene.

Create Costmap from 3D Scene

The visualization integration comes with a number of prebuilt scenes. Each scene comes with a high-resolution image that can be used to explore the scene. Use the `helperShowSceneImage` function to display the images. This example uses the Large Parking Lot scene.

```
% Load and display the image of the parking lot
sceneName = 'LargeParkingLot';
[sceneImage, sceneRef] = helperGetSceneImage(sceneName);

% Visualize the scene image
figure
helperShowSceneImage(sceneImage, sceneRef)
```



Such a high-resolution image is an accurate depiction of the environment up to some resolution. You can use this image to create a `vehicleCostmap` for path planning and navigation.

First, estimate free space from the image. Free space is the area where a vehicle can drive without collision with other static objects, such as parked cars, cones, and road boundaries, and without crossing marked lines. In this example, you can estimate the free space based on the color of the image. Use the Color Thresholder (Image Processing Toolbox) app from Image Processing Toolbox to perform the segmentation and generate a binary image from the image. You can also use the helper function `helperCreateCostmapFromImage` at the end of the example to generate the binary image:

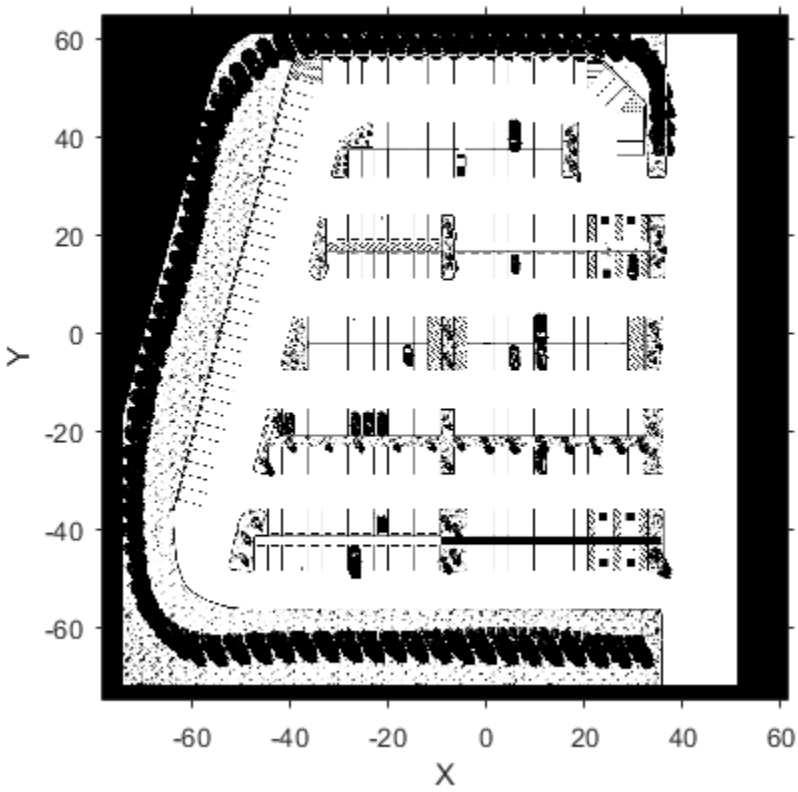
```
sceneImageBinary = helperCreateCostmapFromImage(sceneImage);
```

Alternatively, load a pregenerated binary image.

```
sceneImageBinary = imread('sim3d_LargeParkingLotBinary.bmp');
```

Next, create a costmap from the binary image. Use the binary image to specify the cost value at each cell.

```
% Get the left-bottom corner location of the map
mapLocation = [sceneRef.XWorldLimits(1), sceneRef.YWorldLimits(1)]; % [meters, meters]

% Compute resolution
mapWidth = sceneRef.XWorldLimits(2)-sceneRef.XWorldLimits(1); % meters
cellSize = mapWidth/size(sceneImageBinary, 2);

% Create the costmap
costmap = vehicleCostmap(im2single(sceneImageBinary), 'CellSize', cellSize, 'MapLocation', mapLo

figure
plot(costmap, 'Inflation', 'off');
legend off
```



You also need to specify the dimensions of the vehicle that will park automatically based on the vehicles available in the 3D scene. This example uses the dimension of a Hatchback. These dimensions need to be consistent between the costmap and the Simulink model.

```
centerToFront = 1.104; % meters
centerToRear  = 1.343; % meters
frontOverhang = 0.828; % meters
rearOverhang  = 0.589; % meters
vehicleWidth  = 1.653; % meters
```

```
vehicleHeight = 1.513; % meters
vehicleLength = centerToFront + centerToRear + frontOverhang + rearOverhang;

vehicleDims = vehicleDimensions(vehicleLength, vehicleWidth, vehicleHeight,...
    'FrontOverhang', frontOverhang, 'RearOverhang', rearOverhang);
costmap.CollisionChecker.VehicleDimensions = vehicleDims;
```

Set the inflation radius by specifying the number of circles enclosing the vehicle.

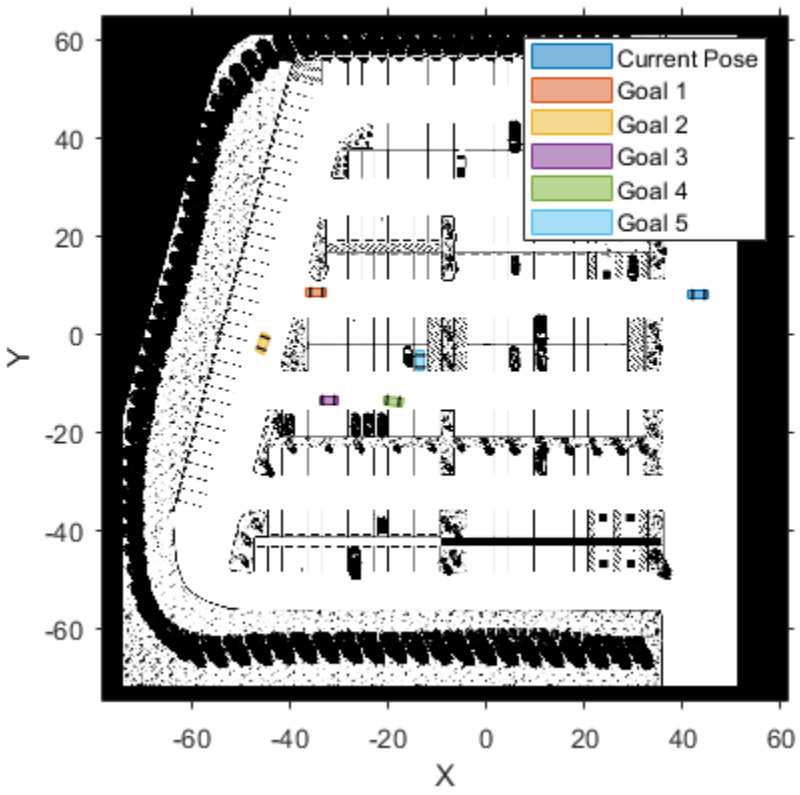
```
costmap.CollisionChecker.NumCircles = 5;
```

Create Route Plan from a 3D Scene

The global route plan is described as a sequence of lane segments to traverse in order to reach a parking spot. You can interactively select intermediate goal positions from the scene image using the tool described in “Select Waypoints for Unreal Engine Simulation” on page 7-626. In this example, the route plan has been created and stored in a table. Before simulation, the `PreLoadFcn` callback function of the model loads the route plan.

```
data      = load('routePlanUnreal.mat');
routePlan = data.routePlan %#ok<NOPTS>

% Plot vehicle at the starting pose
startPose = routePlan.StartPose(1,:);
hold on
helperPlotVehicle(startPose, vehicleDims, 'DisplayName', 'Current Pose')
legend

for n = 1 : height(routePlan)
    % Extract the goal waypoint
    vehiclePose = routePlan{n, 'EndPose'};

    % Plot the pose
    legendEntry = sprintf('Goal %i', n);
    helperPlotVehicle(vehiclePose, vehicleDims, 'DisplayName', legendEntry);
    hold on
end
hold off
```

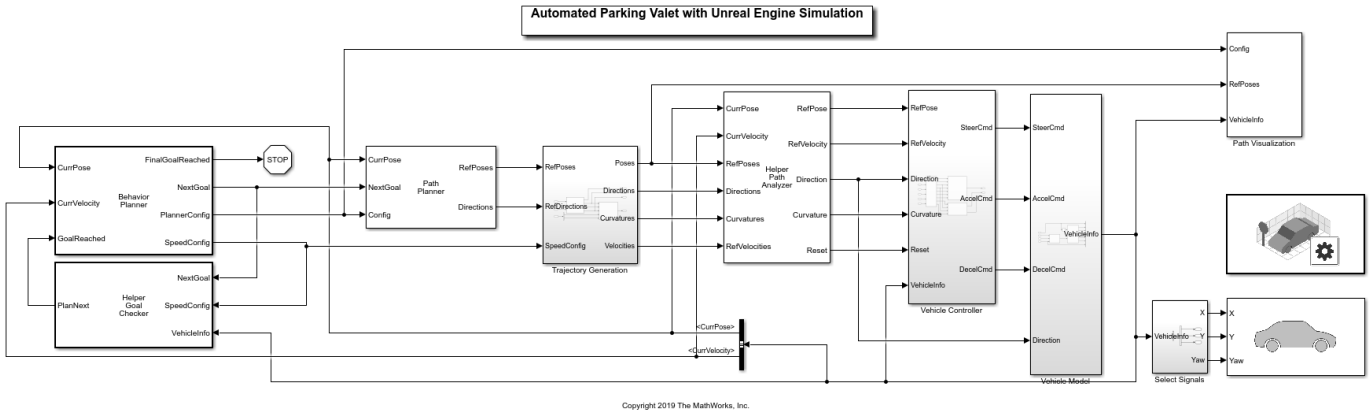
```
routePlan =

  5×3 table

          StartPose                 EndPose              Attributes
    ____________________________    ____________________________    ____________

     44.5        8      180    -33.5      8.5      180    [1×1 struct]
    -33.5      8.5      180    -45.2     -0.7      250    [1×1 struct]
    -45.2     -0.7      250    -33.5    -13.5        0    [1×1 struct]
    -33.5    -13.5        0    -20.3    -13.5       -7    [1×1 struct]
    -20.3    -13.5       -7    -13.5     -6.8       90    [1×1 struct]
```



Configure 3D Scene and Ego Vehicle

Close the figures and open the model.

```
helperCloseFigures

if ~ispc
    error(['3D Simulation is only supported on Microsoft', char(174), ' Windows', char(174), '.'])
end

modelName = 'AutomatedParkingValetWith3DSimulation';
open_system(modelName)
snapnow
```



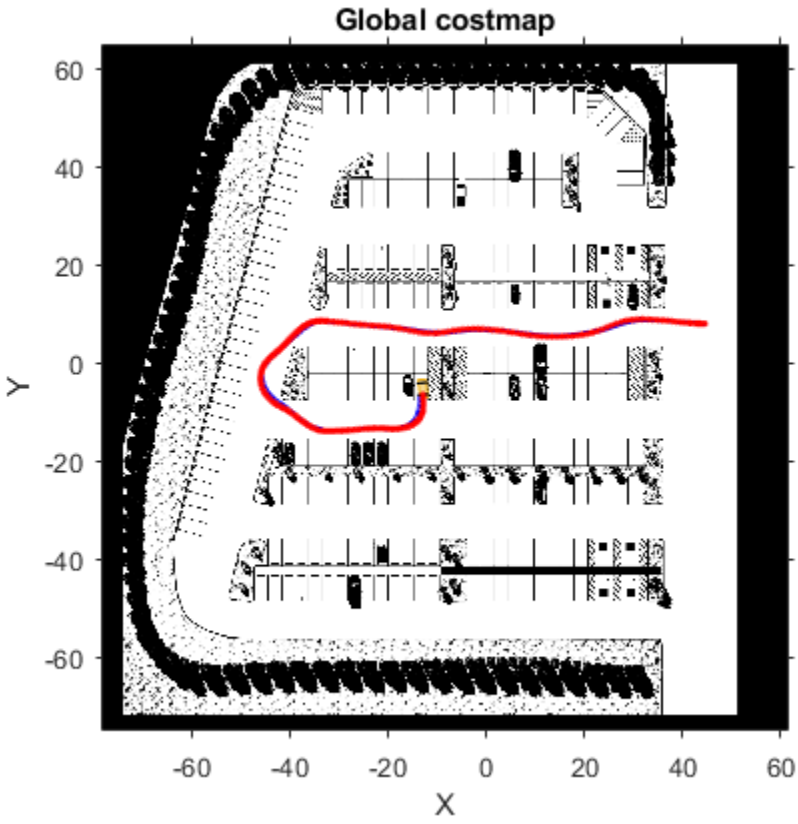
This model extends the one used in the Automated Parking Valet in Simulink example by adding two blocks for visualizing the vehicle in the 3D scene:

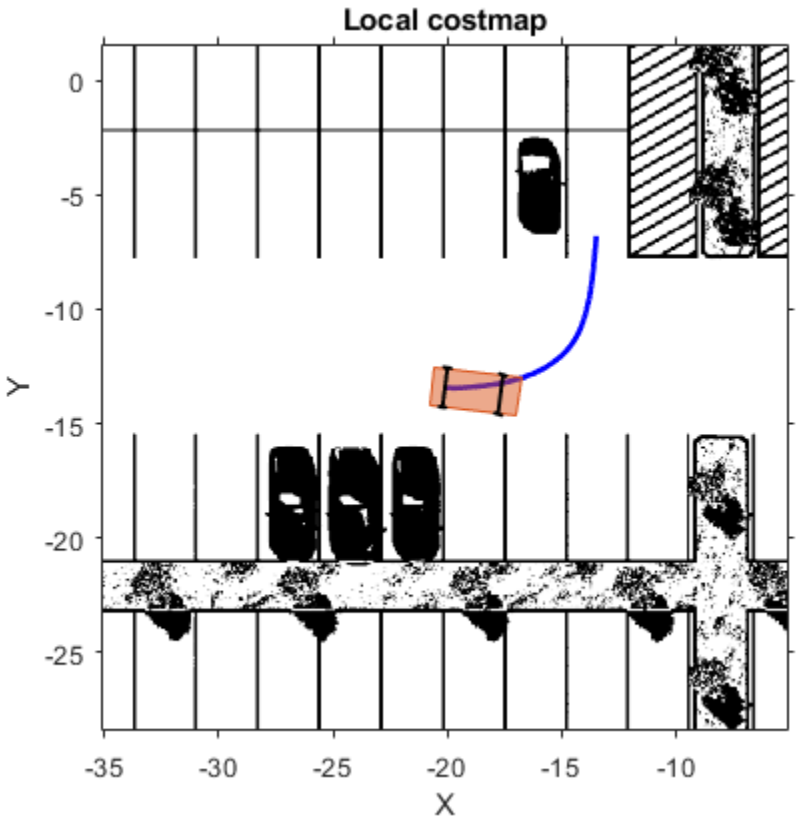
- **Simulation 3D Scene Configuration**: Implements the 3D simulation environment. The **Scene description** parameter is set to `Large parking lot`.
- **Simulation 3D Vehicle with Ground Following**: Provides an interface that changes the position and orientation of the vehicle in the 3D scene. The **Type** of the vehicle is set to `Hatchback` to be consistent with the vehicle dimensions in `costmap`. The inputs to this block are the vehicle's `[X, Y]` position in meters and the `Yaw` heading angle in degrees. These values are in the world coordinate system.

Visualize Vehicle Motion in 3D Scene

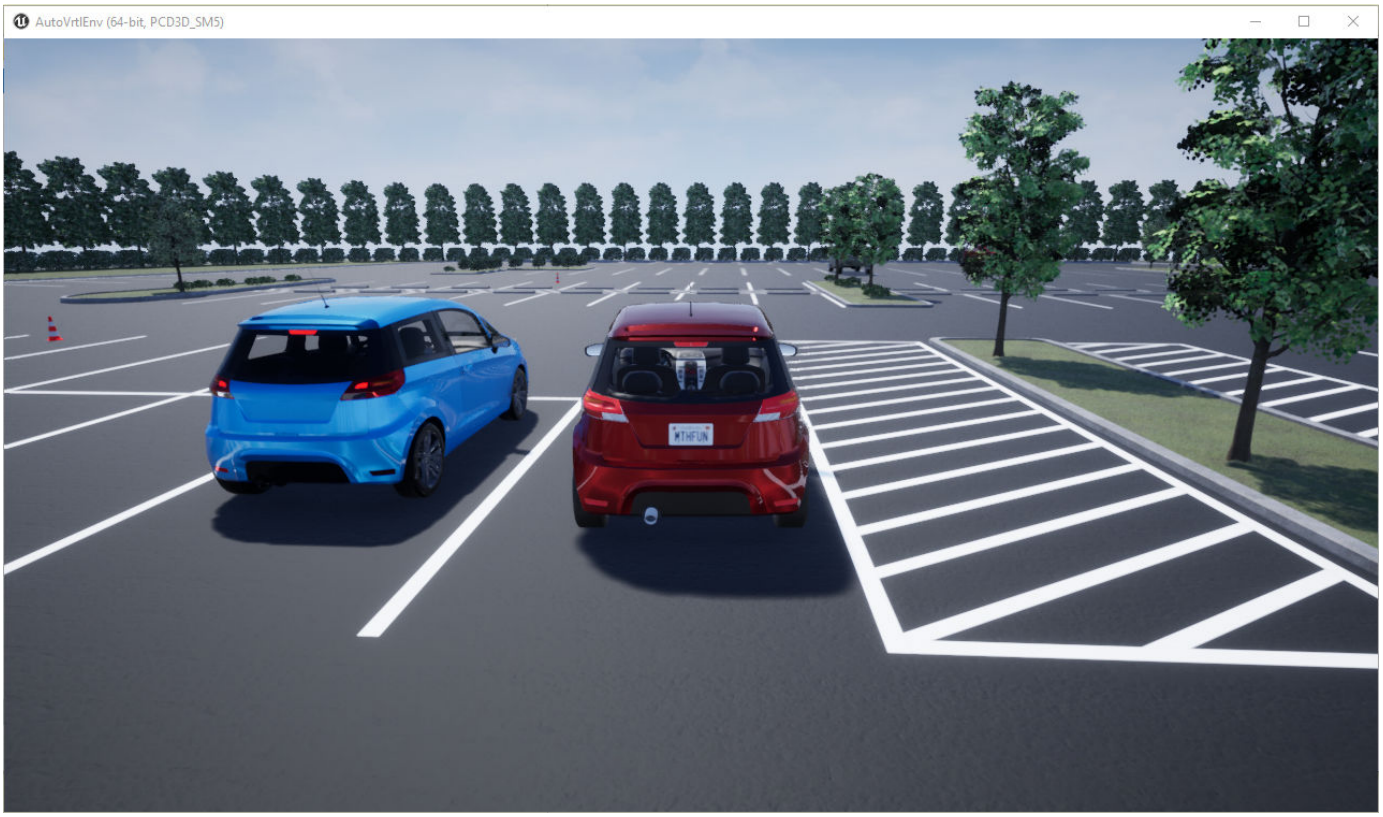
Simulate the model to see how the vehicle drives into the desired parking spot.

```
sim(modelName)
```


Global costmap
60
40
20
0
-20
-40
-60
Y
-60
-40
-20
0
20
40
60
X



As the simulation runs, the Simulink environment updates the position and orientation of the vehicle in the 3D visualization engine through the Simulation 3D Vehicle with Ground Following block. A new window shows the ego vehicle in the 3D visualization engine. The `Automated Parking Valet` figure displays the planned path in blue and the actual path of the vehicle in red. The `Parking Maneuver` figure shows a local costmap used in searching for the final parking maneuver.



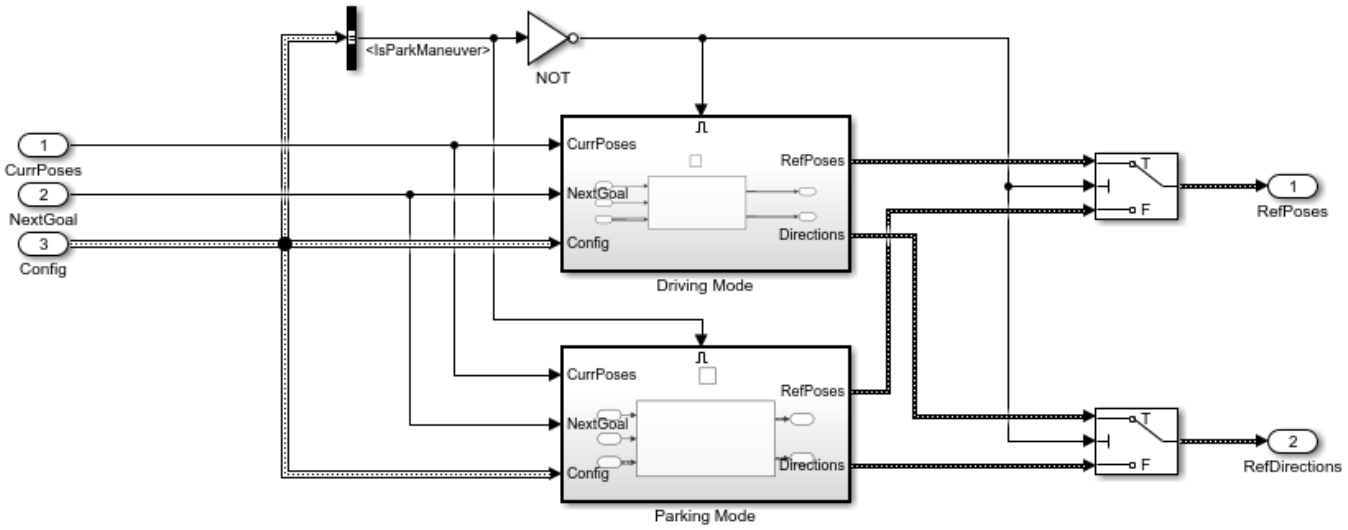
Explore Enhanced Path Planning System

The Path Planner block plans a feasible path through the environment map using the optimal rapidly exploring random tree (RRT*) algorithm. To ensure the performance of the planning algorithm, the path planning module is modified to include two separate modules:

- **Driving Mode**: Uses the costmap of the entire parking lot for navigation. This implementation is also used in the "Automated Parking Valet in Simulink" on page 7-493 example.
- **Parking Mode**: Uses a local costmap for the final parking maneuver. The local costmap is a submap of the costmap with a square shape. To specify the size of this map, use the **Local costmap size (m)** parameter of the Path Planner block dialog box. A costmap with smaller dimensions significantly reduces the computation burden in searching for a feasible path to the final parking spot. It also increases the probability of finding a feasible path given the same planner settings.

Open the Path Planner subsystem.

```
open_system([modelName, '/Path Planner'], 'force')
```



The two path planner modules are implemented as Enabled Subsystem (Simulink) blocks. The enable signal is from the `IsParkingManeuver` signal in the input `Config` bus sent from the Behavior Planner block. When the Parking Mode subsystem is enabled, a local costmap is created with the center as the current position of the vehicle.

Close the model and the figures.

```
bdclose all
helperCloseFigures
```

Conclusions

This example showed how to integrate 3D simulation with the existing Automated Parking Valet in Simulink example to visualize the motion of vehicle in a 3D parking lot scene.

Supporting Functions

helperCreateCostmapFromImage

```
function BW = helperCreateCostmapFromImage(sceneImage) %#ok<DEFNU>
%helperCreateCostmapFromImage Create a costmap from an RGB image.

% Flip the scene image
sceneImage = flipud(sceneImage);

% Call the autogenerated code from the Color Thresholder app
BW = helperCreateMask(sceneImage);

% Smooth the image
BW = im2uint8(medfilt2(BW));

% Resize
BW = imresize(BW, 0.5);

% Compute complement
BW = imcomplement(BW);
end
```

helperCreateMask

```
function [BW,maskedRGBImage] = helperCreateMask(RGB)
%helperCreateMask  Threshold RGB image using auto-generated code from Color Thresholder app.
%  [BW,maskedRGBImage] = createMask(RGB) thresholds image RGB using
%  autogenerated code from the Color Thresholder app. The colorspace and
%  range for each channel of the colorspace were set within the app. The
%  segmentation mask is returned in BW, and a composite of the mask and
%  original RGB image is returned in maskedRGBImage.

% Convert RGB image to chosen colorspace
I = RGB;

% Define thresholds for channel 1 based on histogram settings
channel1Min = 67.000;
channel1Max = 216.000;

% Define thresholds for channel 2 based on histogram settings
channel2Min = 68.000;
channel2Max = 171.000;

% Define thresholds for channel 3 based on histogram settings
channel3Min = 69.000;
channel3Max = 160.000;

% Create mask based on chosen histogram thresholds
sliderBW = (I(:,:,1) >= channel1Min ) & (I(:,:,1) <= channel1Max) & ...
    (I(:,:,2) >= channel2Min ) & (I(:,:,2) <= channel2Max) & ...
    (I(:,:,3) >= channel3Min ) & (I(:,:,3) <= channel3Max);
BW = sliderBW;

% Initialize output masked image based on input image
maskedRGBImage = I;

% When BW is false, set background pixels to zero
maskedRGBImage(repmat(~BW,[1 1 3])) = 0;
end
```

helperCloseFigures

```
function helperCloseFigures()
%helperCloseFigures Close all the figures except the simulation visualization

% Find all the figure objects
figHandles = findobj('Type', 'figure');

% Close the figures
for i = 1: length(figHandles)
    close(figHandles(i));
end
end
```

See Also

Blocks

Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following

Functions
`vehicleCostmap` | `vehicleDimensions`

Apps
Color Thresholder

More About

- “Select Waypoints for Unreal Engine Simulation” on page 7-626
- “Automated Parking Valet in Simulink” on page 7-493
- “Unreal Engine Simulation for Automated Driving” on page 6-2

Simulate Vision and Radar Sensors in Unreal Engine Environment

This example shows how to implement a synthetic data simulation for tracking and sensor fusion in Simulink® with using the Unreal Engine® simulation environment from Epic Games®. It closely follows the “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” on page 7-205 example.

Introduction

Automated Driving Toolbox provides tools for authoring, simulating, and visualizing virtual driving scenarios. With these scenarios, you can simulate rare and potentially dangerous events, generate synthetic radar and vision detections from the scenarios, and use the synthetic detections to test vehicle algorithms. This example covers the entire synthetic data workflow in Simulink using the 3D simulation environment.

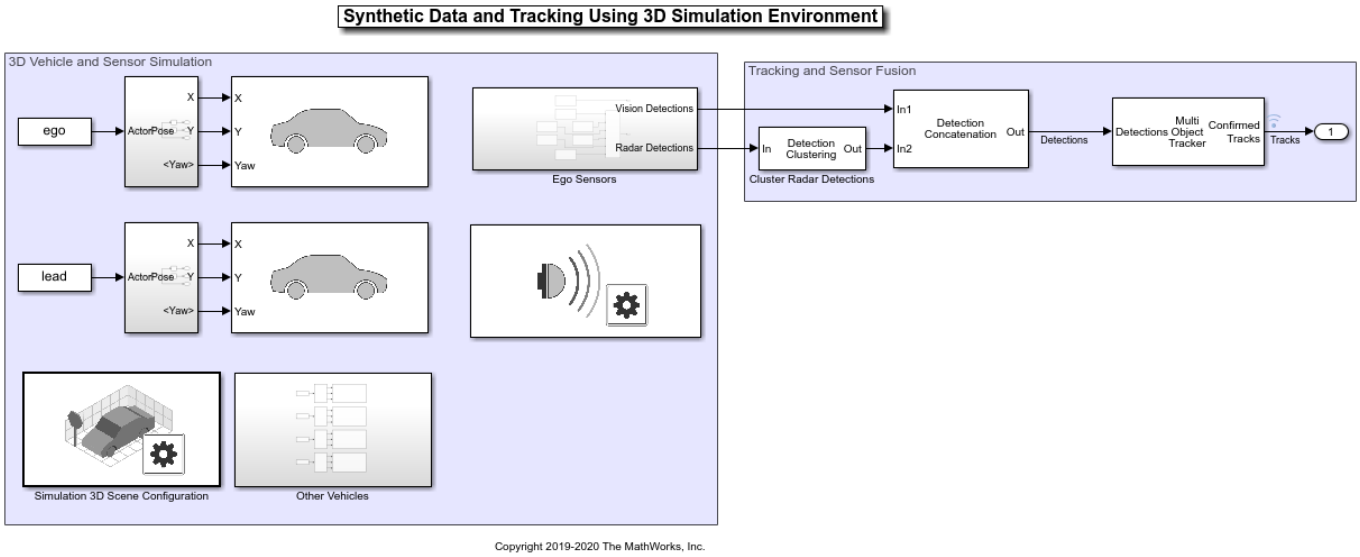
Setup and Overview of Model

Prior to running this example, the roads, actors, and trajectories in the scenario were created using this procedure:

1. Extract the center locations from a portion of the road in the “Define Road Layouts Programmatically” on page 7-453 3D scene, using the techniques introduced in “Select Waypoints for Unreal Engine Simulation” on page 7-626.
2. Create a road in the Driving Scenario Designer that has these extracted locations as its road center values.
3. Define multiple moving vehicles on the road that have trajectories similar to the ones in the scenario defined in “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” on page 7-205.
4. Export the trajectories from the app and load them into the MATLAB® workspace by using the `helperCreateVehicleTrajectories` script.
5. Read these trajectories into the Simulink model using From Workspace blocks.

The actor poses provided by the From Workspace blocks are used by the Simulation 3D Vehicle with Ground Following blocks to define the locations of the ego vehicle, lead vehicle, and other vehicles at each time step of the simulation.

```
close;
if ~ispc
    error(['3D Simulation is supported only on Microsoft', char(174), ' Windows', char(174), '.'])
end

open_system('SimulateSensorsIn3DEnvironmentModel');
```

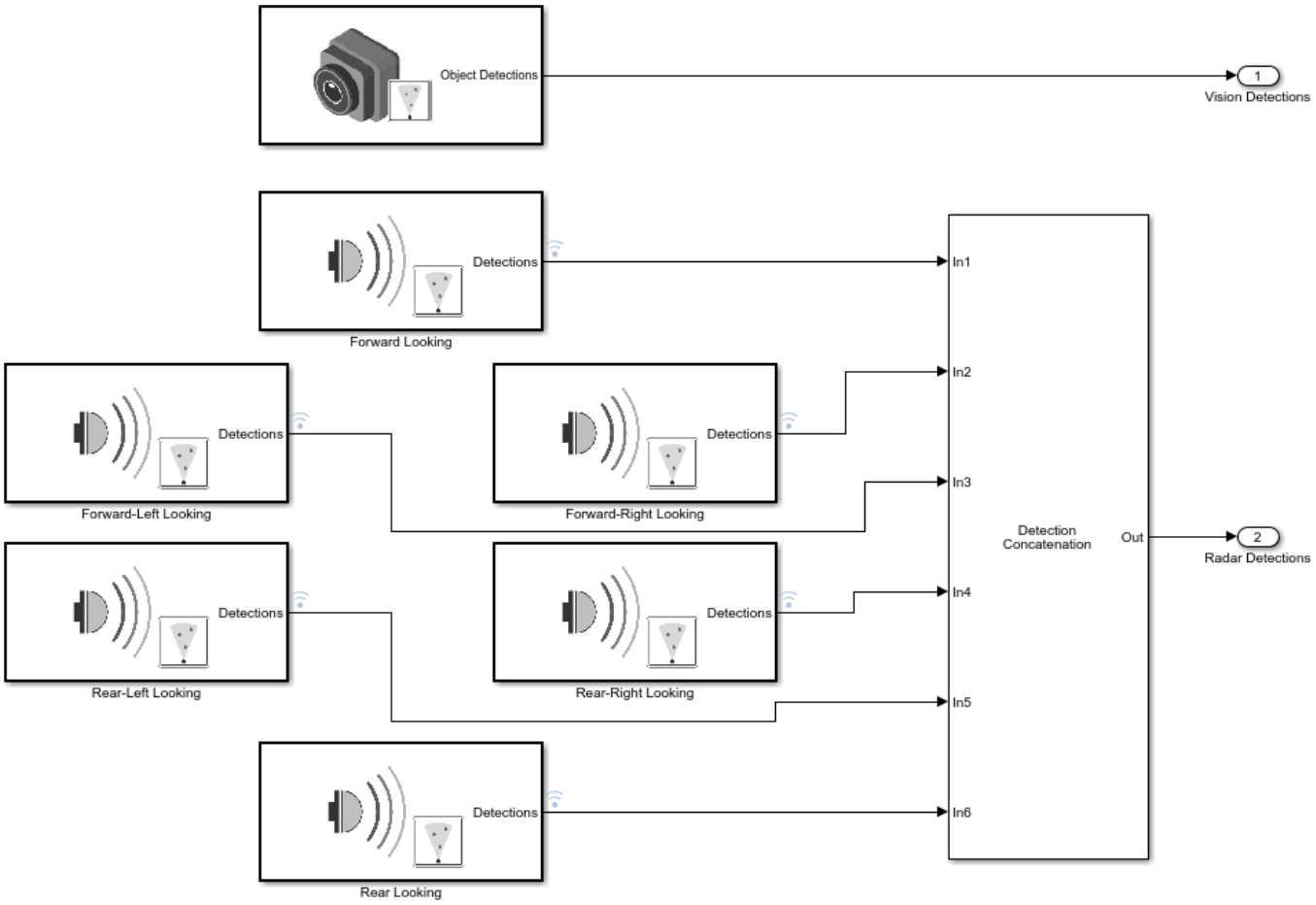


Simulating Sensor Detections

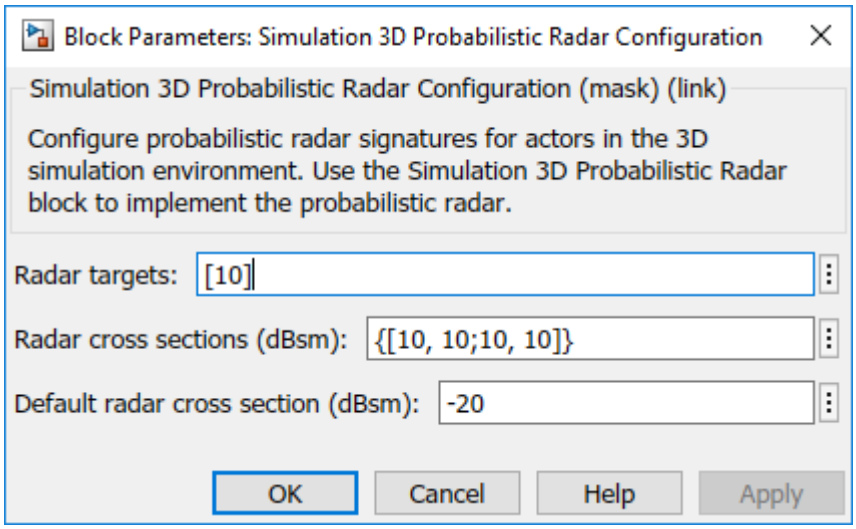
In this example, you simulate an ego vehicle that a vision sensor on its front bumper and six radar sensors covering the full 360 degrees field of view. The ego vehicle is equipped with a long-range radar on both the front and rear of the vehicle. Each side of the vehicle has two short-range radars, each covering 90 degrees. One radar on each side covers from the middle of the vehicle to the back. The other radar on each side covers from the middle of the vehicle forward.

The Ego Sensors subsystem contains the one Simulation 3D Vision Detection Generator block and six Simulation 3D Probabilistic Radar blocks that model the previously described sensors. The outputs of the radar blocks are concatenated using a Detection Concatenation block. In the top-level model, the radar output is then cocatenated with the vision output to create a single stream of detections to be fused by the Multi-Object Tracker block.

```
open_system('SimulateSensorsIn3DEnvironmentModel/Ego Sensors')
```



The probabilistic radars "see" not only an actor's physical dimensions (e.g., length, width, and height) but are also sensitive to an actor's *electrical* size. An actor's electrical size is referred to as its radar cross-section (RCS). The RCS patterns for the vehicles in the simulation are defined using the Simulation 3D Probabilistic Radar Configuration block.



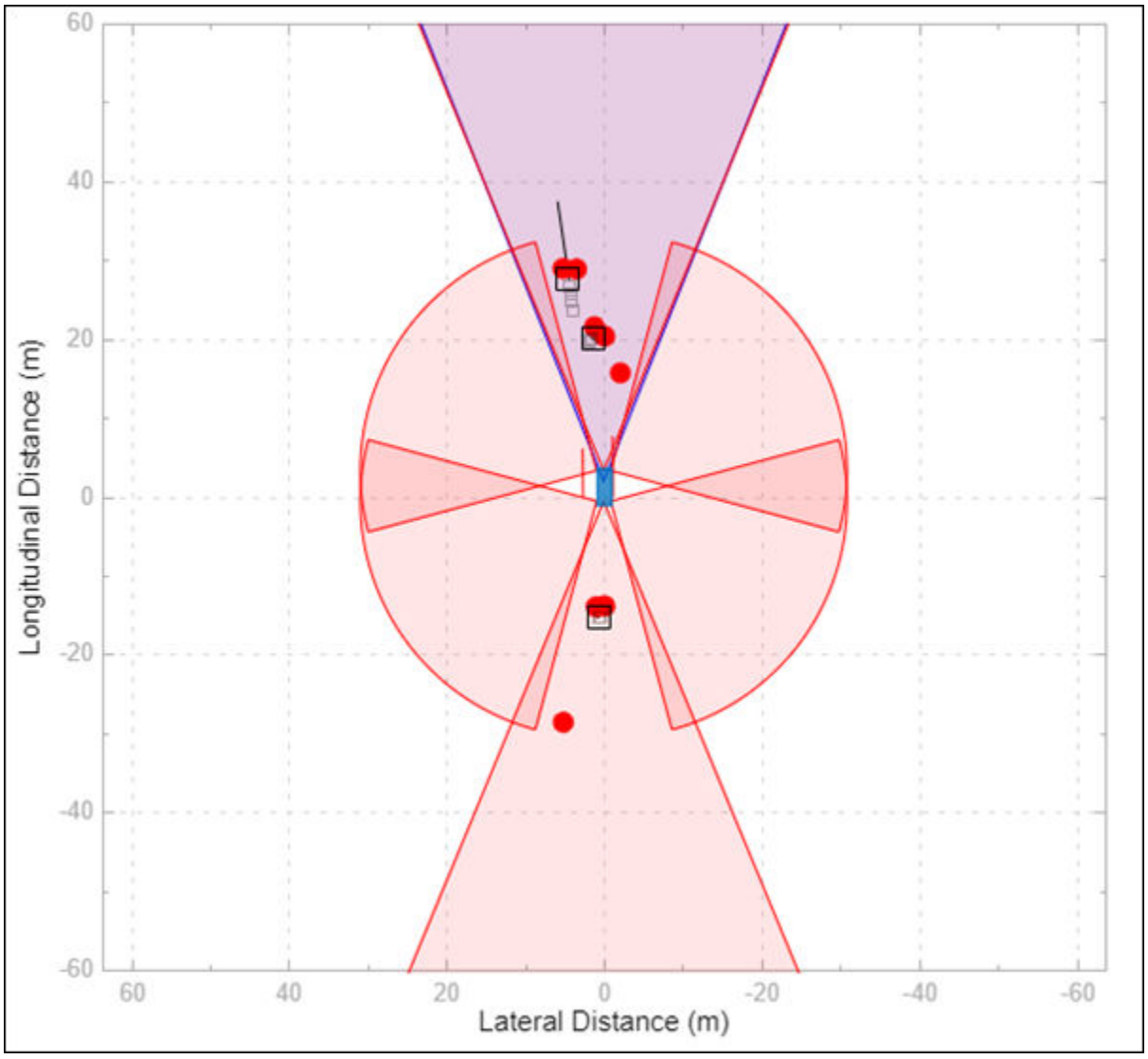
Use this block to define the RCS patterns for all of the actors in the simulation. Any actors that do not have a specified RCS pattern use the default RCS value.

Tracking and Sensor Fusion

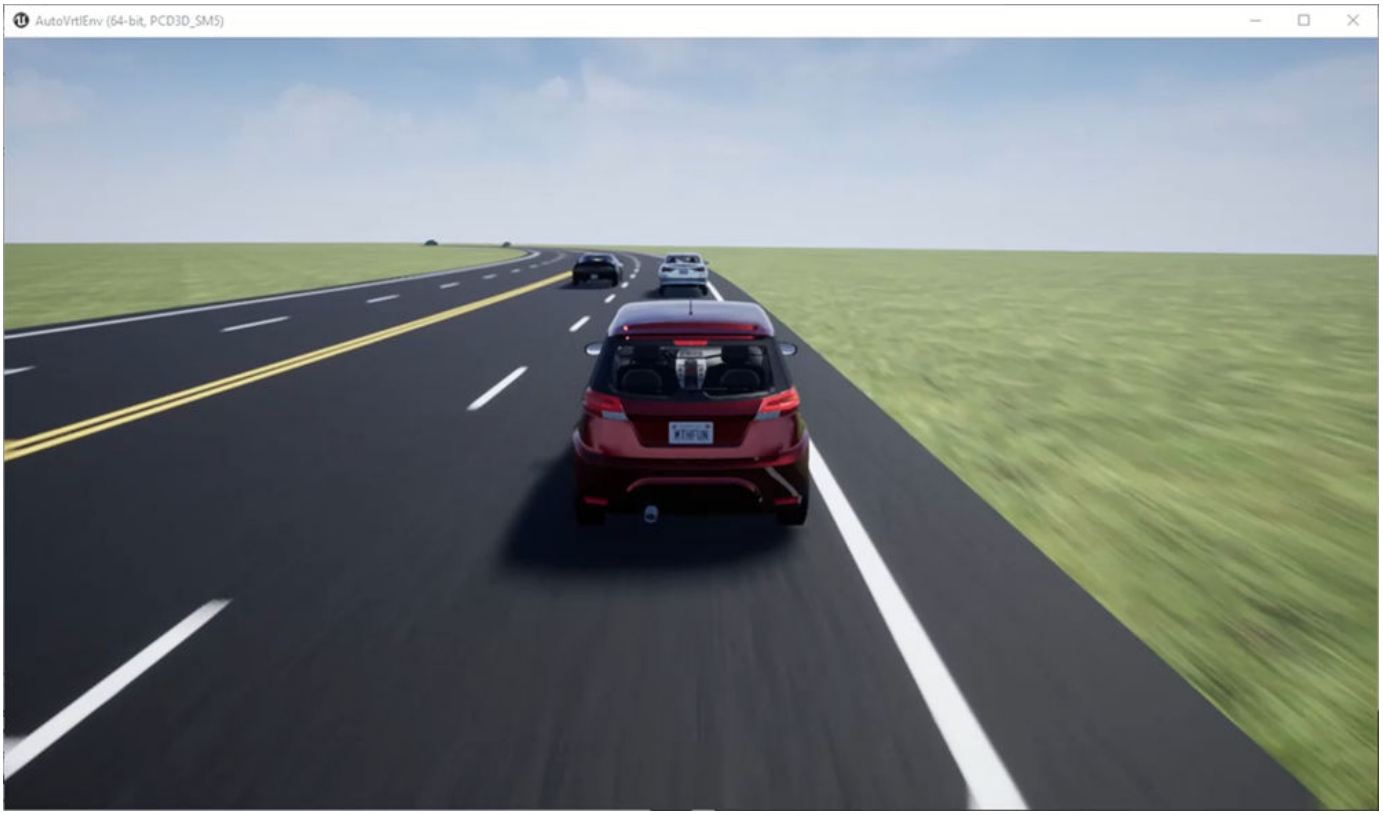
The detections generated by the ego vehicle's suite of radars are preprocessed using a helper Detection Clustering block before they are fused using the Multi-Object Tracker block. The multi-object tracker is configured with the same parameters used in the corresponding Simulink example, “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” on page 7-205. The output from the Multi-Object Tracker block is a list of confirmed tracks.

Display

The Bird's-Eye Scope is a model-level visualization tool in Simulink opened from the Simulink toolstrip. After opening the scope, click **Find Signals** to set up the signals. Then run the simulation to display the ego actor, radar and vision detections, and tracks. The following image shows the scope's display for this example.



When the simulation starts, a few seconds are needed to initialize the Unreal Engine simulation environment, especially when running it for the first time. Once this initialization is complete, the simulation environment opens in a separate window. The following image is a snapshot of the simulation window corresponding to the snapshot of the Bird's-Eye Scope shown in the previous image.



The simulated vehicles are shown in the simulation window. The detections and tracks generated by the simulation appear only in the Bird's-Eye Scope.

Summary

In this example, you learned how to extract road centers from a 3D scenario for use in the Driving Scenario Designer app. You also learned how to export the vehicle trajectories created from the road segments for use in the 3D simulation environment in Simulink. You then learned how to configure a probabilistic camera model and multiple probabilistic radar models in the Unreal Engine environment and how to fuse the detections from the multiple sensors located around the ego vehicle's perimeter using a multi-object tracker. The confirmed tracks generated by the tracker can then be used for control algorithms such as adaptive cruise control (ACC) or forward collision warning (FCW).

```
close_system('SimulateSensorsIn3DEnvironmentModel');
```

See Also

Apps
Bird's-Eye Scope | Driving Scenario Designer

Blocks
Detection Concatenation | Multi-Object Tracker | Simulation 3D Probabilistic Radar | Simulation 3D Probabilistic Radar Configuration | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following

More About

- “Visualize Sensor Data from Unreal Engine Simulation Environment” on page 6-36
- “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” on page 7-205

Highway Lane Following

This example shows how to simulate a highway lane following application with controller, sensor fusion, and vision processing components. These components are tested in a 3D simulation environment that includes camera and radar sensor models.

Introduction

A highway lane-following system steers a vehicle to travel within a marked lane. It also maintains a set velocity or safe distance to a preceding vehicle in the same lane. The system typically uses vision processing algorithms to detect lanes and vehicles from a camera. The vehicle detections from the camera are then fused with detections from a radar to improve the ability to detect surrounding vehicles. The controller uses the lane detections, vehicle detections, and set speed to control steering and acceleration.

This example demonstrates how to create a test bench model to test vision processing, sensor fusion, and controls in a 3D simulation environment. The test bench model can be configured for different scenarios to test the ability to follow lanes and avoid collisions with other vehicles. In this example, you:

1. **Explore the test bench model**: The model contains vision processing, sensor fusion, controls, vehicle dynamics, sensors, and metrics to assess functionality. A cuboid scenario defines vehicle trajectories, specifies ground truth, and models detections from a vision sensor. An equivalent Unreal Engine® scene is used to model detections from a radar sensor and images from a monocular camera sensor.
2. **Visualize a test scenario**: The scenario contains a curved road with multiple vehicles.
3. **Simulate with a probabilistic detection sensor**: The model is configured to test integration of the sensor fusion and controls components. The detection outputs from the vision and radar sensors are inputs to the sensor fusion algorithm.
4. **Simulate with a vision processing algorithm**: The model is configured to test integration of the vision processing, sensor fusion, and controls components. The output of the monocular camera sensor is the input to the vision processing algorithm. The detection output from the vision processing algorithm and the detection output of the radar sensor are inputs to the sensor fusion algorithm.
5. **Explore additional scenarios**: These scenarios test the system under additional conditions.

Testing the integration of the controller and the perception algorithm requires a photorealistic simulation environment. In this example, you enable system-level simulation through integration with the Unreal Engine from Epic Games®. The 3D simulation environment requires a Windows® 64-bit platform.

```
if ~ispc
    error(['3D Simulation is only supported on Microsoft', char(174), ' Windows', char(174), '.'])
end
```

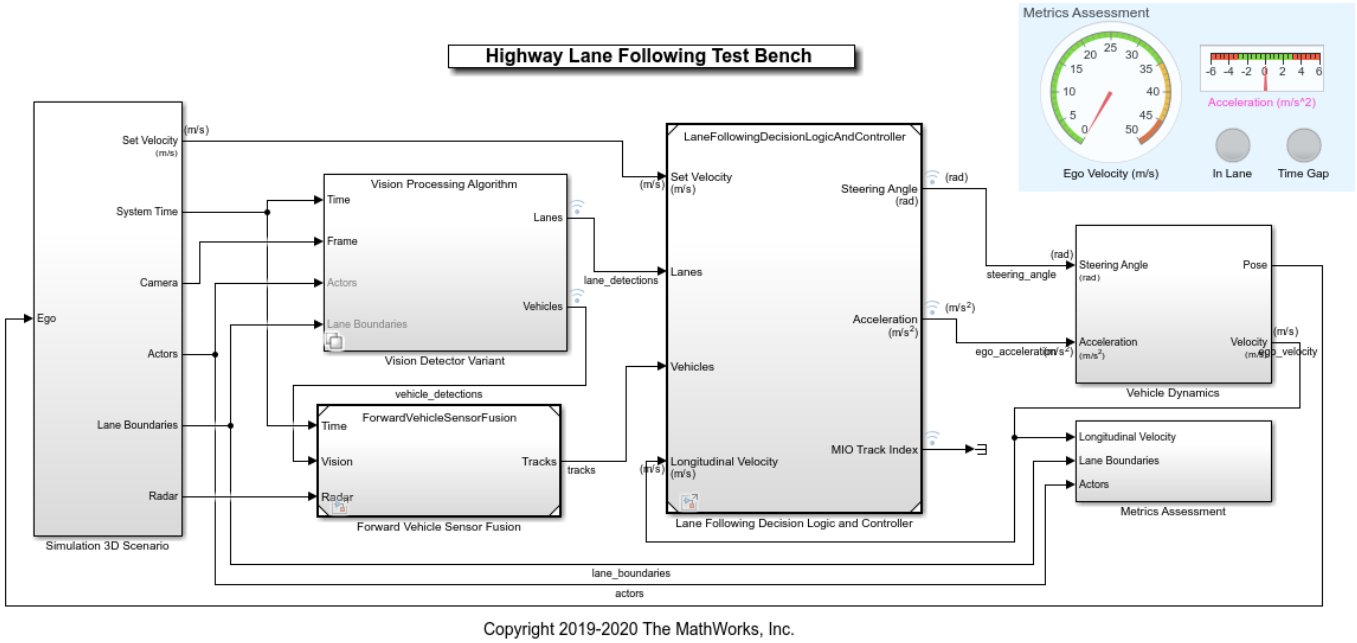
To ensure reproducibility of the simulation results, set the random seed.

```
rng(0)
```

Explore Test Bench Model

In this example, you use a system-level simulation test bench model to explore the behavior of the control and vision processing algorithms for the lane following system. Open the system-level simulation test bench model.

```
open_system("HighwayLaneFollowingTestBench")
```



Copyright 2019-2020 The MathWorks, Inc.

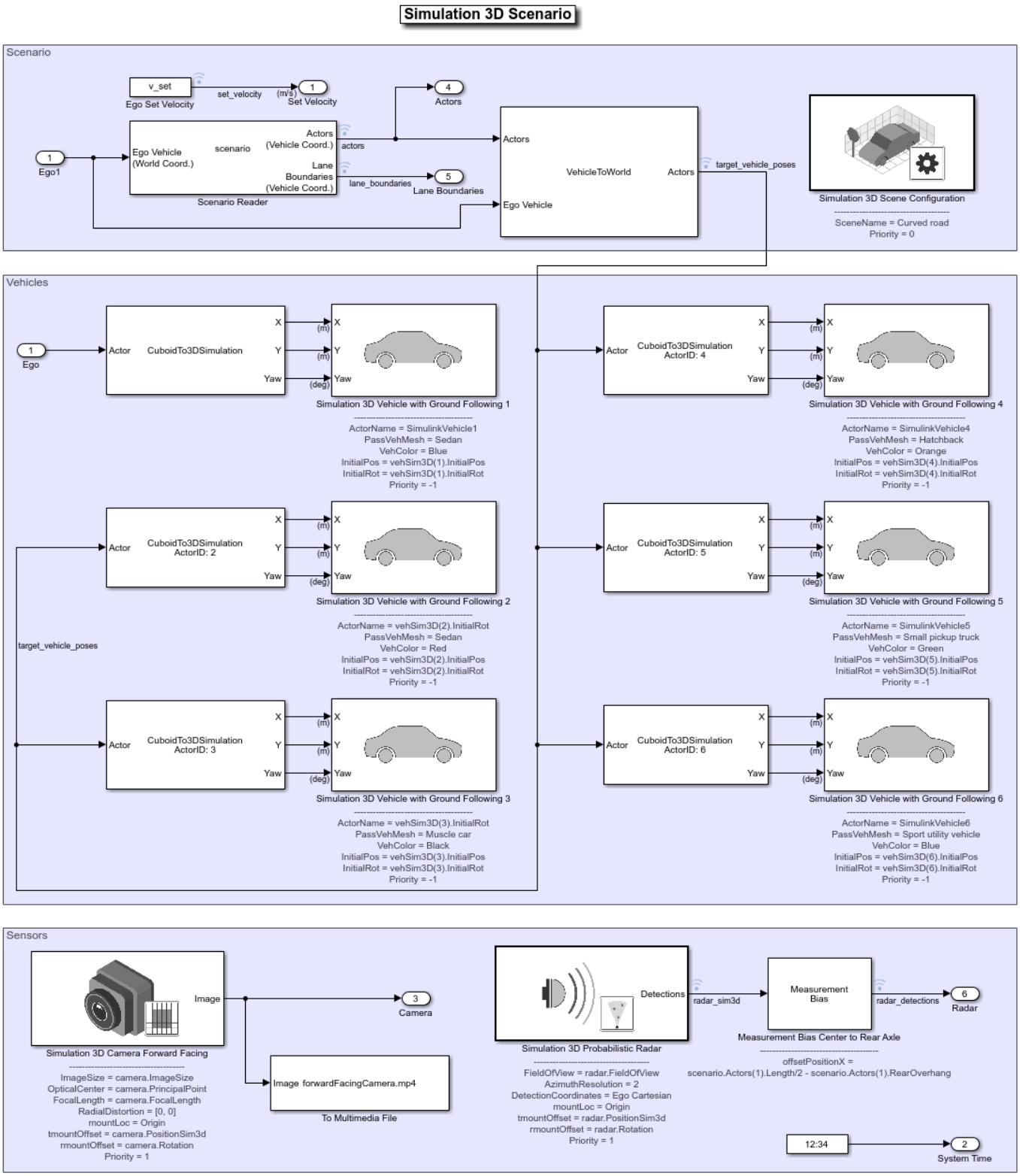
The test bench model contains these subsystems:

1. Simulation 3D Scenario: Specifies road, vehicles, camera and radar sensors used for simulation
2. Vision Detector Variant: Specifies the fidelity of the two different vision detection algorithms to choose from
3. Forward Vehicle Sensor Fusion: Fuses the detections of vehicles in front of the ego vehicle that were obtained from vision and radar sensors
4. Lane Following Decision and Controller: Specifies lateral and longitudinal decision logic and the lane following controller
5. Vehicle Dynamic: Specifies the dynamics model for the ego vehicle
6. Metrics Assessment: Assesses system-level behavior

The Forward Vehicle Sensor Fusion, Lane Following Decision and Controller, Vehicle Dynamics, and Metrics Assessment subsystems are based on the subsystems used in the “Lane Following Control with Sensor Fusion and Lane Detection” on page 7-280. This example focuses on the Simulation 3D Scenario and Vision Detector Variant subsystems.

The Simulation 3D Scenario subsystem configures the road network, sets vehicle positions, and synthesizes sensors. Open the Simulation 3D Scenario subsystem.

```
open_system("HighwayLaneFollowingTestBench/Simulation 3D Scenario")
```



The scene and road network are specified by these parts of the subsystem:

- The Simulation 3D Scene Configuration block has the **SceneName** parameter set to `Curved road`.
- The Scenario Reader block is configured to use a driving scenario that contains a road network that closely matches a section of the road network.

The vehicle positions are specified by these parts of the subsystem:

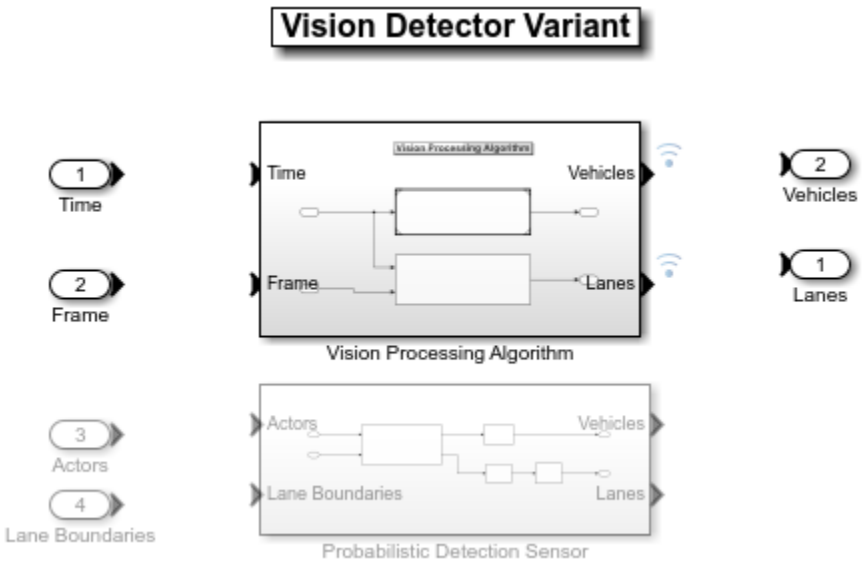
- The Ego input port controls the position of the ego vehicle, which is specified by the Simulation 3D Vehicle with Ground Following 1 block.
- The Vehicle To World block converts actor poses from the coordinates of the input ego vehicle to the world coordinates.
- The Scenario Reader block outputs actor poses, which control the position of the target vehicles. These vehicles are specified by the other Simulation 3D Vehicle with Ground Following blocks.
- The Cuboid To 3D Simulation block converts the ego pose coordinate system (with respect to below the center of the vehicle rear axle) to the 3D simulation coordinate system (with respect to below the vehicle center).

The sensors attached to the ego vehicle are specified by these parts of the subsystem:

- The Simulation 3D Camera block is attached to the ego vehicle to capture its front view. The output Image from this block is processed by vision processing algorithm to detect lanes and vehicles.
- The Simulation 3D Probabilistic Radar Configuration block is attached to the ego vehicle to detect vehicles in 3D Simulation environment.
- The Measurement Bias Center to Rear Axle block converts the coordinate system of the Simulation 3D Probabilistic Radar Configuration block (with respect to below the vehicle center) to the pose coordinates (with respect to below the center of the vehicle rear axle).

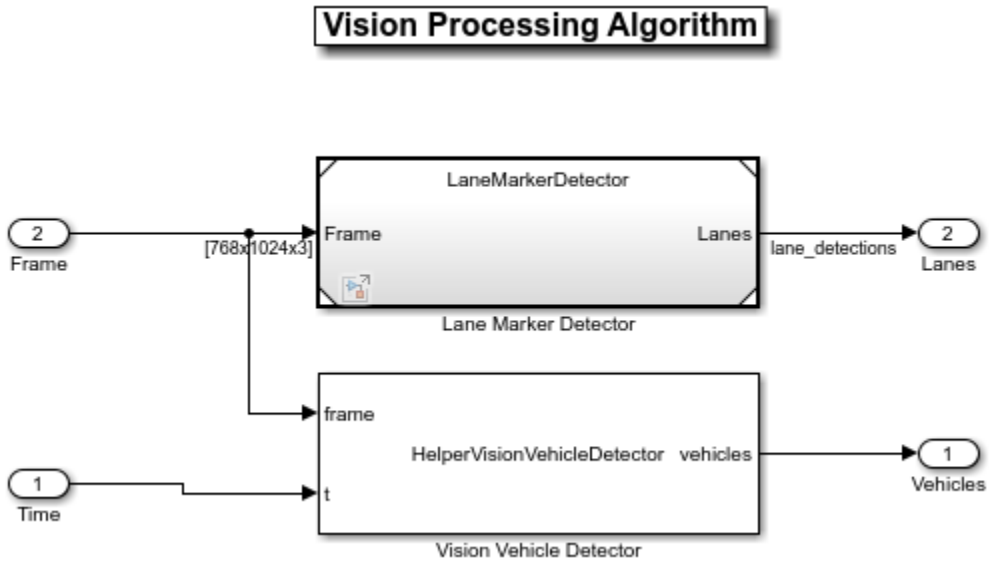
The Vision Detector Variant subsystem allows you to select the fidelity of the vision detection algorithm based on the types of tests you want to run. Open the Vision Detector Variant subsystem.

```
open_system("HighwayLaneFollowingTestBench/Vision Detector Variant")
```



- The Probabilistic Detection Sensor variant enables you to test integration of the control algorithm in the 3D simulation environment, without also integrating the vision processing algorithm. This variant uses a Vision Detection Generator block to synthesize vehicle and lane detections based on actor ground truth positions. This configuration helps you to verify interactions with vehicles and the radar sensor in the 3D simulation environment without vision processing algorithm.
- The Vision Processing Algorithm variant enables you to test integration of the control algorithm and vision processing algorithm in the 3D simulation environment. Open the Vision Processing Algorithm variant.

```
open_system("HighwayLaneFollowingTestBench/Vision Detector Variant/Vision Processing Algorithm")
```



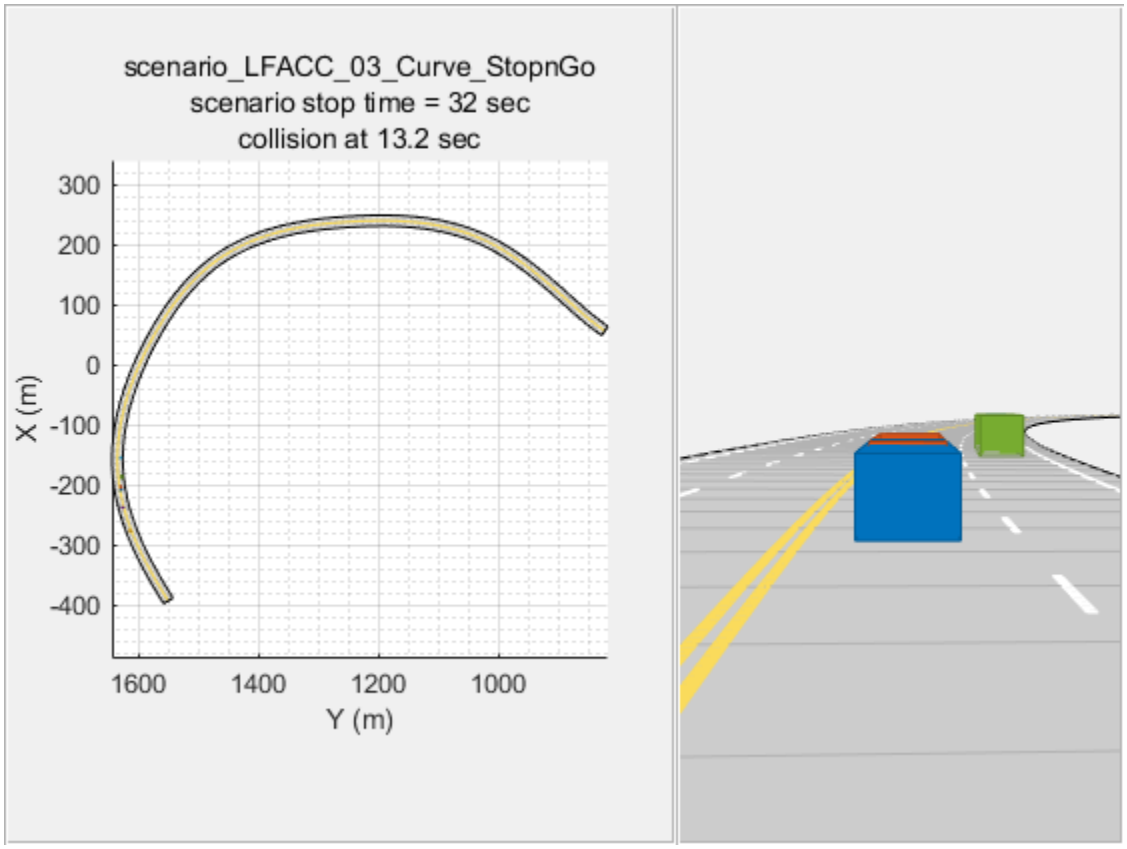
This variant uses a MATLAB based lane boundary detection algorithm and vehicle detection algorithm based on the “Visual Perception Using Monocular Camera” on page 7-78 example. The primary difference from that example is that in this example, lane boundary detection and vehicle detection algorithms are segregated into separate components. Lane Marker Detector is a reference model. If you have Simulink Coder™ license, then you can generate C++ code from this reference model. It uses a System object™, `HelperLaneDetectorWrapper`, to detect the lane markers. It also contains a lane tracker to improve performance of lane detection in crowded conditions. Vision Vehicle Detector uses the `HelperVisionVehicleDetector` System object to detect the vehicles. These System objects pack the output data to buses, as required for further processing. Since the vision processing algorithm operates on an image returned by the camera sensor, the Vision Processing Algorithm takes longer to execute than the Probabilistic Detection Sensor variant.

Visualize a Test Scenario

The helper function `scenario_LFACC_03_Curve_StopnGo` generates a driving scenario that is compatible with the `HighwayLaneFollowingTestBench` model. This is an open-loop scenario on a curved road and includes multiple target vehicles. The road centers and lane markings closely match a section of the curved road scene provided with the 3D simulation environment. The scenario has the same number of vehicles as the model and they have the same dimensions. In this scenario, a lead vehicle slows down in front of the ego vehicle while other vehicles travel in adjacent lanes.

Plot the open-loop scenario to see the interactions of the ego vehicle and target vehicles.

```
hFigScenario = helperPlotLFScenario("scenario_LFACC_03_Curve_StopnGo");
```



The ego vehicle is not under closed-loop control, so a collision occurs with the slower moving lead vehicle. The goal of the closed-loop system is to follow the lane and maintain a safe distance from the lead vehicles. In the `HighwayLaneFollowingTestBench` model, the ego vehicle has the same initial velocity and initial position as in the open-loop scenario.

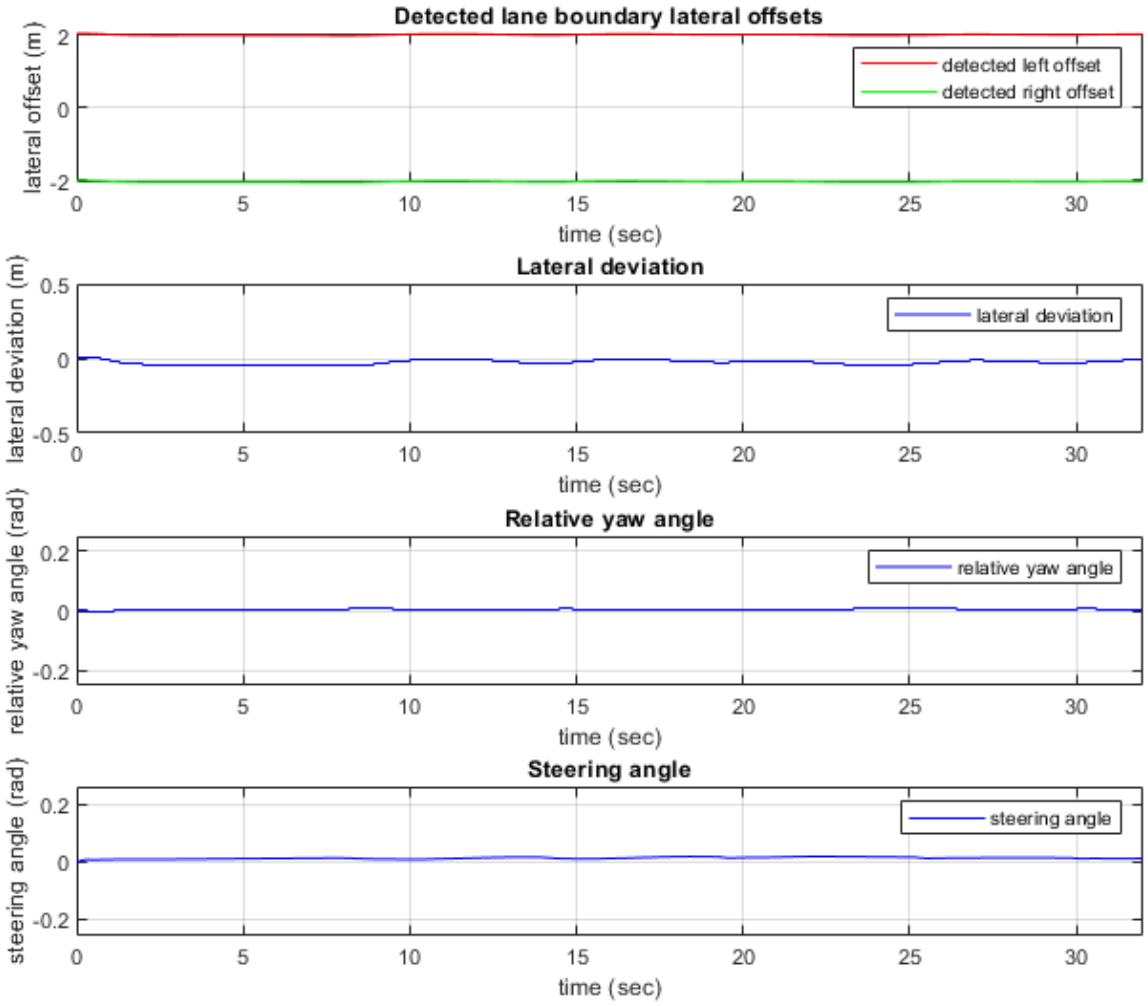
Simulate with Probabilistic Vision Detection Sensor

To verify that interactions with vehicles and the radar sensor are working properly, test the interactions between the control algorithm and the 3D simulation environment using the probabilistic vision detection sensor. Doing so enables you to verify baseline system behavior without integrating the full vision processing algorithm. Configure the test bench model and run the simulation. To reduce command-window output, turn off the MPC update messages.

```
helperSLHighwayLaneFollowingSetup(...
    "scenario_LFACC_03_Curve_StopnGo",...
    "ProbabilisticDetectionSensor");
mpcverbosity('off');
sim("HighwayLaneFollowingTestBench")
```

Plot the lateral controller performance results.

```
hFigLatResults = helperPlotLFLateralResults(logsout);
```

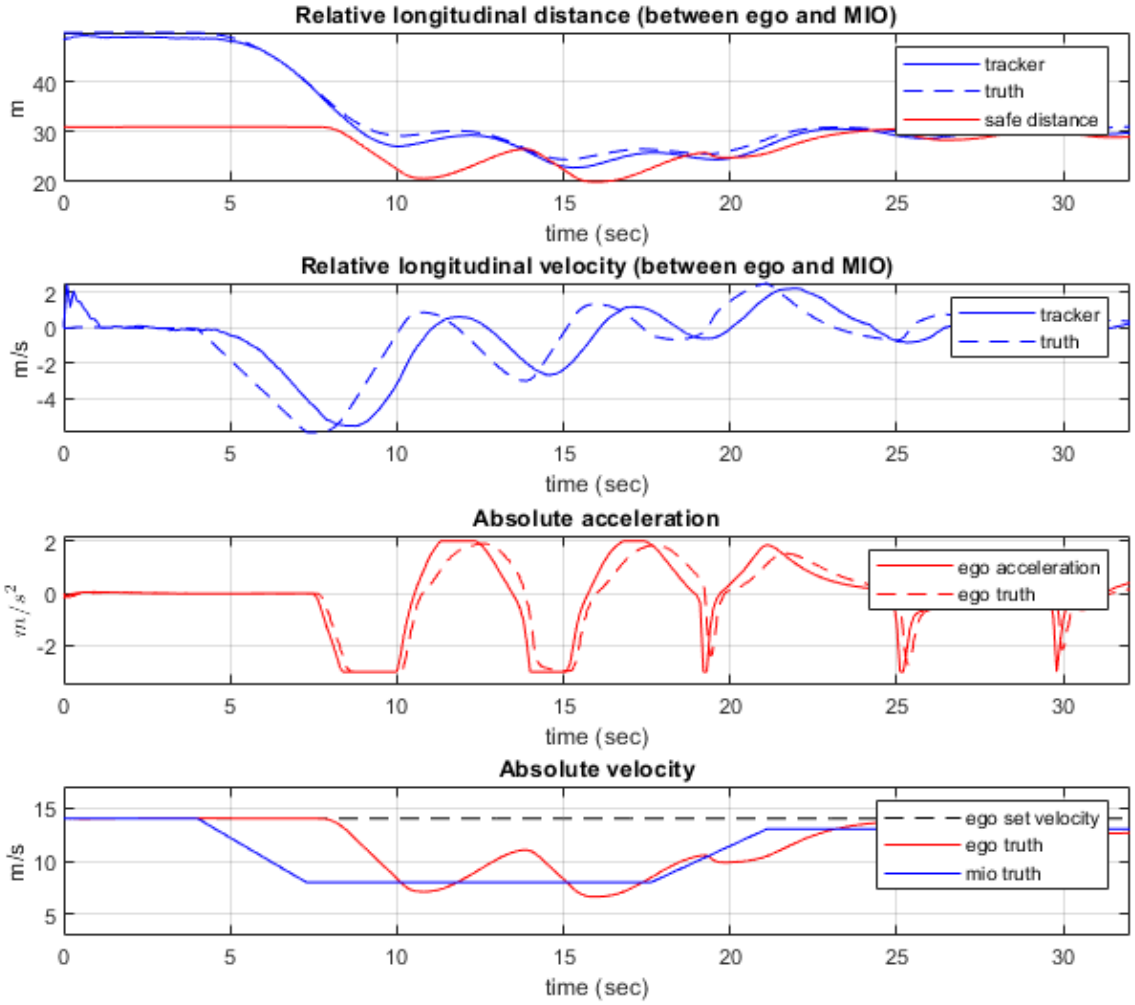


Examine the simulation results.

- The **Detected lane boundary lateral offsets** plot shows the lateral offsets for the detected left-lane and right-lane boundaries. The detected values are close to the ground truth of the lane.
- The **Lateral deviation** plot shows the lateral deviation of the ego vehicle from the centerline of the lane. The lateral deviation is close to 0, which implies that the ego vehicle closely follows the centerline. Small deviations occur when the vehicle is changing velocity to avoid collision with another vehicle.
- The **Relative yaw angle** plot shows the relative yaw angle between ego vehicle and the centerline of the lane. The relative yaw angle is very close to 0, which implies that the heading angle of the ego vehicle matches the yaw angle of the centerline closely.
- The **Steering angle** plot shows the steering angle of the ego vehicle. The steering angle trajectory is smooth.

Plot the longitudinal controller performance results.

```
hFigLongResults = helperPlotLFLongitudinalResults(logsout,time_gap,...
    default_spacing);
```



Examine the simulation results.

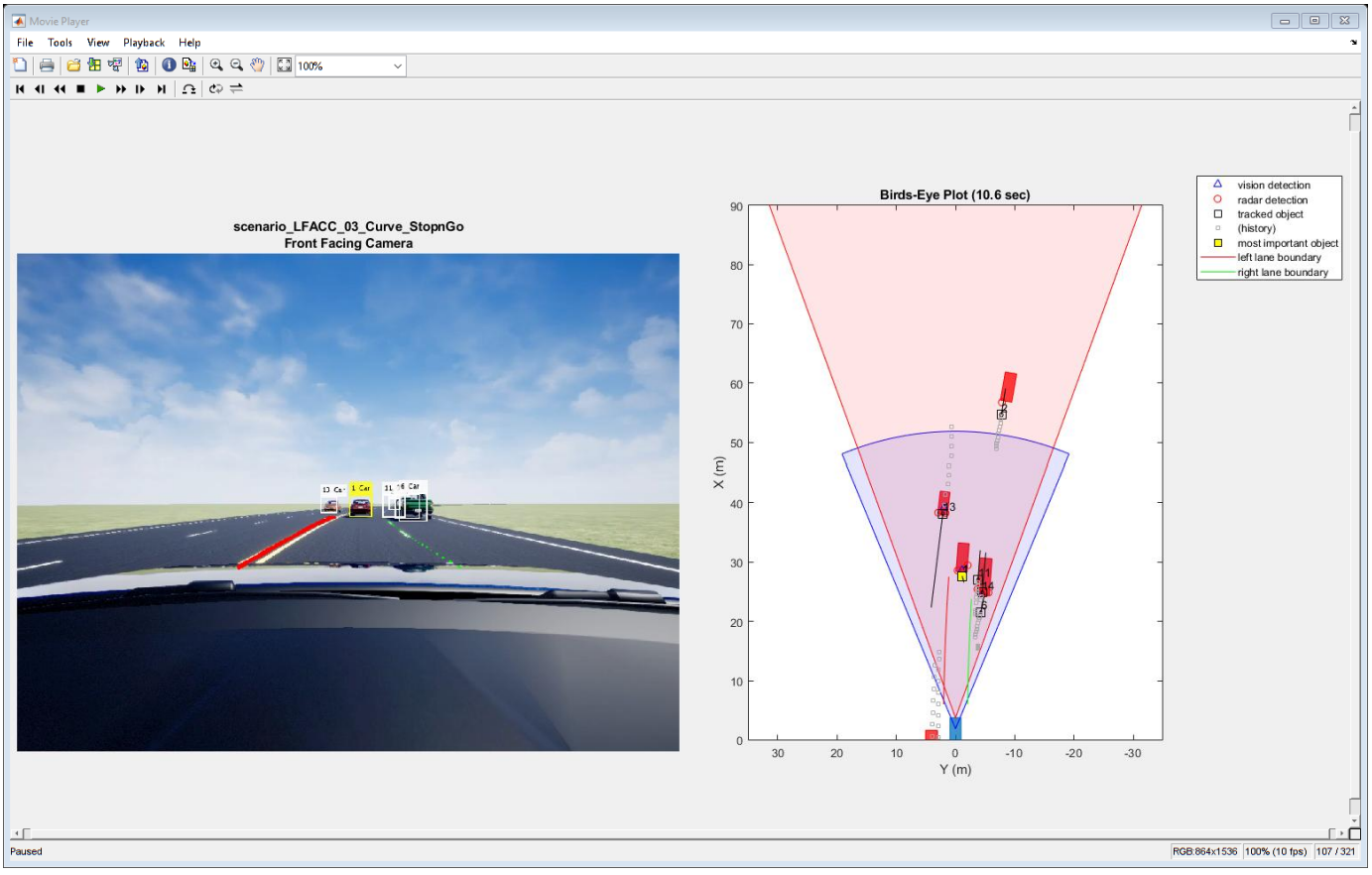
- The **Relative longitudinal distance** plot shows the distance between the ego vehicle and the Most Important Object (MIO). The MIO represents the closest vehicle ahead of and in the same lane as the ego vehicle. In this case, the ego vehicle approaches the MIO and gets close to it or exceeds the safe distance in some cases.
- The **Relative longitudinal velocity** plot shows the relative velocity between the ego vehicle and the MIO. In this example, the vision processing algorithm only detects positions, so the tracker in the control algorithm estimates the velocity. The estimated velocity lags the actual (ground truth) MIO relative velocity.
- The **Absolute acceleration** plot shows that the controller commands the vehicle to decelerate when it gets too close to the MIO.

- The **Absolute velocity** plot shows the ego vehicle initially follows the set velocity, but when the MIO slows down, to avoid a collision, the ego vehicle also slows down.

During simulation, the model logs signals to the base workspace as `logsout` and records the output of the camera sensor to `forwardFacingCamera.mp4`. You can use the `plotLFDetectionResults` function to visualize the simulated detections similar to how recorded data is explored in the "Forward Collision Warning Using Sensor Fusion" on page 7-125 example. You can also record the visualized detections to a video file to enable review by others who do not have access to MATLAB.

Plot the detection results from logged data, generate a video, and open the video in the Video Viewer (Image Processing Toolbox) app.

```
hVideoViewer = helperPlotLFDetectionResults(...
    logsout, "forwardFacingCamera.mp4" , scenario, camera, radar,...
    scenarioFcnName,...
    "RecordVideo", true,...
    "RecordVideoFileName", scenarioFcnName + "_PDS",...
    "OpenRecordedVideoInVideoViewer", true,...
    "VideoViewerJumpToTime", 10.6);
```



Play the generated video.

- **Front Facing Camera** shows the image returned by the camera sensor. The left lane boundary is plotted in red and the right lane boundary is plotted in green. These lanes are returned by the probabilistic detection sensor. Tracked detections are also overlaid on the video.

- **Birds-Eye Plot** shows true vehicle positions, sensor coverage areas, probabilistic detections, and track outputs. The plot title includes the simulation time so that you can correlate events between the video and previous static plots.

Close the figures.

```
close(hFigScenario)
close(hFigLatResults)
close(hFigLongResults)
close(hVideoViewer)
```

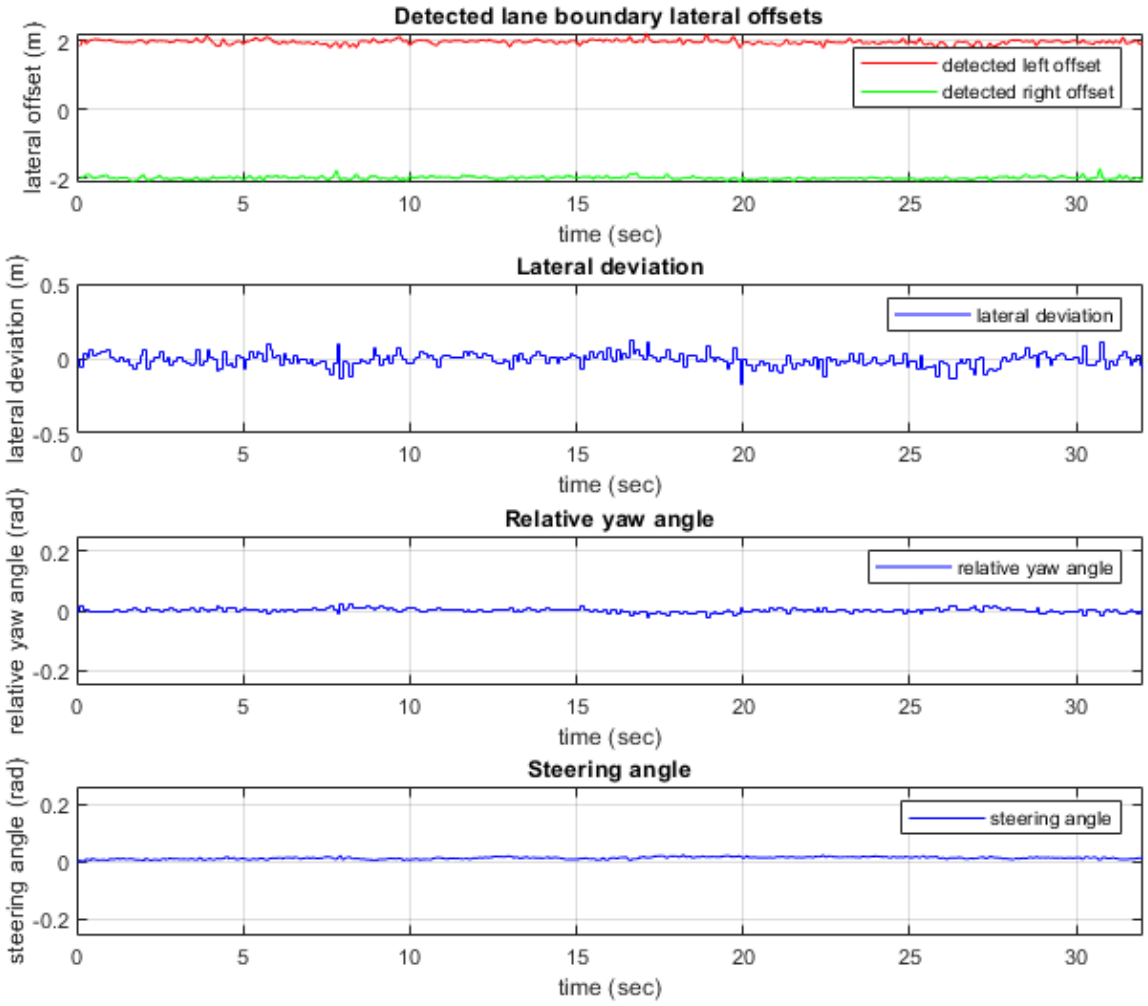
Simulate with Vision Processing Algorithm

Now that you verified the control algorithm, test the control algorithm and vision processing algorithm together in the 3D simulation environment. This enables you to explore the effect of the vision processing algorithm on system performance. Configure the test bench model to use the same scenario with the vision processing variant.

```
helperSLHighwayLaneFollowingSetup(...
    "scenario_LFACC_03_Curve_StopnGo",...
    "VisionProcessingAlgorithm");
sim("HighwayLaneFollowingTestBench")
```

Plot the lateral controller performance results.

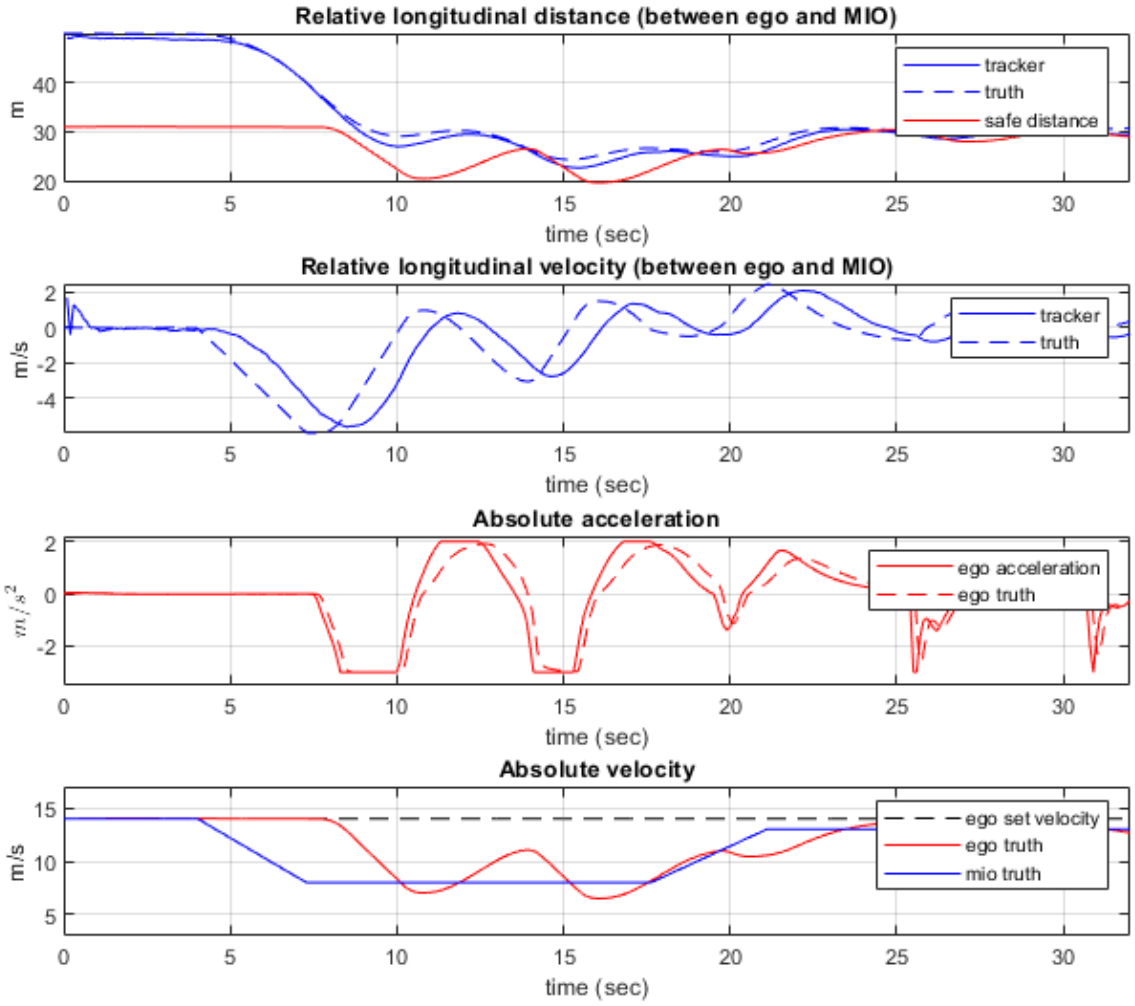
```
hFigLatResults = helperPlotLFLateralResults(logsout);
```



The vision processing algorithm detects the left and right lane boundaries but the detections are noisier, which affects the lateral deviation. The lateral deviation is still small but is larger than the run with the probabilistic detection sensor variant.

Plot the longitudinal controller performance results.

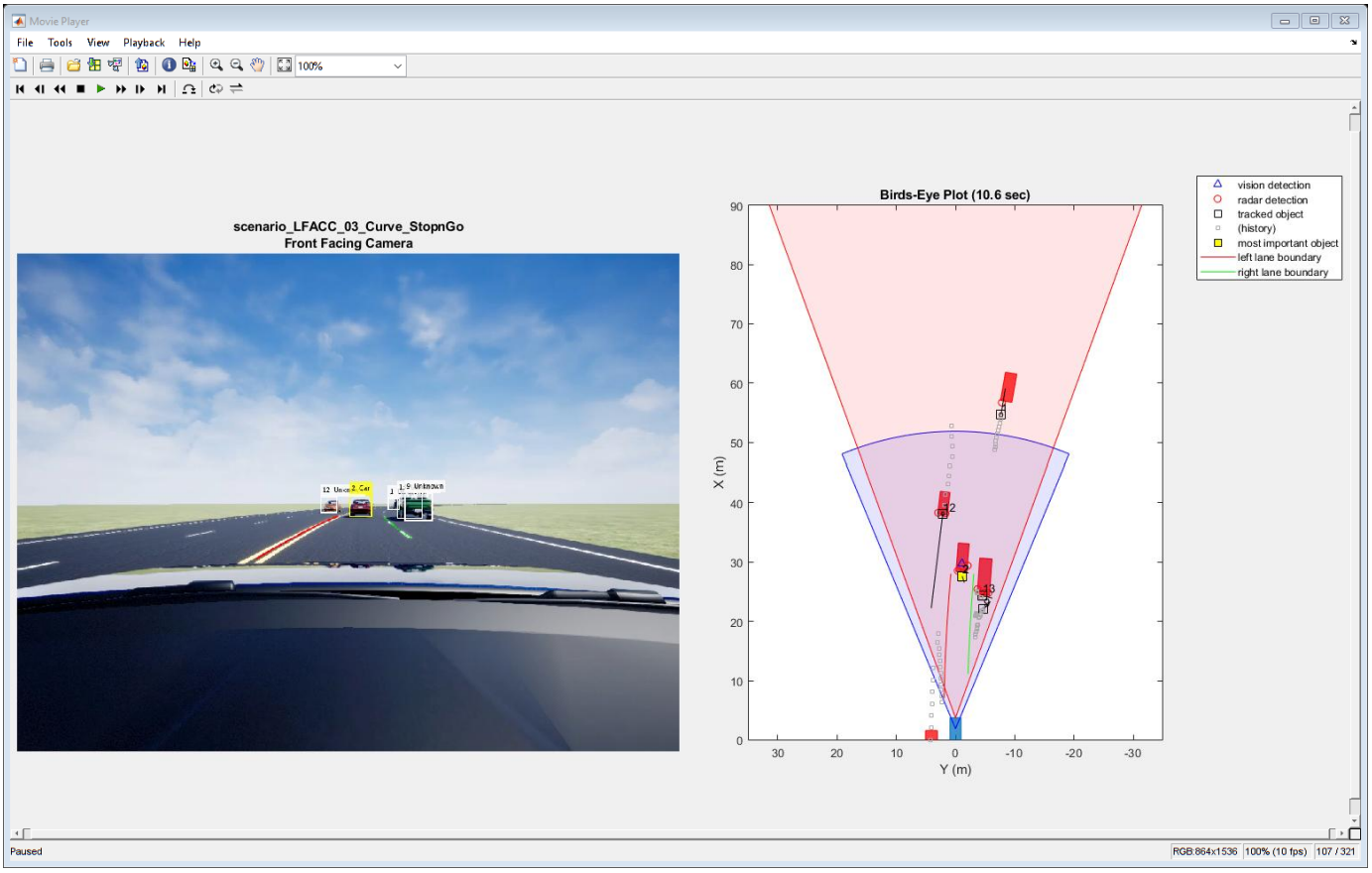
```
hFigLongResults = helperPlotLFLongitudinalResults(logsout,time_gap,...
    default_spacing);
```



The relative distance and relative velocity have some discontinuities. These discontinuities are due to imperfections of the vision processing algorithm on system performance. Even with these discontinuities, the resulting ego acceleration and velocity are similar to the results using the probabilistic detection sensor variant.

Plot the detection results from logged data, generate a video, and open the Video Viewer (Image Processing Toolbox) app.

```
hVideoViewer = helperPlotLFDetectionResults(...
    logsout, "forwardFacingCamera.mp4" , scenario, camera, radar,...
    scenarioFcnName,...
    "RecordVideo", true,...
    "RecordVideoFileName", scenarioFcnName + "_VPA",...
    "OpenRecordedVideoInVideoViewer", true,...
    "VideoViewerJumpToTime", 10.6);
```



Close the figures.

```
close(hFigLatResults)
close(hFigLongResults)
close(hVideoViewer)
```

Explore Additional Scenarios

The previous simulations tested the `scenario_LFACC_03_Curve_StopnGo` scenario using both the probabilistic vision detection sensor and vision processing algorithm variants. This example provides additional scenarios that are compatible with the `HighwayLaneFollowingTestBench` model:

```
scenario_LF_01_Straight_RightLane
scenario_LF_02_Straight_LeftLane
scenario_LF_03_Curve_LeftLane
scenario_LF_04_Curve_RightLane
scenario_LFACC_01_Curve_DecelTarget
scenario_LFACC_02_Curve_AutoRetarget
scenario_LFACC_03_Curve_StopnGo
scenario_LFACC_04_Curve_CutInOut
scenario_LFACC_05_Curve_CutInOut_TooClose
scenario_LFACC_06_Straight_StopandGoLeadCar
```

These scenarios represent two types of testing.

- Use scenarios with the `scenario_LF_` prefix to test lane-detection and lane-following algorithms without obstruction by other vehicles. The vehicles still exist in the scenario, but are positioned such that they are not seen by the ego vehicle on the road.
- Use scenarios with the `scenario_LFACC_` prefix to test lane-detection and lane-following algorithms with other vehicles on the road.

Examine the comments in each file for more details on the road and vehicles in each scenario. You can configure the `HighwayLaneFollowingTestBench` model and workspace to simulate these scenarios using the `helperSLHighwayLaneFollowingSetup` function.

For example, while learning about the effects of a camera-based lane detection algorithm on closed-loop control, it can be helpful to begin with a scenario that has a road but no vehicles. To configure the model and workspace for such a scenario, use the following code.

```
helperSLHighwayLaneFollowingSetup(...
    "scenario_LF_04_Curve_RightLane",...
    "VisionProcessingAlgorithm");
```

Enable the MPC update messages again.

```
mpcverbosity('on');
```

Conclusion

This example shows how to simulate a highway lane following application with controller, sensor fusion, and vision processing components.

See Also

Blocks
Cuboid To 3D Simulation | Scenario Reader | Simulation 3D Camera | Simulation 3D Probabilistic Radar | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following

More About

- “Automate Testing for Highway Lane Following” on page 7-667
- “Highway Lane Following with Intelligent Vehicles” on page 7-779
- “Highway Lane Change” on page 7-598
- “Visual Perception Using Monocular Camera” on page 7-78
- “Forward Collision Warning Using Sensor Fusion” on page 7-125
- “Lane Following Control with Sensor Fusion and Lane Detection” on page 7-280
- “Simulate Vision and Radar Sensors in Unreal Engine Environment” on page 7-647
- “Visualize Sensor Data from Unreal Engine Simulation Environment” on page 6-36
- “Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox” on page 6-10

Automate Testing for Highway Lane Following

This example shows how to assess the functionality of a lane-following application by defining scenarios based on requirements, automating testing of components and the generated code for those components. The components include lane-detection, sensor fusion, decision logic, and controls. This example builds on the "Highway Lane Following" on page 7-653 example.

Introduction

A highway lane-following system steers a vehicle to travel within a marked lane. It also maintains a set velocity or safe distance from a preceding vehicle in the same lane. The system typically includes lane detection, sensor fusion, decision logic, and controls components. System-level simulation is a common technique for assessing functionality of the integrated components. Simulations are configured to test scenarios based on system requirements. Automatically running these simulations enables regression testing to verify system-level functionality.

The "Highway Lane Following" on page 7-653 example showed how to simulate a system-level model for lane-following. This example shows how to automate testing that model against multiple scenarios using Simulink Test™. The scenarios are based on system-level requirements. In this example, you will:

1. **Review requirements:** The requirements describe system-level test conditions. Simulation test scenarios are created to represent these conditions.
2. **Review the test bench model:** Review the system-level lane-following test bench model that contains metric assessments. These metric assessments integrate the test bench model with Simulink Test for the automated testing.
3. **Disable runtime visualizations:** Runtime visualizations are disabled to reduce execution time for the automated testing.
4. **Automate testing:** A test manager is configured to simulate each test scenario, assess success criteria, and report results. The results are explored dynamically in the test manager and exported to a PDF for external reviewers.
5. **Automate testing with generated code:** The lane detection, sensor fusion, decision logic, and controls components are configured to generate C++ code. The automated testing is run on the generated code to verify expected behavior.
6. **Automate testing in parallel:** Overall execution time for running the tests is reduced using parallel computing on a multi-core computer.

Testing the system-level model requires a photorealistic simulation environment. In this example, you enable system-level simulation through integration with the Unreal Engine from Epic Games®. The 3D simulation environment requires a Windows® 64-bit platform.

```
if ~ispc
    error("The 3D simulation environment requires a Windows 64-bit platform");
end
```

To ensure reproducibility of the simulation results, set the random seed.

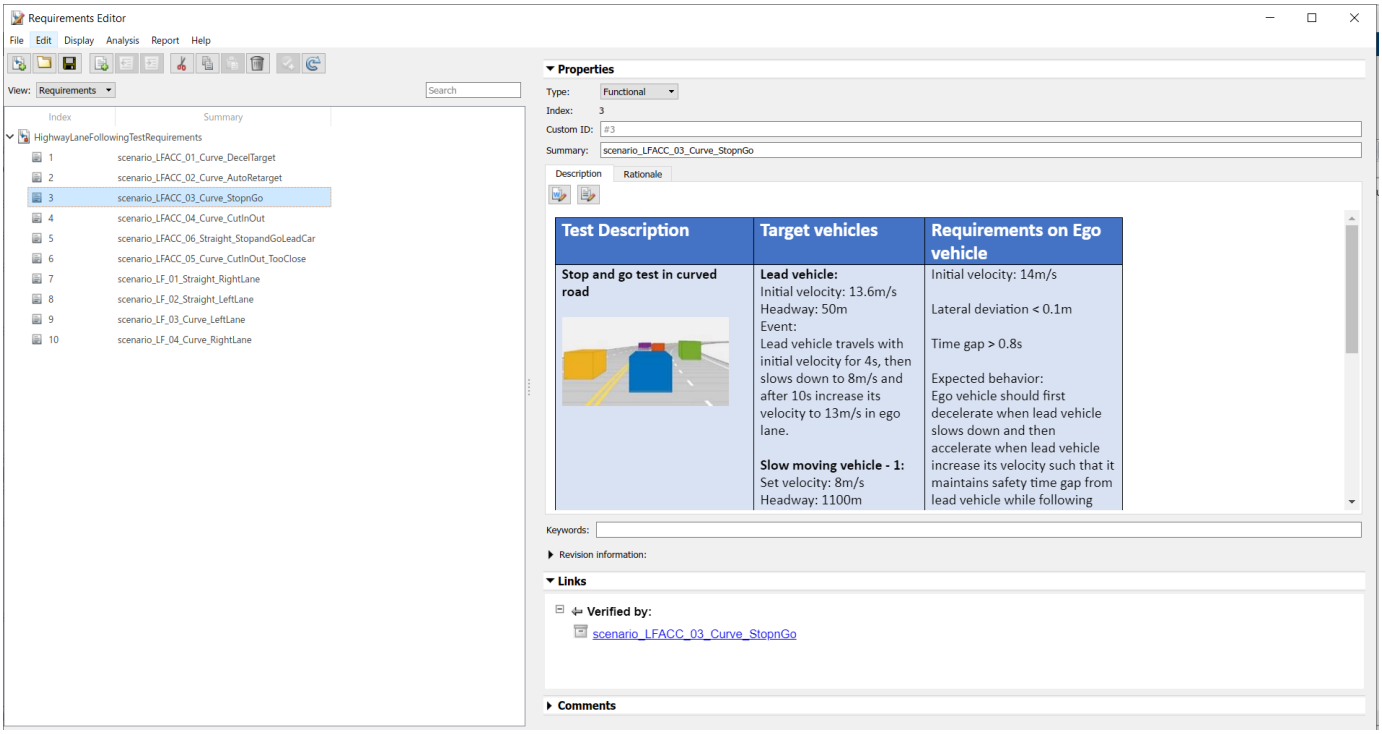
```
rng(0);
```

Review Requirements

Simulink Requirements™ lets you author, analyze, and manage requirements within Simulink. This example contains ten test scenarios, with high-level testing requirements defined for each scenario. Open the requirement set.

```
open('HighwayLaneFollowingTestRequirements.slreqx')
```

Alternatively, you can also open the file from the **Requirements** tab of the Requirements Manager app in Simulink.



Each row in this file specifies the requirements in textual and graphical formats for testing the lane-following system for a test scenario. The scenarios with the **scenario_LF_** prefix enable you to test lane-detection and lane-following algorithms without obstruction by other vehicles. The scenarios with the **scenario_LFACC_** prefix enable you to test lane-detection, lane-following, and ACC behavior with other vehicles on the road.

1. `scenario_LF_01_Straight_RightLane` — Straight road scenario with ego vehicle in right lane.
2. `scenario_LF_02_Straight_LeftLane` — Straight road scenario with ego vehicle in left lane.
3. `scenario_LF_03_Curve_LeftLane` — Curved road scenario with ego vehicle in left lane.
4. `scenario_LF_04_Curve_RightLane` — Curved road scenario with ego vehicle in right lane.
5. `scenario_LFACC_01_Curve_DecelTarget` — Curved road scenario with a decelerating lead vehicle in ego lane.
6. `scenario_LFACC_02_Curve_AutoRetarget` — Curved road scenario with changing lead vehicles in ego lane. This scenario tests the ability of the ego vehicle to retarget to a new lead vehicle while driving along a curve.

7 `scenario_LFACC_03_Curve_StopnGo` — Curved road scenario with a lead vehicle slowing down in ego lane.

8 `scenario_LFACC_04_Curve_CutInOut` — Curved road scenario with a lead car cutting into ego lane.

9 `scenario_LFACC_05_Curve_CutInOut_TooClose` — Curved road scenario with a lead car cutting aggressively into the ego lane.

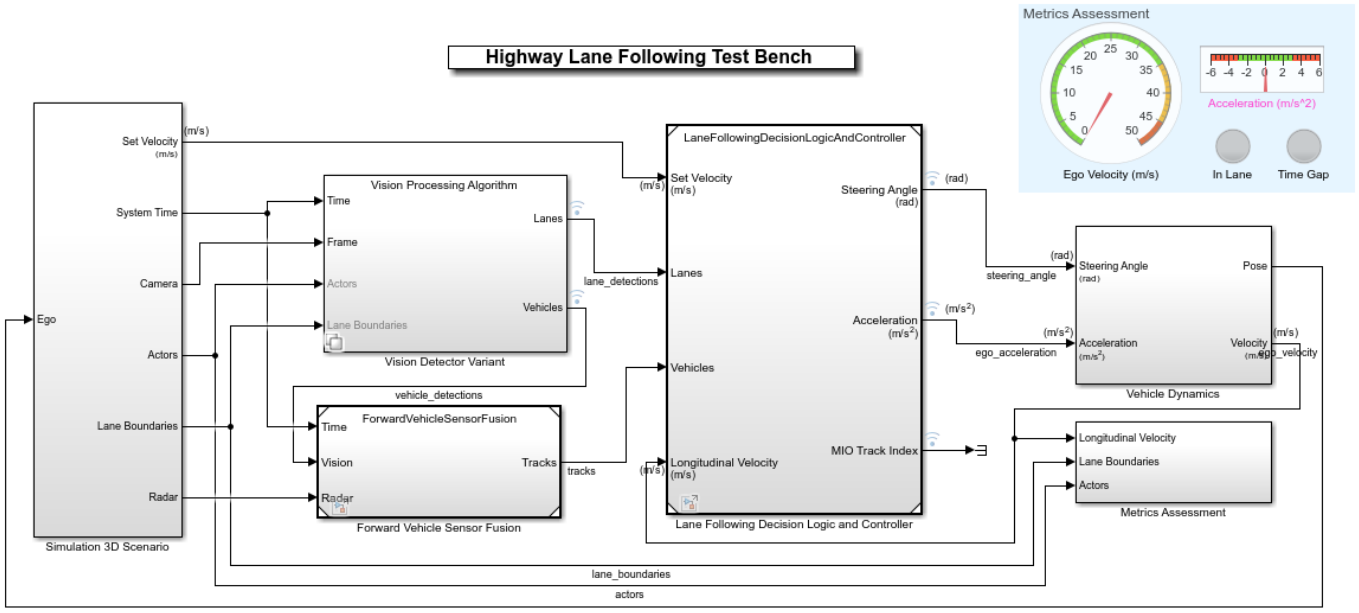
10 `scenario_LFACC_06_Straight_StopandGoLeadCar` — Straight road scenario with a lead vehicle that breaks down in ego lane.

These requirements are implemented as test scenarios with the same names as the scenarios used in the `HighwayLaneFollowingTestBench` model.

Review Test Bench Model

This example reuses the `HighwayLaneFollowingTestBench` model from the "Highway Lane Following" on page 7-653 example. Open the test bench model.

```
open_system("HighwayLaneFollowingTestBench");
```



Copyright 2019-2020 The MathWorks, Inc.

This test bench model has **Simulation 3D Scenario**, **Vision Detector Variant**, **Forward Vehicle Sensor Fusion**, **Lane Following Decision and Controller** and **Vehicle Dynamics** components.

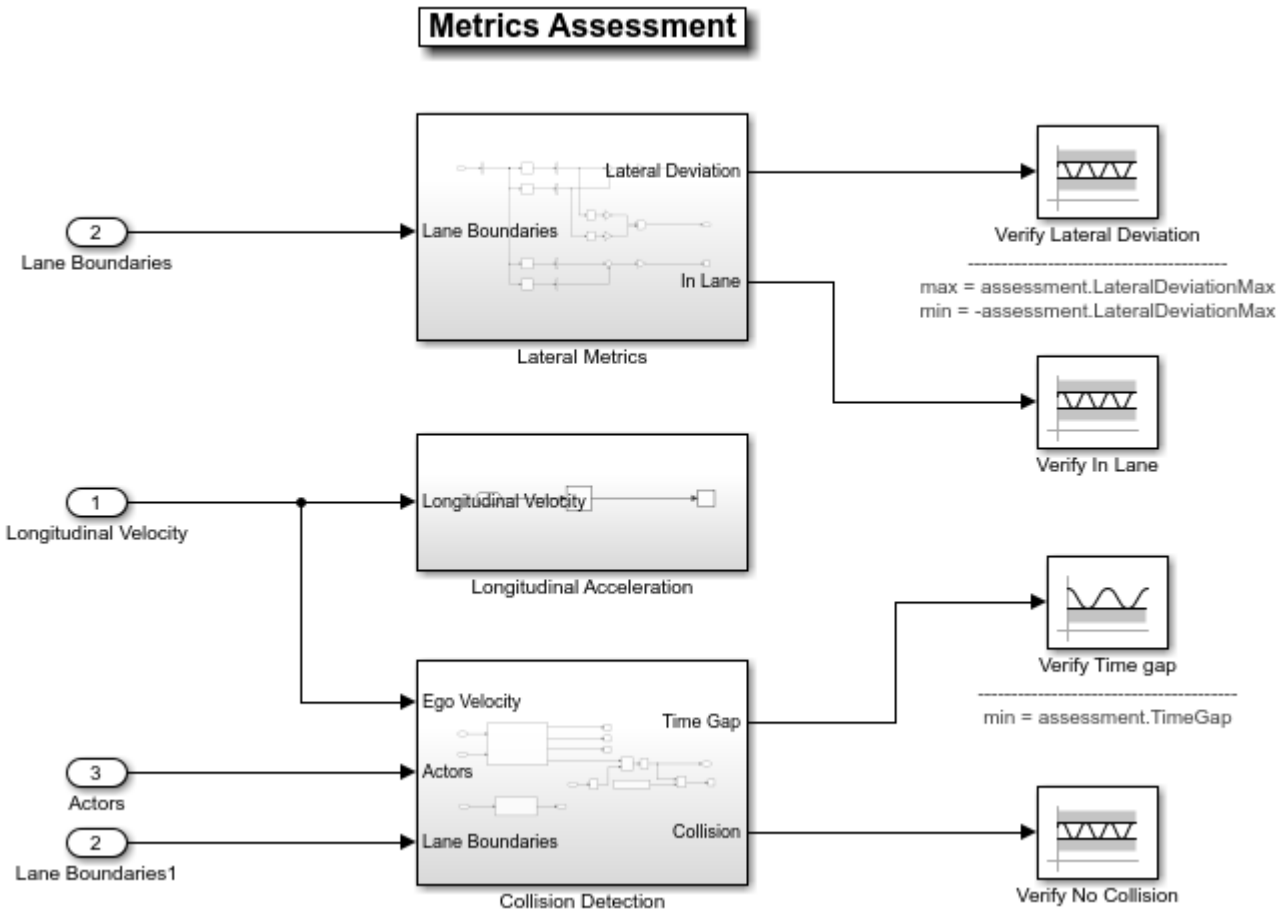
This test bench model is configured using the `helperSLHighwayLaneFollowingSetup` script. This setup script takes two inputs: `scenarioName` and `visionVariantAlgorithm`. `scenarioName` can be any one of the previously described test scenarios. `visionVariantAlgorithm` can be either `ProbabilisticDetectionSensor` or `VisionProcessingAlgorithm`. The `ProbabilisticDetectionSensor` variant enables you to test integration of the control algorithm in the 3D simulation environment, without also integrating the vision processing algorithm. The `VisionProcessingAlgorithm` variant enables you to test integration of the control algorithm and vision processing algorithm in the 3D simulation environment. To run the setup script, use code:

```
scenarioName = "scenario_LFACC_03_Curve_StopnGo";
visionVariantAlgorithm = "VisionProcessingAlgorithm";
helperSLHighwayLaneFollowingSetup(scenarioName,visionVariantAlgorithm);
```

You can now simulate the model and visualize the results. For more details on the analysis of the simulation results and the design of individual components in the test bench model, see the “Highway Lane Following” on page 7-653 example.

In this example, the focus is more on automating the simulation runs for this test bench model using Simulink Test for the different test scenarios. The **Metrics Assessment** subsystem enables integration of system-level metric evaluations with Simulink Test. This subsystem uses Check Static Range (Simulink) blocks for this integration. Open the **Metrics Assessment** subsystem.

```
open_system("HighwayLaneFollowingTestBench/Metrics Assessment");
```



In this example, four metrics are used to assess the lane-following system.

- **Verify Lateral Deviation**: Verifies that the lateral deviation from the centerline of the lane is within prescribed thresholds for the corresponding scenario. Prescribed thresholds are defined while authoring the test scenario.
- **Verify In Lane**: Verifies that the ego vehicle is following one of the lanes on the road throughout the simulation.

- **Verify Time gap**: Verifies that the time gap between the ego vehicle and the lead vehicle is above 0.8 seconds. The time gap between the two vehicles is defined as the ratio of the calculated headway distance to the ego vehicle velocity.
- **Verify No Collision**: Verifies that the ego vehicle does not collide with the lead vehicle at any point during the simulation.

Disable Runtime Visualizations

The system-level test bench model visualizes intermediate outputs during the simulation for the analysis of different components in the model. These visualizations are not required when the tests are automated. You can reduce execution time for the automated testing by disabling them.

Disable runtime visualizations for the **Lane Marker Detector** subsystem.

```
load_system('LaneMarkerDetector');
blk = 'LaneMarkerDetector/Lane Marker Detector';
set_param(blk,'EnableDisplays','off');
```

Disable runtime visualizations for the **Vision Vehicle Detector** subsystem.

```
blk = 'HighwayLaneFollowingTestBench/Vision Detector Variant/Vision Processing Algorithm/Vision V
set_param(blk,'EnableDisplay','off');
```

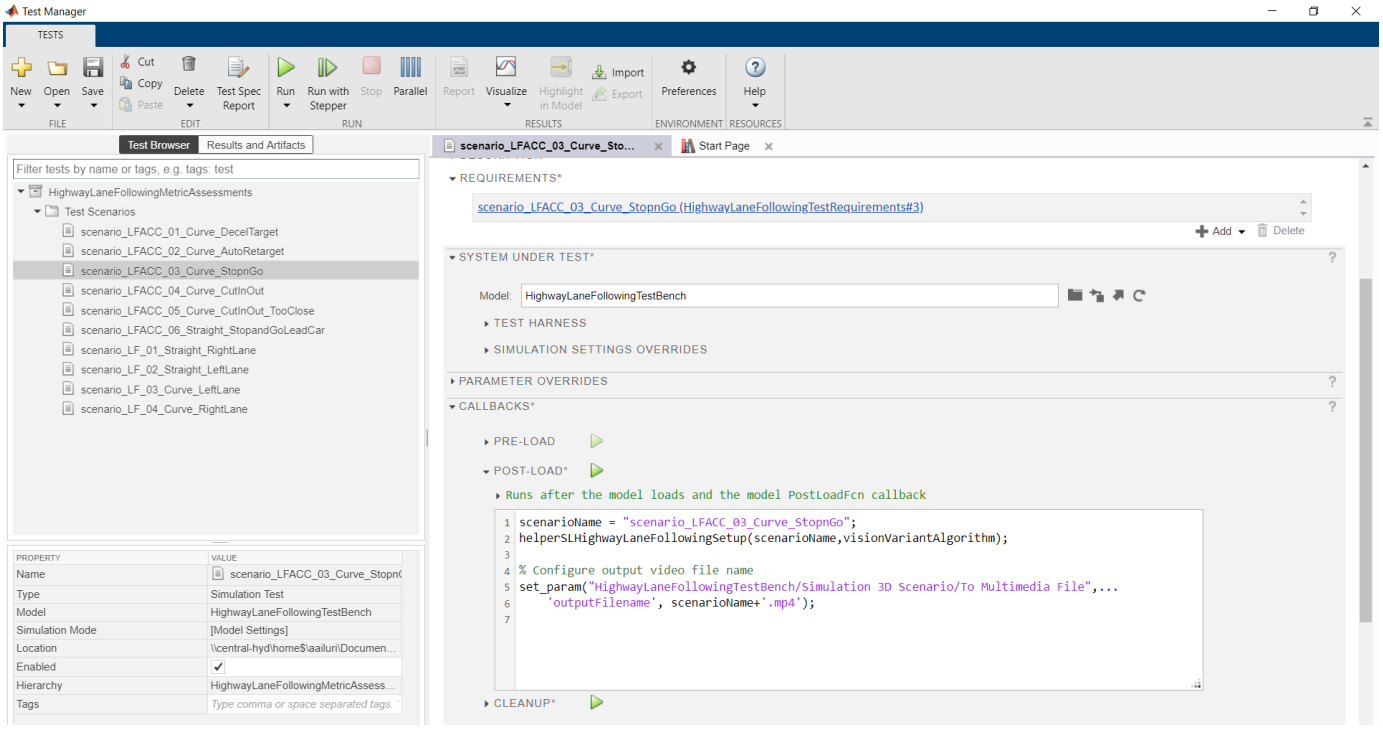
Configure the Simulation 3D Scene Configuration block to run the Unreal Engine in headless mode, where the 3D simulation window is disabled.

```
blk = 'HighwayLaneFollowingTestBench/Simulation 3D Scenario/Simulation 3D Scene Configuration';
set_param(blk,'EnableWindow','off');
```

Automate Testing

The Test Manager is configured to automate the testing of the lane-following application. Open the `HighwayLaneFollowingMetricAssessments.mldatx` test file in the Test Manager.

```
sltestmgr;
sltest.testmanager.load('HighwayLaneFollowingMetricAssessments.mldatx');
```



Observe the populated test cases that were authored previously in this file. Each test case is linked to the corresponding requirement in the Requirements Editor for traceability. These tests are configured to run with the `VisionProcessingAlgorithm` variant. You can also configure tests to run with the `ProbabilisticDetectionSensor` variant by modifying the `visionVariant` variable in the test suite `SETUP` callback.

Each test case uses the `POST-LOAD` callback to run the setup script with appropriate inputs and to configure the output video file name. After the simulation of the test case, it invokes `helperGenerateFilesForLaneFollowingReport` from the `CLEAN-UP` callback to generate the plots explained in the “Highway Lane Following” on page 7-653 example.

Run and explore results for a single test scenario:

To test the system-level model with the `scenario_LFACC_03_Curve_StopnGo` test scenario from Simulink Test, use this code:

```
testFile = sltest.testmanager.TestFile('HighwayLaneFollowingMetricAssessments.mldatx');
testSuite = getTestSuiteByName(testFile,'Test Scenarios');
testCase = getTestCaseByName(testSuite,'scenario_LFACC_03_Curve_StopnGo');
resultObj = run(testCase);
```

To generate a report after the simulation, use this code:

```
sltest.testmanager.report(resultObj,'Report.pdf',...,
    'Title','Highway Lane Following',...
    'IncludeMATLABFigures',true,...
    'IncludeErrorMessages',true,...
    'IncludeTestResults',0,'LaunchReport',true);
```

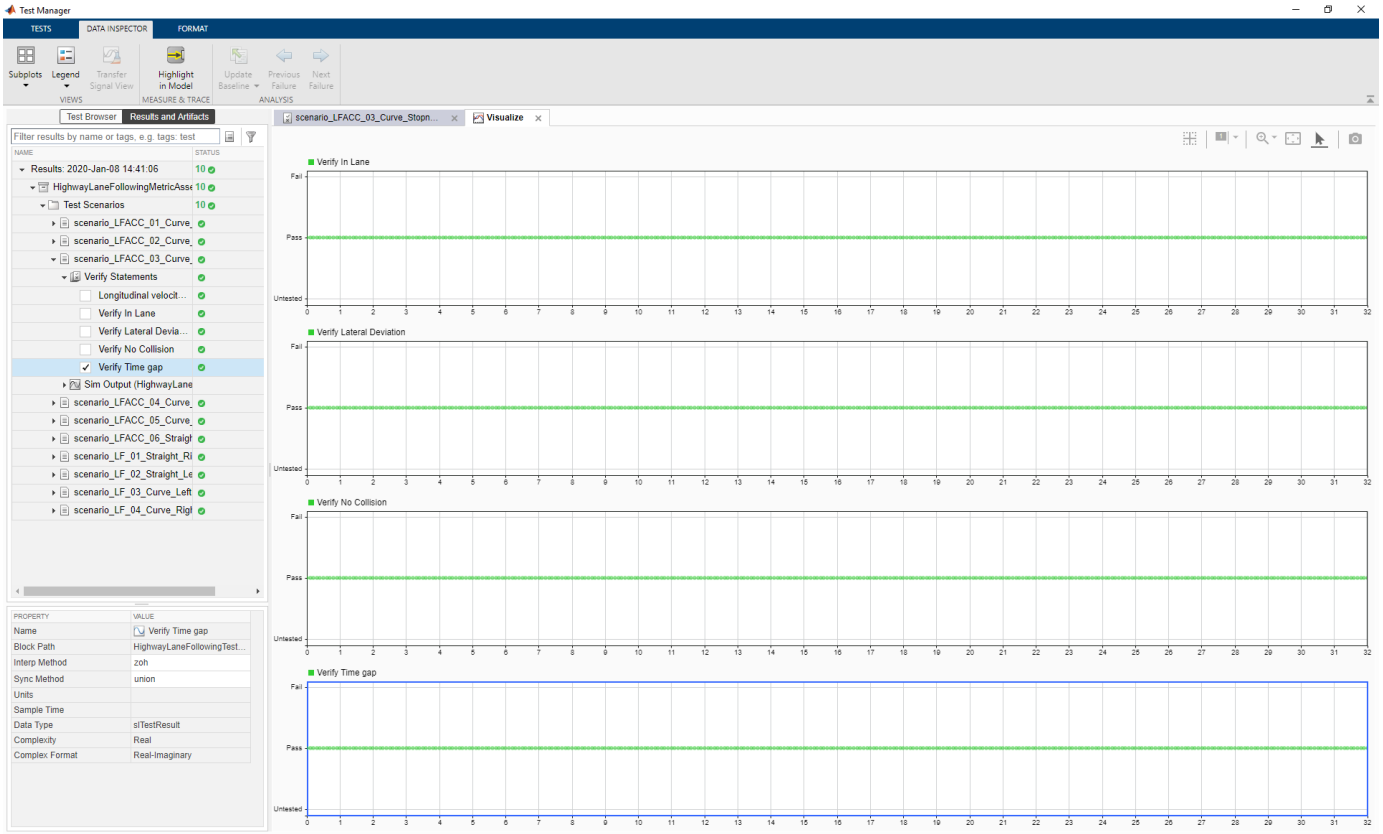
Examine the `Report.pdf`. Observe that the **Test environment** section shows the platform on which the test is run and the MATLAB® version used for testing. The **Summary** section shows the outcome

of the test and duration of the simulation in seconds. The **Results** section shows pass/fail results based on the assessment criteria. This section also shows the plots logged from the `helperGenerateFilesForLaneFollowingReport` function.

Run and explore results for all test scenarios:

You can simulate the system for all the tests by using `sltest.testmanager.run`. Alternatively, you can simulate the system by clicking **Play** in the Test Manager app.

After completion of the test simulations, the results for all the tests can be viewed in the **Results and Artifacts** tab of the Test Manager. For each test case, the Check Static Range (Simulink) blocks in the model are associated with the Test Manager to visualize overall pass/fail results.



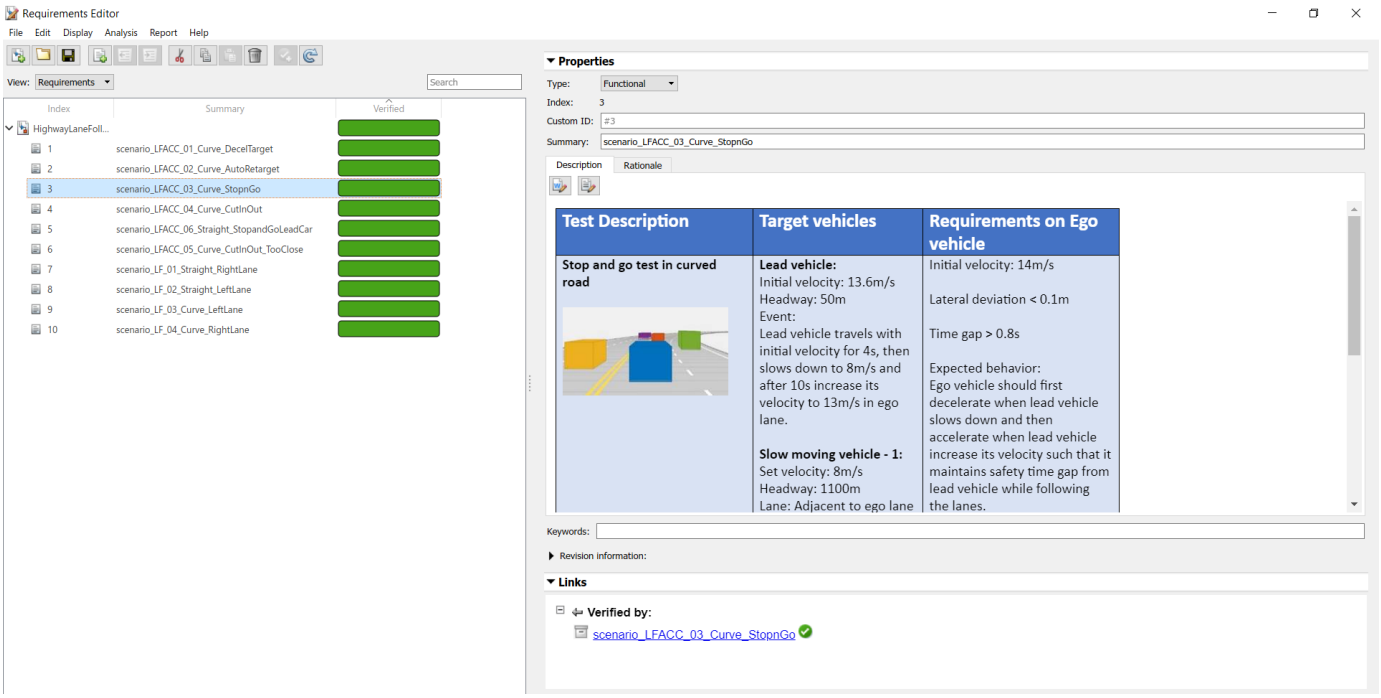
You can find the generated report in current working directory. This report contains a detailed summary of pass/fail statuses and plots for each test case.

Summary

Name	Outcome	Duration (Seconds)
HighwayLaneFollowingMetricAssessments	10	1218.653
Test Scenarios	10	1218.654
scenario_LFACC_01_Curve_DecelTarget		440.839
scenario_LFACC_02_Curve_AutoRetarget		410.52
scenario_LFACC_03_Curve_StopnGo		496.208
scenario_LFACC_04_Curve_CutInOut		261.693
scenario_LFACC_05_Curve_CutInOut_TooClose		468.847
scenario_LFACC_06_Straight_StopandGoLeadCar		263.3
scenario_LF_01_Straight_RightLane		314.418
scenario_LF_02_Straight_LeftLane		128.075
scenario_LF_03_Curve_LeftLane		374.711
scenario_LF_04_Curve_RightLane		190.839

Verify test status in Requirements Editor:

Open the Requirements Editor and select **Display**. Then, select **Verification Status** to see a verification status summary for each requirement. Green and red bars indicate the pass/fail status of simulation results for each test.



Automate Testing with Generated Code

The `HighwayLaneFollowingTestBench` model enables integrated testing of **Lane Marker Detector**, **Forward Vehicle Sensor Fusion**, and **Lane Following Decision Logic and Controller** components. It is often helpful to perform regression testing of these components through software-in-the-loop (SIL) verification. If you have a Simulink Coder™ license, then you can generate code for these components. This workflow lets you verify that the generated code produces expected results that match the system-level requirements throughout simulation.

Set **Lane Marker Detector** to run in Software-in-the-loop mode.

```
model = 'HighwayLaneFollowingTestBench/Vision Detector Variant/Vision Processing Algorithm/Lane M
set_param(model,'SimulationMode','Software-in-the-loop');
```

Set **Forward Vehicle Sensor Fusion** to run in Software-in-the-loop mode.

```
model = 'HighwayLaneFollowingTestBench/Forward Vehicle Sensor Fusion';
set_param(model,'SimulationMode','Software-in-the-loop');
```

Set **Lane Following Decision Logic and Controller** to run in Software-in-the-loop mode.

```
model = 'HighwayLaneFollowingTestBench/Lane Following Decision Logic and Controller';
set_param(model,'SimulationMode','Software-in-the-loop');
```

Now, run `sltest.testmanager.run` to simulate the system for all the test scenarios. After the completion of tests, review the plots and results in the generated report.

Automate Testing in Parallel

If you have a Parallel Computing Toolbox™ license, then you can configure Test Manager to execute tests in parallel using a parallel pool. To run tests in parallel, save the models after disabling the runtime visualizations using `save_system('LaneMarkerDetector')` and

`save_system('HighwayLaneFollowingTestBench')`. Test Manager uses the default Parallel Computing Toolbox cluster and executes tests only on the local machine. Running tests in parallel can speed up execution and decrease the amount of time it takes to get test results. For more information on how to configure tests in parallel from the Test Manager, see “Run Tests Using Parallel Execution” (Simulink Test).

See Also

More About

- “Highway Lane Following” on page 7-653

Traffic Light Negotiation

This example shows how to design and test decision logic for negotiating a traffic light at an intersection.

Introduction

Decision logic for negotiating traffic lights is a fundamental component of an automated driving application. The decision logic must react to inputs like the state of the traffic light and surrounding vehicles. The decision logic then provides the controller with the desired velocity and path. Since traffic light intersections are dangerous to test, simulating such driving scenarios can provide insight into the interactions of the decision logic and the controller.

This example shows how to design and test the decision logic for negotiating a traffic light. The decision logic in this example reacts to the state of the traffic light, distance to the traffic light, and distance to the closest vehicle ahead. In this example, you will:

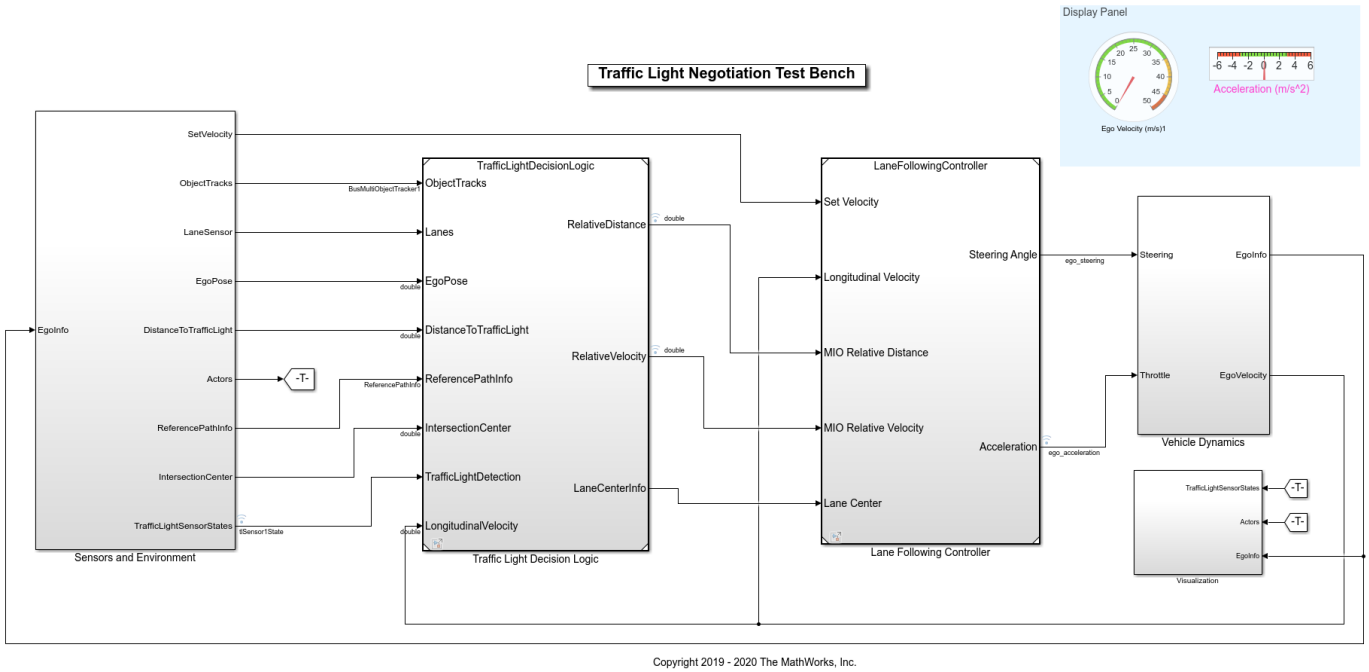
1. **Explore the test bench model**: The model contains the traffic light sensors and environment, traffic light decision logic, controls, and vehicle dynamics.
2. **Model the traffic light decision logic**: The traffic light decision logic arbitrates between a lead vehicle and an upcoming traffic light. It also provides a reference path for the ego vehicle to follow at an intersection in the absence of lanes.
3. **Simulate a left turn with traffic light and a lead vehicle**: The model is configured to test the interactions between the traffic light decision logic and controls of the ego vehicle, while approaching an intersection in the presence of a lead vehicle.
4. **Simulate a left turn with traffic light and cross traffic**: The model is configured to test the interactions between the traffic light decision logic and controls of the ego vehicle when there is cross traffic at the intersection.
5. **Explore other scenarios**: These scenarios test the system under additional conditions.

You can apply the modeling patterns used in this example to test your own decision logic and controls to negotiate traffic lights.

Explore Test Bench Model

To explore the behavior of the traffic light negotiation system, open the simulation test bench model for the system.

```
open_system("TrafficLightNegotiationTestBench");
```



Opening this model runs the `helperSLTrafficLightNegotiationSetup` script that initializes the road scenario using the `drivingScenario` object in the base workspace. It runs the default test scenario, `scenario_TLN_left_turn_with_cross_over_vehicle`, that contains an ego vehicle and two other vehicles. This setup script also configures the controller design parameters, vehicle model parameters, and Simulink® bus signals required for defining the inputs and outputs for the `TrafficLightNegotiationTestBench` model.

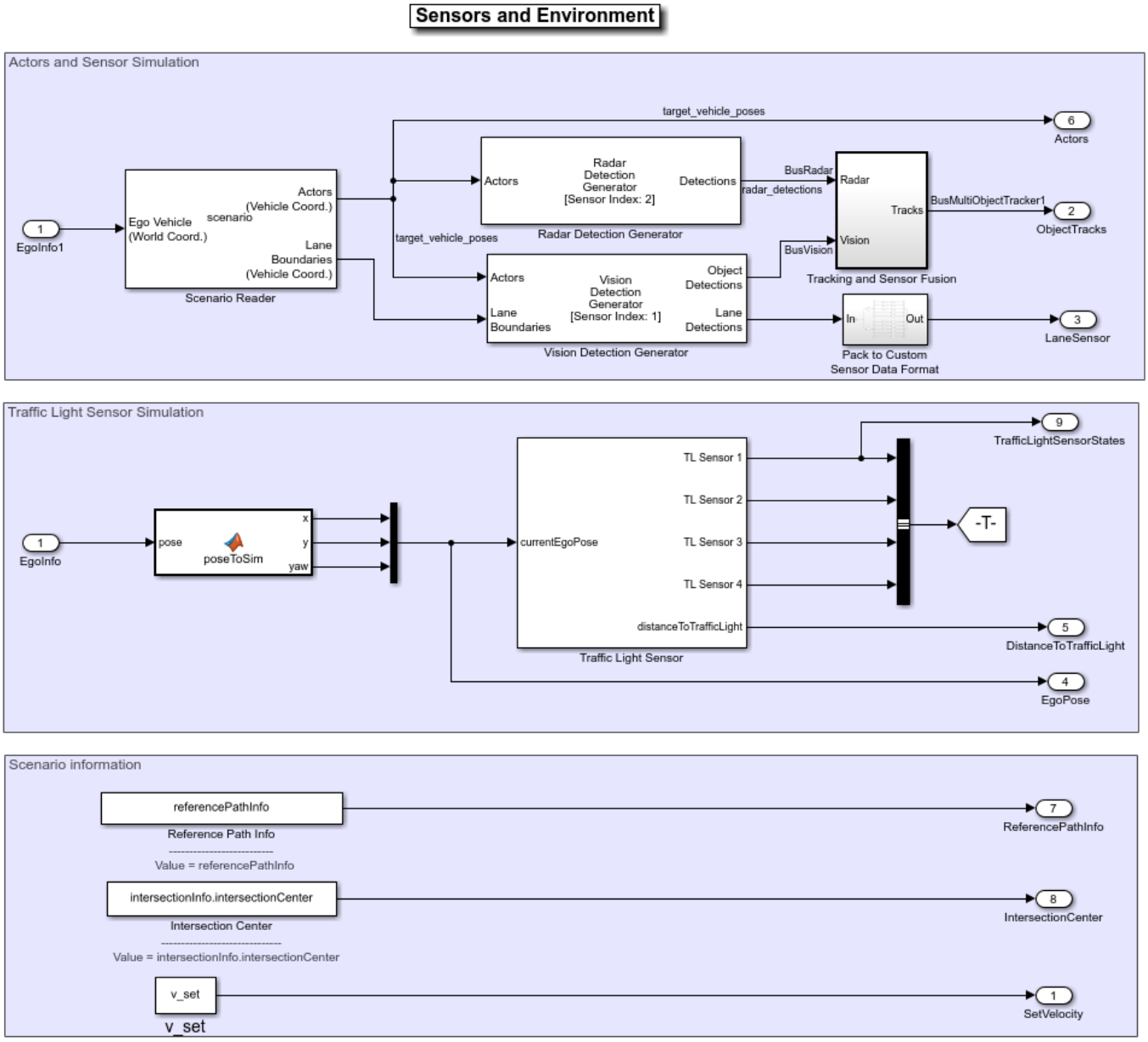
The test bench model contains the following subsystems:

1. **Sensors and Environment**: Models the traffic light sensor, road network, vehicles, and the camera and radar sensors used for simulation.
2. **Traffic Light Decision Logic**: Arbitrates between the traffic light and other lead vehicles or cross-over vehicles at the intersection.
3. **Lane-Following Controller**: Generates longitudinal and lateral controls.
4. **Vehicle Dynamics**: Models the ego vehicle using a Bicycle Model block and updates its state using commands received from the **Lane Following Controller** subsystem.
5. **Visualization**: Plots the world coordinate view of the road network, vehicles, and the traffic light state during simulation.

The **Lane Following Controller** reference model and the **Vehicle Dynamics** subsystem are reused from the “Highway Lane Following” on page 7-653 example. This example focuses on the **Sensors and Environment** and **Traffic Light Decision Logic** subsystems.

The **Sensors and Environment** subsystem configures the road network, defines target vehicle trajectories, and synthesizes sensors. Open the **Sensors and Environment** subsystem.

```
open_system("TrafficLightNegotiationTestBench/Sensors and Environment");
```

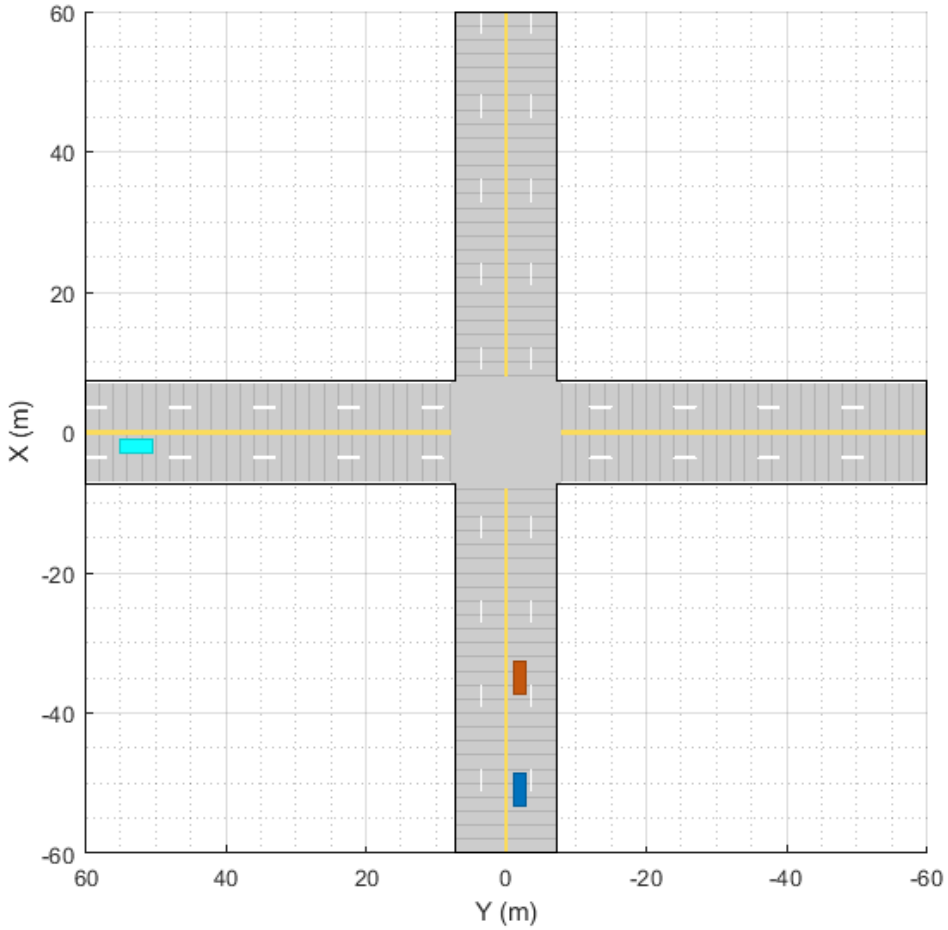


The scenario and sensors on the ego vehicle are specified by the following parts of the subsystem:

- The Scenario Reader block is configured to take in ego vehicle information to perform a closed-loop simulation. It outputs ground truth information of lanes and actors in ego vehicle coordinates. This block reads the `drivingScenario` object variable, `scenario`, from the base workspace, which contains a road network compatible with the `TrafficLightNegotiationTestBench` model.

Plot the road network provided by the scenario.

```
hFigScenario = figure('Position', [1 1 800 600]);
plot(scenario, 'Parent', axes(hFigScenario));
```



This default scenario has one intersection with an ego vehicle, one lead vehicle, and one cross-traffic vehicle.

Close the figure.

```
close(hFigScenario);
```

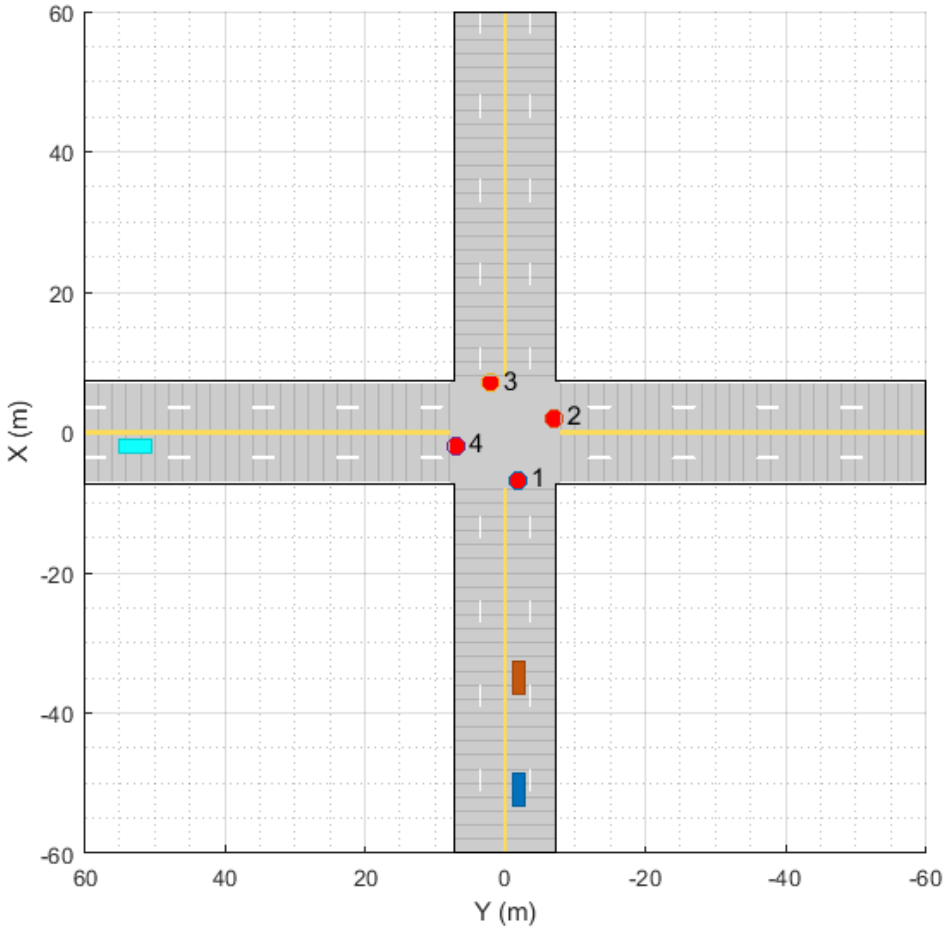
The **Tracking and Sensor Fusion** subsystem fuses vehicle detections from Radar Detection Generator and Vision Detection Generator blocks by using a Multi-Object Tracker block to provide object tracks surrounding the ego vehicle.

The Vision Detection Generator block also provides lane detections with respect to the ego vehicle that helps in identifying vehicles present in the ego lane.

The **Traffic Light Sensor** subsystem simulates the traffic lights. It is configured to support four traffic light sensors at the intersection, **TL Sensor 1**, **TL Sensor 2**, **TL Sensor 3**, and **TL Sensor 4**.

Plot the scenario with traffic light sensors.

```
hFigScenario = helperPlotScenarioWithTrafficLights();
```



Observe that this is the same scenario as before, only with traffic light sensors added. These sensors are represented by red circles at the intersection, indicating red traffic lights. The labels for the traffic lights **1**, **2**, **3**, **4** correspond to **TL Sensor 1**, **TL Sensor 2**, **TL Sensor 3**, and **TL Sensor 4**, respectively.

Close the figure.

```
close(hFigScenario);
```

The test scenarios in `TrafficLightNegotiationTestBench` are configured such that the ego vehicle negotiates with **TL Sensor 1**. There are three modes in which you can configure this **Traffic Light Sensor** subsystem:

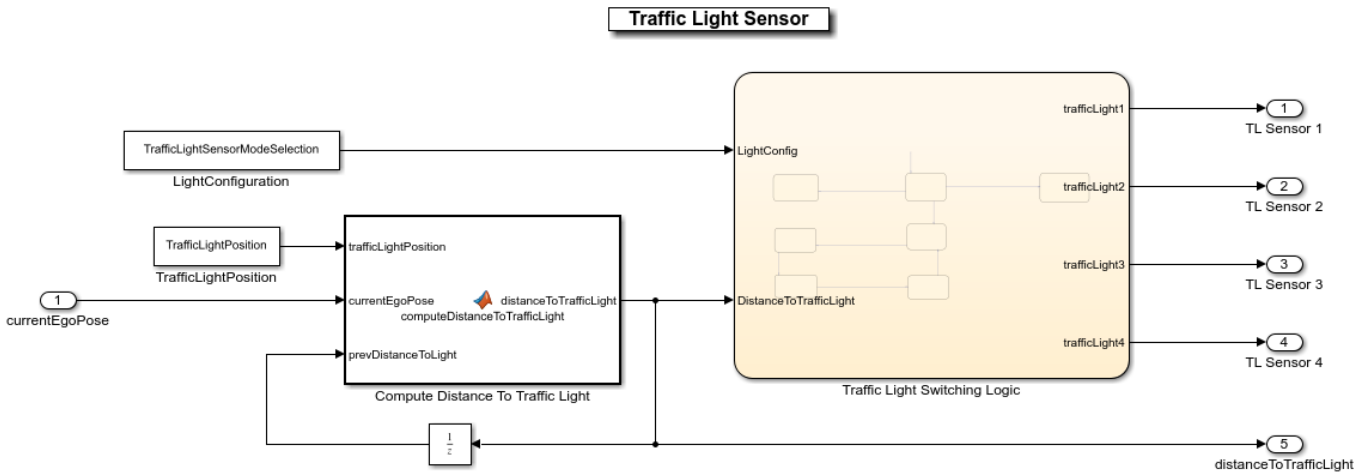
1. **Steady Red**: **TL Sensor 1** and **TL Sensor 3** are always in a red state. The other two traffic lights are always in a green state.
2. **Steady Green**: **TL Sensor 1** and **TL Sensor 3** are always in a green state. The other two traffic lights are always in a red state.

3 **Cycle** [Default]: **TL Sensor 1** and **TL Sensor 3** follow a cyclic pattern: green-yellow-red with predefined timings. Other traffic lights also follow a cyclic pattern: red-green-yellow with predefined timings to complement the **TL Sensor 1** and **TL Sensor 3**.

You can configure this subsystem in one of these modes by using the `Traffic Light Sensor Mode` mask parameter.

Open the **Traffic Light Sensor** subsystem.

```
open_system('TrafficLightNegotiationTestBench/Sensors and Environment/Traffic Light Sensor', 'fo
```



The **Traffic Light Switching Logic** Stateflow® chart implements the traffic light state change logic for the four traffic light sensors. The initial state for all the traffic lights is set to red. Transition to a different mode is based on a trigger condition defined by distance of the ego vehicle to the **TL Sensor 1** traffic light. This distance is defined by the variable `distanceToTrafficLight`. Traffic light transition is triggered if this distance is less than `trafficLightStateTriggerThreshold`. This threshold is currently set to 60 meters and can be changed in the `helperSLTrafficLightNegotiationSetup` script.

The Compute Distance To Traffic Light block calculates `distanceToTrafficLight` using the traffic light position of **TL Sensor 1**, defined by the variable `trafficLightPosition`. This is obtained from the `Traffic Light Position` mask parameter of the **Traffic Light Sensor** subsystem. The value of the mask parameter is set to `intersectionInfo.tlSensor1Position`, a variable set in the base workspace by the `helperSLTrafficLightNegotiationSetup` script. `intersectionInfo` structure is an output from the `helperGetTrafficLightScene` function. This function is used to create the test scenarios that are compatible with the `TrafficLightNegotiationTestBench` model.

The following inputs are needed by the traffic light decision logic and controller to implement their functionalities:

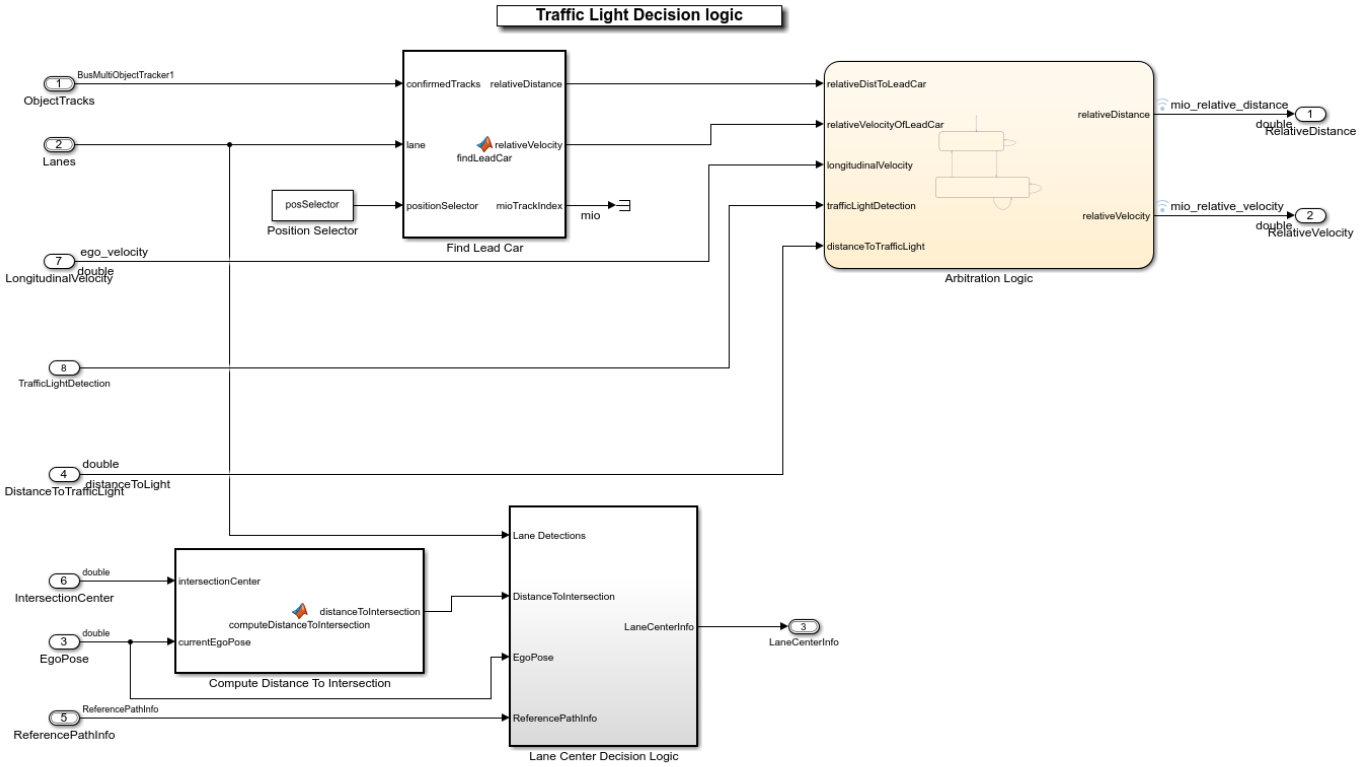
- **ReferencePathInfo** provides a predefined reference trajectory that can be used by the ego vehicle for navigation in absence of lane information. The ego vehicle can go straight, take a left turn, or a right turn at the intersection based on the reference path. This reference path is obtained using `referencePathInfo`, an output from `helperGetTrafficLightScene`. This function takes an input argument to specify the direction of travel at the intersection. The possible values are: `Straight`, `Left`, and `Right`.

- **IntersectionCenter** provides the position of the intersection center of the road network in the scenario. This is obtained using the `intersectionInfo`, an output from `helperGetTrafficLightScene`.
- **Set Velocity** defines the user-set velocity for the controller.

Model Traffic Light Decision Logic

The **Traffic Light Decision Logic** reference model arbitrates between the lead car and the traffic light. It also calculates the lane center information as required by the controller either using the detected lanes or a predefined path. Open the **Traffic Light Decision Logic** reference model.

```
open_system("TrafficLightDecisionLogic");
```

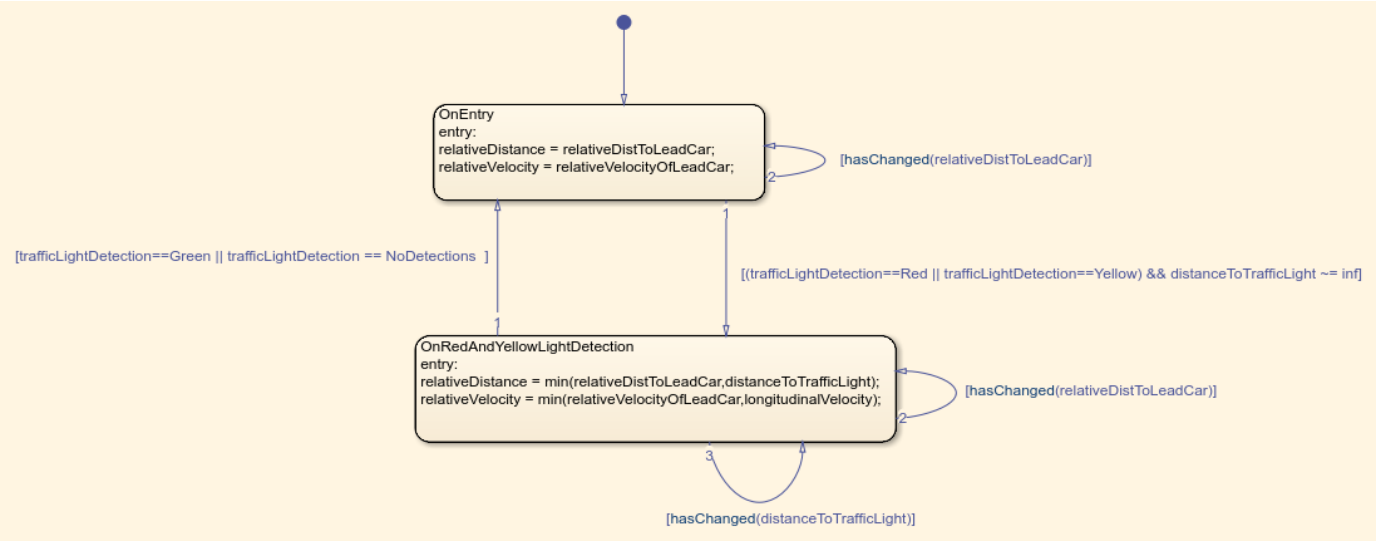


Copyright 2019 The MathWorks, Inc.

The **Find Lead Car** subsystem finds the lead car in the current lane from input object tracks. It provides relative distance, **relativeDistToLeadCar**, and relative velocity, **relativeVelocityOfLeadCar**, with respect to the lead vehicle. If there is no lead vehicle, then this block considers the lead vehicle to be present at infinite distance.

The **Arbitration Logic** Stateflow chart uses the lead car information and implements the logic required to arbitrate between the traffic light and the lead vehicle at the intersection. Open the **Arbitration Logic** Stateflow chart.

```
open_system("TrafficLightDecisionLogic/Arbitration Logic");
```



The **Arbitration Logic** Stateflow chart consists of two states, `OnEntry` and `OnRedAndYellowLightDetection`. If the traffic light state is green or if there are no traffic light detections, the state remains in the `OnEntry` state. If the traffic light state is red or yellow, then the state transitions to the `OnRedAndYellowLightDetection` state. The control flow switches between these states based on `trafficLightDetection` and `distanceToTrafficLight` variables. In each state, relative distance and relative velocity with respect to the most important object (MIO) are calculated. The lead vehicle and the red traffic light are considered as MIOs.

OnEntry:

```
relativeDistance = relativeDistToLeadCar;

relativeVelocity = relativeVelocityOfLeadCar;
```

OnRedAndYellowLightDetection:

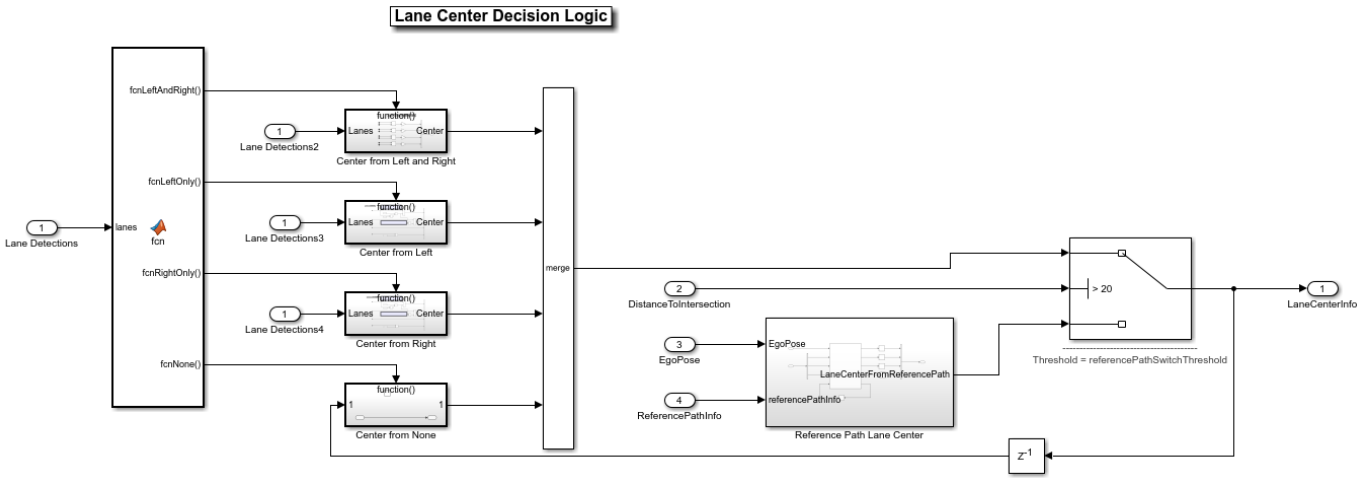
```
relativeDistance = min(relativeDistToLeadCar,distanceToTrafficLight);

relativeVelocity = min(relativeVelocityOfLeadCar,longitudinalVelocity);
```

The **longitudinalVelocity** represents the longitudinal velocity of the ego vehicle.

The Compute Distance To Intersection block computes the distance to the intersection center from the current ego position. Because the intersection has no lanes, the ego vehicle uses this distance to fall back to the predefined reference path at the intersection.

The **Lane Center Decision Logic** subsystem calculates the lane center information as required by the Path Following Control System (Model Predictive Control Toolbox). Open the **Lane Center Decision Logic** subsystem.

```
open_system("TrafficLightDecisionLogic/Lane Center Decision Logic");
```



The **Lane Center Decision Logic** subsystem primarily relies on the lane detections from the Vision Detection Generator block to estimate lane center information like curvature, curvature derivative, lateral offset, and heading angle. However, there are no lane markings to detect at the intersection. In such cases, the lane center information can be estimated from a predefined reference path.

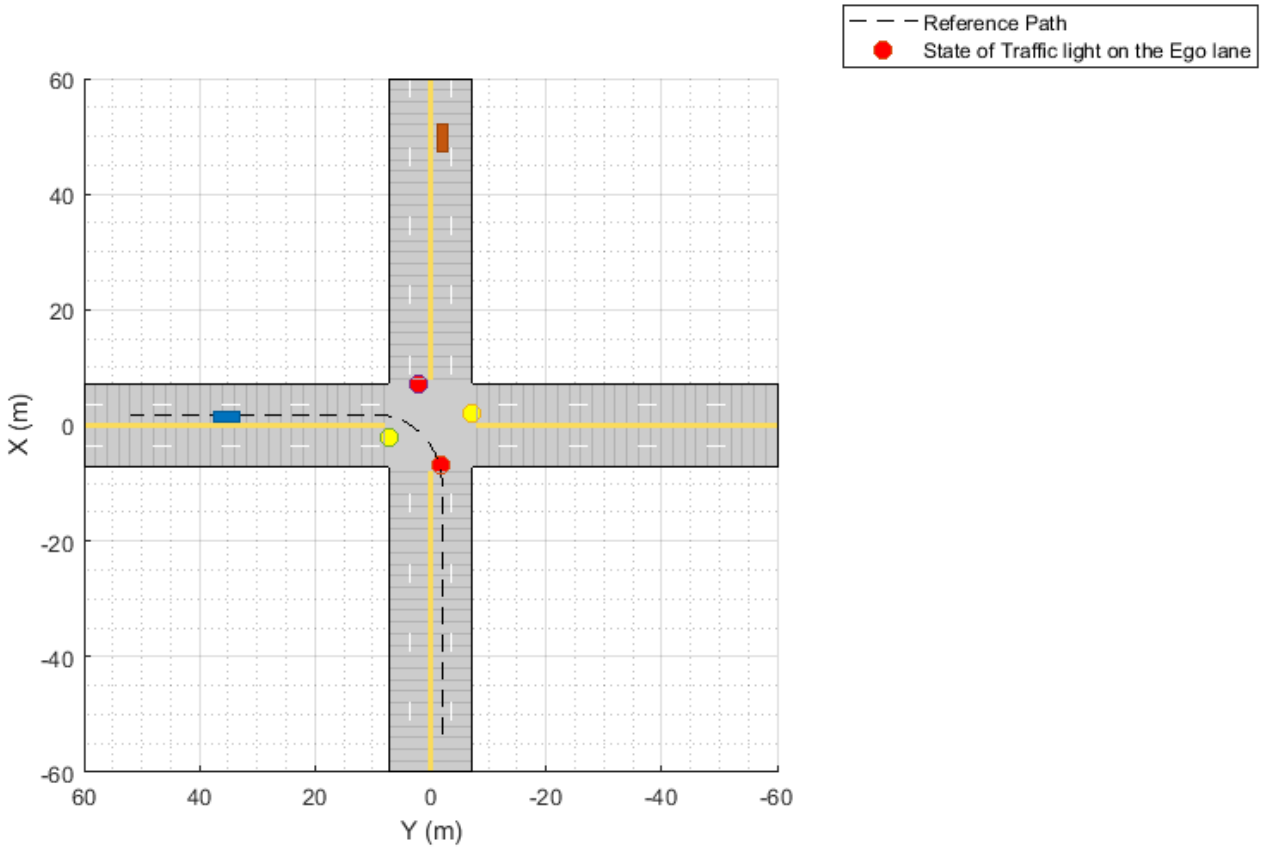
The **Reference Path Lane Center** subsystem computes lane center information based on the current ego pose and predefined reference path. A switch is configured to use **LaneCenterFromReferencePath** when **DistanceToIntersection** is less than `referencePathSwitchThreshold`. This threshold is currently set to 20 meters and can be changed in the `helperSLTrafficLightNegotiationSetup` script.

Simulate Left Turn with Traffic Light and Lead Vehicle

In this test scenario, a lead vehicle travels in the ego lane and crosses the intersection. The traffic light state keeps green for the lead vehicle and turns red for the ego vehicle. The ego vehicle is expected to follow the lead vehicle, negotiate the traffic light, and make a left turn.

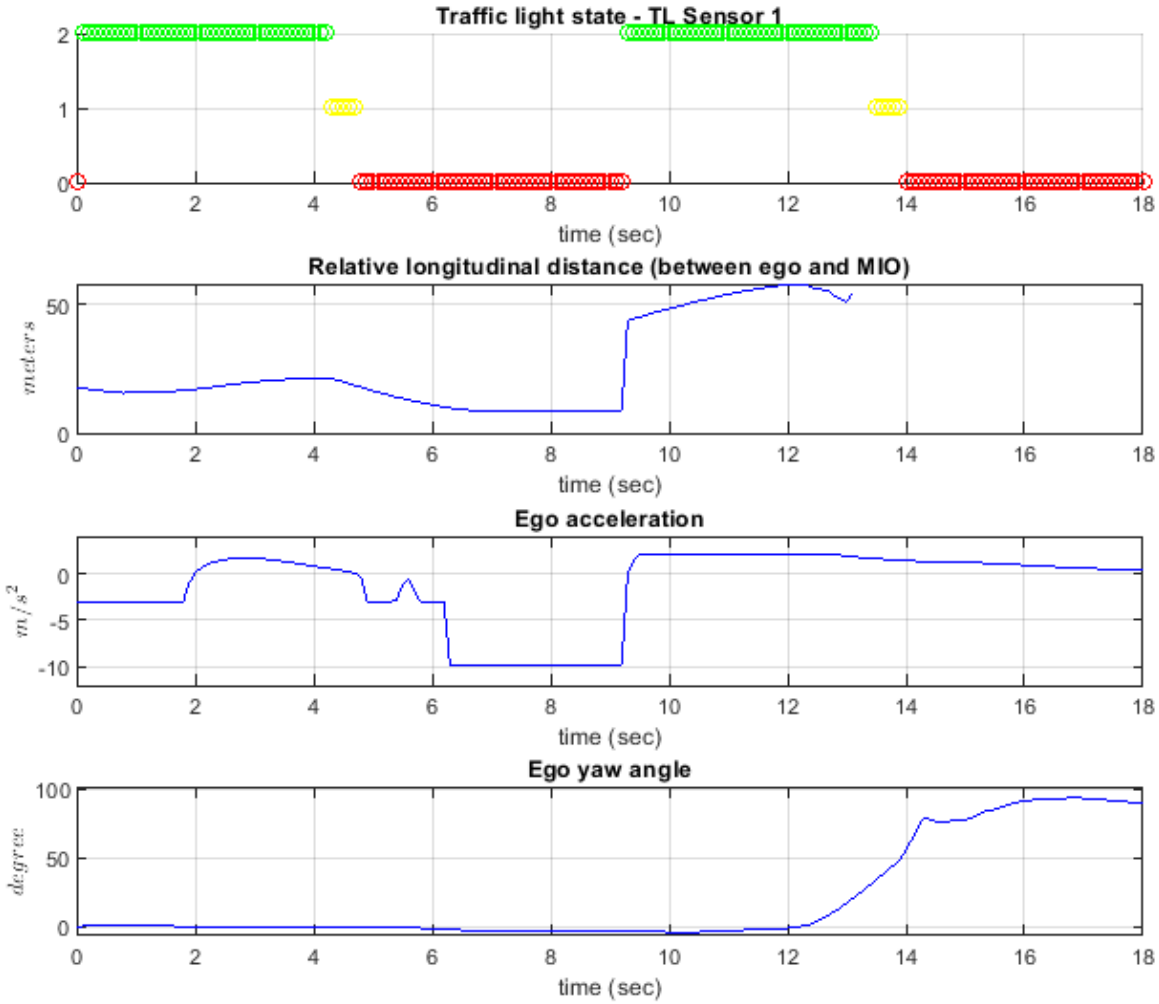
Configure the `TrafficLightNegotiationTestBench` model to use the `scenario_TLN_left_turn_with_lead_vehicle` scenario.

```
helperSLTrafficLightNegotiationSetup("scenario_TLN_left_turn_with_lead_vehicle");
% To reduce command-window output, first turn off the MPC update messages.
mpcverbosity('off');
% Simulate the model.
sim("TrafficLightNegotiationTestBench");
```



Plot the simulation results.

```
hFigResults = helperPlotTrafficLightNegotiationResults(logsout);
```



Examine the results.

- The **Traffic light state - TL Sensor 1** plot shows the traffic light sensor states of **TL Sensor 1**. It changes from green to yellow, then from yellow to red, and then repeats in **Cycle** mode.
- The **Relative longitudinal distance** plot shows the relative distance between the ego vehicle and the MIO. Notice that the ego vehicle follows the lead vehicle from 0 to 4.5 seconds by maintaining a safe distance from it. You can also observe that from 4.5 to 9 seconds, this distance reduces because the red traffic light is detected as an MIO. Also notice the gaps representing infinite distance when there is no MIO.
- The **Ego acceleration** plot shows the acceleration profile from the **Lane Following Controller**. Notice the negative acceleration from 4.5 to 6 seconds, in reaction to the detection of the red traffic light as an MIO. You can also observe the increase in acceleration after 9 seconds, in response to the green traffic light.

- The **Ego yaw angle** plot shows the yaw angle profile of the ego vehicle. Notice the variation in this profile after 12 seconds, in response to the ego vehicle taking a left turn.

Close the figure.

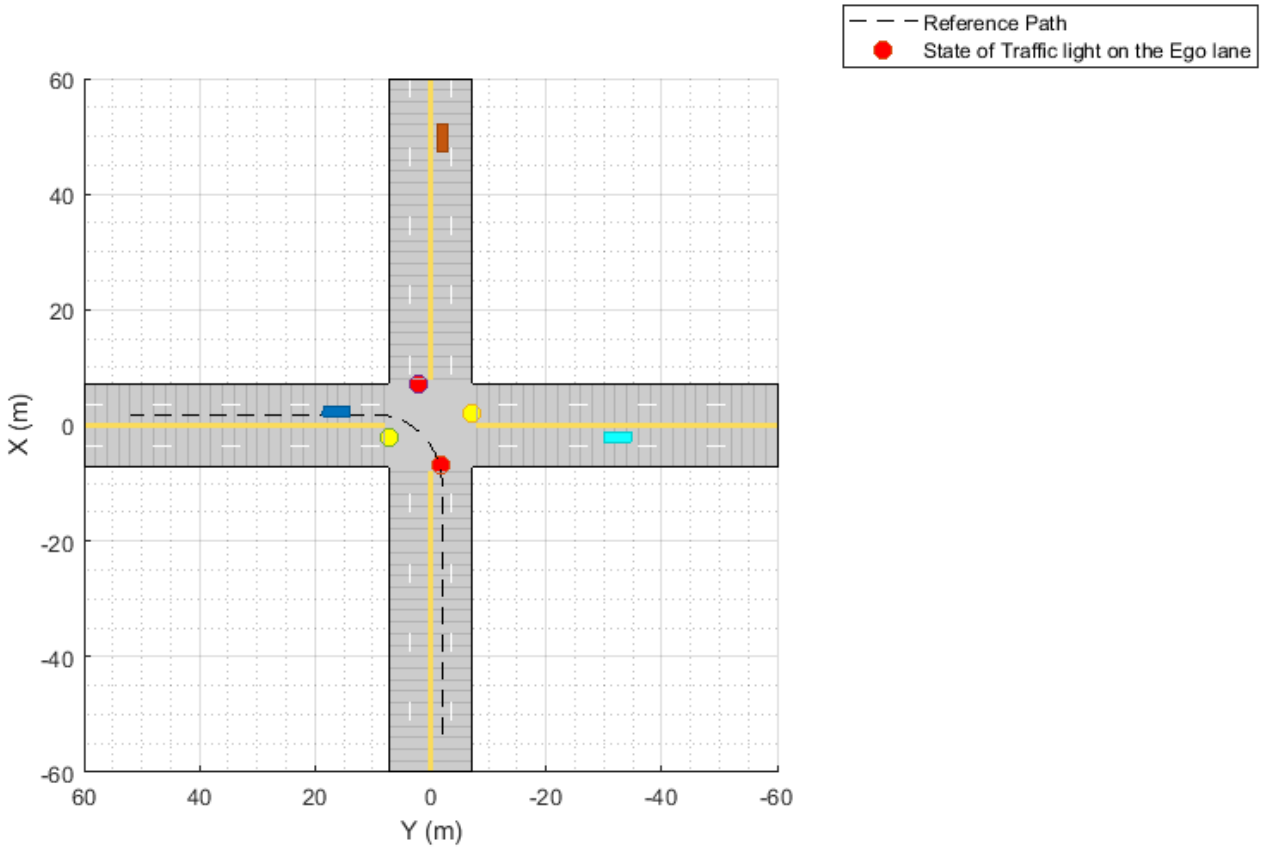
```
close(hFigResults);
```

Simulate Left Turn with Traffic Light and Cross Traffic

This test scenario is an extension to the previous scenario. In addition to the previous conditions, in this scenario, a slow-moving cross-traffic vehicle is in the intersection when the traffic light is green for the ego vehicle. The ego vehicle is expected to wait for the cross-traffic vehicle to pass the intersection before taking the left turn.

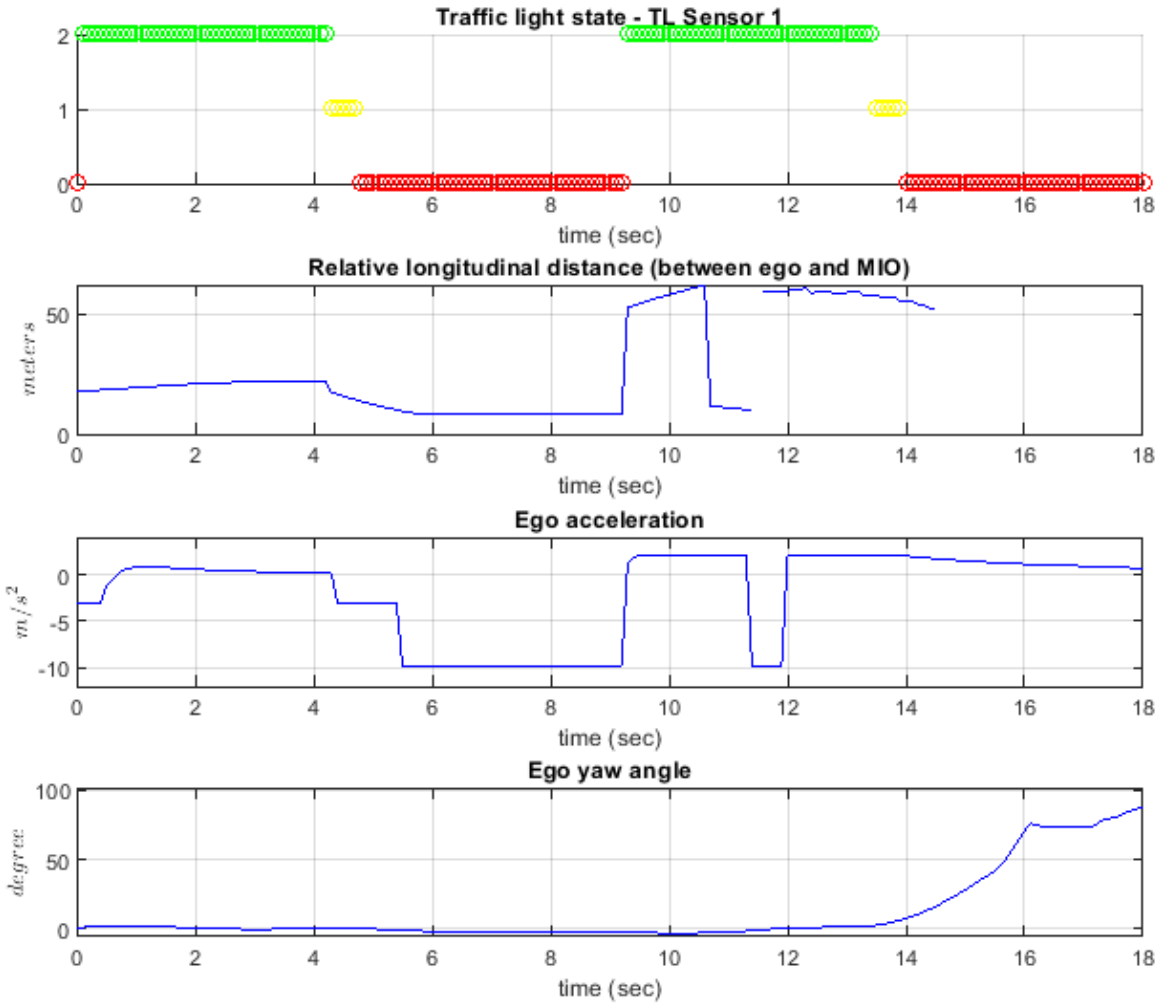
Configure the `TrafficLightNegotiationTestBench` model to use the `scenario_TLN_left_turn_with_cross_over_vehicle` scenario.

```
helperSLTrafficLightNegotiationSetup("scenario_TLN_left_turn_with_cross_over_vehicle");

% Simulate the model.
sim("TrafficLightNegotiationTestBench");
```



Plot the simulation results.

```
hFigResults = helperPlotTrafficLightNegotiationResults(logsout);
```



Examine the results.

- The **Traffic light state - TL Sensor 1** plot is same as the one from the previous simulation.
- The **Relative longitudinal distance** plot diverges from the previous simulation run from 10.5 seconds onward. Notice the sudden dip in the relative distance, in reaction to the detection of the cross-traffic vehicle.
- The **Ego acceleration** plot also closely follows the dip in the relative distance from 11.3 seconds onward. You can notice a hard-braking profile in response to the cross-traffic vehicle at the intersection.
- The **Ego yaw angle** plot shows that the ego vehicle initiates a left turn after 14 seconds, in response to the cross-traffic vehicle leaving the intersection.

Close the figure.

```
close(hFigResults);
```

Explore Other Scenarios

In the previous sections, you explored the system behavior for the `scenario_TLN_left_turn_with_lead_vehicle` and `scenario_TLN_left_turn_with_cross_over_vehicle` scenarios. Below is a list of scenarios that are compatible with `TrafficLightNegotiationTestBench`.

```
  scenario_TLN_straight
  scenario_TLN_straight_with_lead_vehicle
  scenario_TLN_left_turn
  scenario_TLN_left_turn_with_lead_vehicle
  scenario_TLN_left_turn_with_cross_over_vehicle [Default]
```

Use these additional scenarios to analyze `TrafficLightNegotiationTestBench` under different conditions. For example, while learning about the interactions between the traffic light decision logic and controls, it can be helpful to begin with a scenario that has an intersection with a traffic light but no vehicles. To configure the model and workspace for such a scenario, use this code:

```
helperSLTrafficLightNegotiationSetup("scenario_TLN_straight");
```

Enable the MPC update messages.

```
mpcverbosity('on');
```

Conclusion

In this example, you implemented decision logic for traffic light negotiation and tested it with a lane following controller in a closed-loop Simulink model.

See Also

More About

- “Highway Lane Change” on page 7-598
- “Highway Lane Following” on page 7-653

Design Lidar SLAM Algorithm Using Unreal Engine Simulation Environment

This example shows how to record synthetic lidar sensor data from a 3D simulation environment, and develop a simultaneous localization and mapping (SLAM) algorithm using the recorded data. The simulation environment uses the Unreal Engine® by Epic Games®.

Introduction

Automated Driving Toolbox™ integrates an Unreal Engine simulation environment in Simulink®. Simulink blocks related to this simulation environment can be found in the `drivingsim3d` library. These blocks provide the ability to:

- Select different scenes in the 3D simulation environment
- Place and move vehicles in the scene
- Attach and configure sensors on the vehicles
- Simulate sensor data based on the environment around the vehicle

This powerful simulation tool can be used to supplement real data when developing, testing, and verifying the performance of automated driving algorithms, making it possible to test scenarios that are difficult to reproduce in the real world.

In this example, you evaluate a lidar perception algorithm using synthetic lidar data generated from the simulation environment. The example walks you through the following steps:

- Record and visualize synthetic lidar sensor data from the simulation environment.
- Develop a perception algorithm to build a map using SLAM in MATLAB®.

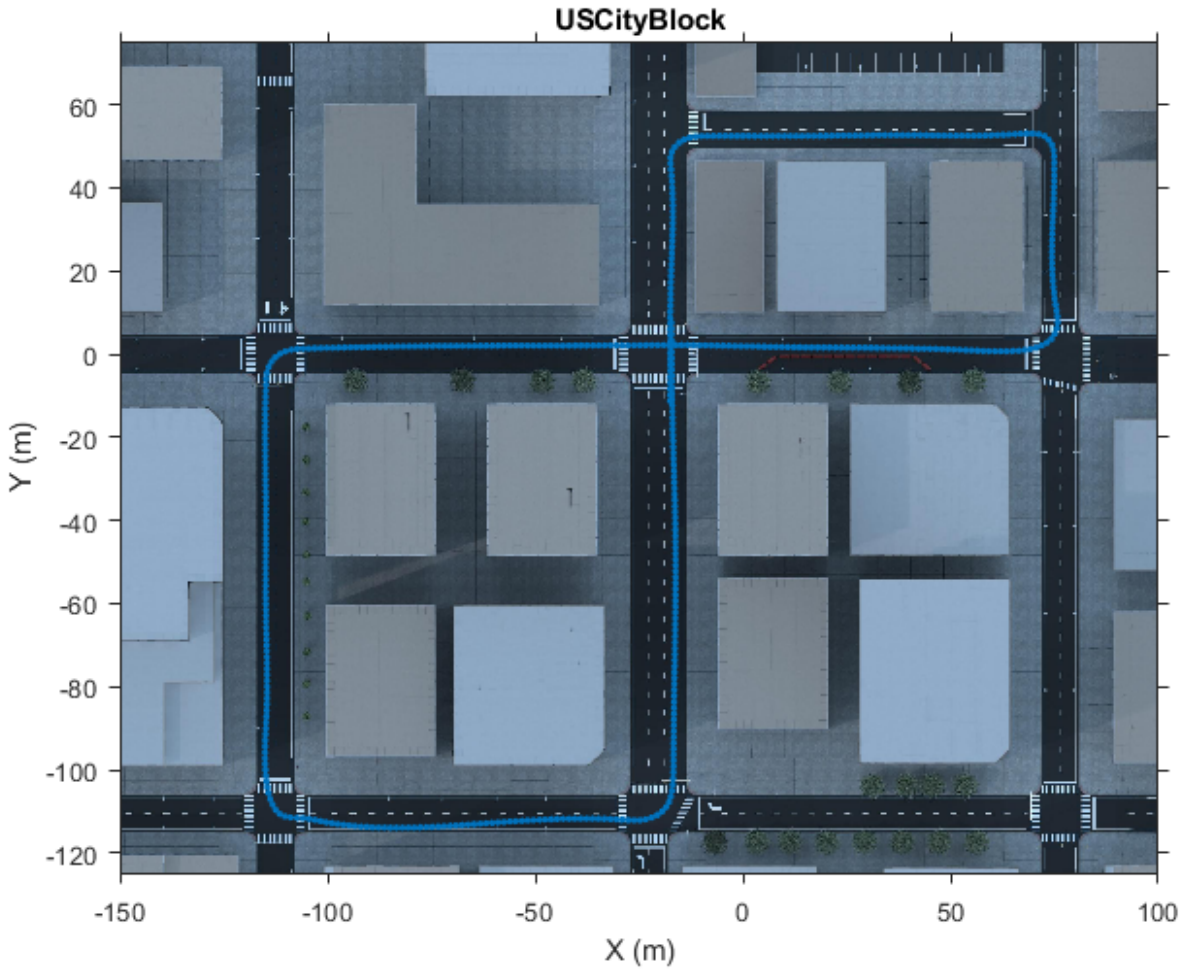
Set Up Scenario in Simulation Environment

First, set up a scenario in the simulation environment that can be used to test the perception algorithm. Use a scene depicting a typical city block with a single vehicle that is the vehicle under test. You can use this scene to test the performance of the algorithm in an urban road setting.

Next, select a trajectory for the vehicle to follow in the scene. The “Select Waypoints for Unreal Engine Simulation” on page 7-626 example describes how to interactively select a sequence of waypoints from a scene and generate a vehicle trajectory. This example uses a recorded drive segment obtained using the `helperSelectSceneWaypoints` function, as described in the waypoint selection example.

```
% Load reference path for recorded drive segment
xData   = load('refPosesX.mat');
yData   = load('refPosesY.mat');
yawData = load('refPosesT.mat');

% Set up workspace variables used by model
refPosesX = xData.refPosesX;
refPosesY = yData.refPosesY;
refPosesT = yawData.refPosesT;

% Display path on scene image
sceneName = 'USCityBlock';
hScene = figure;
```

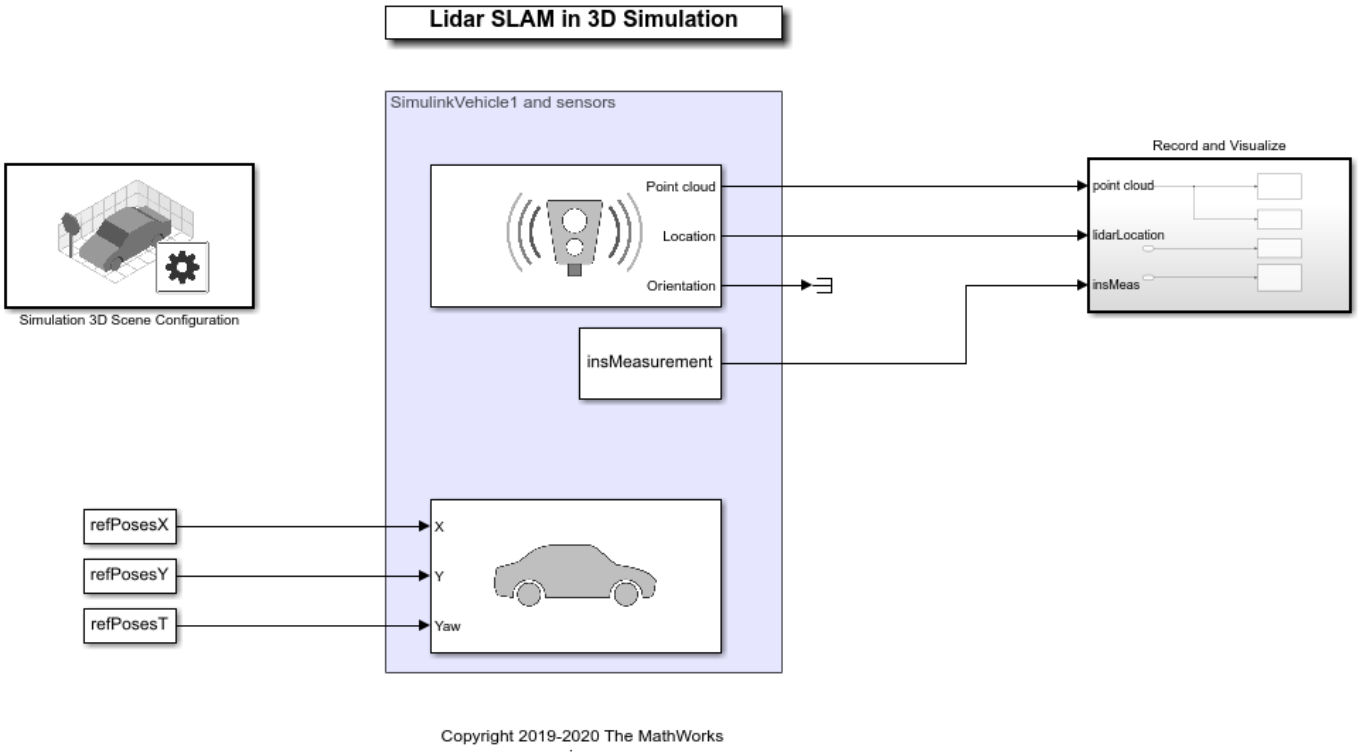
```
helperShowSceneImage(sceneName);
hold on
scatter(refPosesX(:,2), refPosesY(:,2), 7, 'filled')

% Adjust axes limits
xlim([-150 100])
ylim([-125 75])
```



The `LidarSLAMIn3DSimulation` Simulink model is configured with the US City Block scene using the Simulation 3D Scene Configuration block. The model places a vehicle on the scene using the Simulation 3D Vehicle with Ground Following block. A lidar sensor is attached to the vehicle using the Simulation 3D Lidar block. In the block dialog box, use the **Mounting** tab to adjust the placement of the sensor. Use the **Parameters** tab to configure properties of the sensor to simulate different lidar

sensors. In this example, the lidar is mounted on the center of the roof. The lidar sensor is configured to model a typical Velodyne® HDL-32E sensor.

```
close(hScene)

if ~ispc
    error(['3D Simulation is only supported on Microsoft', char(174), ' Windows', char(174), '.'])
end

% Open the model
modelName = 'LidarSLAMIn3DSimulation';
open_system(modelName);
snapnow;
```



The model records and visualizes the synthetic lidar data. The recorded data is available through the simulation output, and can be used for prototyping your algorithm in MATLAB. Additionally, the model uses a From Workspace (Simulink) block to load simulated measurements from an Inertial Navigation Sensor (INS). The INS data was obtained using the `insSensor (Sensor Fusion and Tracking Toolbox)` object from Sensor Fusion and Tracking Toolbox™, and saved in a MAT-file.

The rest of the example follows these steps:

1. Simulate the model to record synthetic lidar data generated by the sensor and save it to the workspace.
2. Use the sensor data saved to the workspace to develop a perception algorithm in MATLAB. The perception algorithm builds a map of the surroundings using SLAM.
3. Visualize the results of the built map.

Record and Visualize Synthetic Lidar Sensor Data

The **Record and Visualize** subsystem records the synthetic lidar data to the workspace using a To Workspace (Simulink) block. The **Visualize Point Cloud** MATLAB Function block uses a `pcplayer` (Computer Vision Toolbox) object to visualize the streaming point clouds. The **Visualize INS Path** MATLAB Function block visualizes the streaming INS data.

Simulate the model. The streaming point cloud display shows the synthetic lidar sensor data. The scene display shows the synthetic INS sensor data. Once the model has completed simulation, the `simOut` variable holds a structure with variables written to the workspace. The `helperGetPointCloud` function extracts the sensor data into an array of `pointCloud` (Computer Vision Toolbox) objects. The `pointCloud` object is the fundamental data structure used to hold lidar data and perform point cloud processing in MATLAB. Additionally, INS data is loaded from a MAT-file, which will later be used to develop the perception algorithm. The INS data was obtained using the `insSensor (Sensor Fusion and Tracking Toolbox)` object from the Sensor Fusion and Tracking Toolbox™. The INS data has been processed to contain [x, y, theta] poses in world coordinates.

```
% Update simulation stop time to end when reference path is completed
simStopTime = refPosesX(end,1);
set_param(gcs, 'StopTime', num2str(simStopTime));

% Load INS data from MAT-file
data = load('insMeasurement.mat');
insData = data.insMeasurement.signals.values;

% Run the simulation
simOut = sim(modelName);

% Create a pointCloud array from the recorded data
ptCloudArr = helperGetPointCloud(simOut);
```

Use Recorded Data to Develop Perception Algorithm

The synthetic lidar sensor data can be used to develop, experiment with, and verify a perception algorithm in different scenarios. This example uses an algorithm to build a 3D map of the environment from streaming lidar data. Such an algorithm is a building block for applications like localization. It can also be used to create high-definition (HD) maps for geographic regions that can then be used for online localization. The map building algorithm is encapsulated in the `helperLidarMapBuilder` class. This class uses point cloud and lidar processing capabilities in MATLAB. For more details, see “Lidar and Point Cloud Processing” (Computer Vision Toolbox).

The `helperLidarMapBuilder` class takes incoming point clouds from a lidar sensor and progressively builds a map using the following steps:

1. **Preprocess point cloud:** Preprocess each incoming point cloud to remove the ground plane and ego vehicle.
2. **Register point clouds:** Register the incoming point cloud with the last point cloud using a normal distribution transform (NDT) registration algorithm. The `pcregisterndt` (Computer Vision Toolbox) function performs the registration. To improve accuracy and efficiency of registration, `pcdownsample` (Computer Vision Toolbox) is used to downsample the point cloud prior to registration. An initial transform estimate can substantially improve registration performance. In this example, INS measurements are used to accomplish this.
3. **Register point clouds:** Use the estimated transformation obtained from registration to transform the incoming point cloud to the frame of reference of the map.

4 **Update view set:** Add the incoming point cloud and the estimated absolute pose as a view in a `pcviewset` (Computer Vision Toolbox) object. Add a connection between the current and previous view with the relative transformation between them.

The `updateMap` method of the `helperLidarMapBuilder` class accomplishes these steps. The `helperEstimateRelativeTransformationFromINS` function computes an initial estimate for registration from simulated INS sensor readings.

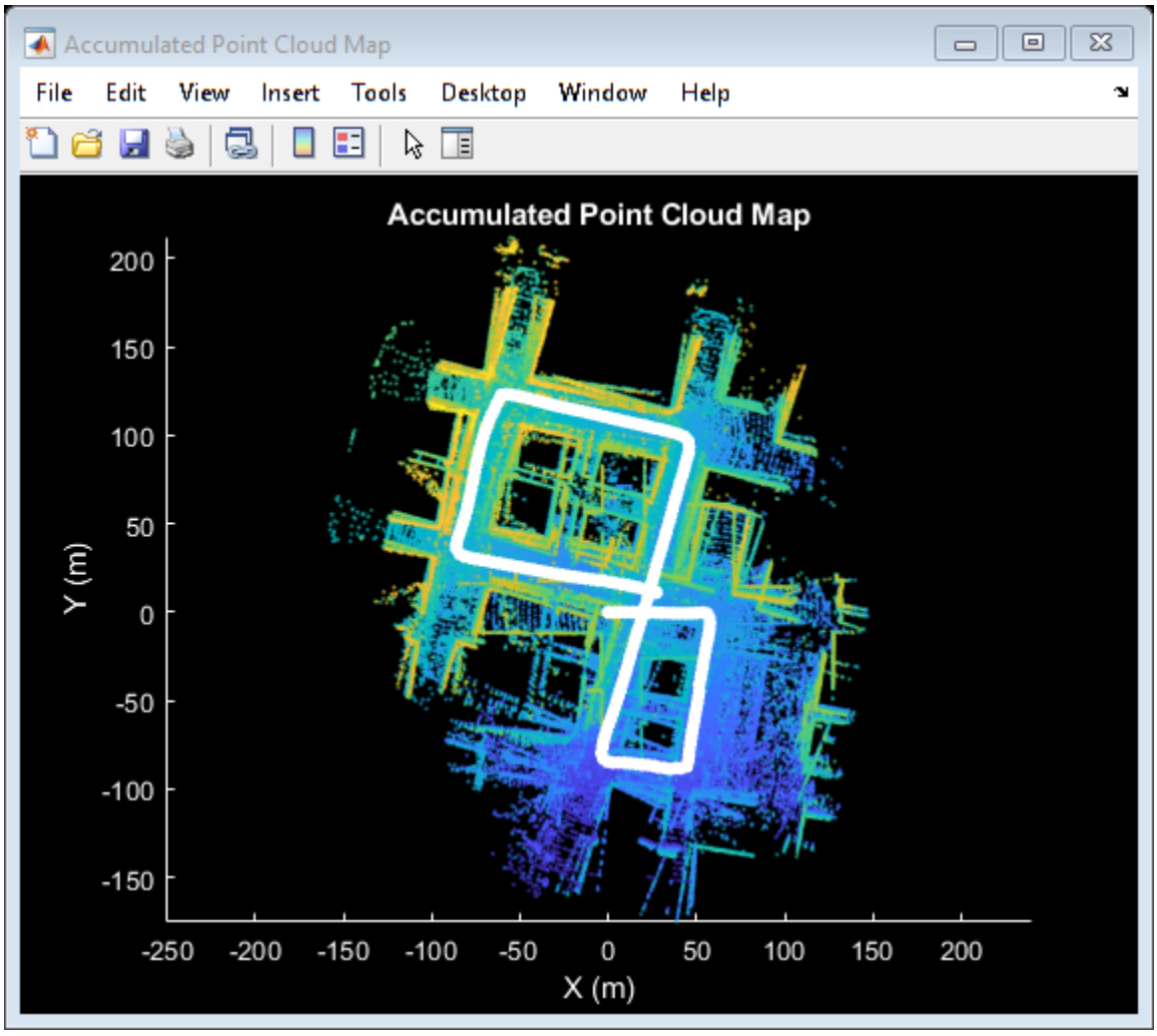
Such an algorithm is susceptible to drift while accumulating a map over long sequences. To reduce the drift, it is typical to detect loop closures and use graph SLAM to correct the drift. See “Build a Map from Lidar Data Using SLAM” on page 7-559 example for a detailed treatment. The `configureLoopDetector` method of the `helperLidarMapBuilder` class configures loop closure detection. Once it is configured, loop closure detection takes place each time `updateMap` is invoked, using the following functions and classes:

- `pcviewset`: Manages data associated with point cloud odometry like point clouds, poses and connections.
- `scanContextDescriptor`: Extracts scan context descriptors from a point cloud. Scan context is a 2D global feature descriptor that is used for loop closure detection.
- `scanContextDistance`: Computes the distance between scan context descriptors.
- `helperFeatureSearcher`: Helper class that stores computed feature descriptors and searches for the closest feature matches. The closest feature matches are loop closure candidates. Feature distance is computed using `scanContextDistance`.
- `helperLoopClosureDetector`: Detects loop closures. Loop closure candidates are identified by computing scan context features for each incoming point cloud and finding the closest feature matches. Then, point cloud registration is used to accept or reject loop closure candidates.

```
% Set the random seed for example reproducibility
rng(0);

% Create a lidar map builder
mapBuilder = helperLidarMapBuilder('DownsamplePercent', 0.25, ...
    'RegistrationGridStep', 3.5, 'Verbose', true);

% Configure the map builder to detect loop closures
configureLoopDetector(mapBuilder, ...
    'LoopConfirmationRMSE', 2.8, ...
    'MatchThreshold',       0.15, ...
    'DistanceThreshold',    0.15);

% Loop through the point cloud array and progressively build a map
skipFrames = 5;
numFrames  = numel(ptCloudArr);
exitLoop   = false;

prevInsMeas = insData(1, :);
for n = 1 : skipFrames : numFrames

    insMeas = insData(n, :);

    % Estimate initial transformation using INS
    initTform = helperEstimateRelativeTransformationFromINS(insMeas, prevInsMeas);

    % Update map with new lidar frame
```

```
    updateMap(mapBuilder, ptCloudArr(n), initTform);

    % Update top-view display
    isDisplayOpen = updateDisplay(mapBuilder, exitLoop);

    % Check and exit if needed
    exitLoop = ~isDisplayOpen;

    prevInsMeas = insMeas;
end

snapnow;

% Close display
closeDisplay = true;
updateDisplay(mapBuilder, closeDisplay);
```

```
Loop closure candidate found between view Id 210 and 2 with RMSE 2.115456...
Accepted
Loop closure candidate found between view Id 211 and 3 with RMSE 2.378373...
Accepted
```



The accumulated drift progressively increases over time resulting in an unusable map.

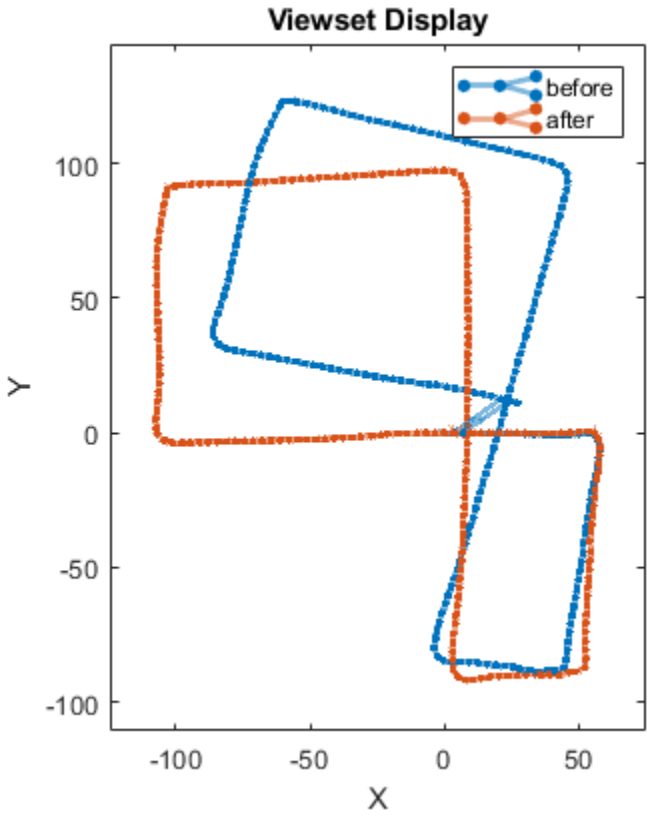
Once sufficient loop closures are detected, the accumulated drift can be corrected using pose graph optimization. This is accomplished by the `optimizeMapPoses` method of the `helperLidarMapBuilder` class, which uses `createPoseGraph` (Computer Vision Toolbox) to create a pose graph and `optimizePoseGraph` to optimize the pose graph.

After the pose graph has been optimized, rebuild the map using the updated poses. This is accomplished by the `rebuildMap` method of `helperLidarMapBuilder` using `pcalign`.

Use `optimizeMapPoses` and `rebuildMap` to correct for the drift and rebuild the map. Visualize the view set before and after pose graph optimization.

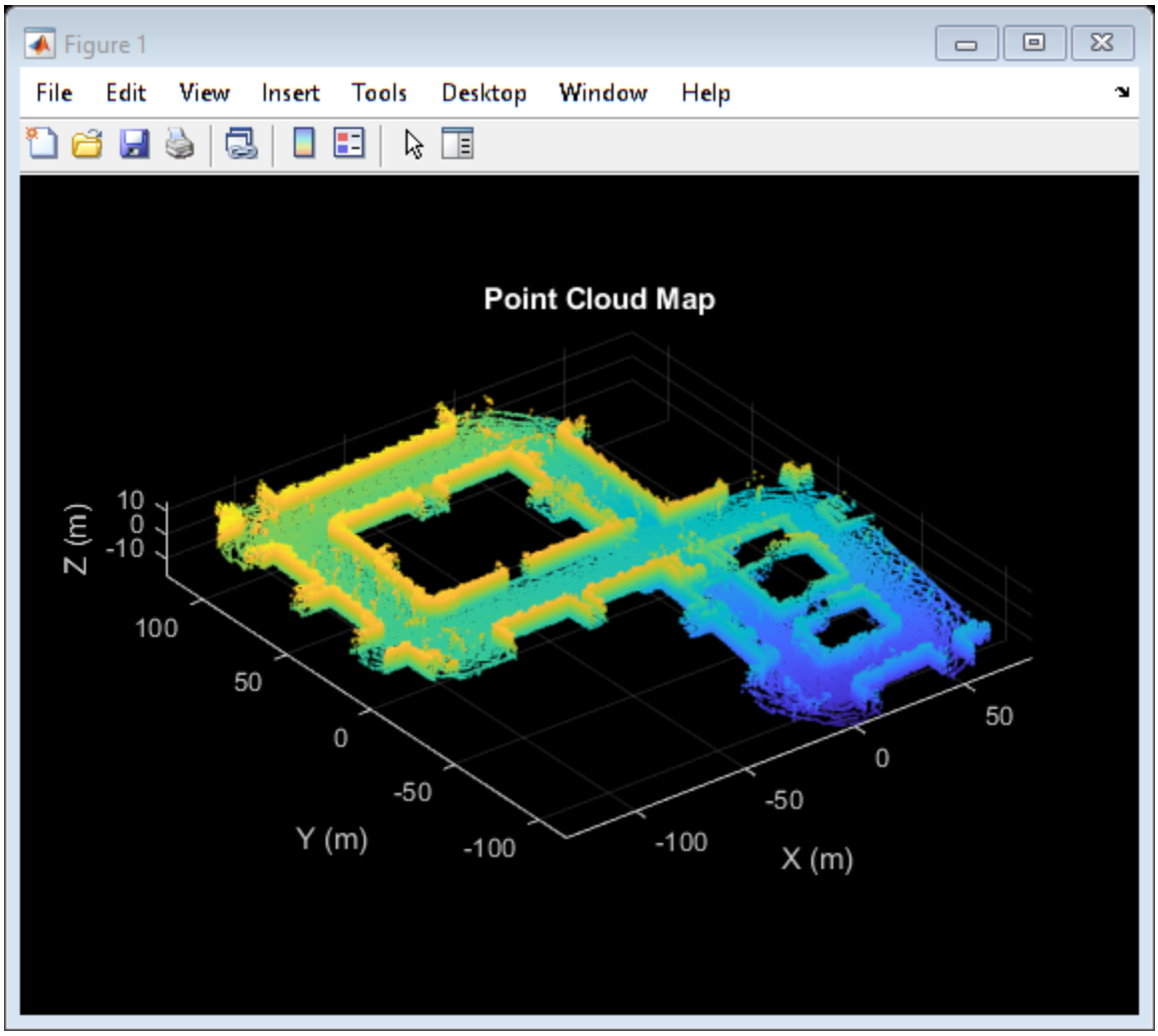
```
% Visualize viewset before pose graph optimization
hFigViewset = figure;
hG = plot(mapBuilder.ViewSet);
view(hG.Parent, 2);
title('Viewset Display')

% Optimize pose graph and rebuild map
optimizeMapPoses(mapBuilder);
rebuildMap(mapBuilder);

% Overlay viewset after pose graph optimization
hold(hG.Parent, 'on');
plot(mapBuilder.ViewSet);
hold(hG.Parent, 'off');

legend(hG.Parent, 'before', 'after')
```

```
Optimizing pose graph...done
Rebuilding map...done
```



Visualize the accumulated point cloud map computed using the recorded data.

```
close(hFigViewset)

hFigMap = figure;
pcshow(mapBuilder.Map)

% Customize axes labels and title
xlabel('X (m)')
ylabel('Y (m)')
zlabel('Z (m)')
title('Point Cloud Map')

helperMakeFigurePublishFriendly(hFigMap);
```



By changing the scene, placing more vehicles in the scene, or updating the sensor mounting and parameters, the perception algorithm can be stress-tested under different scenarios. This approach can be used to increase coverage for scenarios that are difficult to reproduce in the real world.

```
% Close windows
close(hFigMap)
close_system(modelName)
```

Supporting Functions

helperGetPointCloud Extract an array of `pointCloud` objects.

```
function ptCloudArr = helperGetPointCloud(simOut)

% Extract signal
ptCloudData = simOut.ptCloudData.signals.values;

% Create a pointCloud array
ptCloudArr = pointCloud(ptCloudData(:,:,:,1));

for n = 2 : size(ptCloudData,4)
```

```
    ptCloudArr(end+1) = pointCloud(ptCloudData(:,:,:,n));  %#ok<AGROW>
end
end
```

helperMakeFigurePublishFriendly Adjust figure so that screenshot captured by publish is correct.

```
function helperMakeFigurePublishFriendly(hFig)

if ~isempty(hFig) && isvalid(hFig)
    hFig.HandleVisibility = 'callback';
end
end
```

Additional supporting functions or classes used in the example are included below.

helperLidarMapBuilder progressively builds a lidar map using point cloud scans. Each point cloud is processed to remove the ground plane and the ego vehicle, and registered against the previous point cloud. A point cloud map is then progressively built by aligning and merging the point clouds.

helperEstimateRelativeTransformationFromINS estimates a relative transformation from INS data.

helperFeatureSearcher creates an object that can be used to search for closest feature matches.

helperLoopClosureDetector creates an object that can be used to detect loop closures using scan context feature descriptors.

helperShowSceneImage displays top-view image of the Unreal scene.

helperUpdatePolyline updates a polyline position used in conjunction with helperShowSceneImage.

See Also

Functions

`createPoseGraph` | `pcregisterndt` | `pcshow`

Objects

`pcviewset` | `pointCloud` | `rigid3d`

Blocks

Simulation 3D Lidar | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following

More About

- “Select Waypoints for Unreal Engine Simulation” on page 7-626
- “Build a Map from Lidar Data” on page 7-539
- “Build a Map from Lidar Data Using SLAM” on page 7-559
- “Lidar Localization with Unreal Engine Simulation” on page 7-701
- “Develop Visual SLAM Algorithm Using Unreal Engine Simulation” on page 7-714

Lidar Localization with Unreal Engine Simulation

This example shows how to develop and evaluate a lidar localization algorithm using synthetic lidar data from the Unreal Engine® simulation environment.

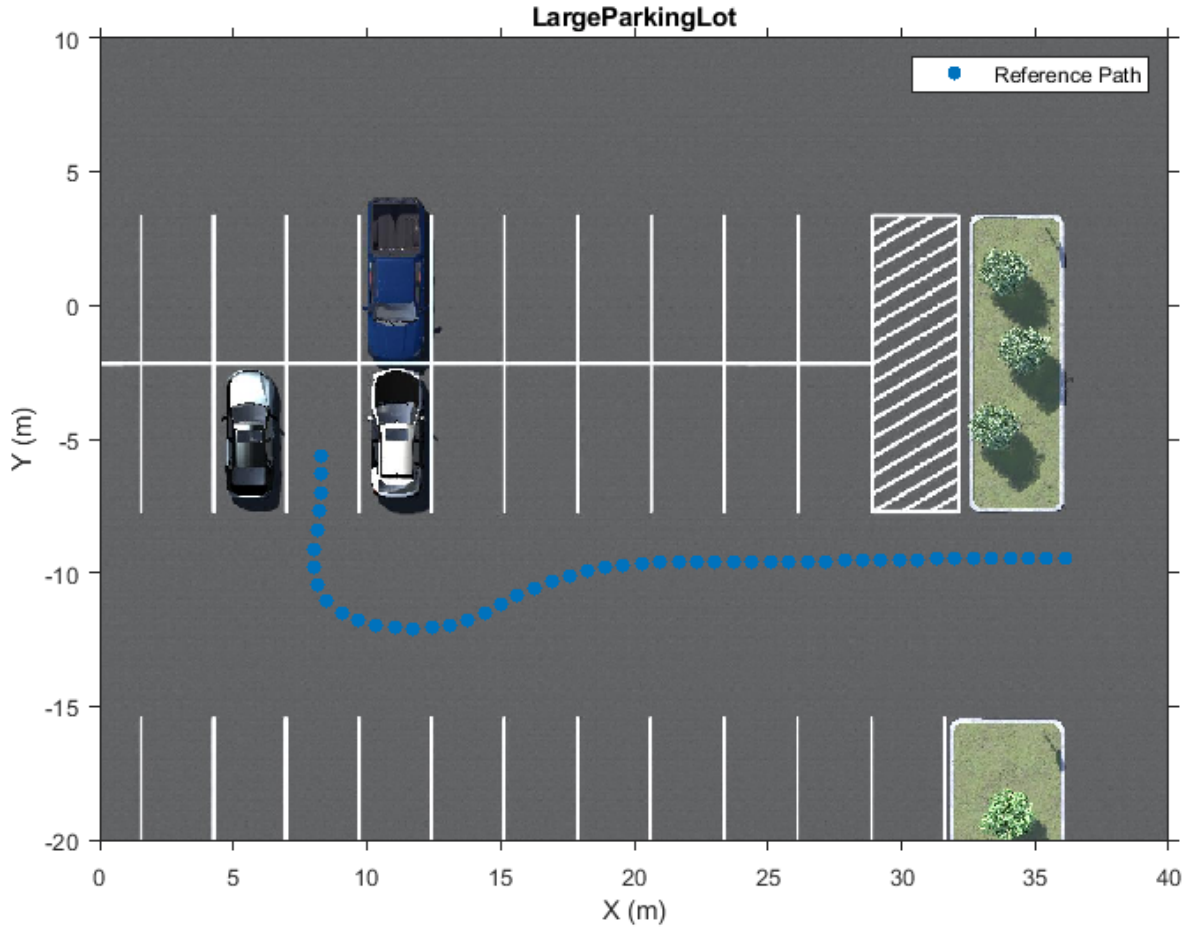
Developing a localization algorithm and evaluating its performance in varying conditions is a challenging task. One of the biggest challenges is obtaining ground truth. Although you can capture ground truth using expensive, high-precision inertial navigation systems (INS), virtual simulation is a cost-effective alternative. The use of simulation enables testing under a variety of scenarios and sensor configurations. It also enables a rapid development iteration, and provides precise ground truth.

This example uses the Unreal Engine simulation environment from Epic Games® to develop and evaluate a lidar localization algorithm from a known initial pose in a parking scenario.

Set Up Scenario in Simulation Environment

Parking a vehicle into a parking spot is a challenging maneuver that relies on accurate localization. Use the prebuilt Large Parking Lot scene to create such a scenario. The “Select Waypoints for Unreal Engine Simulation” on page 7-626 example describes how to interactively select a sequence of waypoints from a scene and how to generate a reference vehicle trajectory. This example uses a recorded reference trajectory obtained using the approach described in the linked example. First, visualize the reference path on a 2-D bird's-eye view of the scene.

```
% Load reference path
data = load('ReferencePathForward.mat');

refPosesX = data.ReferencePathForward.refPosesX;
refPosesY = data.ReferencePathForward.refPosesY;
refPosesT = data.ReferencePathForward.refPosesT;

sceneName = 'LargeParkingLot';
hScene = figure;
helperShowSceneImage(sceneName);
hold on
scatter(refPosesX(:,2), refPosesY(:,2), [], 'filled', 'DisplayName', ...
    'Reference Path');
xlim([0 40])
ylim([-20 10])
legend
hold off
```

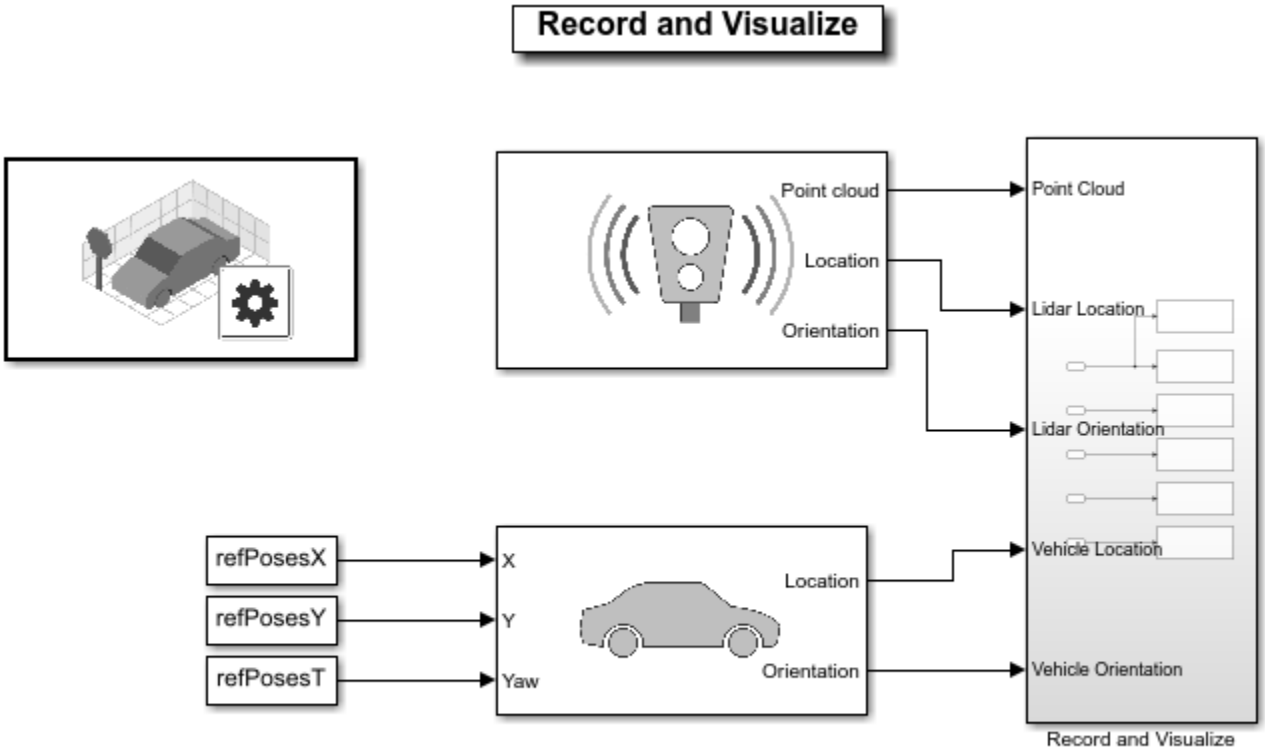


Record and Visualize Sensor Data

Set up a simple model with a hatchback vehicle moving along the specified reference path by using the Simulation 3D Vehicle with Ground Following block. Mount a lidar on the roof center of a vehicle using the Simulation 3D Lidar block. Record and visualize the sensor data. The recorded data is used to develop a localization algorithm.

```
close(hScene)

if ~ispc
    error("Unreal Engine Simulation is supported only on Microsoft" ...
        + char(174) + " Windows" + char(174) + ".");
end
```

```
% Open model
modelName = 'recordAndVisualize';
open_system(modelName);
snapnow;

% Run simulation
simOut = sim(modelName);
```



Copyright 2020 The MathWorks Inc.

The recorded sensor data is returned in the `simOut` variable.

Develop Algorithm Using Recorded Data

In this example, you develop an algorithm based on point cloud registration. Point cloud registration is a common localization technique that estimates the relative motion between two point clouds to derive localization data. Accumulating relative motion like this over long sequences can lead to drift, which can be corrected using loop closure detection and pose graph optimization, as shown in the “Build a Map from Lidar Data Using SLAM” on page 7-559 example. Since this example uses a short reference path, loop closure detection is omitted.

Extract the lidar sensor data and ground truth location and orientation provided by the Simulation 3D Lidar block. The ground truth location and orientation are provided in the world (scene) coordinate system. Extract the known initial pose from the ground truth data by using the `helperPoseToRigidTransform` function.

```
close_system(modelName);
```

```
% Extract lidar sensor data
ptCloudArr = helperGetPointClouds(simOut);

% Extract ground truth
[lidarLocation, lidarOrientation, simTimes] = helperGetLocalizerGroundTruth(simOut);

% Compute initial pose
initPose = [lidarLocation(1, :) lidarOrientation(1, :)];
initTform = helperPoseToRigidTransform(initPose);
```

Develop a lidar localization algorithm by using the extracted sensor data. Use a `pcviewset (Computer Vision Toolbox)` object to process and store odometry data. `pcviewset` organizes odometry data into a set of views, and the associated connections between views, where:

- Each view has an absolute pose describing the rigid transformation to some fixed reference frame.
- Each connection has a relative pose describing the rigid transformation between the two connecting views.

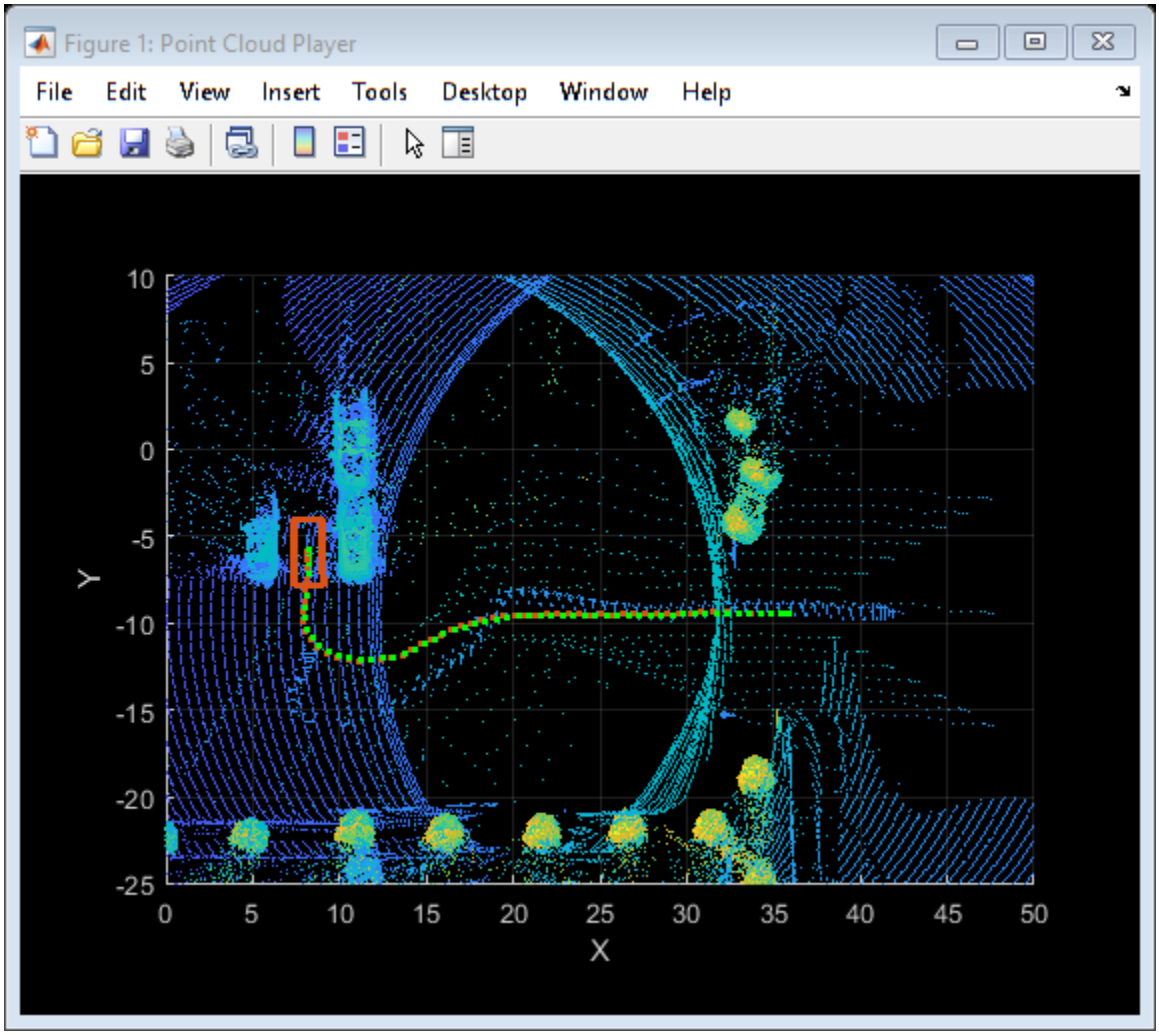
The localization estimate is maintained in the form of the absolute poses for each view with respect to the scene reference frame.

Use a `pcplayer (Computer Vision Toolbox)` object to display streaming point cloud data in the loop as it is registered. Transform the viewing angle to a top view. The orange cuboid and path show the localization position estimated by the algorithm. The green path shows the ground truth.

```
% Create a view set
vSet = pcviewset;

absPose = initTform;
relPose = rigid3d;
viewId  = 1;

% Define rigid transformation between the lidar sensor mounting position
% and the vehicle reference point.
lidarToVehicleTform = helperPoseToRigidTransform(single([0 0 -1.57 0 0 0]));

% Process the point cloud frame
ptCloud = helperProcessPointCloud(ptCloudArr(1));

% Initialize accumulated point cloud map
ptCloudAccum = pctransform(ptCloud, absPose);

% Add first view to the view set
vSet = addView(vSet, viewId, absPose, 'PointCloud', ptCloud);

% Configure display
xlimits = [  0  50];
ylimits = [-25  10];
zlimits = [-30  30];
player = pcplayer(xlimits, ylimits, zlimits);
estimatePathHandle = [];
truthPathHandle    = [];

% Specify vehicle dimensions
centerToFront = 1.104;
centerToRear  = 1.343;
frontOverhang = 0.828;
```

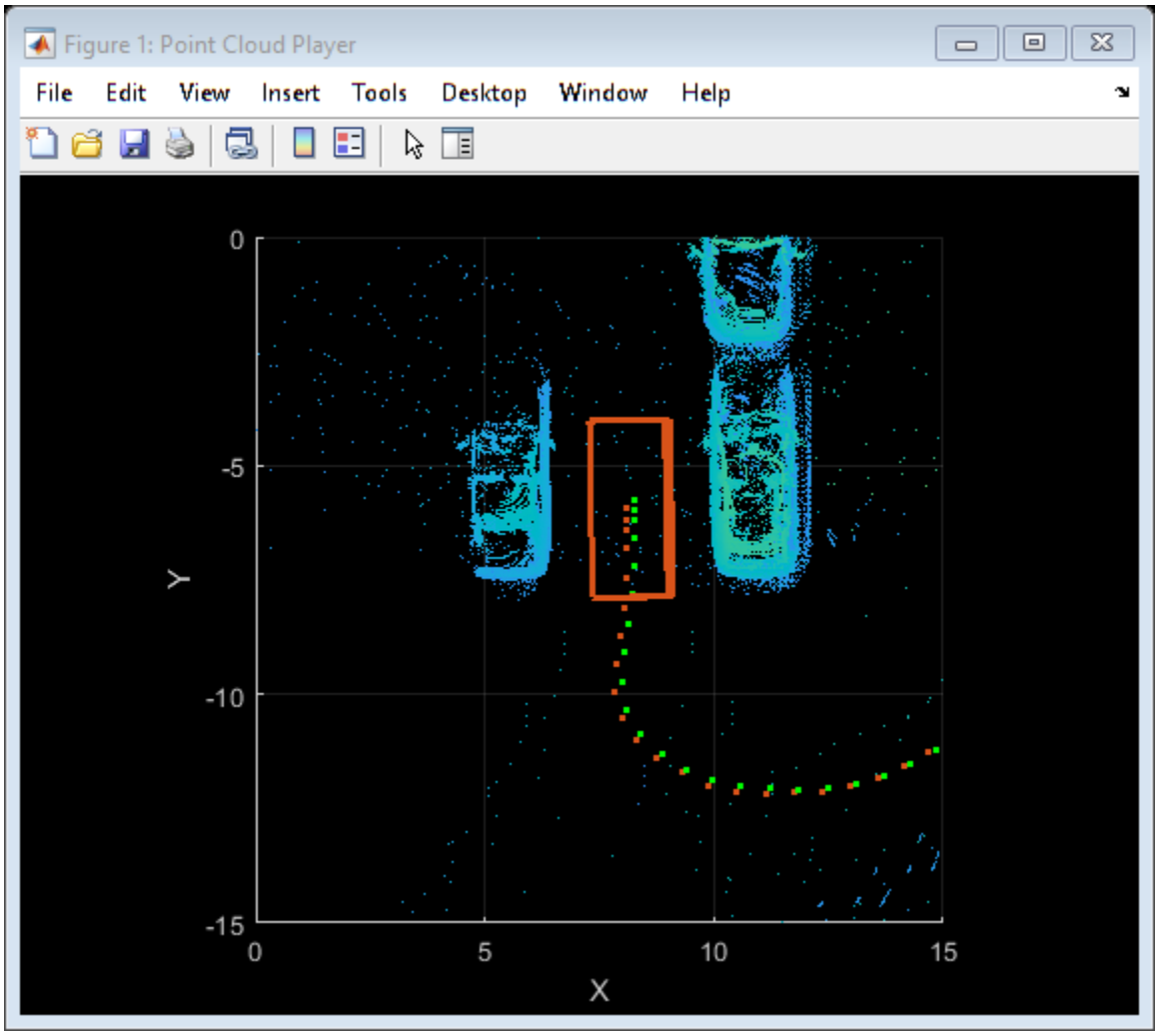
```
rearOverhang  = 0.589;
vehicleWidth  = 1.653;
vehicleHeight = 1.513;
vehicleLength = centerToFront + centerToRear + frontOverhang + rearOverhang;
hatchbackDims = vehicleDimensions(vehicleLength,vehicleWidth,vehicleHeight, ...
'FrontOverhang',frontOverhang,'RearOverhang',rearOverhang);

vehicleDims   = [hatchbackDims.Length, hatchbackDims.Width, hatchbackDims.Height];
vehicleColor  = [0.85 0.325 0.098];

% Initialize parameters
skipFrames    = 5;      % Number of frames to skip to accumulate sufficient motion
prevViewId    = viewId;
prevPtCloud   = ptCloud;

% Loop over lidar sensor frames and localize
for viewId = 6 : skipFrames : numel(ptCloudArr)
    % Process frame
    ptCloud = helperProcessPointCloud(ptCloudArr(viewId));

    % Register current frame to previous frame
    relPose = pcregistericp(ptCloud, prevPtCloud, 'MaxIterations', 40, ...
        'Metric', 'pointToPlane');

    % Since motion is restricted to a 2-D plane, discard motion along Z to
    % prevent accumulation of noise.
    relPose.Translation(3) = 0;

    % Update absolute pose
    height = absPose.Translation(3);
    absPose = rigid3d( relPose.T * absPose.T );
    absPose.Translation(3) = height;

    % Add new view and connection to previous view
    vSet = addView(vSet, viewId, absPose, 'PointCloud', ptCloud);
    vSet = addConnection(vSet, prevViewId, viewId, relPose);

    % Accumulated point cloud map
    ptCloudAccum = pccat([ptCloudAccum, pctransform(ptCloud, absPose)]);

    % Compute ground truth and estimate position
    localizationEstimatePos = absPose.Translation;
    localizationTruthPos    = lidarLocation(viewId, :);

    % Update accumulated point cloud map
    view(player, ptCloudAccum);

    % Set viewing angle to top view
    view(player.Axes, 2);

    % Convert current absolute pose of sensor to vehicle frame
    absVehiclePose = rigid3d( lidarToVehicleTform.T * absPose.T );

    % Draw vehicle at current absolute pose
    helperDrawVehicle(player.Axes, absVehiclePose, vehicleDims, 'Color', vehicleColor);

    % Draw localization estimate and ground truth points
    helperDrawLocalization(player.Axes, ...
```

```
        localizationEstimatePos, estimatePathHandle, vehicleColor, ...
        localizationTruthPos, truthPathHandle, [0 1 0]);

    prevPtCloud = ptCloud;
    prevViewId  = viewId;
end
```



Zoom in to the tail of the trajectory to examine the localization estimate compared to the ground truth.

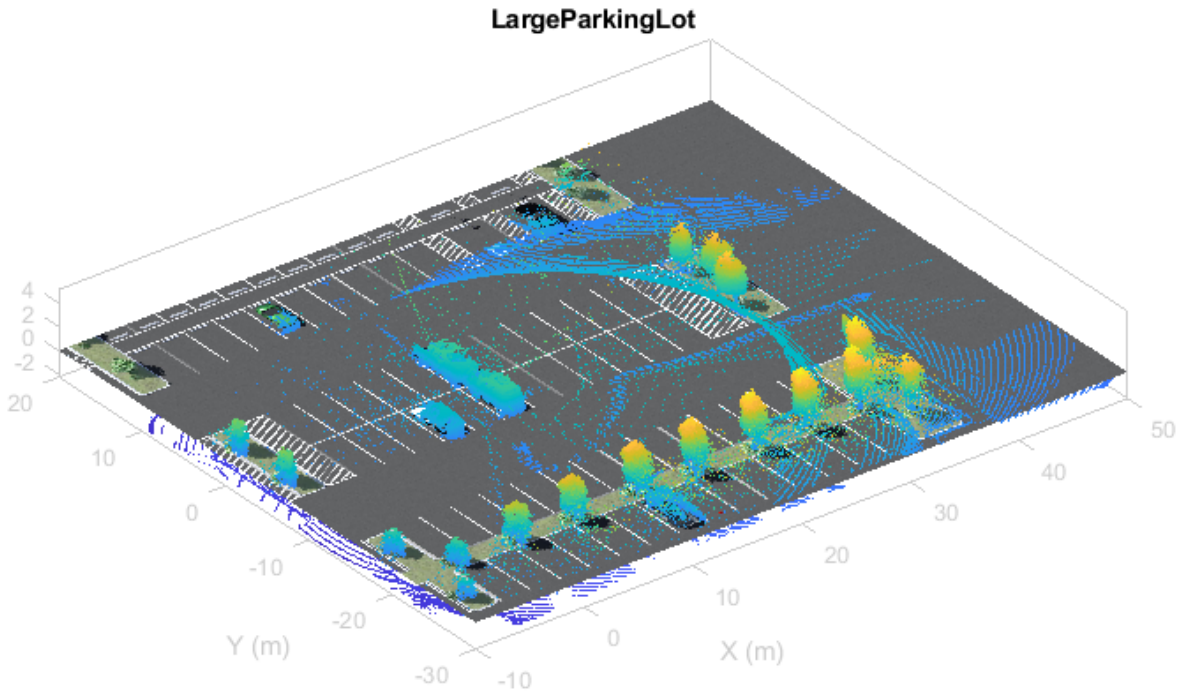
```
xlim(player.Axes, [0 15]);
ylim(player.Axes, [-15 0]);
zlim(player.Axes, [0 15]);

snapnow;

% Close player
hide(player);
```



A useful outcome of a localization algorithm based on point cloud registration is a map of the traversed environment. You can obtain this map by combining all the point clouds to a common reference frame. The `pccat (Computer Vision Toolbox)` function is used in each iteration of the loop above, along with `pctransform (Computer Vision Toolbox)`, to incrementally combine the registered point clouds. Alternatively, you can use the `pcalign (Computer Vision Toolbox)` function to align all point clouds to the common reference frame in one shot at the end.

Superimpose the point cloud map on the top-view image of the scene to visually examine how closely it resembles features in the scene.

```
hMapOnScene = helperSuperimposeMapOnSceneImage(sceneName, ptCloudAccum);

snapnow;

% Close the figure
close(hMapOnScene);
```



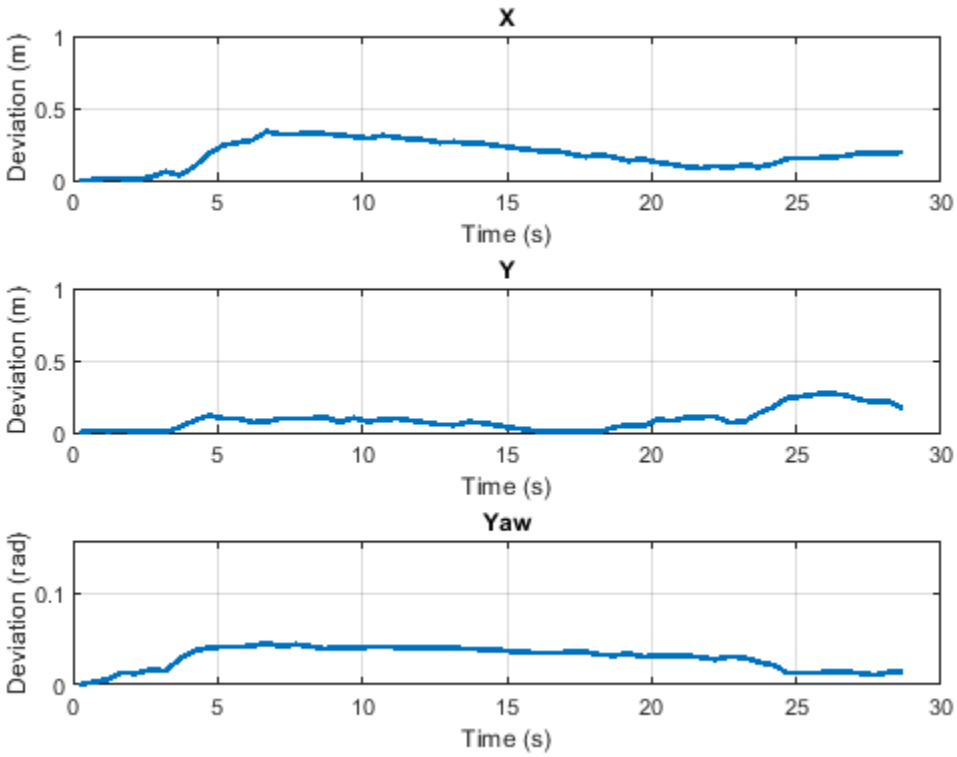
The localization algorithm described above is encapsulated in the `helperLidarRegistrationLocalizer` helper class. This class can be used as a framework to develop a localization pipeline using point cloud registration.

- Use the `'ProcessFcnHandle'` and `'ProcessFcnArguments'` name-value pair arguments to configure how point clouds are processed prior to registration.
- Use the `'RegisterFcnHandle'` and `'RegisterFcnArguments'` name-value pair arguments to configure how point clouds are registered.

Evaluate Localization Accuracy

To quantify the efficacy of localization, measure the deviation in translation and rotation estimates compared to ground truth. Since the vehicle is moving on flat ground, this example is concerned only with motion in the *X*-*Y* plane.

```
hFigMetrics = helperDisplayMetrics(vSet, lidarLocation, lidarOrientation, simTimes);
```



Simulate in the Loop

Although metrics like deviation in translation and rotation estimates are necessary, the performance of a localization system can have downstream impacts. For example, changes to the accuracy or performance of a localization system can affect the vehicle controller, necessitating the retuning of controller gains. Therefore, it is crucial to have a closed-loop verification framework that incorporates downstream components. The `localizeAndControlUsingLidar` model demonstrates this framework by incorporating a localization algorithm, vehicle controller and suitable vehicle model.

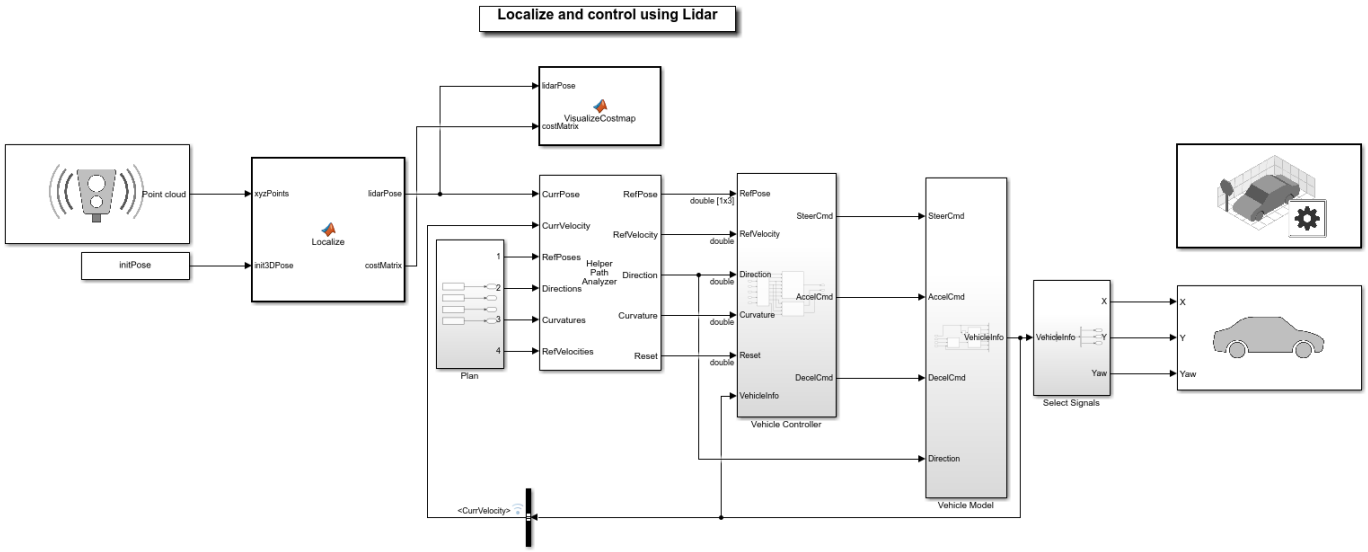
The model has these main components:

- The Localize block is a MATLAB Function block that encapsulates the localization algorithm - implemented using the `helperLidarRegistrationLocalizer` class. This block takes the lidar point cloud generated by the Simulation 3D Lidar block and the initial known pose as inputs and produces a localization estimate. The estimate is returned as (x, y, θ), which represents the 2-D pose of the lidar in the map reference frame.
- The Plan subsystem loads a preplanned trajectory from the workspace using the `refPoses`, `directions`, `curvatures` and `velocities` workspace variables. The **Path Smoother Spline**

block was used to compute the `refPoses`, `directions` and `curvatures` variables. The **Velocity Profiler** block computed the `velocities` variable.

- The Helper Path Analyzer block uses the reference trajectory and the current pose to feed the appropriate reference signal to the vehicle controller.
- The Vehicle Controller subsystem controls the steering and velocity of the vehicle by using a lateral and longitudinal controller to produce a steering and acceleration or deceleration command. The **Lateral Controller Stanley** and **Longitudinal Controller Stanley** blocks are used to implement this. These commands are fed to a vehicle model to simulate the dynamics of the vehicle in the simulation environment using the **Vehicle Body 3DOF** block.

```
close(hFigMetrics);

% Load workspace variables for preplanned trajectory
refPoses   = data.ReferencePathForward.Trajectory.refPoses;
directions = data.ReferencePathForward.Trajectory.directions;
curvatures = data.ReferencePathForward.Trajectory.curvatures;
velocities = data.ReferencePathForward.Trajectory.velocities;
startPose  = refPoses(1, :);

% Open model
modelName = 'localizeAndControlUsingLidar';
open_system(modelName);
snapnow;

% Run simulation
sim(modelName);

close_system(modelName);
```



Copyright 2020 The MathWorks Inc.

With this setup, it is possible to rapidly iterate over different scenarios, sensor configurations, or reference trajectories and refine the localization algorithm before moving to real-world testing.

- To select a different scenario, use the Simulation 3D Scene Configuration block. Choose from the existing prebuilt scenes or create a custom scene in the Unreal® Editor.

- To create a different reference trajectory, use the `helperSelectSceneWaypoints` tool, as shown in the “Select Waypoints for Unreal Engine Simulation” on page 7-626 example.
- To alter the sensor configuration use the Simulation 3D Lidar block. The **Mounting** tab provides options for specifying different sensor mounting placements. The **Parameters** tab provides options for modifying sensor parameters such as detection range, field of view, and resolution.

Supporting Functions

`helperGetPointClouds` Extract an array of `pointCloud` objects that contain lidar sensor data.

```
function ptCloudArr = helperGetPointClouds(simOut)

% Extract signal
ptCloudData = simOut.ptCloudData.signals.values;

% Create a pointCloud array
ptCloudArr = pointCloud(ptCloudData(:,:,:,3)); % Ignore first 2 frames
for n = 4 : size(ptCloudData,4)
    ptCloudArr(end+1) = pointCloud(ptCloudData(:,:,:,n));  %#ok<AGROW>
end
end
```

`helperGetLocalizerGroundTruth` Extract ground truth location and orientation.

```
function [lidarLocation, lidarOrientation, simTimes] = helperGetLocalizerGroundTruth(simOut)

lidarLocation    = squeeze(simOut.lidarLocation.signals.values)';
lidarOrientation = squeeze(simOut.lidarOrientation.signals.values)';
simTimes         = simOut.lidarLocation.time;

% Ignore first 2 frames
lidarLocation(1:2, :)    = [];
lidarOrientation(1:2, :) = [];
simTimes(1:2, :)         = [];
end
```

`helperDrawLocalization` Draw localization estimate and ground truth on axes.

```
function [estHandle,truthHandle] = helperDrawLocalization(axesHandle, ...
    est, estHandle, estColor, truth, truthHandle, truthColor)

% Create scatter objects and adjust legend
if isempty(estHandle) || isempty(truthHandle)
    markerSize = 6;
    hold(axesHandle, 'on');
    estHandle   = scatter3(axesHandle, NaN, NaN, NaN, markerSize, estColor, 'filled');
    truthHandle = scatter3(axesHandle, NaN, NaN, NaN, markerSize, truthColor, 'filled');
    %legend(axesHandle, {'Points', 'Estimate', 'Truth'}, ...
    %    'Color', [1 1 1], 'Location', 'northeast');
    hold(axesHandle, 'off');
end

estHandle.XData(end+1) = est(1);
estHandle.YData(end+1) = est(2);
estHandle.ZData(end+1) = est(3);

truthHandle.XData(end+1) = truth(1);
truthHandle.YData(end+1) = truth(2);
```

```
truthHandle.ZData(end+1) = truth(3);
end
```

helperSuperimposeMapOnSceneImage Superimpose point cloud map on scene image

```
function hFig = helperSuperimposeMapOnSceneImage(sceneName, ptCloudAccum)

hFig = figure('Name', 'Point Cloud Map');
hIm = helperShowSceneImage(sceneName);

hold(hIm.Parent, 'on')
pcshow(ptCloudAccum);
hold(hIm.Parent, 'off')

xlim(hIm.Parent, [-10 50]);
ylim(hIm.Parent, [-30 20]);
end
```

helperDisplayMetrics Display metrics to assess quality of localization.

```
function hFig = helperDisplayMetrics(vSet, lidarLocation, lidarOrientation, simTimes)

absPoses = vSet.Views.AbsolutePose;
translationEstimates = vertcat(absPoses.Translation);
rotationEstimates    = pagetranspose(cat(3, absPoses.Rotation));

viewIds = vSet.Views.ViewId;
viewTimes = simTimes(viewIds);

xEst   = translationEstimates(:, 1);
yEst   = translationEstimates(:, 2);
yawEst = euler(quaternion(rotationEstimates, 'rotmat', 'point'), 'ZYX', 'point');
yawEst = yawEst(:, 1);

xTruth   = lidarLocation(viewIds, 1);
yTruth   = lidarLocation(viewIds, 2);
yawTruth = lidarOrientation(viewIds, 3);

xDeviation   = abs(xEst - xTruth);
yDeviation   = abs(yEst - yTruth);
yawDeviation = abs(helperWrapToPi(yawTruth - yawEst));

hFig = figure('Name', 'Metrics - Absolute Deviation');
subplot(3,1,1)
plot(viewTimes, xDeviation, 'LineWidth', 2);
ylim([0 1])
grid on
title('X')
xlabel('Time (s)')
ylabel('Deviation (m)')

subplot(3,1,2)
plot(viewTimes, yDeviation, 'LineWidth', 2);
ylim([0 1])
grid on
title('Y')
xlabel('Time (s)')
ylabel('Deviation (m)')
```

```
subplot(3,1,3)
plot(viewTimes, yawDeviation, 'LineWidth', 2);
ylim([0 pi/20])
grid on
title('Yaw')
xlabel('Time (s)')
ylabel('Deviation (rad)')
end
```

`helperWrapToPi` Wrap angles to be in the range $[-\pi, \pi]$.

```
function angle = helperWrapToPi(angle)

idx = (angle < -pi) | (angle > pi);
angle(idx) = helperWrapTo2Pi(angle(idx) + pi) - pi;
end
```

`helperWrapTo2Pi` Wrap angles to be in the range $[-2\pi, 2\pi]$.

```
function angle = helperWrapTo2Pi(angle)

pos = (angle>0);
angle = mod(angle, 2*pi);
angle(angle==0 & pos) = 2*pi;
end
```

See Also

Functions
`pcalign` | `pccat` | `pcplayer` | `pctransform` | `pcviewset`

Blocks
Simulation 3D Lidar | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following

More About

- “Select Waypoints for Unreal Engine Simulation” on page 7-626
- “Build a Map from Lidar Data Using SLAM” on page 7-559
- “Design Lidar SLAM Algorithm Using Unreal Engine Simulation Environment” on page 7-691

Develop Visual SLAM Algorithm Using Unreal Engine Simulation

This example shows how to develop a visual Simultaneous Localization and Mapping (SLAM) algorithm using image data obtained from the Unreal Engine® simulation environment.

Visual SLAM refers to the process of calculating the position and orientation of a camera with respect to its surroundings while simultaneously mapping the environment. However, developing a visual SLAM algorithm and evaluating its performance in varying conditions is a challenging task. One of the biggest challenges is generating the ground truth of the camera sensor, especially in outdoor environments. The use of simulation enables testing under a variety of scenarios and camera configurations while providing precise ground truth.

This example demonstrates the use of Unreal Engine simulation to develop a monocular visual SLAM algorithm in a parking scenario. For more information about the implementation of the visual SLAM pipeline, see the “Monocular Visual Simultaneous Localization and Mapping” (Computer Vision Toolbox) example.

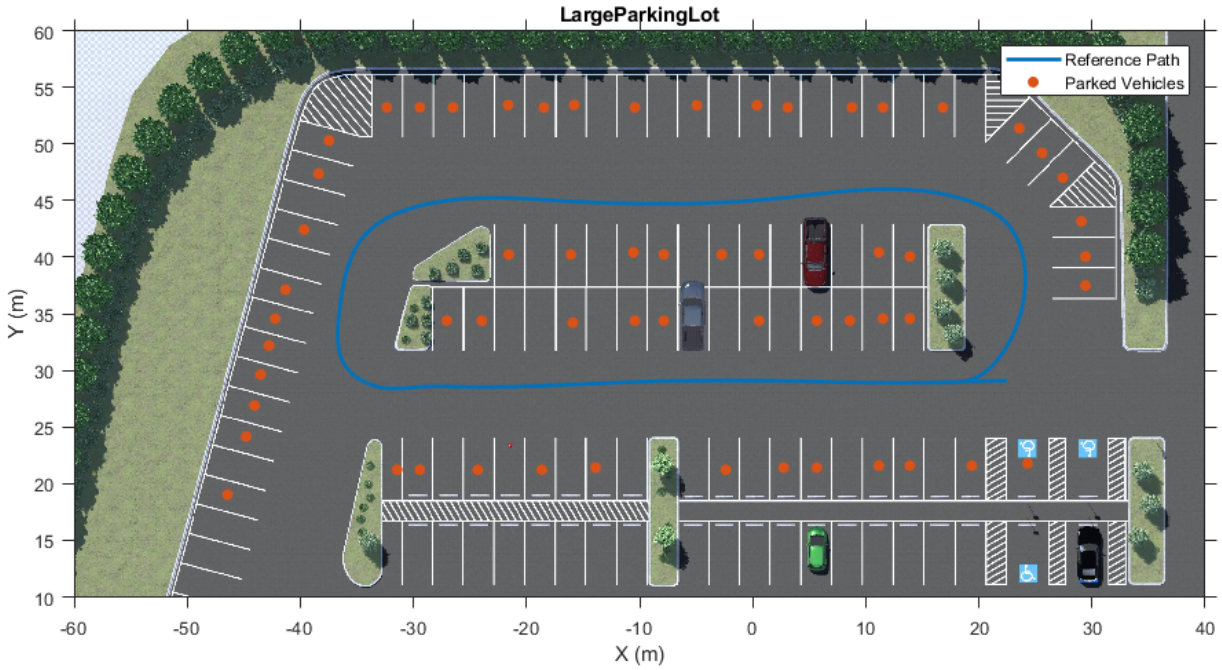
Set up Scenario in Simulation Environment

Use the Simulation 3D Scene Configuration block to set up the simulation environment. Select the built-in Large Parking Lot scene, which contains several parked vehicles. The visual SLAM algorithm matches features across consecutive images. To increase the number of potential feature matches, you can use the Parked Vehicles subsystem to add more parked vehicles to the scene. To specify the parking poses of the vehicles, use the `helperAddParkedVehicle` function. If you select a more natural scene, the presence of additional vehicles is not necessary. Natural scenes usually have enough texture and feature variety suitable for feature matching.

You can follow the “Select Waypoints for Unreal Engine Simulation” on page 7-626 example to interactively select a sequence of parking locations. You can use the same approach to select a sequence of waypoints and generate a reference trajectory for the ego vehicle. This example uses a recorded reference trajectory and parked vehicle locations.

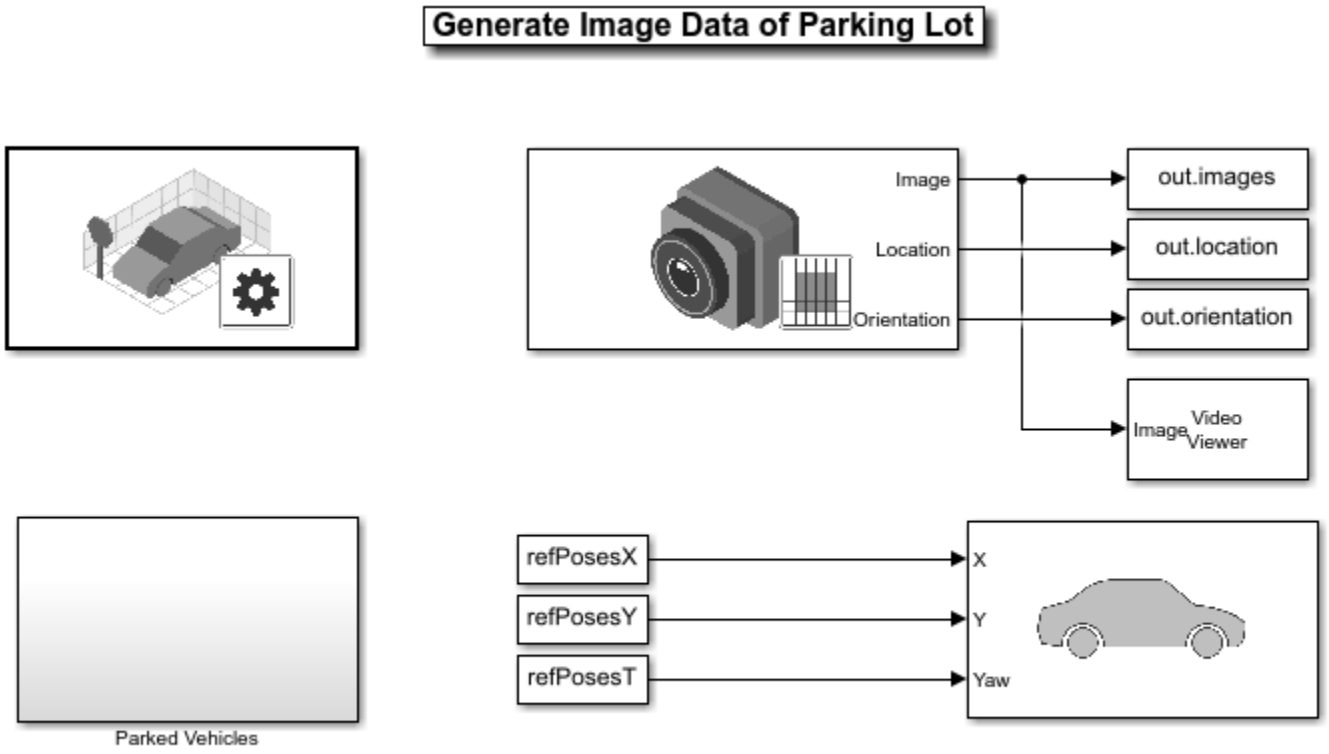
```
% Load reference path
data = load('parkingLotReferenceData.mat');

% Set reference trajectory of the ego vehicle
refPosesX = data.refPosesX;
refPosesY = data.refPosesY;
refPosesT = data.refPosesT;

% Set poses of the parked vehicles
parkedPoses = data.parkedPoses;

% Display the reference path and the parked vehicle locations
sceneName = 'LargeParkingLot';
hScene = figure;
helperShowSceneImage(sceneName);
hold on
plot(refPosesX(:,2), refPosesY(:,2), 'LineWidth', 2, 'DisplayName', 'Reference Path');
scatter(parkedPoses(:,1), parkedPoses(:,2), [], 'filled', 'DisplayName', 'Parked Vehicles');
xlim([-60 40])
ylim([10 60])
hScene.Position = [100, 100, 1000, 500]; % Resize figure
```

```
legend
hold off
```



Open the model and add parked vehicles.

```
modelName = 'GenerateImageDataOfParkingLot';
open_system(modelName);
snapnow;

helperAddParkedVehicles(modelName, parkedPoses);
```



Set Ego Vehicle and Camera Sensor

Set up the ego vehicle moving along the specified reference path by using the Simulation 3D Vehicle with Ground Following block. Mount a camera on the vehicle roof center by using the Simulation 3D Camera block and specify its intrinsic parameters. You can use the **Camera Calibrator (Computer Vision Toolbox)** app to estimate camera intrinsics of the actual camera that you want to simulate.

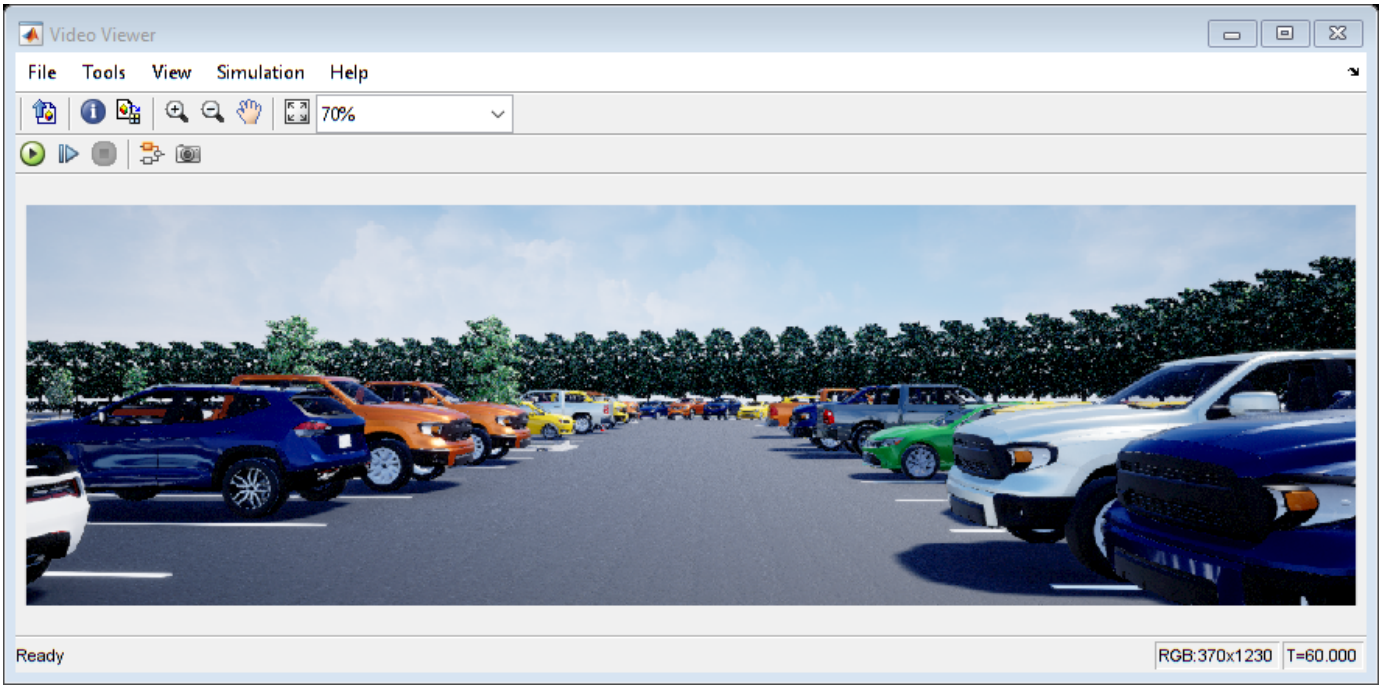
```
% Camera intrinsics
focalLength    = [700, 700];  % specified in units of pixels
principalPoint = [600, 180];  % in pixels [x, y]
imageSize      = [370, 1230]; % in pixels [mrows, ncols]
intrinsics     = cameraIntrinsics(focalLength, principalPoint, imageSize);
```

Visualize and Record Sensor Data

Run the simulation to visualize and record sensor data. Use the Video Viewer block to visualize the image output from the camera sensor. Use the To Workspace block to record the ground truth location and orientation of the camera sensor.

```
close(hScene)

if ~ispc
    error("Unreal Engine Simulation is supported only on Microsoft" + char(174) + " Windows" + cl
end

% Open video viewer to examine camera images
```

```
open_system([modelName, '/Video Viewer']);

% Run simulation
simOut = sim(modelName);
snapnow;

% Extract camera images as an imageDatastore
imds = helperGetCameraImages(simOut);

% Extract ground truth as an array of rigid3d objects
gTruth = helperGetSensorGroundTruth(simOut);

close_system([modelName, '/Video Viewer']);
```



Develop Visual SLAM Algorithm Using Recorded Data

Use the images to evaluate the visual SLAM algorithm. The function `helperVisualSLAM` implements the ORB-SLAM pipeline:

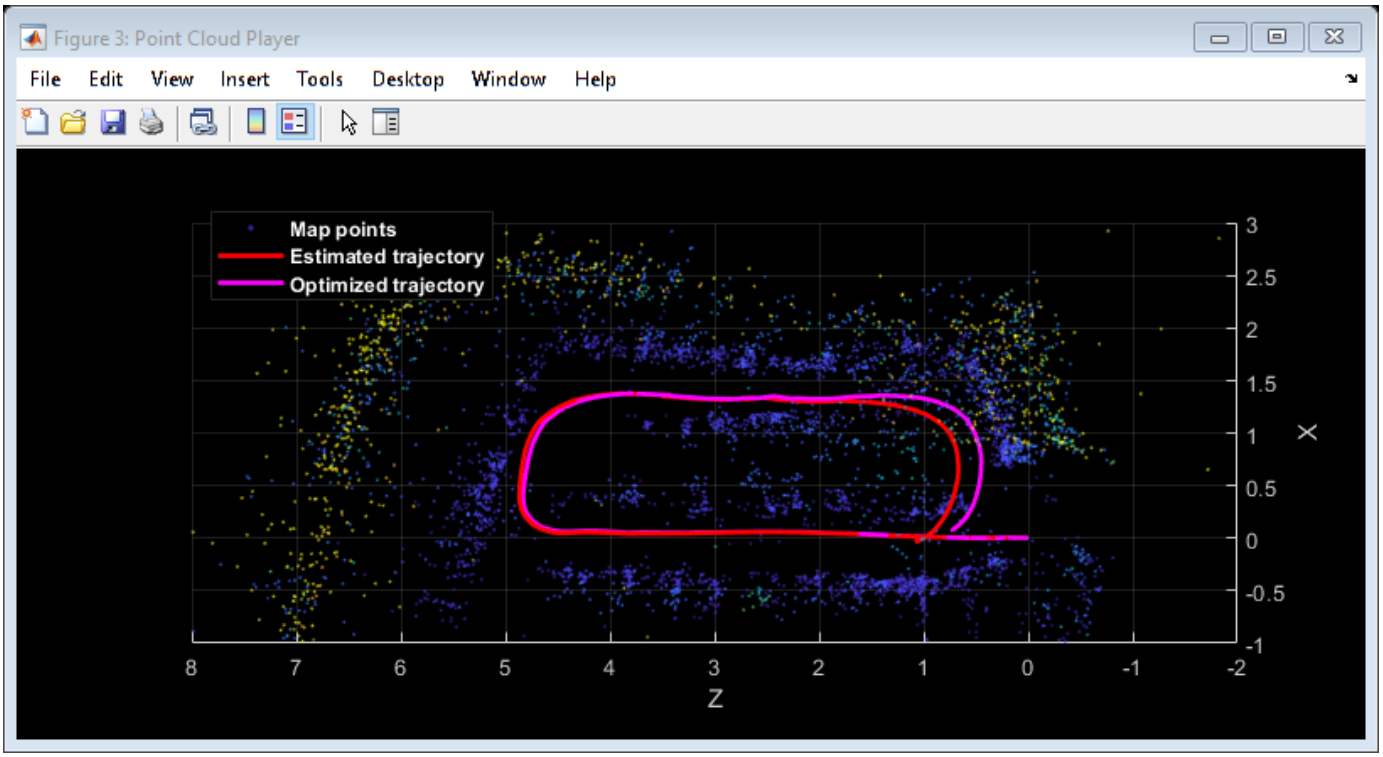
- **Map Initialization**: ORB-SLAM starts by initializing the map of 3-D points from two images. Use `relativeCameraPose (Computer Vision Toolbox)` to compute the relative pose based on 2-D ORB feature correspondences and `triangulate (Computer Vision Toolbox)` to compute the 3-D map points. The two frames are stored in an `imageviewset (Computer Vision Toolbox)` object as key frames. The 3-D map points and their correspondences to the key frames are stored in a `worldpointset` object.
- **Tracking**: Once a map is initialized, for each new image, the function `helperTrackLastKeyFrame` estimates the camera pose by matching features in the current frame to features in the last key frame. The function `helperTrackLocalMap` refines the estimated camera pose by tracking the local map.

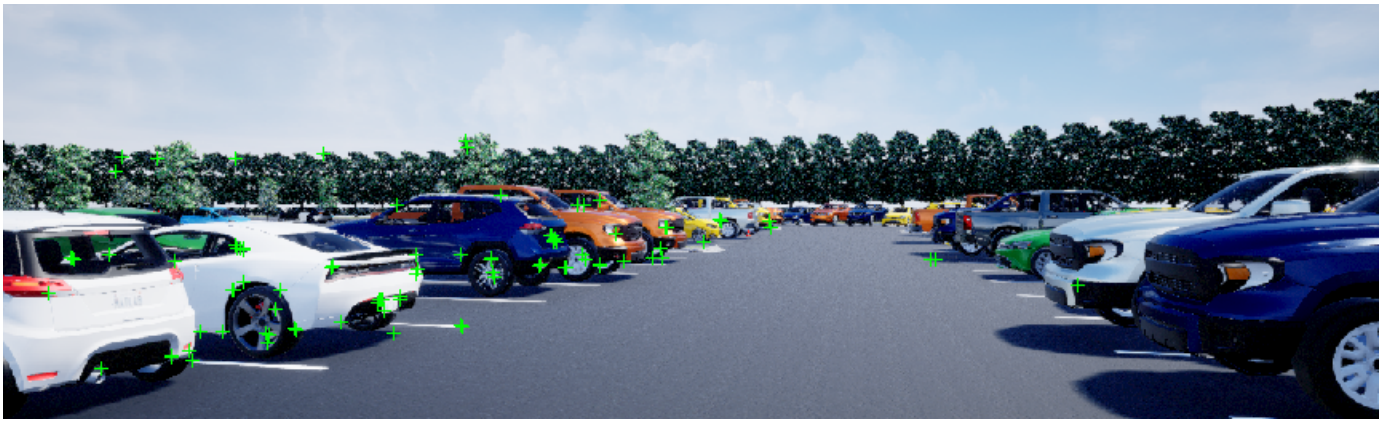
- **Local Mapping**: The current frame is used to create new 3-D map points if it is identified as a key frame. At this stage, `bundleAdjustment (Computer Vision Toolbox)` is used to minimize reprojection errors by adjusting the camera pose and 3-D points.
- **Loop Closure**: Loops are detected for each key frame by comparing it against all previous key frames using the bag-of-features approach. Once a loop closure is detected, the pose graph is optimized to refine the camera poses of all the key frames using the `optimizePoseGraph (Navigation Toolbox)` function.

For the implementation details of the algorithm, see the “Monocular Visual Simultaneous Localization and Mapping” (Computer Vision Toolbox) example.

```
[mapPlot, optimizedPoses, addedFramesIdx] = helperVisualSLAM(imds, intrinsics);
```

```
Map initialized with frame 1 and frame 3
Loop edge added between keyframe: 5 and 173
Iteration 1, residual error 0.329001
Iteration 2, residual error 0.322270
Iteration 3, residual error 0.322259
Iteration 4, residual error 0.322259
Solver stopped because change in function value was less than specified function tolerance.
```

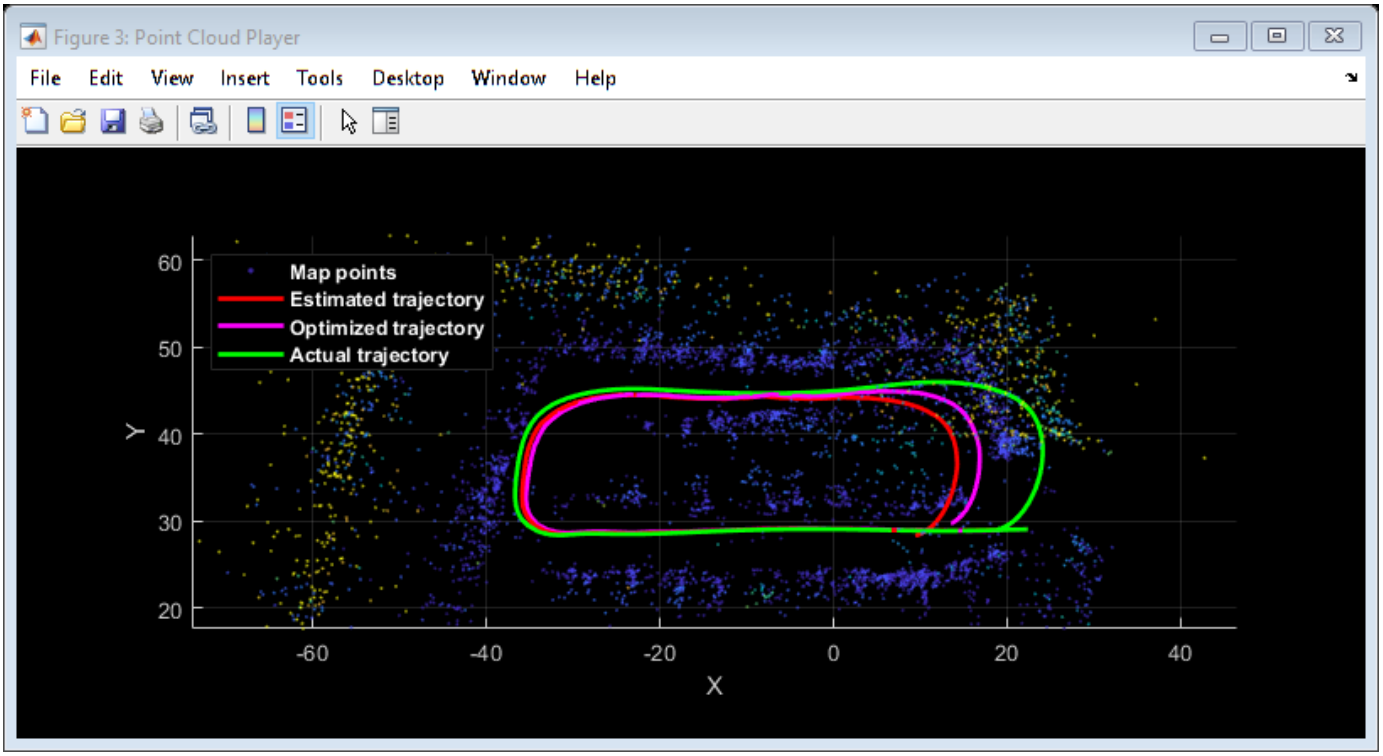




Evaluate Against Ground Truth

You can evaluate the optimized camera trajectory against the ground truth obtained from the simulation. Since the images are generated from a monocular camera, the trajectory of the camera can only be recovered up to an unknown scale factor. You can approximately compute the scale factor from the ground truth, thus simulating what you would normally obtain from an external sensor.

```
% Plot the camera ground truth trajectory
scaledTrajectory = plotActualTrajectory(mapPlot, gTruth(addedFramesIdx), optimizedPoses);

% Show legend
showLegend(mapPlot);
```



You can also calculate the root mean square error (RMSE) of trajectory estimates.

```
helperEstimateTrajectoryError(gTruth(addedFramesIdx), scaledTrajectory);
```

```
Absolute RMSE for key frame trajectory (m): 4.0924
```

Close model and figures.

```
close_system(modelName, 0);
close all
```

Supporting Functions

helperGetCameraImages Get camera output

```
function imds = helperGetCameraImages(simOut)
% Save image data to a temporary folder
dataFolder   = fullfile(tempdir, 'parkingLotImages', filesep);
folderExists = exist(dataFolder, 'dir');
if ~folderExists
    mkdir(dataFolder);
end

files = dir(dataFolder);
if numel(files) < 3
    numFrames = numel(simOut.images.Time);
    for i = 3:numFrames % Ignore the first two frames
        img = squeeze(simOut.images.Data(:,:,:,i));
        imwrite(img, [dataFolder, sprintf('%04d', i-2), '.png'])
    end
end

% Create an imageDatastore object to store all the images
imds = imageDatastore(dataFolder);
end
```

helperGetSensorGroundTruth Save the sensor ground truth

```
function gTruth = helperGetSensorGroundTruth(simOut)
numFrames = numel(simOut.images.Time);
gTruth = repmat(rigid3d, numFrames-2, 1);
for i = 1:numFrames-2 % Ignore the first two frames
    gTruth(i).Translation = squeeze(simOut.location.Data(:, :, i+2));
    % Ignore the roll and the pitch rotations since the ground is flat
    yaw = double(simOut.orientation.Data(:, 3, i+2));
    gTruth(i).Rotation = [cos(yaw), sin(yaw), 0; ...
        -sin(yaw), cos(yaw), 0; ...
        0, 0, 1];
end
end
```

helperEstimateTrajectoryError Calculate the tracking error

```
function rmse = helperEstimateTrajectoryError(gTruth, scaledLocations)
gLocations      = vertcat(gTruth.Translation);

rmse = sqrt(mean( sum((scaledLocations - gLocations).^2, 2) ));
```

```
disp(['Absolute RMSE for key frame trajectory (m): ', num2str(rmse)]);
end
```

See Also

`bundleAdjustment` | `imageviewset` | `optimizePoses` | `relativeCameraPose` | `triangulate`

More About

- “Monocular Visual Simultaneous Localization and Mapping” (Computer Vision Toolbox)
- “Structure From Motion From Multiple Views” (Computer Vision Toolbox)
- “Select Waypoints for Unreal Engine Simulation” on page 7-626
- “Design Lidar SLAM Algorithm Using Unreal Engine Simulation Environment” on page 7-691

Automatic Scenario Generation

This example shows how to automate scenario generation by using `drivingScenario` object. In this example, you will automate

- vehicle placements in a scenario by defining their start and the goal positions
- waypoint selection and trajectory generation for the vehicles to traverse from their start positions to goal positions.
- speed adjustment such that the vehicles accelerate or deaccelerate to avoid colliding between other vehicles that travel in the same lane.

You can use this example to synthesize a number of random scenarios for testing the driving algorithms.

Introduction

The `drivingScenario` object and the Driving Scenario Designer app in the Automated Driving Toolbox™ are efficient tools for generating synthetic driving scenarios. You can create a road network or import a road network from OpenDRIVE®, HERE HD Live Map, and OpenStreetMap®. Then, you can add actors or vehicles to the road network and define their trajectories to synthesize a driving scenario. The waypoints required for generating the trajectories must be selected carefully such that the trajectories of the vehicles lie within the road network and the vehicles does not collide as they travel along their trajectories. Defining such vehicle placements and trajectories often requires multiple trials and is time consuming if you have large road networks and many number of vehicles to configure.

This example provides helper functions and demonstrates the steps that you can use to automate vehicle placements and trajectory generation by using the `drivingScenario` object. You can also export the generated scenario to Driving Scenario Designer app. The rest of the example demonstrates these steps involved in automating scenario generation.

1. **Import road network** - Import OpenStreetMap® road network into a driving scenario object by using the helper function `helperOSMimport`.
2. **Define start and goal positions** - Specify region of interests (ROIs) in the road network to select the start and goal positions for vehicles by using the helper function `helperSamplePositions`.
3. **Generate vehicle trajectories** - Generate waypoints and trajectories by using the helper function `helperGenerateWaypoints` and Automated Driving Toolbox™ function `trajectory`.
4. **Modify speed profiles to avoid collision** - Modify the speed profiles of the vehicles in the scenario by using the Simulink model `CollisionFreeSpeedManipulator`. The model checks the speed profile of each vehicle and prevents them from colliding with each other as they travel along their trajectories. The output from the model is an updated scenario that is free from collision between vehicles. You can convert the output from the `CollisionFreeSpeedManipulator` Simulink model to a driving scenario object by using the helper function `helpergetCFSMScenario`.
5. **Simulate and visualize generated scenario** - Simulate and display the automatically generated scenario by using the `plot` function. You can also read and simulate the automatically generated scenario by using the Driving Scenario Designer app.

Import Road Network

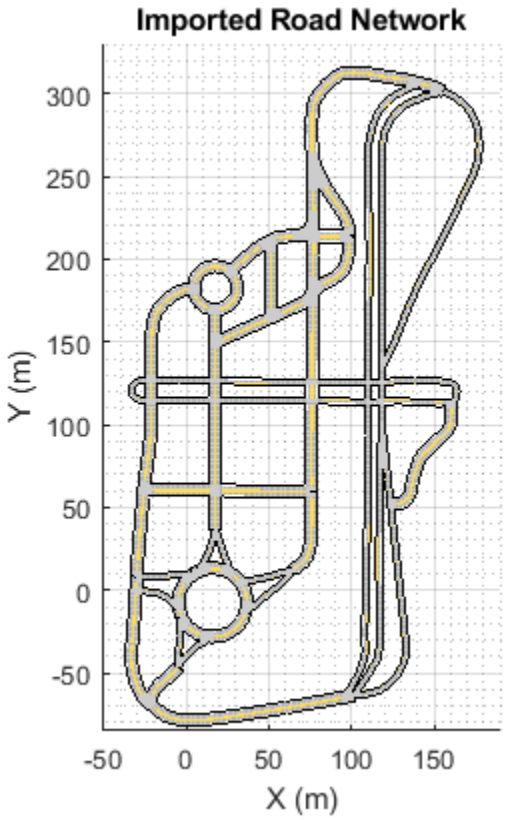
You can download a road network from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

Specify the bounding box coordinates to import the MCity test facility map from `openstreetmap.org` by using the `helperOSMImport` function. The function returns a driving scenario object that contains the road network from the imported map data. You can also use the `roadNetwork` function to import a road network from OpenDRIVE®, HERE HD Live Map, or OpenStreetMap® files.

```
% Import the road network of MCity
minLat =  42.2990;
maxLat =  42.3027;
minLon = -83.6996;
maxLon = -83.6965;
bbox = [minLat maxLat;minLon maxLon];
scenario = helperOSMImport(bbox);
% Create another scenario object for plotting purpose
scenario_in = helperOSMImport(bbox);
```

Display the MCity road network by using the `plot` function.

```
figure
plot(scenario_in)
title('Imported Road Network')
xlim([-50 190])
ylim([-85 330])
```



Define Start and Goal positions

To create a driving scenario, you must first define specific points on the road network that can serve as start and goal positions for the vehicles in the scenario. Use the `helperSamplePositions` function to generate a random set of points in the road network that you can select as start and goal positions for the vehicles. You can use one or more name-value pair arguments of `helperSamplePositions` functions to configure the start and goal positions in different ways.

- Use the '`Seed`' name-value pair argument of `helperSamplePositions` function to specify the random generator settings to be used for generating random points. You can select any points in the generated set as start and goal positions.
- Use the '`ROI`' name-value pair argument of `helperSamplePositions` function to specify one or more region-of-interests (ROIs) in the road network within which you want to define the start and goal positions. The ROIs can be circular, rectangular or a polygon region with any number of vertices. The value for ROI is an N-by-2 matrix specifying the spatial coordinates of a closed region. If not specified, the function generates random points across the entire road network.
- Use the '`Lanes`' name-value pair argument of `helperSamplePositions` function to specify the lanes in which you want to define the start and the goal positions. To select a single lane, specify the lane number as a scalar value. In case of multiple lanes, the value for '`Lanes`' name-value argument must be a vector containing the desired lane numbers. If not specified, the function selects the lanes randomly.
- Use the '`LongitudinalDistance`' name-value pair argument of `helperSamplePositions` function to set the longitudinal distance between two consecutive points. If not specified, the function imposes at least 5 m distance between two consecutive points in the same lane. This

implies that the longitudinal distance between two consecutive vehicles placed in the same lane is at least 5 m.

During simulation, the vehicles spawn at the start points and then, travels to reach the goal points.

1. Select Start Positions

Specify the number of random points to be generated for selecting the start positions as 10. Specify a flag for setting the random number generator. Set the value for `setSeed` to 1 to specify the seed for random number generator. Pass the random generator settings as input to the `helperSamplePositions` function by using the 'Seed' name-value pair argument.

The `helperSamplePositions` function outputs the 3-D spatial coordinates of the randomly selected points in the imported road network. The function also outputs the yaw angles relative to the selected points. The yaw angle obtained relative to a point defines the orientation for the vehicle to be placed at that point.

```
numPoints = 10;
setSeed = 1;
if setSeed == 1
    seed = 2;
    rng(seed);
    s = rng;
    [points,yaw] = helperSamplePositions(scenario,numPoints,'Seed',s);
else
    [points,yaw] = helperSamplePositions(scenario,numPoints);
end
```

Specify the number of vehicles to be placed in the scenario as 3. Select any three points in the generated set as the start positions for the vehicles.

```
numVehicles = 3;
startSet1 = [points(2,:);points(4,:);points(7,:)];
yaw1 = [yaw(2);yaw(4);yaw(7)];
```

Place vehicles in the selected points by using `vehicle` function.

```
for idx = 1 : numVehicles
    vehicle(scenario,'Position',startSet1(idx,:),'Yaw',yaw1(idx),'ClassID',1);
end
```

Generate another set of points by defining the ROIs. Compute the coordinates to specify a circular ROI.

```
xCor = 0;
yCor = 0;
radius = 50;
thetha = 0: pi/10: 2*pi;
roiCircular(:,1) = xCor+radius*cos(thetha);
roiCircular(:,2) = yCor+radius*sin(thetha);
```

Specify the number of points to be generated within the ROI and the number of vehicles to place within the ROI as 3. Select all the points within the circular ROI as the start positions for the vehicles.

```
numPoints = 3;
numVehicles = numPoints;
[startSet2,yaw2] = helperSamplePositions(scenario,numPoints,'ROI',roiCircular);
```

```
for idx = 1 : size(startSet2,1)
    vehicle(scenario,'Position',startSet2(idx,:),'Yaw',yaw2(idx),'ClassID',1);
end
```

Specify the coordinates for a rectangular ROI. Set the number of points to be generated within the ROI and the number of vehicles to place within the ROI as 3. Set the longitudinal distance between two consecutive points in the same lane as 30m. If the area of ROI is not large enough to accomodate the specified number of points at the specified longitudinal distance, then the `helperSamplePositions` function returns only those number of points that can be accommodated within the ROI. To get the desired number of points, you must either reduce the longitudinal distance or increase the area of the ROI.

```
roiRectangular = [0 0;100 100];
numPoints = 3;
[startSet3,yaw3] = helperSamplePositions(scenario,numPoints,'ROI',roiRectangular,'LongitudinalDis
```

Place vehicles in the selected points by using the `vehicle` function.

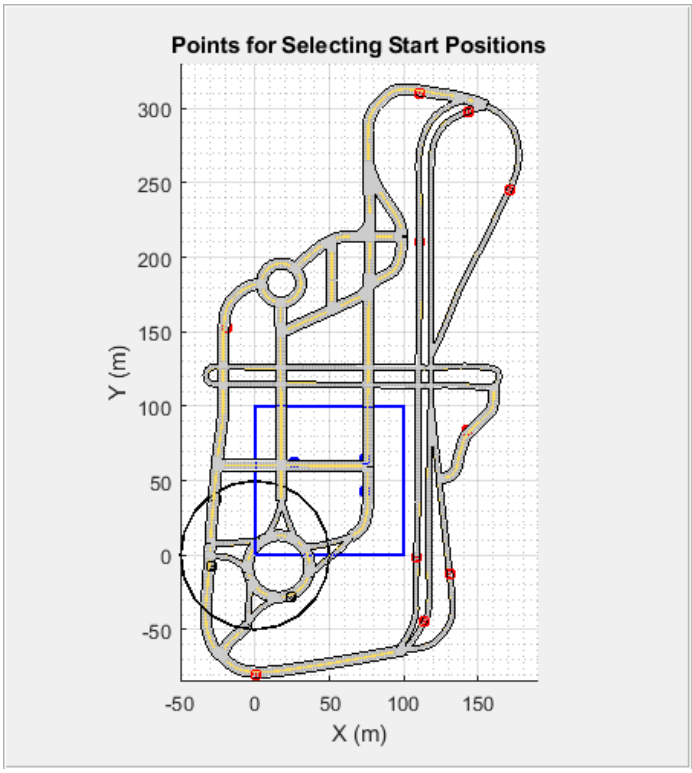
```
for idx = 1 : size(startSet3,1)
    vehicle(scenario,'Position',startSet3(idx,:),'Yaw',yaw3(idx),'ClassID',1);
end
```

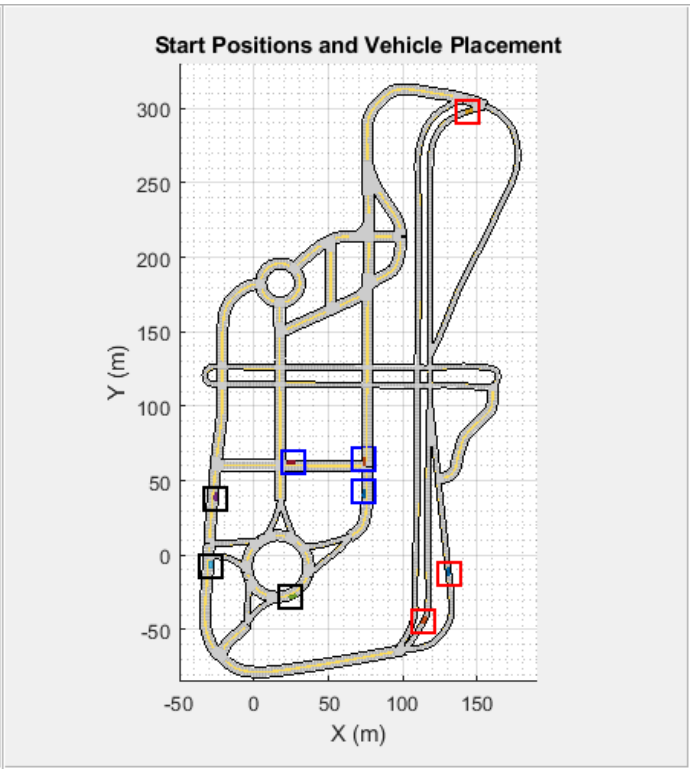
Plot the generated sample points and the ROIs.

```
figScene = figure('Name','AutomaticScenarioGeneration');
set(figScene,'Position',[0, 0, 900,500]);

hPanel1 = uipanel(figScene,'Position',[0 0 0.5 1]);
hPlot1 = axes(hPanel1);
plot(scenario_in,'Parent',hPlot1);
title('Points for Selecting Start Positions')
hold on
plot(points(:,1),points(:,2),'ro','MarkerSize',5,'MarkerFaceColor','r');

plot(roiCircular(:,1),roiCircular(:,2),'LineWidth',1.2,'Color','k')
plot(startSet2(:,1),startSet2(:,2),'ko','MarkerSize',5,'MarkerFaceColor','k');

plot([roiRectangular(1,1);roiRectangular(1,1);roiRectangular(2,1);roiRectangular(2,1);roiRectangu
     [roiRectangular(1,2);roiRectangular(2,2);roiRectangular(2,2);roiRectangular(1,2);roiRectangu
     'LineWidth',1.2,'Color','b');
plot(startSet3(:,1),startSet3(:,2),'bo','MarkerSize',5,'MarkerFaceColor','b');
xlim([-50 190])
ylim([-85 330])
hold off
```

Display the start positions and the vehicles in the scenario.

```
hPanel2 = uipanel(figScene,'Position',[0.5 0 0.5 1]);
hPlot2 = axes(hPanel2);
plot(scenario,'Parent',hPlot2);
title('Start Positions and Vehicle Placement')
hold on
plot(startSet1(:,1),startSet1(:,2),'rs','MarkerSize',15,'LineWidth',1.2);
plot(startSet2(:,1),startSet2(:,2),'ks','MarkerSize',15,'LineWidth',1.2);
plot(startSet3(:,1),startSet3(:,2),'bs','MarkerSize',15,'LineWidth',1.2);
xlim([-50 190])
ylim([-85 330])
hold off
```





Merge all the start positions into a single matrix. The number of start positions implies the total number of vehicles in the driving scenario.

```
startPositions = [startSet1;startSet2;startSet3];
```

2. Inspect Scenario Object

Display the scenario object and inspect its properties. The `Actors` property of the `scenario` object is an 1-by-9 array and it stores information about the 9 vehicles that are added to the driving scenario. Access the details of each vehicle in `Actors` property by using dot indexing. Display the details about the first vehicle in the driving scenario. The `Position` property contains the start position of the vehicle.

```
scenario
```

```
scenario =
  drivingScenario with properties:

        SampleTime: 0.0100
          StopTime: Inf
    SimulationTime: 0
         IsRunning: 1
            Actors: [1x9 driving.scenario.Vehicle]
```

```
scenario.Actors(1)
```

```
ans =
  Vehicle with properties:
```

```
          FrontOverhang: 0.9000
           RearOverhang: 1
              Wheelbase: 2.8000
              EntryTime: 0
               ExitTime: Inf
                ActorID: 1
                ClassID: 1
                   Name: ""
              PlotColor: [0 0.4470 0.7410]
               Position: [130.7903 -12.2335 -2.0759e-04]
               Velocity: [0 0 0]
                    Yaw: 96.6114
                  Pitch: 0
                   Roll: 0
        AngularVelocity: [0 0 0]
                 Length: 4.7000
                  Width: 1.8000
                 Height: 1.4000
                   Mesh: [1x1 extendedObjectMesh]
             RCSPattern: [2x2 double]
       RCSAzimuthAngles: [-180 180]
     RCSElevationAngles: [-90 90]
```

3. Select Goal Positions

Generate the goal positions for the vehicles in the scenario by using the `helperSamplePositions` function. The total number of goal positions must be same as the total number of start positions.

```
numGoalPositions = length(startPositions)
```

```
numGoalPositions = 9
```

Specify the coordinates for a polygon ROI and find 5 random points within the polygon ROI. Select these points as the goal positions for the first 5 vehicles in the scenario.

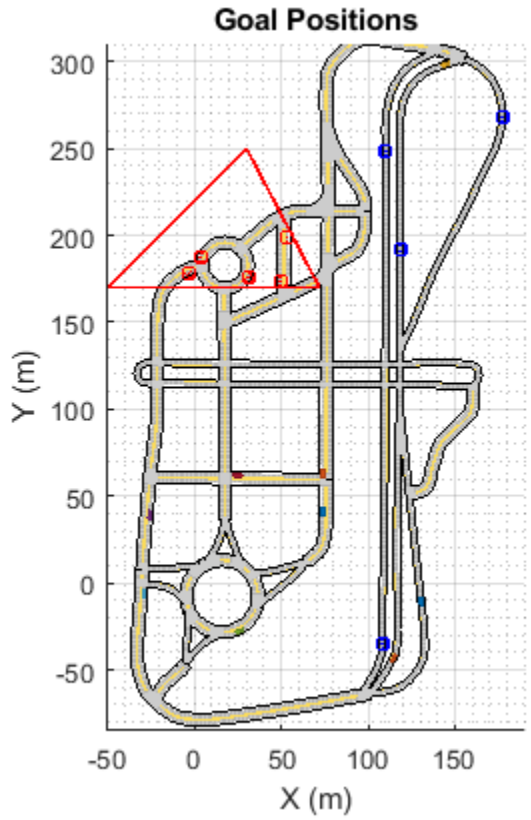
```
roiPolygon = [-50  170;30  250;72 170;-50 170];
numPoints1 = 5;
goalSet1 = helperSamplePositions(scenario,numPoints1,'ROI',roiPolygon);
```

Display the scenario and the selected goal positions.

```
figure
plot(scenario);
title('Goal Positions')
hold on
plot(roiPolygon(:,1), roiPolygon(:,2),'LineWidth',1.2,'Color','r')
plot(goalSet1(:,1), goalSet1(:,2),'ro','MarkerSize',5,'MarkerFaceColor','r')
```

Generate the remaining set of goal positions in such a way that they all lie in a specific lane. Use the '`Lanes`' name-value pair argument to specify the lane number for goal positions.

```
numPoints2 = 4;
goalSet2 = helperSamplePositions(scenario,numPoints2,'Lanes',1);
plot(goalSet2(:,1),goalSet2(:,2),'bo','MarkerSize',5,'MarkerFaceColor','b')
xlim([-50 190])
ylim([-85 310])
hold off
```



Merge all the goal positions into a single matrix.

```
goalPositions = [goalSet1;goalSet2];
```

Display the start positions and the goal positions with respect to each vehicle in the scenario.

```
vehicleNum = 1:length(startPositions);
table(vehicleNum(:),startPositions,goalPositions,'VariableNames',{'Vehicle','Start positions','Go
```

```
ans=9×3 table
    Vehicle              Start positions                        Goal positions
    _______    ____________________________________    ____________________________________

       1          130.79        -12.233    -0.00020759     49.544         173.51      0.0009993
       2          113.86        -44.576    -0.00076866     53.168         198.75     0.00070254
       3          143.37          297.8       -0.00188     3.5187         187.51     0.00057965
       4         -25.863          37.977     0.00020468     30.912         175.81     0.00087745
       5           24.74        -28.221    -0.00031535    -3.3494         178.66     0.00055764
       6         -29.047         -7.817    -0.00053606     108.15        -34.789    -0.00049719
       7          26.393          62.042     0.00095438     118.42          192.5     0.00054377
       8          73.989          42.717     0.00094018     110.09         248.17    -0.00032446
       9          73.996          64.436      0.0011401      177.2         267.68     -0.0015615
```

Generate Vehicle Trajectories

Use the `helperGenerateWaypoints` function to compute waypoints that connect the start and the goal positions. The function returns a structure array that contains the road centers, computed

waypoints, and yaw angle for each vehicle in the scenario. Read the vehicle information from the `scenario` object and specify random speed values for each vehicle. Use the `trajectory` function to generate the trajectories for each vehicle by using the computed waypoints and random speed values.

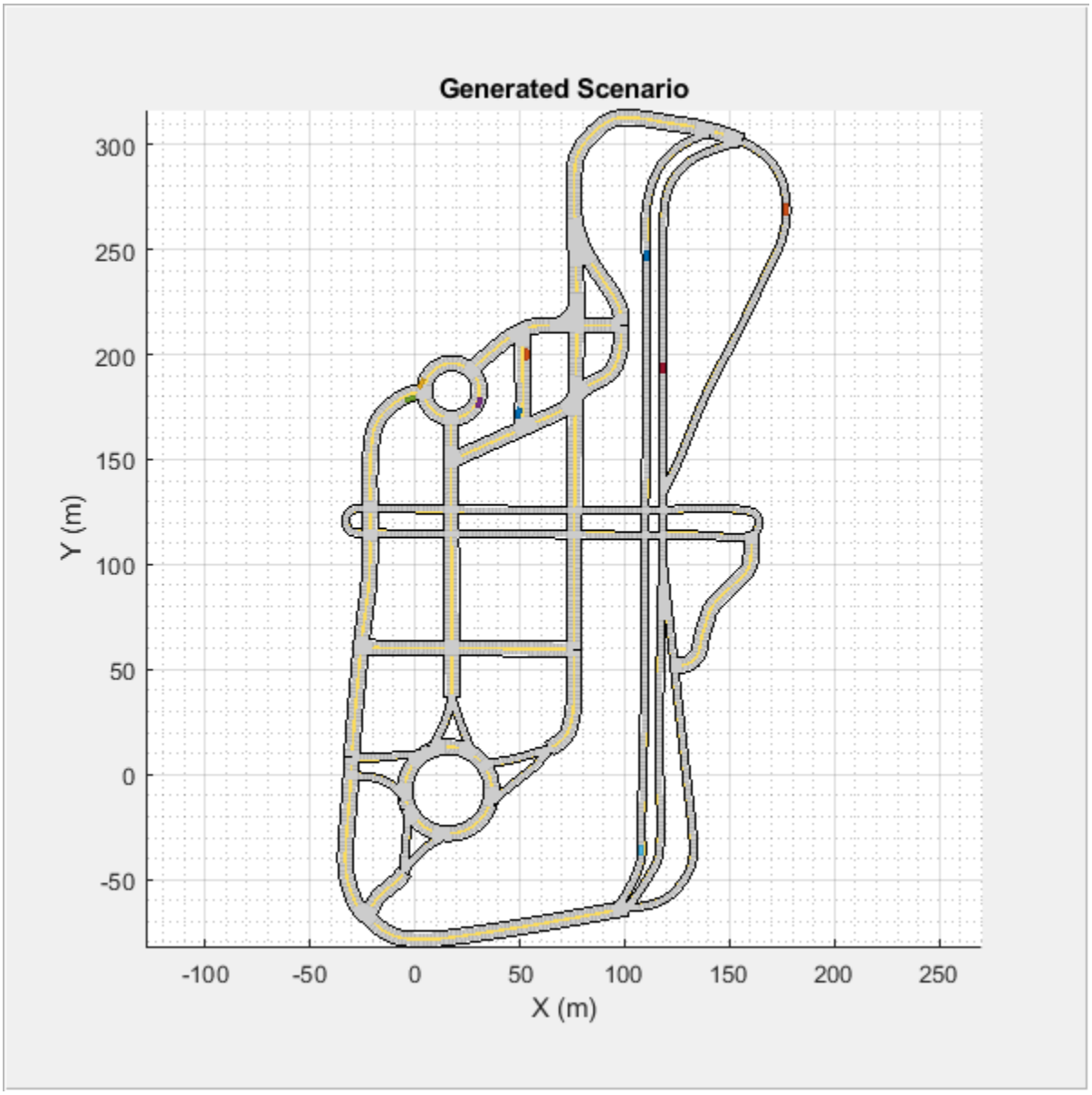
```
info = helperGenerateWaypoints(scenario,startPositions,goalPositions);
for indx = 1:length(startPositions)
    vehicleData = scenario.Actors(indx);
    speed = randi([10,25],1,1);
    waypts = info(indx).waypoints;
    trajectory(vehicleData,waypts,speed);
end
```

Set the stop time for the scenario.

```
scenario.StopTime = 50;
```

Create a custom figure and display the simulated driving scenario.

```
close all;
figScene = figure;
set(figScene,'Position',[0,0,600,600]);
movegui(figScene,'center');
hPanel = uipanel(figScene,'Position',[0 0 1 1]);
hPlot = axes(hPanel);
plot(scenario,'Parent',hPlot);
title('Generated Scenario')
% Run the simulation
while advance(scenario)
    pause(0.01)
end
```



In the generated scenario all the vehicles traverse along their trajectories at a particular speed to reach their goal positions. You can also observe collision between two actors as they traverse along their trajectories. While you synthesize a scenario for testing driving algorithms, it is important that the vehicles in the scenario do not collide. To prevent collision, you must adjust the velocity of the vehicles so that they do not collide with each other while travelling along their paths.

Modify Speed Profile to Avoid Collision

Use the Simulink model `CollisionFreeSpeedManipulator` to correct the velocity of the vehicles such that they do not collide as they traverse along their trajectories. The model uses non-linear time scaling to reactively accelerate or de-accelerate a vehicle without altering its trajectory [1].

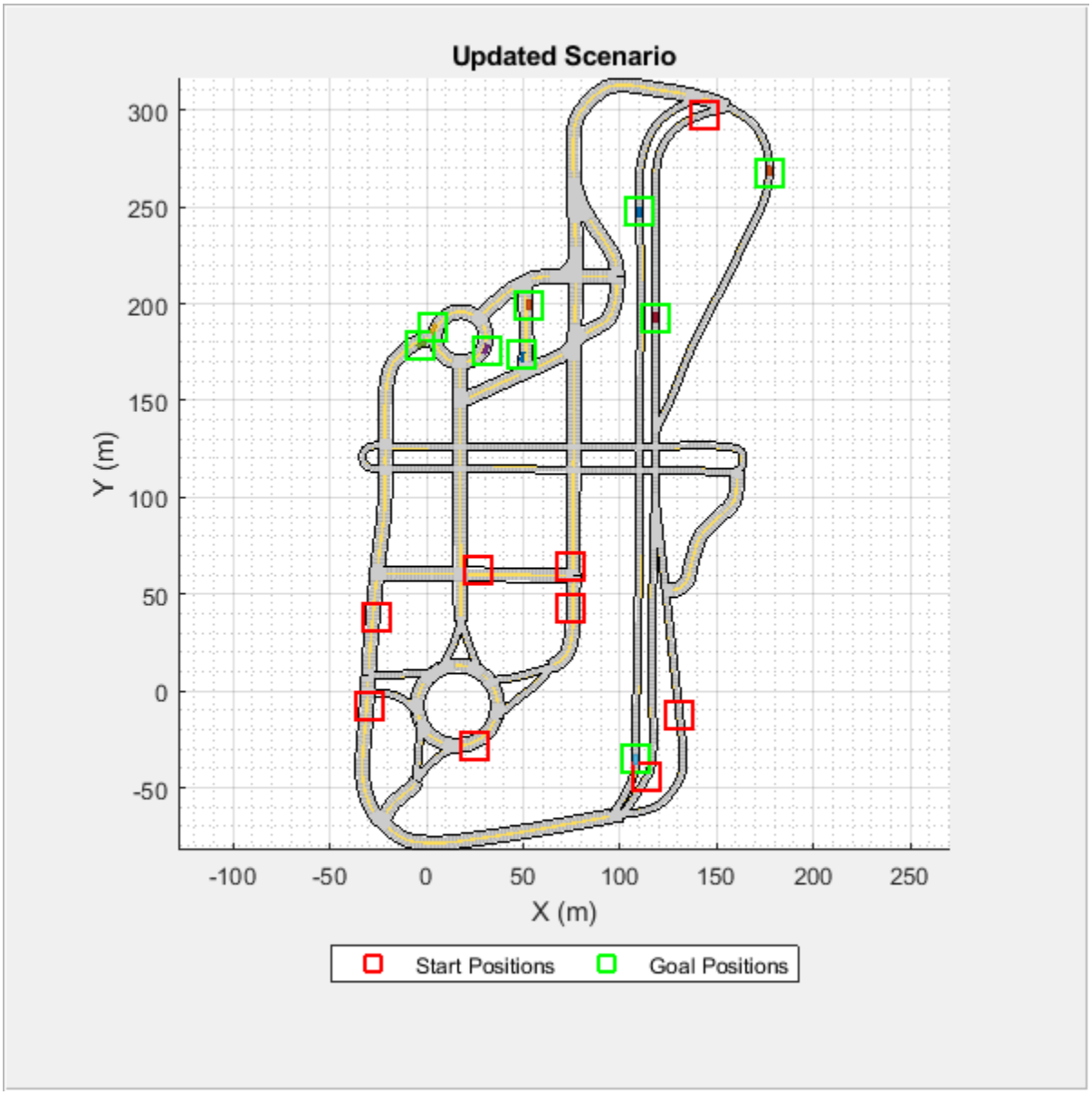
```
% Open the Simulink system block
open_system('CollisionFreeSpeedManipulator');
```

```
% Pass the scenario object as input
set_param('CollisionFreeSpeedManipulator/VelocityUpdate',...
          'ScenarioName','scenario')
% Run the simulation and log the output
out = sim('CollisionFreeSpeedManipulator','StopTime','50');
% Clear all the temporary variables and close the Simulink block
bdclose
```

Simulate and Visualize Generated Scenario

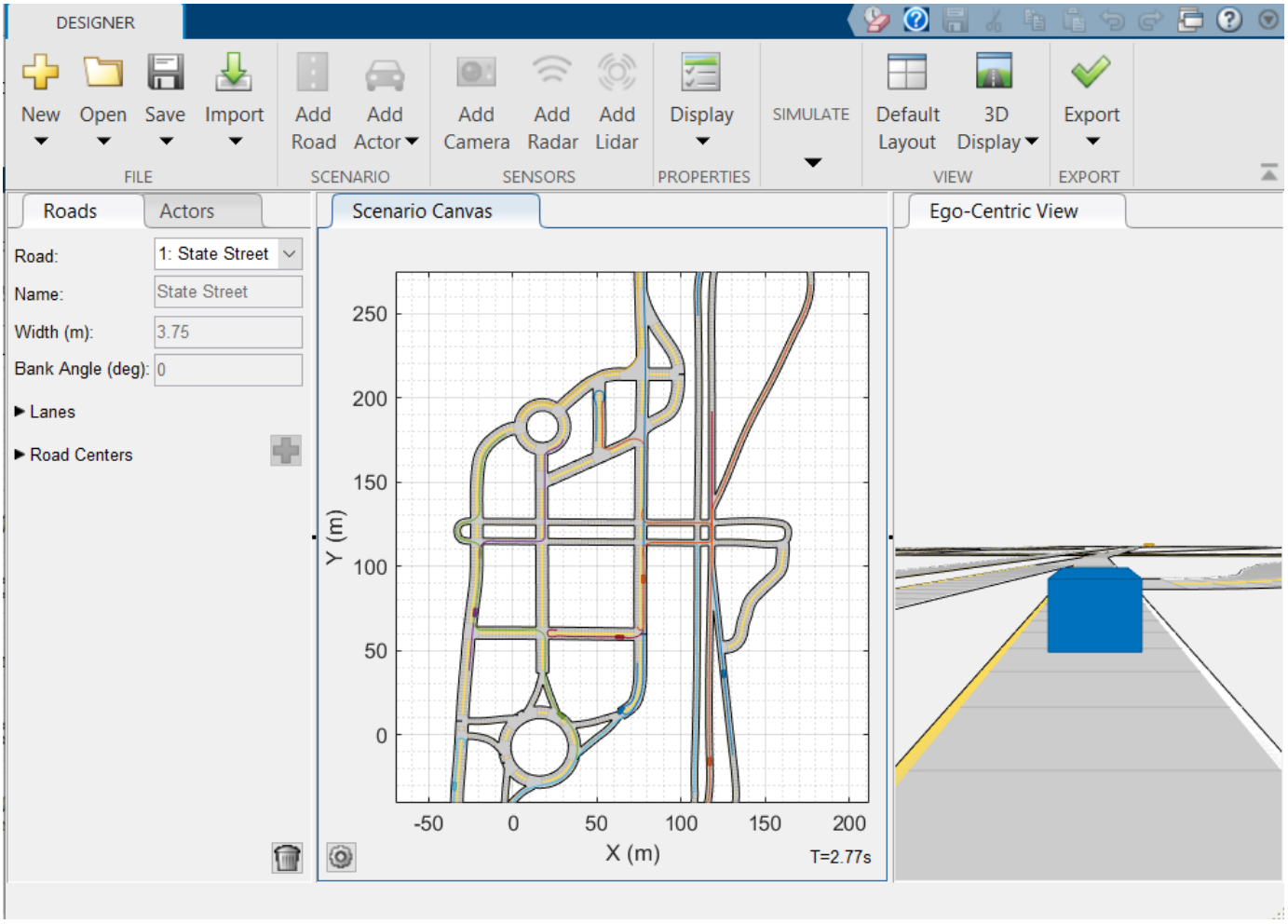
Use the `helpergetCFSMScenario` function to convert the Simulink model output to a driving scenario object. Simulate and display the driving scenario. You can see that the vehicles travel along the specified trajectories to reach their goal positions.

```
newScenario = helpergetCFSMScenario(out,scenario);
close all;
figScene = figure;
set(figScene,'Position',[0,0,600,600]);
movegui(figScene,'center');
hPanel = uipanel(figScene,'Position',[0 0 1 1]);
hPlot = axes(hPanel);
plot(newScenario,'Parent',hPlot);
title('Updated Scenario')
hold on
h1 = plot(goalPositions(:,1),goalPositions(:,2),'gs','MarkerSize',15,'LineWidth',1.2);
h2 = plot(startPositions(:,1),startPositions(:,2),'rs','MarkerSize',15,'LineWidth',1.2);
legend([h2 h1],{'Start Positions';'Goal Positions'},'Location','southoutside','Orientation','hor
hold off
% Run the simulation
while advance(newScenario)
    pause(0.01)
end
```



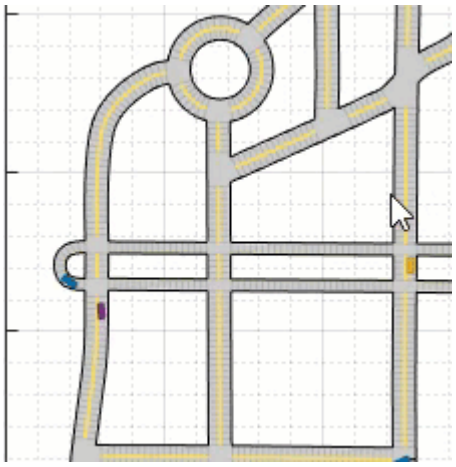
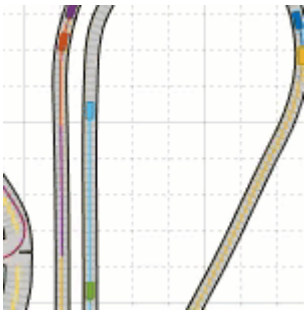
You can also export the scenario to Driving Scenario Designer app and run the simulation.

```
drivingScenarioDesigner(newScenario)
```



Tips To Avoid Collisions

The Simulink model `CollisionFreeSpeedManipulator` adjusts only the speed profile of the active vehicles. Once the vehicle reaches the goal position and becomes non-active in the scenario, it is not considered for checking collisions. If you want to generate a driving scenario with non-colliding vehicles, select points at less proximities and in different lanes as the start and the goal positions. If there is another vehicle whose goal position is close to the non-active vehicle and trajectory is same as the non-active vehicle's trajectory then there will be collision between these vehicles. Similarly, collision occurs when two vehicles travelling in same or different lanes come in close proximity at the road intersections. Also, the chances for collision is more if two or more goal positions lie in the same lane.

1  2. 

If you want to generate a driving scenario with non-colliding vehicles, select random points at less proximities and in different lanes as the start and the goal positions.

References

[1] Singh, Arun Kumar, and K. Madhava Krishna. “Reactive Collision Avoidance for Multiple Robots by Non Linear Time Scaling.” In *52nd IEEE Conference on Decision and Control*, 952–58. Firenze: IEEE, 2013. https://doi.org/10.1109/CDC.2013.6760005.

See Also

`drivingScenario` | `plot` | `roadNetwork` | `vehicle`

Related Examples

- “Create Driving Scenario Variations Programmatically” on page 5-107
- “Scenario Generation from Recorded Vehicle Data” on page 7-350

Highway Lane Following with RoadRunner Scene

This example shows how to configure and simulate a highway lane following application using a scene created in RoadRunner™ 3D scene editing tool. This example closely follows the “Highway Lane Following” on page 7-653 example.

Introduction

A highway lane following system steers a vehicle to travel within a marked lane. It also maintains a set velocity or safe distance to a preceding vehicle in the same lane. The system typically includes vision processing, sensor fusion, decision logic, and controls components. To ensure functional safety and interoperability, this system requires testing on a variety of road conditions. For example, testing on a scene that has varying shadows, lane marking types, and road materials can make it easier to identify edge cases.

RoadRunner is an interactive editor that enables you to design 3D scenes for simulating and testing automated driving systems. You can use RoadRunner to create roads, lane markings, road signs, vegetation, and scenes with varied complexities in a 3D environment.

The “Highway Lane Following” on page 7-653 example that this example is based on shows how to simulate scenarios for curved and straight road scenes. This example demonstrates how to simulate scenarios with a scene created in RoadRunner. The scene contains variations in shadows, lane marking types, and road materials designed to test the impact of the vision processing on system functionality. In this example, you will:

1. **Review scene:** Explore the scene and road segments that were created in RoadRunner.
2. **Integrate scene into driving scenario:** Export the road network from the RoadRunner scene to an OpenDRIVE® file, and then import this file into a driving scenario. Then, add a vehicle to the scenario and simulate the scenario.
3. **Integrate scene into Unreal Engine scenario:** Export the RoadRunner scene to an Unreal Engine® game and connect a Simulink® model to this scene. The imported driving scenario specifies vehicle poses. Add sensors to the vehicle and the simulate the scenario.
4. **Integrate scene into lane following application:** Using the techniques described in the previous sections, you integrate the RoadRunner scene into a scenario for highway lane following. Then, you add additional target vehicles to the scenario and simulate the system in a scenario that transitions from no shadows to shadows.
5. **Explore additional scenarios:** Simulate additional scenarios for lane marking and road type variations. Apply these techniques to your own design.

You can use the modeling patterns and techniques used in this example to import your own scenes and test your algorithms.

In this example, you enable system-level simulation through integration with the Unreal Engine from Epic Games®. This simulation environment requires a Windows® 64-bit platform.

```
if ~ispc
    error(['3D Simulation is only supported on Microsoft', char(174),...
          ' Windows', char(174), '.']);
end
```

To ensure reproducibility of the simulation results, set the random seed.

```
rng(0);
```

This example also requires you to download the Automated Driving Toolbox™ Interface for Unreal Engine 4 Projects support package.

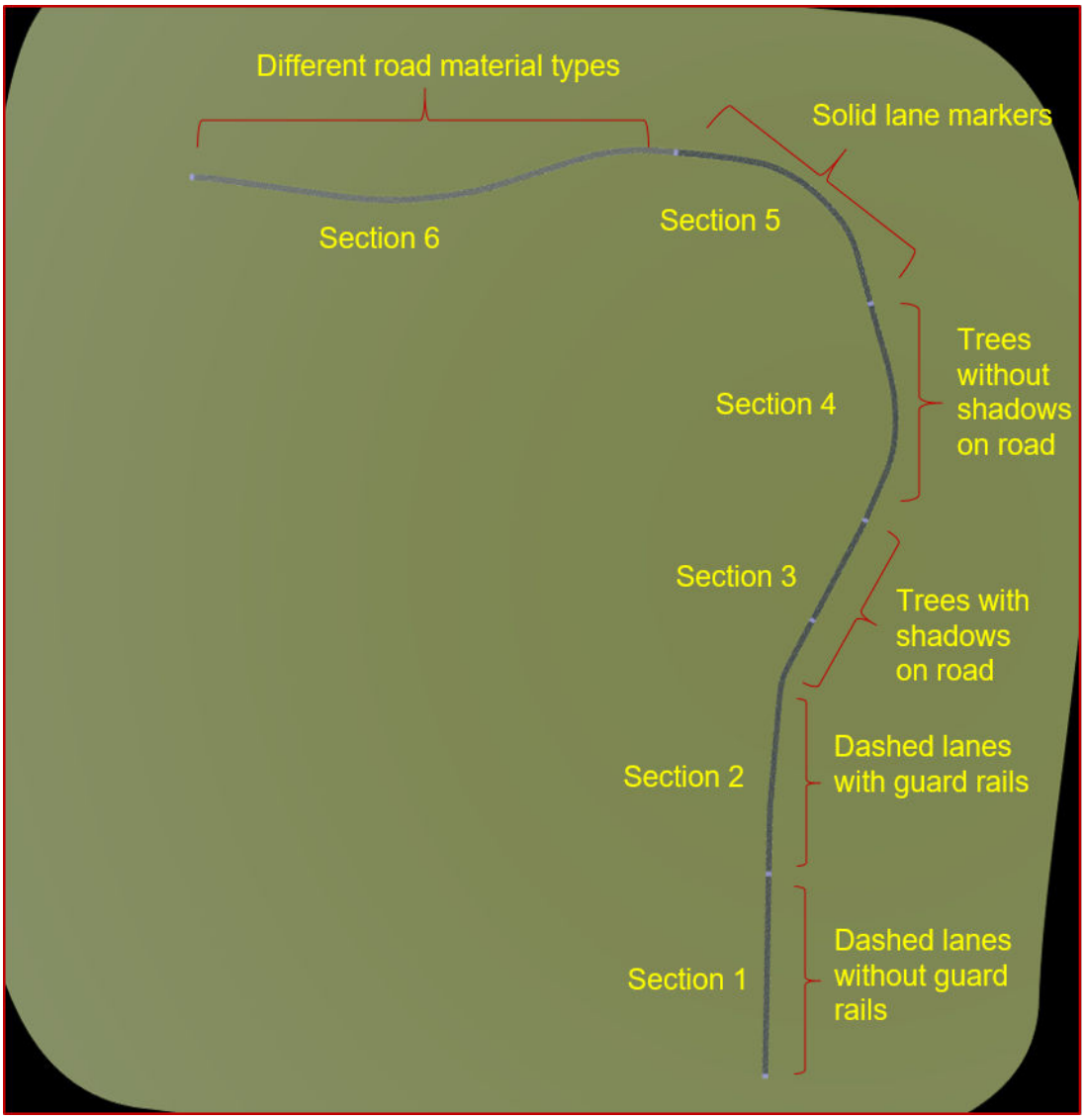
```
pathToUnrealExe = fullfile(...
    matlabshared.supportpkg.getSupportPackageRoot,...
    "toolbox","shared","sim3dprojects","driving","RoadRunnerScenes",....
    "WindowsPackage", "RRScene.exe");
if (~exist(pathToUnrealExe, 'file'))
    error('This example requires you to download and install Automated Driving Toolbox Interface
end
```

Review Scene

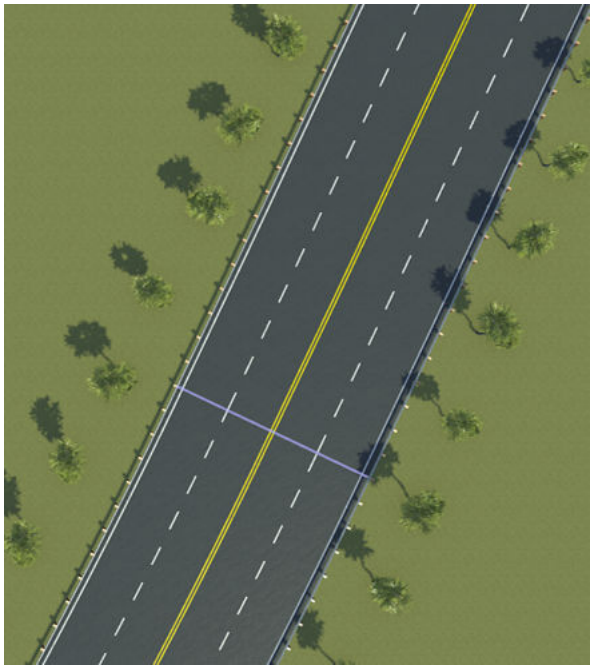
This example includes a scene (`RRHighway.rrscene`) that was designed in RoadRunner. If you have RoadRunner installed, you can follow the workflow in the “Scenes” (RoadRunner) RoadRunner topic to open the `RRHighway.rrscene` scene.

This scene is designed to pose challenges to the lane following system. This image shows that the road network is divided into six sections. Each section incrementally adds variations to the scene as follows:

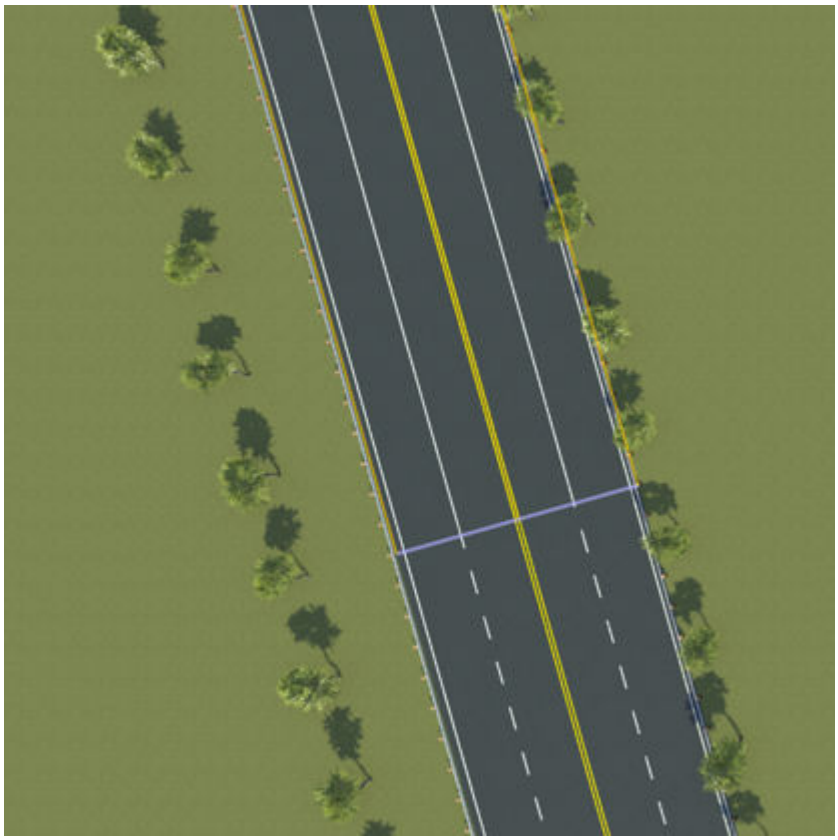
- Section 1 contains dashed lanes without guard rails.
- Section 2 contains dashed lanes with guard rails.
- Section 3 adds trees that do cast shadows on the lane markings.
- Section 4 contains trees that do not cast shadows on lane markings.
- Section 5 contains solid lane markings instead of dashed lane markings.
- Section 6 road material with lighter texture in comparison with other sections of the road.



For example, the following image shows the transition between Section-3 (with trees that do not cast shadows on the lane markings) and Section-4 (with trees that do cast shadows on lane markings).



The following image shows the transition between Section-4 (with dashed lane markings) and Section-5 (with solid lane markings).



The following image shows the transition between Section-5 (with darker road material) and Section-6 (with lighter road material).

Integrate Scene into Driving Scenario

A common motivation for importing scenes into a driving scenario is to enable the addition of vehicles and trajectories, either interactively or programmatically. You can integrate a road network from a RoadRunner scene into a driving scenario by using these steps:

1. Export OpenDRIVE file from RoadRunner.
2. Import OpenDRIVE file into driving scenario.
3. Add vehicle and trajectory to driving scenario.
4. Simulate driving scenario.

Export OpenDRIVE File from RoadRunner

RoadRunner enables you to export scenes to many file formats, include OpenDRIVE®. To learn more about the OpenDRIVE export workflow, see “Exporting to OpenDRIVE” (RoadRunner). This example includes an OpenDRIVE file (`RRHighway.xodr`) that was exported from the RoadRunner scene (`RRHighway.rrscene`) using the process described in that topic.

Import OpenDRIVE File into Driving Scenario

OpenDRIVE files can be imported into cuboid driving scenarios. To learn more about this workflow, see “Highway Trajectory Planning Using Frenet Reference Path” on page 7-500.

Create a driving scenario and import the OpenDRIVE road network. For the purposes of this example, turn off the warnings from the OpenDRIVE importer.

```
scenario = drivingScenario;
warning('off','driving:scenario:OpenDRIVEWarnings');
roadNetwork(scenario,"OpenDrive","RRHighway.xodr");
```

Add Vehicle and Trajectory to Driving Scenario

You can add vehicles to a scenario either programmatically or interactively. This example shows the programmatic workflow. To add vehicles interactively, use the **Driving Scenario Designer** app.

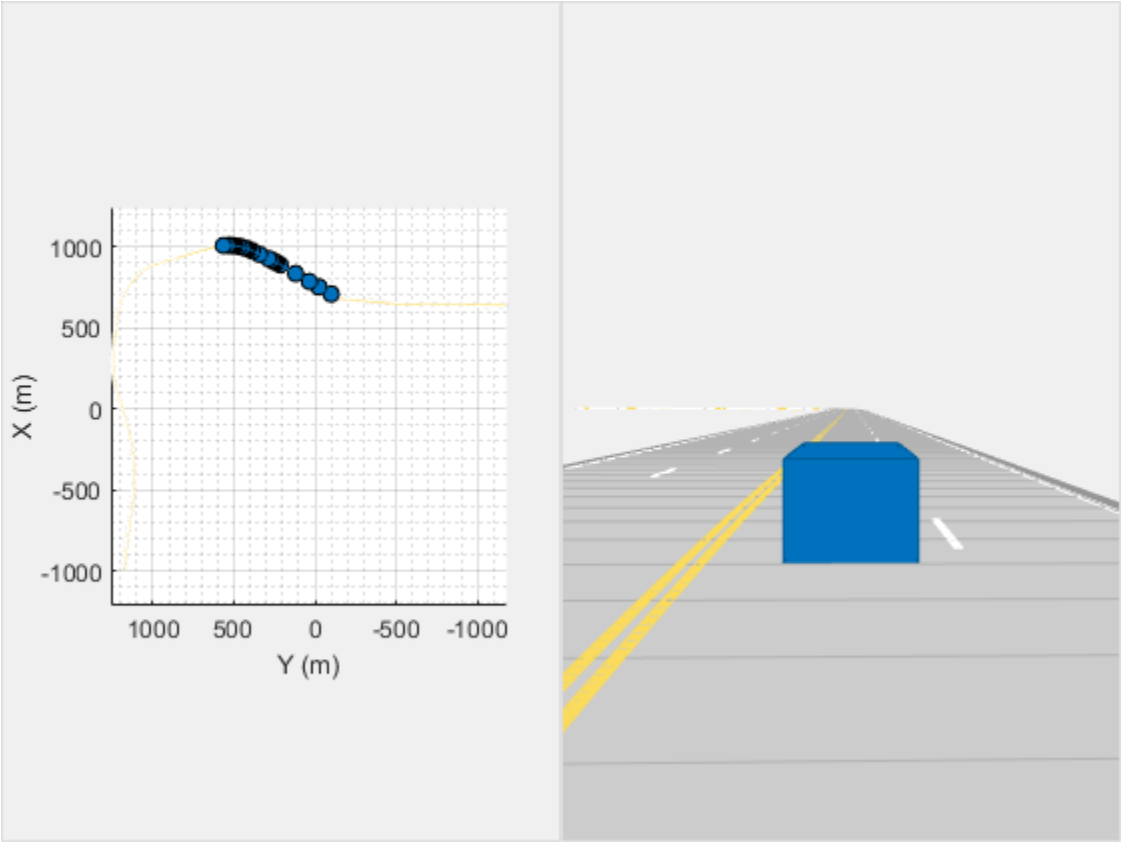
Add a vehicle to the road network using a set of predefined waypoints. These waypoints are attached as a supporting file, `manualwaypoints.mat`.

```
v = vehicle(scenario);
load("manualWaypoints.mat","waypoints");
speed = 20; % m/s
trajectory(v, waypoints, speed);
```

Simulate Driving Scenario

Plot the driving scenario in world coordinates. Also plot the scenario from the vehicle perspective by using a chase plot.

```
hFigScenario = figure;
p1 = uipanel('Position',[0 0 0.5 1]);
h1 = axes("Parent",p1);
plot(scenario,"Waypoints","On","Parent",h1);
p2 = uipanel('Position',[0.5 0 0.5 1]);
h2 = axes("Parent",p2);
chasePlot(v,"Parent",h2);
```



Set the visibility of the figure to off.

```
set(hFigScenario, 'Visible', 'Off');
```

You can optionally continue to explore, simulate, and edit the scenario in Driving Scenario Designer `drivingScenarioDesigner(scenario)`.

Integrate Scene into Unreal Engine Scenario

A common motivation for importing scenes into Unreal Engine is to enable simulation systems with camera, radar, and lidar sensor models. You can integrate a RoadRunner scene with an Unreal Engine driving scenario simulation using these steps:

1. Export Unreal Engine scene from RoadRunner.
2. Configure Unreal Engine scene.
3. Create test bench model.
4. Simulate test bench model.

Export Unreal Engine Scene from RoadRunner

RoadRunner enables exporting to Unreal Engine scene. To learn more about this workflow, see “Exporting to Unreal” (RoadRunner). The workflow includes exporting Filmbox (.fbx) and XML files, which can be imported into the Unreal Editor. After you open a scene in the Unreal Engine editor, you might want to adjust other scene aspects such as lighting.

This example uses an Unreal Engine scene (`RRHighway`) that was exported from the RoadRunner scene (`RRHighway.rrscene`).

Configure Unreal Engine Scene

The Unreal Engine scene can co-simulate with Simulink using a `MathWorksSimulation` plugin from the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package. To install the support package, follow the steps in “Install Support Package for Customizing Scenes” on page 6-45.

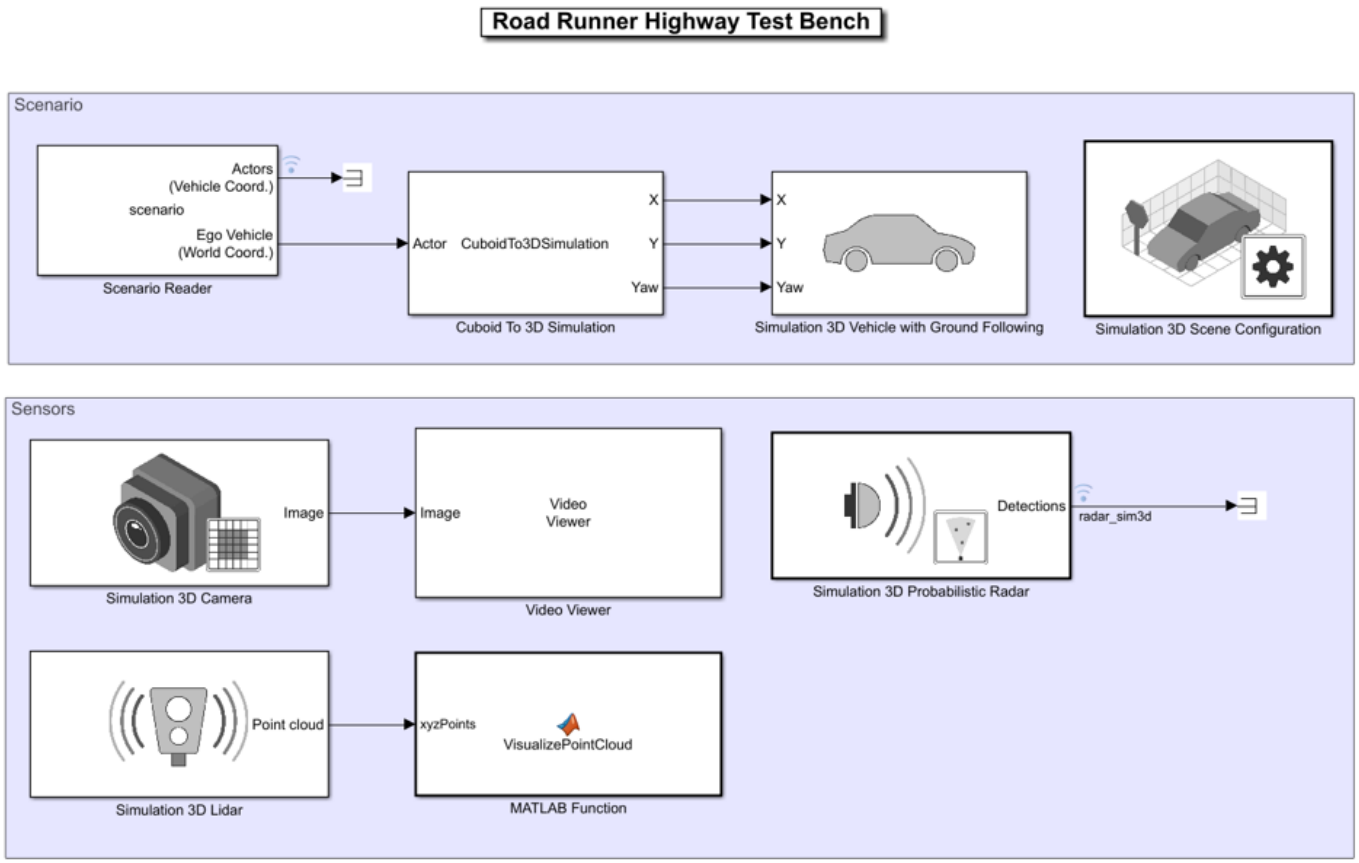
The Unreal Engine scene (`RRHighway`) used in this example has been compiled with the `MathWorksSimulation` plugin.

Create Test Bench Model

You can connect a Simulink model to the Unreal Engine for co-simulation. To learn more about this workflow, see “Simulate Simple Driving Scenario and Sensor in Unreal Engine Environment” on page 6-22. The open-loop test bench model (`RRHighwayTestBench.slx`) uses this workflow to connect to the Unreal Engine scene (`RRHighway`).

Open the test bench model for the scenario.

```
open_system("RRHighwayTestBench");
```



Copyright 2020 The MathWorks, Inc.

Opening this model runs the `helperSLRRHighwaySetup` script. This script configures the Simulation 3D Scene Configuration block in the `RRHighwayTestBench` model.

This model contains blocks that enable simulating a driving scenario with Unreal Engine.

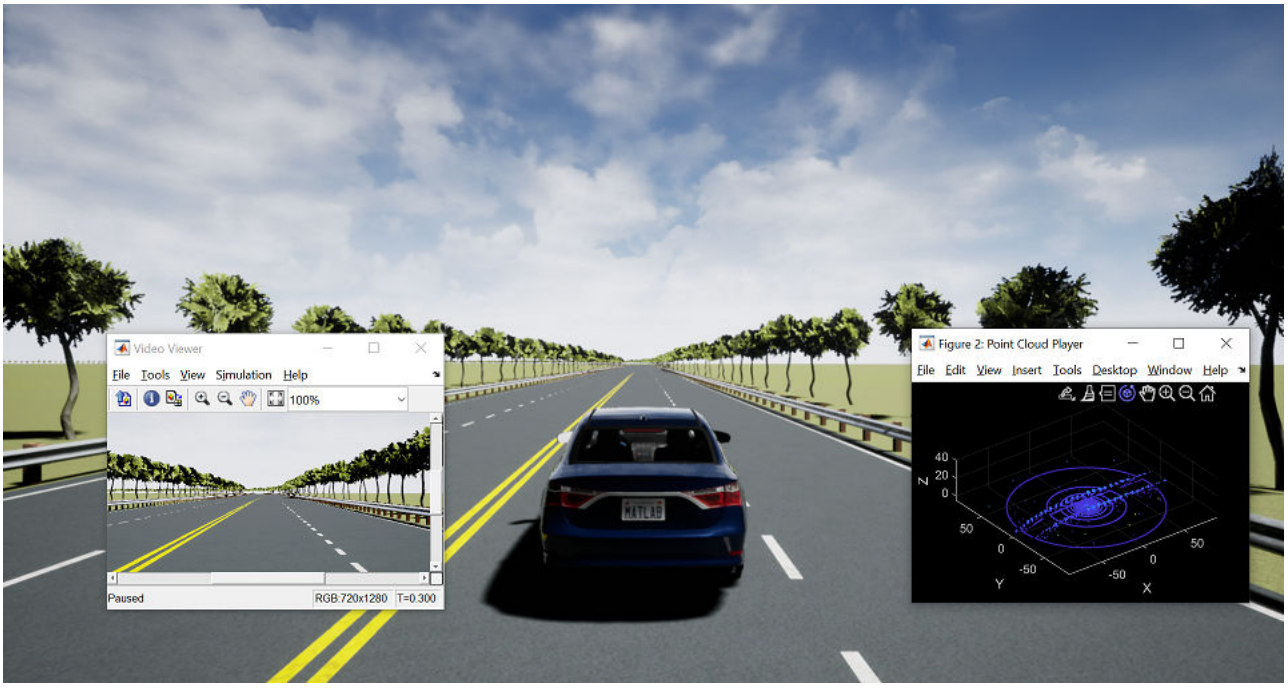
- Scenario Reader reads the driving scenario from the base workspace and outputs the ego vehicle pose.
- Simulation 3D Scene Configuration enables connection to the `RRHighway` scene.
- Simulation 3D Vehicle with Ground Following controls the pose of a vehicle in the scene
- Vehicle To World converts actor poses from the coordinates of the input ego vehicle to the world coordinates of the scenario.
- Simulation 3D Camera synthesizes camera images.
- Simulation 3D Probabilistic Radar synthesizes radar detections.
- Simulation 3D Lidar synthesizes lidar point cloud data.

The model also contains blocks to visualize the camera and lidar sensors. You can use the Bird's-Eye Scope to visualize radar and vision detections from a bird's-eye view. To learn how to configure this scope, see “Visualize Sensor Data from Unreal Engine Simulation Environment” on page 6-36.

Simulate Test Bench Model

Simulate the model. The vehicle follows the trajectory defined in the driving scenario.

```
sim("RRHighwayTestBench");
```

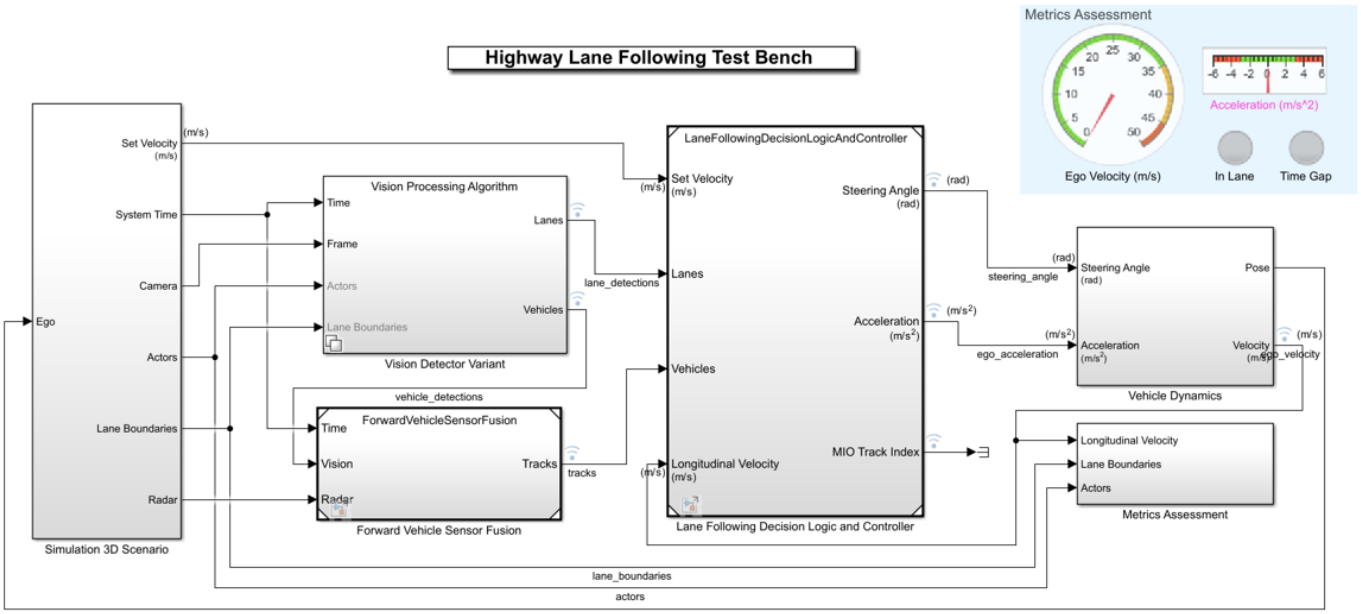


Integrate Scene into Lane Following Application

You can reuse the techniques described in the previous sections to simulate and assess a closed-loop system such as the highway lane following application. This section reuses models, helper functions,

and techniques described in the “Highway Lane Following” on page 7-653 example. Open and configure the system test bench model.

```
open_system("HighwayLaneFollowingTestBench");
scenarioFcnName = "scenario_RRHighway_01_NoShadowToShadow";
visionVariant = "VisionProcessingAlgorithm";
helperSLHighwayLaneFollowingSetup(scenarioFcnName,visionVariant);
```



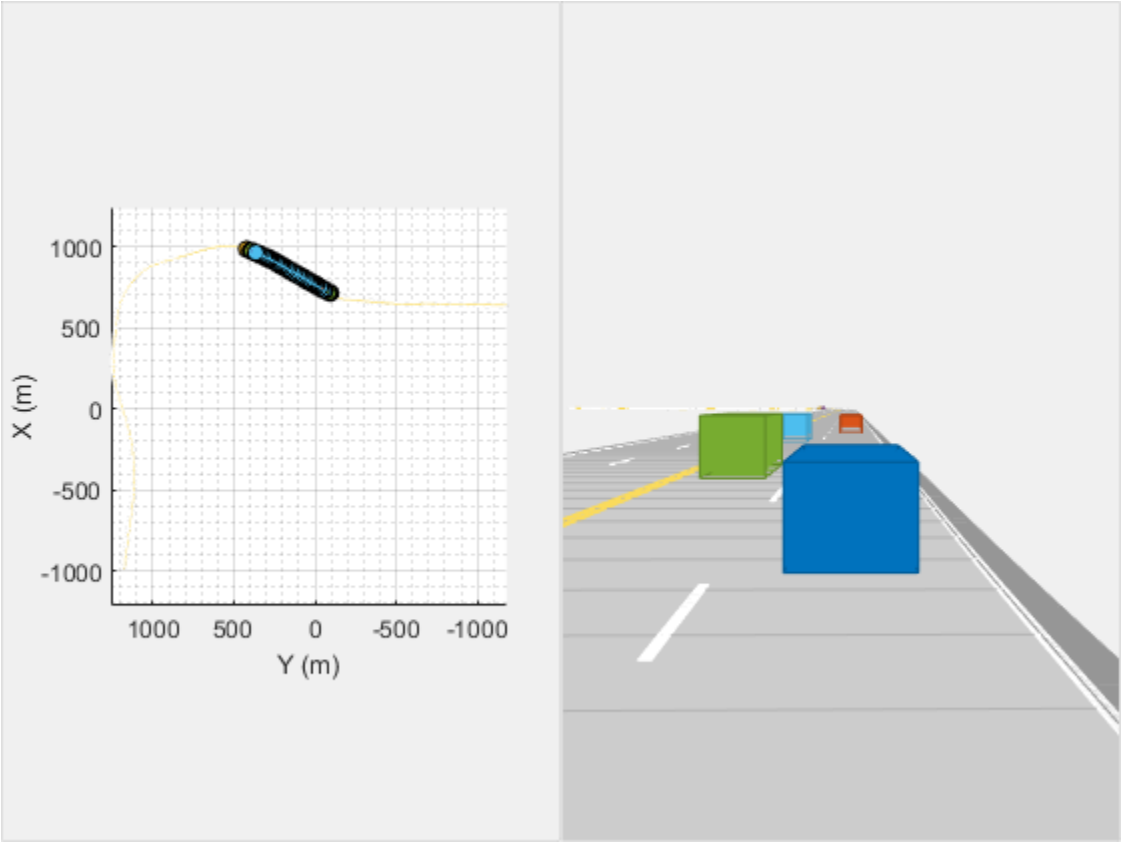
Copyright 2019-2020 The MathWorks, Inc.

Integrate Scene into Driving Scenario

The first argument to the `helperSLHighwayLaneFollowingSetup` is the name of a function that creates a driving scenario that is compatible with the `HighwayLaneFollowingTestBench`. The `scenario_RRHighway_01_NoShadowToShadow` function creates this driving scenario. It imports the `RRHighway.xodr` OpenDRIVE file and adds vehicles around the road segment that transitions from no shadow to shadow. You can explore this function to learn more about programmatic techniques for creating scenarios. It leverages several helper functions that you can use to simplify adding vehicle and their trajectories to the imported road network. Explore the `scenario_RRHighway_01_NoShadowToShadow` to know more about the helper functions and their usage.

The `helperSLHighwayLaneFollowingSetup` function creates a variable scenario in the base workspace. Plot this scenario and notice the region of the scene that will be simulated.

```
hFigScenario = figure;
p1 = uipanel('Position',[0 0 0.5 1]);
h1 = axes("Parent",p1);
plot(scenario,"Waypoints","On","Parent",h1);
p2 = uipanel('Position',[0.5 0 0.5 1]);
h2 = axes("Parent",p2);
chasePlot(scenario.Actors(1),"Parent",h2)
```



Set the visibility of the figure to off.

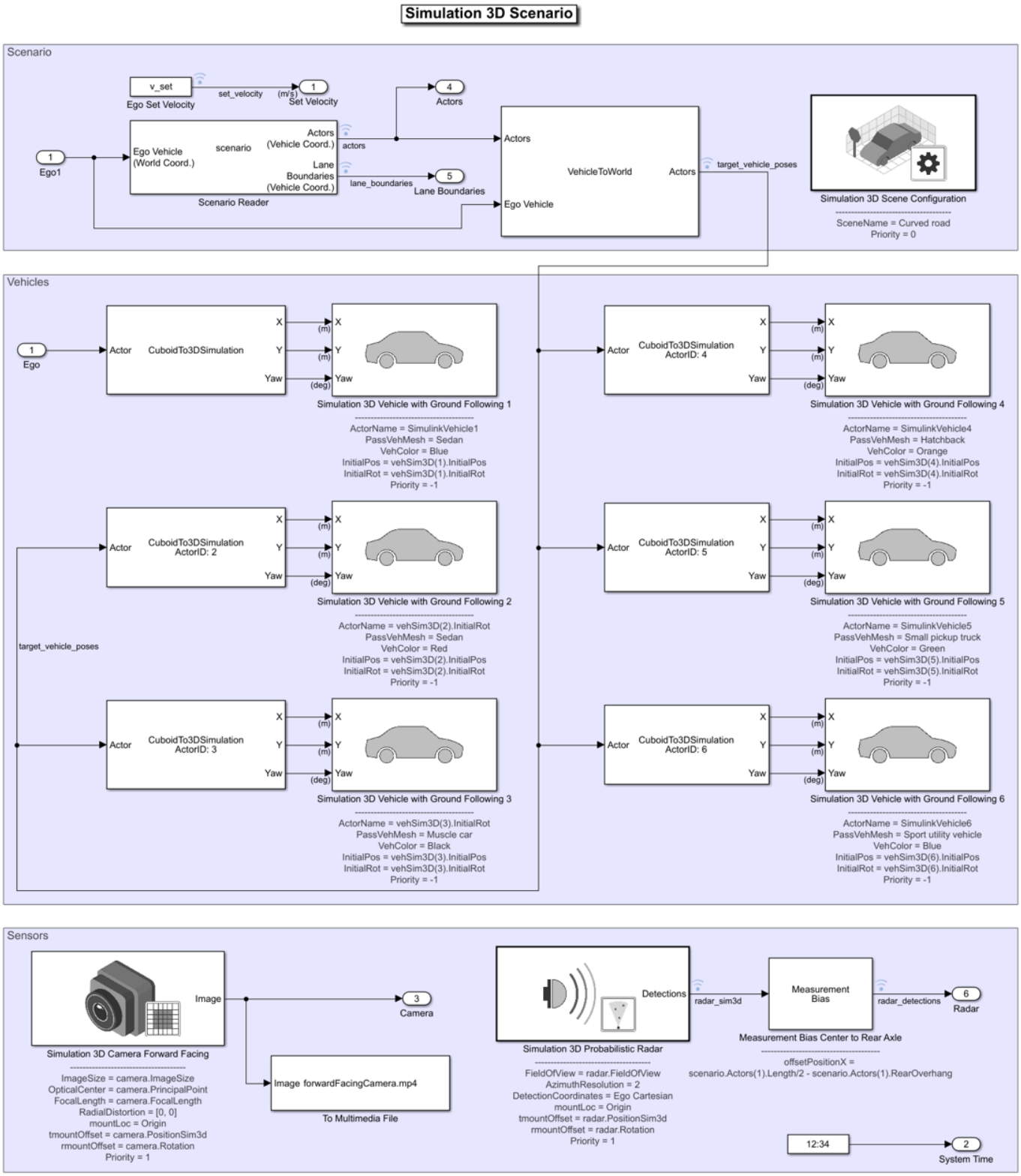
```
set(hFigScenario, 'Visible', 'Off');
```

Integrate Scene into Unreal Engine Scenario

The `HighwayLaneFollowingTestBench` model contains algorithm components for vision detection, forward vehicle sensor fusion, and controls. The Simulation 3D Scenario subsystem integrates the model with the driving scenario and the corresponding Unreal Engine game.

Open the Simulation 3D Scenario subsystem.

```
open_system("HighwayLaneFollowingTestBench/Simulation 3D Scenario")
```


Simulation 3D Scenario
Scenario
v_set
Ego Set Velocity
set_velocity
(m/s)
1
Set Velocity
4
Actors
1
Ego1
Ego Vehicle
(World Coord.)
scenario
Actors
(Vehicle Coord.)
actors
Lane
Boundaries
(Vehicle Coord.)
lane_boundaries
5
Lane Boundaries
Scenario Reader
Actors
VehicleToWorld
Actors
Ego Vehicle
target_vehicle_poses
Simulation 3D Scene Configuration
SceneName = Curved road
Priority = 0
Vehicles
1
Ego
Actor
CuboidTo3DSimulation
X
Y
Yaw
(m)
(deg)
Simulation 3D Vehicle with Ground Following 1
ActorName = SimulinkVehicle1
PassVehMesh = Sedan
VehColor = Blue
InitialPos = vehSim3D(1).InitialPos
InitialRot = vehSim3D(1).InitialRot
Priority = -1
CuboidTo3DSimulation
ActorID: 4
Simulation 3D Vehicle with Ground Following 4
ActorName = SimulinkVehicle4
PassVehMesh = Hatchback
VehColor = Orange
InitialPos = vehSim3D(4).InitialPos
InitialRot = vehSim3D(4).InitialRot
Priority = -1
CuboidTo3DSimulation
ActorID: 2
Simulation 3D Vehicle with Ground Following 2
ActorName = vehSim3D(2).InitialRot
PassVehMesh = Sedan
VehColor = Red
InitialPos = vehSim3D(2).InitialPos
InitialRot = vehSim3D(2).InitialRot
Priority = -1
CuboidTo3DSimulation
ActorID: 5
Simulation 3D Vehicle with Ground Following 5
ActorName = SimulinkVehicle5
PassVehMesh = Small pickup truck
VehColor = Green
InitialPos = vehSim3D(5).InitialPos
InitialRot = vehSim3D(5).InitialRot
Priority = -1
target_vehicle_poses
CuboidTo3DSimulation
ActorID: 3
Simulation 3D Vehicle with Ground Following 3
ActorName = vehSim3D(3).InitialRot
PassVehMesh = Muscle car
VehColor = Black
InitialPos = vehSim3D(3).InitialPos
InitialRot = vehSim3D(3).InitialRot
Priority = -1
CuboidTo3DSimulation
ActorID: 6
Simulation 3D Vehicle with Ground Following 6
ActorName = SimulinkVehicle6
PassVehMesh = Sport utility vehicle
VehColor = Blue
InitialPos = vehSim3D(6).InitialPos
InitialRot = vehSim3D(6).InitialRot
Priority = -1
Sensors
Image
3
Camera
Simulation 3D Camera Forward Facing
ImageSize = camera.ImageSize
OpticalCenter = camera.PrincipalPoint
FocalLength = camera.FocalLength
RadialDistortion = [0, 0]
mountLoc = Origin
tmountOffset = camera.PositionSim3d
rmountOffset = camera.Rotation
Priority = 1
Image forwardFacingCamera.mp4
To Multimedia File
Detections
radar_sim3d
Measurement
Bias
radar_detections
6
Radar
Measurement Bias Center to Rear Axle
offsetPositionX =
scenario.Actors(1).Length/2 - scenario.Actors(1).RearOverhang
Simulation 3D Probabilistic Radar
FieldOfView = radar.FieldOfView
AzimuthResolution = 2
DetectionCoordinates = Ego Cartesian
mountLoc = Origin
tmountOffset = radar.PositionSim3d
rmountOffset = radar.Rotation
Priority = 1
12:34
2
System Time

Notice that this subsystem reuses the modeling techniques and blocks described previously in this example.

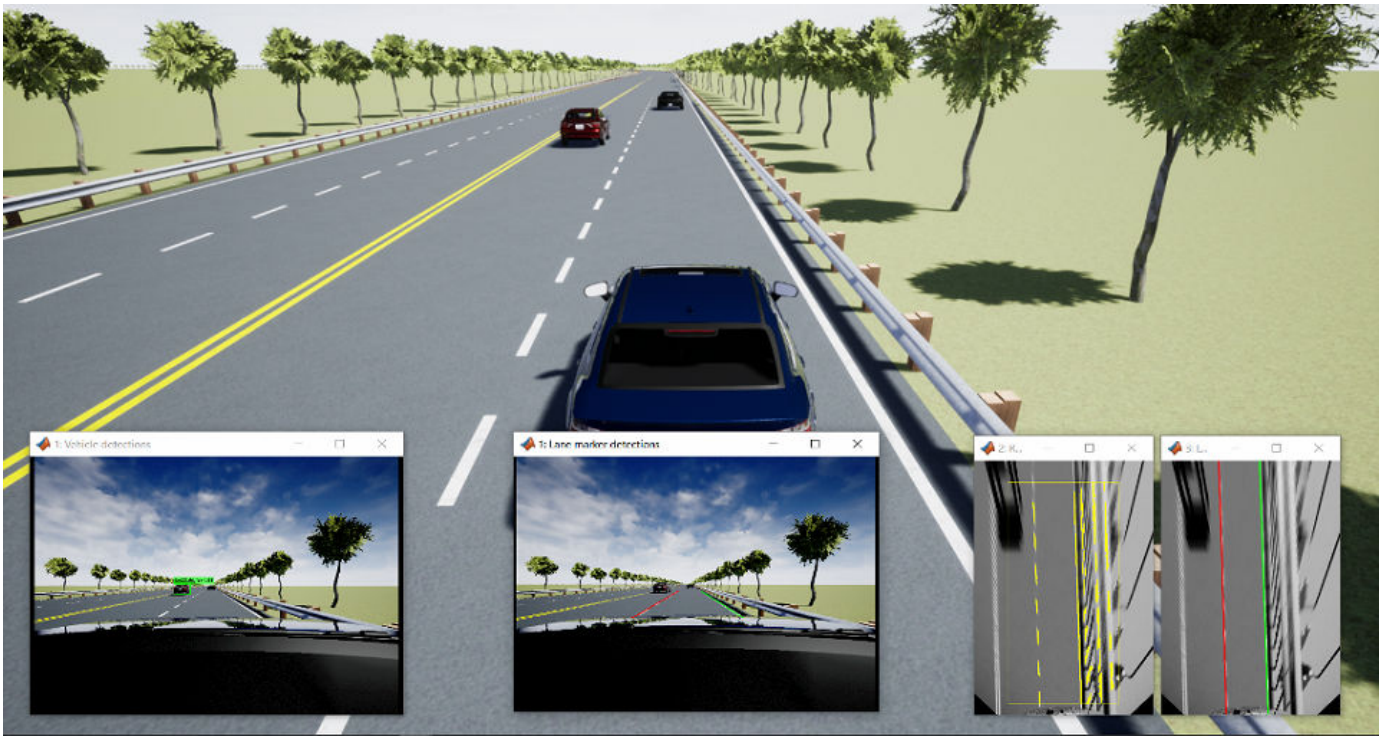
The primary differences are:

- The ego vehicle is under closed-loop control.
- Target vehicles are added to the scenario.
- The lane following example does not use a lidar sensor.

Simulate the model to see the ego vehicle behavior for the scenario.

```
mpcverbosity('off');
sim("HighwayLaneFollowingTestBench");
```

The system is able to detect and follow lanes in the conditions where shadows are present. For more details on how to analyze the simulation results, refer to the “Highway Lane Following” on page 7-653 example.



Close the figure.

```
close(hFigScenario);
```

Explore Additional Scenarios

This example provides additional scenarios that you can use to test the system behavior.

- `scenario_RRHighway_02_DashedToSolidMarkings` function configures the test scenario such that the ego vehicle navigates from Section 4 to Section 5 of the scene. This enables testing the lane following application for the transition from dashed lane markings to solid lane markings.

- `scenario_RRHighway_03_DarkToLightRoadMaterial` function configures the test scenario such that the ego vehicle navigates from Section 5 to Section 6 of the scene. This enables testing the lane following application for the transition from darker textured road material to lighter textured road material.

You can configure the model and workspace with these scenarios by using the `helperSLHighwayLaneFollowingSetup` function. For example, this code configures the test bench to simulate a scenario in the region where the road material changes.

```
helperSLHighwayLaneFollowingSetup(...
    "scenario_RRHighway_03_DarkToLightRoadMaterial",...
    "VisionProcessingAlgorithm");
```

You can apply these techniques to integrate RoadRunner scenes into driving scenarios for simulation and testing of your systems.

Enable the Model Predictive Controller update messages and turn back on warnings from the OpenDRIVE importer.

```
mpcverbosity('on');
warning('on','driving:scenario:OpenDRIVEWarnings');
```

See Also

Blocks
Scenario Reader | Simulation 3D Camera | Simulation 3D Probabilistic Radar | Simulation 3D Probabilistic Radar | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following

Functions
`drivingScenario` | `roadNetwork` | `vehicle`

Apps
Driving Scenario Designer

More About

- “Highway Lane Following” on page 7-653
- “Highway Lane Following with Intelligent Vehicles” on page 7-779

Traffic Light Negotiation with Unreal Engine Visualization

This example shows how to design and simulate a vehicle to negotiate traffic lights in the Unreal Engine® driving simulation environment.

Introduction

Decision logic for negotiating traffic lights is a fundamental component of automated driving applications. The decision logic interacts with a controller to steer the ego vehicle based on the state of the traffic light and other vehicles in the ego lane. Simulating real-world traffic scenarios with realistic conditions can provide more insight into the interactions between the decision logic and the controller. Automated Driving Toolbox™ provides a 3D simulation environment powered by Unreal Engine® from Epic Games®. You can use this engine to visualize the motion of a vehicle in a prebuilt 3D scene. This engine provides an intuitive way to analyze the performance of decision logic and control algorithms when negotiating a traffic light at an intersection.

For information on how to design the decision logic and controls for negotiating traffic lights in a cuboid environment. See the “Traffic Light Negotiation” on page 7-677 example. This example shows how to control a traffic light in an Unreal scene and then how to simulate and visualize vehicle behavior for different test scenarios. In this example, you will:

1. **Explore the architecture of the test bench model**: The model contains sensors and environment, traffic light decision logic, controls, and vehicle dynamics.
2. **Control traffic light in an Unreal scene**: The `Traffic Light Controller` helper block configures the model to control the state of a traffic light in an Unreal scene by using Simulink®.
3. **Simulate vehicle behavior during green to red transition**: The model analyzes the interactions between the decision logic and the controller when the traffic light state transitions from green to red and the ego vehicle is at a distance of 10 meters from the stop line.
4. **Simulate vehicle behavior during red to green transition**: The model analyzes the interactions between the decision logic and the controller when the traffic light transitions from red to green and the ego vehicle is at a distance of 11 meters from stop line. In this case, the ego vehicle also negotiates traffic light as another vehicle crosses the intersection.
5. **Explore other scenarios**: These scenarios test the system under additional conditions.

You can apply the modeling patterns used in this example to test your own decision logic and controls to negotiate traffic lights in an Unreal scene.

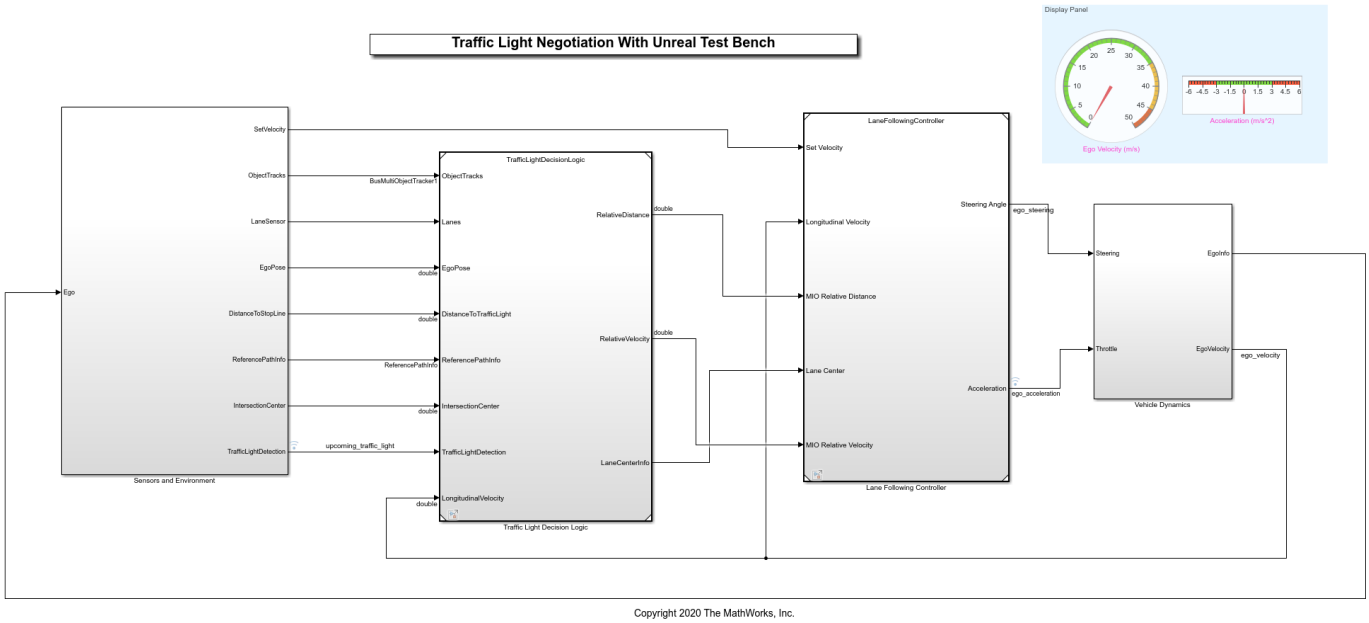
In this example, you enable system-level simulation through integration with the Unreal Engine. This environment requires a Windows® 64-bit platform.

```
if ~ispc
    error(['3D Simulation is only supported on Microsoft', char(174), ' Windows', char(174), '.'])
end
```

Explore Architecture of Test Bench Model

To explore the behavior of the traffic light negotiation system, open the simulation test bench model for the system.

```
open_system("TrafficLightNegotiationWithUnrealTestBench");
```



Copyright 2020 The MathWorks, Inc.

Opening this model runs the `helperSLTrafficLightNegotiationWithUnrealSetup` script to initialize the test scenario stored as a `drivingScenario` object in the base workspace. The default test scenario, `scenario_TLN_straight_greenToRed_with_lead_vehicle`, contains one ego vehicle and two non-ego vehicles. This setup script also configures the controller design parameters, vehicle model parameters, and Simulink® bus signals to define the inputs and outputs for the `TrafficLightNegotiationWithUnrealTestBench` model.

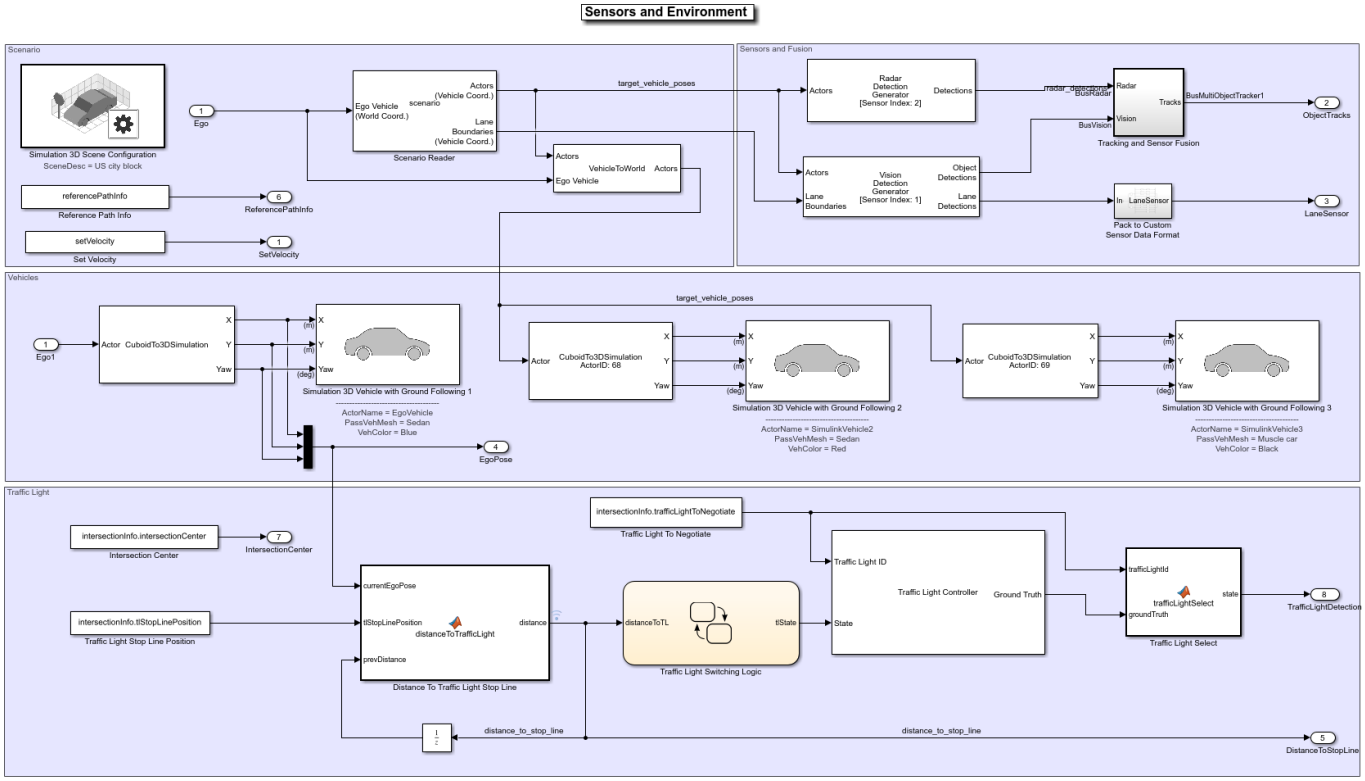
The test bench model contains the following subsystems:

1. **Sensors and Environment**: Models the road network, vehicles, camera, and radar sensors used for simulation. The subsystem uses the `Traffic Light Controller` helper block to control the state of traffic lights in an Unreal scene.
2. **Traffic Light Decision Logic**: Arbitrates between the traffic light and other lead vehicles or cross-traffic vehicles at the intersection.
3. **Lane-Following Controller**: Generates longitudinal and lateral controls for the ego vehicle.
4. **Vehicle Dynamics**: Models the ego vehicle using a Bicycle Model block and updates its state using commands received from the **Lane Following Controller** reference model.

The **Traffic Light Decision Logic**, **Lane Following Controller** reference models, and **Vehicle Dynamics** subsystem are reused from the “Traffic Light Negotiation” on page 7-677 example. This example modifies the **Sensors and Environment** subsystem to make it compatible for simulation with an Unreal scene.

The **Sensors and Environment** subsystem configures the road network, sets vehicle positions, synthesizes sensors, and fuses the vehicle detections from the radar and vision sensors. Open the **Sensors and Environment** subsystem.

```
open_system("TrafficLightNegotiationWithUnrealTestBench/Sensors and Environment");
```



Select Scenario

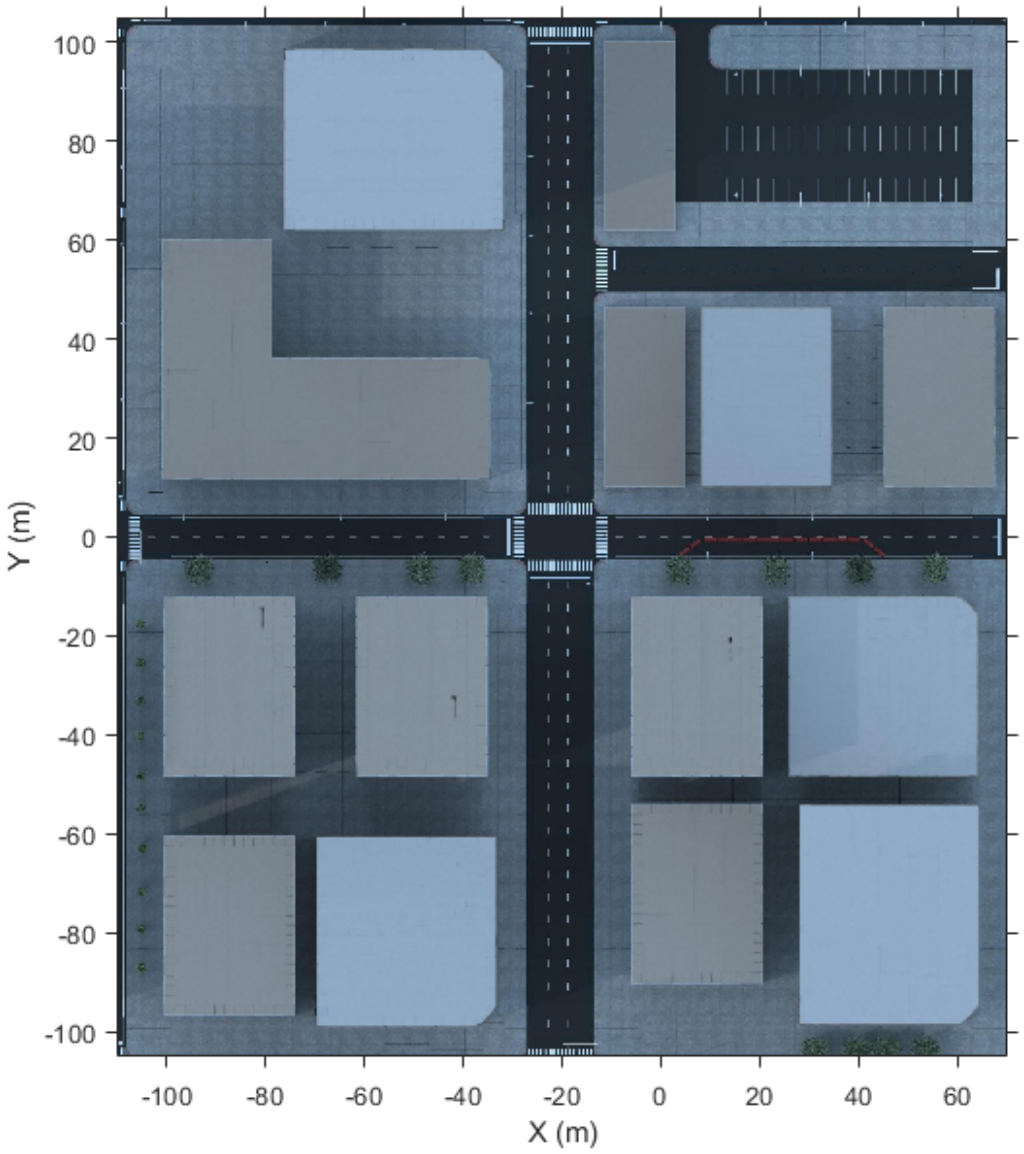
The scene and road network required for the test bench model are specified by the following parts of this subsystem:

- The scene name parameter `Scene name` of the Simulation 3D Scene Configuration block is set to US City Block. The US city block road network consists of fifteen one-way intersections with two traffic lights at each intersection. This example uses a section of the US city block scene to test the model.
- The Scenario Reader block takes the ego vehicle information as input and performs a closed-loop simulation. This block reads the `drivingScenario` object `scenario` from the base workspace. The scenario contains the desired road network. The road network closely matches with a section of the US city block scene and contains one intersection.

You can display the selected section of the US city block scene by using the `helperDisplayTrafficLightScene` function.

Specify the x and the y limits to select the desired scene area and plot the extracted scene.

```
xlimit = [-110 70];
ylimit = [-105 105];
hFigure = helperDisplayTrafficLightScene(xlimit, ylimit);
snapnow;
close(hFigure);
```



The `helperGetTrafficLightScenario` function specifies a reference path for the ego vehicle to follow when the lane information is not available. The **Reference Path Info** block reads the reference path stored in the base workspace variable `referencePathInfo`. The ego vehicle can either go straight or take a left turn at the intersection based on the reference trajectory. You can select one of these reference trajectories by setting the input values of `helperGetTrafficLightScenario` function. Set the value to

- `Straight` - To make the ego vehicle travel straight through the intersection.
- `Left` - To make the ego vehicle take a left turn at the intersection.

The **Set Velocity** block reads the velocity value from the base workspace variable `setVelocity` and gives as input to the controller.

Set Vehicle Positions

The scenario contains one ego vehicle and two non-ego vehicles. The position for each vehicle in the scenario are specified by these parts of the subsystem:

- The Simulation 3D Vehicle with Ground Following block provides an interface that changes the position and orientation of the vehicle in the 3D scene.
- The **Ego** input port controls the position of the ego vehicle, which is specified by the `Simulation 3D Vehicle with Ground Following` 1 block. The `ActorName` mask parameter of `Simulation 3D Vehicle with Ground Following` 1 block is specified as `EgoVehicle`.
- The Cuboid To 3D Simulation block converts the ego pose coordinate system (with respect to below the center of the vehicle rear axle) to the 3D simulation coordinate system (with respect to below the vehicle center).
- The Scenario Reader block also outputs ground truth information of lanes and actor poses in ego vehicle coordinates for the target vehicles. There are two target vehicles in this example, which are specified by the other Simulation 3D Vehicle with Ground Following blocks.
- The Vehicle To World block converts the actor pose coordinates from ego vehicle coordinates to the world coordinates.

The **Tracking and Sensor Fusion** subsystem fuses vehicle detections from Radar Detection Generator and Vision Detection Generator blocks and tracks the fused detections using Multi-Object Tracker block to provide object tracks surrounding the ego vehicle. The Vision Detection Generator block also provides lane detections with respect to the ego vehicle that helps in identifying vehicles present in the ego lane.

Control Traffic Light in Unreal Scene

This model uses the **Traffic Light Controller** helper block to configure and control the state of traffic lights in an Unreal scene. The **Traffic Light Controller** helper block controls the state of traffic lights by using `Timer-Based` or `State-Based` mode. You can select the desired mode by using the `Control mode` mask parameter. By default, this model uses `State-Based` mode. For information on `Timer-Based` mode, see the block mask description.

In `State-Based` mode, the block overwrites the state of a traffic light specified by the `Traffic Light ID` input port. The value for the `Traffic Light ID` input port is set by the `intersectionInfo.trafficLightToNegotiate` variable in the `helperGetTrafficLightScenario` function. In this model, the value for `Traffic Light ID` input port is set to 16. This implies that the block controls the traffic light with ID value 16 in the US city block scene. The states of all the traffic lights present in the US city block scene is returned by the `Ground Truth` output port of the `Traffic Light Controller` helper block. The model tests the decision logic and controls by using the ground truth information and does not require perception-based traffic light detection.

The `Traffic Light Select` block extracts the state of the traffic light with ID value 16 from the `Ground Truth` output. The **Traffic Light Decision Logic** reference model uses the state value to arbitrate between the lead car and the traffic light. For more information about the **Traffic Light Decision Logic** reference model, see the “Traffic Light Negotiation” on page 7-677 example.

The **Traffic Light Stop Line Position** block provides the stop line position at the intersection corresponding to the selected traffic light `trafficLightToNegotiate`. The stop line position value is specified by `intersectionInfo.tlStopLinePosition`.

The **Intersection Center** block provides the position of the intersection center of the road network in the scenario. This is obtained using the `intersectionInfo`, an output from `helperGetTrafficLightScenario`.

It is often important to test the decision logic and controls when the ego vehicle is close to the traffic light and the traffic light changes its state. The model used in this example enables traffic lights to change state when the `EgoVehicle` is close to the traffic light.

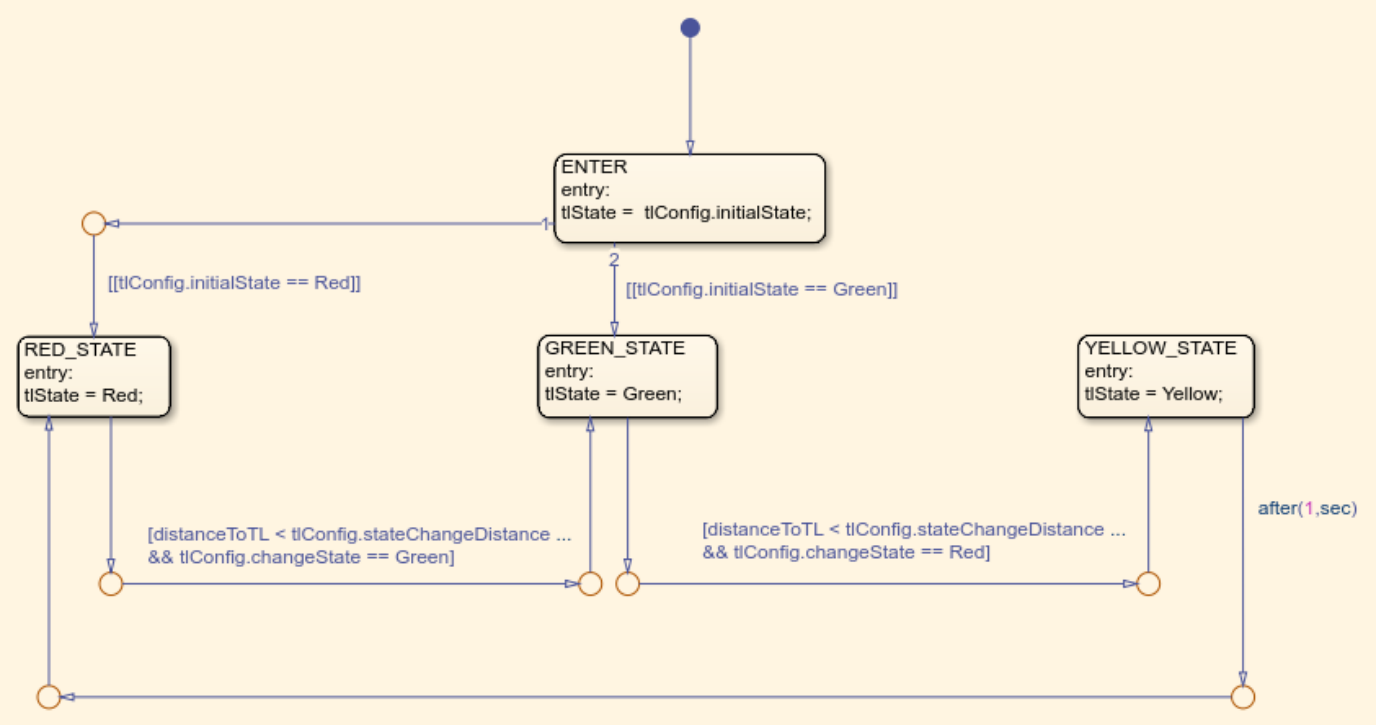
The `Distance To Traffic Light Stop Line` block calculates the Euclidean distance between the stop line corresponding to the selected traffic light `trafficLightToNegotiate` and the current ego vehicle position.

The **Traffic Light Decision Logic** uses the distance value to decide the most important object (MIO), the closest object in front of the ego vehicle. It can be the lead vehicle or traffic light in the ego lane.

The **Traffic Light Switching Logic** block outputs `tlState`, the state of the traffic light that needs to be set. This is implemented using Stateflow™ and uses the distance value to trigger a state change when the `EgoVehicle` is closer to the traffic light than the specified distance.

Open the **Traffic Light Switching Logic** block.

```
open_system("TrafficLightNegotiationWithUnrealTestBench/Sensors and Environment/Traffic Light Sw
```



Traffic Light Switching Logic uses the `Configuration params` mask parameter to read the traffic light configuration, `trafficLightConfig`, from the base workspace. You can use the `trafficLightConfig` structure to configure different test scenarios. This structure is defined in the test scenario function and has the following fields: `stateChangeDistance`, `initialState`, and `changeState`.

- `initialState` specifies the state of the traffic light before the state change.
- `stateChangeDistance` specifies the threshold distance of the `EgoVehicle` to the traffic light at which state change should happen.
- `changeState` specifies the state of the traffic light to be set after state change.

State switching happens based on the set configuration and when `EgoVehicle` reaches `stateChangeDistance`. When the `initialState` is `Red` and `changeState` is `Green` the Stateflow chart switches from `Red` state to `Green` state. Conversely, when the `initialState` is `Green` and `changeState` is `Red` the Stateflow chart is modeled such that the state transition happens from `Green` state to `Yellow` state and after one second, the traffic light switches to `Red` state.

Simulate Vehicle Behavior During Green To Red Transition

This section tests the decision logic when the ego vehicle is at a close distance to the traffic light and the traffic light state changes from green to red. In this test scenario, a lead vehicle travels in the ego lane and crosses the intersection. The traffic light state keeps green for the lead vehicle and turns red when the ego vehicle is at a distance of 10 meters from the stop line. The ego vehicle is expected to follow the lead vehicle, negotiate the state transition, and come to a complete halt before the stop line.

Configure the `TrafficLightNegotiationWithUnrealTestBench` model to use the `scenario_TLN_straight_greenToRed_with_lead_vehicle` test scenario.

```
helperSLTrafficLightNegotiationWithUnrealSetup(...
    "scenario_TLN_straight_greenToRed_with_lead_vehicle");
```

Display the `trafficLightConfig` structure parameters set for the test scenario.

```
disp(trafficLightConfig');
```

```
           initialState: 2
    stateChangeDistance: 10
            changeState: 0
```

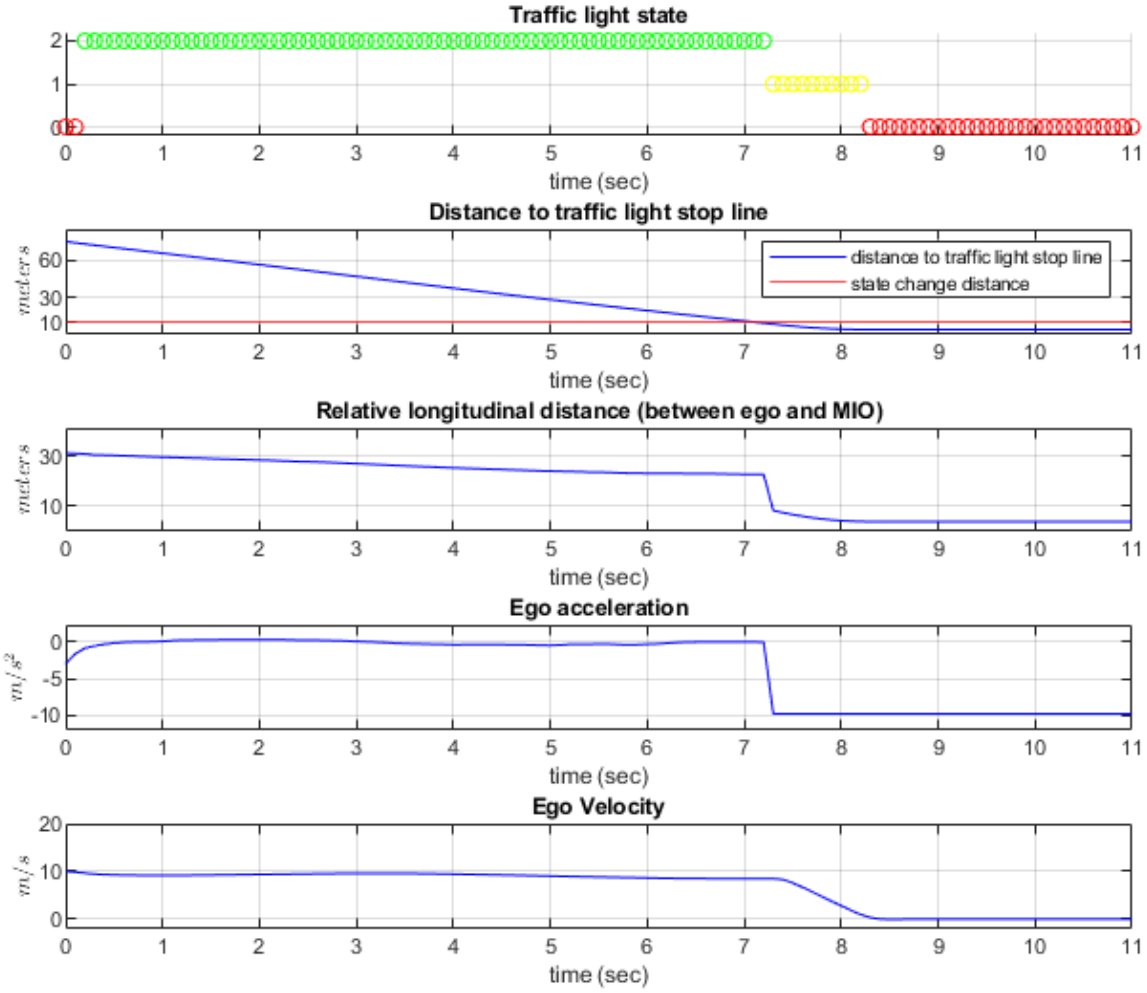
Simulate the model. During the simulation, the model logs the signals required for post simulation analysis to `logsout`.

To reduce command-window output, first turn off the MPC update messages.

```
mpcverbosity('off');
sim("TrafficLightNegotiationWithUnrealTestBench");
```

Plot the simulation results using `helperPlotTrafficLightControlAndNegotiationResults` function.

```
hFigResults = helperPlotTrafficLightControlAndNegotiationResults(logsout, trafficLightConfig.sta
```



Examine the results.

- The **Traffic light state** plot shows the state of the traffic light. The **Distance to traffic light stop line** plot shows the distance between the ego vehicle and the stop line corresponding to the traffic light. You can see that the initial state of the traffic light is green and the state changes from green to yellow as the ego vehicle approaches the stop line. The state changes from yellow to red when the ego vehicle is at a distance of 10 meters from the stop line.
- The **Relative longitudinal distance** plot shows the relative distance between the ego vehicle and the most important object (MIO). The MIO is the closest object in front of the ego vehicle. It can be a lead vehicle or a traffic light in the ego lane. The ego vehicle follows the lead vehicle and maintains a safe distance when the traffic light state is green. The distance between the ego and the lead vehicle decreases when the traffic light transitions from green to red. This is because, as the ego vehicle approaches the stop line the traffic light is detected as an MIO. At this point of time, the traffic light state is either red or yellow.

- The **Ego acceleration** plot shows the acceleration profile from the **Lane Following Controller**. Notice that this closely follows the dip in the relative distance, in reaction to the detection of the red traffic light as an MIO.
- The **Ego velocity** plot shows the velocity profile of the ego vehicle. Notice that the ego velocity slows down in reaction to the yellow and red traffic lights and comes to a complete halt before the stop line. This can be verified by comparing the plot with **Distance to traffic light stop line**, when the velocity is zero.

You can refer to the "Traffic Light Negotiation" on page 7-677 example to learn more about this analysis and the interactions between the decision logic and the controller.

Close the figure.

```
close(hFigResults);
```

Simulate Vehicle Behavior During Red To Green Transition

This section tests the decision logic when the ego vehicle is at a close distance to the traffic light and the traffic light state changes from red to green. In addition, a cross-traffic vehicle is in the intersection when the traffic light is green for the ego vehicle. The traffic light state is initally red for the ego vehicle and turns green when the ego vehicle is at a distance of 11 meters from the stop line. The ego vehicle is expected to slow down as it approaches the traffic light when the state is red and must start accelerating when the traffic light state changes from red to green. It is also expected to wait for the cross-traffic vehicle to pass the intersection before accelerating to continue its travel.

The test scenario function `scenario_TLN_red_to_green_with_cross_traffic_vehicle` implements this scenario. Configure the `TrafficLightNegotiationWithUnrealTestBench` model to use this scenario.

```
helperSLTrafficLightNegotiationWithUnrealSetup(...
    "scenario_TLN_straight_redToGreen_with_cross_vehicle");
```

Display the `trafficLightConfig` structure parameters that are set for this test scenario.

```
disp(trafficLightConfig');
```

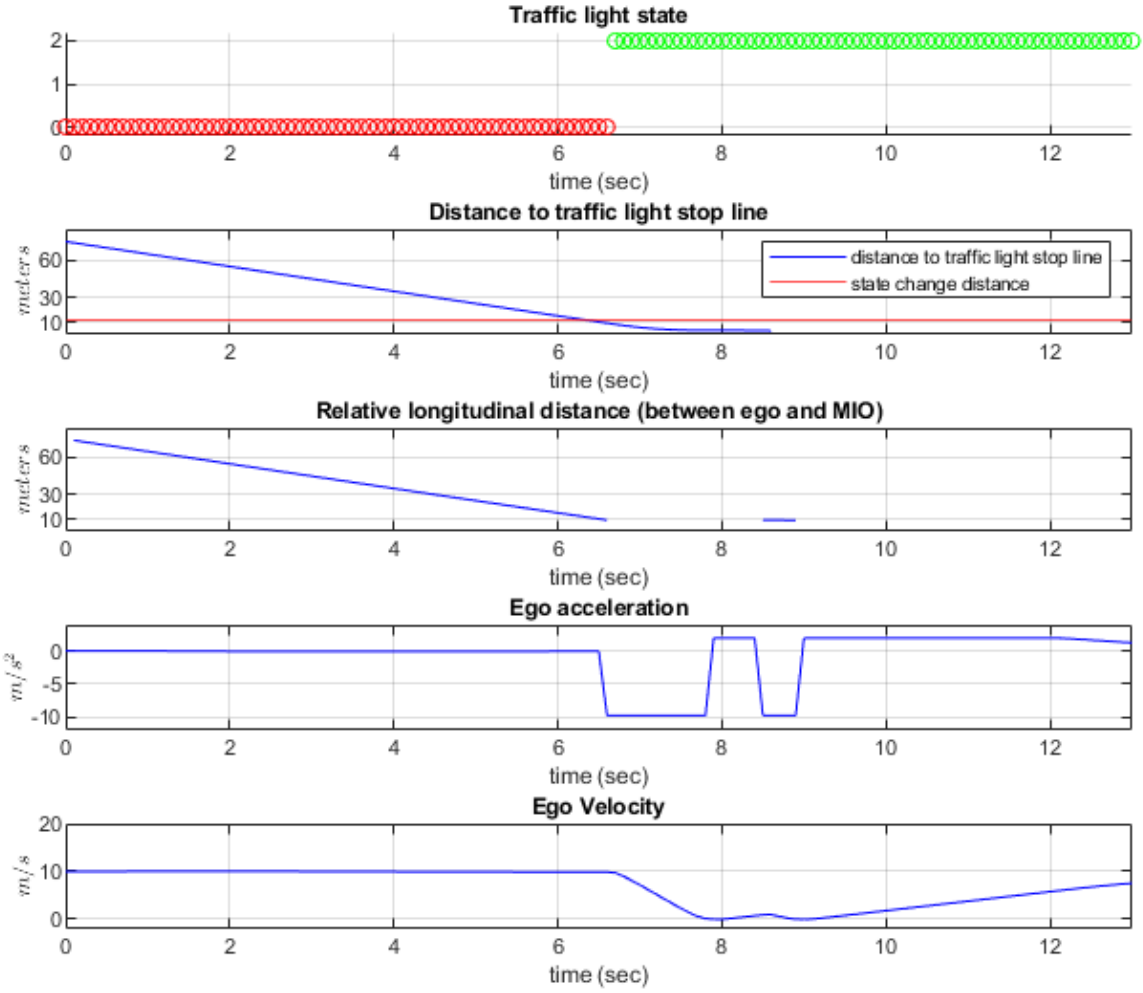
```
           initialState: 0
    stateChangeDistance: 11
            changeState: 2
```

Simulate the model.

```
sim("TrafficLightNegotiationWithUnrealTestBench");
```

Plot the simulation results.

```
hFigResults = helperPlotTrafficLightControlAndNegotiationResults(logsout, trafficLightConfig.sta
```



Examine the results.

- The **Traffic light state** plot shows that the initial traffic light state is red. The traffic light state changes from red to green when the ego vehicle is at a distance of 11 meters from the stop line.
- The **Relative longitudinal distance** plot closely follows the **Distance to traffic light stop line** plot because there is no lead vehicle. Notice the sudden dip in the relative distance in response to the detection of the cross-over vehicle.
- The **Ego acceleration** plot shows that the ego vehicle attempts to slow down on seeing the red traffic light. However, in response to the state change to green, you can observe an increase in acceleration. You can then notice a hard-braking profile in response to the cross-traffic vehicle at the intersection.
- The **Ego velocity** plot closely follows the **Ego acceleration** plot and shows a decrease in velocity as the ego vehicle approaches the intersection. You can also notice a slight increase in velocity in

response to green traffic light and subsequent decrease in velocity in response to the cross-traffic vehicle.

Close the figure.

```
close(hFigResults);
```

Explore Other Scenarios

In the previous sections, you explored the system behavior for the `scenario_TLN_straight_greenToRed_with_lead_vehicle` and `scenario_TLN_straight_redToGreen_with_cross_vehicle` scenarios. Below is a list of scenarios that are compatible with `TrafficLightNegotiationWithUnrealTestBench`.

```
scenario_TLN_straight_greenToRed
scenario_TLN_straight_greenToRed_with_lead_vehicle [Default]
scenario_TLN_straight_redToGreen_with_cross_vehicle
scenario_TLN_left_redToGreen_with_lead_vehicle
```

Use these additional scenarios to analyze `TrafficLightNegotiationWithUnrealTestBench` under different conditions.

Enable the MPC update messages.

```
mpcverbosity('on');
```

You can use the modeling patterns in this example to build your own traffic light negotiation application.

See Also

Cuboid To 3D Simulation | Multi-Object Tracker | Radar Detection Generator | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following | Vehicle To World | Vision Detection Generator

More About

- “Traffic Light Negotiation” on page 7-677

Generate Code for Lane Marker Detector

This example shows how to test a monocular-camera-based lane marker detector and generate C++ code for real-time applications on a prebuilt 3D scene from the Unreal Engine® driving simulation environment. This example validates the lane marker detector algorithm using metrics and verifies the generated C++ code by using software-in-the-loop simulation.

Introduction

A lane marker detector is a fundamental perception component of an automated driving application. The detector analyzes images of roads captured using a monocular camera sensor and returns information about the curvature and marking type of each lane. You can design and simulate a lane marker detector algorithm using MATLAB® or Simulink® and assess its accuracy using a known ground truth. You can do C++ code generation to integrate the detector to an external software environment and deploy to a vehicle. Performing code generation and verification of the Simulink model ensures functional equivalence between simulation and real-time implementation.

For information about how to design a lane marker detector, see the “Design Lane Marker Detector Using Unreal Engine Simulation Environment” on page 7-617 example. This example shows how to test the lane marker detector in a 3D simulation environment and generate C++ code for real-time implementation. In this example, you will:

1. Explore and simulate the lane marker detector model.
2. Assess accuracy of the lane marker detector algorithm by comparing with the ground truth.
3. Generate C++ code from the algorithm model.
4. Verify implementation with software-in-the-loop (SIL) simulation.
5. Assess execution time and perform code coverage analysis. To perform code coverage analysis, you must use Simulink Coverage™.

This example tests the lane marker detector algorithm on a 3D simulation environment that uses Unreal Engine® from Epic Games®. The Unreal Engine driving simulation environment requires a Windows® 64-bit platform.

```
if ~ispc
    error(['Unreal driving simulation environment is only supported on Microsoft', char(174), ' W
end
```

To ensure reproducibility of the simulation results, set the random seed.

```
rng(0);
```

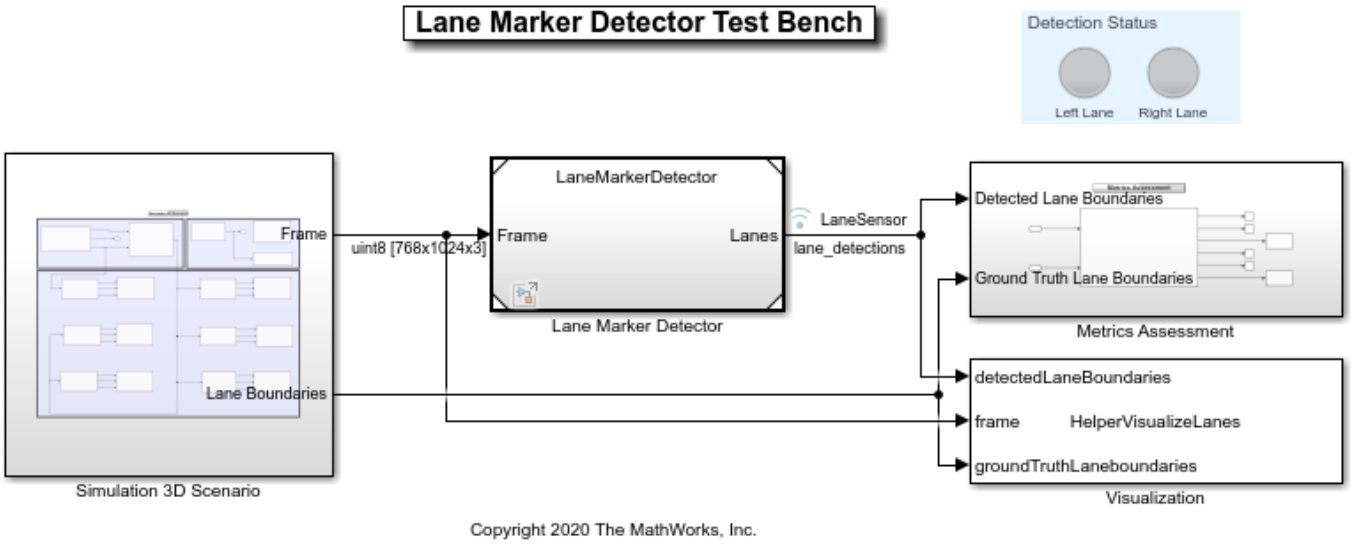
Explore the Model

The lane marker detector system in this example has a lane marker detector test bench and reference model.

- Test Bench Model: The test bench model simulates and tests the behavior of the lane marker detector algorithm in an open-loop.
- Reference Model: The **Lane Marker Detector** block in the test bench model invokes the `LaneMarkerDetector` reference model. The reference model implements the lane marker detection algorithm and generates the C++ code of the algorithm. This reference model can be integrated with closed-loop systems.

Open the test bench model.

```
open_system('LaneMarkerDetectorTestBench');
```

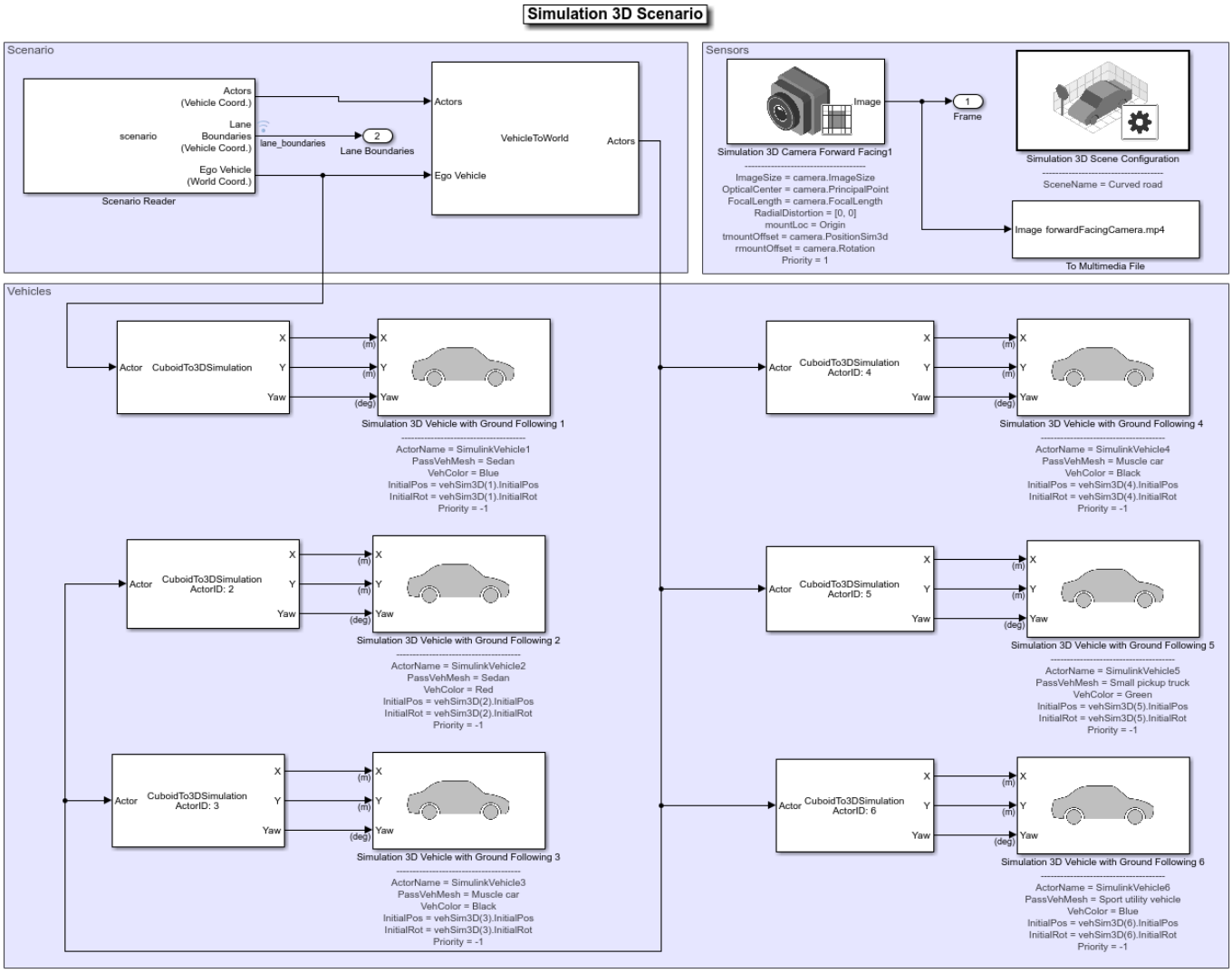


Opening this model runs the `helperSLLaneMarkerDetectorSetup` script that initializes the road scenario using the `drivingScenario` object in the base workspace. It also configures the lane marker detector parameters, vehicle model parameters, and the Simulink bus signals required for defining the inputs and outputs for the `LaneMarkerDetectorTestBench` model. The test bench model contains these subsystems:

1. Simulation 3D Scenario: Specifies the scene, vehicles, and camera sensor used for simulation.
2. Lane Marker Detector: Implements the lane marker detector algorithm.
3. Metrics Assessment: Assesses the lane marker detector algorithm behavior using metrics that include true positives, false positives, and false negatives.
4. Visualization: Displays the frame captured by the camera sensor and overlays the lane detections during the simulation.

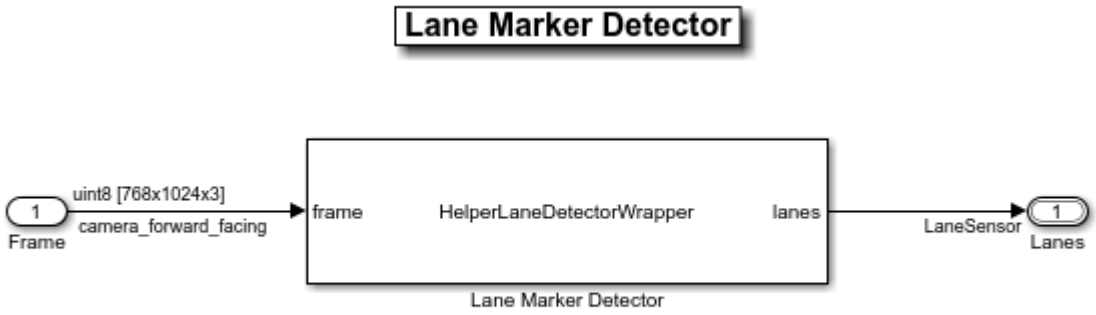
The **Simulation 3D Scenario** subsystem configures the road network, sets vehicle positions, and synthesizes sensors. This is similar to the Simulation 3D Scenario subsystem in the “Highway Lane Following” on page 7-653 example. However, the Simulation 3D Scenario subsystem used in this example does not have a radar sensor. Open the **Simulation 3D Scenario** subsystem.

```
open_system('LaneMarkerDetectorTestBench/Simulation 3D Scenario');
```



Lane Marker Detector is the reference model that detects the lane boundaries in the frames. Open the **Lane Marker Detector** reference model.

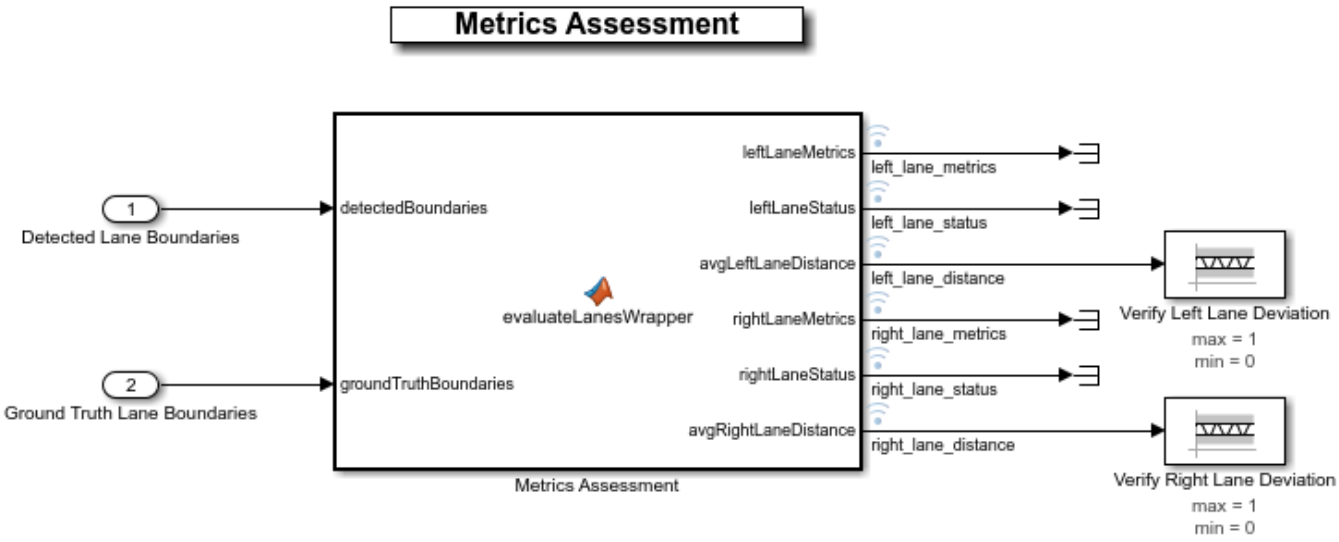
```
open_system('LaneMarkerDetector');
```



The **Lane Marker Detector** block uses a System Object™, `HelperLaneDetectorWrapper`, that configures and implements the lane marker detection algorithm. The block takes the frames captured by a camera sensor as input and outputs the detected lane boundaries by using the `LaneSensor` Simulink bus.

The **Metrics Assessment** subsystem evaluates the accuracy of detection results using the ground truth information. Open the **Metrics Assessment** subsystem.

```
open_system('LaneMarkerDetectorTestBench/Metrics Assessment');
```



The **Metrics Assessment** subsystem compares the detected lane boundaries and the ground truth lane boundaries by using the `evaluateLaneBoundaries` function. The function computes the lateral distance between the detected lane boundaries and the ground truth data. If the computed distance is within a particular threshold, then the detected boundary is considered as a valid match (true positive) and the corresponding lane status is set to 1. Otherwise, the function evaluates whether the boundary is a false negative or false positive and then sets the lane status to 0. The subsystem connects the outputs `left_lane_status` and `right_lane_status` to the lamps in the dashboard. The lamps go `green` when the lane status is 1 and go `red` when the lane status is 0. The outputs `left_lane_metrics` and `right_lane_metrics` are 1-by-3 arrays containing the lane

matches (true positives), misses (false negatives) and false positives for left and right lanes respectively. The average left lane and right lane deviations computed from the detected lane boundaries are returned at the outputs `left_lane_distance` and `right_lane_distance` respectively. The model uses these deviation values to verify the accuracy of the algorithm during the simulation using Check Static Range (Simulink) blocks: **Verify Left Lane Deviation** and **Verify Right Lane Deviation**.

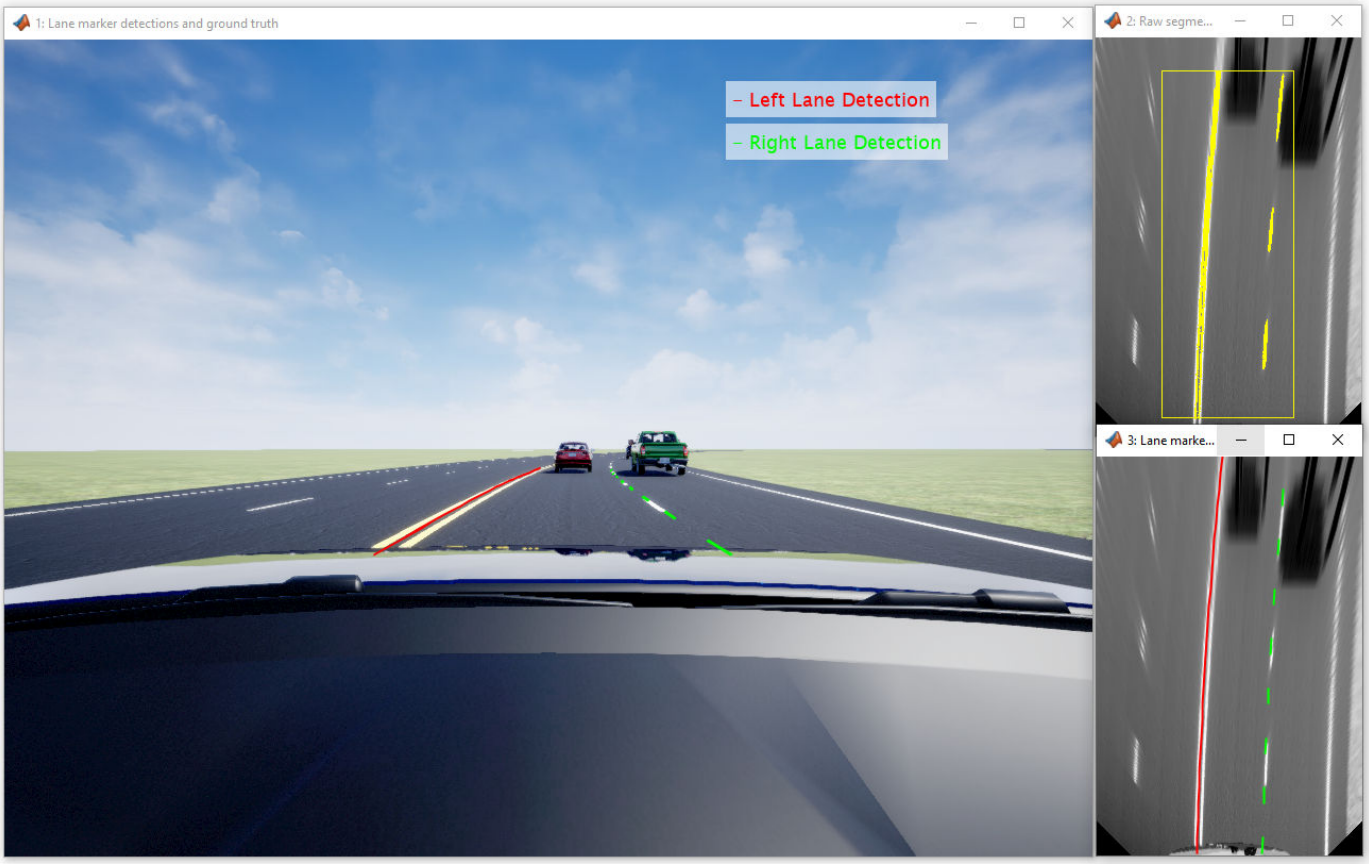
Simulate the Model

Configure the `LaneMarkerDetectorTestBench` model to simulate in the `scenario_LD_02_SixVehicles` scenario. This scenario contains six vehicles, including the ego vehicle, and defines their trajectories.

```
helperSLLaneMarkerDetectorSetup("scenario_LD_02_SixVehicles");
```

Simulate the test bench model. Use the visualization window and the status lamps on the dashboard to view the detection results while the simulation is running.

```
sim('LaneMarkerDetectorTestBench');
```



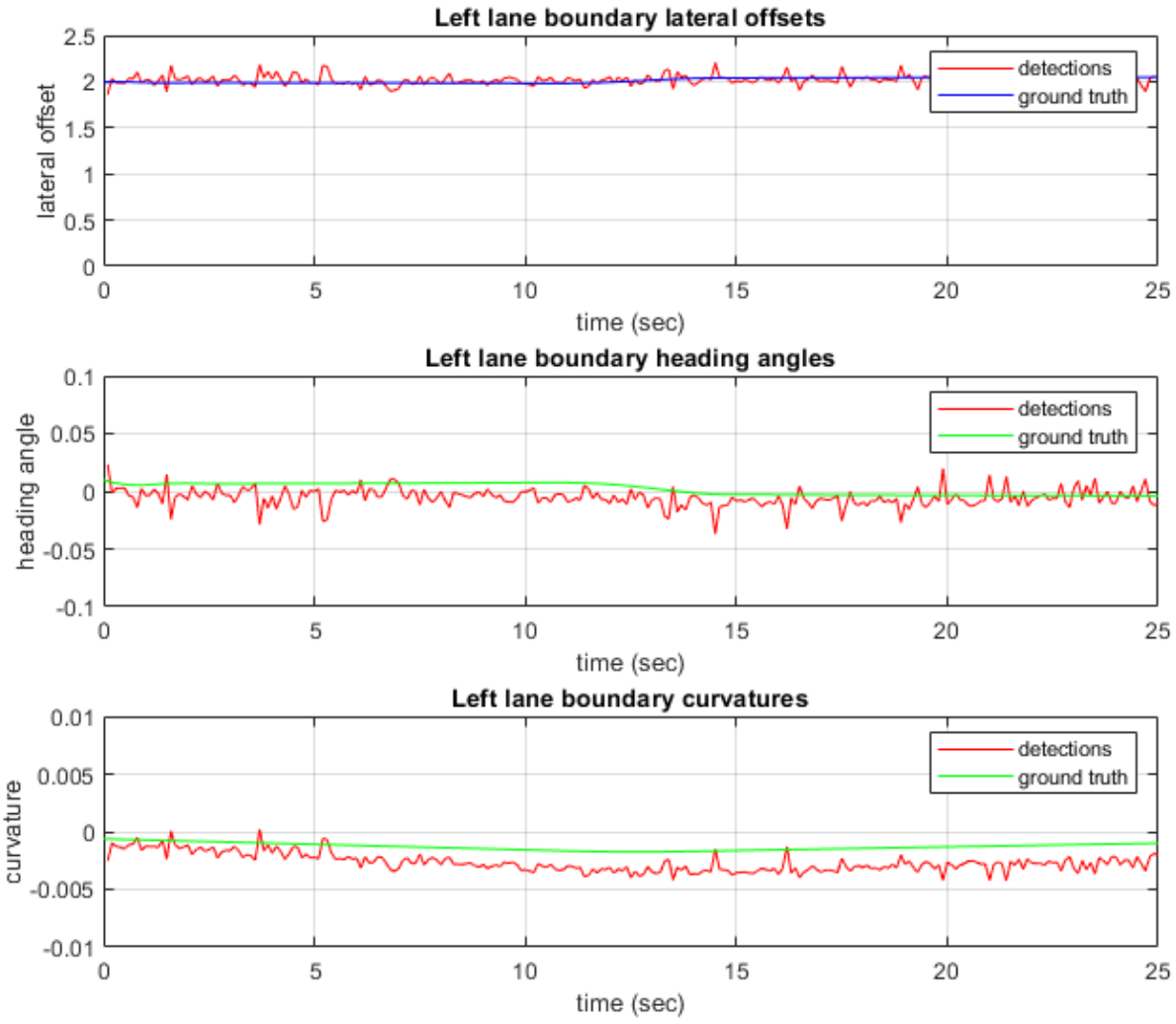
You can also visualize the ground truth lane boundaries by enabling the `EnableTruthDisplay` mask parameter in the **Visualization** block.

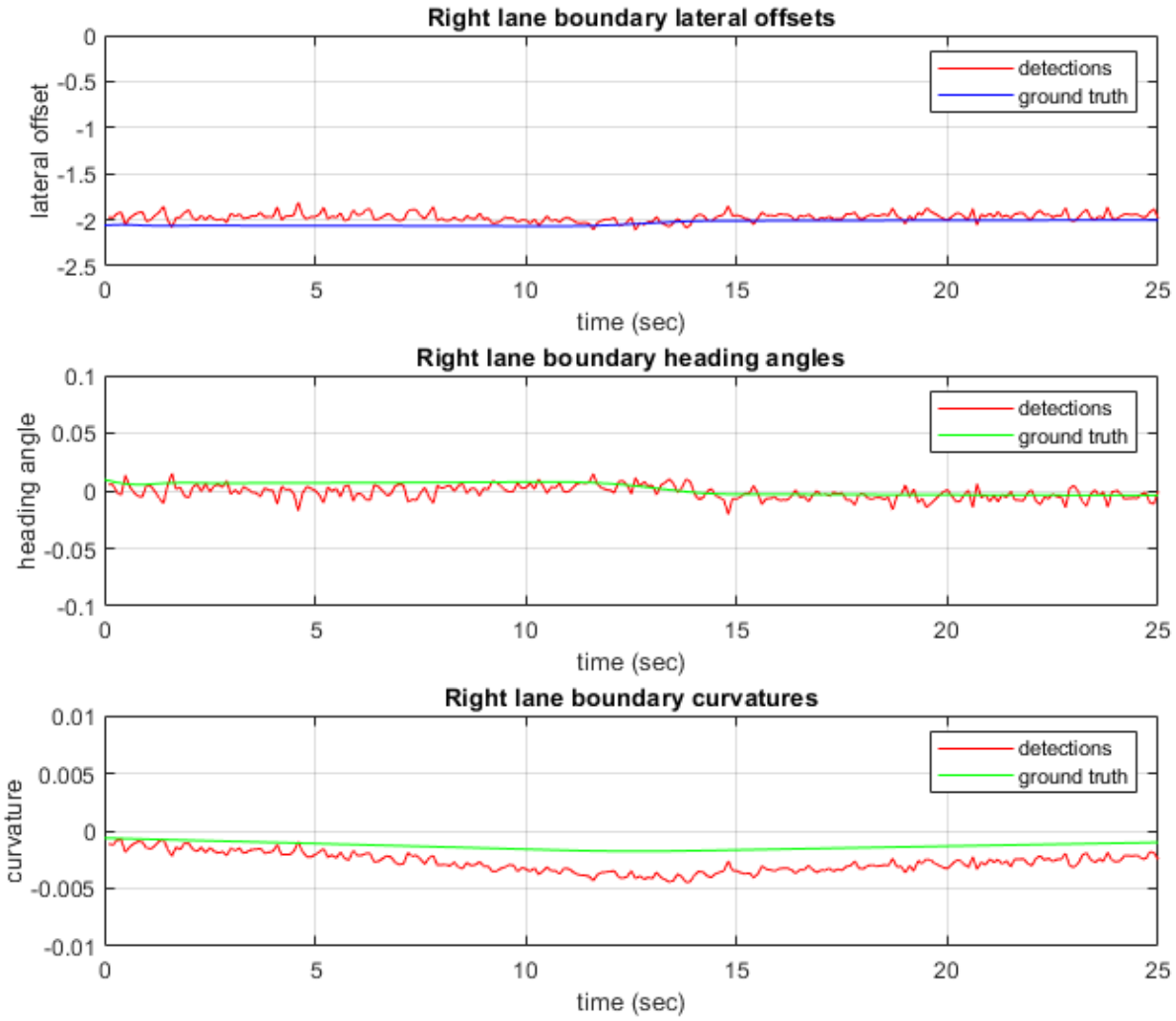
Assess Accuracy of Algorithm

Analyze the detection results and validate the overall performance of the algorithm.

During simulation, the model outputs three parameters that characterize a lane boundary: curvature, heading angle, and lateral offset. The model logs these parameters for the detected lane boundaries and ground truth to the base workspace variable `logsout`. You can plot the values in `logsout` by using the `helperLaneBoundaryParams` function.

```
helperPlotLaneBoundaryParams(logsout);
```





From the plots, you can infer the deviations between the ground truth and the detected lane boundaries. A large deviation between the plots indicates that the detected lane boundary is significantly away from the ground truth.

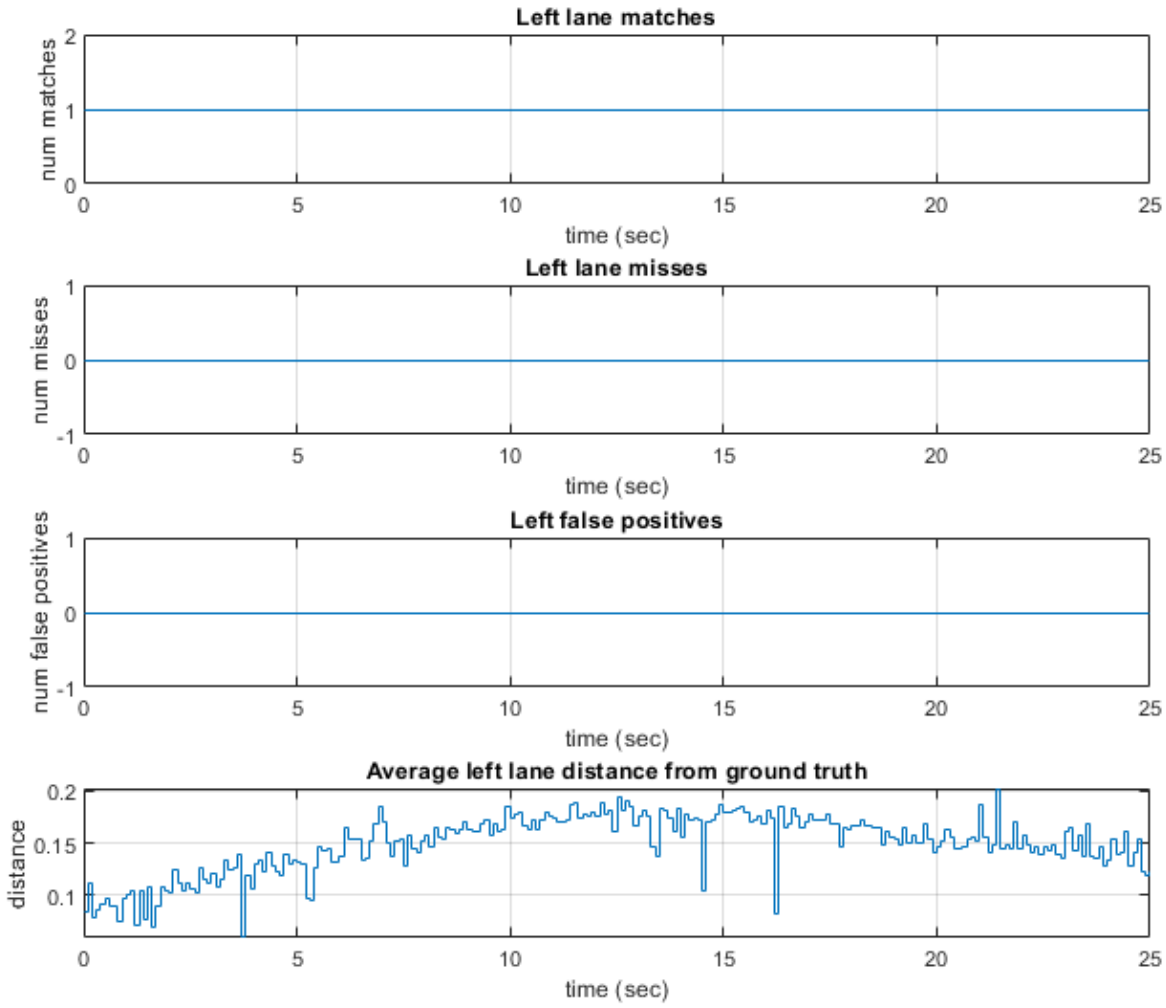
You can also verify the performance of the lane marker detector algorithm by validating and plotting the metrics separately for the left lane and the right lane. The model also logs the detection results for the left and right lanes computed by the **Metric Assesment** subsystem to the base workspace variable `logsout` . The **Metric Assesment** subsystem outputs:

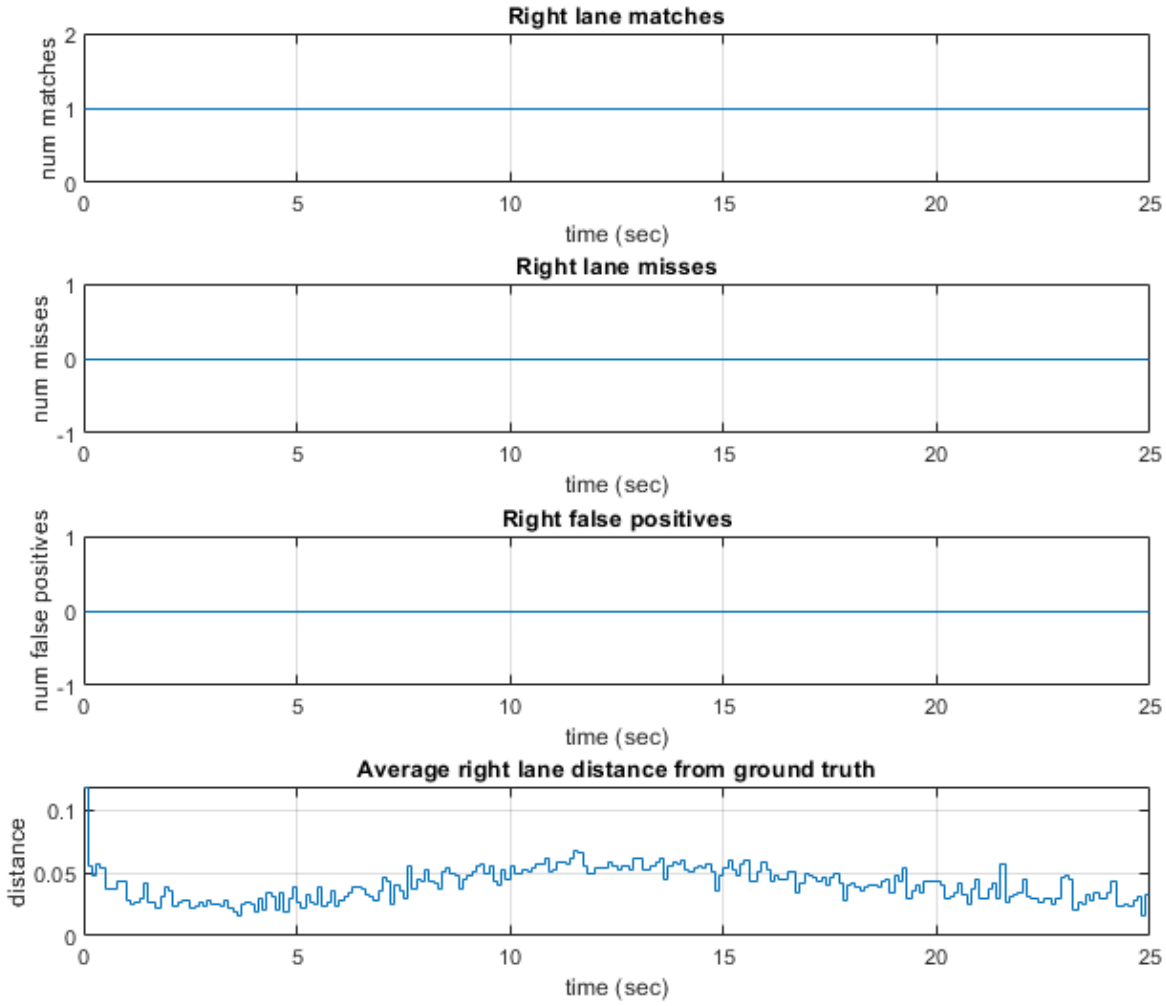
1. **Left lane metrics**: Returned as an array that contains the number of matches (true positives), misses (false negatives), and false positives computed from detections for the left lane of the road.
2. **Left lane status**: Returned as true or false. The value is true for matches in the left lane and false for misses and false positives in the left lane.

3. **Left lane distance**: A scalar specifying the average value of distances between detected left lane boundary points and the ground truth for the left lane.
4. **Right lane metrics**: Returned as an array that contains the number of matches (true positives), misses (false negatives), and false positives computed from detections for the right lane of the road.
5. **Right lane status**: Returned as true or false. The value is true for matches in the right lane and false for misses and false positives in the right lane.
6. **Right lane distance**: A scalar specifying the average value of distances between detected right lane boundary points and the ground truth for the right lane.

Plot the detection results for left and right lanes by using the `helperPlotLaneMetrics` script:

```
[numLaneMatches, numLaneMisses, numFalsePositives] = helperPlotLaneMetrics(logsout);
```

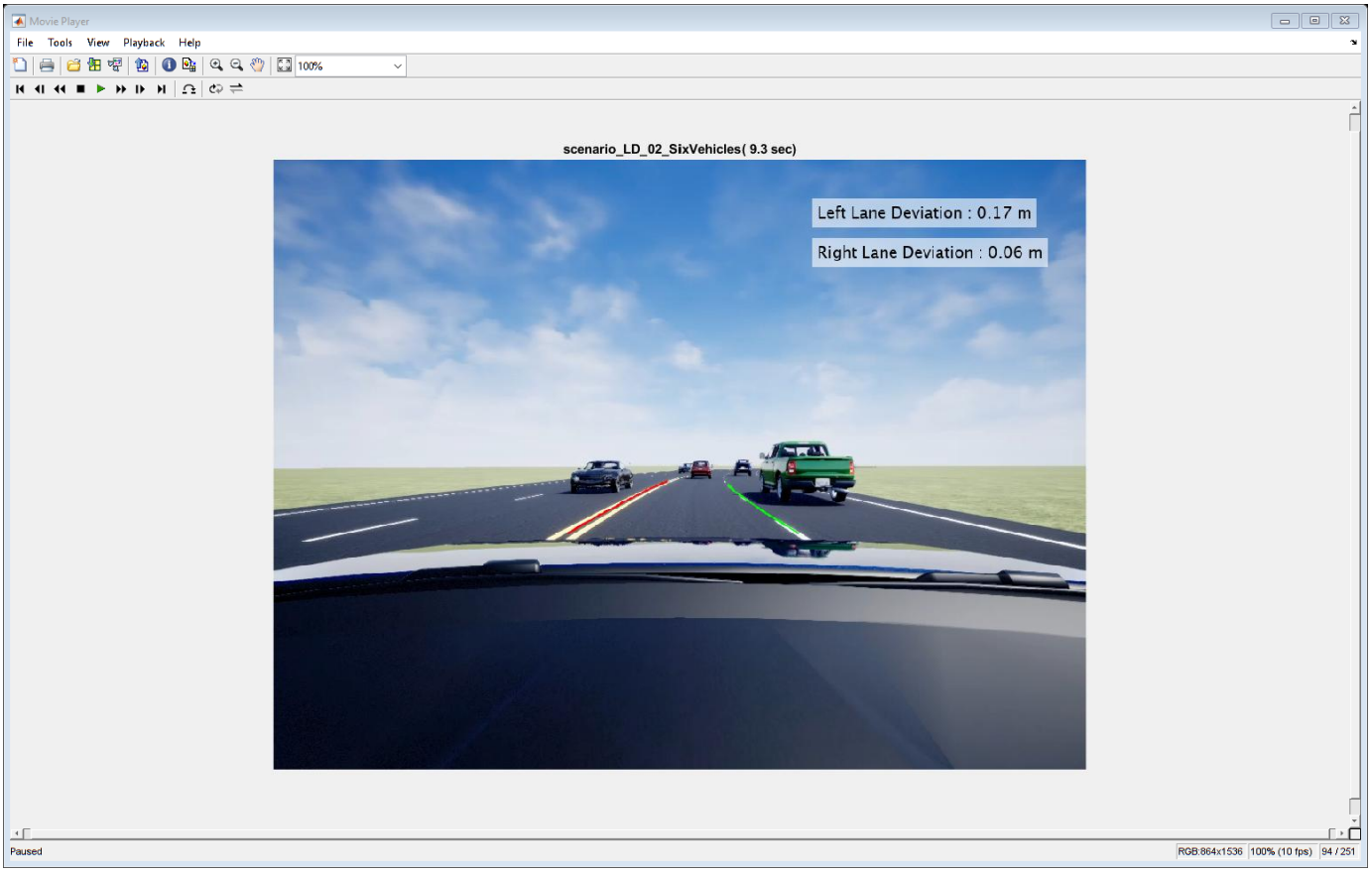




From the plots you can infer that all the detected left lane and the right lane boundaries match with the ground truth and that there are no false positives or false negatives.

During simulation, the model records the output of the camera sensor to `forwardFacingCamera.mp4`. You can overlay the deviation results on the lanes detected in the frames and record to a video file by using `helperPlotLaneDetectionResults` function.

```
hVideoViewer = helperPlotLaneDetectionResults(...
logsout, "forwardFacingCamera.mp4" , scenario, camera,scenarioFcnName,...
"RecordVideo", true,"RecordVideoFileName", scenarioFcnName,...
"OpenRecordedVideoInVideoViewer", true,"VideoViewerJumpToTime", 9.3);
```



You can also compute overall precision and sensitivity of the lane marker detection algorithm as below:

Precision: Compute as the percentage of true positives in the total number of detected lane boundaries.

```
precision = (numLaneMatches./(numLaneMatches+numFalsePositives))*100;
disp(precision);
```

```
   100
```

Sensitivity: Compute as the percentage of true positives in total number of actual lane boundaries.

```
sensitivity = (numLaneMatches./(numLaneMatches+numLaneMisses))*100;
disp(sensitivity);
```

```
   100
```

The precision and the sensitivity values of a robust lane marker detector must be close to 100 for different test scenarios and test conditions.

Generate C++ Code

You can now generate C++ code for the algorithm, apply common optimizations, and generate a report to facilitate exploring the generated code.

Configure the `LaneMarkerDetector` model to generate C++ code for real-time implementation of the algorithm. Set the model parameters to enable code generation and display the configuration values.

Set and view model parameters to enable C++ code generation.

```
helperSetModelParametersForCodeGeneration('LaneMarkerDetector');
save_system('LaneMarkerDetector');
```

```
 Model  configuration parameters:

                Parameter                       Value

    ____________________________________    _____________    ____________________________________

    {'SystemTargetFile'                }    {'ert.tlc'    }    {'Code Generation>System target fi
    {'TargetLang'                      }    {'C++'        }    {'Code Generation>Language'
    {'SolverType'                      }    {'Fixed-step' }    {'Solver>Type'
    {'FixedStep'                       }    {'auto'       }    {'Solver>Fixed-step size (fundamen
    {'EnableMultiTasking'              }    {'on'         }    {'Solver>Treat each discrete rate a
    {'ProdLongLongMode'                }    {'on'         }    {'Hardware Implementation>Support
    {'BlockReduction'                  }    {'on'         }    {'Simulation Target>Block reductio
    {'MATLABDynamicMemAlloc'           }    {'on'         }    {'Simulation Target>Simulation Tar
    {'OptimizeBlockIOStorage'          }    {'on'         }    {'Simulation Target>Signal storage
    {'InlineInvariantSignals'          }    {'on'         }    {'Simulation Target>Inline invaria
    {'BuildConfiguration'              }    {'Faster Runs'}    {'Code Generation>Build configurat
    {'RTWVerbose'                      }    {'of'         }    {'Code Generation>Verbose build'
    {'CombineSignalStateStructs'       }    {'on'         }    {'Code Generation>Interface>Combine
    {'SupportVariableSizeSignals'      }    {'on'         }    {'Code Generation>Interface>Suppor
    {'CodeInterfacePackaging'          }    {'C++ class'  }    {'Code Generation>Interface>Code i
    {'GenerateExternalIOAccessMethods'}    {'Method'     }    {'Code Generation>Interface>Data M
    {'EfficientFloat2IntCast'          }    {'on'         }    {'Code Generation>Optimization>Remo
    {'ZeroExternalMemoryAtStartup'     }    {'off'        }    {'Code Generation>Optimization>Remo
    {'CustomSymbolStrGlobalVar'        }    {'$N$M'       }    {'Code Generation>Symbols>Global va
    {'CustomSymbolStrType'             }    {'$N$M_T'     }    {'Code Generation>Symbols>Global ty
    {'CustomSymbolStrField'            }    {'$N$M'       }    {'Code Generation>Symbols>Field nam
    {'CustomSymbolStrFcn'              }    {'APV_$N$M$F' }    {'Code Generation>Symbols>Subsyste
    {'CustomSymbolStrTmpVar'           }    {'$N$M'       }    {'Code Generation>Symbols>Local te
    {'CustomSymbolStrMacro'            }    {'$N$M'       }    {'Code Generation>Symbols>Constant
```

Generate code and review the code generation report from the reference model.

```
rtwbuild('LaneMarkerDetector');
```

```
### Starting build procedure for: LaneMarkerDetector
### Successful completion of build procedure for: LaneMarkerDetector

Build Summary

Top model targets built:

Model               Action                       Rebuild Reason
=========================================================================================
LaneMarkerDetector  Code generated and compiled  Generated code was out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 2m 35.381s
```

Use the code generation report to explore the generated code. To learn more about the code generation report, see “Reports for Code Generation” (Embedded Coder). Use the Code Interface Report link in the Code Generation Report to explore these generated methods:

- `LaneMarkerDetector_initialize`: Call once on initialization.
- `LaneMarkerDetector_step`: Call periodically every step to execute the lane marker detection algorithm.
- `LaneMarkerDetector_terminate`: Call once on termination.

Additional get and set methods for signal interface are declared in `LaneMarkerDetector.h` and defined in `LaneMarkerDetector.cpp`.

Assess Functionality of Code

After generating C++ code for the lane marker detector, you can now assess the code functionality using software-in-the-loop (SIL) simulation. It provides early insight into the behavior of a deployed application. To learn more about SIL simulation, refer to “SIL and PIL Simulations” (Embedded Coder).

SIL simulation enables you to verify that the compiled generated code on the host is functionally equivalent to the normal mode.

Configure algorithm and test bench model parameters to support SIL simulation and log execution profiling information.

```
helperSetModelParametersForSIL('LaneMarkerDetector');
helperSetModelParametersForSIL('LaneMarkerDetectorTestBench');
```

```
LaneMarkerDetector configuration parameters:

             Parameter                        Value                                 Descripti
    ________________________________    ____________________    _________________________________

    {'SystemTargetFile'            }    {'ert.tlc'         }    {'Code Generation>System target f
    {'TargetLang'                  }    {'C++'             }    {'Code Generation>Language'
    {'CodeExecutionProfiling'      }    {'on'              }    {'Code Generation>Verification>Me
    {'CodeProfilingSaveOptions'    }    {'AllData'         }    {'Code Generation>Verification>Sa
    {'CodeExecutionProfileVariable'}    {'executionProfile'}    {'Code Generation>Verification>Wo

LaneMarkerDetectorTestBench configuration parameters:

             Parameter                        Value                                 Descripti
    ________________________________    ____________________    _________________________________

    {'SystemTargetFile'            }    {'ert.tlc'         }    {'Code Generation>System target f
    {'TargetLang'                  }    {'C++'             }    {'Code Generation>Language'
    {'CodeExecutionProfiling'      }    {'on'              }    {'Code Generation>Verification>Me
    {'CodeProfilingSaveOptions'    }    {'AllData'         }    {'Code Generation>Verification>Sa
    {'CodeExecutionProfileVariable'}    {'executionProfile'}    {'Code Generation>Verification>Wo
```

Configure the test bench model to simulate **Lane Marker Detector** in software-in-the-loop (SIL) mode.

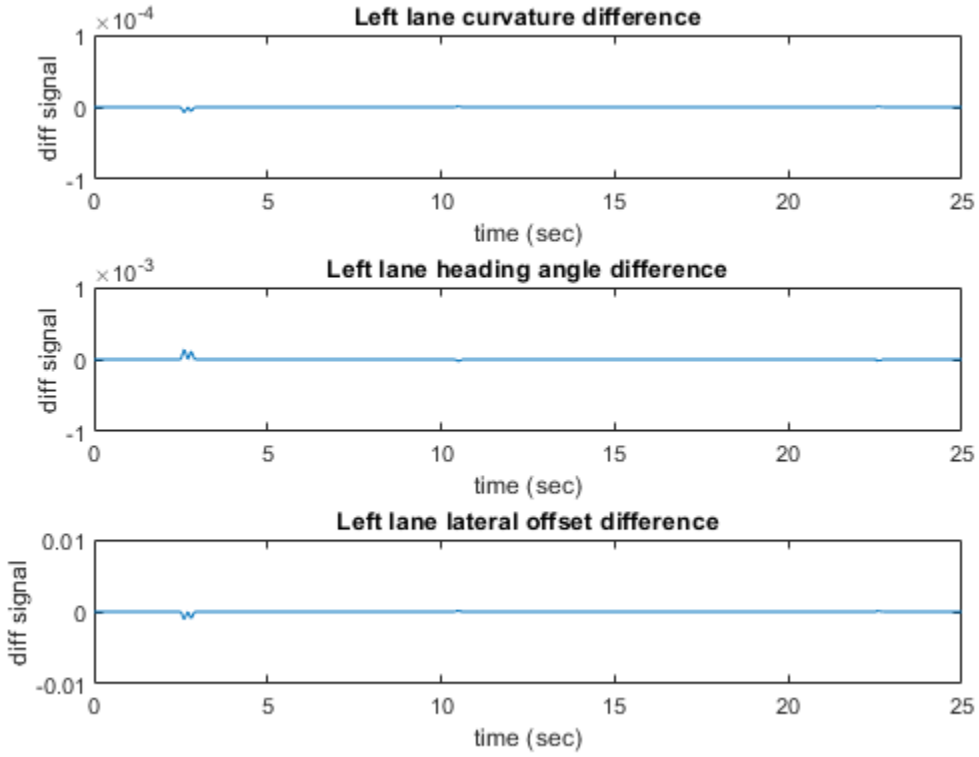
```
set_param('LaneMarkerDetectorTestBench/Lane Marker Detector','SimulationMode','Software-in-the-lo
sim('LaneMarkerDetectorTestBench');

### Starting build procedure for: LaneMarkerDetector
### Generated code for 'LaneMarkerDetector' is up to date because no structural, parameter or coo
### Successful completion of build procedure for: LaneMarkerDetector

Build Summary

Top model targets built:

Model               Action         Rebuild Reason
========================================================================
LaneMarkerDetector  Code compiled  Compilation artifacts were out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 1m 32.486s
### Preparing to start SIL simulation ...
Building with 'MinGW64 Compiler (C)'.
MEX completed successfully.
### Starting SIL simulation for component: LaneMarkerDetector
### Stopping SIL simulation for component: LaneMarkerDetector
### Completed code coverage analysis
```

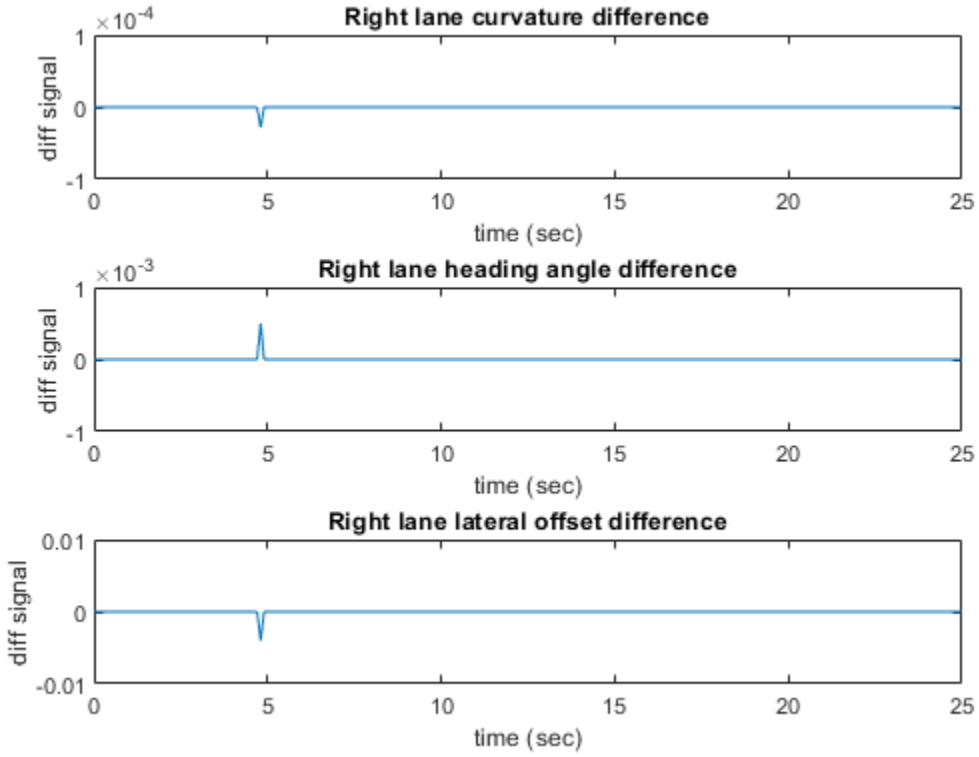
You can compare the outputs from normal simulation mode and software-in-the-loop (SIL) simulation mode. You can verify if the differences between these runs are in the tolerance limits by using the following code. Plot the differences of the detected lane boundary parameters between the normal simulation mode and SIL simulation mode.

```
runIDs = Simulink.sdi.getAllRunIDs;
normalSimRunID = runIDs(end - 1);
SilSimRunID = runIDs(end);
diffResult = Simulink.sdi.compareRuns(normalSimRunID ,SilSimRunID);
```

Plot the differences between lane boundary parameters computed from normal mode and SIL mode.

```
helperPlotDiffSignals(diffResult);
```


Left lane curvature difference
×10^-4
diff signal
time (sec)
Left lane heading angle difference
×10^-3
diff signal
time (sec)
Left lane lateral offset difference
diff signal
time (sec)



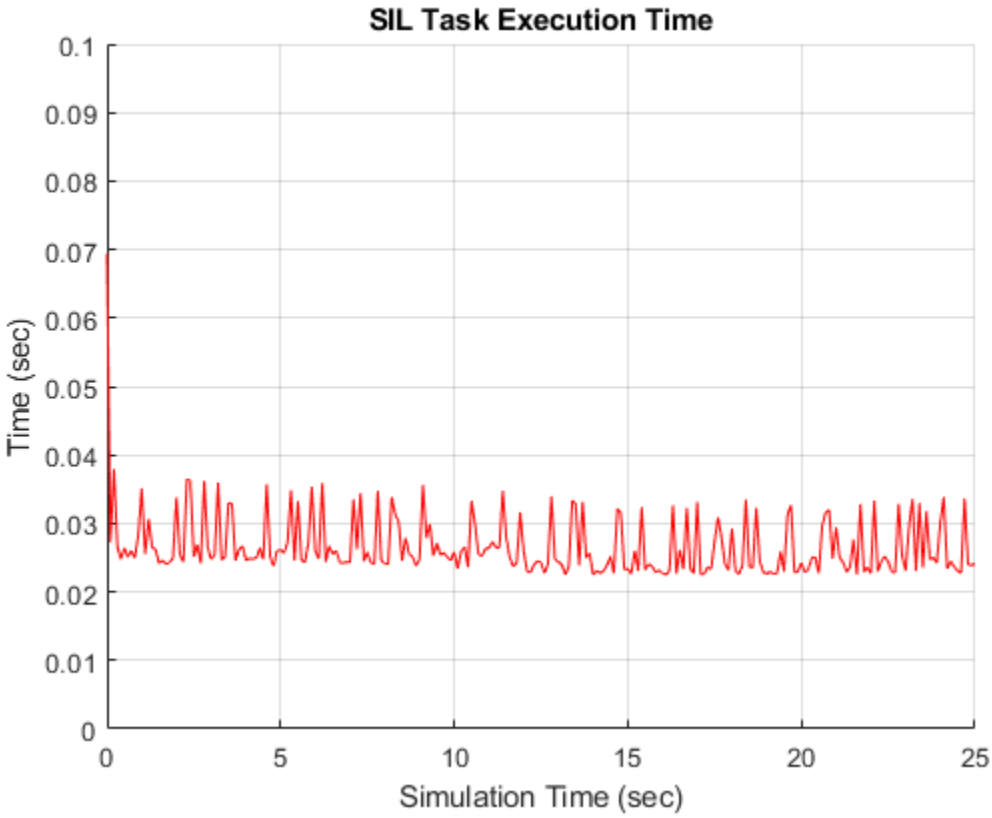
Notice that the differences between the lane boundary parameter values between normal mode of simulation and SIL mode of simulation are approximately zero. However, there are slight differences because of different rounding off techniques used by different compilers.

Assess Execution Time and Coverage of Code

During the software-in-the-loop (SIL) simulation, log the execution time metrics for the generated code on the host computer to the variable `executionProfile` in the MATLAB base workspace. These times can be an early indicator for performance of the generated code. For accurate execution time measurements, profile the generated code when it is integrated into the external environment or when using with processor-in-the-loop (PIL) simulation. To learn more about SIL profiling, refer to “Code Execution Profiling with SIL and PIL” (Embedded Coder).

Plot the execution time taken for `LaneMarkerDetector_step` function using `helperPlotExecutionProfile` function.

```
helperPlotExecutionProfile(executionProfile);
```



Notice that you can deduce the average time taken per frame for the lane marker detector from this plot. For more information on generating execution profiles and analyzing them during SIL simulation, refer to “Execution Time Profiling for SIL and PIL” (Embedded Coder).

If you have a Simulink Coverage™ license, you can also perform the code coverage analysis for the generated code to measure the testing completeness. You can use missing coverage data to find gaps in testing, missing requirements, or unintended functionality. Configure the coverage settings and simulate the test bench model to generate coverage analysis report. Find the generated report `CoverageResults/LaneMarkerDetector.html` in the working directory.

```
if(license('test','Simulink_Coverage'))
    helperCoverageSettings('LaneMarkerDetectorTestBench');
    cvDataObj = cvsim('LaneMarkerDetectorTestBench');
    cvhtml('CoverageResults/LaneMarkerDetector',cvDataObj);
end
```

```
### Starting build procedure for: LaneMarkerDetector
### Generated code for 'LaneMarkerDetector' is up to date because no structural, parameter or co
### Successful completion of build procedure for: LaneMarkerDetector

Build Summary

Top model targets built:

Model               Action         Rebuild Reason
========================================================================
LaneMarkerDetector  Code compiled  Compilation artifacts were out of date.
```

```
1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 49.471s
### Preparing to start SIL simulation ...
### Starting SIL simulation for component: LaneMarkerDetector
### Stopping SIL simulation for component: LaneMarkerDetector
### Completed code coverage analysis
```

Summary

File/Complexity		Test 1				
		Decision	Statement	Function	Function call	Relational Boundary
TOTAL COVERAGE	1144	73%	85%	95%	90%	26%
1. . . . LaneMarkerDetector.cpp	1127	73%	85%	94%	90%	27%
2. . . . rtGetInf.cpp	6	50%	93%	100%	0%	0%
3. . . . rtGetNaN.cpp	3	50%	93%	100%	0%	0%
4. . . . rt_nonfinite.cpp	8	50%	96%	100%	86%	0%

Summary By Model Object

Model Object	Test 1				
	Decision	Statement	Function	Function call	Relational Boundary
1. LaneMarkerDetector	73%	85%	95%	90%	26%

You can find the decision coverage, statements coverage and function coverage results while simulating the generated code for this test scenario, `scenario_LD_02_SixVehicles`. You can test this model with different scenarios to get full coverage of the generated code. For more information on how to analyze coverage results during software-in-the-loop simulation, refer “Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode” (Embedded Coder)

Explore Additional Scenarios

This example provides additional scenarios that you can use with the `LaneMarkerDetectorTestBench` model:

```
scenario_LF_01_Straight_RightLane
scenario_LF_02_Straight_LeftLane
scenario_LF_03_Curve_LeftLane
scenario_LF_04_Curve_RightLane
scenario_LD_01_ThreeVehicles
scenario_LD_02_SixVehicles [Default]
```

- Use scenarios with the prefix `scenario_LF` in the filename to test the lane marker detection algorithm without obstruction by other vehicles. The vehicles still exist in the scenario but are positioned such that they are not within the visible range of the ego vehicle.
- Use scenarios with the prefix `scenario_LD` in the filename to test the lane marker detection algorithm while other vehicles on the road are within the visible range of the ego vehicle.

While designing and testing the lane marker detection algorithm in open loop, it is helpful to begin with a scenario that has only the ego vehicle. To configure the model and workspace for such a scenario, use the following code.

```
helperSLLaneMarkerDetectorSetup("scenario_LF_04_Curve_RightLane");
```

You can use this model to integrate RoadRunner™ scenes into driving scenarios for simulation and testing.

See Also

Functions
`evaluateLaneBoundaries`

Blocks
Cuboid To 3D Simulation | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following | Vehicle To World

More About

- “Visual Perception Using Monocular Camera” on page 7-78
- “Design Lane Marker Detector Using Unreal Engine Simulation Environment” on page 7-617
- “Evaluate and Visualize Lane Boundary Detections Against Ground Truth” on page 7-65

Highway Lane Following with Intelligent Vehicles

This example shows how to simulate lane following application in a scenario that contains intelligent target vehicles. The intelligent target vehicles are the non-ego vehicles in the scenario that are programmed to adapt their trajectories based on the behavior of its neighboring vehicles. In this example, you will:

1. Model the behavior of the target vehicles to dynamically adapt their trajectories in order to perform one of the following behaviors: velocity keeping, lane following or lane change.

2. Simulate and test the lane following application in response to the dynamic behavior of the target vehicles on straight road and curved road scenarios.

You can also apply the modeling patterns used in this example to test your own lane following algorithms.

Introduction

The highway lane following system developed in this example steers the ego vehicle to travel within a marked lane. The system tests the lane following capability in the presence of other non-ego vehicles which are the target vehicles. For regression testing, it is often sufficient for the target vehicles to follow a predefined trajectory. But to randomize the behavior and identify edge cases like aggressive lane change in front of the ego vehicle, it is beneficial to add intelligence to the target vehicles.

This example builds on the “Highway Lane Following” on page 7-653 example that demonstrates lane following with target vehicles that follow predefined trajectories. This example modifies the scenario simulation framework of the “Highway Lane Following” on page 7-653 example by adding functionalities to model and simulate intelligent target vehicles. The intelligent target vehicles added to this example adapt their trajectories based on the behavior of the neighboring vehicles and the environment. In response, the lane following system automatically reacts to ensure that the ego vehicle stays in its lane.

In this example, you achieve system-level simulation through integration with the Unreal Engine® from Epic Games®. The 3D simulation environment requires a Windows® 64-bit platform.

```
if ~ispc
    error(['3D Simulation is only supported on Microsoft', char(174), ' Windows', char(174), '.'])
end
```

To ensure reproducibility of the simulation results, set the random seed.

```
rng(0);
```

In the rest of the example, you will:

1. **Explore the test bench model**: Explore the functionalities in the system-level test bench model that you use to assess lane following with intelligent target vehicles.
2. **Vehicle behaviors**: Explore vehicle behaviors that you can use to model the intelligent target vehicles.
3. **Model the intelligent target vehicles**: Model the target vehicles in the scenario for three different behaviors: velocity keeping, lane following, and lane change.
4. **Simulate lane following with intelligent target vehicles on a straight road**: Simulate velocity keeping, lane following, and lane change behaviors of a target vehicle while testing lane following on a straight road.

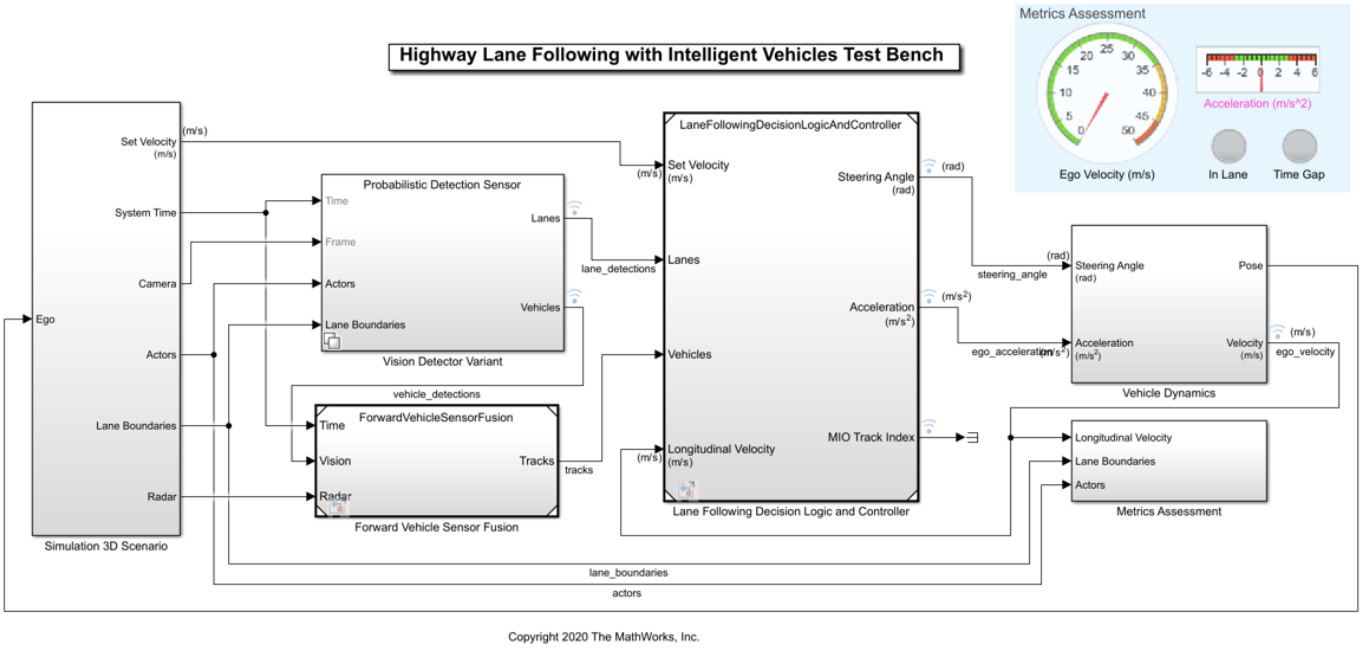
5 **Simulate lane following with intelligent target vehicles on a curved road**: Simulate velocity keeping, lane following, and lane change behaviors of a target vehicle while testing lane following on a curved road.

6 **Test with other scenarios**: Test the model with other scenarios available with this example.

Explore Test Bench Model

Open the system-level simulation test bench model for the lane following application.

```
open_system("HighwayLaneFollowingWithIntelligentVehiclesTestBench")
```



Copyright 2020 The MathWorks, Inc.

The test bench model contains these subsystems:

1 **Simulation 3D Scenario**: Specifies road, ego vehicle, intelligent target vehicles, camera, and radar sensors used for simulation.

2 **Vision Detector Variant**: Specifies the fidelity of the two different vision detection algorithms to choose from.

3 **Forward Vehicle Sensor Fusion**: Fuses the detections of vehicles in front of the ego vehicle that were obtained from vision and radar sensors.

4 **Lane Following Decision and Controller**: Specifies lateral and longitudinal decision logic and the lane following controller.

5 **Vehicle Dynamics**: Specifies the dynamics model for the ego vehicle

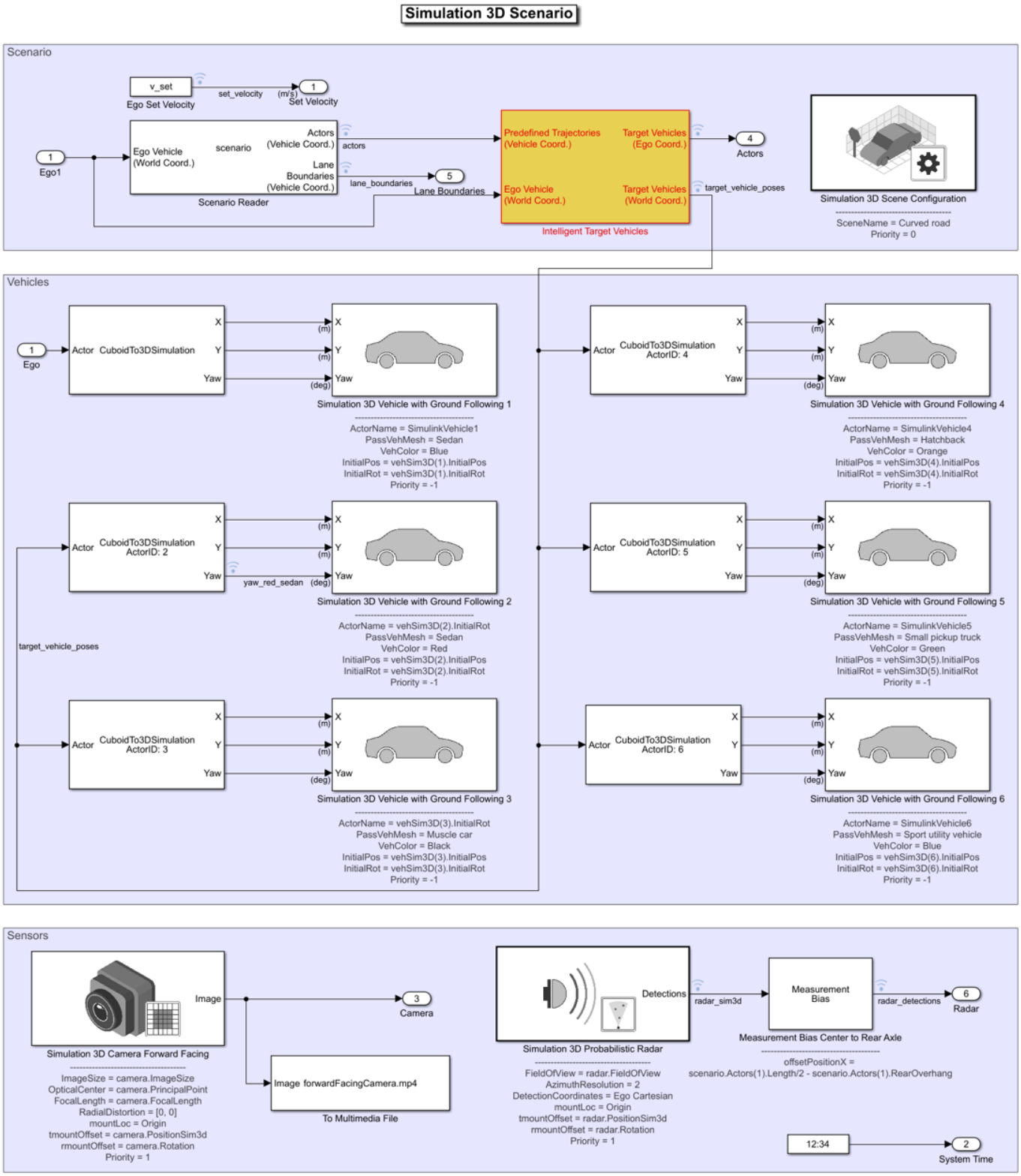
6 **Metrics Assessment**: Assesses system-level behavior.

The Vision Detector Variant, Forward Vehicle Sensor Fusion, Lane Following Decision and Controller, Vehicle Dynamics, and Metrics Assessment subsystems are based on the subsystems used in “Highway Lane Following” on page 7-653. This example focuses only on the **Simulation 3D Scenario** subsystem. An **Intelligent Target Vehicles** subsystem block is added to the **Simulation 3D Scenario** subsystem in order to configure the behavior of target vehicles in the scenario. The Vision Detector Variant, Forward Vehicle Sensor Fusion, Lane Following Decision and Controller,

Vehicle Dynamics, and Metrics Assessment subsystems steer the ego vehicle in response to the behavior of the target vehicles configured by **Simulation 3D Scenario** subsystem.

Open the **Simulation 3D Scenario** subsystem and highlight the **Intelligent Target Vehicles** subsystem.

```
open_system("HighwayLaneFollowingWithIntelligentVehiclesTestBench/Simulation 3D Scenario")
hilite_system("HighwayLaneFollowingWithIntelligentVehiclesTestBench/Simulation 3D Scenario/Intel
```


Simulation 3D Scenario
Scenario
v_set
Ego Set Velocity
set_velocity
(m/s)
1
Set Velocity
1
Ego1
Ego Vehicle
(World Coord.)
scenario
Actors
(Vehicle Coord.)
actors
Lane
Boundaries
(Vehicle Coord.)
lane_boundaries
5
Lane Boundaries
Scenario Reader
Predefined Trajectories
(Vehicle Coord.)
Target Vehicles
(Ego Coord.)
Ego Vehicle
(World Coord.)
Target Vehicles
(World Coord.)
Intelligent Target Vehicles
4
Actors
target_vehicle_poses
Simulation 3D Scene Configuration
SceneName = Curved road
Priority = 0
Vehicles
1
Ego
Actor
CuboidTo3DSimulation
X
Y
Yaw
(m)
(deg)
Simulation 3D Vehicle with Ground Following 1
ActorName = SimulinkVehicle1
PassVehMesh = Sedan
VehColor = Blue
InitialPos = vehSim3D(1).InitialPos
InitialRot = vehSim3D(1).InitialRot
Priority = -1
CuboidTo3DSimulation
ActorID: 4
Simulation 3D Vehicle with Ground Following 4
ActorName = SimulinkVehicle4
PassVehMesh = Hatchback
VehColor = Orange
InitialPos = vehSim3D(4).InitialPos
InitialRot = vehSim3D(4).InitialRot
Priority = -1
CuboidTo3DSimulation
ActorID: 2
yaw_red_sedan
Simulation 3D Vehicle with Ground Following 2
ActorName = vehSim3D(2).InitialRot
PassVehMesh = Sedan
VehColor = Red
InitialPos = vehSim3D(2).InitialPos
InitialRot = vehSim3D(2).InitialRot
Priority = -1
CuboidTo3DSimulation
ActorID: 5
Simulation 3D Vehicle with Ground Following 5
ActorName = SimulinkVehicle5
PassVehMesh = Small pickup truck
VehColor = Green
InitialPos = vehSim3D(5).InitialPos
InitialRot = vehSim3D(5).InitialRot
Priority = -1
target_vehicle_poses
CuboidTo3DSimulation
ActorID: 3
Simulation 3D Vehicle with Ground Following 3
ActorName = vehSim3D(3).InitialRot
PassVehMesh = Muscle car
VehColor = Black
InitialPos = vehSim3D(3).InitialPos
InitialRot = vehSim3D(3).InitialRot
Priority = -1
CuboidTo3DSimulation
ActorID: 6
Simulation 3D Vehicle with Ground Following 6
ActorName = SimulinkVehicle6
PassVehMesh = Sport utility vehicle
VehColor = Blue
InitialPos = vehSim3D(6).InitialPos
InitialRot = vehSim3D(6).InitialRot
Priority = -1
Sensors
Image
Simulation 3D Camera Forward Facing
ImageSize = camera.ImageSize
OpticalCenter = camera.PrincipalPoint
FocalLength = camera.FocalLength
RadialDistortion = [0, 0]
mountLoc = Origin
tmountOffset = camera.PositionSim3d
rmountOffset = camera.Rotation
Priority = 1
3
Camera
Image forwardFacingCamera.mp4
To Multimedia File
Detections
Simulation 3D Probabilistic Radar
FieldOfView = radar.FieldOfView
AzimuthResolution = 2
DetectionCoordinates = Ego Cartesian
mountLoc = Origin
tmountOffset = radar.PositionSim3d
rmountOffset = radar.Rotation
Priority = 1
radar_sim3d
Measurement
Bias
Measurement Bias Center to Rear Axle
offsetPositionX =
scenario.Actors(1).Length/2 - scenario.Actors(1).RearOverhang
radar_detections
6
Radar
12:34
2
System Time

The **Simulation 3D Scenario** subsystem configures the road network, models the target vehicles, sets vehicle positions, and synthesizes sensors. The subsystem is initialized by using the `helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup` script. The `helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup` script defines the driving scenario for testing the highway lane following. This setup script defines the road network and sets the behavior for each target vehicle in the scenario.

- The Scenario Reader block reads the roads and actors (ego and target vehicles) from a scenario file specified using the `helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup` script. The block outputs the poses of target vehicles and the lane boundaries with respect to the coordinate system of the ego vehicle.
- The **Intelligent Target Vehicles** is a function-call subsystem block that models the behavior of the actors in the driving scenario. The initial values for this subsystem block parameters are set by the `helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup` script. The Cuboid To 3D Simulation and the Simulation 3D Vehicle with Ground Following blocks sets the actor poses for the 3D simulation environment.
- The Simulation 3D Scene Configuration block implements a 3D simulation environment by using the road network and the actor positions.

This setup script also configures the controller design parameters, vehicle model parameters, and the Simulink® bus signals required for the `HighwayLaneFollowingWithIntelligentVehiclesTestBench` model. This script assigns an array of structures `targetVehicles` to the base workspace that contains the behavior type for each target vehicle.

Vehicle Behaviors

This example allows you to use four modes of vehicle behaviors for configuring the target vehicles using the `targetVehicles` structure.

- `Default`: In this mode, the target vehicles in the scenario follow predefined trajectories. The target vehicles are non-adaptive and are not configured for intelligent behavior.
- `VelocityKeeping`: In this mode, the target vehicles are configured to travel in a lane at a constant set velocity. Each target vehicle maintains the set velocity regardless of the presence of a lead vehicle in its current lane and does not check for collision.
- `LaneFollowing`: In this mode, the target vehicles are configured to travel in a lane by adapting their velocities in response to a lead vehicle. If a target vehicle encounters a lead vehicle in its current lane, the model performs collision checking and adjusts the velocity of the target vehicle. Collision checking ensures that the target vehicle maintains a safe distance from the lead vehicle.
- `LaneChange`: In this mode, the target vehicles are configured to travel in a lane at a particular velocity and follow the lead vehicle. If the target vehicle gets too close to the lead vehicle, then it performs a lane change. Before changing the lane, the model checks for potential forward and side collisions and adapts the velocity of the target vehicle to maintain safe distance from other vehicles in the scenario.

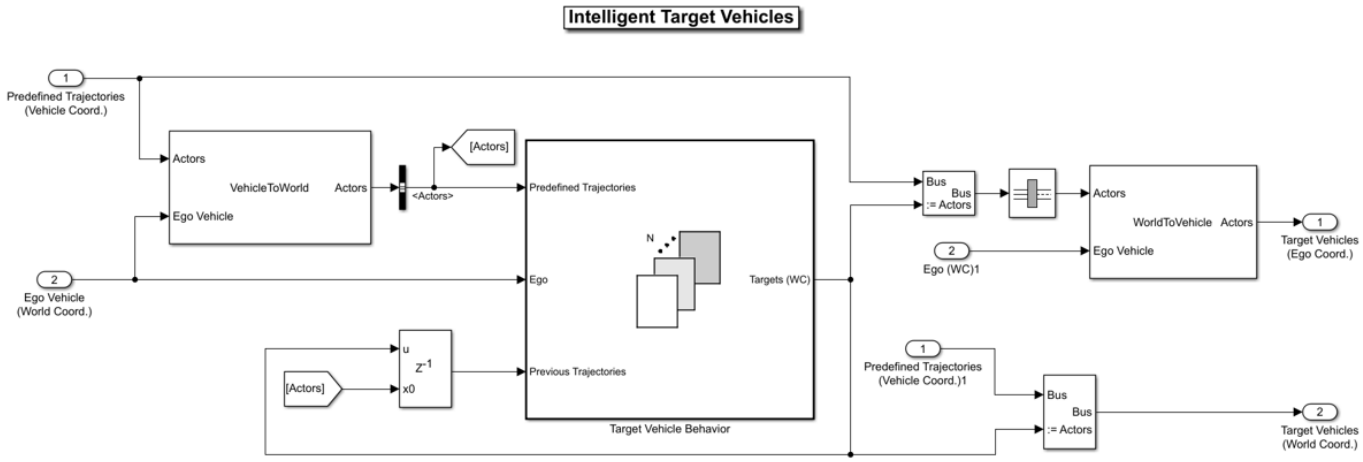
Model Intelligent Target Vehicles

The **Intelligent Target Vehicles** subsystem dynamically updates the vehicle poses for all the target vehicles based on their predefined vehicle behavior. As mentioned already, the `helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup` script defines the scenario and the behavior for each target vehicle in the scenario. The setup script stores the vehicle behavior and other attributes as an array of structures `targetVehicles` to the base workspace. The structure stores these attributes:

- ActorID
- Position
- Velocity
- Roll
- Pitch
- Yaw
- AngularVelocity
- InitialLaneID
- BehaviorType

The **Intelligent Target Vehicles** subsystem uses a mask to load the configuration in `targetVehicles` from the base workspace. You can set the values of these attributes to modify the position, orientation, velocities, and behavior of target vehicles. Open the **Intelligent Target Vehicles** subsystem.

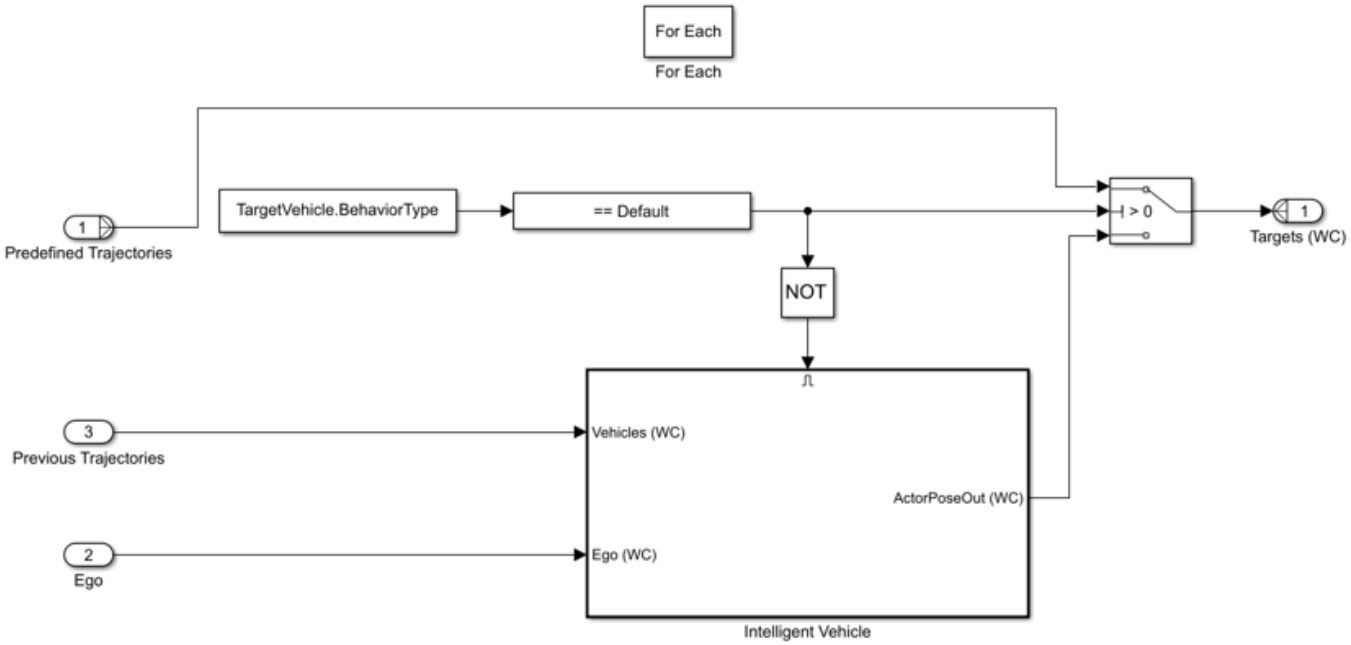
```
open_system("HighwayLaneFollowingWithIntelligentVehiclesTestBench/Simulation 3D Scenario/Intelli
```



The Vehicle To World block converts the predefined actor (ego and target vehicle) poses and trajectories from ego-vehicle coordinates to world coordinates. The **Target Vehicle Behavior** subsystem block computes the next state of the target vehicles by using predefined target vehicles poses, ego-vehicle pose, and current state of target vehicles. The subsystem outputs the target vehicles poses in world coordinates for navigating the vehicles in 3D simulation environment.

Open the **Target Vehicle Behavior** subsystem.

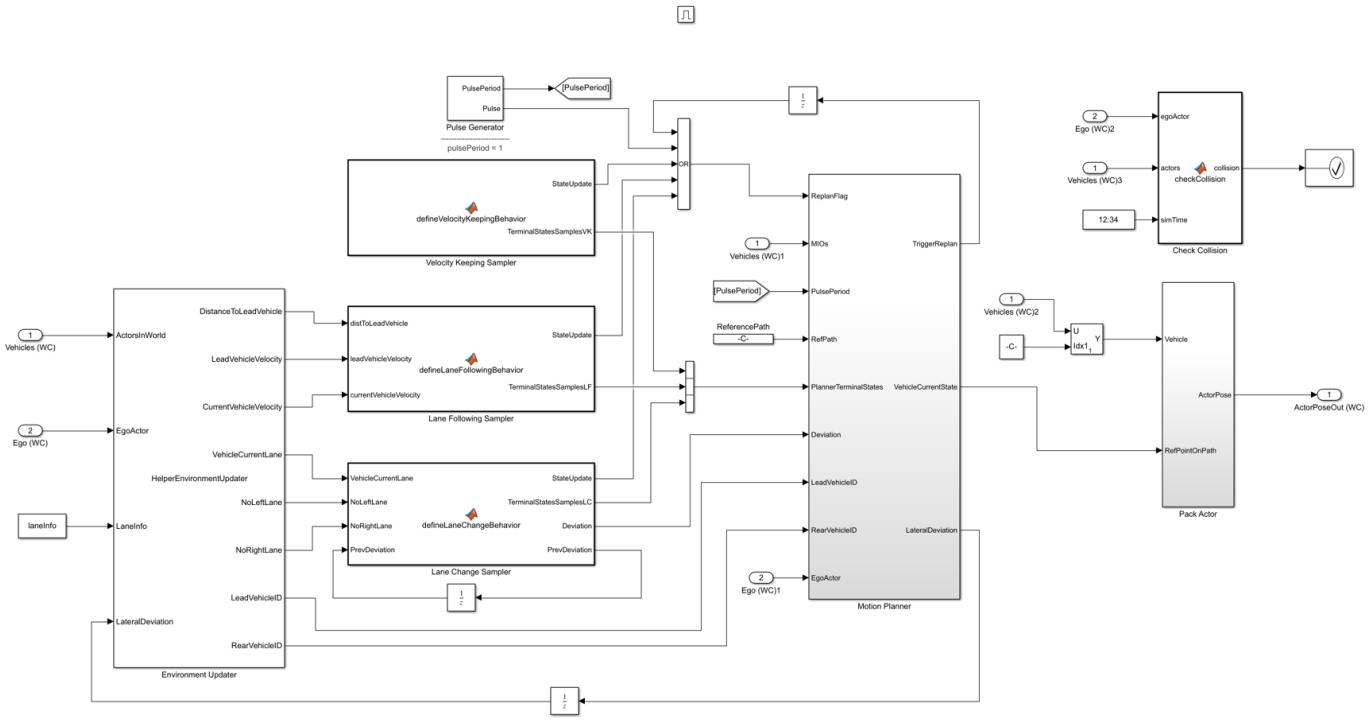
```
open_system("HighwayLaneFollowingWithIntelligentVehiclesTestBench/Simulation 3D Scenario/Intelli
```



The **Target Vehicle Behavior** subsystem allows you to switch between the default and other vehicle behaviors. If the behavior type for a target vehicle is set to `Default`, the subsystem configures the target vehicles to follow predefined trajectories. Otherwise, the position of the vehicle is dynamically computed and updated using the **Intelligent Vehicle** subsystem block. The **Intelligent Vehicle** subsystem block configures the `VelocityKeeping`, `LaneFollowing`, and `LaneChange` behaviors for the target vehicles.

Open **Intelligent Vehicle** subsystem.

```
open_system("HighwayLaneFollowingWithIntelligentVehiclesTestBench/Simulation 3D Scenario/Intelli
```



The **Intelligent Vehicle** subsystem computes the pose of a target vehicle by using information about the neighboring vehicles and the vehicle behavior. The subsystem is similar to the **Lane Change Planner** component of the "Highway Lane Change" on page 7-598 example. The **Intelligent Vehicle** subsystem has:

- The **Environment Updater** block computes the lead and rear vehicle information, current lane number, and existence of adjacent lanes (`NoLeftLane`, `NoRightLane`) with respect to the current state of the target vehicle. This block is configured by the System Object™ `HelperEnvironmentUpdater`.
- The **Velocity Keeping Sampler** block defines terminal states required for the `VelocityKeeping` behavior. This block reads the set velocity from the mask parameter `norm(TargetVehicle.Velocity)`.
- The **Lane Following Sampler** block defines terminal states required for the `LaneFollowing` behavior. This block reads the set velocity from the mask parameter `norm(TargetVehicle.Velocity)`.
- The **Lane Change Sampler** block defines terminal states required for the `LaneChange` behavior. This block also defines deviation offset from the reference path to keep the vehicle in a specific lane after a lane change. This block reads `TargetVehicle.Velocity`, `laneInfo`, and `TargetVehicle.InitialLaneID` from the base workspace by using mask parameters.

The table shows the configuration of terminal states and parameters for different vehicle behaviors:

Vehicle Behavior	Longitudinal States	Lateral States	Velocity States
Velocity Keeping	Constant set to 60 meters	Constant set to 0, to travel in the current lane	Configurable constant
Lane Following	Varies based on distance to the lead vehicle in the current lane of the vehicle	Constant set to 0, to travel in the current lane	Varies based on velocity of the lead vehicle in the current lane of the vehicle
Lane Change	Constant set to 60 meters	Varies based on the current lane of the lead vehicle and lane geometry	Configurable constant

- The **Check Collision** block checks for collision with any other vehicle in the scenario. The simulation stops if collision is detected.
- The **Pulse Generator** block defines the replan period for the **Motion Planner** subsystem. The default value is set to 1 second. Replanning can be triggered every pulse period, or if any of the samplers has a state update, or by the **Motion Planner** subsystem.
- The **MotionPlanner** subsystem generates trajectory for a target vehicle by using the terminal states defined by the vehicle behavior. It uses `trajectoryOptimalFrenet` (Navigation Toolbox) from to generate a trajectory. The subsystem estimates the position of the vehicle along its trajectory at every simulation step. This subsystem internally uses `HelperTrajectoryPlanner` system object to implement a fallback mechanism for different vehicle behaviors when `trajectoryOptimalFrenet` function is unable to generate a feasible trajectory.
- If the vehicle behavior is set to `LaneChange`, the trajectory planner attempts to generate a trajectory with `LaneFollowing` behavior. If it is unable to generate a trajectory, then stops the vehicle using stop behavior.
- If the vehicle behavior is set to `LaneFollowing` or `VelocityKeeping`, the trajectory planner stops the vehicle using stop behavior.

The system implements the stop behavior by constructing a trajectory with previous state of the vehicle that results in an immediate stop of the target vehicle.

Simulate Intelligent Target Vehicle Behavior on Straight Road

This example uses a test scenario that has three target vehicles (red sedan, black muscle car, and orange hatchback) and one ego vehicle (blue sedan) travelling on a straight road with two lanes.

- Red sedan is the first target vehicle and travels in the lane adjacent to the ego lane.
- Orange hatchback is a lead vehicle for the ego vehicle in the ego lane.
- Black muscle car is slow moving and a lead vehicle for the red sedan in the adjacent lane of the ego vehicle. The figure shows the initial positions of these vehicles.

You can run the simulation any number of times by changing the behavior type for each vehicle during each run. This example runs the simulation three times and at each run the behavior type for the first target vehicle is modified.

Configure All Target Vehicles Behavior to Velocity Keeping and Run Simulation

Run the setup script to configure `VelocityKeeping` behavior for all target vehicles.

```
helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup(...
        "scenario_LFACC_01_Straight_IntelligentVelocityKeeping");
```

Display the `BehaviorType` of all the target vehicles.

```
disp([targetVehicles(:).BehaviorType]');
```

```
    VelocityKeeping
    VelocityKeeping
    VelocityKeeping
    Default
    Default
```

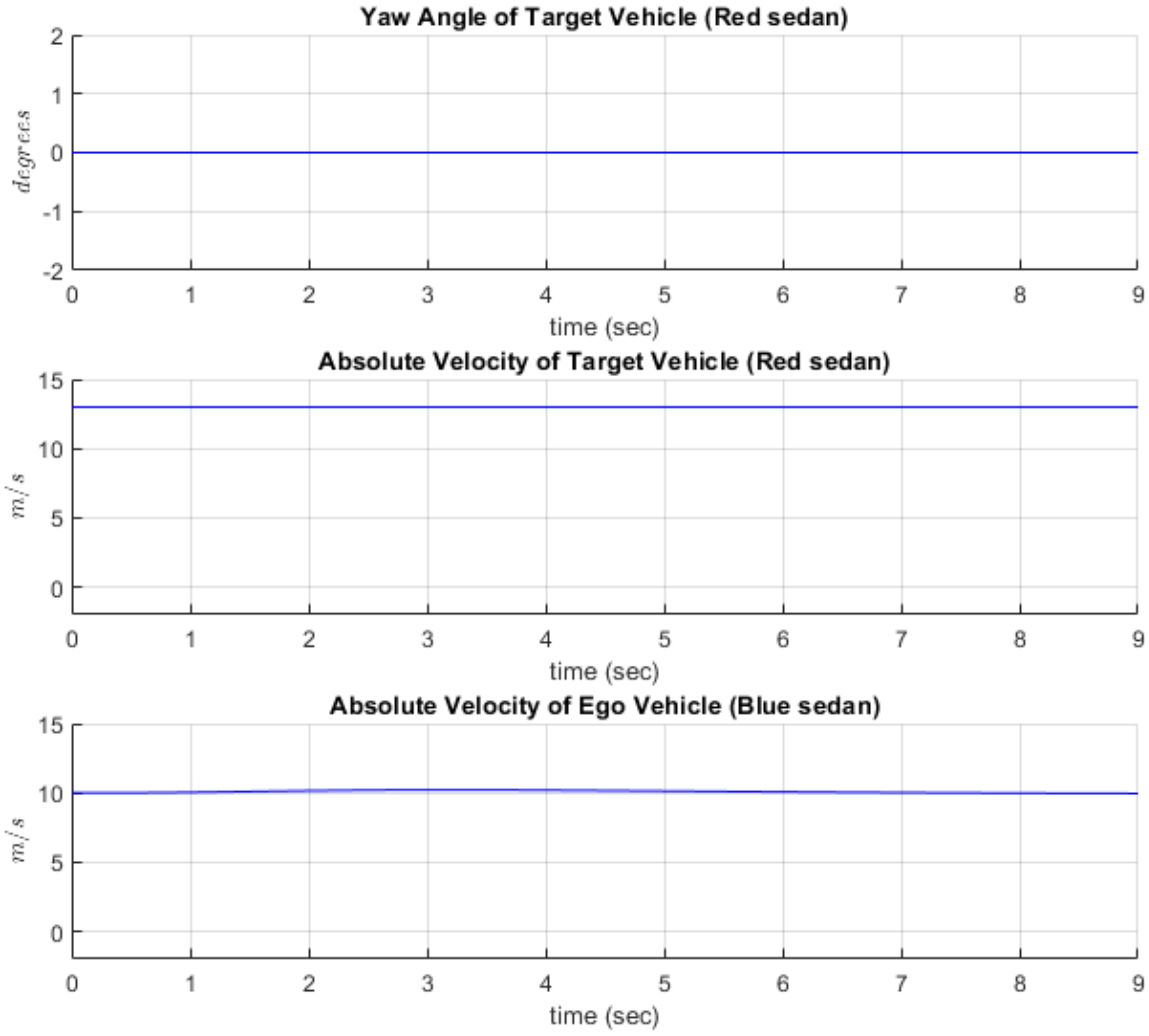
Run the simulation and visualize the results. The target vehicles in the scenario travels in their respective lanes at a constant velocity. The red sedan and the black muscle car maintains their velocity and does not check for collisions.

To reduce command-window output, turn off the model predictive control (MPC) update messages.

```
mpcverbosity('off');
% Run the model
simout = sim("HighwayLaneFollowingWithIntelligentVehiclesTestBench","StopTime","9");
```

Plot the velocity profiles of ego and first target vehicle (red sedan) to analyze the results.

```
hFigVK = helperPlotEgoAndTargetVehicleProfiles(simout.logsout);
```



- The **Yaw Angle of Target Vehicle (Red sedan)** plot shows the yaw angle of the red sedan. There is no variation in the yaw angle as the vehicle travels on a straight lane road.
- The **Absolute Velocity of Target Vehicle (Red sedan)** plot shows the absolute velocity of the red sedan. The velocity profile of the vehicle is constant as the vehicle is configured to `VelocityKeeping` behavior.
- The **Absolute Velocity of Ego Vehicle (Blue sedan)** plot shows that there is no effect of the red sedan on the ego vehicle as both vehicles travel in adjacent lanes.

Configure First Target Vehicle Behavior to Lane Following and Run Simulation

Configure the behavior type for first target vehicle(Red sedan) to lane following. Display the updated values for the `BehaviorType` of target vehicles.

```
targetVehicles(1).BehaviorType = VehicleBehavior.LaneFollowing;
disp([targetVehicles(:).BehaviorType]');
```

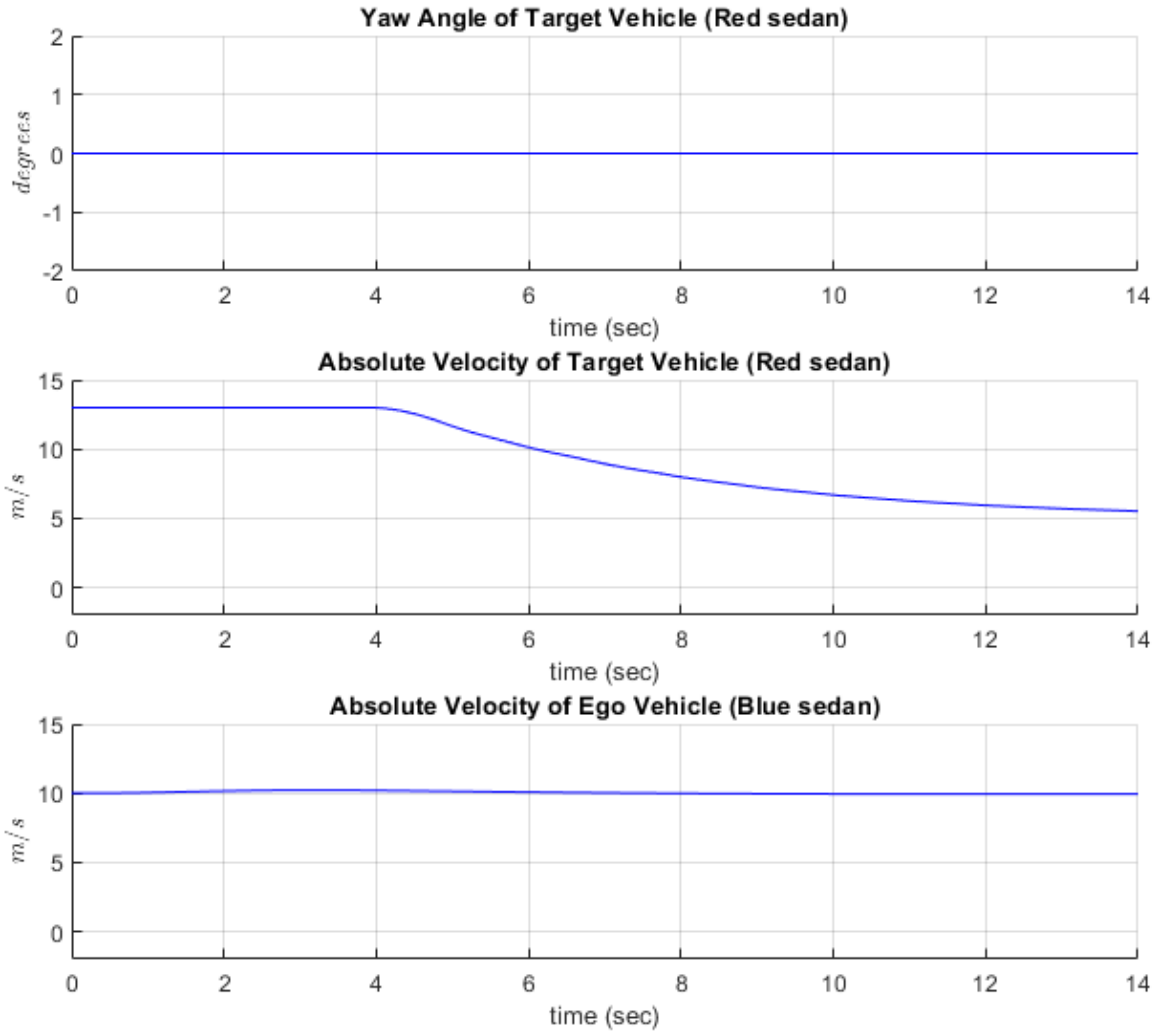
```
    LaneFollowing
    VelocityKeeping
    VelocityKeeping
    Default
    Default
```

Run the simulation and visualize the results. The target vehicles in the scenario are travelling in their respective lanes. The first target vehicle (red sedan) slows down to avoid colliding with the slow-moving black muscle car in its lane.

```
sim("HighwayLaneFollowingWithIntelligentVehiclesTestBench");
```

Plot the velocity profiles of ego and first target vehicle (red sedan) to analyze the results.

```
hFigLF = helperPlotEgoAndTargetVehicleProfiles(logsout);
```



- The **Yaw Angle of Target Vehicle (Red sedan)** plot is same as the one obtained in previous simulation. There is no variation in the yaw angle as the vehicle travels on a straight lane road.
- The **Absolute Velocity of Target Vehicle (Red sedan)** plot diverges from the previous simulation. The velocity of the red sedan gradually decreases from 13 m/s to 5 m/s to avoid colliding with the Black muscle car and maintains a safety gap.
- The **Absolute Velocity of Ego Vehicle (Blue sedan)** plot is same as the one in previous simulation. The ego vehicle is not affected by the change in the behavior of the Red sedan.

Configure First Target Vehicle Behavior to Lane Changing and Run Simulation

```
targetVehicles(1).BehaviorType = VehicleBehavior.LaneChange;
```

Display the `BehaviorType` of all the target vehicles.

```
disp([targetVehicles(:).BehaviorType]');
```

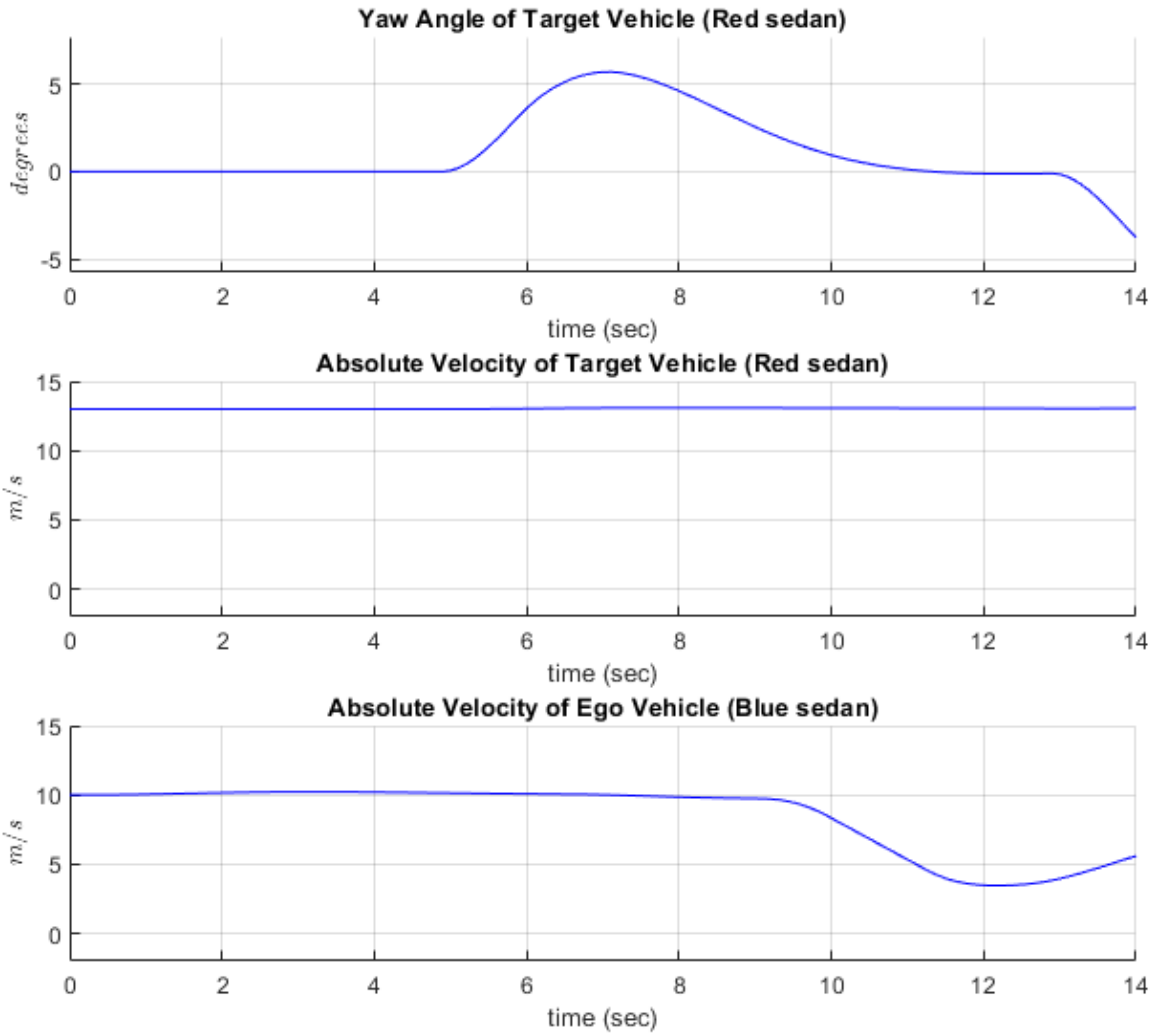
```
LaneChange
VelocityKeeping
VelocityKeeping
Default
Default
```

Run the simulation and visualize the results. The orange hatchback and black muscle cars are travelling at constant velocity in their respective lanes. The first target vehicle (red sedan) performs a lane change as it gets close to the black muscle car. It also does another lane change when it gets close to the orange hatchback.

```
sim("HighwayLaneFollowingWithIntelligentVehiclesTestBench");
```

Plot the velocity profiles of ego and first target vehicle (Red sedan) to analyze the results.

```
hFigLC = helperPlotEgoAndTargetVehicleProfiles(logsout);
```

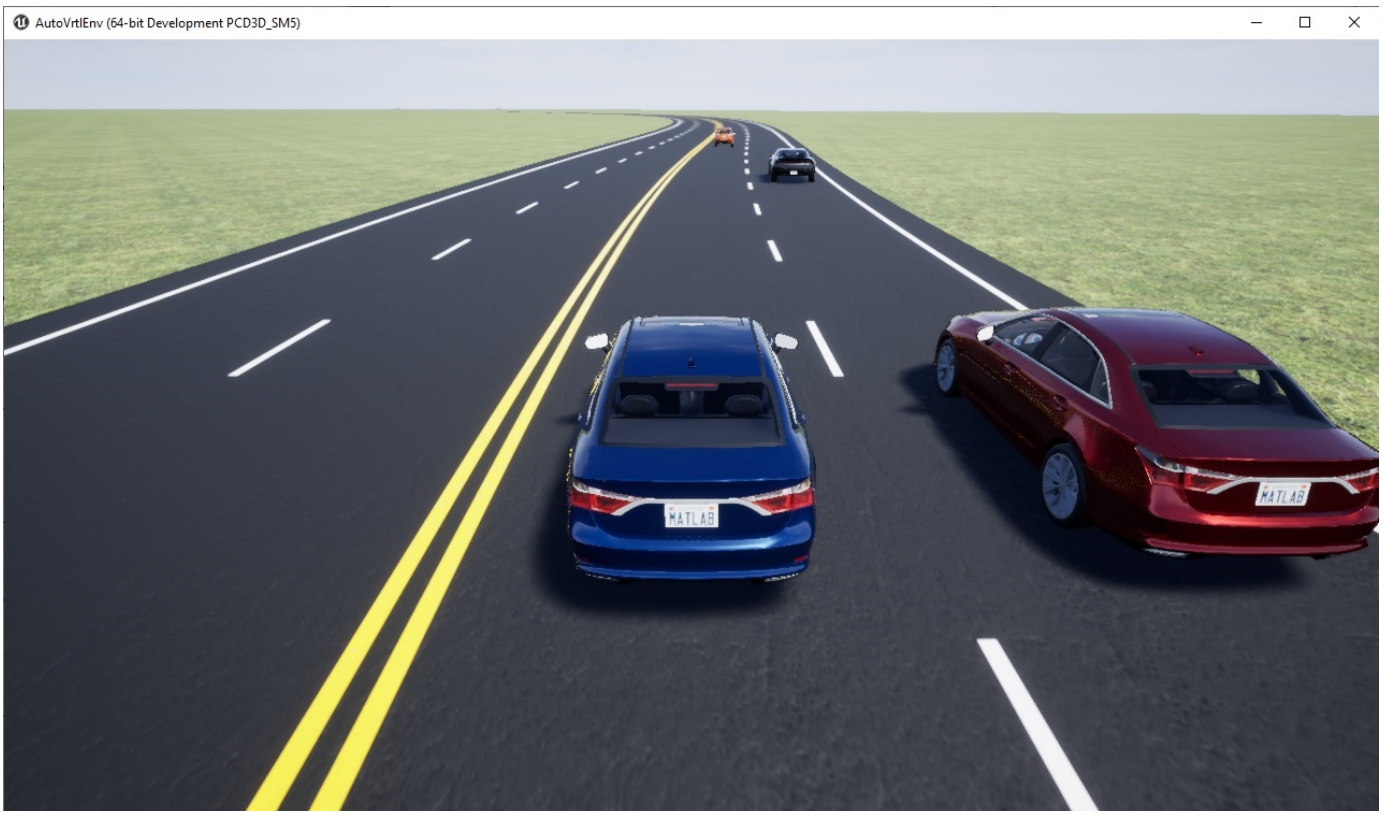


- The **Yaw Angle of Target Vehicle (Red sedan)** plot diverges from the previous simulation results. The yaw angle profile of the first target vehicle shows deviations as the vehicle perform a lane change.
- The **Absolute Velocity of Target Vehicle (Red sedan)** plot is similar to the `VelocityKeeping` behavior. The red sedan maintains a constant velocity even during lane change.
- The **Absolute Velocity of Ego Vehicle (Blue sedan)** plot shows the ego vehicle response to the lane change maneuver by the first target vehicle (red sedan). The velocity of the ego vehicle decreases as the red sedan change lanes. The red sedan moves to the ego lane and travels in front of the ego vehicle. The ego vehicle reacts by varying decreasing its velocity in order to travel in the same lane. Close all the figures.

```
close(hFigVK);
close(hFigLF);
close(hFigLC);
```

Simulate Intelligent Target Vehicle Behavior on Curved Road

Test the model on a scenario with curved roads. The vehicle configuration and position of vehicles are similar to the previous simulation. The test scenario contains a curved road and the first target vehicle (Red sedan) is configured to `LaneChange` behavior. The other two target vehicles are configured to `VelocityKeeping` behavior. The figure below shows the initial positions of the vehicles in the curved road scene.

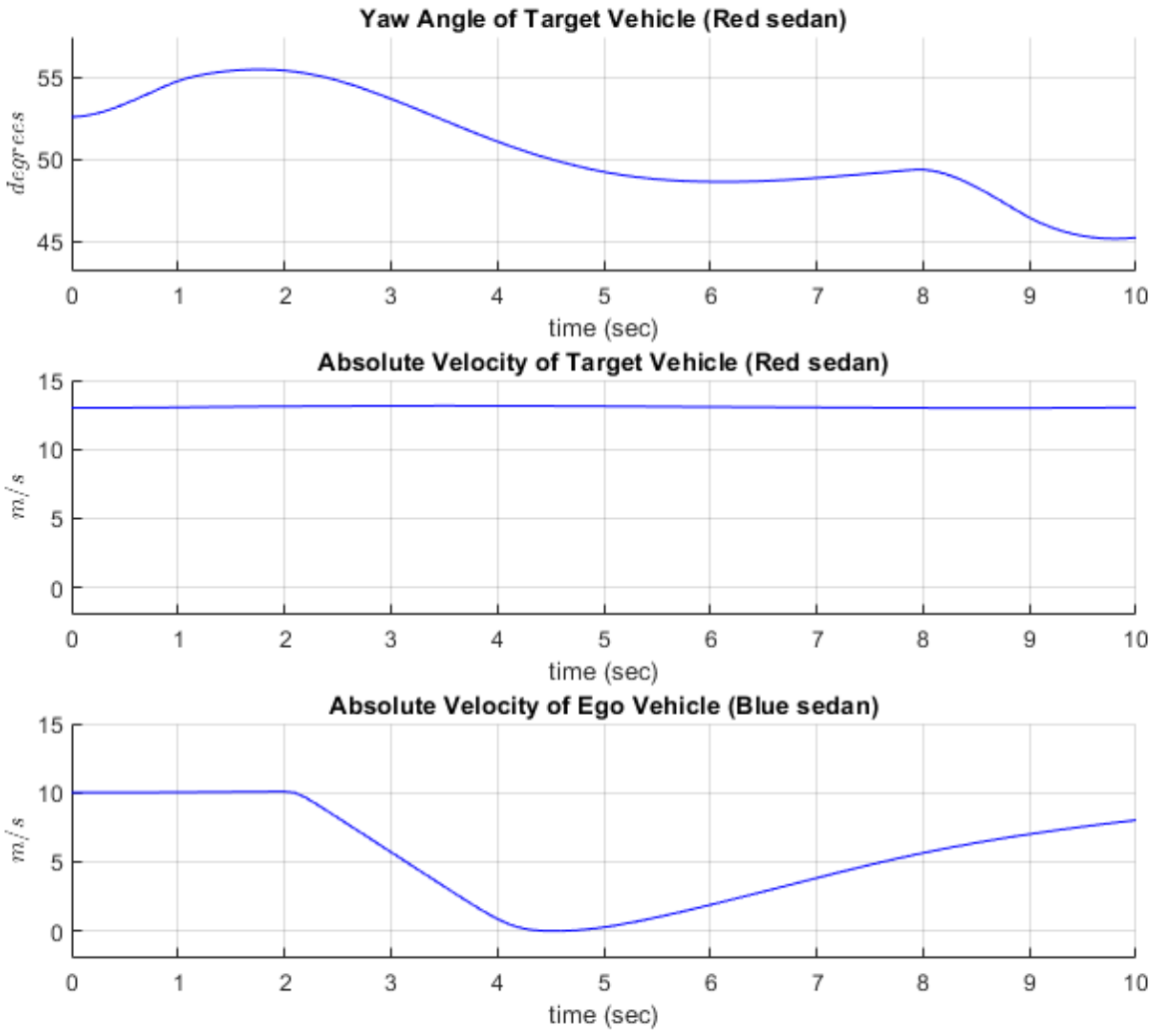


Run the setup script to configure the model parameters.

```
helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup(...
        "scenario_LFACC_04_Curved_IntelligentLaneChange");
```

Run simulation and visualize the results. Plot the yaw angle and velocity profiles of ego and target vehicles.

```
sim("HighwayLaneFollowingWithIntelligentVehiclesTestBench");
hFigCurvedLC = helperPlotEgoAndTargetVehicleProfiles(logsout);
```



- The **Yaw Angle of Target Vehicle (Red sedan)** plot shows variation in the profile as the red sedan performs lane change on a curved road. The curvature of the road also impacts the yaw angle of the target vehicle.
- The **Absolute Velocity of Target Vehicle (Red sedan)** plot is similar to the `VelocityKeeping` behavior, as the red sedan maintains a constant velocity during lane change on a curved road.

- The **Absolute Velocity of Ego Vehicle (Blue sedan)** plot shows the response of the ego vehicle to the lane change maneuver by red sedan. The ego vehicle reacts by varying decreasing its velocity in order to travel in the same lane.

Close the figure.

```
close(hFigCurvedLC);
```

Explore Other Scenarios

This example provides additional scenarios that are compatible with the `HighwayLaneFollowingWithIntelligentVehiclesTestBench` model. Below is a list of compatible scenarios that are provided with this example.

- `scenario_LFACC_01_Straight_IntelligentVelocityKeeping` function configures the test scenario such that all the target vehicles are configured to perform `VelocityKeeping` behavior on a straight road.
- `scenario_LFACC_02_Straight_IntelligentLaneFollowing` function configures the test scenario such that the red sedan performs `LaneFollowing` behavior while all other target vehicles perform `VelocityKeeping` behavior on a straight road.
- `scenario_LFACC_03_Straight_IntelligentLaneChange` function configures the test scenario such that the red sedan performs `LaneChange` behavior while all other target vehicles perform `VelocityKeeping` behavior on a straight road.
- `scenario_LFACC_04_Curved_IntelligentLaneChange` function configures the test scenario such that the red sedan performs `LaneChange` behavior while all other target vehicles perform `VelocityKeeping` behavior on a curved road. This is configured as the default scenario.
- `scenario_LFACC_05_Curved_IntelligentDoubleLaneChange` function configures the test scenario such that the red sedan performs `LaneChange` behavior while all other target vehicles perform `VelocityKeeping` behavior on a curved road. The placement of other vehicles in this scenario is such that the red sedan performs a double lane change during the simulation.

For more details on the road and target vehicle configurations in each scenario, view the comments in each file. You can configure the Simulink model and workspace to simulate these scenarios using the `helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup` function.

```
helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup('scenario_LFACC_05_Curved_IntelligentDou
```

Conclusion

This example demonstrates how to test the functionality of a lane following application in a scenario with an ego vehicle and multiple intelligent target vehicles.

Enable the MPC update messages again.

```
mpcverbosity('on');
```

See Also

Scenario Reader | Cuboid To 3D Simulation | Simulation 3D Vehicle with Ground Following | Simulation 3D Scene Configuration | Vehicle To World | `trajectoryOptimalFrenet` (Navigation Toolbox)

More About

- “Highway Lane Following with RoadRunner Scene” on page 7-736
- “Highway Lane Following” on page 7-653
- “Highway Lane Change” on page 7-598