Automated Driving Toolbox™

Reference

MATLAB® & SIMULINK®

R2020**b**

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™ Reference

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

Apps

Bird's-Eye Scope

Visualize sensor coverages, detections, and tracks

Description

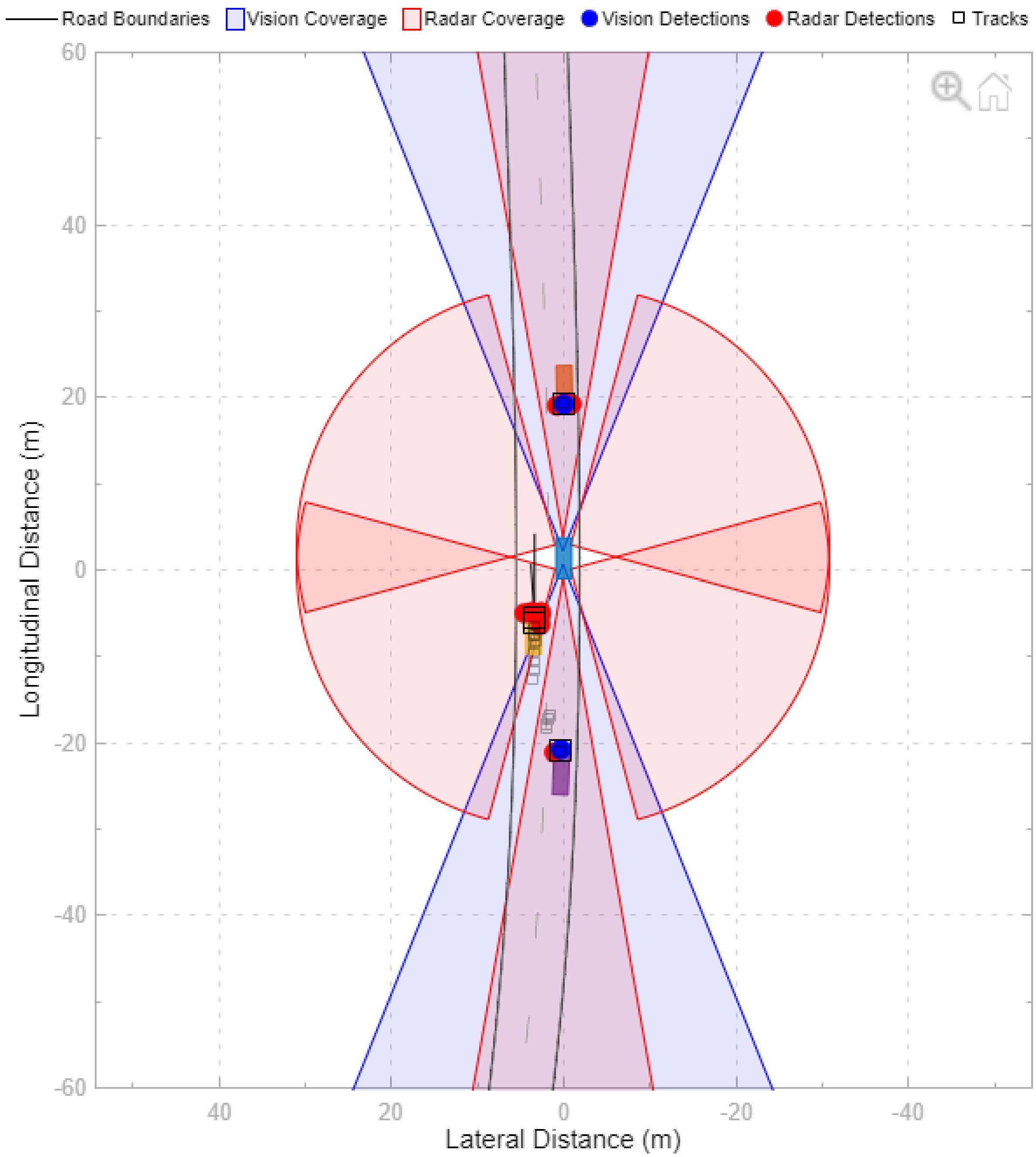
The **Bird's-Eye Scope** visualizes aspects of a driving scenario found in your Simulink® model.

Using the scope, you can:

- Inspect the coverage areas of radar, vision, and lidar sensors.
- Analyze the sensor detections of actors, road boundaries, and lane boundaries.
- Analyze the tracking results of moving actors within the scenario.

To get started, open the scope and click **Find Signals**. The scope updates the block diagram, finds signals representing aspects of the driving scenario, organizes the signals into groups, and displays the signals. You can then analyze the signals as you simulate, organize the signals into new groups, and modify the graphical display of the signals.

For more details about using the scope, see “Visualize Sensor Data and Tracks in Bird's-Eye Scope”.



Open the Bird's-Eye Scope App

Simulink Toolstrip:

- On the **Simulation** tab, under **Review Results**, click **Bird's-Eye Scope**.
- On the **Apps** tab, under **Signal Processing and Wireless Communications**, click **Bird's-Eye Scope**.

Examples

- "Visualize Sensor Data and Tracks in Bird's-Eye Scope"
- "Visualize Sensor Data from Unreal Engine Simulation Environment"
- "Sensor Fusion Using Synthetic Radar and Vision Data in Simulink"
- "Lane Following Control with Sensor Fusion and Lane Detection"
- "Autonomous Emergency Braking with Sensor Fusion"
- "Test Open-Loop ADAS Algorithm Using Driving Scenario"
- "Test Closed-Loop ADAS Algorithm Using Driving Scenario"

Parameters

Settings

To access the settings of the **Bird's-Eye Scope**, on the scope toolstrip, click **Settings**.

Vehicle Coordinates View Settings

`Longitudinal axis limits` — Longitudinal axis limits
`[-60,60]` (default) | [*min*, *max*] vector

Longitudinal axis limits, specified as a [*min*, *max*] vector.

Tunable: Yes

`Lateral axis limits` — Lateral axis limits
`[-30,30]` (default) | [*min*, *max*] vector

Lateral axis limits, specified as a [*min*, *max*] vector.

Tunable: Yes

`Track position selector` — Selection matrix used to extract positions of tracked objects
`[1,0,0,0,0,0; 0,0,1,0,0,0]` (default) | 2-by-*n* matrix of zeros and ones

Selection matrix used to extract the positions of tracked objects, specified as a 2-by-*n* matrix of zeros and ones. *n* is the size of the state vector for each tracked object in the scenario. The scope multiplies the selection matrix by the state vector of a tracked object to return the (*x*, *y*) position of the object.

- The first row of the matrix corresponds to the *x*-coordinate stored within the state vector.
- The second row of the matrix corresponds to the *y*-coordinate stored within the state vector.

This parameter applies to signals from a Multi-Object Tracker block that were initialized by a linear Kalman filter. The state vector format depends on the motion model used to initialize the Kalman filter. For more details on these motion models, see `trackingKF` and "Linear Kalman Filters".

The default selection matrix is for a 3-D constant velocity motion model. In this motion model, the state vectors of tracked objects are of the form `[x;vx;y;vy;z;vz]`, where:

- `x` is the *x*-coordinate of a tracked object.

- vx is the velocity of a tracked object in the *x*-direction.
- y is the *y*-coordinate of a tracked object.
- vy is the velocity of a tracked object in the *y*-direction.
- z is the *z*-coordinate of a tracked object.
- vz is the velocity of a tracked object in the *z*-direction.

Multiplying the state vector by this selection matrix returns only the first element of the state vector, x, and the third element of the state vector, y.

```
[1,0,0,0,0,0; 0,0,1,0,0,0] * [x;vx;y;vy;z;vz] = [x;y]
```

Tunable: No

Track velocity selector — Selection matrix used to extract velocities of tracked objects
[0,1,0,0,0,0; 0,0,0,1,0,0] (default) | 2-by-*n* matrix of zeros and ones

Selection matrix used to extract the velocities of tracked objects, specified as a 2-by-*n* matrix of zeros and ones. *n* is the size of the state vector for each tracked object in the scenario. The scope multiplies the selection matrix by the state vector of a tracked object to return the velocity of the object in the (*x*, *y*) direction.

- The first row of the matrix corresponds to the *x*-direction velocity stored within the state vector.
- The second row of the matrix corresponds to the *y*-direction velocity stored within the state vector.

This parameter applies to signals from a Multi-Object Tracker block that were initialized by a linear Kalman filter. The state vector format depends on the motion model used to initialize the Kalman filter. For more details on these motion models, see trackingKF and "Linear Kalman Filters".

The default selection matrix is for a 3-D constant velocity motion model. In this motion model, the state vectors of tracked objects are of the form [x;vx;y;vy;z;vz], where:

- x is the *x*-coordinate of a tracked object.
- vx is the velocity of a tracked object in the *x*-direction.
- y is the *y*-coordinate of a tracked object.
- vy is the velocity of a tracked object in the *y*-direction.
- z is the *z*-coordinate of a tracked object.
- vz is the velocity of a tracked object in the *z*-direction.

Multiplying the state vector by this selection matrix returns only the second element of the state vector, vx, and the fourth element of the state vector, vy.

```
[0,1,0,0,0,0; 0,0,0,1,0,0] * [x;vx;y;vy;z;vz] = [vx;vy]
```

Tunable: No

Global Settings

Display short signal names — Display signal names without path information
on (default) | off

- Select this parameter to display short signal names (signals without path information).

- Clear this parameter to display long signal names (signals with path information).

Consider the signal `VisionDetection` within subsystem `Sensor Simulation`. When you select this parameter, the short name, `VisionDetection`, is displayed. When you clear this parameter, the long name, `Sensor Simulation/VisionDetection`, is displayed.

Tunable: Yes

Signal Properties

These properties are a subset of the available signal properties. To view all the properties of a signal, first select that signal from the left pane. Then, on the scope toolstrip, click **Properties**.

`Alpha` — Transparency of coverage area

`0.1` (default) | real scalar in the range [0, 1]

Transparency of the coverage area, specified as a real scalar in the range [0, 1]. A value of 0 makes the coverage area fully transparent. A value of 1 makes the coverage area fully opaque.

This property is available only for signals in the **Sensor Coverage** group.

Tunable: Yes

`Velocity Scaling` — Scale factor for magnitude length of velocity vectors

`1` (default) | real scalar in the range [0, 20]

Scale factor for the magnitude length of the velocity vectors, specified as a real scalar in the range [0, 20]. The scope renders the magnitude vector value as M × **Velocity Scaling**, where M is the magnitude of the velocity.

This property is available only for signals in the **Detections** or **Tracks** groups.

Tunable: Yes

Limitations

General Limitations

- Referenced models are not supported. To visualize signals that are within referenced models, move the output of these signals to the top-level model.
- Rapid accelerator mode is not supported.
- If you initialize your model in fast restart, then after the first time you simulate, the **Find Signals** button is disabled. To enable **Find Signals** again, on the **Debug** tab of the Simulink toolstrip, click **Fast Restart**.

Scenario Reader Block Limitations

- The **Bird's-Eye Scope** does not support visualization in a model that contains:
  - More than one Scenario Reader block.
  - A Scenario Reader block within a nonvirtual subsystem, such as an atomic or enabled subsystem.
  - A Scenario Reader block that is configured to output actors and lane boundaries in world coordinates (**Coordinate system of outputs** parameter set to `World Coordinates`).

- For Scenario Reader blocks in which you specify the ego vehicle using the **Ego Vehicle** input port, the ego vehicle signal must be connected directly to the block. Visualization of ego vehicle signals that are output from a nonvirtual subsystem or referenced model are not supported.

3D Simulation Block Limitations

- The visualization of roads, lanes, and actors from Simulation 3D Scene Configuration blocks is not supported. If your block contains a Simulation 3D Scene Configuration block, the **Bird's-Eye Scope** still displays an ego vehicle, but it has default vehicle dimensions.

More About

Applicable Signals

When the **Bird's-Eye Scope** finds signals in your model, it automatically groups signals by type. These groupings are based on the sources of the signals 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" on page 1-9.	• Scenario Reader block

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" on page 1-9.
		• 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.

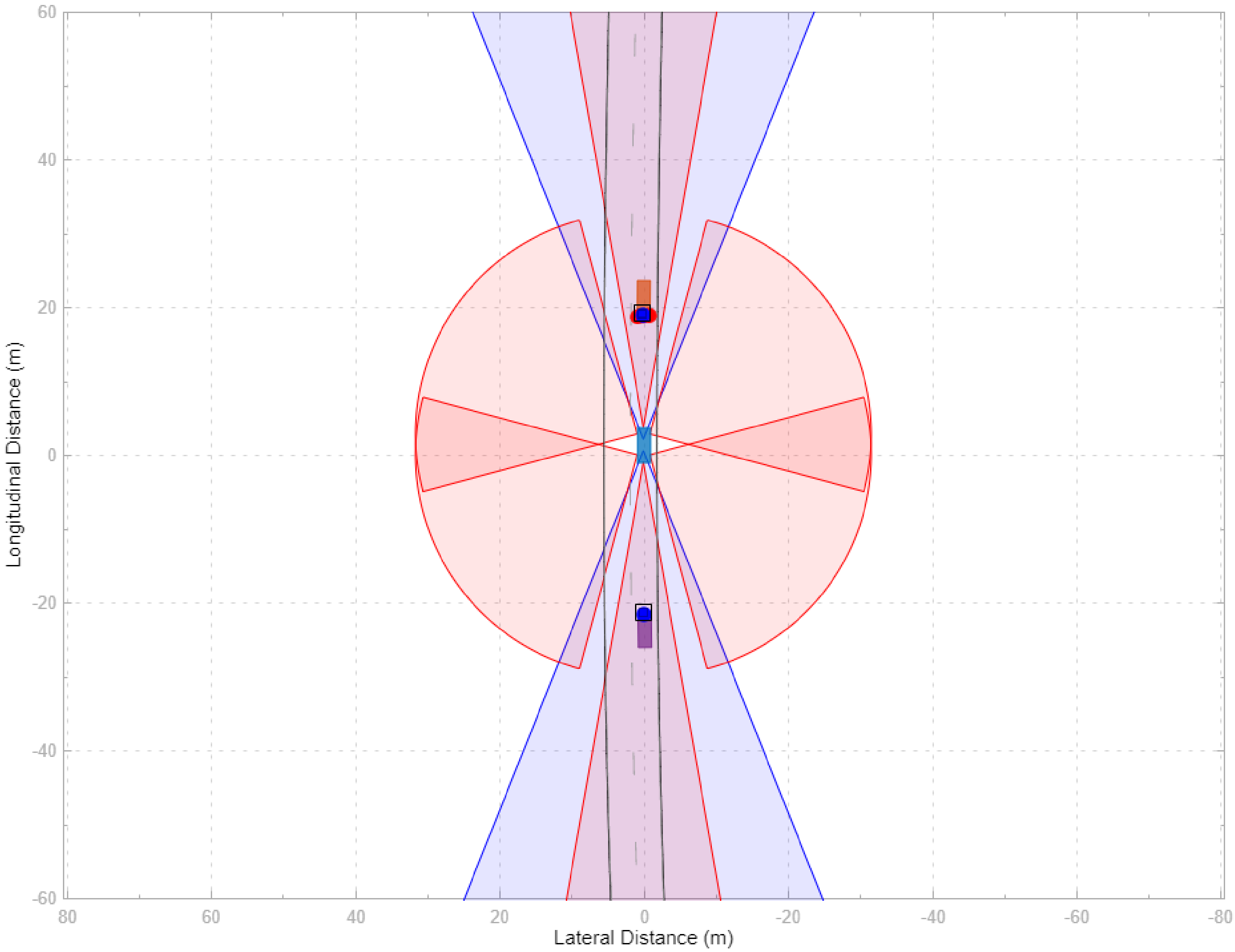
Signal Group	Description	Signal Sources
Sensor Coverage	Coverage areas of vision, radar, and lidar sensors, sorted into **Vision**, **Radar**, and **Lidar** subgroups You can modify signals in this group. You can rename or delete subgroups but not the top-level **Sensor Coverage** group. You can also 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
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

Signal Group	Description	Signal Sources
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.

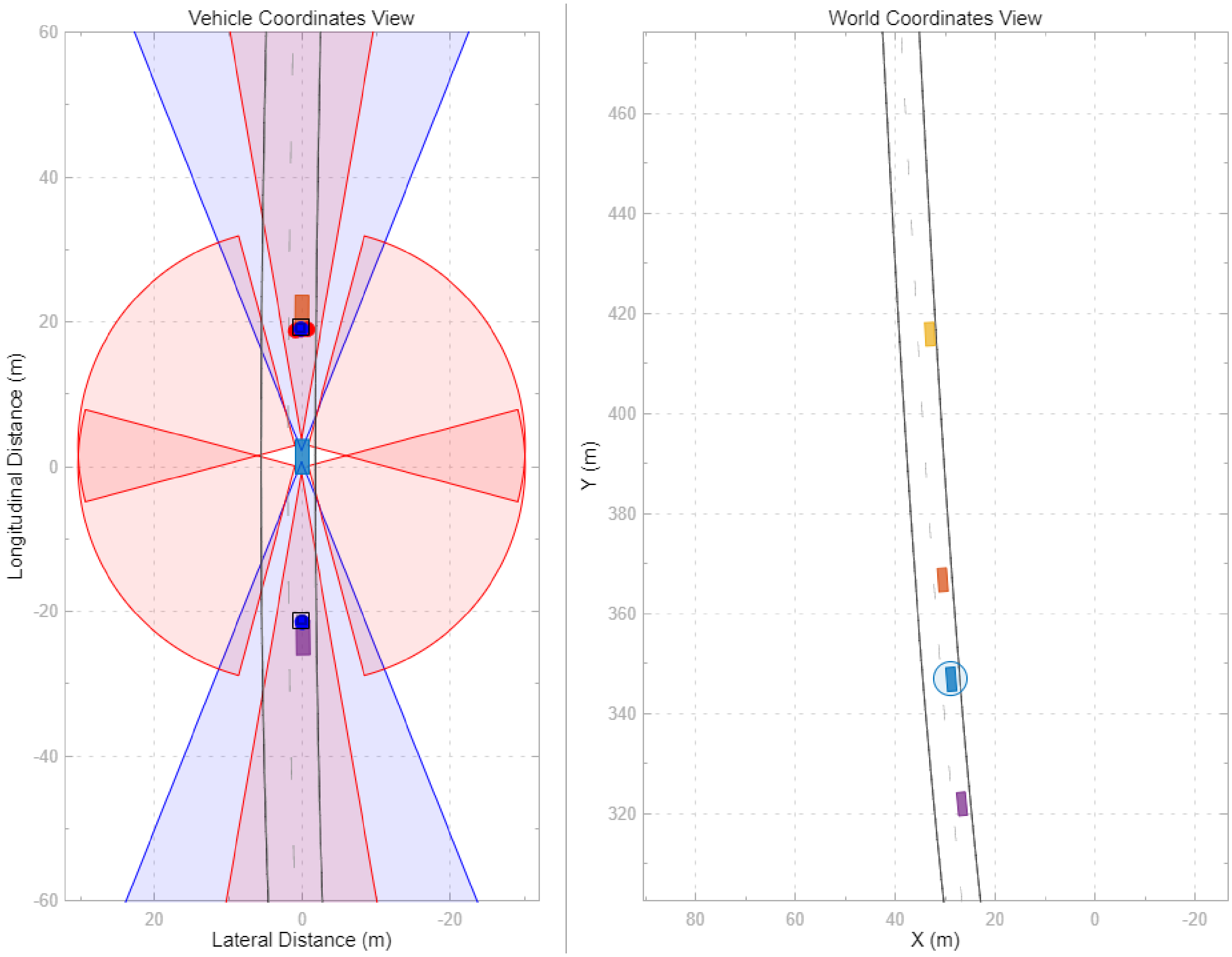
To view a model that includes samples of all these signals types, see the “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” example.

Vehicle and World Coordinate Views

In the **Bird's-Eye Scope**, the default view displays the driving scenario in vehicle coordinates. During simulation, this view displays the scenario from the perspective of the ego vehicle. Use this view to inspect aspects of the scenario in the immediate vicinity of the ego vehicle.



You can also display the driving scenario in world coordinates. On the scope toolstrip, click **World Coordinates** to open the **World Coordinates View** window. Use this window to view the scenario as a whole. You can also use this view to inspect the trajectories of actors that are not in the immediate vicinity of the ego vehicle.



To display the roads and lanes within the **World Coordinates View**, click **Find Signals**. To display the ego vehicle and other actors in the scenario, run the simulation. This view does not display detections, tracks, sensor coverage areas, and other applicable signals. You can view these signals only in the **Vehicle Coordinates View** window.

Note In the **World Coordinates View** window, the circle around the ego vehicle highlights the location of the vehicle in the scenario. It is not a sensor coverage area.

Tips

- To find the source of a signal within the model, in the left pane of the scope, right-click a signal and select **Highlight in Model**.
- You can show or hide signals while simulating. For example, to hide a sensor coverage, first select it from the left pane. Then, from the **Properties** tab, clear the **Show Sensor Coverage** check box.

- When you reopen the scope after saving and closing a model, the scope canvas is initially blank. Click **Find Signals** to find the signals again. The signals have the same properties from when you last saved the model.
- If the simulation runs too quickly, you can slow it down by using simulation pacing. On the **Simulation** tab of the Simulink toolstrip, select **Run > Simulation Pacing**. Then, select the **Enable pacing to slow down simulation** check box and decrease the simulation time to less than the default of one second per wall clock second.
- To better inspect the scenario, you can pan and zoom within the **Vehicle Coordinates View** and **World Coordinates View** windows. To return to the default display of either window, in the upper-right corner of that window, click the home button .

See Also

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

Topics

"Visualize Sensor Data and Tracks in Bird's-Eye Scope"
"Visualize Sensor Data from Unreal Engine Simulation Environment"
"Sensor Fusion Using Synthetic Radar and Vision Data in Simulink"
"Lane Following Control with Sensor Fusion and Lane Detection"
"Autonomous Emergency Braking with Sensor Fusion"
"Test Open-Loop ADAS Algorithm Using Driving Scenario"
"Test Closed-Loop ADAS Algorithm Using Driving Scenario"

Introduced in R2018b

Driving Scenario Designer

Design driving scenarios, configure sensors, and generate synthetic data

Description

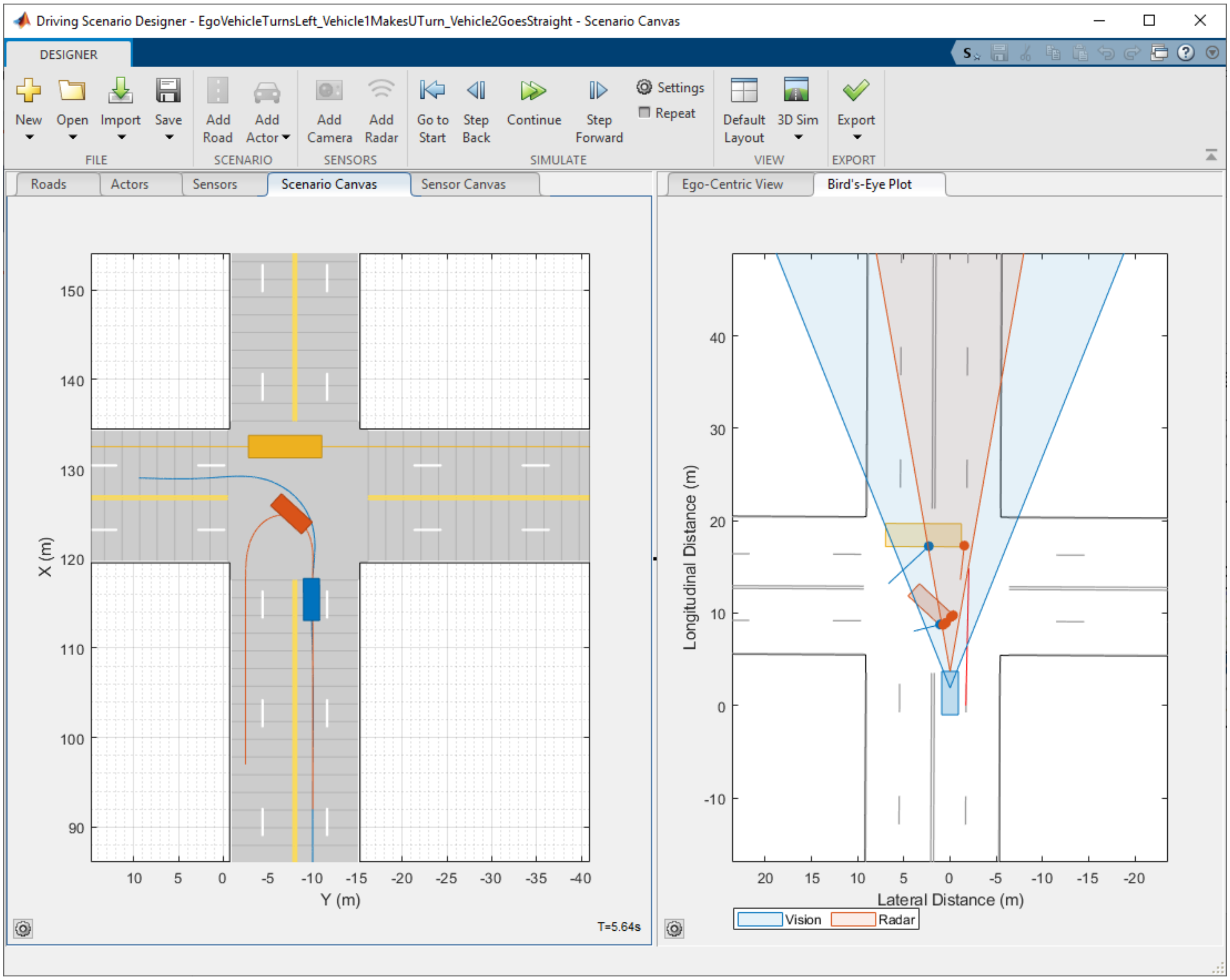
The **Driving Scenario Designer** app enables you to design synthetic driving scenarios for testing your autonomous driving systems.

Using the app, you can:

- Create road and actor models using a drag-and-drop interface.
- Configure vision, radar, and lidar sensors mounted on the ego vehicle. You can use these sensors to simulate detections of actors and lane boundaries in the scenario and to generate point cloud data from a scenario.
- Load driving scenarios representing European New Car Assessment Programme (Euro NCAP®) test protocols [1][2][3] and other prebuilt scenarios.
- Import OpenDRIVE® roads and lanes into a driving scenario. The app supports OpenDRIVE format specification version 1.4H [4].
- Import road data from OpenStreetMap® or HERE HD Live Map[1] web services into a driving scenario.
- Export the road network in a driving scenario to an OpenDRIVE file.
- Export synthetic sensor detections to MATLAB®.
- Generate MATLAB code of the scenario and sensors, and then programmatically modify the scenario and import it back into the app for further simulation.
- Generate a Simulink model from the scenario and sensors, and use the generated models to test your sensor fusion or vehicle control algorithms.

To learn more about the app, see Driving Scenario Designer.

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.



Open the Driving Scenario Designer App

- MATLAB Toolstrip: On the **Apps** tab, under **Automotive**, click the app icon.
- MATLAB command prompt: Enter `drivingScenarioDesigner`.

Examples

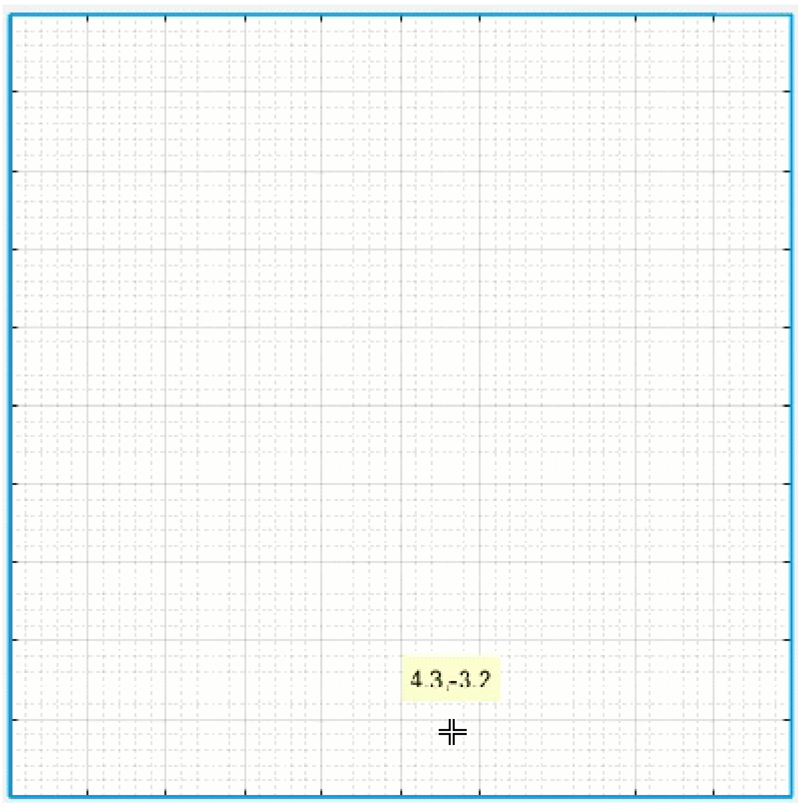
Create a Driving Scenario

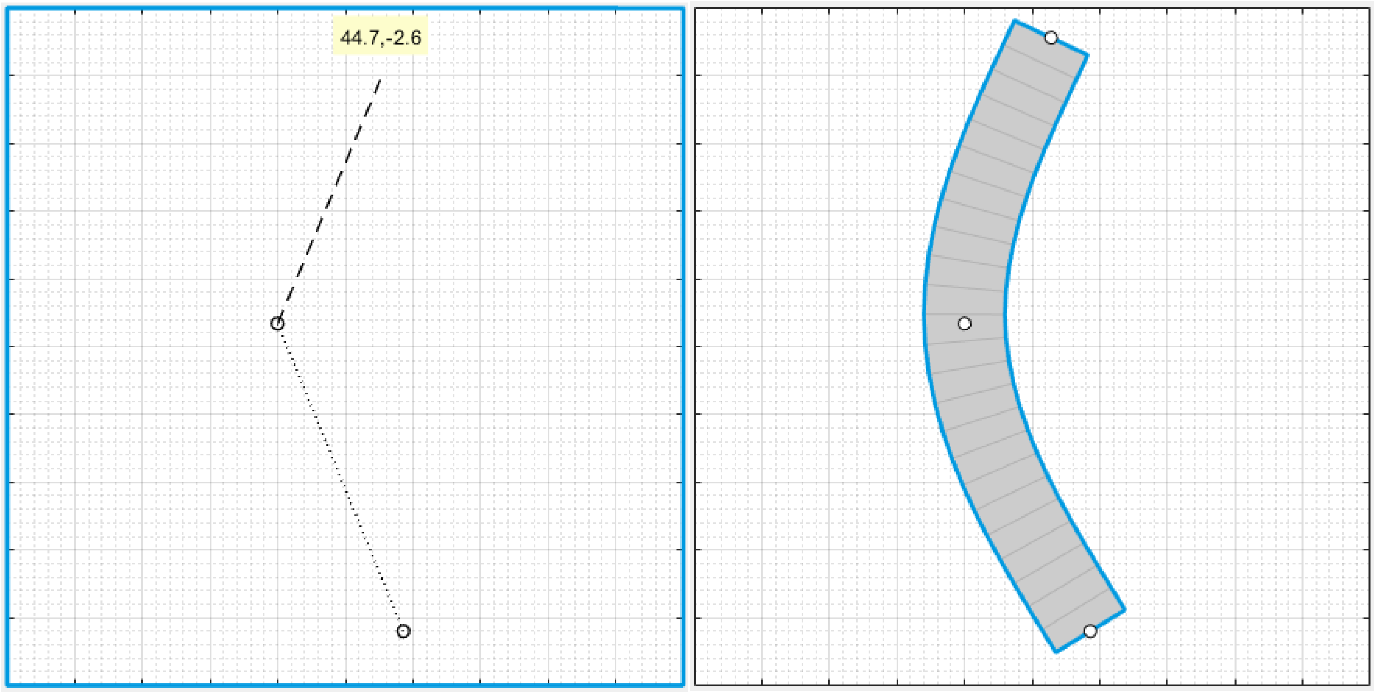
Create a driving scenario of a vehicle driving down a curved road, and export the road and vehicle models to the MATLAB workspace. For a more detailed example of creating a driving scenario, see “Create Driving Scenario Interactively and Generate Synthetic Sensor Data”.

Open the **Driving Scenario Designer** app.

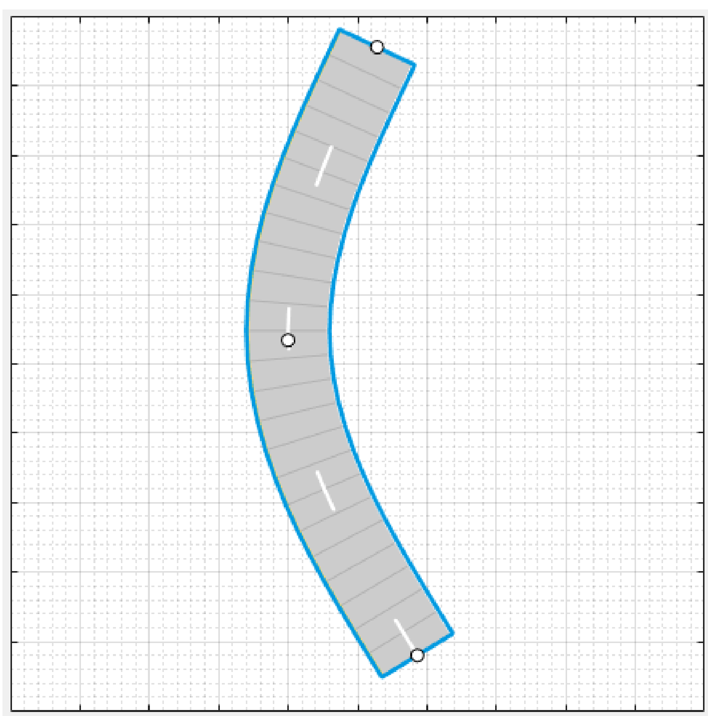
```
drivingScenarioDesigner
```

Create a curved road. On the app toolstrip, click **Add Road**. Click the bottom of the canvas, extend the road path to the middle of the canvas, and click the canvas again. Extend the road path to the top of the canvas, and then double-click to create the road. To make the curve more complex, click and drag the road centers (open circles), or double-click the road to add more road centers.

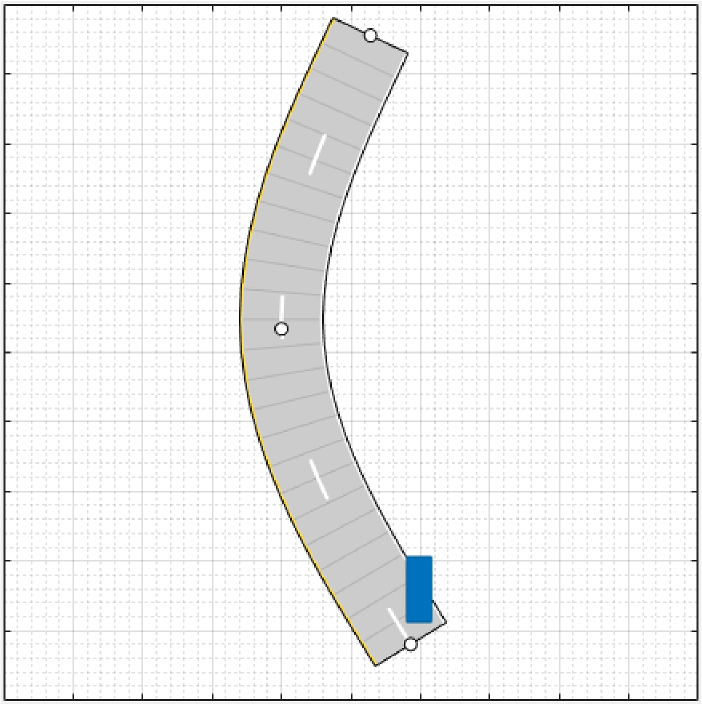




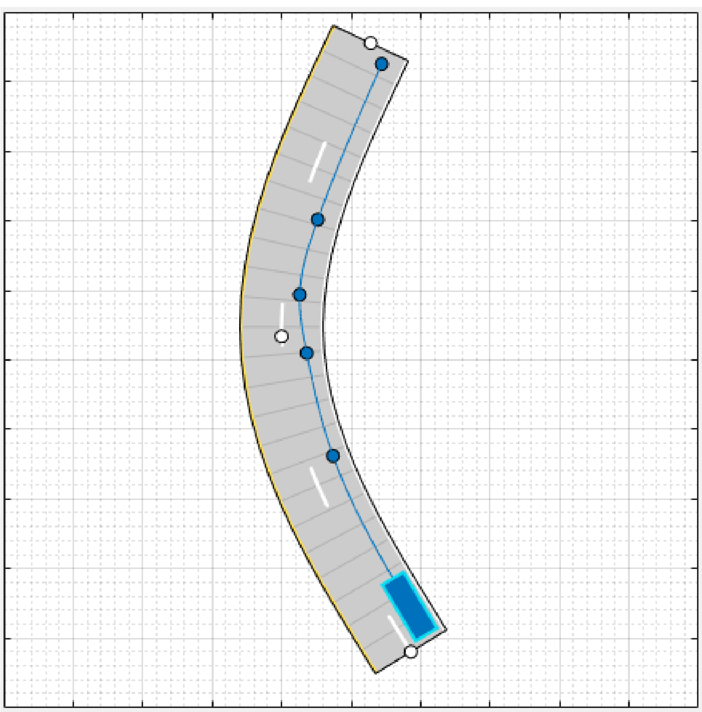
Add lanes to the road. In the left pane, on the **Roads** tab, expand the **Lanes** section. Set **Number of Lanes** to 2. By default, the road is one-way and has solid lane markings on either side to indicate the shoulder.



Add a vehicle at one end of the road. On the app toolstrip, select **Add Actor** > **Car**. Then click the road to set the initial position of the car.



Set the driving trajectory of the car. Right-click the car, select **Add Forward Waypoints**, and add waypoints for the car to pass through. After you add the last waypoint, press **Enter**. The car autorotates in the direction of the first waypoint.



Adjust the speed of the car as it passes between waypoints. In the **Waypoints, Speeds, Wait Times, and Yaw** table in the left pane, set the velocity, **v (m/s)**, of the ego vehicle as it enters each waypoint segment. Increase the speed of the car for the straight segments and decrease its speed for the curved segments. For example, the trajectory has six waypoints, set the **v (m/s)** cells to 30, 20, 15, 15, 20, and 30.

v (m/s)
30
20
15
15
20
30

Run the scenario, and adjust settings as needed. Then click **Save > Roads & Actors** to save the road and car models to a MAT-file.

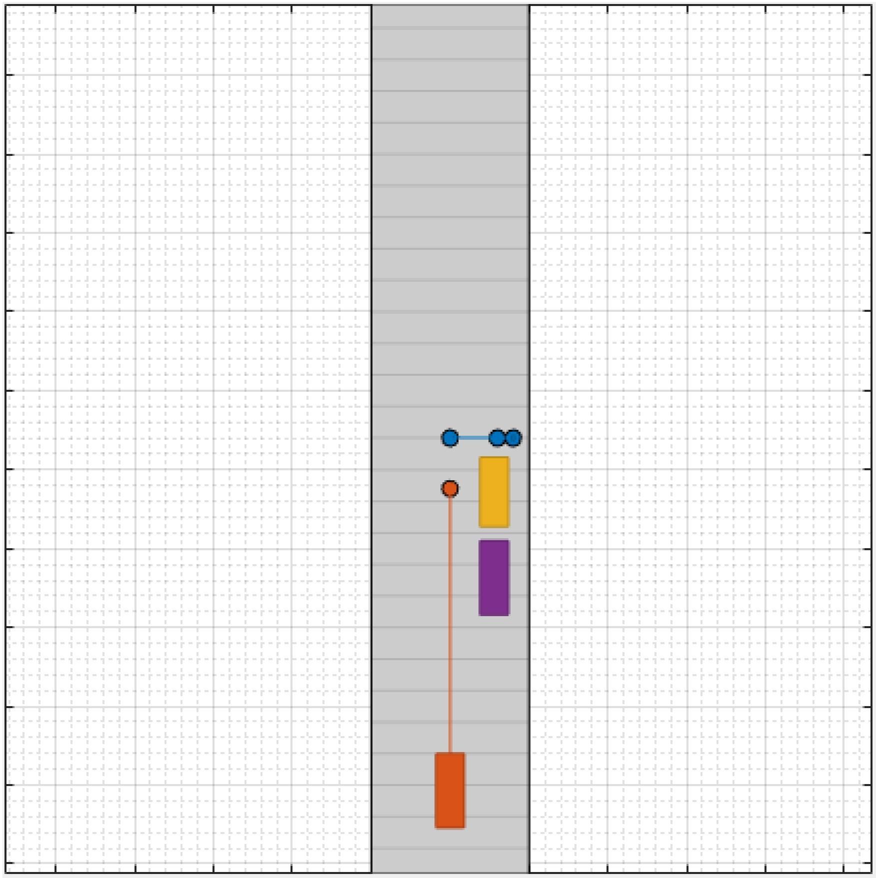
Generate Sensor Data from Scenario

Generate lidar point cloud data from a prebuilt Euro NCAP driving scenario.

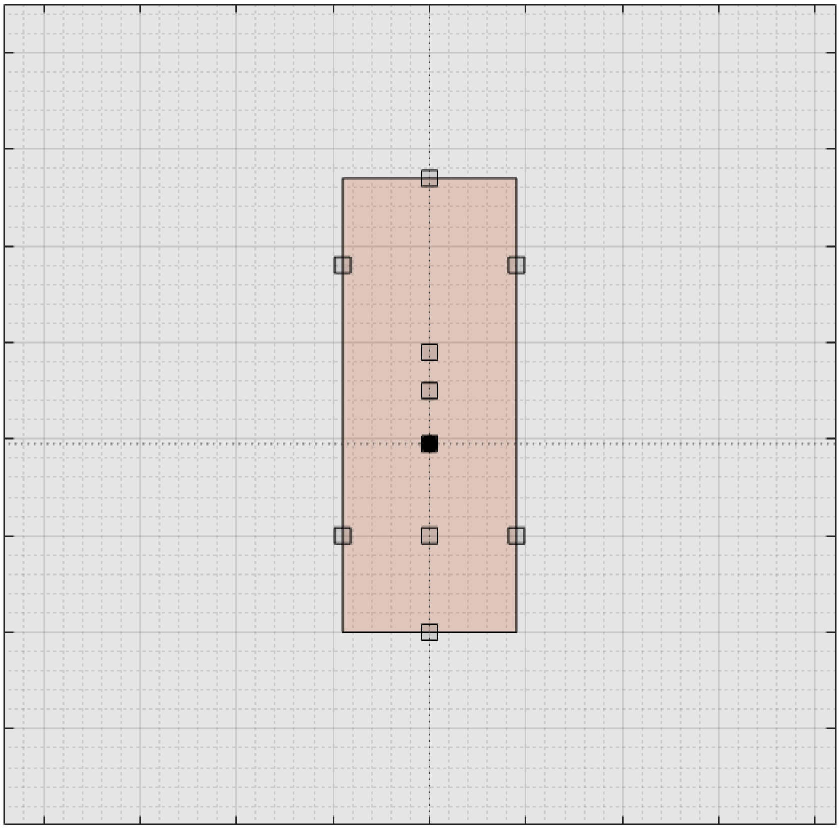
- For more details on prebuilt scenarios available from the app, see “Prebuilt Driving Scenarios in Driving Scenario Designer”.
- For more details on available Euro NCAP scenarios, see “Euro NCAP Driving Scenarios in Driving Scenario Designer”.

Load a Euro NCAP autonomous emergency braking (AEB) scenario of a collision with a pedestrian child. At collision time, the point of impact occurs 50% of the way across the width of the car.

```
path = fullfile(matlabroot,'toolbox','shared','drivingscenario', ...
    'PrebuiltScenarios','EuroNCAP');
addpath(genpath(path)) % Add folder to path
drivingScenarioDesigner('AEB_PedestrianChild_Nearside_50width.mat')
rmpath(path) % Remove folder from path
```

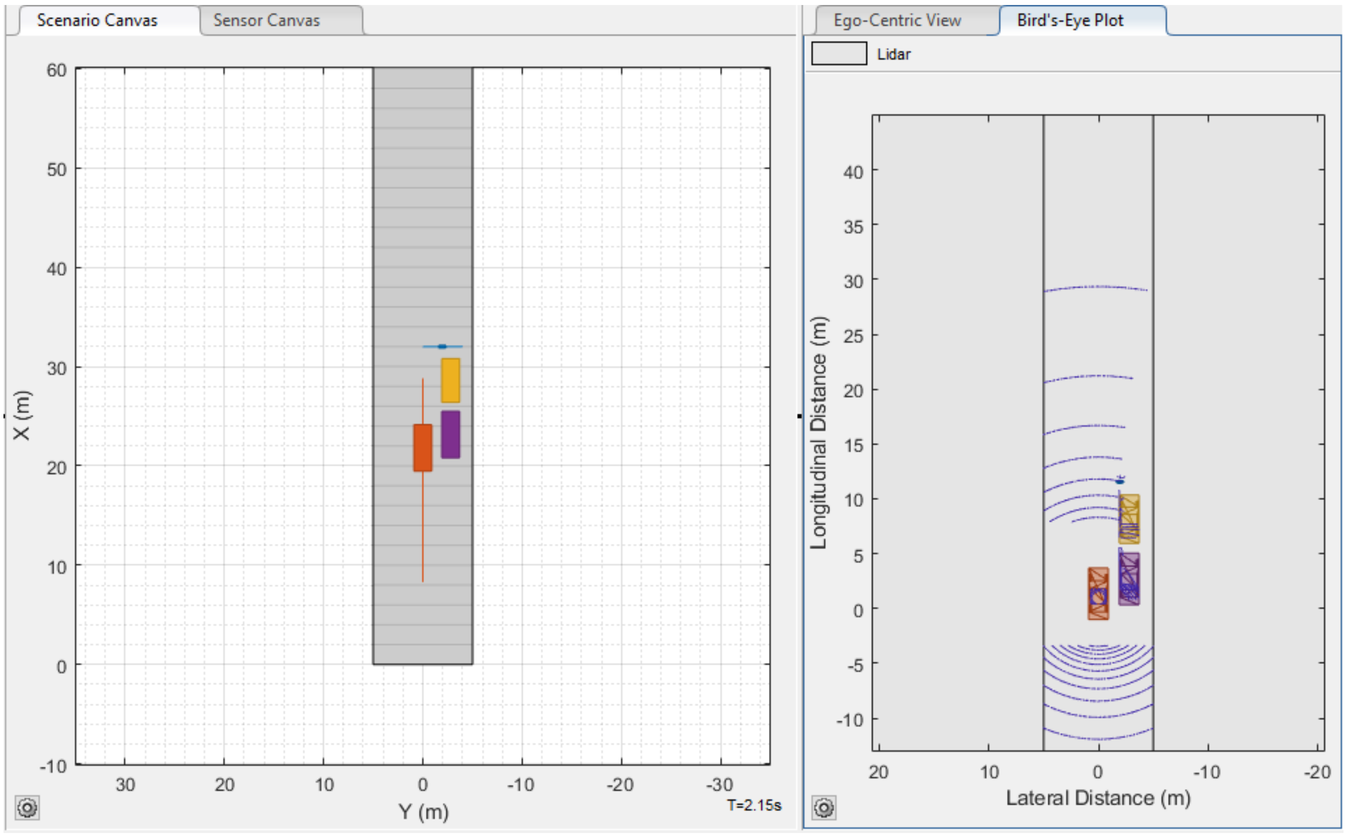


Add a lidar sensor to the ego vehicle. First click **Add Lidar**. Then, on the **Sensor Canvas**, click the predefined sensor location at the roof center of the car. The lidar sensor appears in black at the predefined location. The gray color that surrounds the car is the coverage area of the sensor.



Run the scenario. Inspect different aspects of the scenario by toggling between canvases and views. You can toggle between the **Sensor Canvas** and **Scenario Canvas** and between the **Bird's-Eye Plot** and **Ego-Centric View**.

In the **Bird's-Eye Plot** and **Ego-Centric View**, the actors are displayed as meshes instead of as cuboids. To change the display settings, use the **Display** options on the app toolstrip.



Export the sensor data to the MATLAB workspace. Click **Export** > **Export Sensor Data**, enter a workspace variable name, and click **OK**.

Import Programmatic Driving Scenario and Sensors

Programmatically create a driving scenario, radar sensor, and camera sensor, and then import the scenario and sensors into the app. For more details on working with programmatic driving scenarios and sensors, see "Create Driving Scenario Variations Programmatically".

Create a simple driving scenario by using a `drivingScenario` object. In this scenario, the ego vehicle travels straight on a 50-meter road segment at a constant speed of 30 meters per second. For the ego vehicle, specify a `ClassID` of `1`. This value corresponds to the app **Class ID** of `1`, which refers to actors of class `Car`. For more details on how the app defines classes, see the **Class** parameter description in the "Actors" on page 1-0 parameter tab.

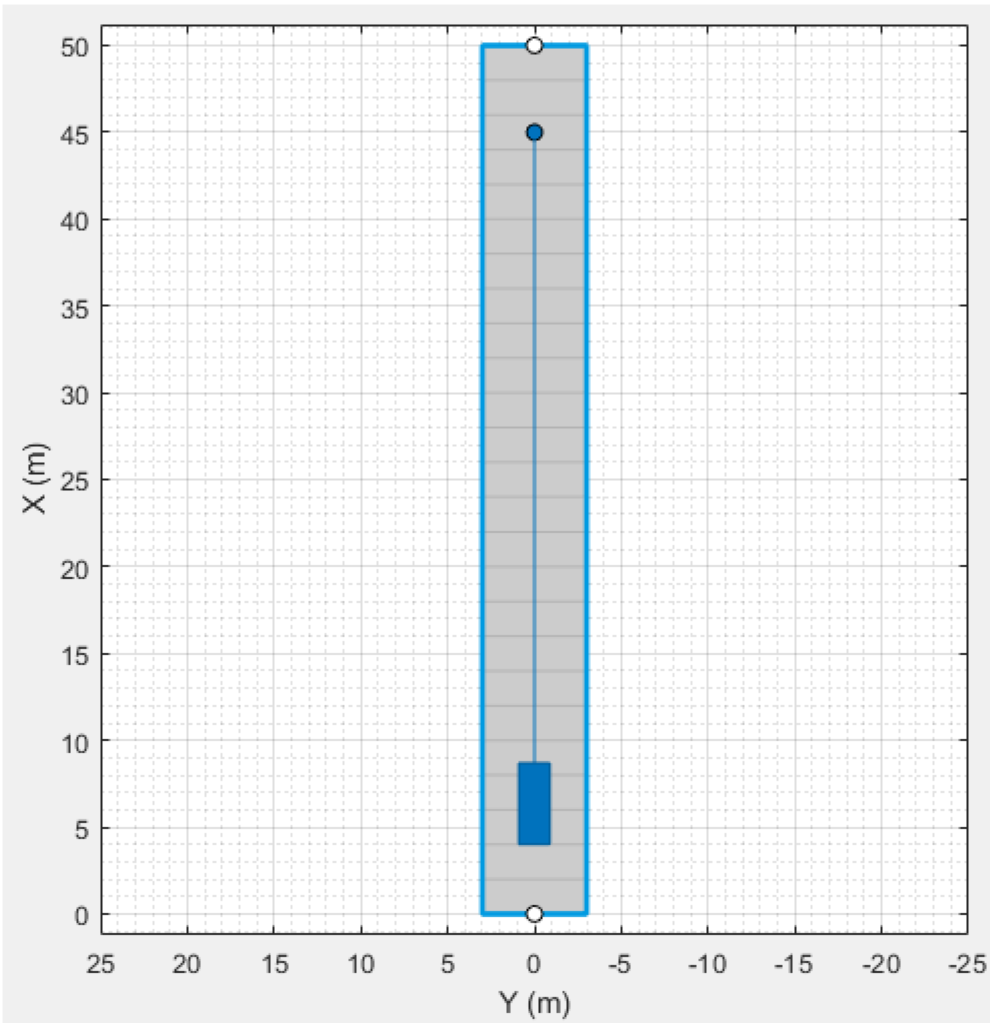
```
scenario = drivingScenario;
roadCenters = [0 0 0; 50 0 0];
road(scenario, roadCenters);

egoVehicle = vehicle(scenario,'ClassID',1,'Position',[5 0 0]);
waypoints = [5 0 0; 45 0 0];
speed = 30;
trajectory(egoVehicle,waypoints,speed);
```

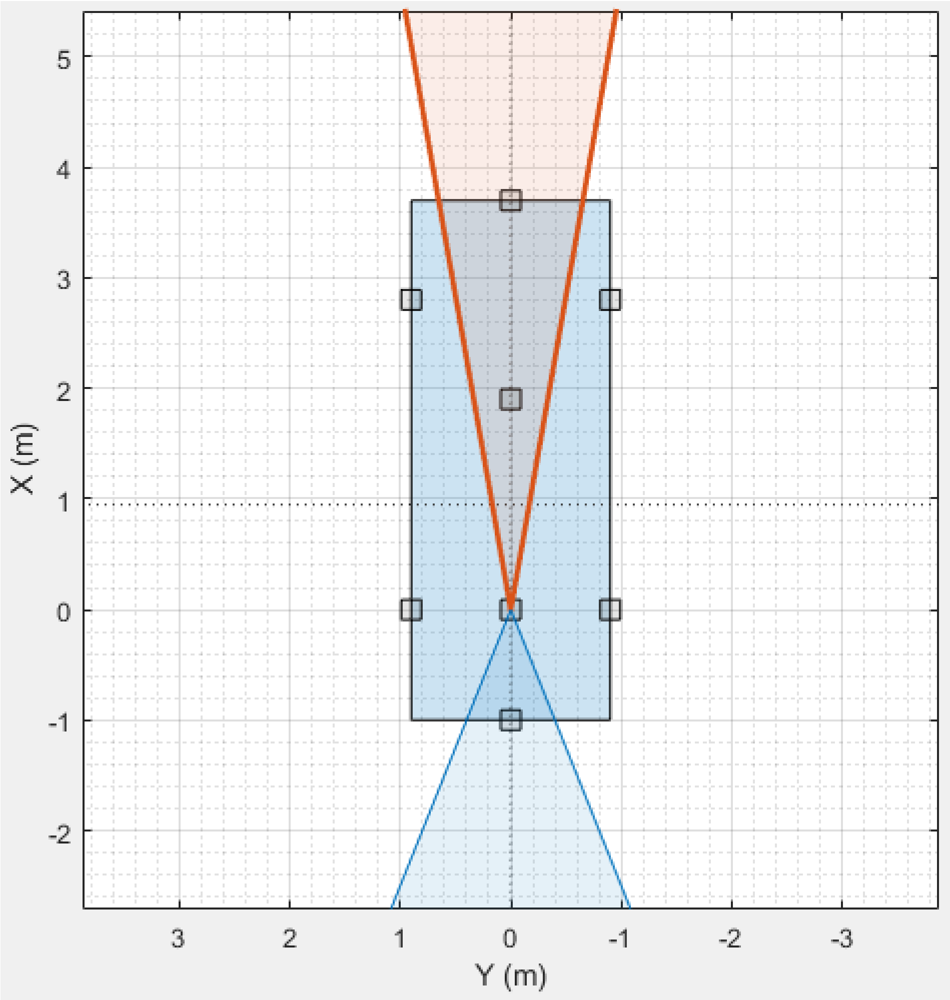
Create a radar sensor by using a `radarDetectionGenerator` object, and create a camera sensor by using a `visionDetectionGenerator` object. Place both sensors at the vehicle origin, with the radar facing forward and the camera facing backward.

```
radar = radarDetectionGenerator('SensorLocation',[0 0]);
camera = visionDetectionGenerator('SensorLocation',[0 0],'Yaw',-180);
```

Import the scenario, front-facing radar sensor, and rear-facing camera sensor into the app.

```
drivingScenarioDesigner(scenario,{radar,camera})
```





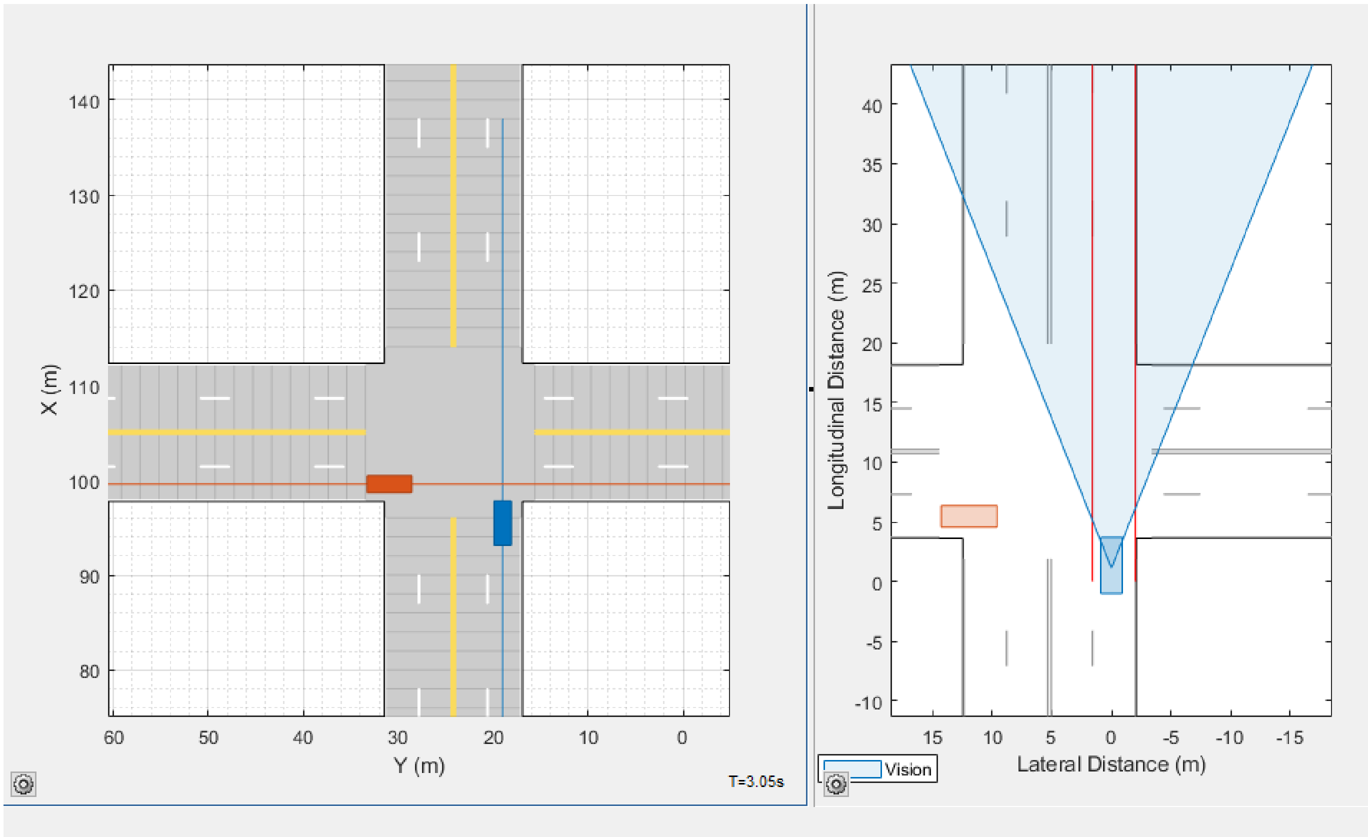
You can then run the scenario and modify the scenario and sensors. To generate new `drivingScenario`, `radarDetectionGenerator`, and `visionDetectionGenerator` objects, on the app toolstrip, select **Export > Export MATLAB Function**, and then run the generated function.

Generate Simulink Model of Scenario and Sensor

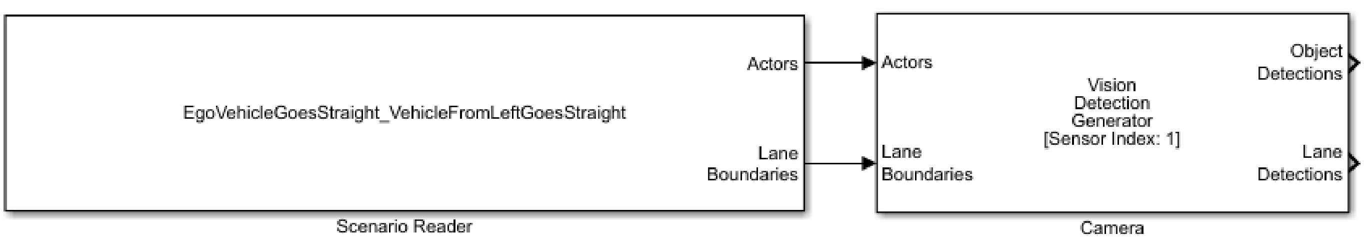
Load a driving scenario containing a sensor and generate a Simulink model from the scenario and sensor. For a more detailed example on generating Simulink models from the app, see “Generate Sensor Detection Blocks Using Driving Scenario Designer”.

Load a prebuilt driving scenario into the app. The scenario contains two vehicles crossing through an intersection. The ego vehicle travels north and contains a camera sensor. This sensor is configured to detect both objects and lanes.

```
path = fullfile(matlabroot,'toolbox','shared','drivingscenario','PrebuiltScenarios');
addpath(genpath(path)) % Add folder to path
drivingScenarioDesigner('EgoVehicleGoesStraight_VehicleFromLeftGoesStraight.mat')
rmpath(path) % Remove folder from path
```



Generate a Simulink model of the scenario and sensor. On the app toolstrip, select **Export** > **Export Simulink Model**. If you are prompted, save the scenario file.



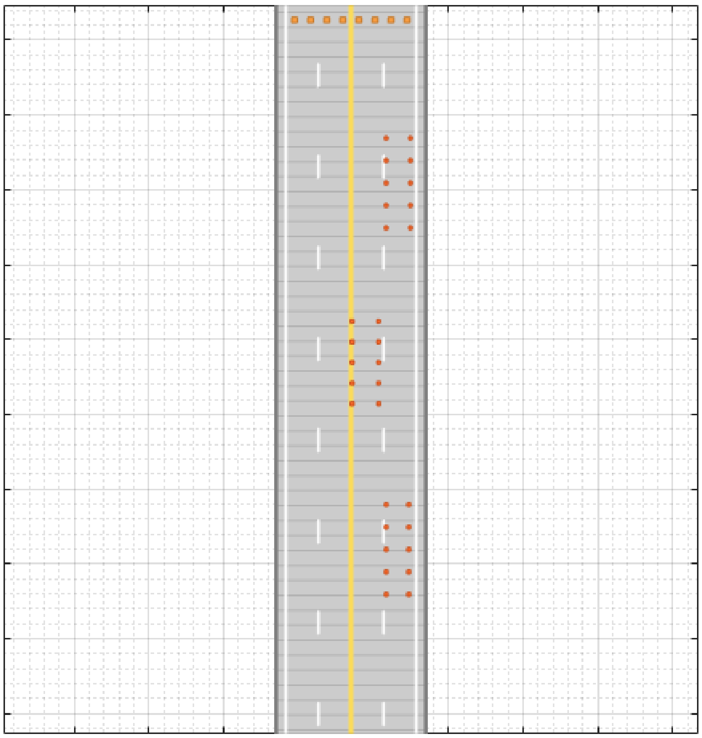
The Scenario Reader block reads the road and actors from the scenario file. To update the scenario data in the model, update the scenario in the app and save the file.

The Vision Detection Generator block recreates the camera sensor defined in the app. To update the sensor in the model, update the sensor in the app, select **Export** > **Export Sensor Simulink Model**, and copy the newly generated sensor block into the model. If you updated any roads or actors while updating the sensors, then select **Export** > **Export Simulink Model**. In this case, the Scenario Reader block accurately reads the actor profile data and passes it to the sensor.

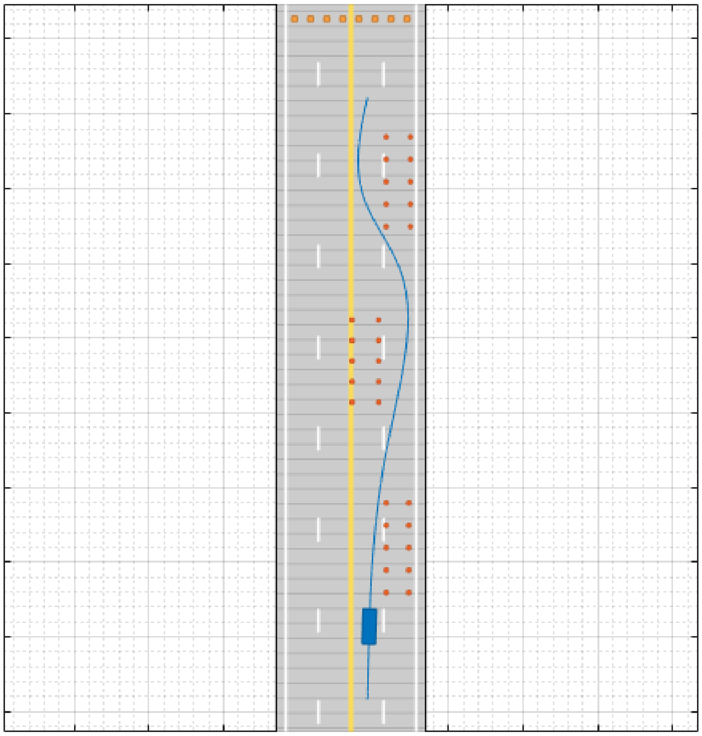
Specify Vehicle Trajectories for 3D Simulation

Create a scenario with vehicle trajectories that you can later recreate in Simulink for simulation in a 3D environment.

Open one of the prebuilt scenarios that recreates a default scene available through the 3D environment. On the app toolstrip, select **Open > Prebuilt Scenario > Simulation3D** and select a scenario. For example, select the `DoubleLaneChange.mat` scenario.



Specify a vehicle and its trajectory.



Update the dimensions of the vehicle to match the dimensions of the predefined vehicle types in the 3D simulation environment.

1 On the **Actors** tab, select the **3D Display Type** option you want.
2 On the app toolstrip, select **3D Display > Use 3D Simulation Actor Dimensions**. In the **Scenario Canvas**, the actor dimensions update to match the predefined dimensions of the actors in the 3D simulation environment.

Preview how the scenario will look when you later recreate it in Simulink. On the app toolstrip, select **3D Display > View Simulation in 3D Display**. After the 3D display window opens, click **Run**.

Modify the vehicle and trajectory as needed. Avoid changing the road network or the actors that were predefined in the scenario. Otherwise, the app scenario will not match the scenario that you later recreate in Simulink. If you change the scenario, the 3D display window closes.

When you are done modifying the scenario, you can recreate it in a Simulink model for use in the 3D simulation environment. For an example that shows how to set up such a model, see “Visualize Sensor Data from Unreal Engine Simulation Environment”.

- “Create Driving Scenario Interactively and Generate Synthetic Sensor Data”
- “Create Reverse Motion Driving Scenarios Interactively”
- “Import OpenDRIVE Roads into Driving Scenario”
- “Import HERE HD Live Map Roads into Driving Scenario”
- “Import OpenStreetMap Data into Driving Scenario”
- “Generate Sensor Detection Blocks Using Driving Scenario Designer”
- “Test Open-Loop ADAS Algorithm Using Driving Scenario”
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario”

Parameters

Roads — Road width, bank angle, lane specifications, and road center locations
tab

To enable the **Roads** parameters, add at least one road to the scenario. Then, select a road from either the **Scenario Canvas** or the **Road** parameter. The parameter values in the **Roads** tab are based on the road you select.

Parameter	Description
Road	Road to modify, specified as a list of the roads in the scenario.
Name	Name of road.
Width (m)	Width of the road, in meters, specified as a decimal scalar in the range (0, 50]. If the curvature of the road is too sharp to accommodate the specified road width, the app does not generate the road. **Default**: 6

Parameter	Description
Bank Angle (deg)	Side-to-side incline of the road, in degrees, specified as one of these values: • Decimal scalar — Applies a uniform bank angle along the entire length of the road • *N*-element vector of decimal values — Applies a different bank angle to each road center, where *N* is the number of road centers in the selected road When you add an actor to a road, you do not have to change the actor position to match the bank angles specified by this parameter. The actor automatically follows the bank angles of the road. **Default**: `0`

Lanes — Lane specifications, such as lane types and lane markings
tab section

Use these parameters to specify lane information, such as lane types and lane markings.

Parameter	Description
Number of lanes	Number of lanes in the road, specified as one of these values: • Integer, *M*, in the range [1, 30] — Creates an *M*-lane road whose default lane markings indicate that the road is one-way. • Two-element vector, [*M N*], where *M* and *N* are positive integers whose sum must be in the range [2, 30] — Creates a road with (*M* + *N*) lanes. The default lane markings of this road indicate that it is two-way. The first *M* lanes travel in one direction. The next *N* lanes travel in the opposite direction. If you increase the number of lanes, the added lanes are of the width specified in the **Lane Width (m)** parameter. If **Lane Width (m)** is a vector of differing lane widths, then the added lanes are of the width specified in the last vector element.

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>Lane Width (m)</td><td>Width of each lane in the road, in meters, specified as one of these values:
Decimal scalar in the range (0, 50] — The same width applies to all lanes.
N-element vector of decimal values in the range (0, 50] — A different width applies to each lane, where N is the total number of lanes specified in the Number of lanes parameter.
The width of each lane must be greater than the width of the lane markings it contains. These lane markings are specified by the Marking > Width (m) parameter.</td></tr>
<tr><td>Lane Types</td><td>Lanes in the road, specified as a list of the lane types in the selected road. To modify one or more lane parameters that include lane type, color, and strength, select the desired lane from the drop-down list.</td></tr>
<tr><td>Lane Types > Type</td><td>Type of lane, specified as one of these values:
<code>'Driving'</code> — Lanes for driving.
<code>'Border'</code> — Lanes at the road borders.
<code>'Restricted'</code> — Lanes reserved for high occupancy vehicles.
<code>'Shoulder'</code> — Lanes reserved for emergency stopping.
<code>'Parking'</code>— Lanes alongside driving lanes, intended for parking vehicles.
Default: <code>'Driving'</code></td></tr>
</table>

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>Lane Types > Color</td><td>Color of lane, specified as an RGB triplet with default values as:
<table><tr><th>Type</th><th>Color (Default values)</th></tr><tr><td>'Driving'</td><td>[0.8 0.8 0.8]</td></tr><tr><td>'Border'</td><td>[0.72 0.72 0.72]</td></tr><tr><td>'Restricted'</td><td>[0.59 0.56 0.62]</td></tr><tr><td>'Shoulder'</td><td>[0.59 00.59 0.59]</td></tr><tr><td>'Parking'</td><td>[0.28 0.28 0.28]</td></tr></table>
Alternatively, you can also specify some common colors as an RGB triplet, hexadecimal color code, color name, or short color name. For more information, see “Color Specifications for Lanes and Markings” on page 1-69.</td></tr>
<tr><td>Lane Types > Strength</td><td>Saturation strength of lane color, specified as a decimal scalar in the range [0, 1].
• A value of 0 specifies that the lane color is fully unsaturated, resulting in a gray colored lane.
• A value of 1 specifies that the lane color is fully saturated, resulting in a true colored lane.
Default: 1</td></tr>
<tr><td>Lane Markings</td><td>Lane markings, specified as a list of the lane markings in the selected road. To modify one or more lane marking parameters which include marking type, color, and strength, select the desired lane marking from the drop-down list.
A road with N lanes has (N + 1) lane markings.</td></tr>
<tr><td>Lane Markings > Specify multiple marker types along a lane</td><td>Select this parameter to define composite lane markings. A composite lane marking comprises multiple marker types along a lane. The portion of the lane marking that contains each marker type is referred as a marker segment. For more information on composite lane markings, see Composite Lane Marking on page 1-72.</td></tr>
</table>

Parameter	Description
Lane Markings > Number of Segments	Number of marker segments in a composite lane marking, specified as an integer greater than or equal to 2. A composite lane marking must have at least two marker segments. **Default**: 2 **Dependencies** To enable this parameter, select the **Specify multiple marker types along a lane** parameter.
Lane Markings > Segment Range	Normalized range for each marker segment in a composite lane marking, specified as a row vector of values in the range [0, 1].. The length of the vector must be equal to the **Number of Segments** parameter value. **Default**: `[0.5 0.5]` **Dependencies** To enable this parameter, select the **Specify multiple marker types along a lane** parameter.
Lane Markings > Marker Segment	Marker segments, specified as a list of marker types in the selected lane marking. To modify one or more marker segment parameters that include marking type, color, and strength, select the desired marker segment from the drop-down list. **Dependencies** To enable this parameter, select the **Specify multiple marker types along a lane** parameter.

Parameter	Description
Lane Markings > Type	Type of lane marking, specified as one of these values: • `Unmarked` — No lane marking • `Solid` — Solid line • `Dashed` — Dashed line • `DoubleSolid` — Two solid lines • `DoubleDashed` — Two dashed lines • `SolidDashed` — Solid line on left, dashed line on right • `DashedSolid` — Dashed line on left, solid line on right By default, for a one-way road, the leftmost lane marking is a solid yellow line, the rightmost lane marking is a solid white line, and the markings for the inner lanes are dashed white lines. For two-way roads, the default outermost lane markings are both solid white lines and the dividing lane marking is two solid yellow lines. If you enable the **Specify multiple marker types along a lane** parameter, then this value is applied to the selected marker segment in a composite lane marking.
Lane Markings > Color	Color of lane marking, specified as an RGB triplet, hexadecimal color code, color name, or short color name. For a lane marker specifying a double line, the same color is used for both lines. You can also specify some common colors as an RGB triplet, hexadecimal color code, color name, or short color name. For more information, see “Color Specifications for Lanes and Markings” on page 1-69. If you enable the **Specify multiple marker types along a lane** parameter, then this value is applied to the selected marker segment in a composite lane marking.

Parameter	Description
Lane Markings > Strength	Saturation strength of lane marking color, specified as a decimal scalar in the range [0, 1]. • A value of `0` specifies that the lane marking color is fully unsaturated, resulting in a gray colored lane marking. • A value of `1` specifies that the lane marking color is fully saturated, resulting in a true colored lane marking. For a lane marker specifying a double line, the same strength is used for both lines. **Default**: `1` If you enable the **Specify multiple marker types along a lane** parameter, then this value is applied to the selected marker segment in a composite lane marking.
Lane Markings > Width (m)	Width of lane marking, in meters, specified as a positive decimal scalar. The width of the lane marking must be less than the width of its enclosing lane. The enclosing lane is the lane directly to the left of the lane marking. For a lane marker specifying a double line, the same width is used for both lines. **Default**: `0.15` If you enable the **Specify multiple marker types along a lane** parameter, then this value is applied to the selected marker segment in a composite lane marking.
Lane Markings > Length (m)	Length of dashes in dashed lane markings, in meters, specified as a decimal scalar in the range (0, 50]. For a lane marker specifying a double line, the same length is used for both lines. **Default**: `3` If you enable the **Specify multiple marker types along a lane** parameter, then this value is applied to the selected marker segment in a composite lane marking.

Parameter	Description
Lane Markings > Space (m)	Length of spaces between dashes in dashed lane markings, in meters, specified as a decimal scalar in the range (0, 150]. For a lane marker specifying a double line, the same space is used for both lines. **Default**: 9 If you enable the **Specify multiple marker types along a lane** parameter, then this value is applied to the selected marker segment in a composite lane marking.

`Road Centers` — Road center locations
tab section

Each row of the **Road Centers** table contains the *x*-, *y*-, and *z*-positions of a road center within the selected road. All roads must have at least two unique road center positions. When you update a cell within the table, the **Scenario Canvas** updates to reflect the new road center position. The orientation of the road depends on the values of the road centers. The road centers specifies the direction in which the road renders in the **Scenario Canvas**. For more information, see Draw Direction of Road and Numbering of Lanes on page 1-69.

Parameter	Description
x (m)	*x*-axis position of the road center, in meters, specified as a decimal scalar.
y (m)	*y*-axis position of the road center, in meters, specified as a decimal scalar.

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>z (m)</td><td> z-axis position of the road center, in meters, specified as a decimal scalar.

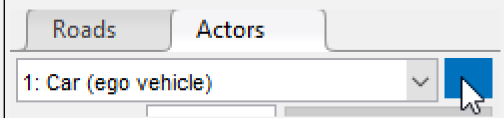
The z-axis specifies the elevation of the road. If the elevation between road centers is too abrupt, adjust these elevation values.
When you add an actor to a road, you do not have to change the actor position to match changes in elevation. The actor automatically follows the elevation of the road.
When two elevated roads form a junction, the elevation around that junction can vary widely. The exact amount of elevation depends on how close the road centers of each road are to each other. If you try to place an actor at the junction, the app might be unable to compute the precise elevation of the actor. In this case, the app cannot place the actor at that junction.

To address this issue, in the Scenario Canvas, modify the intersecting roads by moving the road centers of each road away from each other. Alternatively, manually adjust the elevation of the actor to match the elevation of the road surface.

Default: <code>0</code> </td></tr>
</table>

`Actors` — Actor positions, orientations, RCS patterns, and trajectories
tab

To enable the **Actors** parameters, add at least one actor to the scenario. Then, select an actor from either the **Scenario Canvas** or from the list on the **Actors** tab. The parameter values in the **Actors** tab are based on the actor you select.

Parameter	Description
Color	To change the color of an actor, next to the actor selection list, click the color patch for that actor.  Then, use the color picker to select one of the standard colors commonly used in MATLAB graphics. Alternatively, select a custom color from the **Custom Colors** tab by first clicking  in the upper-right corner of the Color dialog box. You can then select custom colors from a gradient or specify a color using an RGB triplet, hexadecimal color code, or HSV triplet. By default, the app sets each newly created actor to a new color. This color order is based on the default color order of `Axes` objects. For more details, see the `ColorOrder` property for `Axes` objects. To set a single default color for all newly created actors of a specific class, on the app toolstrip, select **Add Actor** > **Edit Actor Classes**. Then, select **Set Default Color** and click the corresponding color patch to set the color. To select a default color for a class, the **Scenario Canvas** must contain no actors of that class. Color changes made in the app are carried forward into **Bird's-Eye Scope** visualizations.
Set as Ego Vehicle	Set the selected actor as the ego vehicle in the scenario. When you add sensors to your scenario, the app adds them to the ego vehicle. In addition, the **Ego-Centric View** and **Bird's-Eye Plot** windows display simulations from the perspective of the ego vehicle. Only actors who have vehicle classes, such as `Car` or `Truck`, can be set as the ego vehicle. The ego vehicle must also have a **3D Display Type** parameter value other than `Cuboid`. For more details on actor classes, see the **Class** parameter description.
Name	Name of actor.

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>Class</td><td>
Class of actor, specified as the list of classes to which you can change the selected actor.
You can change the class of vehicle actors only to other vehicle classes. The default vehicle classes are <code>Car</code> and <code>Truck</code>. Similarly, you can change the class of nonvehicle actors only to other nonvehicle classes. The default nonvehicle classes are <code>Pedestrian</code>, <code>Bicycle</code>, and <code>Barrier</code>.
The list of vehicle and nonvehicle classes appear in the app toolstrip, in the Add Actor > Vehicles and Add Actor > Other sections, respectively.
Actors created in the app have default sets of dimensions, radar cross-section patterns, and other properties based on their Class ID value. The table shows the default Class ID values and actor classes.
<table>
<tr><th>Class ID</th><th>Actor Class</th></tr>
<tr><td>1</td><td><code>Car</code></td></tr>
<tr><td>2</td><td><code>Truck</code></td></tr>
<tr><td>3</td><td><code>Bicycle</code></td></tr>
<tr><td>4</td><td><code>Pedestrian</code></td></tr>
<tr><td>5</td><td><code>Barrier</code></td></tr>
</table>
To modify actor classes or create new actor classes, on the app toolstrip, select Add Actor > Edit Actor Classes or Add Actor > New Actor Class, respectively.
</td></tr>
</table>

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>**3D Display Type**</td><td>Display type of actor as it appears in the 3D display window, specified as the list of display types to which you can change the selected actor.

To display the scenario in the 3D display window during simulation, on the app toolstrip, click **3D Display** > **View Simulation in 3D Display**. The app renders this display by using the Unreal Engine® from Epic Games®.

For any actor, the available **3D Display Type** options depend on the actor class specified in the **Class** parameter.

<table>
<tr><th>Actor Class</th><th>3D Display Type Options</th></tr>
<tr><td>`Car`</td><td rowspan="3">• `Sedan` (default for `Car` class)
• `Muscle Car`
• `SUV`
• `Small Pickup Truck`
• `Hatchback`
• `Box Truck` (default for `Truck` class)
• `Cuboid` (default for custom vehicle classes)</td></tr>
<tr><td>`Truck`</td></tr>
<tr><td>Custom vehicle class

To create a custom vehicle class:

1. On the app toolstrip, select **Add Actor** > **New Actor Class**.
2. In the Class Editor window, select the **Vehicle** parameter.
3. Set other class properties as needed and click **OK**.</td></tr>
<tr><td>`Bicycle`</td><td rowspan="4">• `Bicyclist` (default for `Bicycle` class)
• `Male Pedestrian` (default for `Pedestrian` class)
• `Female Pedestrian`
• `Barrier` (default for `Barrier` class)
• `Cuboid` (default for custom nonvehicle classes)</td></tr>
<tr><td>`Pedestrian`</td></tr>
<tr><td>`Barrier`</td></tr>
<tr><td>Custom nonvehicle class

To create a custom nonvehicle class:

1. On the app toolstrip, select **Add Actor** > **New Actor Class**.
2. In the Class Editor window, clear the **Vehicle** parameter.
3. Set other class properties as needed and click **OK**.</td></tr>
</table>

If you change the dimensions of an actor using the **Actor Properties** parameters, the app applies these changes in the **Scenario Canvas** display but not in the 3D display. This case</td></tr>
</table>

Parameter	Description
	does not apply to actors whose **3D Display Type** is set to `Barrier` or `Cuboid`. The dimensions of these actors change in both displays. In the 3D display, actors of all other display types have predefined dimensions. To use the same dimensions in both displays, you can apply the predefined 3D display dimensions to the actors in the **Scenario Canvas** display. On the app toolstrip, under **3D Display**, select **Use 3D Simulation Actor Dimensions**.

`Actor Properties` — Actor properties, including position and orientation
tab section

Use these parameters to specify properties such as the position and orientation of an actor.

Parameter	Description
Length (m)	Length of actor, in meters, specified as a decimal scalar in the range (0, 60]. For vehicles, the length must be greater than (**Front Overhang** + **Rear Overhang**).
Width (m)	Width of actor, in meters, specified as a decimal scalar in the range (0, 20].
Height (m)	Height of actor, in meters, specified as a decimal scalar in the range (0, 20].
Front Overhang	Distance between the front axle and front bumper, in meters, specified as a decimal scalar. The front overhang must be less than (**Length (m)** – **Rear Overhang**). This parameter applies to vehicles only. **Default**: `0.9`
Rear Overhang	Distance between the rear axle and rear bumper, in meters, specified as a decimal scalar. The rear overhang must be less than (**Length (m)** – **Front Overhang**). This parameter applies to vehicles only. **Default**: `1`

Parameter	Description
Roll (°)	Orientation angle of the actor about its *x*-axis, in degrees, specified as a decimal scalar. **Roll (°)** is clockwise-positive when looking in the forward direction of the *x*-axis, which points forward from the actor. When you export the MATLAB function of the driving scenario and run that function, the roll angles of actors in the output scenario are wrapped to the range [–180, 180]. **Default**: `0`
Pitch (°)	Orientation angle of the actor about its *y*-axis, in degrees, specified as a decimal scalar. **Pitch (°)** is clockwise-positive when looking in the forward direction of the *y*-axis, which points to the left of the actor. When you export the MATLAB function of the driving scenario and run that function, the pitch angles of actors in the output scenario are wrapped to the range [–180, 180]. **Default**: `0`
Yaw (°)	Orientation angle of the actor about its *z*-axis, in degrees, specified as a decimal scalar. **Yaw (°)** is clockwise-positive when looking in the forward direction of the *z*-axis, which points up from the ground. However, the **Scenario Canvas** has a bird's-eye-view perspective that looks in the reverse direction of the *z*-axis. Therefore, when viewing actors on this canvas, **Yaw (°)** is counterclockwise-positive. When you export the MATLAB function of the driving scenario and run that function, the yaw angles of actors in the output scenario are wrapped to the range [–180, 180]. **Default**: `0`

`Radar Cross Section` — RCS of actor
tab section

Use these parameters to manually specify the radar cross-section (RCS) of an actor. Alternatively, to import an RCS from a file or from the MATLAB workspace, expand this parameter section and click **Import**.

Parameter	Description
Azimuth Angles (deg)	Horizontal reflection pattern of actor, in degrees, specified as a vector of monotonically increasing decimal values in the range [–180, 180]. **Default**: `[-180 180]`
Elevation Angles (deg)	Vertical reflection pattern of actor, in degrees, specified as a vector of monotonically increasing decimal values in the range [–90, 90]. **Default**: `[-90 90]`
Pattern (dBsm)	RCS pattern, in decibels per square meter, specified as a *Q*-by-*P* table of decimal values. RCS is a function of the azimuth and elevation angles, where: • *Q* is the number of elevation angles specified by the **Elevation Angles (deg)** parameter. • *P* is the number of azimuth angles specified by the **Azimuth Angles (deg)** parameter.

`Trajectory` — Actor trajectories
tab section

Spawn and Despawn an Actor During Simulation

Parameter	Description
Actor spawn and despawn	Select this parameter to spawn or despawn an actor in the driving scenario, while the simulation is running. To enable this parameter, you must first select an actor in the scenario by clicking on the actor. Specify values for the **Entry Time(s)** and **Exit Time(s)** parameters to make the actor enter (spawn) and exit (despawn) the scenario, respectively.
Entry Time(s)	Specify the time at which an actor spawns into the scenario during simulation. The entry time must always be less than the exit time. Units are in seconds. The default value for entry time is `0`.
Exit Time(s)	Specify the time at which an actor despawns from the scenario during simulation. The exit time must always be greater than the entry time. Units are in seconds. The default value for exit time is `Inf`.

The values for the **Entry Time(s)** and **Exit Time(s)** parameters must be less than the entire simulation time that is set by either the stop condition or the stop time. If you specify values for **Entry Time(s)** and **Exit Time(s)** that are greater than the simulation duration, as determined by either the set stop time or stop condition, then the actor will either not spawn or not despawn, respectively

Use the **Waypoints, Speeds, Wait Times, and Yaw** table to manually set or modify the positions, speeds, wait times, and yaw orientation angles of actors at their specified waypoints. When specifying trajectories, to switch between adding forward and reverse motion waypoints, use the add forward and reverse motion waypoint buttons  .

Parameter	Description
Constant Speed (m/s)	Default speed of actors as you add waypoints, specified as a positive decimal scalar in meters per second. If you set specific speed values in the **v (m/s)** column of the **Waypoints, Speeds, Wait Times, and Yaw** table, then the app clears the **Constant Speed (m/s)** value. If you then specify a new **Constant Speed (m/s)** value, then the app sets all waypoints to the new constant speed value. The default speed of an actor varies by actor class. For example, cars and trucks have a default constant speed of 30 meters per second, whereas pedestrians have a default constant speed of 1.5 meters per second.

Parameter	Description
Waypoints, Speeds, Wait Times, and Yaw	Actor waypoints, specified as a table. Each row corresponds to a waypoint and contains the position, speed, and orientation of the actor at that waypoint. The table has these columns. • **x (m)** — World coordinate *x*-position of each waypoint in meters. • **y (m)** — World coordinate *y*-position of each waypoint in meters. • **z (m)** — World coordinate *z*-position of each waypoint in meters. • **v (m/s)** — Actor speed, in meters per second, at each waypoint. By default, the app sets the **v (m/s)** of newly added waypoints to the **Constant Speed (m/s)** parameter value. To specify a reverse motion between trajectories, set **v (m/s)** to a negative value. Positive speeds (forward motions) and negative speeds (reverse motions) must be separated by a waypoint with a speed of `0`. • **wait (s)** — Wait time for an actor, in seconds, at each waypoint. When you set the wait time to a positive value, the corresponding velocity value **v (m/s)** resets to `0`. You cannot set wait times at consecutive waypoints along the trajectory of an actor to positive values. • **yaw (°)** — Yaw orientation angle of an actor, in degrees, at each waypoint. Yaw angles are counterclockwise-positive when looking at the scenario from the top down. By default, the app computes the yaw automatically based on the specified trajectory. To constrain the trajectory such that the vehicle has specific orientations at certain waypoints, set the desired **yaw (°)** values at those waypoints. To restore a yaw back to its default value, right-click the waypoint and select **Restore Default Yaw**.

`Sensors (Camera)` — Camera sensor placement, intrinsic camera parameters, and detection parameters
tab

To access these parameters, add at least one camera sensor to the scenario by following these steps:

1. On the app toolstrip, click **Add Camera**.
2. From the **Sensors** tab, select the sensor from the list. The parameter values in this tab are based on the sensor you select.

Parameter	Description
Enabled	Enable or disable the selected sensor. Select this parameter to capture sensor data during simulation and visualize that data in the **Bird's-Eye Plot** pane.
Name	Name of sensor.
Update Interval (ms)	Frequency at which the sensor updates, in milliseconds, specified as an integer multiple of the app sample time defined under **Settings**, in the **Sample Time (ms)** parameter. The default **Update Interval (ms)** value of `100` is an integer multiple of the default **Sample Time (ms)** parameter value of `10`. When the update interval is a multiple of the sample time, it ensures that the app samples and displays the detections found at these intervals during simulation. If you update the app sample time such that a sensor is no longer a multiple of the app sample time, the app prompts you with the option to automatically update the **Update Interval (ms)** parameter to the closest integer multiple. **Default**: `100`
Type	Type of sensor, specified as **Radar** for radar sensors, **Vision** for camera sensors, or **Lidar** for lidar sensors.

`Sensor Placement` — Camera position and orientation
tab section

Use these parameters to set the position and orientation of the selected camera sensor.

Parameter	Description
X (m)	*X*-axis position of the sensor in the vehicle coordinate system, in meters, specified as a decimal scalar. The *X*-axis points forward from the vehicle. The origin is located at the center of the vehicle's rear axle.
Y (m)	*Y*-axis position of the sensor in the vehicle coordinate system, in meters, specified as a decimal scalar. The *Y*-axis points to the left of the vehicle. The origin is located at the center of the vehicle's rear axle.

Parameter	Description
Height (m)	Height of the sensor above the ground, in meters, specified as a positive decimal scalar. **Default**: `1.1`
Roll (°)	Orientation angle of the sensor about its *X*-axis, in degrees, specified as a decimal scalar. **Roll (°)** is clockwise-positive when looking in the forward direction of the *X*-axis, which points forward from the sensor. **Default**: `0`
Pitch (°)	Orientation angle of the sensor about its *Y*-axis, in degrees, specified as a decimal scalar. **Pitch (°)** is clockwise-positive when looking in the forward direction of the *Y*-axis, which points to the left of the sensor. **Default**: `1`
Yaw (°)	Orientation angle of the sensor about its *Z*-axis, in degrees, specified as a decimal scalar. **Yaw (°)** is clockwise-positive when looking in the forward direction of the *Z*-axis, which points up from the ground. The **Sensor Canvas** has a bird's-eye-view perspective that looks in the reverse direction of the *Z*-axis. Therefore, when viewing sensor coverage areas on this canvas, **Yaw (°)** is counterclockwise-positive.

`Camera Settings` — Intrinsic camera parameters
tab section

Use these parameters to set the intrinsic parameters of the camera sensor.

Parameter	Description
Focal Length X	Horizontal point at which the camera is in focus, in pixels, specified as a positive decimal scalar. The default focal length changes depending on where you place the sensor on the ego vehicle.
Focal Length Y	Vertical point at which the camera is in focus, in pixels, specified as a positive decimal scalar. The default focal length changes depending on where you place the sensor on the ego vehicle.

Parameter	Description
Image Width	Horizontal camera resolution, in pixels, specified as a positive integer. **Default**: `640`
Image Height	Vertical camera resolution, in pixels, specified as a positive integer. **Default**: `480`
Principal Point X	Horizontal image center, in pixels, specified as a positive decimal scalar. **Default**: `320`
Principal Point Y	Vertical image center, in pixels, specified as a positive decimal scalar. **Default**: `240`

`Detection Parameters` — Camera detection parameters
tab section

To view all camera detection parameters in the app, expand the **Sensor Limits**, **Lane Settings**, and **Accuracy & Noise Settings** sections.

Parameter	Description
Detection Type	Type of detections reported by camera, specified as one of these values: • `Objects` — Report object detections only. • `Objects & Lanes` — Report object and lane boundary detections. • `Lanes` — Report lane boundary detections only. **Default**: `Objects`
Detection Probability	Probability that the camera detects an object, specified as a decimal scalar in the range (0, 1]. **Default**: `0.9`
False Positives Per Image	Number of false positives reported per update interval, specified as a nonnegative decimal scalar. This value must be less than or equal to the maximum number of detections specified in the **Limit # of Detections** parameter. **Default**: `0.1`

Parameter	Description
Limit # of Detections	Select this parameter to limit the number of simultaneous object detections that the sensor reports. Specify **Limit # of Detections** as a positive integer less than 2^{63}. To enable this parameter, set the **Detection Type** parameter to `Objects` or `Objects & Lanes`. **Default**: `off`
Detection Coordinates	Coordinate system of output detection locations, specified as one of these values: • `Ego Cartesian` — The app outputs detections in the coordinate system of the ego vehicle. • `Sensor Cartesian` — The app outputs detections in the coordinate system of the sensor. **Default**: `Ego Cartesian`

Sensor Limits

Parameter	Description
Max Speed (m/s)	Fastest relative speed at which the camera can detect objects, in meters per second, specified as a nonnegative decimal scalar. **Default**: `100`
Max Range (m)	Farthest distance at which the camera can detect objects, in meters, specified as a positive decimal scalar. **Default**: `150`
Max Allowed Occlusion	Maximum percentage of object that can be blocked while still being detected, specified as a decimal scalar in the range [0, 1). **Default**: `0.5`
Min Object Image Width	Minimum horizontal size of objects that the camera can detect, in pixels, specified as positive decimal scalar. **Default**: `15`
Min Object Image Height	Minimum vertical size of objects that the camera can detect, in pixels, specified as positive decimal scalar. **Default**: `15`

Lane Settings

Parameter	Description
Lane Update Interval (ms)	Frequency at which the sensor updates lane detections, in milliseconds, specified as a decimal scalar. **Default**: `100`
Min Lane Image Width	Minimum horizontal size of objects that the sensor can detect, in pixels, specified as a decimal scalar. To enable this parameter, set the **Detection Type** parameter to `Lanes` or `Objects & Lanes`. **Default**: `3`
Min Lane Image Height	Minimum vertical size of objects that the sensor can detect, in pixels, specified as a decimal scalar. To enable this parameter, set the **Detection Type** parameter to `Lanes` or `Objects & Lanes`. **Default**: `20`
Boundary Accuracy	Accuracy with which the sensor places a lane boundary, in pixels, specified as a decimal scalar. To enable this parameter, set the **Detection Type** parameter to `Lanes` or `Objects & Lanes`. **Default**: `3`
Limit # of Lanes	Select this parameter to limit the number of lane detections that the sensor reports. Specify **Limit # of Lanes** as a positive integer. To enable this parameter, set the **Detection Type** parameter to `Lanes` or `Objects & Lanes`. **Default**: `off`

Accuracy & Noise Settings

Parameter	Description
Bounding Box Accuracy	Positional noise used for fitting bounding boxes to targets, in pixels, specified as a positive decimal scalar. **Default**: 5
Process Noise Intensity (m/s^2)	Noise intensity used for smoothing position and velocity measurements, in meters per second squared, specified as a positive decimal scalar. **Default**: 5
Has Noise	Select this parameter to enable adding noise to sensor measurements. **Default**: `off`

`Sensors (Radar)` — Radar sensor placement and detection parameters

tab

To access these parameters, add at least one radar sensor to the scenario.

1. On the app toolstrip, click **Add Radar**.
2. On the **Sensors** tab, select the sensor from the list. The parameter values changes based on the sensor you select.

Parameter	Description
Enabled	Enable or disable the selected sensor. Select this parameter to capture sensor data during simulation and visualize that data in the **Bird's-Eye Plot** pane.
Name	Name of sensor.

Parameter	Description
Update Interval (ms)	Frequency at which the sensor updates, in milliseconds, specified as an integer multiple of the app sample time defined under **Settings**, in the **Sample Time (ms)** parameter. The default **Update Interval (ms)** value of `100` is an integer multiple of the default **Sample Time (ms)** parameter value of `10`. When the update interval is a multiple of the sample time, it ensures that the app samples and displays the detections found at these intervals during simulation. If you update the app sample time such that a sensor is no longer a multiple of the app sample time, the app prompts you with the option to automatically update the **Update Interval (ms)** parameter to the closest integer multiple. **Default**: `100`
Type	Type of sensor, specified as **Radar** for radar sensors, **Vision** for camera sensors, or **Lidar** for lidar sensors.

`Sensor Placement` — Radar position and orientation
tab section

Use these parameters to set the position and orientation of the selected radar sensor.

Parameter	Description
X (m)	*X*-axis position of the sensor in the vehicle coordinate system, in meters, specified as a decimal scalar. The *X*-axis points forward from the vehicle. The origin is located at the center of the vehicle's rear axle.
Y (m)	*Y*-axis position of the sensor in the vehicle coordinate system, in meters, specified as a decimal scalar. The *Y*-axis points to the left of the vehicle. The origin is located at the center of the vehicle's rear axle.
Height (m)	Height of the sensor above the ground, in meters, specified as a positive decimal scalar. **Default**: `1.1`

Parameter	Description
Roll (°)	Orientation angle of the sensor about its *X*-axis, in degrees, specified as a decimal scalar. **Roll (°)** is clockwise-positive when looking in the forward direction of the *X*-axis, which points forward from the sensor. **Default**: `0`
Pitch (°)	Orientation angle of the sensor about its *Y*-axis, in degrees, specified as a decimal scalar. **Pitch (°)** is clockwise-positive when looking in the forward direction of the *Y*-axis, which points to the left of the sensor. **Default**: `1`
Yaw (°)	Orientation angle of the sensor about its *Z*-axis, in degrees, specified as a decimal scalar. **Yaw (°)** is clockwise-positive when looking in the forward direction of the *Z*-axis, which points up from the ground. The **Sensor Canvas** has a bird's-eye-view perspective that looks in the reverse direction of the *Z*-axis. Therefore, when viewing sensor coverage areas on this canvas, **Yaw (°)** is counterclockwise-positive.

`Detection Parameters` — Radar detection parameters
tab section

To view all radar detection parameters in the app, expand the **Advanced Parameters** and **Accuracy & Noise Settings** sections.

Parameter	Description
Detection Probability	Probability that the radar detects an object, specified as a decimal scalar in the range (0, 1]. **Default**: `0.9`
False Alarm Rate	Probability of a false detection per resolution rate, specified as a decimal scalar in the range [`1e-07`, `1e-03`]. **Default**: `1e-06`
Field of View Azimuth	Horizontal field of view of radar, in degrees, specified as a positive decimal scalar. **Default**: `20`

Parameter	Description
Field of View Elevation	Vertical field of view of radar, in degrees, specified as a positive decimal scalar. **Default**: `5`
Max Range (m)	Farthest distance at which the radar can detect objects, in meters, specified as a positive decimal scalar. **Default**: `150`
Range Rate Min, Range Rate Max	Select this parameter to set minimum and maximum range rate limits for the radar. Specify **Range Rate Min** and **Range Rate Max** as decimal scalars, in meters per second, where **Range Rate Min** is less than **Range Rate Max**. **Default (Min)**: `-100` **Default (Max)**: `100`
Has Elevation	Select this parameter to enable the radar to measure the elevation of objects. This parameter enables the elevation parameters in the **Accuracy & Noise Settings** section. **Default**: `off`
Has Occlusion	Select this parameter to enable the radar to model occlusion. **Default**: `on`

Advanced Parameters

Parameter	Description
Reference Range	Reference range for a given probability of detection, in meters, specified as a positive decimal scalar. The reference range is the range at which the radar detects a target of the size specified by **Reference RCS**, given the probability of detection specified by **Detection Probability**. **Default**: `100`
Reference RCS	Reference RCS for a given probability of detection, in decibels per square meter, specified as a nonnegative decimal scalar. The reference RCS is the target size at which the radar detects a target, given the reference range specified by **Reference Range** and the probability of detection specified by **Detection Probability**. **Default**: `0`
Limit # of Detections	Select this parameter to limit the number of simultaneous detections that the sensor reports. Specify **Limit # of Detections** as a positive integer less than 2^{63}. **Default**: `off`
Detection Coordinates	Coordinate system of output detection locations, specified as one of these values: • `Ego Cartesian` — The app outputs detections in the coordinate system of the ego vehicle. • `Sensor Cartesian` — The app outputs detections in the coordinate system of the sensor. • `Sensor spherical` — The app outputs detections in a spherical coordinate system. This coordinate system is centered at the radar and aligned with the orientation of the radar on the ego vehicle. **Default**: `Ego Cartesian`

Accuracy & Noise Settings

Parameter	Description
Azimuth Resolution	Minimum separation in azimuth angle at which the radar can distinguish between two targets, in degrees, specified as a positive decimal scalar. The azimuth resolution is typically the 3 dB downpoint in the azimuth angle beamwidth of the radar. **Default**: `4`
Azimuth Bias Fraction	Maximum azimuth accuracy of the radar, specified as a nonnegative decimal scalar. The azimuth bias is expressed as a fraction of the azimuth resolution specified by the **Azimuth Resolution** parameter. Units are dimensionless. **Default**: `0.1`
Elevation Resolution	Minimum separation in elevation angle at which the radar can distinguish between two targets, in degrees, specified as a positive decimal scalar. The elevation resolution is typically the 3 dB downpoint in the elevation angle beamwidth of the radar. To enable this parameter, in the **Detection Parameters** section, select the **Has Elevation** parameter. **Default**: `10`
Elevation Bias Fraction	Maximum elevation accuracy of the radar, specified as a nonnegative decimal scalar. The elevation bias is expressed as a fraction of the elevation resolution specified by the **Elevation Resolution** parameter. Units are dimensionless. To enable this parameter, under **Detection Parameters**, select the **Has Elevation** parameter. **Default**: `0.1`
Range Resolution	Minimum range separation at which the radar can distinguish between two targets, in meters, specified as a positive decimal scalar. **Default**: `2.5`

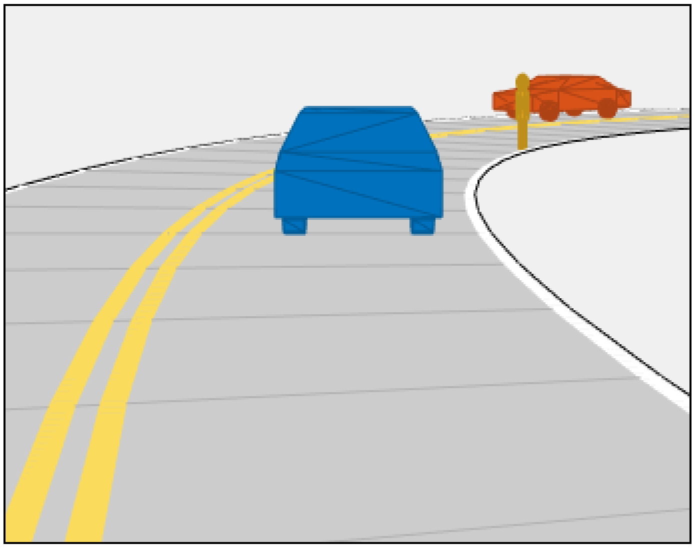
Parameter	Description
Range Bias Fraction	Maximum range accuracy of the radar, specified as a nonnegative decimal scalar. The range bias is expressed as a fraction of the range resolution specified in the **Range Resolution** parameter. Units are dimensionless. **Default**: `0.05`
Range Rate Resolution	Minimum range rate separation at which the radar can distinguish between two targets, in meters per second, specified as a positive decimal scalar. To enable this parameter, in the **Detection Parameters** section, select the **Range Rate Min, Range Rate Max** parameter and set the range rate values. **Default**: `0.5`
Range Rate Bias Fraction	Maximum range rate accuracy of the radar, specified as a nonnegative decimal scalar. The range rate bias is expressed as a fraction of the range rate resolution specified in the **Range Rate Resolution** parameter. Units are dimensionless. To enable this parameter, under the **Detection Parameters** section, select the **Range Rate Min, Range Rate Max** parameter and set the range rate values. **Default**: `0.05`
Has Noise	Select this parameter to enable adding noise to sensor measurements. **Default**: `off`
Has False Alarms	Select this parameter to enable false alarms in sensor detections. **Default**: `off`

`Sensors (Lidar)` — Lidar sensor placement, point cloud reporting, and detection parameters
tab

To access these parameters, add at least one lidar sensor to the scenario.

1. On the app toolstrip, click **Add Lidar**.
2. On the **Sensors** tab, select the sensor from the list. The parameter values change based on the sensor you select.

When you add a lidar sensor to a scenario, the **Bird's-Eye Plot** and **Ego-Centric View** display the mesh representations of actors. For example, here is a sample view of actor meshes on the **Ego-Centric View**.



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 other sensors still base their detections on the cuboid representations.

To turn off actor meshes, use the properties under **Display** on the app toolstrip. To modify the mesh display types of actors, select **Add Actor > Edit Actor Classes**. In the Class Editor, modify the **Mesh Display Type** parameter of that actor class.

Parameter	Description
Enabled	Enable or disable the selected sensor. Select this parameter to capture sensor data during simulation and visualize that data in the **Bird's-Eye Plot** pane.
Name	Name of sensor.

Parameter	Description
Update Interval (ms)	Frequency at which the sensor updates, in milliseconds, specified as an integer multiple of the app sample time defined under **Settings**, in the **Sample Time (ms)** parameter. The default **Update Interval (ms)** value of `100` is an integer multiple of the default **Sample Time (ms)** parameter value of `10`. When the update interval is a multiple of the sample time, it ensures that the app samples and displays the detections found at these intervals during simulation. If you update the app sample time such that a sensor is no longer a multiple of the app sample time, the app prompts you with the option to automatically update the **Update Interval (ms)** parameter to the closest integer multiple. **Default**: `100`
Type	Type of sensor, specified as **Radar** for radar sensors, **Vision** for camera sensors, or **Lidar** for lidar sensors.

`Sensor Placement` — Lidar position and orientation

tab section

Use these parameters to set the position and orientation of the selected lidar sensor.

Parameter	Description
X (m)	*X*-axis position of the sensor in the vehicle coordinate system, in meters, specified as a decimal scalar. The *X*-axis points forward from the vehicle. The origin is located at the center of the vehicle's rear axle.
Y (m)	*Y*-axis position of the sensor in the vehicle coordinate system, in meters, specified as a decimal scalar. The *Y*-axis points to the left of the vehicle. The origin is located at the center of the vehicle's rear axle.
Height (m)	Height of the sensor above the ground, in meters, specified as a positive decimal scalar. **Default**: `1.1`

Parameter	Description
Roll (°)	Orientation angle of the sensor about its *X*-axis, in degrees, specified as a decimal scalar. **Roll (°)** is clockwise-positive when looking in the forward direction of the *X*-axis, which points forward from the sensor. **Default**: `0`
Pitch (°)	Orientation angle of the sensor about its *Y*-axis, in degrees, specified as a decimal scalar. **Pitch (°)** is clockwise-positive when looking in the forward direction of the *Y*-axis, which points to the left of the sensor. **Default**: `1`
Yaw (°)	Orientation angle of the sensor about its *Z*-axis, in degrees, specified as a decimal scalar. **Yaw (°)** is clockwise-positive when looking in the forward direction of the *Z*-axis, which points up from the ground. The **Sensor Canvas** has a bird's-eye-view perspective that looks in the reverse direction of the *Z*-axis. Therefore, when viewing sensor coverage areas on this canvas, **Yaw (°)** is counterclockwise-positive.

`Point Cloud Reporting` — Point cloud reporting parameters
tab section

Parameter	Description
Detection Coordinates	Coordinate system of output detection locations, specified as one of these values: • `Ego Cartesian` — The app outputs detections in the coordinate system of the ego vehicle. • `Sensor Cartesian` — The app outputs detections in the coordinate system of the sensor. **Default**: `Ego Cartesian`
Output organized point cloud locations	Select this parameter to output the generated sensor data as an organized point cloud. If you clear this parameter, the output is unorganized. **Default**: `on`

Parameter	Description
Include ego vehicle in generated point cloud	Select this parameter to include the ego vehicle in the generated point cloud. **Default**: `on`
Include roads in generated point cloud	Select this parameter to include roads in the generated point cloud. **Default**: `off`

`Detection Parameters` — Lidar detection parameters
tab section

Sensor Limits

Parameter	Description
Max Range (m)	Farthest distance at which the lidar can detect objects, in meters, specified as a positive decimal scalar. **Default**: `50`
Range Accuracy (m)	Accuracy of range measurements, in meters, specified as a positive decimal scalar. **Default**: `0.002`
Azimuth	Azimuthal resolution of the lidar sensor, in degrees, specified as a positive decimal scalar. The azimuthal resolution defines the minimum separation in azimuth angle at which the lidar can distinguish two targets. **Default**: `1.6`
Elevation	Elevation resolution of the lidar sensor, in degrees, specified as a positive decimal scalar. The elevation resolution defines the minimum separation in elevation angle at which the lidar can distinguish two targets. **Default**: `1.25`
Azimuthal Limits (deg)	Azimuthal limits of the lidar sensor, in degrees, specified as a two-element vector of decimal scalars of the form `[min, max]`. **Default**: `[-45 45]`
Elevation Limits (deg)	Elevation limits of the lidar sensor, in degrees, specified as a two-element vector of decimal scalars of the form `[min, max]`. **Default**: `[-20 20]`
Has Noise	Select this parameter to enable adding noise to sensor measurements. **Default**: `off`

`Settings` — Simulation sample time, stop condition, and stop time
dialog box

To access these parameters, on the app toolstrip, click **Settings**.

Simulation Settings

Parameter	Description
Sample Time (ms)	Frequency at which the simulation updates, in milliseconds. Increase the sample time to speed up simulation. This increase has no effect on actor speeds, even though actors can appear to go faster during simulation. The actor positions are just being sampled and displayed on the app at less frequent intervals, resulting in faster, choppier animations. Decreasing the sample time results in smoother animations, but the actors appear to move slower, and the simulation takes longer. The sample time does not correlate to the actual time. For example, if the app samples every 0.1 seconds (**Sample Time (ms)** = `100`) and runs for 10 seconds, the amount of elapsed actual time might be less than the 10 seconds of elapsed simulation time. Any apparent synchronization between the sample time and actual time is coincidental. **Default**: `10`
Stop Condition	Stop condition of simulation, specified as one of these values: • `First actor stops` — Simulation stops when the first actor reaches the end of its trajectory. • `Last actor stops` — Simulation stops when the last actor reaches the end of its trajectory. • `Set time` — Simulation stops at the time specified by the **Stop Time (s)** parameter. **Default**: `First actor stops`
Stop Time (s)	Stop time of simulation, in seconds, specified as a positive decimal scalar. To enable this parameter, set the **Stop Condition** parameter to `Set time`. **Default**: `0.1`

Parameter	Description
Use RNG Seed	Select this parameter to use a random number generator (RNG) seed to reproduce the same results for each simulation. Specify the RNG seed as a nonnegative integer less than 2^{32}. **Default**: `off`

Programmatic Use

`drivingScenarioDesigner` opens the **Driving Scenario Designer** app.

`drivingScenarioDesigner(scenarioFileName)` opens the app and loads the specified scenario MAT-file into the app. This file must be a scenario file saved from the app. This file can include all roads, actors, and sensors in the scenario. It can also include only the roads and actors component, or only the sensors component.

If the scenario file is not in the current folder or not in a folder on the MATLAB path, specify the full path name. For example:

```
drivingScenarioDesigner('C:\Desktop\myDrivingScenario.mat');
```

You can also load prebuilt scenario files. Before loading a prebuilt scenario, add the folder containing the scenario to the MATLAB path. For an example, see “Generate Sensor Data from Scenario” on page 1-18.

`drivingScenarioDesigner(scenario)` loads the specified `drivingScenario` object into the app. The `ClassID` properties of actors in this object must correspond to these default **Class ID** parameter values in the app:

- `1` — Car
- `2` — Truck
- `3` — Bicycle
- `4` — Pedestrian
- `5` — Barrier

When you create actors in the app, the actors with these **Class ID** values have a default set of dimensions, radar cross-section patterns, and other properties. The camera and radar sensors process detections differently depending on type of actor specified by the **Class ID** values.

When importing `drivingScenario` objects into the app, the behavior of the app depends on the `ClassID` of the actors in that scenario.

- If an actor has a `ClassID` of `0`, the app returns an error. In `drivingScenario` objects, a `ClassID` of `0` is reserved for an object of an unknown or unassigned class. The app does not recognize or use this value. Assign these actors one of the app **Class ID** values and import the `drivingScenario` object again.
- If an actor has a nonzero `ClassID` that does not correspond to a **Class ID** value, the app returns an error. Either change the `ClassID` of the actor or add a new actor class to the app. On the app toolstrip, select **Add Actor** > **New Actor Class**.

- If an actor has properties that differ significantly from the properties of its corresponding **Class ID** actor, the app returns a warning. The `ActorID` property referenced in the warning corresponds to the ID value of an actor in the list at the top of the **Actors** tab. The ID value precedes the actor name. To address this warning, consider updating the actor properties or its `ClassID` value. Alternatively, consider adding a new actor class to the app.

`drivingScenarioDesigner(___,sensors)` loads the specified sensors into the app, using any of the previous syntaxes. Specify `sensors` as a `radarDetectionGenerator` object, `visionDetectionGenerator` object, `lidarPointCloudGenerator` object, or cell array of such objects. If you specify `sensors` along with a scenario file that contains sensors, the app does not import the sensors from the scenario file.

For an example of importing sensors, see "Import Programmatic Driving Scenario and Sensors" on page 1-21.

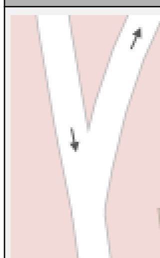
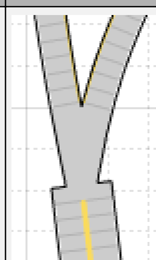
Limitations

OpenStreetMap — Import Limitations

When importing OpenStreetMap data, road and lane features have these limitations:

- Lane-level information is not imported from OpenStreetMap roads. Lane specifications are based only on the direction of travel specified in the OpenStreetMap road network, where:
  - One-way roads are imported as single-lane roads with default lane specifications. These lanes are programmatically equivalent to `lanespec(1)`.
  - Two-way roads are imported as two-lane roads with bidirectional travel and default lane specifications. These lanes are programmatically equivalent to `lanespec([1 1])`.

  The table shows these differences in the OpenStreetMap road network and the road network in the imported driving scenario.

OpenStreetMap Road Network	Imported Driving Scenario
	

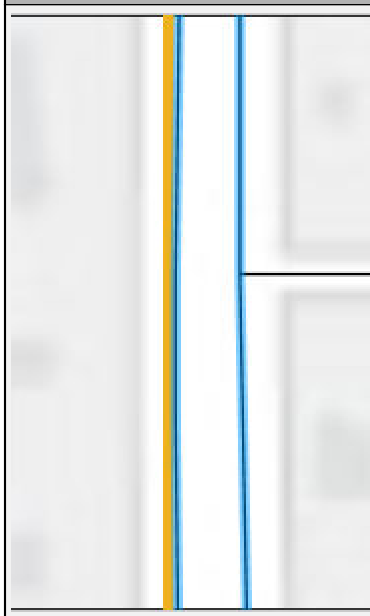
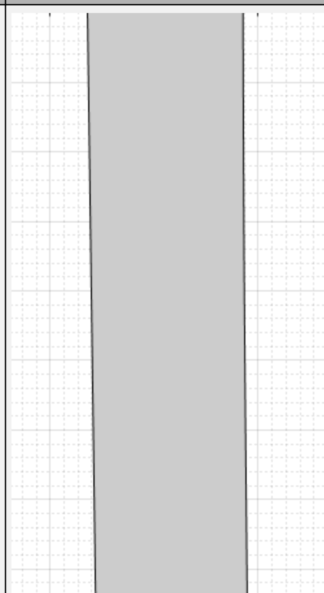
- When importing OpenStreetMap road networks that specify elevation data, if elevation data is not specified for all roads being imported, then the generated road network might contain inaccuracies and some roads might overlap.
- The basemap used in the app can have slight differences from the map used in the OpenStreetMap service. Some imported road issues might also be due to missing or inaccurate map data in the OpenStreetMap service. To check whether the data is missing or inaccurate due to the map service, consider viewing the map data on an external map viewer.
- If you receive a warning that the geometry of a road is unable to be computed, then the curvature of the road is too sharp for it to render properly and it is not imported.

HERE HD Live Map — Import Limitations

When importing HERE HDLM data, these road and lane features are not supported:

- Lanes with varying widths — In the generated road network, each lane is set to have the maximum width found along its entire length. Consider a HERE HDLM lane with a width that varies from 2 to 4 meters along its length. In the generated road network, the lane width is 4 meters along its entire length.
- Roads with varying numbers of lanes along their lengths — In the generated road network, each road is set to have the maximum number of lanes along its entire length. Consider a HERE HDLM road with 3 lanes on one half and 2 lanes on the other half. In the generated road network, the road has 3 lanes along its entire length.
- Multiple lane marking styles along a lane — In the generated road network, each lane is set to have the marking style of the lane segment with the maximum width along the road. Consider a HERE HDLM lane with 2 lane segments. The first lane segment is 2 meters wide and has solid markings. The second lane segment is 4 meters wide and has dashed markings. In the generated road network, the lane has a fixed width of 4 meters throughout and dashed markings along its entire length.

These modifications to the road networks can sometimes cause roads to overlap in the driving scenario. Consider the HERE HDLM roads for the divided highway highlighted in blue in the table. Due to the unsupported features, in the imported driving scenario, the lane widths of the roads increase. This limitation causes the roads to overlap and appear as one road. Sensors that detect lanes are unable to detect the covered lanes.

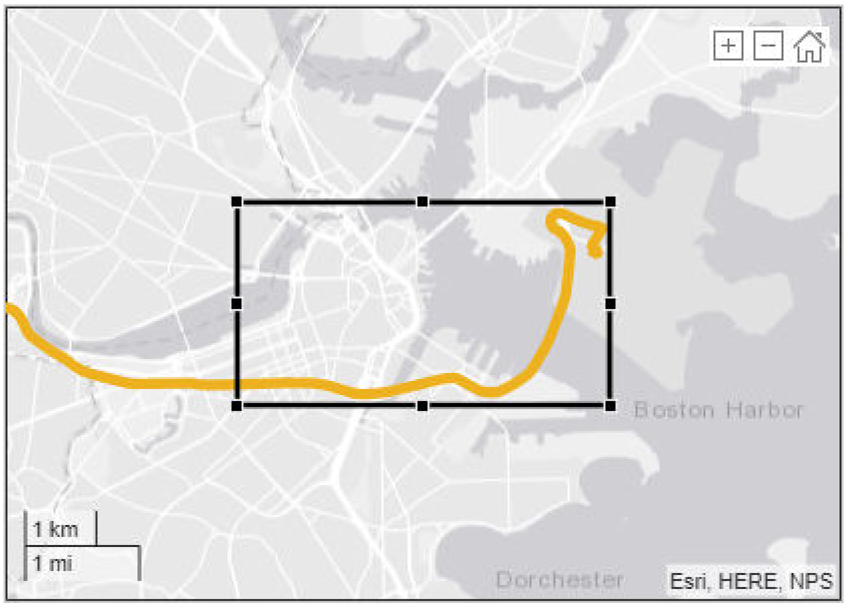
HERE HDLM Road Network	Imported Driving Scenario
	

If you receive a warning that the geometry of a road is unable to be computed, then the curvature of the road is too sharp for it to render properly and it is not imported.

In addition to the unsupported features, the basemap used in the app might have slight differences from the map used in the HERE HDLM service. Some issues with the imported roads might also be due to missing or inaccurate map data in the HERE HDLM service. To check where the issue stems from in the map data, use the HERE HD Live Map Viewer to view the geometry of the HERE HDLM road network. This viewer requires a valid HERE license. For more details, see the HERE Technologies website.

HERE HD Live Map — Route Selection Limitations

When selecting HERE HD Live Map roads to import from a region of interest, the maximum allowable size of the region is 20 square kilometers. If you specify a driving route that is greater than 20 square kilometers, the app draws a region that is optimized to fit as much of the beginning of the route as possible into the display. This figure shows an example of a region drawn around the start of a route that exceeds this maximum size.



OpenDRIVE Import Limitations

- 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.

Euro NCAP Limitations

- Scenarios of speed assistance systems (SAS) are not supported. These scenarios require the detection of speed limits from traffic signs, which the app does not support.

3D Display Limitations

These limitations describe how **3D Display** visualizations differ from the cuboid visualizations that appear on the **Scenario Canvas**.

- Roads do not form junctions with unmarked lanes at intersections. The roads and their lane markings overlap.
- Not all actor or lane marking colors are supported. The 3D display matches the selected color to the closest available color that it can render.
- Lane type colors of nondriving lanes are not supported. If you select a nondriving lane type, in the 3D display, the lane displays as a driving lane.
- On the **Actors** tab, specified **Roll (°)** and **Pitch (°)** parameter values of an actor are ignored. In the **Waypoints** table, **z (m)** values (that is, elevation values) are also ignored. During simulation, actors follow the elevation and banking angle of the road surface.
- Multiple marking styles along a lane are not supported. The 3D display applies the first lane marking style of the first lane segment along the entire length of the lane.
- Actors with a **3D Display Type** of `Cuboid` do not move in the 3D display. During simulation, these actors remain stationary at their initial specified positions.

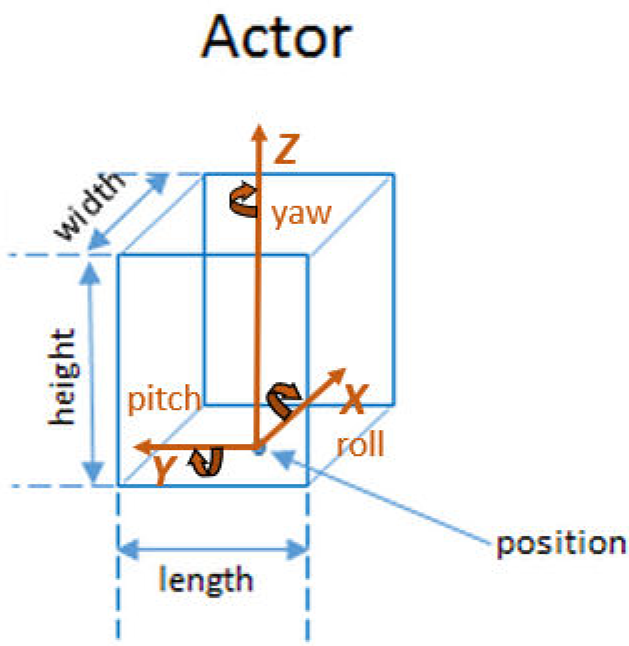
More About

Actor and Vehicle Positions and Dimensions

In driving scenarios, an actor is a cuboid (box-shaped) object with a specific length, width, and height. Actors also have a radar cross-section (RCS) pattern, specified in dBsm, which you can refine by setting angular azimuth and elevation coordinates. The position of an actor is defined as the center of its bottom face. This center point is used as the actor's rotational center, its point of contact with the ground, and its origin in its local coordinate system. In this coordinate system:

- The *X*-axis points forward from the actor.
- The *Y*-axis points left from the actor.
- 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.



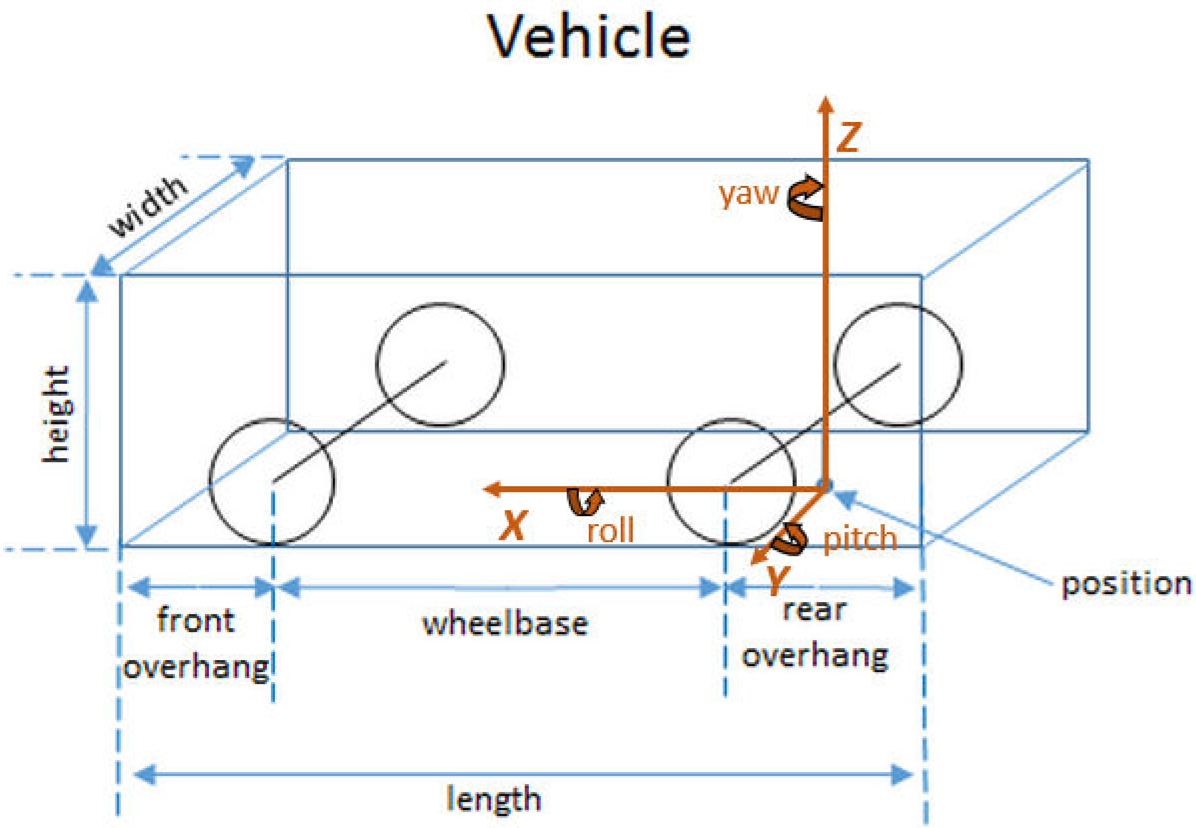
A vehicle is an actor that moves on wheels. Vehicles have three extra properties that govern the placement of their front and rear axle.

- Wheelbase — Distance between the front and rear axles
- Front overhang — Distance between the front of the vehicle and the front axle
- Rear overhang — Distance between the rear axle and the rear of the vehicle

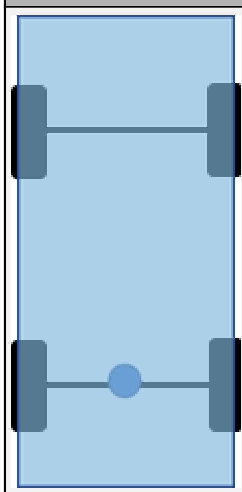
Unlike other types of actors, the position of a vehicle is defined by the point on the ground that is below the center of its rear axle. This point corresponds to the natural center of rotation of the vehicle. As with nonvehicle actors, this point is the origin in the local coordinate system of the vehicle, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points left from 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.



The origin (that is, the position) of cuboid vehicles differs from the origin of vehicles in the 3D simulation environment. In the 3D simulation environment, vehicle origins are on the ground, at the geometric center of the vehicle.

Cuboid Vehicle Origin	3D Simulation Vehicle Origin
	

For nonvehicle actors, the origins are identical and located at the bottom of the geometric center of the actors.

In Simulink, to convert a vehicle from the cuboid origin to the 3D simulation origin, use a Cuboid To 3D Simulation block. For more details about 3D simulation coordinates, see “Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox”.

Color Specifications for Lanes and Markings

This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes that you can use for specifying the color of lanes and markings in a road.

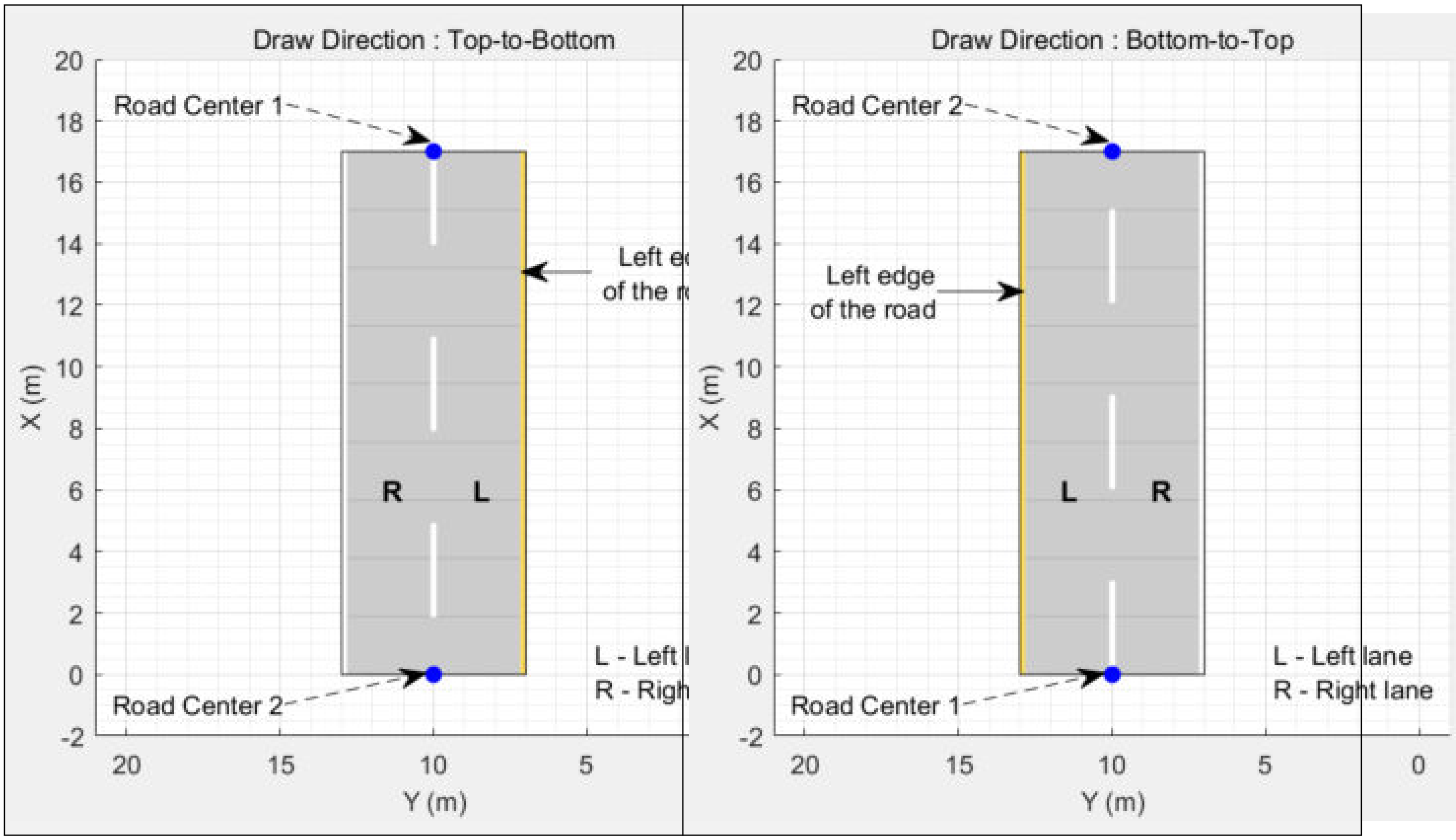
Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`red`	`r`	`[1 0 0]`	`#FF0000`	
`green`	`g`	`[0 1 0]`	`#00FF00`	
`blue`	`b`	`[0 0 1]`	`#0000FF`	
`cyan`	`c`	`[0 1 1]`	`#00FFFF`	
`magenta`	`m`	`[1 0 1]`	`#FF00FF`	
`yellow`	`y`	`[0.98 0.86 0.36]`	`#FADB5C`	
`black`	`k`	`[0 0 0]`	`#000000`	
`white`	`w`	`[1 1 1]`	`#FFFFFF`	

Draw Direction of Road and Numbering of Lanes

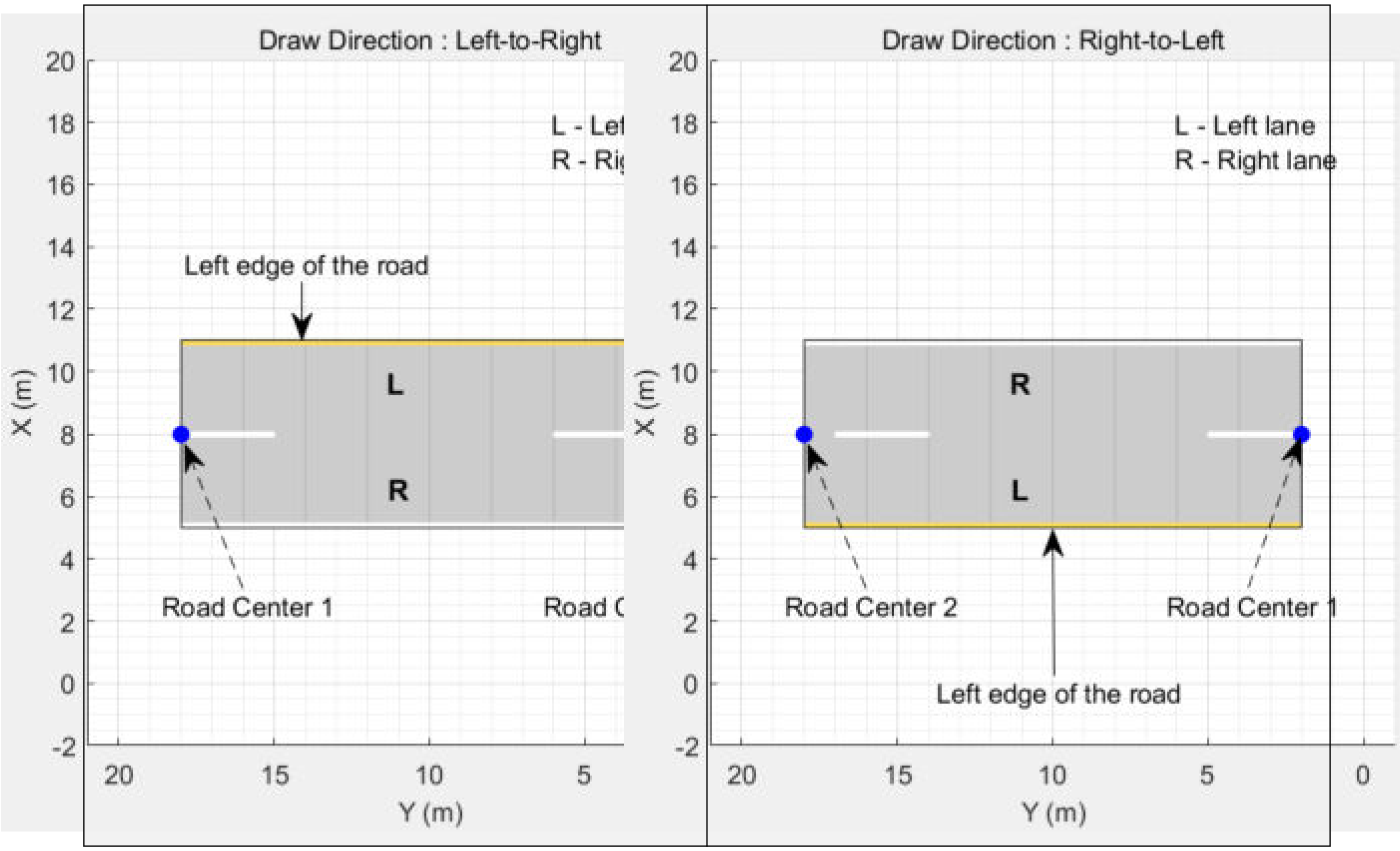
To create a road by using the `road` function, specify the road centers as a matrix input. The function creates a directed line that traverses the road centers, starting from the coordinates in the first row of the matrix and ending at the coordinates in the last row of the matrix. The coordinates in the first two rows of the matrix specify the draw direction of the road. These coordinates correspond to the first two consecutive road centers. The draw direction is the direction in which the roads render in the scenario plot.

To create a road by using the **Driving Scenario Designer** app, you can either specify the **Road Centers** parameter or interactively draw on the **Scenario Canvas**. For a detailed example, see “Create a Driving Scenario” on page 1-14. In this case, the draw direction is the direction in which roads render in the **Scenario Canvas**.

- For a road with a top-to-bottom draw direction, the difference between the *x*-coordinates of the first two consecutive road centers is positive.
- For a road with a bottom-to-top draw direction, the difference between the *x*-coordinates of the first two consecutive road centers is negative.



- For a road with a left-to-right draw direction, the difference between the *y*-coordinates of the first two consecutive road centers is positive.
- For a road with a right-to-left draw direction, the difference between the *y*-coordinates of the first two consecutive road centers is negative.



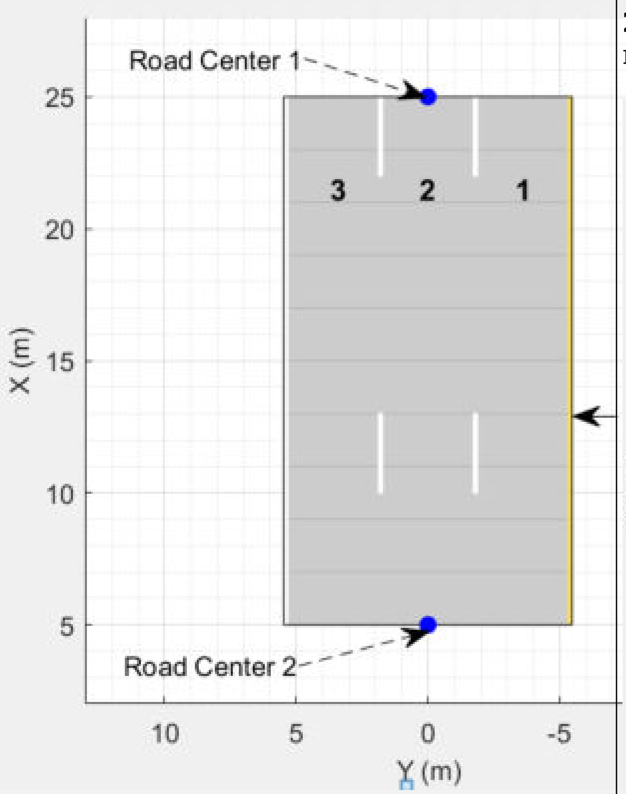
Numbering Lanes

Lanes must be numbered from left to right, with the left edge of the road defined relative to the draw direction of the road. For a one-way road, by default, the left edge of the road is a solid yellow marking which indicates the end of the road in transverse direction (direction perpendicular to draw direction). For a two-way road, by default, both edges are marked with solid white lines.

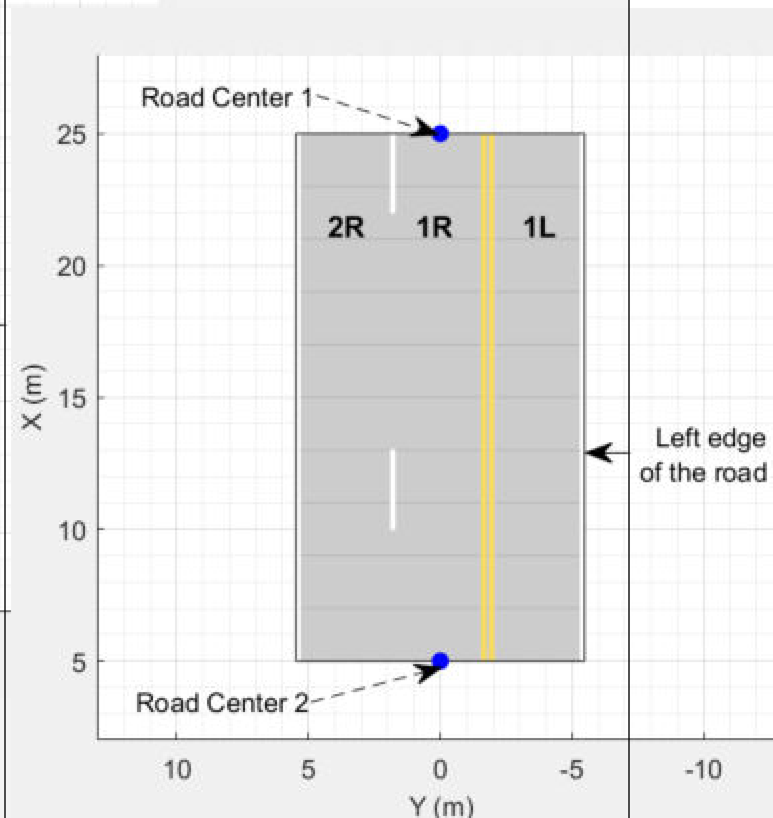
For example, these diagrams show how the lanes are numbered in a one-way and two-way road with a draw direction from top-to-bottom.

Numbering Lanes in a One-Way Road	Numbering Lanes in a Two-Way Road

<table>
<tr>
<td>
Specify the number of lanes as a positive integer for a one-way road. If you set the integer value as 3, then the road has three lanes that travel in the same direction. The lanes are numbered starting from the left edge of the road.

1, 2, 3 denote the first, second, and third lanes of the road, respectively.


</td>
<td>
Specify the number of lanes as a two-element vector of positive integer for a two-way road. If you set the vector as [1 2], then the road has three lanes: two lanes traveling in one direction and one lane traveling in the opposite direction. Because of the draw direction, the road has one left lane and two right lanes. The lanes are numbered starting from the left edge of the road.

1L denote the only left lane of the road. 1R and 2R denote the first and second right lanes of the road, respectively.


</td>
</tr>
</table>

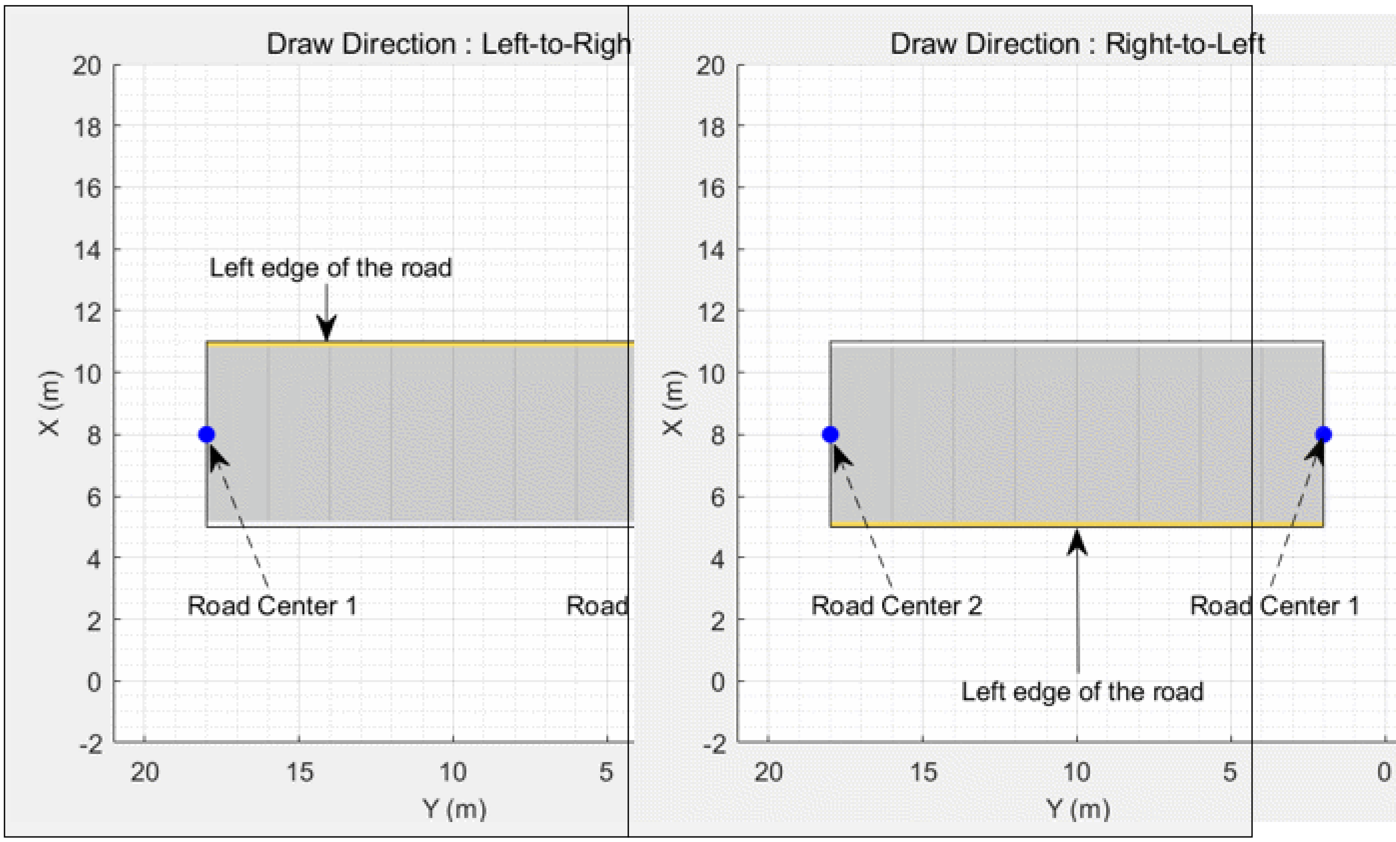
The lane specifications apply by the order in which the lanes are numbered.

Composite Lane Marking

A composite lane marking comprises two or more marker segments that define multiple marking types along a lane. The geometric properties for a composite lane marking include the geometric properties of each marking type and the normalized lengths of the marker segments.

The order in which the specified marker segments occur in a composite lane marking depends on the draw direction of the road. Each marker segment is a directed segment with a start point and moves towards the last road center. The first marker segment starts from the first road center and moves towards the last road center for a specified length. The second marker segment starts from the end point of the first marker segment and moves towards the last road center for a specified length. The same process applies for each marker segment that you specify for the composite lane marking. You can set the normalized length for each of these marker segments by specifying the `range` input argument.

For example, consider a one-way road with two lanes. The second lane marking from the left edge of the road is a composite lane marking with marking types `Solid` and `Dashed`. The normalized range for each marking type is 0.5. The first marker segment is a solid marking and the second marker segment is a dashed marking. These diagrams show the order in which the marker segments apply for left-to-right and right-to-left draw directions of the road.

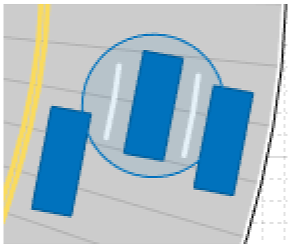


For information on the geometric properties of lane markings, see “Lane Specifications” on page 4-502.

Tips

- When importing map data, the map regions you specify and the number of roads you select have a direct effect on app performance. To improve performance, specify the smallest map regions and select the fewest roads that you need to create your driving scenario.
- You can undo (press **Ctrl+Z**) and redo (press **Ctrl+Y**) changes you make on the scenario and sensor canvases. For example, you can use these shortcuts to delete a recently placed road center or redo the movement of a radar sensor. For more shortcuts, see “Keyboard Shortcuts and Mouse Actions for Driving Scenario Designer”
- In scenarios that contain many actors, to keep track of the ego vehicle, you can add an indicator around the vehicle. On the app toolstrip, select **Display > Show ego indicator**. The circle around

the ego vehicle highlights the location of the vehicle in the scenario. This circle is not a sensor coverage area.



Compatibility Considerations

Corrections to Image Width and Image Height camera parameters of Driving Scenario Designer
Behavior changed in R2018b

Starting in R2018b, in the **Camera Settings** group of the **Driving Scenario Designer** app, the **Image Width** and **Image Height** parameters set their expected values. Previously, **Image Width** set the height of images produced by the camera, and **Image Height** set the width of images produced by the camera.

If you are using R2018a, to produce the expected image sizes, transpose the values set in the **Image Width** and **Image Height** parameters.

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.

[4] 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
Bird's-Eye Scope

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

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

Topics
"Create Driving Scenario Interactively and Generate Synthetic Sensor Data"
"Create Reverse Motion Driving Scenarios Interactively"
"Import OpenDRIVE Roads into Driving Scenario"
"Import HERE HD Live Map Roads into Driving Scenario"
"Import OpenStreetMap Data into Driving Scenario"
"Generate Sensor Detection Blocks Using Driving Scenario Designer"
"Test Open-Loop ADAS Algorithm Using Driving Scenario"
"Test Closed-Loop ADAS Algorithm Using Driving Scenario"
"Keyboard Shortcuts and Mouse Actions for Driving Scenario Designer"
"Prebuilt Driving Scenarios in Driving Scenario Designer"

External Websites
Euro NCAP Safety Assist Protocols
ASAM OpenDRIVE
HERE Technologies
openstreetmap.org

Introduced in R2018a

Ground Truth Labeler

Label ground truth data for automated driving applications

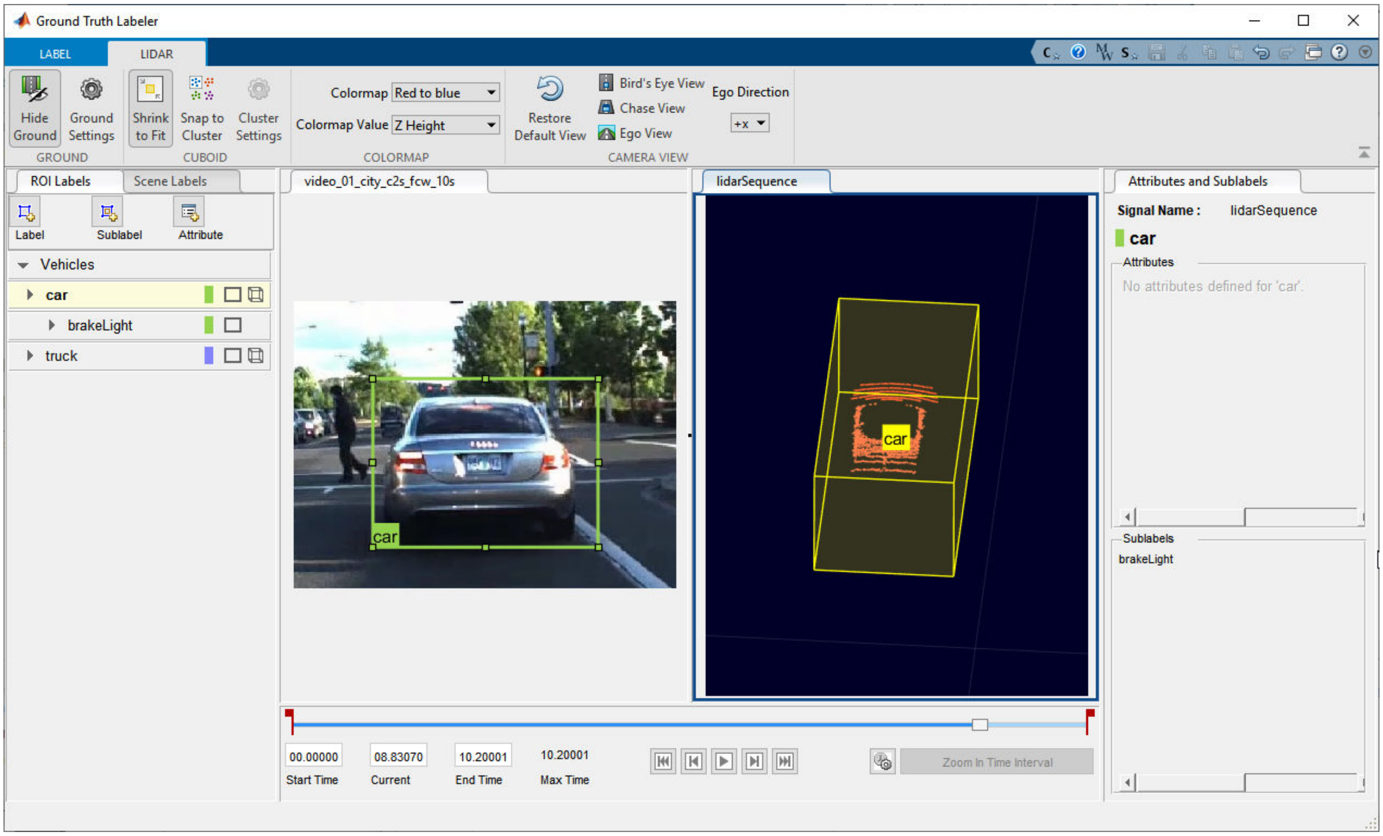
Description

The **Ground Truth Labeler** app enables you to label ground truth data in multiple videos, image sequences, or lidar point clouds.

Using the app, you can:

- Simultaneously label multiple time-overlapped signals representing the same scene.
- Define rectangular region of interest (ROI) labels, polyline ROI labels, pixel ROI labels, cuboid ROI labels for lidar labeling, and scene label definitions. Use these labels to interactively label your ground truth data.
- Use built-in detection or tracking algorithms to label ground truth data.
- Write, import, and use custom automation algorithms to automatically label ground truth data.
- Evaluate the performance of your label automation algorithms by using a visual summary.
- Export the ground truth labels as a `groundTruthMultisignal` object. You can use this object for system verification or for training an object detector or semantic segmentation network.
- Display time-synchronized signals, such as CAN bus data, by using the `driving.connector.Connector` API.

To learn more about this app, see “Get Started with the Ground Truth Labeler”.



Open the Ground Truth Labeler App

- MATLAB Toolstrip: On the **Apps** tab, under **Automotive**, click the app icon.
- MATLAB command prompt: Enter `groundTruthLabeler`.

Examples

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

Programmatic Use

`groundTruthLabeler` opens a new session of the app, enabling you to label ground truth data.

`groundTruthLabeler(videoFileName)` opens the app and loads the input video. The video file must have an extension supported by `VideoReader`.

Example: `groundTruthLabeler('caltech_cordova1.avi')`

`groundTruthLabeler(imageSeqFolder)` opens the app and loads the image sequence from the input folder. An image sequence is an ordered set of images that resembles a video.

`imageSeqFolder` must be a string scalar or character vector that specifies the folder containing the image files. The image files must have extensions supported by `imformats` and are loaded in the order returned by the `dir` function.

The images in `imageSeqFolder` must be the same size. If the images vary in size, the app imports only the images that are of the same size as the first image in the sequence. To label a collection of unordered images that vary in size, use the **Image Labeler** app instead.

`groundTruthLabeler(imageSeqFolder,timestamps)` opens the app and loads a sequence of images with their corresponding timestamps. `timestamps` must be a `duration` vector of the same length as the number of images in the sequence.

For example, load a sequence of road images and their corresponding timestamps into the app.

```
imageDir = fullfile(toolboxdir('driving'),'drivingdata','roadSequence');
load(fullfile(imageDir,'timeStamps.mat'))
groundTruthLabeler(imageDir,timeStamps)
```

`groundTruthLabeler( ___ ,'ConnectorTargetHandle',connector)` opens the app and loads both of these components:

- A video or image sequence signal, depending on the input argument combination you specify
- An external analysis or visualization tool that is time-synchronized with the specified signal

The `connector` input is a handle to a `driving.connector.Connector` class that implements the external tool.

For example, this syntax opens the app with a video signal and synchronized lidar visualization tool.

```
groundTruthLabeler('01_city_c2s_fcw_10s.mp4','ConnectorTargetHandle',@LidarDisplay);
```

When you have an external tool connected to a signal in the app, consider these tips.

- If you remove the signal that is connected to the tool, the app disconnects the tool and closes it.
- The signal connected to the tool must be the master signal, that is, the signal whose timestamps are used in the playback controls at the bottom of the app. If you change the master signal, the app disconnects the tool and closes it.
- If you start a new app session, the app disconnects the tool and closes it.

`groundTruthLabeler(sessionFile)` opens the app and loads a saved app session, `sessionFile`. The `sessionFile` input contains the path and file name. The MAT-file that `sessionFile` points to contains the saved session.

Limitations

- Lidar signals do not support line or pixel ROI labels.
- Pixel ROI labels do not support sublabels or attributes.
- Cuboid ROI labels do not support sublabels.

- The Label Summary window does not support sublabels or attributes

More About

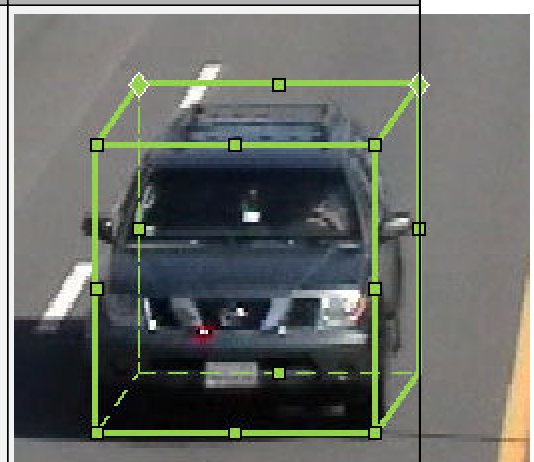
ROI Labels, Sublabels, and Attributes

On the left side of the app, the **ROI Labels** pane contains the region of interest (ROI) label definitions that you can mark on the frames. You can create label definitions directly from this pane. Alternatively, you can create label definitions programmatically by using a `labelDefinitionCreatorMultisignal` object and then import these label definitions into an app session.

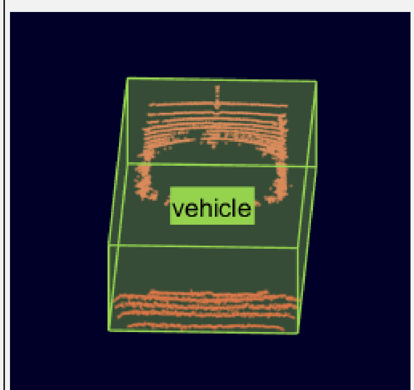
The app supports the definition of ROI labels, sublabels, and attributes.

ROI Labels

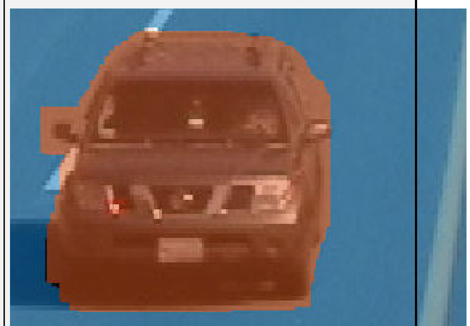
An ROI label is a label that corresponds to a region of interest (ROI) in a signal frame. The table describes the supported label types.

ROI Label	Description	Example: Driving Scene
`Projected cuboid`	Draw cuboidal ROI labels (3-D bounding boxes).	

<table>
<tr><th>ROI Label</th><th>Description</th><th>Example: Driving Scene</th></tr>
<tr><td><code>Rectangle/Cuboid</code></td><td>Draw rectangular or cuboidal ROI labels around objects, depending on the signal type.
In image signals, draw rectangular ROI labels (2-D bounding boxes).
In lidar signals, draw cuboidal ROI labels (3-D bounding boxes). For more on lidar labeling, see “Label Lidar Point Clouds for Object Detection”.</td><td>Vehicles, pedestrians, road signs
<code>Rectangle</code>:


<code>Cuboid</code>:

</td></tr>
<tr><td><code>Line</code></td><td>Draw linear ROI labels to represent lines. To draw a polyline ROI, use two or more points.</td><td>Lane boundaries, guard rails, road curbs

</td></tr>
</table>

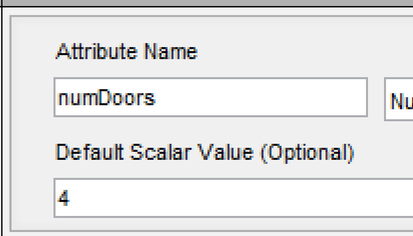
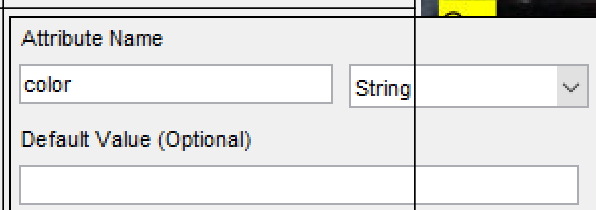
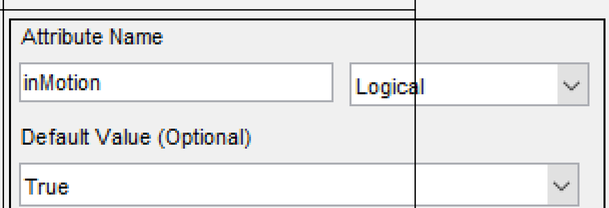
ROI Label	Description	Example: Driving Scene
`Pixel label`	Assign labels to pixels for semantic segmentation. You can label pixels manually using polygons, brushes, or flood fill. For more on pixel labeling, see “Label Pixels for Semantic Segmentation” (Computer Vision Toolbox).	Vehicles, road surface,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

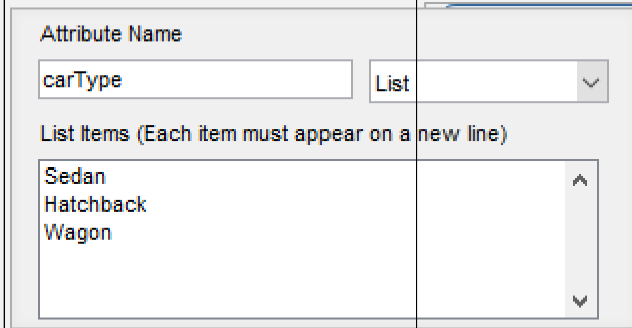
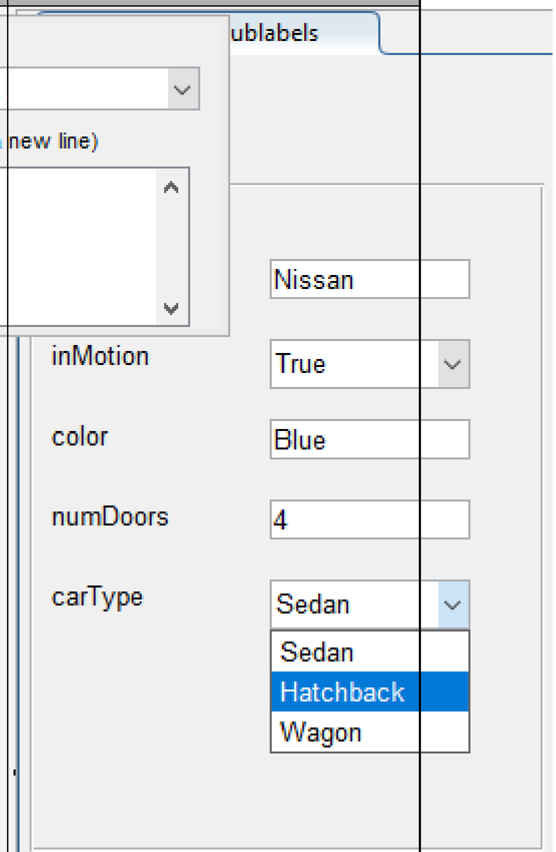
ROI Sublabels

An ROI sublabel is an ROI label that belongs to a parent label. Use ROI sublabels to provide a greater level of detail about the ROIs in your labeled ground truth data. For example, a **vehicle** label might contain **headlight**, **licensePlate**, and **wheel** sublabels. You can create sublabels only for rectangular and polyline labels. For more details about sublabels, see “Use Sublabels and Attributes to Label Ground Truth Data” (Computer Vision Toolbox).

ROI Attributes

An ROI attribute specifies additional information about an ROI label or sublabel. For example, in a driving scene, attributes might include the type or color of a vehicle. The table describes the supported attribute types.

Attribute Type	Sample Attribute Definition	Sample Default Values
`Numeric Value`		
`String`		
`Logical`		

Attribute Type	Sample Attribute Definition	Sample Default Values
`List`		

For more details about attributes, see "Use Sublabels and Attributes to Label Ground Truth Data" (Computer Vision Toolbox).

Tips

- To avoid having to relabel ground truth with new labels, organize the labeling scheme you want to use before marking your ground truth.
- You can copy and paste labels between signals that are of the same type.

Algorithms

You can use label automation algorithms to speed up labeling within the app. To create your own label automation algorithm to use within the app, see "Create Automation Algorithm for Labeling" (Computer Vision Toolbox). You can also use one of the provided built-in algorithms. Follow these steps:

1. Load the data you want to label, and create at least one label definition.
2. On the app toolstrip, click **Select Algorithm**, and select one of the built-in automation algorithms.
3. Click **Automate**, and then follow the automation instructions in the right pane of the automation window.

ACF Vehicle Detector

Detect and label vehicles using aggregate channel features (ACF). This algorithm is based on the `vehicleDetectorACF` function. To use this algorithm, you must define at least one rectangle ROI label. You do not need to draw any ROI labels.

To help improve the algorithm results, first click **Settings**. You can change any of these settings.

- The pretrained vehicle detector model that the algorithm uses — The `'full-view'` model was trained using unoccluded images of the front, rear, left, and right sides of vehicles. The `'front-rear-view'` model was trained using images of only the front and rear sides of the vehicle.
- The overlap ratio threshold, from 0 to 1, for detecting vehicles — When rectangle ROIs overlap by more than this threshold, the algorithm discards one of the ROIs.
- The classification score threshold for detecting vehicles — Increase the score to increase the prediction confidence of the algorithm. Rectangles with scores below this threshold are discarded.

You can also configure the detector with a calibrated monocular camera by importing a `monoCamera` object into the MATLAB workspace. Specify the length and width ranges of the vehicle in world units, such as meters.

ACF People Detector

Detect and label people using aggregate channel features (ACF). This algorithm is based on the `peopleDetectorACF` function. To use this algorithm, you must define at least one rectangle ROI label. You do not need to draw any ROI labels.

To help improve the algorithm results, first click **Settings**. You can change any of these settings.

- The pretrained people detector model that the algorithm uses — The `'inria-100x41'` model was trained using the INRIA person data set. The `'caltech-50x21'` model was trained using the Caltech Pedestrian data set.
- The overlap ratio threshold, from 0 to 1, for detecting people — When rectangle ROIs overlap by more than this threshold, the algorithm discards one of the ROIs.
- The classification score threshold for detecting people — Increase the score to increase the prediction confidence of the algorithm. Rectangles with scores below this threshold are discarded.

Point Tracker

Track and label one or more rectangle ROI labels over short intervals by using the Kanade-Lucas-Tomasi (KLT) algorithm. This algorithm is based on the `vision.PointTracker` System object™. To use this algorithm, you must define at least one rectangle ROI label, but you do not need to draw any ROI labels.

To change the feature detector used to obtain the initial points for tracking, click **Settings**. This table shows the feature detector options.

Feature Detector	Description	Equivalent Function
`Minimum Eigen Value`	Detect corners by using the minimum eigenvalue algorithm.	`detectMinEigenFeatures`
`Harris`	Detect corners by using the Harris–Stephens algorithm.	`detectHarrisFeatures`

Feature Detector	Description	Equivalent Function
`FAST`	Detect corners by using the features from accelerated segment test (FAST) algorithm.	`detectFASTFeatures`
`BRISK`	Detect features by using the binary robust invariant scalable keypoints (BRISK) algorithm.	`detectBRISKFeatures`
`KAZE`	Detect features by using nonlinear diffusion to construct a scale space of an image, and then detecting multiscale corner features (KAZE features) from that scale space.	`detectKAZEFeatures`
`SURF`	Detect blob features by using the speeded-up robust features (SURF) algorithm.	`detectSURFFeatures`
`MSER`	Detect regions by using the maximally stable extremal regions (MSER) algorithm.	`detectMSERFeatures`

Temporal Interpolator

Estimate rectangle ROIs between frames by interpolating the ROI locations across the time interval. To use this algorithm, you must draw a rectangle ROI on a minimum of two frames: one at the beginning of the interval and one at the end of the interval. The interpolation algorithm estimates and draws ROIs in the intermediate frames.

Consider a video with 10 frames. The first frame has a rectangle ROI centered at [5, 5]. The 10th frame has a rectangle ROI centered at [25, 25]. At each frame, the algorithm moves the ROI 2 pixels in the *x*-direction and 2 pixels in the *y*-direction. Therefore, the algorithm centers the ROI at [7, 7] in the second frame, [9, 9] in the third frame, and so on, up to [23, 23] in the second-to-last frame.

Point Cloud Temporal Interpolator

Estimate cuboid ROIs between point cloud frames by interpolating the ROI locations across the time interval. To use this algorithm, you must draw a cuboid ROI on a minimum of two frames: one at the beginning of the interval and one at the end of the interval. The interpolation algorithm estimates and draws ROIs in the intermediate frames.

Consider a point cloud sequence with 10 frames. The first frame has a cuboid ROI centered at [5, 5, 0]. The 10th frame has a cuboid ROI centered at [25, 25, 0]. At each frame, the algorithm moves the ROI 2 points in the *x*-direction, 2 points in the *y*-direction, and 0 points in the *z*-direction. Therefore, the algorithm centers the ROI at [7, 7, 0] in the second frame, [9, 9, 0] in the third frame, and so on, up to [23, 23, 0] in the second-to-last frame.

Lane Boundary Detector

Detect and label lane boundaries using an estimated bird's-eye-view projected image. To use this algorithm, you must define at least one line ROI label. You do not need to draw any ROI labels. To detect lane boundaries, the algorithm follows these steps:

1 It makes an initial guess at the placement of the lane boundaries in the image.
2 It transforms the ROI around the lanes into a bird's-eye view image to make the lanes parallel and remove distortion.
3 It uses this image to segment the lane boundaries.

To help improve the algorithm results, first click **Settings**. You can change any of these settings.

- The placement of the lane lines for generating the bird's-eye view image
- The ROI around the lanes, which you can expand to include more than just the ego lane boundaries in the image
- The pixel width of detected lane boundaries in the image

You can also change the number of lane boundaries that you want to detect. The default number of lane boundaries is 2.

Compatibility Considerations

Ground Truth Labeler app no longer exports `groundTruth` objects
Behavior change in future release

If you import labels or open an app session created before R2020a, the **Ground Truth Labeler** exports labeled data as a `groundTruthMultisignal` object instead of as a `groundTruth` object.

If you do not need to label multiple signals simultaneously and do not require lidar labeling, import the labels or session into the **Video Labeler** app instead. The **Video Labeler** app continues to export `groundTruth` objects.

See Also

Apps
Image Labeler | **Video Labeler**

Objects
`groundTruthDataSource` | `groundTruthMultisignal` | `labelDefinitionCreatorMultisignal`

Classes
`driving.connector.Connector` | `vision.labeler.AutomationAlgorithm` | `vision.labeler.loading.MultiSignalSource` | `vision.labeler.mixin.Temporal`

Topics
"Get Started with the Ground Truth Labeler"
"Automate Ground Truth Labeling of Lane Boundaries"
"Automate Ground Truth Labeling for Semantic Segmentation"
"Automate Attributes of Labeled Objects"
"Evaluate Lane Boundary Detections Against Ground Truth Data"
"Evaluate and Visualize Lane Boundary Detections Against Ground Truth"
"Choose an App to Label Ground Truth Data" (Computer Vision Toolbox)
"Keyboard Shortcuts and Mouse Actions for Ground Truth Labeler"
"Label Pixels for Semantic Segmentation" (Computer Vision Toolbox)
"Label Lidar Point Clouds for Object Detection"

“Create Class for Loading Custom Ground Truth Data Sources”
“Create Automation Algorithm for Labeling” (Computer Vision Toolbox)
“Share and Store Labeled Ground Truth Data” (Computer Vision Toolbox)

Introduced in R2017a

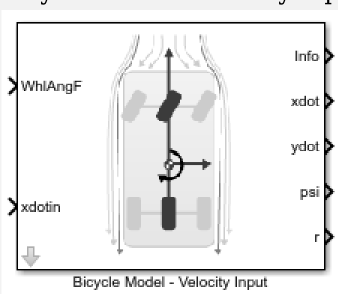
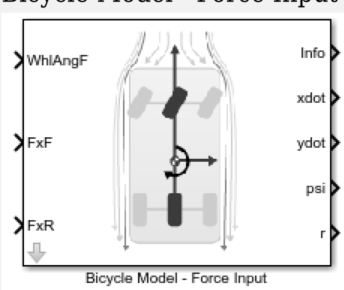
Blocks

Bicycle Model

Implement a single track 3DOF rigid vehicle body to calculate longitudinal, lateral, and yaw motion

Description

The Bicycle Model block implements a rigid two-axle single track vehicle body model to calculate longitudinal, lateral, and yaw motion. The block accounts for body mass, aerodynamic drag, and weight distribution between the axles due to acceleration and steering. There are two types of Bicycle Model blocks.

Block	Implementation
Bicycle Model - Velocity Input  	• Block assumes that the external longitudinal velocity is quasi-steady state so the longitudinal acceleration is approximately zero. • Since the motion is quasi-steady, the block calculates only lateral forces using the tire slip angles and linear cornering stiffness.
Bicycle Model - Force Input  	• Block uses the external longitudinal force to accelerate or brake the vehicle. • Block calculates lateral forces using the tire slip angles and linear cornering stiffness.

To calculate the normal forces on the front and rear axles, the block uses rigid-body vehicle motion, suspension system forces, and wind and drag forces. The block resolves the force and moment components on the rigid vehicle body frame.

Ports

Input

`WhlAngF` — Wheel angle
`scalar`

Front wheel angle, in rad.

FxF — Force Input: Total longitudinal force on the front axle
`scalar`

Longitudinal force on the front axle, Fx_F, along vehicle-fixed x-axis, in N.

Bicycle Model - Force Input block input port.

FxR — Force Input: Total longitudinal force on the rear axle
scalar

Longitudinal force on the rear axle, Fx_R, along vehicle-fixed x-axis, in N.

Bicycle Model - Force Input block input port.

xdotin — Velocity Input: Longitudinal velocity
scalar

Vehicle CG velocity along vehicle-fixed *x*-axis, in m/s.

Bicycle Model - Velocity Input block input port.

Output

Info — Bus signal
bus

Bus signal containing these block values.

Signal				Description	Value	Units
InertFrm	Cg	Disp	X	Vehicle CG displacement along the earth-fixed *X*-axis	Computed	m
			Y	Vehicle CG displacement along the earth-fixed *Y*-axis	Computed	m
			Z	Vehicle CG displacement along the earth-fixed *Z*-axis	0	m
		Vel	Xdot	Vehicle CG velocity along the earth-fixed *X*-axis	Computed	m/s
			Ydot	Vehicle CG velocity along the earth-fixed *Y*-axis	Computed	m/s
			Zdot	Vehicle CG velocity along the earth-fixed *Z*-axis	0	m/s
		Ang	phi	Rotation of the vehicle-fixed frame about the earth-fixed *X*-axis (roll)	0	rad
			theta	Rotation of the vehicle-fixed frame about the earth-fixed *Y*-axis (pitch)	0	rad
			psi	Rotation of the vehicle-fixed frame about the earth-fixed *Z*-axis (yaw)	Computed	rad

Signal				Description	Value	Units
	FrntAxl	Disp	X	Front wheel displacement along the earth-fixed *X*-axis	Computed	m
			Y	Front wheel displacement along the earth-fixed *Y*-axis	Computed	m
			Z	Front wheel displacement along the earth-fixed *Z*-axis	0	m
		Vel	Xdot	Front wheel velocity along the earth-fixed *X*-axis	Computed	m/s
			Ydot	Front wheel velocity along the earth-fixed *Y*-axis	Computed	m/s
			Zdot	Front wheel velocity along the earth-fixed *Z*-axis	0	m/s
	RearAxl	Disp	X	Rear wheel displacement along the earth-fixed *X*-axis	Computed	m
			Y	Rear wheel displacement along the earth-fixed *Y*-axis	Computed	m
			Z	Rear wheel displacement along the earth-fixed *Z*-axis	0	m
		Vel	Xdot	Rear wheel velocity along the earth-fixed *X*-axis	Computed	m/s
			Ydot	Rear wheel velocity along the earth-fixed *Y*-axis	Computed	m/s
			Zdot	Rear wheel velocity along the earth-fixed *Z*-axis	0	m/s
	Hitch	Disp	X	Hitch offset from axle plane along the earth-fixed *X*-axis	Computed	m
			Y	Hitch offset from center plane along the earth-fixed *Y*-axis	Computed	m
			Z	Hitch offset from axle plane along the earth-fixed *Z*-axis	Computed	m
		Vel	Xdot	Hitch offset velocity from axle plane along the earth-fixed *X*-axis	Computed	m

Signal				Description	Value	Units
			Ydot	Hitch offset velocity from center plane along the earth-fixed *Y*-axis	Computed	m
			Zdot	Hitch offset velocity from axle plane along the earth-fixed *Z*-axis	Computed	m
	Geom	Disp	X	Vehicle chassis offset from axle plane along the earth-fixed *X*-axis	Computed	m
			Y	Vehicle chassis offset from center plane along the earth-fixed *Y*-axis	Computed	m
			Z	Vehicle chassis offset from axle plane along the earth-fixed *Z*-axis	Computed	m
		Vel	Xdot	Vehicle chassis offset velocity along the earth-fixed *X*-axis	Computed	m/s
			Ydot	Vehicle chassis offset velocity along the earth-fixed *Y*-axis	Computed	m/s
			Zdot	Vehicle chassis offset velocity along the earth-fixed *Z*-axis	Computed	m/s
BdyFrm	Cg	Vel	xdot	Vehicle CG velocity along the vehicle-fixed *x*-axis	Computed	m/s
			ydot	Vehicle CG velocity along the vehicle-fixed *y*-axis	Computed	m/s
			zdot	Vehicle CG velocity along the vehicle-fixed *z*-axis	0	m/s
		Ang	Beta	Body slip angle, β $$\beta = \frac{V_y}{V_x}$$	Computed	rad
		AngVel	p	Vehicle angular velocity about the vehicle-fixed *x*-axis (roll rate)	0	rad/s
			q	Vehicle angular velocity about the vehicle-fixed *y*-axis (pitch rate)	0	rad/s
			r	Vehicle angular velocity about the vehicle-fixed *z*-axis (yaw rate)	Computed	rad/s

Signal				Description	Value	Units
		Acc	ax	Vehicle CG acceleration along the vehicle-fixed *x*-axis	Computed	gn
			ay	Vehicle CG acceleration along the vehicle-fixed *y*-axis	Computed	gn
			az	Vehicle CG acceleration along the vehicle-fixed *z*-axis	0	gn
			xddot	Vehicle CG acceleration along the vehicle-fixed *x*-axis	Computed	m/s^2
			yddot	Vehicle CG acceleration along the vehicle-fixed *y*-axis	Computed	m/s^2
			zddot	Vehicle CG acceleration along the vehicle-fixed *z*-axis	0	m/s^2
		AngAcc	pdot	Vehicle angular acceleration about the vehicle-fixed *x*-axis	0	rad/s
			qdot	Vehicle angular acceleration about the vehicle-fixed *y*-axis	0	rad/s
			rdot	Vehicle angular acceleration about the vehicle-fixed *z*-axis	Computed	rad/s
		DCM	Direction cosine matrix		Computed	rad
	Forces	Body	Fx	Net force on vehicle CG along the vehicle-fixed *x*-axis	Computed	N
			Fy	Net force on vehicle CG along the vehicle-fixed *y*-axis	Computed	N
			Fz	Net force on vehicle CG along the vehicle-fixed *z*-axis	0	N
		Ext	Fx	External force on vehicle CG along the vehicle-fixed *x*-axis	Computed	N
			Fy	External force on vehicle CG along the vehicle-fixed *y*-axis	Computed	N

Signal					Description	Value	Units
			Fz		External force on vehicle CG along the vehicle-fixed *z*-axis	0	N
		Hitch	Fx		Hitch force applied to body at the hitch location along the vehicle-fixed *x*-axis	Input	N
			Fy		Hitch force applied to body at the hitch location along the vehicle-fixed *y*-axis	Input	N
			Fz		Hitch force applied to body at the hitch location along the vehicle-fixed *z*-axis	Input	N
		FrntAxl	Fx		Longitudinal force on front wheel, along the vehicle-fixed *x*-axis	Computed	N
			Fy		Lateral force on front wheel along the vehicle-fixed *y*-axis	Computed	N
			Fz		Normal force on front wheel, along the vehicle-fixed *z*-axis	Computed	N
		RearAxl	Fx		Longitudinal force on rear wheel, along the vehicle-fixed *x*-axis	Computed	N
			Fy		Lateral force on rear wheel along the vehicle-fixed *y*-axis	Computed	N
			Fz		Normal force on rear wheel, along the vehicle-fixed *z*-axis	Computed	N
		Tires	FrntTire	Fx	Front tire force, along the vehicle-fixed *x*-axis	Computed	N
				Fy	Front tire force, along the vehicle-fixed *y*-axis	Computed	N
				Fz	Front tire force, along the vehicle-fixed *z*-axis	Computed	N
			RearTire	FxFx	Rear tire force, along the vehicle-fixed *x*-axis	Computed	N
				Fy	Rear tire force, along the vehicle-fixed *y*-axis	Computed	N

Signal					Description	Value	Units
				Fz	Rear tire force, along the vehicle-fixed *z*-axis	Computed	N
		Drag	Fx		Drag force on vehicle CG along the vehicle-fixed *x*-axis	Computed	N
			Fy		Drag force on vehicle CG along the vehicle-fixed *y*-axis	Computed	N
			Fz		Drag force on vehicle CG along the vehicle-fixed *z*-axis	Computed	N
		Grvty	Fx		Gravity force on vehicle CG along the vehicle-fixed *x*-axis	Computed	N
			Fy		Gravity force on vehicle CG along the vehicle-fixed *y*-axis	Computed	N
			Fz		Gravity force on vehicle CG along the vehicle-fixed *z*-axis	Computed	N
	Moments	Body	Mx		Body moment on vehicle CG about the vehicle-fixed *x*-axis	0	N·m
			My		Body moment on vehicle CG about the vehicle-fixed *y*-axis	Computed	N·m
			Mz		Body moment on vehicle CG about the vehicle-fixed *z*-axis	0	N·m
		Drag	Mx		Drag moment on vehicle CG about the vehicle-fixed *x*-axis	0	N·m
			My		Drag moment on vehicle CG about the vehicle-fixed *y*-axis	Computed	N·m
			Mz		Drag moment on vehicle CG about the vehicle-fixed *z*-axis	0	N·m
		Ext	Mx		External moment on vehicle CG about the vehicle-fixed *x*-axis	0	N·m
			My		External moment on vehicle CG about the vehicle-fixed *y*-axis	Computed	N·m

Signal				Description	Value	Units
			Mz	External moment on vehicle CG about the vehicle-fixed *z*-axis	0	N·m
		Hitch	Mx	Hitch moment at the hitch location about vehicle-fixed *x*-axis	0	N·m
			My	Hitch moment at the hitch location about vehicle-fixed *y*-axis	Computed	N·m
			Mz	Hitch moment at the hitch location about vehicle-fixed *z*-axis	0	N·m
	FrntAxl	Disp	x	Front wheel displacement along the vehicle-fixed *x*-axis	Computed	m
			y	Front wheel displacement along the vehicle-fixed *y*-axis	Computed	m
			z	Front wheel displacement along the vehicle-fixed *z*-axis	Computed	m
		Vel	xdot	Front wheel velocity along the vehicle-fixed *x*-axis	Computed	m/s
			ydot	Front wheel velocity along the vehicle-fixed *y*-axis	Computed	m/s
			zdot	Front wheel velocity along the vehicle-fixed *z*-axis	0	m/s
		Steer	WhlAngFL	Front left wheel steering angle	Computed	rad
			WhlAngFR	Front right wheel steering angle	Computed	rad
	RearAxl	Disp	x	Rear wheel displacement along the vehicle-fixed *x*-axis	Computed	m
			y	Rear wheel displacement along the vehicle-fixed *y*-axis	Computed	m
			z	Rear wheel displacement along the vehicle-fixed *z*-axis	Computed	m
		Vel	xdot	Rear wheel velocity along the vehicle-fixed *x*-axis	Computed	m/s
			ydot	Rear wheel velocity along the vehicle-fixed *y*-axis	Computed	m/s

Signal					Description	Value	Units
			`zdot`		Rear wheel velocity along the vehicle-fixed *z*-axis	`0`	m/s
		`Steer`	`WhlAngRL`		Rear left wheel steering angle	Computed	rad
			`WhlAngRR`		Rear right wheel steering angle	Computed	rad
	`Hitch`	`Disp`		`x`	Hitch offset from axle plane along the vehicle-fixed *x*-axis	Input	m
				`y`	Hitch offset from center plane along the vehicle-fixed *y*-axis	Input	m
				`z`	Hitch offset from axle plane along the earth-fixed *z*-axis	Input	m
		`Vel`		`xdot`	Hitch offset velocity along the vehicle-fixed *x*-axis	Computed	m/s
				`ydot`	Hitch offset velocity along the vehicle-fixed *y*-axis	Computed	m/s
				`zdot`	Hitch offset velocity along the vehicle-fixed *z*-axis	Computed	m/s
	`Pwr`	`Ext`			Applied external power	Computed	W
		`Hitch`			Power loss due to hitch	Computed	W
		`Drag`			Power loss due to drag	Computed	W
	`Geom`	`Disp`		`x`	Vehicle chassis offset from axle plane along the vehicle-fixed *x*-axis	Input	m
				`y`	Vehicle chassis offset from center plane along the vehicle-fixed *y*-axis	Input	m
				`z`	Vehicle chassis offset from axle plane along the earth-fixed *z*-axis	Input	m
		`Vel`		`xdot`	Vehicle chassis offset velocity along the vehicle-fixed *x*-axis	Computed	m/s
				`ydot`	Vehicle chassis offset velocity along the vehicle-fixed *y*-axis	Computed	m/s
				`zdot`	Vehicle chassis offset velocity along the vehicle-fixed *z*-axis	`0`	m/s

Signal				Description	Value	Units
		Ang	Beta	Body slip angle, β $$\beta = \frac{V_y}{V_x}$$	Computed	rad

Signal			Description	Value	Units
PwrInfo	PwrTrnsfrd	PwrFxExt	Externally applied longitudinal force power	Computed	W
		PwrFyExt	Externally applied lateral force power	Computed	W
		PwrMzExt	Externally applied roll moment power	Computed	W
		PwrFwFx	Longitudinal force applied at the front axle power	Computed	W
		PwrFwFy	Lateral force applied at the front axle power	Computed	W
		PwrFwRx	Longitudinal force applied at the rear axle power	Computed	W
		PwrFwRy	Lateral force applied at the rear axle power	Computed	W
	PwrNotTrnsfrd	PwrFxDrag	Longitudinal drag force power	Computed	W
		PwrFyDrag	Lateral drag force power	Computed	W
		PwrMzDrag	Drag pitch moment power	Computed	W
	PwrStored	PwrStoredGrvty	Rate change in gravitational potential energy	Computed	W
		PwrStoredxdot	Rate of change of longitudinal kinetic energy	Computed	W
		PwrStoredydot	Rate of change of lateral kinetic energy	Computed	W
		PwrStoredr	Rate of change of rotational yaw kinetic energy	Computed	W

xdot — Vehicle body longitudinal velocity
scalar

Vehicle CG velocity along vehicle-fixed x-axis, in m/s.

ydot — Vehicle body lateral velocity
scalar

Vehicle CG velocity along vehicle-fixed y-axis, in m/s.

psi — Yaw
scalar

Rotation of the vehicle-fixed frame about earth-fixed Z-axis (yaw), in rad..

r — Yaw rate
scalar

Vehicle angular velocity, r, about the vehicle-fixed z-axis (yaw rate), in rad/s.

Parameters

Longitudinal

Number of wheels on front axle, NF — Front wheel count
2 (default) | scalar

Number of wheels on front axle, N_F. The value is dimensionless.

Number of wheels on rear axle, NR — Rear wheel count
2 (default) | scalar

Number of wheels on rear axle, N_R. The value is dimensionless.

Vehicle mass, m — Vehicle mass
2000 (default) | scalar

Vehicle mass, *m*, in kg.

Longitudinal distance from center of mass to front axle, a — Front axle distance
1.4 (default) | scalar

Horizontal distance *a* from the vehicle CG to the front wheel axle, in m.

Longitudinal distance from center of mass to rear axle, b — Rear axle distance
1.6 (default) | scalar

Horizontal distance *b* from the vehicle CG to the rear wheel axle, in m.

Vertical distance from center of mass to axle plane, h — Height
0.35 (default) | scalar

Height of vehicle CG above the axles, *h*, in m.

Longitudinal distance from center of mass to hitch, dh — Distance from CM to hitch
1 (default) | scalar

Longitudinal distance from center of mass to hitch, *dh*, in m.

Dependencies

To enable this parameter, on the **Input signals** pane, select **Hitch forces** or **Hitch moments**.

Vertical distance from hitch to axle plane, hh — Distance from hitch to axle plane
0.2 (default) | scalar

Vertical distance from hitch to axle plane, *hh*, in m.

Dependencies

To enable this parameter, on the **Input signals** pane, select **Hitch forces** or **Hitch moments**.

`Initial inertial frame longitudinal position, X_o` — Position

`0` (default) | `scalar`

Initial vehicle CG displacement along earth-fixed *X*-axis, in m.

`Initial longitudinal velocity, xdot_o` — Velocity

`0` (default) | `scalar`

Initial vehicle CG velocity along vehicle-fixed *x*-axis, in m/s.

Dependencies

For the Vehicle Body 3DOF Single Track or Vehicle Body 3DOF Dual Track blocks, to enable this parameter, set **Axle forces** to one of these options:

- `External longitudinal forces`
- `External forces`

Lateral

`Front tire corner stiffness, Cy_f` — Stiffness

`12e3` (default) | `scalar`

Front tire corner stiffness, Cy_f, in N/rad.

Dependencies

For the Vehicle Body 3DOF Single Track or Vehicle Body 3DOF Dual Track blocks, to enable this parameter:

1. Set **Axle forces** to one of these options:
   - `External longitudinal velocity`
   - `External longitudinal forces`
2. Clear **Mapped corner stiffness**.

`Rear tire corner stiffness, Cy_r` — Stiffness

`11e3` (default) | `scalar`

Rear tire corner stiffness, Cy_r, in N/rad.

Dependencies

For the Vehicle Body 3DOF Single Track or Vehicle Body 3DOF Dual Track blocks, to enable this parameter:

1. Set **Axle forces** to one of these options:
   - `External longitudinal velocity`
   - `External longitudinal forces`

2 Clear **Mapped corner stiffness**.

Initial inertial frame lateral displacement, Y_o — Position
0 (default) | scalar

Initial vehicle CG displacement along earth-fixed *Y*-axis, in m.

Initial lateral velocity, ydot_o — Velocity
0 (default) | scalar

Initial vehicle CG velocity along vehicle-fixed *y*-axis, in m/s.

Yaw

Yaw polar inertia, Izz — Inertia
4000 (default) | scalar

Yaw polar inertia, in kg*m^2.

Initial yaw angle, psi_o — Psi rotation
0 (default) | scalar

Rotation of the vehicle-fixed frame about earth-fixed *Z*-axis (yaw), in rad.

Initial yaw rate, r_o — Yaw rate
0 (default) | scalar

Vehicle angular velocity about the vehicle-fixed *z*-axis (yaw rate), in rad/s.

Aerodynamic

Longitudinal drag area, Af — Effective vehicle cross-sectional area
2 (default) | scalar

Effective vehicle cross-sectional area, A_f, to calculate the aerodynamic drag force on the vehicle, in m^2.

Longitudinal drag coefficient, Cd — Air drag coefficient
.3 (default) | scalar

Air drag coefficient, C_d. The value is dimensionless.

Longitudinal lift coefficient, Cl — Air lift coefficient
.1 (default) | scalar

Air lift coefficient, C_l. The value is dimensionless.

Longitudinal drag pitch moment, Cpm — Pitch drag
.1 (default) | scalar

Longitudinal drag pitch moment coefficient, C_{pm}. The value is dimensionless.

Relative wind angle vector, beta_w — Wind angle
[0:0.01:0.3] (default) | vector

Relative wind angle vector, β_w, in rad.

Side force coefficient vector, Cs — Side force coefficient
[0:0.03:0.9] (default) | vector

Side force coefficient vector coefficient, C_s. The value is dimensionless.

Yaw moment coefficient vector, Cym — Yaw moment drag
[0:0.01:0.3] (default) | vector

Yaw moment coefficient vector coefficient, C_{ym}. The value is dimensionless.

Environment

Absolute air pressure, Pabs — Pressure
101325 (default) | scalar | scalar

Environmental absolute pressure, P_{abs}, in Pa.

Air temperature, Tair — Temperature
273 (default) | scalar

Environmental absolute temperature, T, in K.

Dependencies

To enable this parameter, clear **Air temperature**.

Gravitational acceleration, g — Gravity
9.81 (default) | scalar

Gravitational acceleration, g, in m/s^2.

Nominal friction scaling factor, mu — Friction scale factor
1 (default) | scalar

Nominal friction scale factor, μ. The value is dimensionless.

Dependencies

For the Vehicle Body 3DOF Single Track or Vehicle Body 3DOF Dual Track blocks, to enable this parameter:

1. Set **Axle forces** to one of these options:
   - `External longitudinal velocity`
   - `External longitudinal forces`
2. Clear **External Friction**.

Simulation

Longitudinal velocity tolerance, xdot_tol — Tolerance
.01 (default) | scalar

Longitudinal velocity tolerance, in m/s.

Nominal normal force, Fznom — Normal force
5000 (default) | scalar

Nominal normal force, in N.

Dependencies

For the Vehicle Body 3DOF Single Track or Vehicle Body 3DOF Dual Track blocks, to enable this parameter, set **Axle forces** to one of these options:

- `External longitudinal velocity`
- `External longitudinal forces`

`Geometric longitudinal offset from axle plane, longOff` — Longitudinal offset
`0` (default) | `scalar`

Vehicle chassis offset from axle plane along body-fixed *x*-axis, in m. When you use the 3D visualization engine, consider using the offset to locate the chassis independent of the vehicle CG.

`Geometric lateral offset from center plane, latOff` — Lateral offset
`0` (default) | `scalar`

Vehicle chassis offset from center plane along body-fixed *y*-axis, in m. When you use the 3D visualization engine, consider using the offset to locate the chassis independent of the vehicle CG.

`Geometric vertical offset from axle plane, vertOff` — Vertical offset
`0` (default) | `scalar`

Vehicle chassis offset from axle plane along body-fixed *z*-axis, in m. When you use the 3D visualization engine, consider using the offset to locate the chassis independent of the vehicle CG.

`Wrap Euler angles, wrapAng` — Selection
`off` (default) | `on`

Wrap the Euler angles to the interval `[-pi, pi]`. For vehicle maneuvers that might undergo vehicle yaw rotations that are outside of the interval, consider deselecting the parameter if you want to:

- Track the total vehicle yaw rotation.
- Avoid discontinuities in the vehicle state estimators.

References

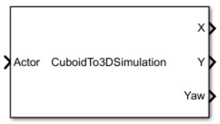
[1] Gillespie, Thomas. *Fundamentals of Vehicle Dynamics*. Warrendale, PA: Society of Automotive Engineers (SAE), 1992.

Introduced in R2018a

Cuboid To 3D Simulation

Convert actor from cuboid coordinates to 3D simulation coordinates

Library: Automated Driving Toolbox / Driving Scenario and Sensor Modeling

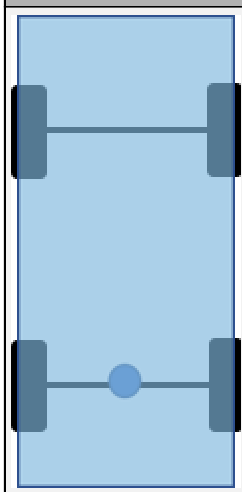


Description

The Cuboid To 3D Simulation block converts a cuboid actor pose in world coordinates to the **X**, **Y**, and **Yaw** coordinates used by the Simulation 3D Vehicle with Ground Following block. Use the converted values to set vehicle positions within the 3D simulation environment for actors created using the **Driving Scenario Designer** app. The ground terrain of the scene determines the roll (*x*-axis rotation), pitch (*y*-axis rotation), and elevation (*z*-axis position) of the vehicle.

You can specify a bus containing a single actor pose or multiple actor poses. By default, the block converts the pose of the first actor in the bus. To specify the actor whose pose you want to convert, specify the `ActorID` of that actor.

In cuboid and 3D simulation driving scenarios, the coordinate systems are the same, but the origins of vehicles differ. In cuboid driving scenarios, the vehicle origin is on the ground, under the center of the rear axle. The block transforms this origin to the origin used in the 3D simulation environment, which is under the geometric center of the vehicle. The table shows the origin difference between the two environments.

Cuboid Vehicle Origin	3D Simulation Vehicle Origin
	

Ports

Input

`Actor` — Cuboid actor pose in world coordinates
Simulink bus containing MATLAB structure

Cuboid actor pose in world coordinates, specified as a Simulink bus containing a MATLAB structure.

To obtain this structure input, use the Scenario Reader block to read actors from a scenario. By default, the Scenario Reader block outputs actors in ego vehicle coordinates. To convert these poses from ego vehicle to world coordinates, use the Vehicle To World block.

The structure in this bus can contain a single actor pose or multiple actor poses.

Single-Pose Structure

To specify a single actor pose, the structure must contain these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [x, y, z]. Units are in meters.
`Velocity`	Velocity (v) of actor in the x-, y-, and z-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (ω) of actor in the x-, y-, and z-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

Multiple-Pose Structure

To specify multiple actor poses, the structure must contain these fields.

Field	Description	Type
`NumActors`	Number of actors	Nonnegative integer
`Time`	Current simulation time	Real-valued scalar
`Actors`	Actor poses	`NumActors`-length array of actor pose structures

Each actor pose structure in `Actors` must have these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
`Velocity`	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (*ω*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

The block converts only one pose from the `Actors` array. To specify which pose to convert, select **Specify Actor ID**, and then specify the `ActorID` of the actor by using the **ActorID used for conversion** parameter.

Output

X — Longitudinal position of actor in 3D simulation coordinates
numeric scalar

Longitudinal position of the actor in 3D simulation coordinates, returned as a numeric scalar. Units are in meters.

In this coordinate system, when looking in the positive direction of the *X*-axis, the positive *Y*-axis points left, and the *Z*-axis points up.

To specify the *X*-position of a vehicle in the 3D simulation environment, connect this port to the **X** input port of a Simulation 3D Vehicle with Ground Following block.

Y — Lateral position of actor in 3D simulation coordinates
numeric scalar

Lateral position of the actor in 3D simulation coordinates, returned as a numeric scalar. Units are in meters.

In this coordinate system, when looking in the positive direction of the *X*-axis, the positive *Y*-axis points left, and the *Z*-axis points up.

To specify the *Y*-position of a vehicle in the 3D simulation environment, connect this port to the **Y** input port of a Simulation 3D Vehicle with Ground Following block.

Yaw — Yaw orientation angle of actor in 3D simulation coordinates
numeric scalar

Yaw orientation angle of the actor about the *Z*-axis in 3D simulation coordinates, returned as a numeric scalar. Units are in degrees.

In this coordinate system, when looking in the positive direction of the *Z*-axis, yaw is clockwise-positive. However, if you view the simulation from a 2D top-down perspective, then yaw is counterclockwise-positive, because you are viewing the scene along the negative *Z*-axis.

To specify the yaw orientation angle of a vehicle in the 3D simulation environment, connect this port to the **Yaw** input port of a Simulation 3D Vehicle with Ground Following block.

Parameters

`Specify Actor ID` — Enable ID specification of cuboid actor

`off` (default) | `on`

Select this parameter to enable the **ActorID used for conversion** parameter, where you can specify the `ActorID` of the cuboid actor pose to convert to 3D simulation coordinates.

If you clear this parameter, then the block converts the first actor pose in the input **Actor** bus.

`ActorID used for conversion` — `ActorID` value of cuboid actor

`1` (default) | positive integer

`ActorID` value of the cuboid actor to convert to 3D simulation coordinates, specified as a positive integer. This parameter must be a valid `ActorID` from the input **Actor** bus.

Dependencies

To enable this parameter, select **Specify Actor ID**.

`Simulate using` — Type of simulation to run

`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.
- `Code generation` — Simulate the model using generated C/C++ code. The first time you run a simulation, Simulink generates C/C++ code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Scenario Reader | Simulation 3D Vehicle with Ground Following | Vehicle To World | World To Vehicle

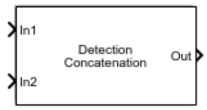
Topics

"Coordinate Systems in Automated Driving Toolbox"
"Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox"

Introduced in R2020a

Detection Concatenation

Combine detection reports from different sensors
Library: Automated Driving Toolbox



Description

The Detection Concatenation block combines detection reports from multiple Radar Detection Generator or Vision Detection Generator blocks onto a single output bus. Concatenation is useful when detections from multiple sensor blocks are passed into a Multi-Object Tracker block. You can accommodate additional sensors by changing the **Number of input sensors to combine** parameter to increase the number of input ports.

Ports

Input

`In1, In2, ..., InN` — Sensor detections to combine

Simulink buses containing MATLAB structures

Sensor detections to combine, where each detection is a Simulink bus containing a MATLAB structure. See “Create Nonvirtual Buses” (Simulink) for more details.

The definitions of the detection lists are found in the **Detections** output port descriptions of the Radar Detection Generator and Vision Detection Generator blocks.

By default, the block includes two ports for input detections. To add more ports, use the **Number of input sensors to combine** parameter.

Output

`Out` — Combined sensor detections

Simulink bus containing MATLAB structure

Combined sensor detections from all input buses, returned as a Simulink bus containing a MATLAB structure. See “Create Nonvirtual Buses” (Simulink). The definitions of the detection lists are found in the **Detections** output port descriptions of the Radar Detection Generator and Vision Detection Generator blocks

The **Maximum number of reported detections** output is the sum of the **Maximum number of reported detections** of all input ports. The number of actual detections is the sum of the number of actual detections in each input port. The `ObjectAttributes` fields in the detection structure are the union of the `ObjectAttributes` fields in each input port.

Parameters

`Number of input sensors to combine` — Number of input sensor ports

2 (default) | positive integer

Number of input sensor ports, specified as a positive integer. Each input port is labeled **In1**, **In2**, …, **InN**, where *N* is the value set by this parameter.

Data Types: `double`

`Source of output bus name` — Source of output bus name

`Auto` (default) | `Property`

Source of output bus name, specified as `Auto` or `Property`.

- If you select `Auto`, the block automatically generates a bus name.
- If you select `Property`, specify the bus name using the **Specify an output bus name** parameter.

`Specify an output bus name` — Name of output bus

no default

Dependencies

To enable this parameter, set the **Source of output bus name** parameter to `Property`.

`Simulate using` — Type of simulation to run

`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.
- `Code generation` — Simulate the model using generated C/C++ code. The first time you run a simulation, Simulink generates C/C++ code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Apps

Bird's-Eye Scope

Blocks

Multi-Object Tracker | Radar Detection Generator | Scenario Reader | Vision Detection Generator

Topics

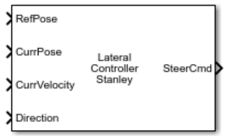
"Create Nonvirtual Buses" (Simulink)

Introduced in R2017b

Lateral Controller Stanley

Control steering angle of vehicle for path following by using Stanley method

Library: Automated Driving Toolbox / Vehicle Control



Description

The Lateral Controller Stanley block computes the steering angle command, in degrees, that adjusts the current pose of a vehicle to match a reference pose, given the vehicle's current velocity and direction. The controller computes this command using the Stanley method [1], whose control law is based on both a kinematic and dynamic bicycle model. To change between models, use the **Vehicle model** parameter.

- The kinematic bicycle model is suitable for path following in low-speed environments such as parking lots, where inertial effects are minimal.
- The dynamic bicycle model is suitable for path following in high-speed environments such as highways, where inertial effects are more pronounced. This vehicle model provides additional parameters that describe the dynamics of the vehicle.

Ports

Input

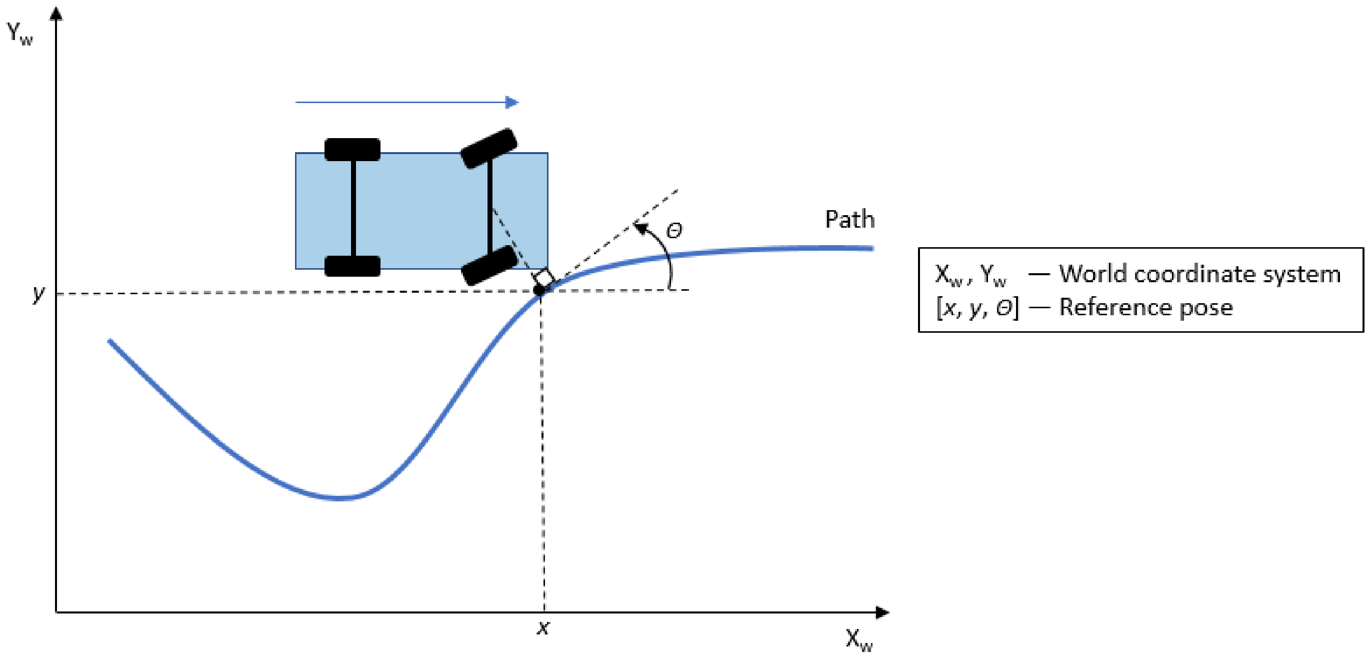
`RefPose` — Reference pose

[x, y, Θ] vector

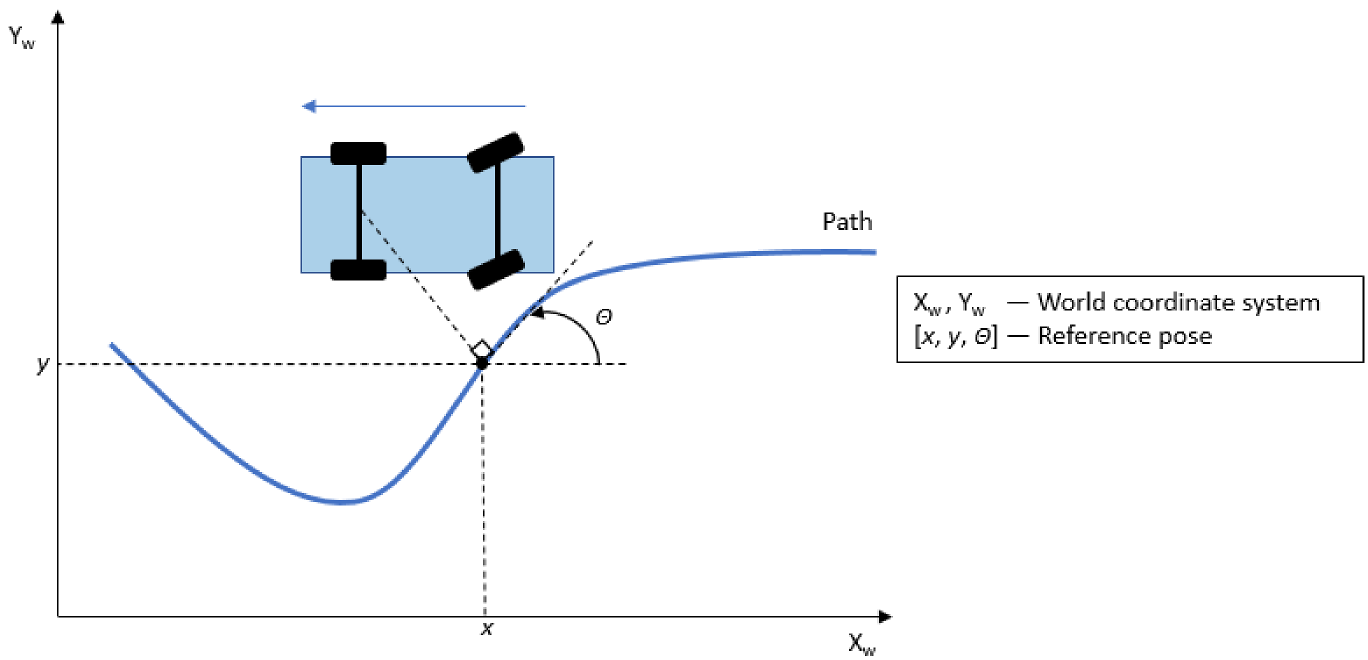
Reference pose, specified as an [x, y, Θ] vector. x and y are in meters, and Θ is in degrees.

x and y specify the reference point to steer the vehicle toward. Θ specifies the orientation angle of the path at this reference point and is positive in the counterclockwise direction.

- For a vehicle in forward motion, the reference point is the point on the path that is closest to the center of the vehicle's front axle.



- For a vehicle in reverse motion, the reference point is the point on the path that is closest to the center of the vehicle's rear axle.



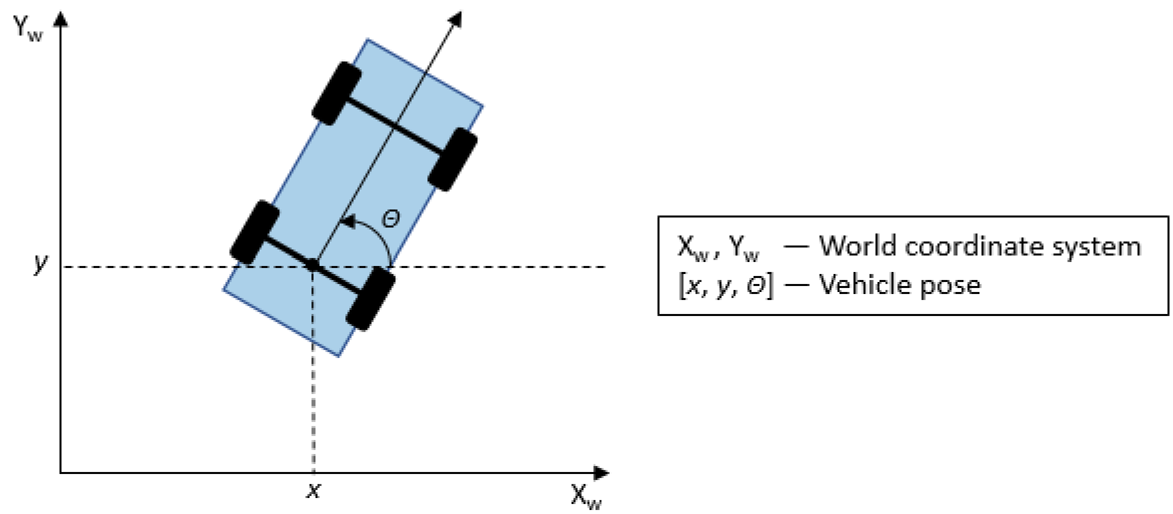
Data Types: `single` | `double`

`CurrPose` — Current pose
[*x*, *y*, *Θ*] vector

Current pose of the vehicle, specified as an [*x*, *y*, *Θ*] vector. *x* and *y* are in meters, and *Θ* is in degrees.

x and *y* specify the location of the vehicle, which is defined as the center of the vehicle's rear axle.

Θ specifies the orientation angle of the vehicle at location (*x*,*y*) and is positive in the counterclockwise direction.



For more details on vehicle pose, see “Coordinate Systems in Automated Driving Toolbox”.

Data Types: `single` | `double`

`CurrVelocity` — Current longitudinal velocity
real scalar

Current longitudinal velocity of the vehicle, specified as a real scalar. Units are in meters per second.

- If the vehicle is in forward motion, then this value must be greater than 0.
- If the vehicle is in reverse motion, then this value must be less than 0.
- A value of 0 represents a vehicle that is not in motion.

Data Types: `single` | `double`

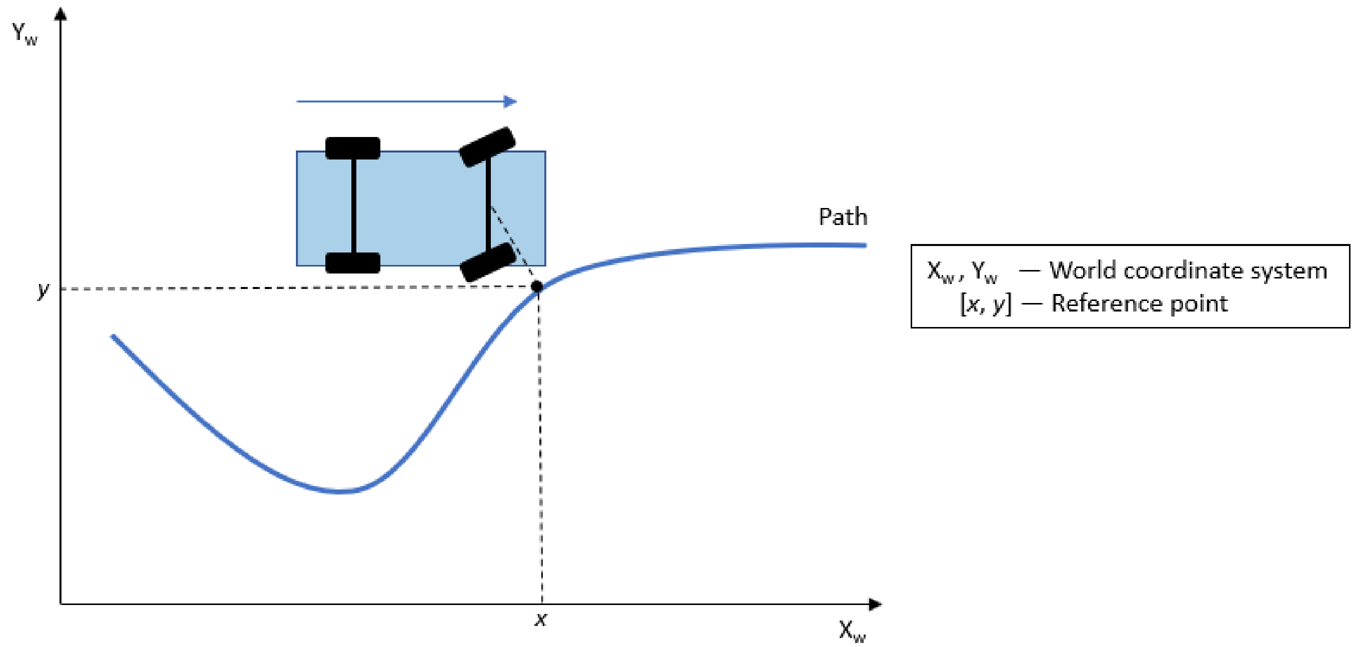
`Direction` — Driving direction of vehicle
`1` (forward motion) | `-1` (reverse motion)

Driving direction of the vehicle, specified as `1` for forward motion or `-1` for reverse motion. The driving direction determines the position error and angle error used to compute the steering angle command. For more details, see “Algorithms” on page 2-31.

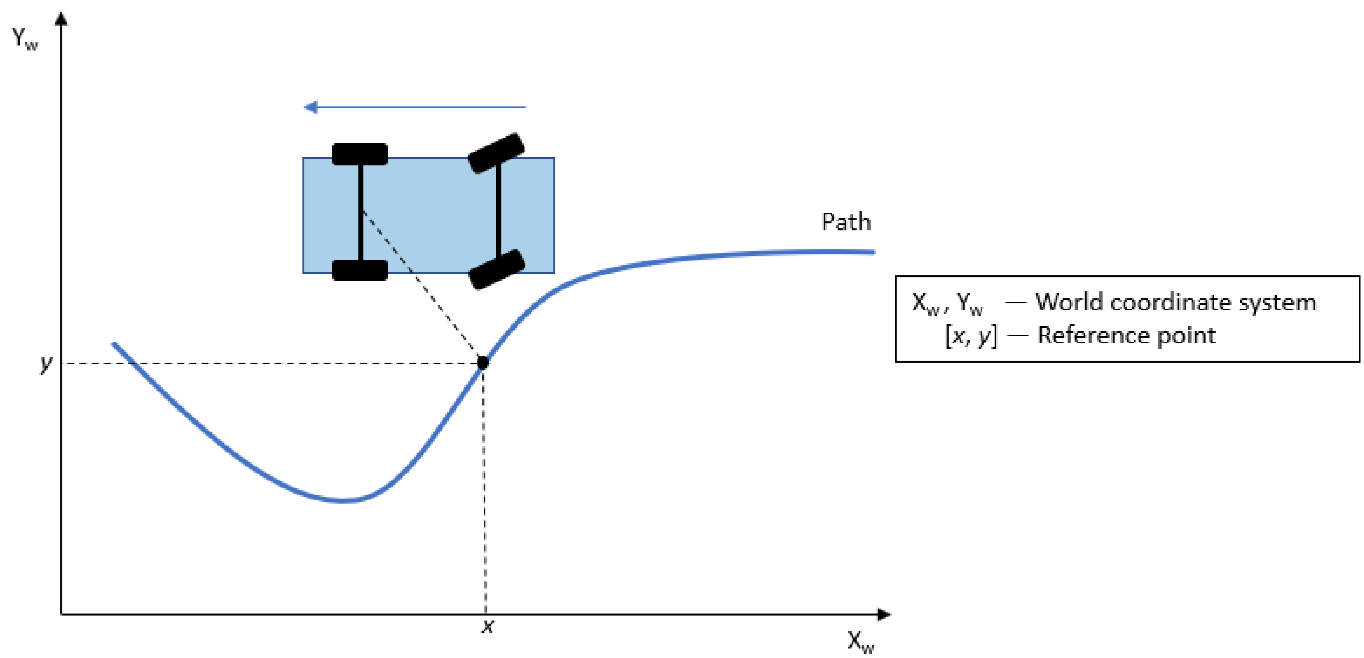
`Curvature` — Curvature of path
real scalar

Curvature of the path at the reference point, in radians per meter, specified as a real scalar.

- For a vehicle in forward motion, the reference point is the point on the path that is closest to the center of the vehicle's front axle.



- For a vehicle in reverse motion, the reference point is the point on the path that is closest to the center of the vehicle's rear axle.



You can obtain the curvature of a path from the **Curvatures** output port of a Path Smoother Spline block. You can also obtain curvatures of lane boundaries from the output lane boundary structures of a Scenario Reader block.

Dependencies

To enable this port, set **Vehicle model** to `Dynamic bicycle model`.

CurrYawRate — Current yaw rate
real scalar

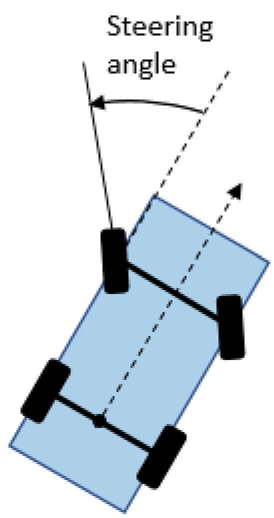
Current yaw rate of the vehicle, in degrees per second, specified as a real scalar. The current yaw rate is the rate of change in the angular velocity of the vehicle.

Dependencies

To enable this port, set **Vehicle model** to `Dynamic bicycle model`.

CurrSteer — Current steering angle
real scalar

Current steering angle of the vehicle, in degrees, specified as a real scalar. This value is positive in the counterclockwise direction.



For more details, see "Coordinate Systems in Automated Driving Toolbox".

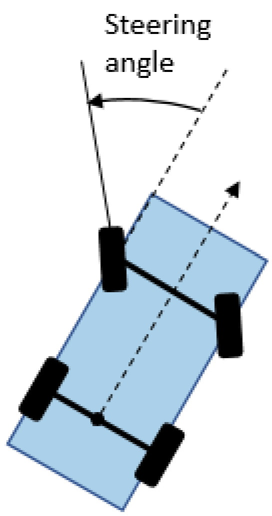
Dependencies

To enable this port, set **Vehicle model** to `Dynamic bicycle model`.

Output

SteerCmd — Steering angle command
real scalar

Steering angle command, in degrees, returned as a real scalar. This value is positive in the counterclockwise direction.



For more details, see “Coordinate Systems in Automated Driving Toolbox”.

Parameters

`Vehicle model` — Vehicle model
`Kinematic bicycle model` (default) | `Dynamic bicycle model`

Select the type of vehicle model to set the Stanley method control law used by the block.

- `Kinematic bicycle model` — Kinematic bicycle model for path following in low-speed environments such as parking lots, where inertial effects are minimal
- `Dynamic bicycle model` — Dynamic bicycle model for path following in high-speed environments such as highways, where inertial effects are more pronounced

`Position gain of forward motion` — Position gain of vehicle in forward motion
`2.5` (default) | positive real scalar

Position gain of the vehicle when it is in forward motion, specified as a positive scalar. This value determines how much the position error affects the steering angle. Typical values are in the range [1, 5]. Increase this value to increase the magnitude of the steering angle.

`Position gain of reverse motion` — Position gain of vehicle in reverse motion
`2.5` (default) | positive real scalar

Position gain of the vehicle when it is in reverse motion, specified as a positive scalar. This value determines how much the position error affects the steering angle. Typical values are in the range [1, 5]. Increase this value to increase the magnitude of the steering angle.

`Yaw rate feedback gain` — Yaw rate feedback gain
`2.5` (default) | nonnegative real scalar

Yaw rate feedback gain, specified as a nonnegative real scalar. This value determines how much weight is given to the current yaw rate of the vehicle when the block computes the steering angle command.

Dependencies

To enable this parameter, set **Vehicle model** to `Dynamic bicycle model`.

`Steering angle feedback gain` — Steering angle feedback gain
`2.5` (default) | nonnegative real scalar

Steering angle feedback gain, specified as a nonnegative real scalar. This value determines how much the difference between the current steering angle command, **SteerCmd**, and the current steering angle, **CurrSteer**, affects the next steering angle command.

Dependencies

To enable this parameter, set **Vehicle model** to `Dynamic bicycle model`.

`Wheelbase of vehicle (m)` — Distance between front and rear axle
`2.8` (default) | real scalar

Distance between the front and rear axle of the vehicle, in meters, specified as a real scalar. This value applies only when the vehicle is in forward motion, that is, when the **Direction** input port is `1`.

Dependencies

To enable this parameter, set **Vehicle model** to `Kinematic bicycle model`.

`Vehicle mass (kg)` — Vehicle mass
`1575` (default) | positive real scalar

Vehicle mass, in kilograms, specified as a positive real scalar.

Dependencies

To enable this parameter, set **Vehicle model** to `Dynamic bicycle model`.

`Longitudinal distance from center of mass to front axle (m)` — Distance to front axle
`1.2` (default) | positive real scalar

Longitudinal distance from the vehicle's center of mass to its front wheel axle, in meters, specified as a positive real scalar.

Dependencies

To enable this parameter, set **Vehicle model** to `Dynamic bicycle model`.

`Longitudinal distance from center of mass to rear axle (m)` — Distance to rear axle
`1.6` (default) | positive real scalar

Longitudinal distance from the vehicle's center of mass to its rear wheel axle, in meters, specified as a positive real scalar.

Dependencies

To enable this parameter, set **Vehicle model** to `Dynamic bicycle model`.

`Front tire corner stiffness (N/rad)` — Cornering stiffness of front tires
`19000` (default) | positive real scalar

Cornering stiffness of front tires, in Newtons per radian, specified as a positive real scalar.

Dependencies

To enable this parameter, set **Vehicle model** to `Dynamic bicycle model`.

`Maximum steering angle (deg)` — Maximum allowed steering angle

`35` (default) | real scalar in the range (0, 180)

Maximum allowed steering angle of the vehicle, in degrees, specified as a real scalar in the range (0, 180).

The output from the **SteerCmd** port is saturated to the range [–*M*, *M*], where *M* is the value of the **Maximum steering angle (deg)** parameter.

- Values below –*M* are set to –*M*.
- Values above *M* are set to *M*.

Tips

- You can switch between bicycle models as the vehicle environment changes. Add two Lateral Controller Stanley blocks to a variant subsystem and specify a different bicycle model for each block. For an example, see “Lateral Control Tutorial”.

Algorithms

To compute the steering angle command, the controller minimizes the position error and the angle error of the current pose with respect to the reference pose. The driving direction of the vehicle determines these error values.

When the vehicle is in forward motion (**Direction** parameter is `1`):

- The position error is the lateral distance from the center of the front axle to the reference point on the path.
- The angle error is the angle of the front wheel with respect to reference path.

When the vehicle is in reverse motion (**Direction** parameter is `-1`):

- The position error is the lateral distance from the center of the rear axle to the reference point on the path.
- The angle error is the angle of the rear wheel with respect to reference path.

For details on how the controller minimizes these errors for kinematic and dynamic bicycle models, see [1].

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. doi:10.1109/ACC.2007.4282788

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Longitudinal Controller Stanley | Path Smoother Spline | Velocity Profiler

Functions

`lateralControllerStanley`

Objects

`pathPlannerRRT`

Topics

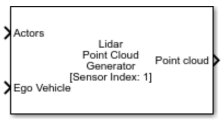
“Coordinate Systems in Automated Driving Toolbox”

Introduced in R2018b

Lidar Point Cloud Generator

Generate lidar point cloud data for driving scenario

Library: Automated Driving Toolbox / Driving Scenario and Sensor Modeling



Description

The Lidar Point Cloud Generator block generates a point cloud from lidar measurements taken by a lidar sensor mounted on an ego vehicle.

The block derives the point cloud from simulated roads and actor poses in a driving scenario and generates the point cloud at intervals equal to the sensor update interval. By default, detections are referenced to the coordinate system of the ego vehicle. The block can simulate added noise at a specified range accuracy by using a statistical model. The block also provides parameters to exclude the ego vehicle and roads from the generated point cloud.

The lidar generates point cloud data based on the mesh representations of the roads and actors in the scenario. A mesh is a 3-D geometry of an object that is composed of faces and vertices.

When building scenarios and sensor models using the **Driving Scenario Designer** app, the lidar sensors exported to Simulink are output as Lidar Point Cloud Generator blocks.

Limitations

- C/C++ code generation is not supported.
- For Each subsystems are not supported.
- Rapid acceleration mode is not supported.
- Use of the Detection Concatenation block with this block is not supported. You cannot concatenate point cloud data with detections from other sensors.
- If a model does not contain a Scenario Reader block, then this block does not include roads in the generated point cloud.
- Point cloud data is not generated for lane markings.

Ports

Input

`Actors` — Scenario actor poses

Simulink bus containing MATLAB structure

Scenario actor poses in ego vehicle coordinates, specified as a Simulink bus containing a MATLAB structure.

The structure must contain these fields.

Field	Description	Type
`NumActors`	Number of actors	Nonnegative integer
`Time`	Current simulation time	Real-valued scalar
`Actors`	Actor poses	`NumActors`-length array of actor pose structures

Each actor pose structure in `Actors` must have these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [x, y, z]. Units are in meters.
`Velocity`	Velocity (v) of actor in the x-, y-, and z-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (ω) of actor in the x-, y-, and z-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

`Ego Vehicle` — Ego vehicle pose
Simulink bus containing MATLAB structure

Ego vehicle pose, specified as a Simulink bus containing a MATLAB structure.

The structure must have these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [x, y, z]. Units are in meters.
`Velocity`	Velocity (v) of actor in the x-, y-, and z-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.

Field	Description
Yaw	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
AngularVelocity	Angular velocity (ω) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

You can output the ego vehicle pose from a Scenario Reader block. In the Scenario Reader block used in your model, select the **Output ego vehicle pose** parameter.

Output

Point Cloud — Point cloud data
m-by-*n*-by-3 array of positive real-valued [*x*, *y*, *z*] points

Point cloud data, returned as an *m*-by-*n*-by 3 array of positive real-valued [*x*, *y*, *z*] points. *m* is the number of elevation (vertical) channels in the point cloud. *n* is the number of azimuthal (horizontal) channels in the point cloud. *m* and *n* define the number of points in the point cloud, as shown in this equation:

$$m \times n = \frac{V_{FOV}}{V_{RES}} \times \frac{H_{FOV}}{H_{RES}}$$

- V_{FOV} is the vertical field of view of the lidar, in degrees, as specified by the **Elevation limits of lidar (deg)** parameter.
- V_{RES} is the vertical angular resolution of the lidar, in degrees, as specified by the **Elevation resolution of lidar (deg)** parameter.
- H_{FOV} is the horizontal field of view of the lidar, in degrees, as specified by the **Azimuthal limits of lidar (deg)** parameter.
- H_{RES} is the horizontal angular resolution of the lidar, in degrees, as specified by the **Azimuthal resolution of lidar (deg)** parameter.

Each *m*-by-*n* entry in the array specifies the *x*-, *y*-, and *z*-coordinates of a detected point in the ego vehicle coordinate system. If the lidar does not detect a point at a given coordinate, then *x*, *y*, and *z* are returned as `NaN`.

By default, the Lidar Point Cloud Generator block includes road data in the generated point cloud. The block obtains the road data in world coordinates from a Scenario Reader block that is in the same model as the Lidar Point Cloud Generator block. The Lidar Point Cloud Generator block computes the road mesh in ego vehicle coordinates based on the road data and the ego vehicle pose at the **Ego Vehicle** input port. The **Maximum detection range (m)** parameter of the Lidar Point Cloud Generator block determines the extent of the road mesh. To exclude road data from the point cloud, clear the **Include roads in generated point cloud** parameter.

Parameters

Parameters

Sensor Identification

Unique identifier of sensor — Unique sensor identifier
1 (default) | positive integer

Unique sensor identifier, specified as a positive integer. The sensor identifier distinguishes detections that come from different sensors in a multisensor system. If a model contains multiple sensor blocks that have the same sensor identifier, the **Bird's-Eye Scope** displays an error.

`Required interval between sensor updates (s)` — Required time interval between sensor updates
`0.1` (default) | positive scalar

Required time interval between sensor updates, specified as a positive scalar. The value of this parameter must be an integer multiple of the **Actors** input port data interval. Updates requested from the sensor between update intervals contain no detections. Units are in seconds.

Sensor Extrinsics

`Sensor's (x,y) position (m)` — Location of center of lidar sensor
`[1.5 0]` (default) | real-valued 1-by-2 vector

Location of the center of the lidar sensor, specified as a real-valued 1-by-2 vector. The **Sensor's (x,y) position (m)** and **Sensor's height (m)** parameters define the coordinates of the lidar sensor with respect to the ego vehicle coordinate system. The default value corresponds to a lidar sensor mounted on a sedan, at the center of the roof's front edge. Units are in meters.

`Sensor's height (m)` — Height of lidar sensor
`1.6` (default) | positive scalar

Height of the lidar sensor above the ground plane, specified as a positive scalar. The **Sensor's (x,y) position (m)** and **Sensor's height (m)** parameters define the coordinates of the lidar sensor with respect to the ego vehicle coordinate system. The default value corresponds to a lidar sensor mounted on a sedan, at the center of the roof front edge. Units are in meters.

`Yaw angle of sensor mounted on ego vehicle (deg)` — Yaw angle of lidar sensor
`0` (default) | real-valued scalar

Yaw angle of the lidar sensor, specified as a real-valued scalar. The yaw angle is the angle between the center line of the ego vehicle and the downrange axis of the lidar sensor. A positive yaw angle corresponds to a clockwise rotation when you look in the positive direction of the *z*-axis of the ego vehicle coordinate system. Units are in degrees.

`Pitch angle of sensor mounted on ego vehicle (deg)` — Pitch angle of lidar sensor
`0` (default) | real-valued scalar

Pitch angle of the lidar sensor, specified as a real-valued scalar. The pitch angle is the angle between the downrange axis of the lidar sensor and the *xy*-plane of the ego vehicle coordinate system. A positive pitch angle corresponds to a clockwise rotation when you look in the positive direction of the *y*-axis of the ego vehicle coordinate system. Units are in degrees.

`Roll angle of sensor mounted on ego vehicle (deg)` — Roll angle of lidar sensor
`0` (default) | real-valued scalar

Roll angle of the lidar sensor, specified as a real-valued scalar. The roll angle is the angle of rotation of the downrange axis of the lidar sensor around the *x*-axis of the ego vehicle coordinate system. A positive roll angle corresponds to a clockwise rotation when you look in the positive direction of the *x*-axis of the ego vehicle coordinate system. Units are in degrees.

Point Cloud Reporting

`Coordinate system used to report point cloud` — Coordinate system of reported detections
`Ego Cartesian` (default) | `Sensor Cartesian`

Coordinate system of reported detections, specified as one of these values:

- `Ego Cartesian` — Detections are reported in the ego vehicle Cartesian coordinate system.
- `Sensor Cartesian` — Detections are reported in the sensor Cartesian coordinate system.

`Include ego vehicle in generated point cloud` — Include ego vehicle in point cloud
`on` (default) | `off`

Select this parameter to include the ego vehicle in the generated point cloud.

`ActorID of ego vehicle` — `ActorID` value of ego vehicle
`1` (default) | positive integer

`ActorID` value of the ego vehicle, specified as a positive integer. `ActorID` is the unique identifier for an actor. This parameter must be a valid `ActorID` from the input **Actor** bus.

Dependencies

To enable this parameter, select the **Include ego vehicle in generated point cloud** parameter.

`Include roads in generated point cloud` — Include roads in point cloud
`on` (default) | `off`

Select this parameter to include the roads in the generated point cloud.

`Source of actor profiles` — Source of actor profiles
`From Scenario Reader block` (default) | `From workspace`

Source of actor profiles, which are the physical and radar characteristics of all actors in the driving scenario, specified as one of these options:

- `From Scenario Reader block` — The block obtains the actor profiles from the scenario specified by the Scenario Reader block.
- `From workspace` — The block obtains the actor profiles from the MATLAB or model workspace variable specified by the **MATLAB or model workspace variable name** parameter.

`MATLAB or model workspace variable name` — Variable name of actor profiles
`actor_profiles` (default) | valid variable name

Variable name of actor profiles, specified as the name of a MATLAB or model workspace variable containing actor profiles.

Actor profiles are the physical and radar characteristics of all actors in a driving scenario and are specified as a structure or structure array.

- If the actor profiles variable contains a single structure, then all actors specified in the input **Actors** bus use this profile.
- If the actor profiles variable is a structure array, then each actor specified in the input **Actors** bus must have a unique actor profile.

To generate an array of structures for your driving scenario, use the `actorProfiles` function. The table shows the valid structure fields. If you do not specify a field, the fields are set to their default values.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`ClassID`	Classification identifier, specified as a nonnegative integer. `0` represents an object of an unknown or unassigned class.
`Length`	Length of actor, specified as a positive real-valued scalar. Units are in meters.
`Width`	Width of actor, specified as a positive real-valued scalar. Units are in meters.
`Height`	Height of actor, specified as a positive real-valued scalar. Units are in meters.
`OriginOffset`	Offset of actor's rotational center from its geometric center, specified as a real-valued vector of the form [*x*, *y*, *z*]. The rotational center, or origin, is located at the bottom center of the actor. For vehicles, the rotational center is the point on the ground beneath the center of the rear axle. Units are in meters.
`MeshVertices`	Mesh vertices of actor, specified as an *n*-by-*3* real-valued matrix of vertices. Each row in the matrix defines a point in 3-D space.
`MeshFaces`	Mesh faces of actor, specified as an *m*-by-*3* matrix of integers. Each row of `MeshFaces` represents a triangle defined by the vertex IDs, which are the row numbers of vertices.
`RCSPattern`	Radar cross-section (RCS) pattern of actor, specified as a `numel(RCSElevationAngles)`-by-`numel(RCSAzimuthAngles)` real-valued matrix. Units are in decibels per square meter.
`RCSAzimuthAngles`	Azimuth angles corresponding to rows of `RCSPattern`, specified as a vector of values in the range [–180, 180]. Units are in degrees.
`RCSElevationAngles`	Elevation angles corresponding to rows of `RCSPattern`, specified as a vector of values in the range [–90, 90]. Units are in degrees.

For complete definitions of the structure fields, see the `actor` and `vehicle` functions.

Dependencies

To enable this parameter, set the **Source of actor profiles** parameter to `From workspace`.

Measurements

Settings

`Maximum detection range (m)` — Maximum detection range
`120` (default) | positive scalar

Maximum detection range of the lidar sensor, specified as a positive scalar. The sensor cannot detect actors beyond this range. This parameter also determines the extent of the road mesh. Units are in meters.

`Range accuracy (m)` — Accuracy of range measurements
`0.002` (default) | positive scalar

Accuracy of range measurements, specified as a positive scalar. Units are in meters.

`Azimuthal resolution of lidar (deg)` — Azimuthal resolution of lidar sensor
`0.16` (default) | positive scalar

Azimuthal resolution of the lidar sensor, specified as a positive scalar. The azimuthal resolution defines the minimum separation in azimuth angle at which the lidar can distinguish between two targets. Units are in degrees.

`Elevation resolution of lidar (deg)` — Elevation resolution of lidar sensor
`1.25` (default) | positive scalar

Elevation resolution of the lidar sensor, specified as a positive scalar. The elevation resolution defines the minimum separation in elevation angle at which the lidar can distinguish between two targets. Units are in degrees.

`Azimuthal limits of lidar (deg)` — Azimuthal limits of lidar sensor
`[-180 180]` (default) | 1-by-2 real-valued vector of form [*min*, *max*]

Azimuthal limits of the lidar sensor, specified as a 1-by-2 real-valued vector of the form [*min*, *max*]. Units are in degrees.

`Elevation limits of lidar (deg)` — Elevation limits of lidar sensor
`[-20 20]` (default) | 1-by-2 real-valued vector of form [*min*, *max*]

Elevation limits of the lidar sensor, specified as a 1-by-2 real-valued vector of the form [*min*, *max*]. Units are in degrees.

`Add noise to measurements` — Add noise to measurements
`on` (default) | `off`

Select this parameter to add noise to lidar sensor measurements. When you clear this parameter, the measurements have no noise.

See Also

Apps
Bird's-Eye Scope

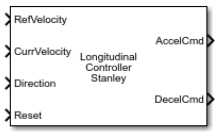
Blocks
Radar Detection Generator | Scenario Reader | Simulation 3D Lidar | Vision Detection Generator

Objects
`lidarPointCloudGenerator`

Introduced in R2020b

Longitudinal Controller Stanley

Control longitudinal velocity of vehicle by using Stanley method
Library: Automated Driving Toolbox / Vehicle Control



Description

The Longitudinal Controller Stanley block computes the acceleration and deceleration commands, in meters per second, that control the velocity of the vehicle. Specify the reference velocity, current velocity, and current driving direction. The controller computes these commands using the Stanley method [1], which the block implements as a discrete proportional-integral (PI) controller with integral anti-windup. For more details, see “Algorithms” on page 2-42.

You can also compute the steering angle command of a vehicle using the Stanley method. See the Lateral Controller Stanley block.

Ports

Input

`RefVelocity` — Reference velocity
real scalar

Reference velocity, in meters per second, specified as a real scalar.

`CurrVelocity` — Current velocity
real scalar

Current velocity of the vehicle, in meters per second, specified as a real scalar.

`Direction` — Driving direction
`1` (forward motion) | `-1` (reverse motion)

Driving direction of vehicle, specified as `1` for forward motion and `-1` for reverse motion.

`Reset` — Trigger to reset integral of velocity error
`0` (hold steady) | nonzero scalar (reset)

Trigger to reset the integral of velocity error, $e(k)$, to zero. A value of `0` holds $e(k)$ steady. A nonzero value resets $e(k)$.

Output

`AccelCmd` — Acceleration command
real scalar in the range [0, M_A]

Acceleration command, returned as a real scalar in the range [0, M_A], where M_A is the value of the **Maximum longitudinal acceleration (m/s^2)** parameter.

DecelCmd — Deceleration command
real scalar in the range [0, M_D]

Deceleration command, returned as a real scalar in the range [0, M_D], where M_D is the value of the **Maximum longitudinal deceleration (m/s^2)** parameter.

Parameters

Proportional gain, Kp — Proportional gain
2.5 (default) | positive real scalar

Proportional gain of controller, K_p, specified as a positive real scalar.

Integral gain, Ki — Integral gain
1 (default) | positive real scalar

Integral gain of controller, K_i, specified as a positive real scalar.

Sample time (s) — Sample time
0.05 (default) | positive real scalar

Sample time of controller, in seconds, specified as a positive real scalar.

Maximum longitudinal acceleration (m/s^2) — Maximum longitudinal acceleration
3 (default) | positive real scalar

Maximum longitudinal acceleration, in meters per second squared, specified as a positive real scalar.

The block saturates the output from the **AccelCmd** to the range [0, M_A], where M_A is the value of this parameter. Values above M_A are set to M_A.

Maximum longitudinal deceleration (m/s^2) — Maximum longitudinal deceleration
6 (default) | positive real scalar

Maximum longitudinal deceleration, in meters per second squared, specified as a positive real scalar.

The block saturates the output from the **DecelCmd** port to the range [0, M_D], where M_D is the value of this parameter. Values above M_D are set to M_D.

Algorithms

The Longitudinal Controller Stanley block implements a discrete proportional-integral (PI) controller with integral anti-windup, as described by the "Anti-windup method" (Simulink) parameter of the PID Controller block. The block uses this equation:

$$u(k) = (K_p + K_i \frac{T_s z}{z - 1})\, e(k)$$

- $u(k)$ is the control signal at the kth time step.
- K_p is the proportional gain, as set by the **Proportional gain, Kp** parameter.
- K_i is the integral gain, as set by the **Integral gain, Ki** parameter.
- T_s is the sample time of the block in seconds, as set by the **Sample time (s)** parameter.

- $e(k)$ is the velocity error (**CurrVelocity** – **RefVelocity**) at the kth time step. For each k, this error is equal to the difference between the current velocity and reference velocity inputs (**CurrVelocity** – **RefVelocity**).

The control signal, u, determines the value of acceleration command **AccelCmd** and deceleration command **DecelCmd**. The block saturates the acceleration and deceleration commands to respective ranges of $[0, M_A]$ and $[0, M_D]$, where:

- M_A is value of the **Maximum longitudinal acceleration (m/s^2)** parameter.
- M_D is the value of the **Maximum longitudinal deceleration (m/s^2)** parameter.

At each time step, only one of the **AccelCmd** and **DecelCmd** port values is positive, and the other port value is `0`. In other words, the vehicle can either accelerate or decelerate in one time step, but it cannot do both at one time.

The direction of motion, as specified in the **Direction** input port, determines which command is positive at the given time step.

Direction Port Value	Control Signal Value $u(k)$	AccelCmd Port Value	DecelCmd Port Value	Description
`1` (forward motion)	$u(k) > 0$	positive real scalar	`0`	Vehicle speeds up as it travels forward
	$u(k) < 0$	`0`	positive real scalar	Vehicle slows down as it travels forward
`-1` (reverse motion)	$u(k) > 0$	`0`	positive real scalar	Vehicle slows down as it travels in reverse
	$u(k) < 0$	positive real scalar	`0`	Vehicle speeds up as it travels in reverse

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*. August 2007, pp. 2296–2301. doi:10.1109/ACC.2007.4282788.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

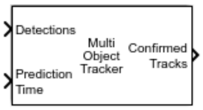
Blocks

Lateral Controller Stanley | PID Controller | Path Smoother Spline | Velocity Profiler

Introduced in R2019a

Multi-Object Tracker

Create and manage tracks of multiple objects
Library: Automated Driving Toolbox



Description

The Multi-Object Tracker block initializes, confirms, predicts, corrects, and deletes the tracks of moving objects. Inputs to the multi-object tracker are detection reports generated by Radar Detection Generator and Vision Detection Generator blocks. The multi-object tracker accepts detections from multiple sensors and assigns them to tracks using a global nearest neighbor (GNN) criterion. Each detection is assigned to a separate track. If the detection cannot be assigned to any track, the multi-object tracker creates a new track.

A new track starts in a *tentative* state. If enough detections are assigned to a tentative track, its status changes to *confirmed*. When a track is confirmed, the multi-object tracker considers that track to represent a physical object. If detections are not added to the track within a specifiable number of updates, the track is deleted.

The multi-object tracker also estimates the state vector and state vector covariance matrix for each track using a Kalman filter. These state vectors are used to predict a track's location in each frame and determine the likelihood of each detection being assigned to each track.

Ports

Input

`Detections` — Detection list

Simulink bus containing MATLAB structure

Detection list, specified as a Simulink bus containing a MATLAB structure. See “Group Signal Lines into Virtual Buses” (Simulink). The structure has the form:

Field	Description	Type
`NumDetections`	Number of detections	integer
`IsValidTime`	False when updates are requested at times that are between block invocation intervals	Boolean
`Detections`	Object detections	Array of object detection structures. The first `NumDetections` of these detections are actual detections.

The definitions of the object detection structures are found in the **Detections** output port descriptions of the Radar Detection Generator and Vision Detection Generator blocks.

Note The object detection structure contains a `Time` field. The time tag of each object detection must be less than or equal to the time of the current invocation of the block. The time tag must also be greater than the update time specified in the previous invocation of the block.

`Prediction Time` — Track update time

real scalar

Track update time, specified as a real scalar. The multi-object tracker updates all tracks to this time. Update time must always increase with each invocation of the block. Units are in seconds.

Note The object detection structure contains a `Time` field. The time tag of each object detection must be less than or equal to the time of the current invocation of the block. The time tag must also be greater than the update time in the previous invocation of the block.

Dependencies

To enable this port, set **Prediction time source** to `Input port`.

`Cost Matrix` — Cost matrix

real-valued N_t-by-N_d matrix

Cost matrix, specified as a real-valued N_t-by-N_d matrix, where N_t is the number of existing tracks and N_d is the number of current detections.

The rows of the cost matrix correspond to the existing tracks. The columns correspond to the detections. Tracks are ordered as they appear in the list of tracks in the **All Tracks** output port of the previous invocation of the block.

In the first update to the multi-object tracker, or if the track has no previous tracks, assign the cost matrix a size of $[0, N_d]$. The cost must be calculated so that lower costs indicate a higher likelihood that the multi-object tracker assigns a detection to a track. To prevent certain detections from being assigned to certain tracks, use `Inf`.

Dependencies

To enable this port, select **Enable cost matrix input**.

`Detectable Track IDs` — Detectable track IDs

real-valued *M*-by-1 vector | real-valued *M*-by-2 matrix

Detectable track IDs, specified as a real-valued *M*-by-1 vector or *M*-by-2 matrix. Detectable tracks are tracks that the sensors expect to detect. The first column of the matrix contains a list of track IDs that the sensors report as detectable. The optional second column contains the detection probability for the track.

Tracks whose identifiers are not included in **Detectable Track IDs** are considered undetectable. The track deletion logic does not count the lack of detection as a "missed detection" for track deletion purposes.

If this port is not enabled, the tracker assumes all tracks to be detectable at each invocation of the block.

Dependencies

To enable this port, in the **Port Setting** tab, select **Enable detectable track IDs Input**.

Output

`Confirmed Tracks` — Confirmed tracks

Simulink bus containing MATLAB structure

Confirmed tracks, returned as a Simulink bus containing a MATLAB structure. See “Create Nonvirtual Buses” (Simulink).

This table shows the structure fields.

Field	Description
`NumTracks`	Number of tracks
`Tracks`	Array of track structures of a length set by the **Maximum number of tracks** parameter. Only the first `NumTracks` of these are actual tracks.

This table shows the fields of each track structure.

Field	Description
`TrackID`	Unique integer that identifies the track.
`SourceIndex`	Unique identifier the tracker in a multiple tracker environment. The `SourceIndex` is exactly the same with the `TrackerIndex`.
`UpdateTime`	The time the track was updated.
`Age`	Number of times the track survived.
`State`	Value of state vector at the update time.
`StateCovariance`	Uncertainty covariance matrix.
`Extent`	Spatial extent estimate of the tracked object, returned as a *d*-by-*d* matrix, where *d* is the dimension of the object. This field is only returned when the tracking filter is specified as a `ggiwphd` filter.
`MeasurementRate`	Expected number of detections from the tracked object. This field is only returned when the tracking filter is specified as a `ggiwphd` filter.
`IsConfirmed`	True if the track is assumed to be of a real target.
`IsCoasted`	`trackerPHD` does not support the `IsCoasted` field. The value is always 0.
`ObjectClassID`	`trackerPHD` does not support the `ObjectClassID` field. The value is always 0.
`StateParameters`	Parameters about the track state reference frame specified in the `StateParameters` property of the PHD tracker.

IsSelfReported	Indicate if the track is reported by the tracker. This field is used in a track fusion environment. It is returned as `true` by default.

A track is confirmed if:

- At least *M* detections are assigned to the track during the first *N* updates after track initialization. To specify the values *M* and *N*, use the **M and N for the M-out-of-N confirmation** parameter.
- The detection initiating the track has an `ObjectClassID` greater than zero.

`Tentative Tracks` — Tentative tracks

Simulink bus containing MATLAB structure

Tentative tracks, returned as a Simulink bus containing a MATLAB structure. See “Create Nonvirtual Buses” (Simulink). A track is tentative before it is confirmed.

This table shows the structure fields.

Field	Description
NumTracks	Number of tracks
Tracks	Array of track structures of a length set by the **Maximum number of tracks** parameter. Only the first `NumTracks` of these are actual tracks.

This table shows the fields of each track structure.

Field	Description
TrackID	Unique integer that identifies the track.
SourceIndex	Unique identifier the tracker in a multiple tracker environment. The `SourceIndex` is exactly the same with the `TrackerIndex`.
UpdateTime	The time the track was updated.
Age	Number of times the track survived.
State	Value of state vector at the update time.
StateCovariance	Uncertainty covariance matrix.
Extent	Spatial extent estimate of the tracked object, returned as a *d*-by-*d* matrix, where *d* is the dimension of the object. This field is only returned when the tracking filter is specified as a `ggiwphd` filter.
MeasurementRate	Expected number of detections from the tracked object. This field is only returned when the tracking filter is specified as a `ggiwphd` filter.
IsConfirmed	True if the track is assumed to be of a real target.
IsCoasted	`trackerPHD` does not support the `IsCoasted` field. The value is always 0.

`ObjectClassID`	`trackerPHD` does not support the `ObjectClassID` field. The value is always 0.
`StateParameters`	Parameters about the track state reference frame specified in the `StateParameters` property of the PHD tracker.
`IsSelfReported`	Indicate if the track is reported by the tracker. This field is used in a track fusion environment. It is returned as `true` by default.

Dependencies

To enable this port, select **Enable tentative tracks output**.

`All Tracks` — All tracks

Simulink bus containing MATLAB structure

Combined list of confirmed and tentative tracks, returned as a Simulink bus containing a MATLAB structure. See “Create Nonvirtual Buses” (Simulink).

This table shows the structure fields.

Field	Description
`NumTracks`	Number of tracks
`Tracks`	Array of track structures of a length set by the **Maximum number of tracks** parameter. Only the first `NumTracks` of these are actual tracks.

This table shows the fields of each track structure.

Field	Description
`TrackID`	Unique integer that identifies the track.
`SourceIndex`	Unique identifier the tracker in a multiple tracker environment. The `SourceIndex` is exactly the same with the `TrackerIndex`.
`UpdateTime`	The time the track was updated.
`Age`	Number of times the track survived.
`State`	Value of state vector at the update time.
`StateCovariance`	Uncertainty covariance matrix.
`Extent`	Spatial extent estimate of the tracked object, returned as a *d*-by-*d* matrix, where *d* is the dimension of the object. This field is only returned when the tracking filter is specified as a `ggiwphd` filter.
`MeasurementRate`	Expected number of detections from the tracked object. This field is only returned when the tracking filter is specified as a `ggiwphd` filter.
`IsConfirmed`	True if the track is assumed to be of a real target.

IsCoasted	trackerPHD does not support the IsCoasted field. The value is always 0.
ObjectClassID	trackerPHD does not support the ObjectClassID field. The value is always 0.
StateParameters	Parameters about the track state reference frame specified in the StateParameters property of the PHD tracker.
IsSelfReported	Indicate if the track is reported by the tracker. This field is used in a track fusion environment. It is returned as true by default.

Dependencies

To enable this port, select **Enable all tracks output**.

Parameters

Tracker Management

Tracker identifier — Unique tracker identifier
0 (default) | nonnegative integer

Unique tracker identifier, specified as a nonnegative integer. This parameter is used as the SourceIndex in the outputs, and distinguishes tracks that come from different trackers in a multiple-tracker system. You must specify this property as a positive integer to use the track outputs as inputs to a track fuser.

Example: 1

Filter initialization function name — Kalman filter initialization function
initcvkf (default) | function name

Kalman filter initialization function, specified as a function name. The toolbox provides several initialization functions. For an example of an initialization function, see initcvekf.

Threshold for assigning detections to tracks — Detection assignment threshold
30.0 (default) | positive real scalar

Detection assignment threshold, specified as a positive real scalar. To assign a detection to a track, the detection's normalized distance from the track must be less than the assignment threshold. If some detections remain unassigned to tracks that you want them assigned to, then increase the threshold. If some detections are assigned to incorrect tracks, decrease the threshold.

M and N for the M-out-of-N confirmation — Confirmation parameters for track creation
[2,3] (default) | two-element vector of positive integers

Confirmation parameters for track creation, specified as a two-element vector of positive integers, [M,N]. A track is confirmed when at least M detections are assigned to the track during the first N updates after track initialization. M must be less than or equal to N.

- When setting N, consider the number of times you want the tracker to update before it confirms a track. For example, if a tracker updates every 0.05 seconds, and you allow 0.5 seconds to make a confirmation decision, set N = 10.
- When setting M, take into account the probability of object detection for the sensors. The probability of detection depends on factors such as occlusion or clutter. You can reduce M when tracks fail to be confirmed or increase M when too many false detections are assigned to tracks.

Example: [3,5]

P and R for the P-out-of-R deletion — Track deletion threshold
[5 5] (default) | real-valued 1-by-2 vector of positive integers

Track deletion threshold for history logic, specified as a real-valued 1-by-2 vector of positive integers [P R]. If a confirmed track is not assigned to any detection P times in the last Q tracker updates, then the track is deleted.

Maximum number of tracks — Maximum number of tracks
200 (default) | positive integer

Maximum number of tracks that the block can process, specified as a positive integer.

Maximum number of sensors — Maximum number of sensors
20 (default) | positive integer

Maximum number of sensors that the block can process, specified as a positive integer. This value should be greater than or equal to the highest SensorIndex value used in the **Detections** input port.

Track state parameters — Parameters of the track state reference frame
struct (default) | struct | struct array

Parameters of the track state reference frame, specified as a struct or a struct array. Use this property to define the track state reference frame and how to transform the track from the tracker (called source) coordinate system to the fuser coordinate system.

Inputs and Outputs

Prediction time source — Source for prediction time
Input port (default) | Auto

Source for prediction time, specified as Input port or Auto. Select Input port to input an update time by using the **Prediction Time** input port. Otherwise, the simulation clock managed by Simulink determines the update time.

Example: Auto

Enable cost matrix input — Enable input port for cost matrix
off (default) | on

Select this check box to enable the input of a cost matrix by using the **Cost Matrix** input port.

Enable detectable track IDs input — Enable detectable track IDs input
off (default) | on

Select this check box to enable the **Detectable Track IDs** input port.

`Source of output bus name` — Source of output bus name
`Auto` (default) | `Property`

Source of output bus name, specified as `Auto` or `Property`.

- If you select `Auto`, the block automatically creates a bus name.
- If you select `Property`, specify the bus name using the **Specify an output bus name** parameter.

`Specify an output bus name` — Name of output bus
no default

Dependencies

To enable this parameter, set the **Source of output bus name** parameter to `Property`.

`Enable tentative tracks output` — Enable output port for tentative tracks
`off` (default) | `on`

Select this check box to enable the output of tentative tracks by using the **Tentative Tracks** output port.

`Enable all tracks output` — Enable output port for all tracks
`off` (default) | `on`

Select this check box to enable the output of all the tracks by using the **All Tracks** output port.

`Simulate using` — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.
- `Code generation` — Simulate the model using generated C/C++ code. The first time you run a simulation, Simulink generates C/C++ code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Apps
Bird's-Eye Scope

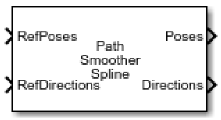
Blocks
Detection Concatenation | Radar Detection Generator | Scenario Reader | Vision Detection Generator

Objects
`multiObjectTracker`

Introduced in R2017b

Path Smoother Spline

Smooth vehicle path using cubic spline interpolation
Library: Automated Driving Toolbox



Description

The Path Smoother Spline block generates a smooth vehicle path, consisting of a sequence of discretized poses, by fitting the input reference path poses to a cubic spline. Given the input reference path directions, the block also returns the directions that correspond to each pose.

Use this block to convert a C^1-continuous path to a C^2-continuous path. C^1-continuous paths include Dubins or Reeds-Shepp paths that are returned by path planners. For more details on these path types, see “C1-Continuous and C2-Continuous Paths” on page 2-55.

You can use the returned poses and directions with a vehicle controller, such as the Lateral Controller Stanley block.

Ports

Input

`RefPoses` — Reference poses

M-by-3 matrix of [*x*, *y*, *Θ*] vectors

Reference poses of the vehicle along the path, specified as an *M*-by-3 matrix of [*x*, *y*, *Θ*] vectors, where *M* is the number of poses.

x and *y* specify the location of the vehicle in meters. *Θ* specifies the orientation angle of the vehicle in degrees.

Data Types: `single` | `double`

`RefDirections` — Reference directions

M-by-1 column vector of 1s (forward motion) and –1s (reverse motion)

Reference directions of the vehicle along the path, specified as an *M*-by-1 column vector of 1s (forward motion) and –1s (reverse motion). *M* is the number of reference directions. Each element of **RefDirections** corresponds to a pose in the **RefPoses** input port.

Data Types: `single` | `double`

Output

`Poses` — Discretized poses of smoothed path

N-by-3 matrix of [*x*, *y*, *Θ*] vectors

Discretized poses of the smoothed path, returned as an *N*-by-3 matrix of [*x*, *y*, *Θ*] vectors. *N* is the number of poses specified in the **Number of output poses** parameter.

x and *y* specify the location of the vehicle in meters. *Θ* specifies the orientation angle of the vehicle in degrees.

The values in **Poses** are of the same data type as the values in the **RefPoses** input port.

`Directions` — Driving directions at each output pose

N-by-1 column vector of 1s (forward motion) and –1s (reverse motion)

Driving directions of the vehicle at each output pose in **Poses**, returned as an *N*-by-1 column vector of 1s (forward motion) and –1s (reverse motion). *N* is the number of poses specified in the **Number of output poses** parameter.

The values in **Directions** are of the same data type as the values in the **RefDirections** input port.

You can use **Directions** to specify the reference path of a vehicle. You can also use **Directions**, along with **CumLengths** and **Curvatures**, to generate a reference velocity profile for the vehicle. See the Velocity Profiler block and the "Automated Parking Valet in Simulink" example.

`CumLengths` — Cumulative path lengths

N-by-1 real-valued column vector

Cumulative path lengths at each output pose in **Poses**, returned as an *N*-by-1 real-valued column vector. *N* is the number of poses specified in the **Number of output poses** parameter. Units are in meters.

You can use **CumLengths**, along with **Directions** and **Curvatures**, to generate a reference velocity profile for the vehicle. See the Velocity Profiler block and the "Automated Parking Valet in Simulink" example.

Dependencies

To enable this port, select the **Show CumLengths and Curvatures output ports** parameter.

`Curvatures` — Signed path curvatures

N-by-1 real-valued column vector

Signed path curvatures at each output pose in **Poses**, returned as an *N*-by-1 real-valued column vector. *N* is the number of poses specified in the **Number of output poses** parameter. Units are in radians per meter.

You can use **Curvatures**, along with **Directions** and **CumLengths**, to generate a reference velocity profile for the vehicle. See the Velocity Profiler block and the "Automated Parking Valet in Simulink" example.

Dependencies

To enable this port, select the **Show CumLengths and Curvatures output ports** parameter.

Parameters

`Number of output poses` — Number of smooth poses to return

`100` (default) | positive integer

Number of smooth poses to return in the **Poses** output port, specified as a positive integer. To increase the granularity of the returned poses, increase this parameter value.

`Minimum separation of input poses` — Minimum separation between poses
`1e-3` (default) | positive real scalar

Minimum separation between poses, in meters, specified as a positive real scalar. If the Euclidean (*x*, *y*) distance between two poses is less than this value, then the block uses only one of these poses for interpolation.

`Sample time` — Sample time
`-1` (default) | positive real scalar

Sample time of the block, in seconds, specified as `-1` or as a positive real scalar. The default of `-1` means that the block inherits its sample time from upstream blocks.

`Show CumLengths and Curvatures output ports` — Output cumulative path lengths and curvatures
`off` (default) | `on`

Select this parameter to enable the **CumLengths** and **Curvatures** output ports.

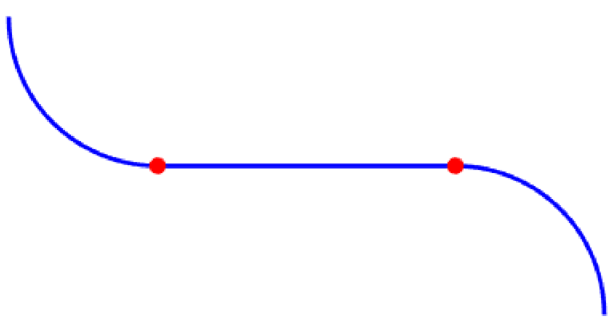
`Simulate using` — Type of simulation to run
`Code Generation` (default) | `Interpreted Execution`

- `Code generation` — Simulate the model using generated C/C++ code. The first time you run a simulation, Simulink generates C/C++ code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.
- `Interpreted execution` — Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.

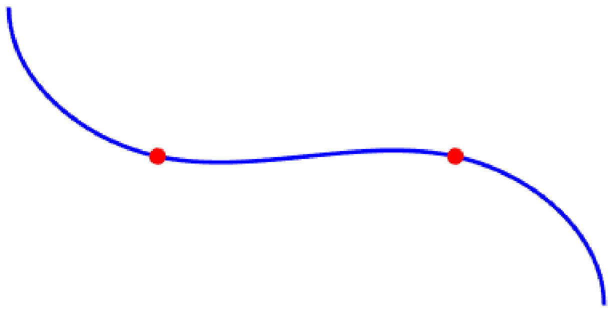
More About

C^1-Continuous and C^2-Continuous Paths

A path is C^1-continuous if its derivative exists and is continuous. Paths that are only C^1-continuous have discontinuities in their curvature. For example, a path composed of Dubins or Reeds-Sheep path segments has discontinuities in curvature at the points where the segments join. These discontinuities result in changes in direction that are not smooth enough for driving with passengers.



A path is also C^2-continuous if its second derivative exists and is continuous. C^2-continuous paths have continuous curvature and are smooth enough for driving with passengers.



Algorithms

- The path-smoothing algorithm interpolates a parametric cubic spline that passes through all input reference pose points. The parameter of the spline is the cumulative chord length at these points. [1]
- The tangent direction of the smoothed output path approximately matches the orientation angle of the vehicle at the starting and goal poses.

References

[1] Floater, Michael S. "On the Deviation of a Parametric Cubic Spline Interpolant from Its Data Polygon." *Computer Aided Geometric Design*. Vol. 25, Number 3, 2008, pp. 148–156.

[2] Lepetic, Marko, Gregor Klancar, Igor Skrjanc, Drago Matko, and Bostjan Potocnik. "Time Optimal Path Planning Considering Acceleration Limits." *Robotics and Autonomous Systems*. Vol. 45, Numbers 3–4, 2003, pp. 199–210.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Functions
`smoothPathSpline`

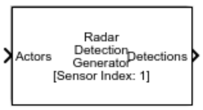
Blocks
Lateral Controller Stanley | Longitudinal Controller Stanley | Velocity Profiler

Introduced in R2019a

Radar Detection Generator

Create detection objects from radar measurements

Library: Automated Driving Toolbox / Driving Scenario and Sensor Modeling



Description

The Radar Detection Generator block generates detections from radar measurements taken by a radar sensor mounted on an ego vehicle. Detections are derived from simulated actor poses and are generated at intervals equal to the sensor update interval. By default, detections are referenced to the coordinate system of the ego vehicle. The generator can simulate real detections with added random noise and also generate false alarm detections. A statistical model generates the measurement noise, true detections, and false positives. The random numbers generated by the statistical model are controlled by random number generator settings on the **Measurements** tab. You can use the Radar Detection Generator to create input to a Multi-Object Tracker block. When building scenarios and sensor models using the **Driving Scenario Designer** app, the radar sensors exported to Simulink are output as Radar Detection Generator blocks.

Ports

Input

`Actors` — Scenario actor poses

Simulink bus containing MATLAB structure

Scenario actor poses in ego vehicle coordinates, specified as a Simulink bus containing a MATLAB structure.

The structure must contain these fields.

Field	Description	Type
`NumActors`	Number of actors	Nonnegative integer
`Time`	Current simulation time	Real-valued scalar
`Actors`	Actor poses	`NumActors`-length array of actor pose structures

Each actor pose structure in `Actors` must contain these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.

Field	Description
`Position`	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
`Velocity`	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (*ω*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

Output

Detections — Detections
Simulink bus containing MATLAB structure

Object detections, returned as a Simulink bus containing a MATLAB structure. For more details about buses, see “Create Nonvirtual Buses” (Simulink).

You can pass object detections from these sensors and other sensors to a tracker, such as a Multi-Object Tracker block, and generate tracks.

Field	Description	Type
`NumDetections`	Number of detections	integer
`IsValidTime`	False when updates are requested at times that are between block invocation intervals	Boolean
`Detections`	Object detections	Array of object detection structures of length set by the **Maximum number of reported detections** parameter. Only `NumDetections` of these are actual detections.

Each object detection structure contains these properties.

Property	Definition
`Time`	Measurement time
`Measurement`	Object measurements
`MeasurementNoise`	Measurement noise covariance matrix

Property	Definition
SensorIndex	Unique ID of the sensor
ObjectClassID	Object classification
ObjectAttributes	Additional information passed to tracker
MeasurementParameters	Parameters used by initialization functions of nonlinear Kalman tracking filters

- For Cartesian coordinates, Measurement and MeasurementNoise are reported in the coordinate system specified by the **Coordinate system used to report detections** parameter.
- For spherical coordinates, Measurement and MeasurementNoise are reported in the spherical coordinate system based on the sensor Cartesian coordinate system.

Measurement and Measurement Noise

<table>
<tr><th>Coordinate System Used to Report Detections</th><th>Measurement and Measurement Noise Coordinates</th></tr>
<tr><td>'Ego Cartesian'</td><td rowspan="2">Coordinate dependence on Enable range rate measurements
<table>
<tr><th>Enable range rate measurements</th><th>Coordinates</th></tr>
<tr><td>true</td><td>[x;y;z;vx;vy;vz]</td></tr>
<tr><td>false</td><td>[x;y;z]</td></tr>
</table></td></tr>
<tr><td>'Sensor Cartesian'</td></tr>
<tr><td>'Sensor spherical'</td><td>Coordinate dependence on Enable elevation angle measurements and Enable range rate measurements
<table>
<tr><th>Enable range rate measurements</th><th>Enable elevation angle measurements</th><th>Coordinates</th></tr>
<tr><td>true</td><td>true</td><td>[az;el;rng;rr]</td></tr>
<tr><td>true</td><td>false</td><td>[az;rng;rr]</td></tr>
<tr><td>false</td><td>true</td><td>[az;el;rng]</td></tr>
<tr><td>false</td><td>false</td><td>[az;rng]</td></tr>
</table></td></tr>
</table>

`MeasurementParameters`

Parameter	Definition
`Frame`	Enumerated type indicating the frame used to report measurements. When `Frame` is set to `'rectangular'`, detections are reported in Cartesian coordinates. When `Frame` is set `'spherical'`, detections are reported in spherical coordinates.
`OriginPosition`	3-D vector offset of the sensor origin from the ego vehicle origin. The vector is derived from the `SensorLocation` and `Height` properties specified in the `radarDetectionGenerator`.
`Orientation`	Orientation of the radar sensor coordinate system with respect to the ego vehicle coordinate system. The orientation is derived from the `Yaw`, `Pitch`, and `Roll` properties of the `radarDetectionGenerator`.
`HasVelocity`	Indicates whether measurements contain velocity or range rate components.
`HasElevation`	Indicates whether measurements contain elevation components.

The `ObjectAttributes` property of each detection is a structure with these fields.

Field	Definition
`TargetIndex`	Identifier of the actor, `ActorID`, that generated the detection. For false alarms, this value is negative.
`SNR`	Signal-to-noise ratio of the detection. Units are in dB.

Parameters

Parameters

Sensor Identification

`Unique identifier of sensor` — Unique sensor identifier
1 (default) | positive integer

Unique sensor identifier, specified as a positive integer. The sensor identifier distinguishes detections that come from different sensors in a multisensor system. If a model contains multiple sensor blocks with the same sensor identifier, the **Bird's-Eye Scope** displays an error.

Example: 5

`Required interval between sensor updates (s)` — Required time interval
`0.1` (default) | positive real scalar

Required time interval between sensor updates, specified as a positive real scalar. The value of this parameter must be an integer multiple of the **Actors** input port data interval. Updates requested from the sensor between update intervals contain no detections. Units are in seconds.

Sensor Extrinsics

`Sensor's (x,y) position (m)` — Location of the radar sensor center
`[3.4 0]` (default) | real-valued 1-by-2 vector

Location of the radar sensor center, specified as a real-valued 1-by-2 vector. The **Sensor's (x,y) position (m)** and **Sensor's height (m)** parameters define the coordinates of the radar sensor with respect to the ego vehicle coordinate system. The default value corresponds to a radar mounted at the center of the front grill of a sedan. Units are in meters.

`Sensor's height (m)` — Radar sensor height above the ground plane
`0.2` (default) | positive real scalar

Radar sensor height above the ground plane, specified as a positive real scalar. The height is defined with respect to the vehicle ground plane. The **Sensor's (x,y) position (m)** and **Sensor's height (m)** parameters define the coordinates of the radar sensor with respect to the ego vehicle coordinate system. The default value corresponds to a radar mounted at the center of the front grill of a sedan. Units are in meters.

Example: `0.25`

`Yaw angle of sensor mounted on ego vehicle (deg)` — Yaw angle of sensor
`0` (default) | real scalar

Yaw angle of radar sensor, specified as a real scalar. Yaw angle is the angle between the center line of the ego vehicle and the downrange axis of the radar sensor. A positive yaw angle corresponds to a clockwise rotation when looking in the positive direction of the *z*-axis of the ego vehicle coordinate system. Units are in degrees.

Example: `-4.0`

`Pitch angle of sensor mounted on ego vehicle (deg)` — Pitch angle of sensor
`0` (default) | real scalar

Pitch angle of sensor, specified as a real scalar. The pitch angle is the angle between the downrange axis of the radar sensor and the *x*-*y* plane of the ego vehicle coordinate system. A positive pitch angle corresponds to a clockwise rotation when looking in the positive direction of the *y*-axis of the ego vehicle coordinate system. Units are in degrees.

Example: `3.0`

`Roll angle of sensor mounted on ego vehicle (deg)` — Roll angle of sensor
`0` (default) | real scalar

Roll angle of the radar sensor, specified as a real scalar. The roll angle is the angle of rotation of the downrange axis of the radar around the *x*-axis of the ego vehicle coordinate system. A positive roll angle corresponds to a clockwise rotation when looking in the positive direction of the *x*-axis of the coordinate system. Units are in degrees.

Port Settings

`Source of output bus name` — Source of output bus name
`Auto` (default) | `Property`

Source of output bus name, specified as `Auto` or `Property`. If you choose `Auto`, the block will automatically create a bus name. If you choose `Property`, specify the bus name using the **Specify an output bus name** parameter.

Example: `Property`

`Specify an output bus name` — Name of output bus
no default

Name of output bus.

Dependencies

To enable this parameter, set the **Source of output bus name** parameter to `Property`.

Detection Reporting

`Maximum number of reported detections` — Maximum number of reported detections
`50` (default) | positive integer

Maximum number of detections reported by the sensor, specified as a positive integer. Detections are reported in order of increasing distance from the sensor until the maximum number is reached.

Example: `100`

`Coordinate system used to report detections` — Coordinate system of reported detections
`Ego Cartesian` (default) | `Sensor Cartesian` | `Sensor Spherical`

Coordinate system of reported detections, specified as one of these values:

- `Ego Cartesian` — Detections are reported in the ego vehicle Cartesian coordinate system.
- `Sensor Cartesian`— Detections are reported in the sensor Cartesian coordinate system.
- `Sensor spherical` — Detections are reported in a spherical coordinate system. This coordinate system is centered at the radar and aligned with the orientation of the radar on the ego vehicle.

`Simulate using` — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.
- `Code generation` — Simulate the model using generated C/C++ code. The first time you run a simulation, Simulink generates C/C++ code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.

Measurements

Accuracy Settings

`Azimuthal resolution of radar (deg)` — Azimuth resolution of radar
`4.0` (default) | positive real scalar

Azimuth resolution of the radar, specified as a positive real scalar. The azimuth resolution defines the minimum separation in azimuth angle at which the radar can distinguish two targets. The azimuth resolution is typically the 3dB-downpoint in azimuth angle beamwidth of the radar. Units are in degrees.

Example: `6.5`

`Elevation resolution of radar (deg)` — Elevation resolution of radar
`10.0` (default) | positive real scalar

Elevation resolution of the radar, specified as a positive real scalar. The elevation resolution defines the minimum separation in elevation angle at which the radar can distinguish two targets. The elevation resolution is typically the 3dB-downpoint in elevation angle beamwidth of the radar. Units are in degrees.

Example: `3.5`

Dependencies

To enable this parameter, select the **Enable elevation angle measurements** check box.

`Range resolution of radar (m)` — Range resolution of radar
`2.5` (default) | positive real scalar

Range resolution of the radar, specified as a positive real scalar. The range resolution defines the minimum separation in range at which the radar can distinguish between two targets. Units are in meters.

Example: `5.0`

`Range rate resolution of radar (m/s)` — Range rate resolution of the radar
`0.5` (default) | positive real scalar

Range rate resolution of the radar, specified as a positive real scalar. The range rate resolution defines the minimum separation in range rate at which the radar can distinguish between two targets. Units are in meters per second.

Example: `0.75`

Dependencies

To enable this parameter, select the **Enable range rate measurements** check box.

Bias Settings

`Fractional azimuthal bias component of radar` — Azimuth bias fraction
`0.1` (default) | nonnegative real scalar

Azimuth bias fraction of the radar, specified as a nonnegative real scalar. The azimuth bias is expressed as a fraction of the azimuth resolution specified in the **Azimuthal resolution of radar (deg)** parameter. Units are dimensionless.

Example: 0.3

`Fractional elevation bias component of radar` — Elevation bias fraction
`0.1` (default) | nonnegative real scalar

Elevation bias fraction of the radar, specified as a nonnegative real scalar. The elevation bias is expressed as a fraction of the elevation resolution specified in the **Elevation resolution of radar (deg)** parameter. Units are dimensionless.

Example: 0.2

Dependencies

To enable this parameter, select the **Enable elevation angle measurements** check box.

`Fractional range bias component of radar` — Range bias fraction

`0.05` (default) | nonnegative real scalar

Range bias fraction of the radar, specified as a nonnegative real scalar. Range bias is expressed as a fraction of the range resolution specified in the **Range resolution of radar (m)** parameter. Units are dimensionless.

Example: `0.15`

`Fractional range rate bias component of radar` — Range rate bias fraction of the radar

`0.05` (default) | nonnegative real scalar

Range rate bias fraction of the radar, specified as a nonnegative real scalar. Range rate bias is expressed as a fraction of the range rate resolution specified in **Range rate resolution of radar (m)** parameter. Units are dimensionless.

Example: `0.2`

Dependencies

To enable this parameter, select the **Enable range rate measurements** check box.

Detector Settings

`Total angular field of view for radar (deg)` — Field of view of radar sensor

`[20 5]` (default) | real-valued 1-by-2 vector of positive values

Field of view of radar sensor, specified as a real-valued 1-by-2 vector of positive values, `[azfov elfov]`. The field of view defines the angular extent spanned by the sensor. Each component must lie in the interval (0,180]. Targets outside of the field of view of the radar are not detected. Units are in degrees.

Example: `[14 7]`

`Maximum detection range (m)` — Maximum detection range

`150` (default) | positive real scalar

Maximum detection range, specified as a positive real scalar. The radar cannot detect a target beyond this range. Units are in meters.

Example: `250`

`Minimum and maximum range rates that can be reported` — Minimum and maximum detection range rates

`[-100 100]` (default) | real-valued 1-by-2 vector

Minimum and maximum detection range rates, specified as a real-valued 1-by-2 vector. The radar cannot detect a target outside of this range rate interval. Units are in meters per second.

Example: `[-200 200]`

Dependencies

To enable this parameter, select the **Enable range rate measurements** check box.

`Detection probability` — Probability of detecting a target

`0.9` (default) | positive real scalar less than or equal to 1

Probability of detecting a target, specified as a positive real scalar less than or equal to one. This quantity defines the probability of detecting target that has a radar cross-section specified by the **Radar cross section at which detection probability is achieved (dBsm)** parameter at the reference detection range specified by the **Range where detection probability is achieved (m)** parameter.

Example: `0.95`

`Rate at which false alarms are reported` — False alarm rate

`1e-6` (default) | positive real scalar

False alarm rate within a radar resolution cell, specified as a positive real scalar in the range [10^{-7}, 10^{-3}]. Units are dimensionless.

Example: `1e-5`

`Range where detection probability is achieved (m):` — Reference range for given probability of detection

`100` (default) | positive real scalar

Reference range for a given probability of detection, specified as a positive real scalar. The reference range is the range when a target having a radar cross-section specified by **Radar cross section at which detection probability is achieved (dBsm)** is detected with a probability of specified by **Detection probability**. Units are in meters.

Example: `150`

`Radar cross section at which detection probability is achieved (dBsm)` — Reference radar cross-section for given probability of detection

`0.0` (default) | nonnegative real scalar

Reference radar cross-section (RCS) for given probability of detection, specified as a nonnegative real scalar. The reference RCS is the value at which a target is detected with probability specified by **Detection probability**. Units are in dBsm.

Example: `2.0`

Measurement Settings

`Enable elevation angle measurements` — Enable radar to measure elevation

`off` (default) | `on`

Select this check box to model a radar that can measure target elevation angles.

`Enable range rate measurements` — Enable radar to measure range rate

`on` (default) | `off` | `on`

Select this check box to model a radar that can measure target range rate.

`Add noise to measurements` — Enable adding noise to radar sensor measurements

`on` (default) | `off`

Select this check box to add noise to radar sensor measurements. Otherwise, the measurements are noise-free. The `MeasurementNoise` property of each detection is always computed and is not affected by the value you specify for the **Add noise to measurements** parameter. By leaving this check box `off`, you can pass the sensor's ground truth measurements into a Multi-Object Tracker block.

`Enable false detections` — Enable creating false alarm radar detections

`on` (default) | `off`

Select this check box to enable reporting false alarm radar measurements. Otherwise, only actual detections are reported.

Random Number Generator Settings

`Select method to specify initial seed` — Method to specify random number generator seed

`Repeatable` (default) | `Specify seed` | `Not repeatable`

Method to set the random number generator seed, specified as one of the options in the table.

Option	Description
`Repeatable`	The block generates a random initial seed for the first simulation and reuses this seed for all subsequent simulations. Select this parameter to generate repeatable results from the statistical sensor model. To change this initial seed, at the MATLAB command prompt, enter: `clear all`.
`Specify seed`	Specify your own random initial seed for reproducible results by using the **Specify seed** parameter.
`Not repeatable`	The block generates a new random initial seed after each simulation run. Select this parameter to generate nonrepeatable results from the statistical sensor model.

`Initial seed` — Random number generator seed

`0` (default) | nonnegative integer less than 2^{32}

Random number generator seed, specified as a nonnegative integer less than 2^{32}.

Example: `2001`

Dependencies

To enable this parameter, set the `Random Number Generator Settings` parameter to `Specify seed`.

Actor Profiles

`Select method to specify actor profiles` — Method to specify actor profiles

`Parameters` (default) | `MATLAB expression`

Method to specify actor profiles, specified as `Parameters` or `MATLAB expression`. When you select `Parameters`, you set the actor profiles using the parameters in the **Actor Profiles** tab. When you select `MATLAB expression`, set the actor profiles using the **MATLAB expression for actor profiles** parameter.

`MATLAB expression for actor profiles` — MATLAB expression for actor profiles
`struct('ClassID',0,'Length',4.7,'Width',1.8,'Height',1.4,'OriginOffset',[-1.35,0,0])` (default) | MATLAB structure | MATLAB structure array | valid MATLAB expression

MATLAB expression for actor profiles, specified as a MATLAB structure, a MATLAB structure array, or a valid MATLAB expression that produces such a structure or structure array.

If your Scenario Reader block reads data from a `drivingScenario` object, to obtain the actor profiles directly from this object, set this expression to call the `actorProfiles` function on the object. For example: `actorProfiles(scenario)`.

Example: `struct('ClassID',5,'Length',5.0,'Width',2,'Height',2,'OriginOffset',[-1.55,0,0])`

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `MATLAB expression`.

`Unique identifier for actors` — Scenario-defined actor identifier
`[]` (default) | positive integer | length-*L* vector of unique positive integers

Scenario-defined actor identifier, specified as a positive integer or length-*L* vector of unique positive integers. *L* must equal the number of actors input into the **Actor** input port. The vector elements must match `ActorID` values of the actors. You can specify **Unique identifier for actors** as `[]`. In this case, the same actor profile parameters apply to all actors.

Example: `[1,2]`

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `Parameters`.

`User-defined integer to classify actors` — User-defined classification identifier
`0` (default) | integer | length-*L* vector of integers

User-defined classification identifier, specified as an integer or length-*L* vector of integers. When **Unique identifier for actors** is a vector, this parameter is a vector of the same length with elements in one-to-one correspondence to the actors in **Unique identifier for actors**. When **Unique identifier for actors** is empty, `[]`, you must specify this parameter as a single integer whose value applies to all actors.

Example: 2

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `Parameters`.

`Length of actors cuboids (m)` — Length of cuboid
`4.7` (default) | positive real scalar | length-*L* vector of positive values

Length of cuboid, specified as a positive real scalar or length-*L* vector of positive values. When **Unique identifier for actors** is a vector, this parameter is a vector of the same length with elements in one-to-one correspondence to the actors in **Unique identifier for actors**. When **Unique identifier for actors** is empty, `[]`, you must specify this parameter as a positive real scalar whose value applies to all actors. Units are in meters.

Example: 6.3

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `Parameters`.

`Width of actors cuboids (m)` — Width of cuboid

`4.7` (default) | positive real scalar | length-*L* vector of positive values

Width of cuboid, specified as a positive real scalar or length-*L* vector of positive values. When **Unique identifier for actors** is a vector, this parameter is a vector of the same length with elements in one-to-one correspondence to the actors in **Unique identifier for actors**. When **Unique identifier for actors** is empty, `[]`, you must specify this parameter as a positive real scalar whose value applies to all actors. Units are in meters.

Example: 4.7

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `Parameters`.

`Height of actors cuboids (m)` — Height of cuboid

`4.7` (default) | positive real scalar | length-*L* vector of positive values

Height of cuboid, specified as a positive real scalar or length-*L* vector of positive values. When **Unique identifier for actors** is a vector, this parameter is a vector of the same length with elements in one-to-one correspondence to the actors in **Unique identifier for actors**. When **Unique identifier for actors** is empty, `[]`, you must specify this parameter as a positive real scalar whose value applies to all actors. Units are in meters.

Example: `2.0`

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `Parameters`.

`Rotational center of actors from bottom center (m)` — Rotational center of the actor

{ [-1.35, 0, 0] } (default) | length-*L* cell array of real-valued 1-by-3 vectors

Rotational center of the actor, specified as a length-*L* cell array of real-valued 1-by-3 vectors. Each vector represents the offset of the rotational center of the actor from the bottom-center of the actor. For vehicles, the offset corresponds to the point on the ground beneath the center of the rear axle. When **Unique identifier for actors** is a vector, this parameter is a cell array of vectors with cells in one-to-one correspondence to the actors in **Unique identifier for actors**. When **Unique identifier for actors** is empty, `[]`, you must specify this parameter as a cell array of one element containing the offset vector whose values apply to all actors. Units are in meters.

Example: [-1.35, .2, .3]

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `Parameters`.

`Radar cross section pattern (dBsm)` — Radar cross-section

`{[10,10;10,10]}` (default) | real-valued *Q*-by-*P* matrix | length-*L* cell array of real-valued *Q*-by-*P* matrices

Radar cross-section (RCS) of actor, specified as a real-valued *Q*-by-*P* matrix or length-*L* cell array of real-valued *Q*-by-*P* matrices. *Q* is the number of elevation angles specified by the corresponding cell in the **Elevation angles defining RCSPattern (deg)** parameter. *P* is the number of azimuth angles specified by the corresponding cell in **Azimuth angles defining RCSPattern (deg)** property. When **Unique identifier for actors** is a vector, this parameter is a cell array of matrices with cells in one-to-one correspondence to the actors in **Unique identifier for actors**. *Q* and *P* can vary in the cell array. When **Unique identifier for actors** is empty, `[]`, you must specify this parameter as a cell array with one element containing a matrix whose values apply to all actors. Units are in dBsm.

Example: [10 14 10; 9 13 9]

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `Parameters`.

`Azimuth angles defining RCSPattern (deg)` — Azimuth angles of radar cross-section pattern

`{[-180 180]}` (default) | length-*L* cell array of real-valued *P*-length vectors

Azimuth angles of radar cross-section pattern, specified as a length-*L* cell array of real-valued *P*-length vectors . Each vector represents the azimuth angles of the *P*-columns of the radar cross section specified in **Radar cross section pattern (dBsm)**. When **Unique identifier for actors** is a vector, this parameter is a cell array of vectors with cells in one-to-one correspondence to the actors in **Unique identifier for actors**. *P* can vary in the cell array. When **Unique identifier for actors** is empty, `[]`, you must specify this parameter as a cell array with one element containing a vector whose values apply to all actors. Units are in degrees. Azimuth angles lie in the range -180° to 180° and must be in strictly increasing order.

When the radar cross sections specified in the cells of **Radar cross section pattern (dBsm)** all have the same dimensions, you need only specify a cell array with one element containing the azimuth angle vector.

Example: `[-90:90]`

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `Parameters`.

`Elevation angles defining RCSPattern (deg)` — Elevation angles of radar cross-section pattern

`{[-90 90]}` (default) | length-*L* cell array of real-valued *Q*-length vectors

Elevation angles of radar cross-section pattern, specified as a length-*L* cell array of real-valued *Q*-length vectors . Each vector represent the elevation angles of the *Q*-columns of the radar cross section specified in **Radar cross section pattern (dBsm)**. When **Unique identifier for actors** is a vector, this parameter is a cell array of vectors with cells in one-to-one correspondence to the actors in **Unique identifier for actors**. *Q* can vary in the cell array. When **Unique identifier for actors** is empty, `[]`, you must specify this parameter as a cell array with one element containing a vector whose values apply to all actors. Units are in degrees. Elevation angles lie in the range -90° to 90° and must be in strictly increasing order.

When the radar cross sections that are specified in the cells of **Radar cross section pattern (dBsm)** all have the same dimensions, you need only specify a cell array with one element containing an elevation angle vector.

Example: `[-25:25]`

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `Parameters`.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Apps
Bird's-Eye Scope

Blocks
Detection Concatenation | Lidar Point Cloud Generator | Multi-Object Tracker | Scenario Reader | Vision Detection Generator

Objects
`radarDetectionGenerator`

Topics
“Create Nonvirtual Buses” (Simulink)

Introduced in R2017b

Scenario Reader

Read driving scenario into model

Library: Automated Driving Toolbox / Driving Scenario and Sensor Modeling

Description

The Scenario Reader block reads the roads and actors from a scenario file created using the **Driving Scenario Designer** app or from a `drivingScenario` object. The block outputs the poses of actors in either the coordinate system of the ego vehicle or the world coordinates of the scenario. You can also output the lane boundaries or output the ego vehicle pose for use in the 3D simulation environment.

To generate object and lane boundary detections from output actor poses and lane boundaries, pass the pose and boundary outputs to sensor blocks. Use the synthetic detections generated from these sensors to test the performance of sensor fusion algorithms, tracking algorithms, and other automated driving assistance system (ADAS) algorithms. To visualize the performance of these algorithms, use the **Bird's-Eye Scope**.

You can read the ego vehicle from the scenario or specify an ego vehicle defined in your model as an input to the Scenario Reader block. Use this option to test closed-loop vehicle controller algorithms, such as autonomous emergency braking (AEB), lane keeping assist (LKA), or adaptive cruise control (ACC).

Limitations

- The Scenario Reader block does not read sensor data from scenario files saved from the **Driving Scenario Designer** app. To reproduce sensors in Simulink, in the app, open the scenario file that contains the sensors. Then, from the app toolstrip, select **Export** > **Export Sensor Simulink Model**. Copy the generated sensor blocks into an existing model. Alternatively, select **Export** > **Export Simulink Model** and start a new model from the generated Scenario Reader block and sensor blocks.
- Large road networks, including OpenDRIVE road networks, can take up to several minutes to read into models.

Ports

Input

`Ego Vehicle` — Ego vehicle pose

Simulink bus containing MATLAB structure

Ego vehicle pose, specified as a Simulink bus containing a MATLAB structure.

The structure must have these fields.

Field	Description
ActorID	Scenario-defined actor identifier, specified as a positive integer.
Position	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
Velocity	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
Roll	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
Pitch	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
Yaw	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
AngularVelocity	Angular velocity (*ω*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

Output the ego vehicle pose when you are converting actors from ego vehicle coordinates to world coordinates for use in the 3D simulation environment. For example, see "Visualize Sensor Data from Unreal Engine Simulation Environment".

Dependencies

To enable this port, set these parameters in this order:

1. Set **Coordinate system of actors output** to `Vehicle coordinates`.
2. Set **Source of ego vehicle** to `Input port`.

Output

`Actors` — Scenario actor poses

Simulink bus containing MATLAB structure

Scenario actor poses, returned as a Simulink bus containing a MATLAB structure.

The structure has these fields.

Field	Description	Type
NumActors	Number of actors	Nonnegative integer
Time	Current simulation time	Real-valued scalar
Actors	Actor poses	`NumActors`-length array of actor pose structures

Each actor pose structure in `Actors` has these fields.

Field	Description
ActorID	Scenario-defined actor identifier, specified as a positive integer.

Field	Description
`Position`	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
`Velocity`	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (*ω*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

The pose of the ego vehicle is excluded from the `Actors` array.

To return actor poses from the block, you must run the entire driving scenario simulation to completion.

`Lane Boundaries` — Scenario lane boundaries
Simulink bus containing MATLAB structure

Scenario lane boundaries, returned as a Simulink bus containing a MATLAB structure.

The structure has these fields.

Field	Description	Type
`NumLaneBoundaries`	Number of lane boundaries	Nonnegative integer
`Time`	Current simulation time	Real scalar
`LaneBoundaries`	Lane boundaries	`NumLaneBoundaries`-length array of lane boundary structures

Each lane boundary structure in `LaneBoundaries` has these fields.

Field	Description

`Coordinates`	Lane boundary coordinates, specified as a real-valued *N*-by-3 matrix, where *N* is the number of lane boundary coordinates. Lane boundary coordinates define the position of points on the boundary at specified longitudinal distances away from the ego vehicle, along the center of the road. • In MATLAB, specify these distances by using the `'XDistance'` name-value pair argument of the `laneBoundaries` function. • In Simulink, specify these distances by using the **Distances from ego vehicle for computing boundaries (m)** parameter of the Scenario Reader block or the **Distance from parent for computing lane boundaries** parameter of the Simulation 3D Vision Detection Generator block. This matrix also includes the boundary coordinates at zero distance from the ego vehicle. These coordinates are to the left and right of the ego-vehicle origin, which is located under the center of the rear axle. Units are in meters.
`Curvature`	Lane boundary curvature at each row of the `Coordinates` matrix, specified as a real-valued *N*-by-1 vector. *N* is the number of lane boundary coordinates. Units are in radians per meter.
`CurvatureDerivative`	Derivative of lane boundary curvature at each row of the `Coordinates` matrix, specified as a real-valued *N*-by-1 vector. *N* is the number of lane boundary coordinates. Units are in radians per square meter.
`HeadingAngle`	Initial lane boundary heading angle, specified as a real scalar. The heading angle of the lane boundary is relative to the ego vehicle heading. Units are in degrees.
`LateralOffset`	Distance of the lane boundary from the ego vehicle position, specified as a real scalar. An offset to a lane boundary to the left of the ego vehicle is positive. An offset to the right of the ego vehicle is negative. Units are in meters.

`BoundaryType`	Type of lane boundary marking, specified as one of these values: • `'Unmarked'` — No physical lane marker exists • `'Solid'` — Single unbroken line • `'Dashed'` — Single line of dashed lane markers • `'DoubleSolid'` — Two unbroken lines • `'DoubleDashed'` — Two dashed lines • `'SolidDashed'` — Solid line on the left and a dashed line on the right • `'DashedSolid'` — Dashed line on the left and a solid line on the right
`Strength`	Saturation strength of the lane boundary marking, specified as a real scalar from 0 to 1. A value of `0` corresponds to a marking whose color is fully unsaturated. The marking is gray. A value of `1` corresponds to a marking whose color is fully saturated.
`Width`	Lane boundary width, specified as a positive real scalar. In a double-line lane marker, the same width is used for both lines and for the space between lines. Units are in meters.
`Length`	Length of dash in dashed lines, specified as a positive real scalar. In a double-line lane marker, the same length is used for both lines.
`Space`	Length of space between dashes in dashed lines, specified as a positive real scalar. In a dashed double-line lane marker, the same space is used for both lines.

The number of returned lane boundary structures depends on the **Lane boundaries to output** parameter value.

Dependencies

To enable this port, set these parameters in this order:

1. Set **Coordinate system of actors output** to `Vehicle coordinates`.
2. Set **Lane boundaries to output** to `Ego lane boundaries` or `All lane boundaries`.

`Ego Vehicle` — Ego vehicle pose

Simulink bus containing MATLAB structure

Ego vehicle pose, returned as a Simulink bus containing a MATLAB structure.

The structure must contain these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
`Velocity`	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (*ω*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

Dependencies

To enable this port, set these parameters in this order:

1. Set **Coordinate system of actors output** to `Vehicle coordinates`.
2. Set **Source of ego vehicle** to `Scenario`.
3. Select **Output ego vehicle pose**.

Parameters

`Source of driving scenario` — Source of driving scenario

`From file` (default) | `From workspace`

Source of driving scenario, specified as one of these options:

- `From file` — In the **Driving Scenario Designer file name** parameter, specify the name of a scenario file that was saved from the **Driving Scenario Designer** app.
- `From workspace` — In the **MATLAB or model workspace variable name** parameter, specify the name of a MATLAB or model workspace variable that contains a `drivingScenario` object.

`Driving Scenario Designer file name` — Scenario file name

`EgoVehicleGoesStraight.mat` (default) | scenario file on MATLAB search path | path to scenario file

Scenario file name, specified as a scenario file on the MATLAB search path or as the full path to a scenario file. A scenario file must be a MAT-file saved from the **Driving Scenario Designer** app. If the **Source of ego vehicle** parameter is set to `Scenario`, then the scenario must contain an ego vehicle. Otherwise, the block returns an error during simulation.

If the specified scenario file contains sensors, the block ignores them. To include sensors from the scenario in your model, see “Tips” on page 2-83.

The default scenario file shows an ego vehicle traveling north on a straight, two-lane road, with another vehicle traveling south in the opposite lane.

To add a scenario file to the MATLAB search path, use the `addpath` function. For example, this code adds the set of folders containing prebuilt Euro NCAP scenarios to the MATLAB search path.

```
path = fullfile(matlabroot,'toolbox','driving','drivingdata', ...
    'PrebuiltScenarios','EuroNCAP');
addpath(genpath(path))
```

In the **Driving Scenario Designer file name** parameter, you can then specify the name of any scenario located in these folders, without having to specify the full file path. For example: `AEB_PedestrianChild_Nearside_50width.mat`.

When you are done using the scenario in your models, you can remove any added folders from the MATLAB search path by using the `rmpath` function.

```
rmpath(genpath(path))
```

Dependencies

To enable this parameter, set **Source of driving scenario** to `From file`.

`MATLAB or model workspace variable name` — Scenario variable name

`scenario` (default) | `drivingScenario` object variable name

Scenario variable name, specified as the name of a MATLAB or model workspace variable that contains a valid `drivingScenario` object. If a scenario variable with the same name appears in both the MATLAB and model workspace, the block uses the variable defined in the model workspace.

If the **Source of ego vehicle** parameter is set to `Scenario`, then the `drivingScenario` object must contain an ego vehicle. To designate which actor in the object is the ego vehicle, in the **Ego vehicle ActorID** parameter, specify the `ActorID` property value of that actor.

When connecting the **Actors** output port to Radar Detection Generator and Vision Detection Generator blocks, update these blocks to obtain the actor profiles directly from the `drivingScenario` object. On the **Actor Profiles** tab of each block, set the **Select method to specify actor profiles** parameter to `MATLAB expression`. Then, set the **MATLAB expression for actor profiles** parameter to call the `actorProfiles` function on the object. For example: `actorProfiles(scenario)`.

When connecting the **Actors** output port to Lidar Point Cloud Generator blocks, leave the **Source of actor profiles** parameter in these blocks set to the default `From Scenario Reader block`. With this option selected, the lidar sensors obtain the actor profiles directly from the scenario read by the Scenario Reader block.

The default variable name, `scenario`, is the default name of `drivingScenario` objects produced by the MATLAB functions that are exported from the **Driving Scenario Designer** app. By default, this variable is not included in the MATLAB or model workspace.

Dependencies

To enable this parameter, set **Source of driving scenario** to `From workspace`.

`Coordinate system of actors output` — Coordinate system of actors output

`Vehicle coordinates` (default) | `World coordinates`

Coordinate system of the output actors, specified as one of these values:

- `Vehicle coordinates` — Coordinates are defined with respect to the ego vehicle. Select this value when your scenario has only one ego vehicle.
- `World coordinates` — Coordinates are defined with respect to the driving scenario. Select this value in multi-agent scenarios that contain more than one ego vehicle. If you select this value, model visualization using the **Bird's-Eye Scope** is not supported.

For more details on the vehicle and world coordinate systems, see "Coordinate Systems in Automated Driving Toolbox".

`Source of ego vehicle` — Source of ego vehicle

`Scenario` (default) | `Input port`

Source of ego vehicle, specified as one of these options:

- `Scenario` — Use the ego vehicle defined in the scenario that is specified by the **Driving Scenario Designer file name** or **MATLAB or model workspace variable name** parameter. The pose of the ego vehicle is excluded from the **Actors** output port. Actor positions are in vehicle coordinates, meaning that they are relative to the world coordinate position of the ego vehicle in the scenario.

  Select this option to test open-loop ADAS algorithms, where the ego vehicle behavior is predefined and does not change as the scenario advances. For an example, see "Test Open-Loop ADAS Algorithm Using Driving Scenario".
- `Input port` — Specify the ego vehicle by using the **Ego Vehicle** input port. The pose of the ego vehicle is not included in the **Actors** output port.

  With this option, the ego vehicle in your model must include a starting position that is in world coordinates. All other actor poses are in vehicle coordinates and are positioned relative to the ego vehicle. For an example of an ego vehicle with defined position information, see "Lane Keeping Assist with Lane Detection". When defining the starting position of the ego vehicle, consider using the position that is already defined in the scenario. By using this position, if you set **Source of ego vehicle** to `Scenario` and then back to `Input port`, you do not have to manually change the starting position.

  Select this option to test closed-loop ADAS algorithms, where the ego vehicle reacts to changes as the scenario advances. For an example, see "Test Closed-Loop ADAS Algorithm Using Driving Scenario".

Dependencies

To enable this parameter, set **Coordinate system of actors output** to `Vehicle coordinates`.

`Ego vehicle ActorID` — Actor ID of ego vehicle

`1` (default) | positive integer

Actor ID of ego vehicle, specified as a positive integer. Use this parameter to simulate using the ego vehicle that is read from a `drivingScenario` object.

- When **Source of ego vehicle** is set to `Scenario`, set this parameter to an `ActorID` value that is stored in the `Actors` property of the specified `drivingScenario` object. To check valid `ActorID` values, use this syntax, where `scenario` is the name of the `drivingScenario` variable name.

  ```
  actorIDs = [scenario.Actors.ActorID]
  ```

- When **Source of ego vehicle** is set to `Input Port`, you must set this parameter to the `ActorID` value at the **Ego Vehicle** input port of the block.

Dependencies

To enable this parameter, set these parameters in this order:

1. Set **Source of driving scenario** to `From workspace`.
2. Set **Coordinate system of actors output** to `Vehicle coordinates`.

`Output ego vehicle pose` — Output pose of ego vehicle

`off` (default) | `on`

Select this parameter to output the pose of the ego vehicle at the **Ego Vehicle** port.

Dependencies

To enable this parameter, set **Coordinate system of actors output** to `Vehicle coordinates` and **Source of ego vehicle** to `Scenario`.

`Ego vehicle follows ground` — Orient ego vehicle to follow road surface

`off` (default) | `on`

Select this parameter to orient the ego vehicle to follow the elevation of the road surface. The block updates the elevation, roll, pitch, and yaw of the ego vehicle and outputs actors and lane boundaries relative to the updated ego vehicle coordinates. The block does not update the velocity or angular velocity of the ego vehicle.

Use this parameter in closed-loop simulations where the elevation of the road network varies.

Note At the junctions of roads that have different elevations and banking angles, the updated ego vehicle values might not be accurate.

In open-loop simulations, where **Source of ego vehicle** is set to `Scenario`, the ego vehicle follows the elevation specified in the driving scenario.

Dependencies

To enable this parameter, set **Coordinate system of actors output** to `Vehicle coordinates` and **Source of ego vehicle** to `Input port`.

`Sample time (s)` — Sample time of simulation

`0.1` (default) | positive real scalar

Sample time of simulation, in seconds, specified as a positive real scalar. Inherited and continuous sample times are not supported. This sample time is separate from the sample times that the **Driving Scenario Designer** app and `drivingScenario` object use for simulations.

`Lane boundaries to output` — Lane boundaries to output

`None` (default) | `Ego vehicle lane boundaries` | `All lane boundaries`

Lane boundaries to output, specified as one of these options:

- `None` — Do not output any lane boundaries.

- `Ego vehicle lane boundaries` — Output the left and right lane boundaries of the ego vehicle.
- `All lane boundaries` — Output all lane boundaries of the road on which the ego vehicle is traveling.

If you select `Ego vehicle lane boundaries` or `All lane boundaries`, then the block returns the lane boundaries in the **Lane Boundaries** output port.

Dependencies

To enable this parameter, set **Coordinate system of actors output** to `Vehicle coordinates`.

`Distances from ego vehicle for computing boundaries (m)` — Distances from ego vehicle at which to compute lane boundaries

`linspace(-150,150,101)` (default) | *N*-element real-valued vector

Distances from the ego vehicle at which to compute the lane boundaries, specified as an *N*-element real-valued vector. *N* is the number of distance values. When detecting lanes from rear-facing cameras, specify negative distances. When detecting lanes from front-facing cameras, specify positive distances. Units are in meters.

By default, the block computes 101 lane boundaries over the range from 150 meters behind the ego vehicle to 150 meters ahead of the ego vehicle. These distances are linearly spaced 3 meters apart.

Example: `1:0.1:10` computes a lane boundary every 0.1 meters over the range from 1 to 10 meters ahead of the ego vehicle.

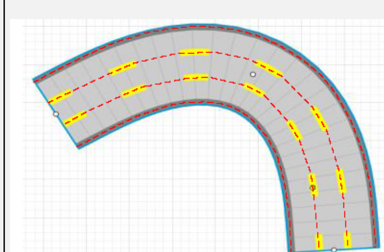
Dependencies

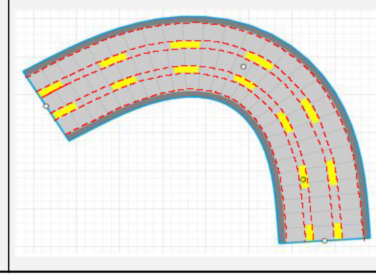
To enable this parameter, set **Lane boundaries to output** to `Ego vehicle lane boundaries` or `All lane boundaries`.

`Location of boundaries on lane markings` — Lane boundary location

`Center of lane markings` (default) | `Inner edge of lane markings`

Lane boundary location on the lane markings, specified as one of the options in this table.

Lane Boundary Location	Description	Example
`Center of lane markings`	Lane boundaries are centered on the lane markings.	A three-lane road has four lane boundaries: one per lane marking. 

Lane Boundary Location	Description	Example
`Inner edge of lane markings`	Lane boundaries are placed at the inner edges of the lane markings.	A three-lane road has six lane boundaries: two per lane. 

Dependencies

To enable this parameter, set **Lane boundaries to output** to `Ego vehicle lane boundaries` or `All lane boundaries`.

`Source of actors bus name` — Source of name for actor poses bus

`Auto` (default) | `Property`

Source of the name for the actor poses bus returned in the **Actors** output port, specified as one of these options:

- `Auto` — The block automatically creates an actor poses bus name.
- `Property` — Specify the actor poses bus name by using the **Actors bus name** parameter.

`Actors bus name` — Name of actor poses bus

valid bus name

Name of the actor poses bus returned in the **Actors** output port, specified as a valid bus name.

Dependencies

To enable this parameter, set **Source of actors bus name** to `Property`.

`Source of lane boundaries bus name` — Source of name for lane boundaries bus

`Auto` (default) | `Property`

Source of the name for the lane boundaries bus returned in the **Lane Boundaries** output port, specified as one of these options:

- `Auto` — The block automatically creates a lane boundaries bus name.
- `Property` — Specify the lane boundaries bus name by using the **Lane boundaries bus name** parameter.

Dependencies

To enable this parameter, set **Lane boundaries to output** to `Ego vehicle lane boundaries` or `All lane boundaries`.

`Lane boundaries bus name` — Name of lane boundaries bus

valid bus name

Name of the lane boundaries bus returned in the **Lane Boundaries** output port, specified as a valid bus name.

Dependencies

To enable this parameter:

1. Set **Lane boundaries to output** to `Ego vehicle lane boundaries` or `All lane boundaries`.
2. Set **Source of lane boundaries bus name** to `Property`.

`Source of ego vehicle bus name` — Source of name for ego vehicle pose bus
`Auto` (default) | `Property`

Source of the name for the ego vehicle pose bus returned in the **Ego Vehicle** output port, specified as one of these options:

- `Auto` — The block automatically creates an ego vehicle pose bus name.
- `Property` — Specify the ego vehicle pose bus name by using the **Ego vehicle bus name** parameter.

Dependencies

To enable this parameter, select the **Output ego vehicle pose** parameter.

`Ego vehicle bus name` — Name of ego vehicle pose bus
valid bus name

Name of the ego vehicle pose bus returned in the **Ego Vehicle** output port, specified as a valid bus name.

Dependencies

To enable this parameter, select the **Output ego vehicle pose** parameter and set **Source of ego vehicle bus name** to `Property`.

`Show coordinate labels` — Display coordinate system of inputs and outputs
`on` (default) | `off`

Select this parameter to display the coordinate system of block inputs and outputs on the Scenario Reader block in the block diagram.

- The **Ego Vehicle** input and output are always in world coordinates.
- The **Lane Boundaries** output is always in vehicle coordinates.
- You can return the **Actors** output in either vehicle or world coordinates, depending on the **Coordinate system of actors output** parameter selection.

`Simulate using` — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.

- `Code generation` — Simulate the model using generated C/C++ code. The first time you run a simulation, Simulink generates C/C++ code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.

Tips

- For best results, use only one active Scenario Reader block per model. To use multiple Scenario Reader blocks in one model, switch between the blocks by specifying them in a variant subsystem.
- To test your algorithm on variations of a driving scenario, you can update the scenario between simulations.
  - If the source of the scenario is a scenario file, open the scenario file in the **Driving Scenario Designer** app, update the parameters, and resave the file.
  - If the source of the scenario is a `drivingScenario` object, update the object in the MATLAB or model workspace. Alternatively, import the object into the app, modify the scenario in the app, and then generate a new object from the app. For more details, see “Create Driving Scenario Variations Programmatically”.
- To switch between scenarios with different parameter settings, you can use Simulink Test™ software. For an example, see “Automate Testing for Highway Lane Following”.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- When a model is in rapid accelerator mode, the Scenario Reader block does not automatically regenerate code based on changes made to the driving scenario between simulations. To regenerate these changes, manually delete the Simulink project folder, `slprj`, that was generated from the previous simulation. Then, rerun the simulation. Alternatively, either change modes or disable code generation by setting the **Simulate using** parameter to `Interpreted execution`.
- The **Driving Scenario Designer file name** and **MATLAB or model workspace variable name** parameters are character vectors. The limitations described in “Encoding of Characters in Code Generation” (Simulink) apply to these parameters.

See Also

Apps
Bird's-Eye Scope | Driving Scenario Designer

Blocks
Cuboid To 3D Simulation | Detection Concatenation | Lidar Point Cloud Generator | Multi-Object Tracker | Radar Detection Generator | Vehicle To World | Vision Detection Generator | World To Vehicle

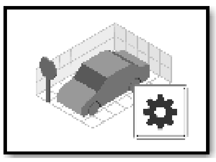
Topics
“Coordinate Systems in Automated Driving Toolbox”
“Create Nonvirtual Buses” (Simulink)

Introduced in R2019a

Simulation 3D Scene Configuration

Scene configuration for 3D simulation environment

Library: Automated Driving Toolbox / Simulation 3D
Vehicle Dynamics Blockset / Vehicle Scenarios / Sim3D / Sim3D Core



Description

The Simulation 3D Scene Configuration block implements a 3D simulation environment that is rendered by using the Unreal Engine from Epic Games. Automated Driving Toolbox integrates the 3D simulation environment with Simulink so that you can query the world around the vehicle and virtually test perception, control, and planning algorithms.

You can simulate from a set of prebuilt scenes or from your own custom scenes. Scene customization requires the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package. For more details, see "Customize Unreal Engine Scenes for Automated Driving".

Note The Simulation 3D Scene Configuration block must execute after blocks that send data to the 3D environment and before blocks that receive data from the 3D environment. To verify the execution order of such blocks, right-click the blocks and select **Properties**. Then, on the **General** tab, confirm these **Priority** settings:

- For blocks that send data to the 3D environment, such as Simulation 3D Vehicle with Ground Following blocks, **Priority** must be set to `-1`. That way, these blocks prepare their data before the 3D environment receives it.
- For the Simulation 3D Scene Configuration block in your model, **Priority** must be set to `0`.
- For blocks that receive data from the 3D environment, such as Simulation 3D Camera blocks, **Priority** must be set to `1`. That way, the 3D environment can prepare the data before these blocks receive it.

For more information about execution order, see "How Unreal Engine Simulation for Automated Driving Works".

Parameters

Scene Selection

`Scene source` — Source of scene

`Default Scenes` (default) | `Unreal Executable` | `Unreal Editor`

Source of the scene in which to simulate, specified as one of the options in the table.

Option	Description
`Default Scenes`	Simulate in one of the default, prebuilt scenes specified in the **Scene name** parameter.
`Unreal Executable`	Simulate in a scene that is part of an Unreal Engine executable file. Specify the executable file in the **File name** parameter. Specify the scene in the **Scene** parameter. Select this option to simulate in custom scenes that have been packaged into an executable for faster simulation.
`Unreal Editor`	Simulate in a scene that is part of an Unreal Engine project (`.uproject`) file and is open in the Unreal® Editor. Specify the project file in the **Project** parameter. Select this option when developing custom scenes. By clicking **Open Unreal Editor**, you can co-simulate within Simulink and the Unreal Editor and modify your scenes based on the simulation results.

Scene name — Name of prebuilt 3D scene

`Straight road` (default) | `Curved road` | `Parking lot` | `Double lane change` | `Open surface` | `US city block` | `US highway` | `Virtual Mcity` | `Large parking lot`

Name of the prebuilt 3D scene in which to simulate, specified as one of these options. For details about a scene, see its listed corresponding reference page.

- `Straight road` — **Straight Road**
- `Curved road` — **Curved Road**
- `Parking lot` — **Parking Lot**
- `Double lane change` — **Double Lane Change**
- `Open surface` — **Open Surface**
- `US city block` — **US City Block**
- `US highway` — **US Highway**
- `Virtual Mcity` — **Virtual Mcity**
- `Large parking lot` — **Large Parking Lot**

The Automated Driving Toolbox Interface for Unreal Engine 4 Projects contains customizable versions of these scenes. For details about customizing scenes, see "Customize Unreal Engine Scenes for Automated Driving".

Dependencies

To enable this parameter, set **Scene source** to `Default Scenes`.

File name — Name of Unreal Engine executable file

`VehicleSimulation.exe` (default) | valid executable file name

Name of the Unreal Engine executable file, specified as a valid executable file name. You can either browse for the file or specify the full path to the file, using backslashes. To specify a scene from this file to simulate in, use the **Scene** parameter.

By default, **File name** is set to `VehicleSimulation.exe`, which is on the MATLAB search path.

Example: `C:\Local\WindowsNoEditor\AutoVrtlEnv.exe`

Dependencies

To enable this parameter, set **Scene source** to `Unreal Executable`.

Scene — Name of scene from executable file

`/Game/Maps/HwStrght` (default) | path to valid scene name

Name of a scene from the executable file specified by the **File name** parameter, specified as a path to a valid scene name.

When you package scenes from an Unreal Engine project into an executable file, the Unreal Editor saves the scenes to an internal folder within the executable file. This folder is located at the path `/Game/Maps`. Therefore, you must prepend `/Game/Maps` to the scene name. You must specify this path using forward slashes. For the file name, do not specify the `.umap` extension. For example, if the scene from the executable in which you want to simulate is named `myScene.umap`, specify **Scene** as `/Game/Maps/myScene`.

Alternatively, you can browse for the scene in the corresponding Unreal Engine project. These scenes are typically saved to the `Content/Maps` subfolder of the project. This subfolder contains all the scenes in your project. The scenes have the extension `.umap`. Select one of the scenes that you packaged into the executable file specified by the **File name** parameter. Use backward slashes and specify the `.umap` extension for the scene.

By default, **Scene** is set to `/Game/Maps/HwStrght`, which is a scene from the default `VehicleSimulation.exe` executable file specified by the **File name** parameter. This scene corresponds to the prebuilt **Straight Road** scene.

Example: `/Game/Maps/scene1`

Example: `C:\Local\myProject\Content\Maps\scene1.umap`

Dependencies

To enable this parameter, set **Scene source** to `Unreal Executable`.

`Project` — Name of Unreal Engine project file

valid project file name

Name of the Unreal Engine project file, specified as a valid project file name. You can either browse for the file or specify the full path to the file, using backslashes. The file must contain no spaces. To simulate scenes from this project in the Unreal Editor, click **Open Unreal Editor**. If you have an Unreal Editor session open already, then this button is disabled.

To run the simulation, in Simulink, click **Run**. Before you click **Play** in the Unreal Editor, wait until the Diagnostic Viewer window displays this confirmation 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 the scene actors, including the vehicles and cameras, in the Unreal Engine 3D environment. If you click **Play** before the Diagnostic Viewer window displays this confirmation message, Simulink might not instantiate the actors in the Unreal Editor.

Dependencies

To enable this parameter, set **Scene source** to `Unreal Editor`.

Scene Parameters

`Scene view` — Configure placement of virtual camera that displays scene
`Scene Origin` (default) | vehicle name

Configure the placement of the virtual camera that displays the scene during simulation.

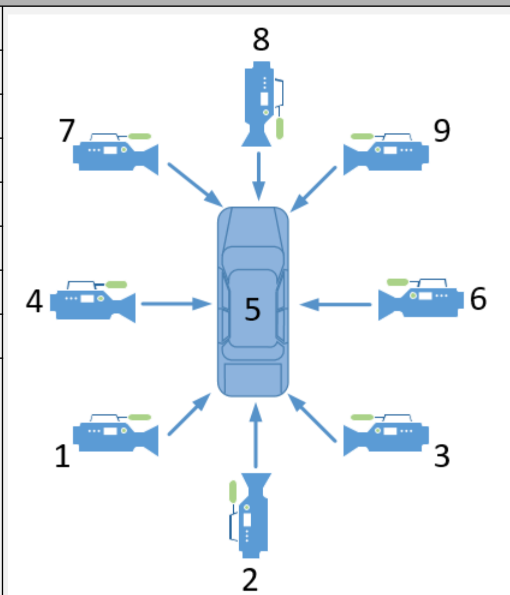
- If your model contains no Simulation 3D Vehicle with Ground Following blocks, then during simulation, you view the scene from a camera positioned at the scene origin.
- If your model contains at least one vehicle block, then by default, you view the scene from behind the first vehicle that was placed in your model. To change the view to a different vehicle, set **Scene view** to the name of that vehicle. The **Scene view** parameter list is populated with all the **Name** parameter values of the vehicle blocks contained in your model.

If you add a Simulation 3D Scene Configuration block to your model before adding any vehicle blocks, the virtual camera remains positioned at the scene. To reposition the camera to follow a vehicle, update this parameter.

When **Scene view** is set to a vehicle name, during simulation, you can change the location of the camera around the vehicle.

To smoothly change the camera views, use these key commands.

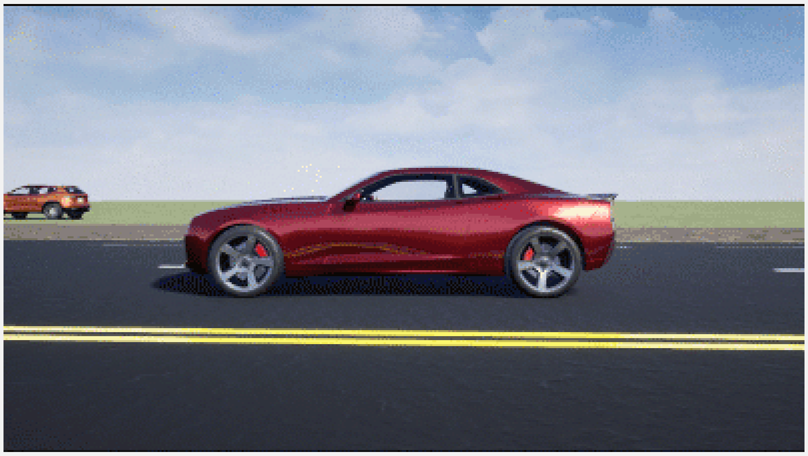
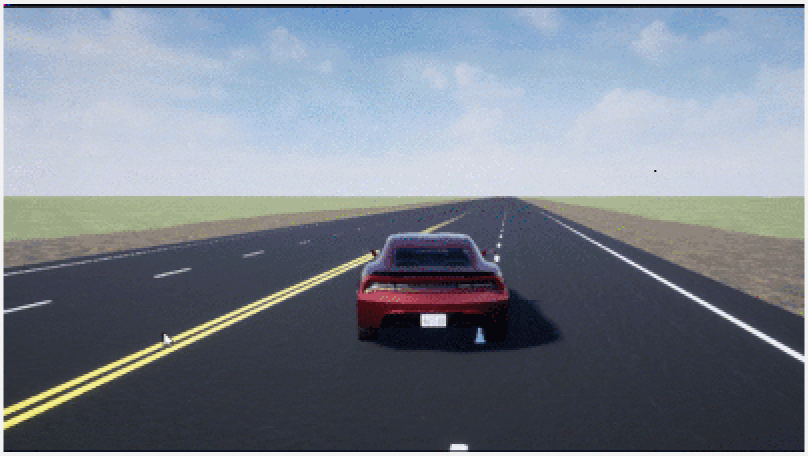
Key	Camera View
1	Back left
2	Back
3	Back right
4	Left
5	Internal
6	Right
7	Front left
8	Front
9	Front right

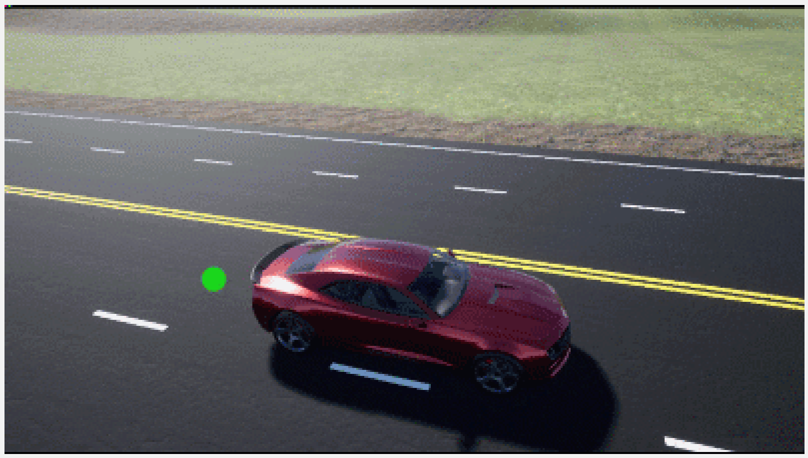


Key	Camera View	
0	Overhead	**View Animated GIF** 

For additional camera controls, use these key commands.

Key	Camera Control
Tab	Cycle the view between all vehicles in the scene. **View Animated GIF** 

Key	Camera Control
Mouse scroll wheel	Control the camera distance from the vehicle. **View Animated GIF** 
L	Toggle a camera lag effect on or off. When you enable the lag effect, the camera view includes: • Position lag, based on the vehicle translational acceleration • Rotation lag, based on the vehicle rotational velocity This lag enables improved visualization of overall vehicle acceleration and rotation. **View Animated GIF** 

Key	Camera Control
F	Toggle the free camera mode on or off. When you enable the free camera mode, you can use the mouse to change the pitch and yaw of the camera. This mode enables you to orbit the camera around the vehicle. **View Animated GIF** 

`Sample time` — Sample time of visualization engine
`1/60` (default) | scalar greater than or equal to 0.01

Sample time, T_s, of the visualization engine, specified as a scalar greater than or equal to 0.01. Units are in seconds.

The graphics frame rate of the visualization engine is the inverse of the sample time. For example, if **Sample time** is `1/60`, then the visualization engine solver tries to achieve a frame rate of 60 frames per second. However, the real-time graphics frame rate is often lower due to factors such as graphics card performance and model complexity.

By default, blocks that receive data from the visualization engine, such as Simulation 3D Camera blocks, inherit this sample rate.

`Display 3D simulation window` — Unreal Engine visualization
on (default) | `off`

Select whether to run simulations in the 3D visualization environment without visualizing the results, that is, in headless mode.

Consider running in headless mode in these cases:

- You want to run multiple 3D simulations in parallel to test models in different Unreal Engine scenarios.
- You want to capture sensor data to analyze in MATLAB but do not need to watch the visualization.

Dependencies

To enable this parameter, set **Scene source** to `Default Scenes` or `Unreal Executable`.

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

Topics

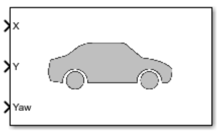
"Unreal Engine Simulation for Automated Driving"
"Unreal Engine Simulation Environment Requirements and Limitations"
"How Unreal Engine Simulation for Automated Driving Works"
"Customize Unreal Engine Scenes for Automated Driving"

Introduced in R2019b

Simulation 3D Vehicle with Ground Following

Implement vehicle that follows ground in 3D environment

Library: Automated Driving Toolbox / Simulation 3D
Vehicle Dynamics Blockset / Vehicle Scenarios / Sim3D / Sim3D Vehicle / Components



Description

The Simulation 3D Vehicle with Ground Following block implements a vehicle with four wheels in a 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The block uses the input (*X*, *Y*) position and yaw angle of the vehicle to adjust the elevation, roll angle, and pitch angle of the vehicle so that it follows the ground terrain. The block determines the vehicle velocity and heading and adjusts the steering angle and rotation for each wheel. Use this block for automated driving applications.

To use this block, ensure that the Simulation 3D Scene Configuration block is in your model. If you set the **Sample time** parameter of the Simulation 3D Vehicle with Ground Following block to `-1`, the block inherits the sample time specified in the Simulation 3D Scene Configuration block.

The block input uses the vehicle *Z*-up right-handed (RH) Cartesian coordinate system defined in SAE J670 [1] and ISO 8855 [2]. The coordinate system is inertial and initially aligned with the vehicle geometric center:

- The *X*-axis is along the longitudinal axis of the vehicle and points forward.
- The *Y*-axis is along the lateral axis of the vehicle and points to the left.
- The *Z*-axis points upward.

The yaw, pitch, and roll angles of the *Z*-axis, *Y*-axis, and *X*-axis, respectively, are positive in the clockwise directions, when looking in the positive directions of these axes. Vehicles are placed in the world coordinate system of the scenes. For more details, see “Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox”.

Note The Simulation 3D Vehicle with Ground Following block must execute before the Simulation 3D Scene Configuration block. That way, the Simulation 3D Vehicle with Ground Following block prepares the signal data before the Unreal Engine 3D visualization environment receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Vehicle with Ground Following — `-1`

For more information about execution order, see “How Unreal Engine Simulation for Automated Driving Works”.

Limitations

- The **Bird's-Eye Scope** is unable to find ground truth signals, such as roads, lanes, and actors, from the Simulation 3D Scene Configuration block.

Ports

Input

`X` — Longitudinal position of vehicle
scalar

Longitudinal position of the vehicle along the *X*-axis of the scene. **X** is in the inertial *Z*-up coordinate system. Units are in meters.

The *X* value of the **Initial position [X, Y, Z] (m)** parameter must match the value of this port at the start of simulation.

To specify multiple positions at port **X** along an entire vehicle path, first define a time series of *X* waypoints in MATLAB. Then, feed these waypoints to **X** by using a From Workspace block. To learn how to select and specify waypoints, see the "Select Waypoints for Unreal Engine Simulation" example.

`Y` — Lateral position of vehicle
scalar

Lateral position of the vehicle along the *Y*-axis of the scene. **Y** is in the inertial *Z*-up coordinate system. Units are in meters.

The *Y* value of the **Initial position [X, Y, Z] (m)** parameter must match the value of this port at the start of simulation.

To specify multiple positions at port **Y** along an entire vehicle path, first define a time series of *Y* waypoints in MATLAB. Then, feed these waypoints to **Y** by using a From Workspace block. To learn how to select and specify waypoints, see the "Select Waypoints for Unreal Engine Simulation" example.

`Yaw` — Yaw orientation angle of vehicle
scalar

Yaw orientation angle of the vehicle along the *Z*-axis of the scene. **Yaw** is in the *Z*-up coordinate system. Units are in degrees.

The yaw value of the **Initial rotation [Roll, Pitch, Yaw] (deg)** parameter must match the value of this port at the start of simulation.

To specify multiple orientation angles at port **Yaw** along an entire vehicle path, first define a time series of yaw waypoints in MATLAB. Then, feed these waypoints to **Yaw** by using a From Workspace block. To learn how to select and specify waypoints, see the "Select Waypoints for Unreal Engine Simulation" example.

Output

`Location` — Location of vehicle
real-valued 1-by-3 vector

(X, Y, Z) location of the vehicle in the scene, returned as a real-valued 1-by-3 vector. This location is based on the vehicle origin, which is on the ground, at the geometric center of the vehicle. **Location** values are in the inertial Z-up world coordinate system. Units are in meters.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

`Orientation` — Orientation of vehicle

real-valued 1-by-3 vector

Yaw, pitch, and roll orientation angles of the vehicle about the Z-axis, Y-axis, and X-axes of the scene, respectively, returned as a real-valued 1-by-3 vector. This orientation is based on the vehicle origin, which is on the ground, at the geometric center of the vehicle. **Orientation** values are in the inertial Z-up coordinate system. Units are in radians.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

Parameters

Vehicle Parameters

Type — Type of vehicle

`Muscle car` (default) | `Sedan` | `Sport utility vehicle` | `Small pickup truck` | `Hatchback` | `Box truck`

Select the type of vehicle. To obtain the dimensions of each vehicle type, see these reference pages:

- `Muscle car` — **Muscle Car**
- `Sedan` — **Sedan**
- `Sport utility vehicle` — **Sport Utility Vehicle**
- `Small pickup truck` — **Small Pickup Truck**
- `Hatchback` — **Hatchback**
- `Box truck` —**Box Truck**

`Color` — Color of vehicle

`Red` (default) | `Orange` | `Yellow` | `Green` | `Blue` | `Black` | `White` | `Silver`

Select the color of the vehicle.

`Initial position [X, Y, Z] (m)` — Initial vehicle position

`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Initial vehicle position along the X-axis, Y-axis, and Z-axis of the scene. This position is in the inertial Z-up coordinate system. Units are in meters.

Set the X and Y values of this parameter to match the **X** and **Y** input port values at the start of simulation.

Initial rotation [Roll, Pitch, Yaw] (deg) — Initial angle of vehicle rotation
[0, 0, 0] (default) | real-valued 1-by-3 vector

Initial angle of vehicle rotation. The angle of rotation is defined by the roll, pitch, and yaw of the vehicle. Units are in degrees.

Set the yaw value of this parameter to match the **Yaw** input port value at the start of simulation.

Name — Name of vehicle
SimulinkVehicle1 (default) | vehicle name

Name of vehicle. By default, when you use the block in your model, the block sets the **Name** parameter to SimulinkVehicle*X*. The value of *X* depends on the number of Simulation 3D Vehicle with Ground Following blocks that you have in your model.

The vehicle name appears as a selection in the **Parent name** parameter of any Automated Driving Toolbox Simulation 3D sensor blocks within the same model as the vehicle. With the **Parent name** parameter, you can select the vehicle on which to mount the sensor.

Sample time — Sample time
-1 (default) | positive scalar

Sample time, T_s, in seconds. The graphics frame rate is the inverse of the sample time.

If you set the sample time to -1, the block uses the sample time specified in the Simulation 3D Scene Configuration block.

Ground Truth

Output location (m) and orientation (rad) — Output location and orientation of vehicle
off (default) | on

Select this parameter to output the location and orientation of the vehicle at the **Location** and **Orientation** ports, respectively.

References

[1] Vehicle Dynamics Standards Committee. *Vehicle Dynamics Terminology*. SAE J670. Warrendale, PA: Society of Automotive Engineers, 2008.

[2] Technical Committee. *Road vehicles — Vehicle dynamics and road-holding ability — Vocabulary*. ISO 8855:2011. Geneva, Switzerland: International Organization for Standardization, 2011.

See Also

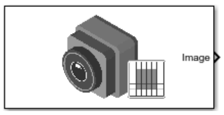
Simulation 3D Camera | Simulation 3D Fisheye Camera | Simulation 3D Lidar | Simulation 3D Probabilistic Radar | Simulation 3D Scene Configuration | Simulation 3D Vision Detection Generator

Topics
"How Unreal Engine Simulation for Automated Driving Works"
"Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox"

Introduced in R2019b

Simulation 3D Camera

Camera sensor model with lens in 3D simulation environment
Library: Automated Driving Toolbox / Simulation 3D

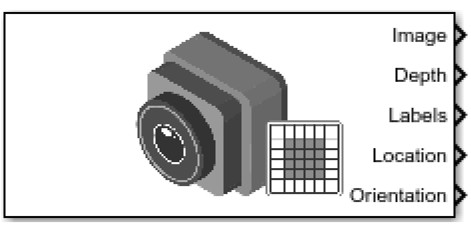


Description

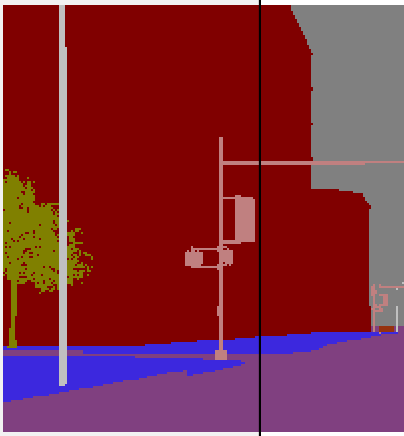
The Simulation 3D Camera block provides an interface to a camera with a lens in a 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The sensor is based on the ideal pinhole camera model, with a lens added to represent a full camera model, including lens distortion. For more details, see “Algorithms” on page 2-108.

If you set **Sample time** to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block. To use this sensor, you must include a Simulation 3D Scene Configuration block in your model.

The block outputs images captured by the camera during simulation. You can use these images to visualize and verify your driving algorithms. In addition, on the **Ground Truth** tab, you can select options to output the ground truth data for developing depth estimation and semantic segmentation algorithms. You can also output the location and orientation of the camera in the world coordinate system of the scene. The image shows the block with all ports enabled.



The table summarizes the ports and how to enable them.

Port	Description	Parameter for Enabling Port	Sample Visualization
Image	Outputs an RGB image captured by the camera	n/a	
Depth	Outputs a depth map with values from 0 m to 1000 meters	**Output depth**	
Labels	Outputs a semantic segmentation map of label IDs that correspond to objects in the scene	**Output semantic segmentation**	
Location	Outputs the location of the camera in the world coordinate system	**Output location (m) and orientation (rad)**	n/a
Orientation	Outputs the orientation of the camera in the world coordinate system	**Output location (m) and orientation (rad)**	n/a

Note The Simulation 3D Scene Configuration block must execute before the Simulation 3D Camera block. That way, the Unreal Engine 3D visualization environment prepares the data before the Simulation 3D Camera block receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Camera — `1`

For more information about execution order, see "How Unreal Engine Simulation for Automated Driving Works".

Ports

Output

`Image` — 3D output camera image

m-by-*n*-by-3 array of RGB triplet values

3D output camera image, returned as an *m*-by-*n*-by-3 array of RGB triplet values. *m* is the vertical resolution of the image, and *n* is the horizontal resolution of the image.

Data Types: `int8` | `uint8`

`Depth` — Object depth from 0 m to 1000 m

m-by-*n* array of object depths

Object depth for each pixel in the image, output as an *m*-by-*n* array. *m* is the vertical resolution of the image, and *n* is the horizontal resolution of the image. Depth is in the range from 0 to 1000 meters.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output depth**.

Data Types: `double`

`Labels` — Label identifiers

m-by-*n* array of label identifiers

Label identifier for each pixel in the image, output as an *m*-by-*n* array. *m* is the vertical resolution of the image, and *n* is the horizontal resolution of the image.

The table shows the object IDs used in the default scenes that are selectable from the Simulation 3D Scene Configuration block. If you are using a custom scene, in the Unreal Editor, you can assign new object types to unused IDs. If a scene contains an object that does not have an assigned ID, that object is assigned an ID of `0`. The detection of lane markings is not supported.

ID	Type
`0`	None/default
`1`	Building
`2`	*Not used*
`3`	Other
`4`	*Not used*

ID	Type
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*
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

ID	Type
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*

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output semantic segmentation**.

Data Types: `uint8`

`Location` — Sensor location

real-valued 1-by-3 vector

Sensor location along the *X*-axis, *Y*-axis, and *Z*-axis of the scene. The **Location** values are in the world coordinates of the scene. In this coordinate system, the *Z*-axis points up from the ground. Units are in meters.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

`Orientation` — Sensor orientation

real-valued 1-by-3 vector

Roll, pitch, and yaw sensor orientation about the *X*-axis, *Y*-axis, and *Z*-axis of the scene. The **Orientation** values are in the world coordinates of the scene. These values are positive in the clockwise direction when looking in the positive directions of these axes. Units are in radians.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

Parameters

Mounting

`Sensor identifier` — Unique sensor identifier

`1` (default) | positive integer

Unique sensor identifier, specified as a positive integer. In a multisensor system, the sensor identifier distinguishes between sensors. When you add a new sensor block to your model, the **Sensor**

identifier of that block is $N + 1$. N is the highest **Sensor identifier** value among existing sensor blocks in the model.

Example: 2

`Parent name` — Name of parent to which sensor is mounted
`Scene Origin` (default) | vehicle name

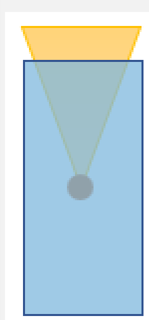
Name of the parent to which the sensor is mounted, specified as `Scene Origin` or as the name of a vehicle in your model. The vehicle names that you can select correspond to the **Name** parameters of the Simulation 3D Vehicle with Ground Following blocks in your model. If you select `Scene Origin`, the block places a sensor at the scene origin.

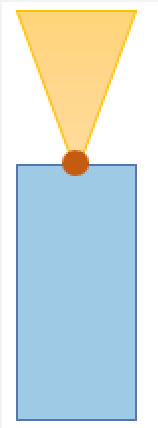
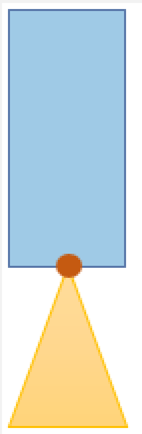
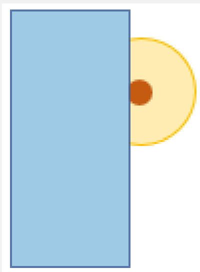
Example: `SimulinkVehicle1`

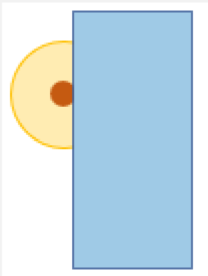
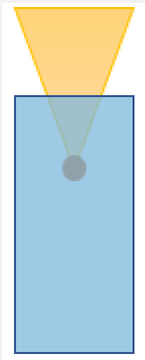
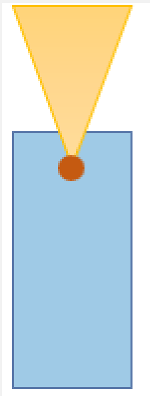
`Mounting location` — Sensor mounting location
`Origin` (default) | `Front bumper` | `Rear bumper` | `Right mirror` | `Left mirror` | `Rearview mirror` | `Hood center` | `Roof center`

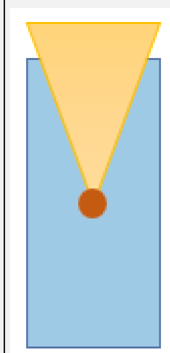
Sensor mounting location.

- When **Parent name** is `Scene Origin`, the block mounts the sensor to the origin of the scene. You can set the **Mounting location** to `Origin` only. During simulation, the sensor remains stationary.
- When **Parent name** is the name of a vehicle (for example, `SimulinkVehicle1`) the block mounts the sensor to one of the predefined mounting locations described in the table. During simulation, the sensor travels with the vehicle.

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Origin`	Forward-facing sensor mounted to the vehicle origin, which is on the ground, at the geometric center of the vehicle (see “Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox”) 	[0, 0, 0]

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Front bumper`	Forward-facing sensor mounted to the front bumper 	[0, 0, 0]
`Rear bumper`	Backward-facing sensor mounted to the rear bumper 	[0, 0, 180]
`Right mirror`	Downward-facing sensor mounted to the right side-view mirror 	[0, –90, 0]

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Left mirror`	Downward-facing sensor mounted to the left side-view mirror 	[0, –90, 0]
`Rearview mirror`	Forward-facing sensor mounted to the rearview mirror, inside the vehicle 	[0, 0, 0]
`Hood center`	Forward-facing sensor mounted to the center of the hood 	[0, 0, 0]

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Roof center`	Forward-facing sensor mounted to the center of the roof 	[0, 0, 0]

Roll, pitch, and yaw are clockwise-positive when looking in the positive direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. When looking at a vehicle from the top down, the yaw angle (that is, the orientation angle) is counterclockwise-positive, because you are looking in the negative direction of the axis.

The (*X*, *Y*, *Z*) mounting location of the sensor relative to the vehicle depends on the vehicle type. To specify the vehicle type, use the **Type** parameter of the Simulation 3D Vehicle with Ground Following block to which you are mounting the sensor. To obtain the (*X*, *Y*, *Z*) mounting locations for a vehicle type, see the reference page for that vehicle.

To determine the location of the sensor in world coordinates, open the sensor block. Then, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)** and inspect the data from the **Location** output port.

`Specify offset` — Specify offset from mounting location

`off` (default) | `on`

Select this parameter to specify an offset from the mounting location by using the **Relative translation [X, Y, Z] (m)** and **Relative rotation [Roll, Pitch, Yaw] (deg)** parameters.

`Relative translation [X, Y, Z] (m)` — Translation offset relative to mounting location

`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Translation offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*X*, *Y*, *Z*]. Units are in meters.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see "Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox".

Example: `[0,0,0.01]`

Dependencies

To enable this parameter, select **Specify offset**.

`Relative rotation [Roll, Pitch, Yaw] (deg)` — Rotational offset relative to mounting location

`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Rotational offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*Roll*, *Pitch*, *Yaw*] . Roll, pitch, and yaw are the angles of rotation about the *X*-, *Y*-, and *Z*-axes, respectively. Units are in degrees.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.
- Roll, pitch, and yaw are clockwise-positive when looking in the forward direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. If you view a scene from a 2D top-down perspective, then the yaw angle (also called the orientation angle) is counterclockwise-positive because you are viewing the scene in the negative direction of the *Z*-axis.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see "Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox".

Example: `[0,0,10]`

Dependencies

To enable this parameter, select **Specify offset**.

`Sample time` — Sample time

`-1` (default) | positive scalar

Sample time of the block in seconds, specified as a positive scalar. The 3D simulation environment frame rate is the inverse of the sample time.

If you set the sample time to `-1`, the block inherits its sample time from the Simulation 3D Scene Configuration block.

Parameters

These intrinsic camera parameters are equivalent to the properties of a `cameraIntrinsics` object. To obtain the intrinsic parameters for your camera, use the **Camera Calibrator** app.

`Focal length (pixels)` — Focal length of camera

`[1109, 1109]` (default) | 1-by-2 positive integer vector

Focal length of the camera, specified as a 1-by-2 positive integer vector of the form [*fx*, *fy*]. Units are in pixels.

$fx = F \times sx$

$fy = F \times sy$

where:

- *F* is the focal length in world units, typically millimeters.
- [*sx*, *sy*] are the number of pixels per world unit in the *x* and *y* direction, respectively.

This parameter is equivalent to the `FocalLength` property of a `cameraIntrinsics` object.

`Optical center (pixels)` — Optical center of camera

`[640, 360]` (default) | 1-by-2 positive integer vector

Optical center of the camera, specified as a 1-by-2 positive integer vector of the form [*cx*,*cy*]. Units are in pixels.

This parameter is equivalent to the `PrincipalPoint` property of a `cameraIntrinsics` object.

`Image size (pixels)` — Image size produced by camera

`[720, 1280]` (default) | 1-by-2 positive integer vector

Image size produced by the camera, specified as a 1-by-2 positive integer vector of the form [*mrows*,*ncols*]. Units are in pixels.

This parameter is equivalent to the `ImageSize` property of a `cameraIntrinsics` object.

`Radial distortion coefficients` — Radial distortion coefficients

`[0, 0]` (default) | real-valued 1-by-2 nonnegative vector | real-valued 1-by-3 nonnegative vector

Radial distortion coefficients, specified as a real-valued 1-by-2 or 1-by-3 nonnegative vector. Radial distortion occurs when light rays bend more than the edges of a lens than they do at its optical center. The distortion is greater when the lens is smaller. The block calculates the radial-distorted location of a point. Units are dimensionless.

This parameter is equivalent to the `RadialDistortion` property of a `cameraIntrinsics` object.

`Tangential distortion coefficients` — Tangential distortion coefficients

`[0, 0]` (default) | real-valued 1-by-2 nonnegative vector

Tangential distortion coefficients, specified as a real-valued 1-by-2 nonnegative vector. Tangential distortion occurs when the lens and the image plane are not parallel. The coordinates are expressed in world units. Units are dimensionless.

This parameter is equivalent to the `TangentialDistortion` property of a `cameraIntrinsics` object.

`Axis skew` — Skew angle of camera axes

`0` (default) | nonnegative scalar

Skew angle of the camera axes, specified as a nonnegative scalar. If the *X*-axis and *Y*-axis are exactly perpendicular, then the skew must be `0`. Units are dimensionless.

This parameter is equivalent to the `Skew` property of a `cameraIntrinsics` object.

Ground Truth

`Output depth` — Output depth map

`off` (default) | `on`

Select this parameter to output a depth map at the **Depth** port.

`Output semantic segmentation` — Output semantic segmentation map of label IDs

`off` (default) | `on`

Select this parameter to output a semantic segmentation map of label IDs at the **Labels** port.

`Output location (m) and orientation (rad)` — Output location and orientation of sensor

`off` (default) | `on`

Select this parameter to output the location and orientation of the sensor at the **Location** and **Orientation** ports, respectively.

Tips

- To visualize the camera images that are output by the **Image** port, use a Video Viewer or To Video Display block.

  To learn how to visualize the depth and semantic segmentation maps that are output by the **Depth** and **Labels** ports, see the “Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation” example.
- Because the Unreal Engine can take a long time to start between simulations, consider logging the signals that the sensors output. You can then use this data to develop perception algorithms in MATLAB. See “Configure a Signal for Logging” (Simulink).

  You can also save image data as a video by using a To Multimedia File block. For an example of this setup, see “Design Lane Marker Detector Using Unreal Engine Simulation Environment”.

Algorithms

The block uses the camera model proposed by Jean-Yves Bouguet [1]. The model includes:

- The pinhole camera model [2]
- Lens distortion [3]

The pinhole camera model does not account for lens distortion because an ideal pinhole camera does not have a lens. To accurately represent a real camera, the full camera model used by the block includes radial and tangential lens distortion.

For more details, see “What Is Camera Calibration?” (Computer Vision Toolbox)

References

[1] Bouguet, J. Y. *Camera Calibration Toolbox for Matlab*. http://www.vision.caltech.edu/bouguetj/calib_doc

[2] Zhang, Z. "A Flexible New Technique for Camera Calibration." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 22, No. 11, 2000, pp. 1330–1334.

[3] Heikkila, J., and O. Silven. “A Four-step Camera Calibration Procedure with Implicit Image Correction.” *IEEE International Conference on Computer Vision and Pattern Recognition*. 1997.

See Also

Blocks
Simulation 3D Lidar | Simulation 3D Probabilistic Radar | Simulation 3D Fisheye Camera | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following | Simulation 3D Vision Detection Generator

Apps
Camera Calibrator

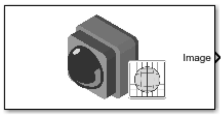
Objects
`cameraIntrinsics`

Topics
“Unreal Engine Simulation for Automated Driving”
“Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox”
“Choose a Sensor for Unreal Engine Simulation”
“What Is Camera Calibration?” (Computer Vision Toolbox)
“Depth Estimation From Stereo Video” (Computer Vision Toolbox)
“Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)

Introduced in R2019b

Simulation 3D Fisheye Camera

Fisheye camera sensor model in 3D simulation environment
Library: Automated Driving Toolbox / Simulation 3D



Description

The Simulation 3D Fisheye Camera block provides an interface to a camera with a fisheye lens in a 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The sensor is based on the fisheye camera model proposed by Scaramuzza [1] on page 2-117. The block outputs an image with the specified camera distortion and size. You can also output the location and orientation of the camera in the world coordinate system of the scene.

If you set **Sample time** to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block. To use this sensor, you must include a Simulation 3D Scene Configuration block in your model.

Note The Simulation 3D Scene Configuration block must execute before the Simulation 3D Fisheye Camera block. That way, the Unreal Engine 3D visualization environment prepares the data before the Simulation 3D Fisheye Camera block receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Fisheye Camera — `1`

For more information about execution order, see "How Unreal Engine Simulation for Automated Driving Works".

Ports

Output

`Image` — 3D output camera image
m-by-*n*-by-3 array of RGB triplet values

3D output camera image, returned as an *m*-by-*n*-by-3 array of RGB triplet values. *m* is the vertical resolution of the image, and *n* is the horizontal resolution of the image.

Data Types: `int8` | `uint8`

`Location` — Sensor location
real-valued 1-by-3 vector

Sensor location along the *X*-axis, *Y*-axis, and *Z*-axis of the scene. The **Location** values are in the world coordinates of the scene. In this coordinate system, the *Z*-axis points up from the ground. Units are in meters.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

`Orientation` — Sensor orientation
real-valued 1-by-3 vector

Roll, pitch, and yaw sensor orientation about the *X*-axis, *Y*-axis, and *Z*-axis of the scene. The **Orientation** values are in the world coordinates of the scene. These values are positive in the clockwise direction when looking in the positive directions of these axes. Units are in radians.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

Parameters

Mounting

`Sensor identifier` — Unique sensor identifier
`1` (default) | positive integer

Unique sensor identifier, specified as a positive integer. In a multisensor system, the sensor identifier distinguishes between sensors. When you add a new sensor block to your model, the **Sensor identifier** of that block is $N + 1$. N is the highest **Sensor identifier** value among existing sensor blocks in the model.

Example: 2

`Parent name` — Name of parent to which sensor is mounted
`Scene Origin` (default) | vehicle name

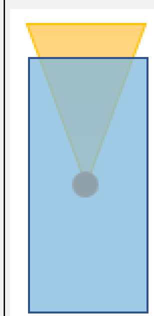
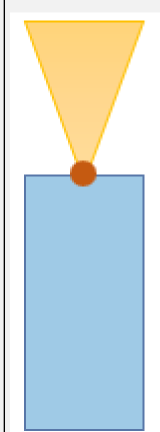
Name of the parent to which the sensor is mounted, specified as `Scene Origin` or as the name of a vehicle in your model. The vehicle names that you can select correspond to the **Name** parameters of the Simulation 3D Vehicle with Ground Following blocks in your model. If you select `Scene Origin`, the block places a sensor at the scene origin.

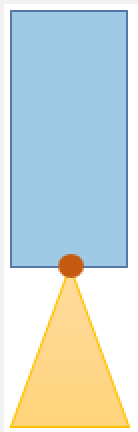
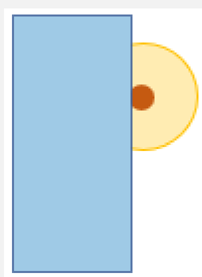
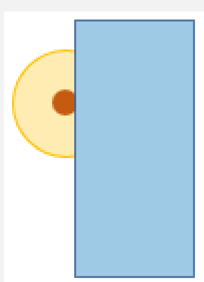
Example: `SimulinkVehicle1`

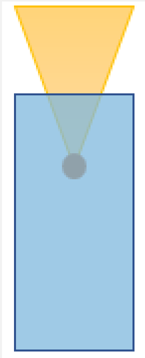
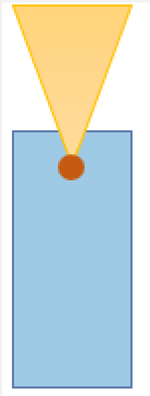
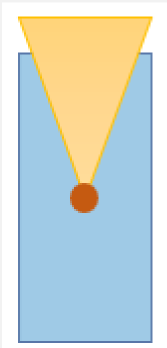
`Mounting location` — Sensor mounting location
`Origin` (default) | `Front bumper` | `Rear bumper` | `Right mirror` | `Left mirror` | `Rearview mirror` | `Hood center` | `Roof center`

Sensor mounting location.

- When **Parent name** is `Scene Origin`, the block mounts the sensor to the origin of the scene. You can set the **Mounting location** to `Origin` only. During simulation, the sensor remains stationary.
- When **Parent name** is the name of a vehicle (for example, `SimulinkVehicle1`) the block mounts the sensor to one of the predefined mounting locations described in the table. During simulation, the sensor travels with the vehicle.

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Origin`	Forward-facing sensor mounted to the vehicle origin, which is on the ground, at the geometric center of the vehicle (see "Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox") 	[0, 0, 0]
`Front bumper`	Forward-facing sensor mounted to the front bumper 	[0, 0, 0]

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Rear bumper`	Backward-facing sensor mounted to the rear bumper 	[0, 0, 180]
`Right mirror`	Downward-facing sensor mounted to the right side-view mirror 	[0, –90, 0]
`Left mirror`	Downward-facing sensor mounted to the left side-view mirror 	[0, –90, 0]

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Rearview mirror`	Forward-facing sensor mounted to the rearview mirror, inside the vehicle 	[0, 0, 0]
`Hood center`	Forward-facing sensor mounted to the center of the hood 	[0, 0, 0]
`Roof center`	Forward-facing sensor mounted to the center of the roof 	[0, 0, 0]

Roll, pitch, and yaw are clockwise-positive when looking in the positive direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. When looking at a vehicle from the top down, the yaw angle (that is, the

orientation angle) is counterclockwise-positive, because you are looking in the negative direction of the axis.

The (*X*, *Y*, *Z*) mounting location of the sensor relative to the vehicle depends on the vehicle type. To specify the vehicle type, use the **Type** parameter of the Simulation 3D Vehicle with Ground Following block to which you are mounting the sensor. To obtain the (*X*, *Y*, *Z*) mounting locations for a vehicle type, see the reference page for that vehicle.

To determine the location of the sensor in world coordinates, open the sensor block. Then, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)** and inspect the data from the **Location** output port.

`Specify offset` — Specify offset from mounting location

`off` (default) | `on`

Select this parameter to specify an offset from the mounting location by using the **Relative translation [X, Y, Z] (m)** and **Relative rotation [Roll, Pitch, Yaw] (deg)** parameters.

`Relative translation [X, Y, Z] (m)` — Translation offset relative to mounting location

`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Translation offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*X*, *Y*, *Z*]. Units are in meters.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see “Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox”.

Example: `[0,0,0.01]`

Dependencies

To enable this parameter, select **Specify offset**.

`Relative rotation [Roll, Pitch, Yaw] (deg)` — Rotational offset relative to mounting location

`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Rotational offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*Roll*, *Pitch*, *Yaw*] . Roll, pitch, and yaw are the angles of rotation about the *X*-, *Y*-, and *Z*-axes, respectively. Units are in degrees.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.
- Roll, pitch, and yaw are clockwise-positive when looking in the forward direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. If you view a scene from a 2D top-down perspective, then the yaw angle (also called the orientation angle) is counterclockwise-positive because you are viewing the scene in the negative direction of the *Z*-axis.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see "Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox".

Example: `[0,0,10]`

Dependencies

To enable this parameter, select **Specify offset**.

`Sample time` — Sample time

`-1` (default) | positive scalar

Sample time of the block in seconds, specified as a positive scalar. The 3D simulation environment frame rate is the inverse of the sample time.

If you set the sample time to `-1`, the block inherits its sample time from the Simulation 3D Scene Configuration block.

Parameters

These intrinsic camera parameters are equivalent to the properties of a `fisheyeIntrinsics` object. To obtain the intrinsic parameters for your camera, use the **Camera Calibrator** app.

`Distortion center (pixels)` — Center of distortion

`[640, 360]` (default) | real-valued 1-by-2 vector

Center of distortion, specified as real-valued 2-element vector. Units are in pixels.

`Image size (pixels)` — Image size produced by camera

`[720, 1280]` (default) | real-valued 1-by-2 vector of positive integers

Image size produced by the camera, specified as a real-valued 1-by-2 vector of positive integers of the form [*mrows*,*ncols*]. Units are in pixels.

`Mapping coefficients` — Polynomial coefficients for projection function

`[320, 0, 0, 0]` (default) | real-valued 1-by-4 vector

Polynomial coefficients for the projection function described by Scaramuzza's Taylor model [1], specified as a real-valued 1-by-4 vector of the form `[a0 a2 a3 a4]`.

Example: `[320, -0.001, 0, 0]`

`Stretch matrix` — Transforms point from sensor plane to camera plane
`[1, 0; 0, 1]` (default) | real-valued 2-by-2 matrix

Transforms a point from the sensor plane to a pixel in the camera image plane. The misalignment occurs during the digitization process when the lens is not parallel to sensor.

Example: `[0, 1; 0, 1]`

Ground Truth

`Output location (m) and orientation (rad)` — Output location and orientation of sensor
`off` (default) | `on`

Select this parameter to output the location and orientation of the sensor at the **Location** and **Orientation** ports, respectively.

Tips

- To visualize the camera images that are output by the **Image** port, use a Video Viewer or To Video Display block.
- Because the Unreal Engine can take a long time to start up between simulations, consider logging the signals that the sensors output. You can then use this data to develop perception algorithms in MATLAB. See "Configure a Signal for Logging" (Simulink).

  You can also save image data as a video by using a To Multimedia File block. For an example of this setup, see "Design Lane Marker Detector Using Unreal Engine Simulation Environment".

References

[1] Scaramuzza, D., A. Martinelli, and R. Siegwart. "A Toolbox for Easy Calibrating Omindirectional Cameras." *Proceedings to IEEE International Conference on Intelligent Robots and Systems (IROS 2006)*. Beijing, China, October 7–15, 2006.

See Also

Blocks
Simulation 3D Camera | Simulation 3D Lidar | Simulation 3D Probabilistic Radar | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following | Simulation 3D Vision Detection Generator

Apps
Camera Calibrator

Objects
`fisheyeIntrinsics`

Topics
"Unreal Engine Simulation for Automated Driving"

“Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox”
“Choose a Sensor for Unreal Engine Simulation”
“Fisheye Calibration Basics” (Computer Vision Toolbox)

Introduced in R2019b

Simulation 3D Lidar

Lidar sensor model in 3D simulation environment
Library: Automated Driving Toolbox / Simulation 3D

Description

The Simulation 3D Lidar block provides an interface to the lidar sensor in a 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The block returns a point cloud with the specified field of view and angular resolution. You can also output the distances from the sensor to object points. In addition, you can output the location and orientation of the sensor in the world coordinate system of the scene.

If you set **Sample time** to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block. To use this sensor, ensure that the Simulation 3D Scene Configuration block is in your model.

Note The Simulation 3D Scene Configuration block must execute before the Simulation 3D Lidar block. That way, the Unreal Engine 3D visualization environment prepares the data before the Simulation 3D Lidar block receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Lidar — `1`

For more information about execution order, see “How Unreal Engine Simulation for Automated Driving Works”.

Ports

Output

`Point cloud` — Point cloud data

m-by-*n*-by-3 array of positive real-valued [*x*, *y*, *z*] points

Point cloud data, returned as an *m*-by-*n*-by 3 array of positive, real-valued [*x*, *y*, *z*] points. *m* and *n* define the number of points in the point cloud, as shown in this equation:

$$m \times n = \frac{V_{FOV}}{V_{RES}} \times \frac{H_{FOV}}{H_{RES}}$$

where:

- V_{FOV} is the vertical field of view of the lidar, in degrees, as specified by the **Vertical field of view (deg)** parameter.

- V_{RES} is the vertical angular resolution of the lidar, in degrees, as specified by the **Vertical resolution (deg)** parameter.
- H_{FOV} is the horizontal field of view of the lidar, in degrees, as specified by the **Horizontal field of view (deg)** parameter.
- H_{RES} is the horizontal angular resolution of the lidar, in degrees, as specified by the **Horizontal resolution (deg)** parameter.

Each *m*-by-*n* entry in the array specifies the *x*, *y*, and *z* coordinates of a detected point in the sensor coordinate system. If the lidar does not detect a point at a given coordinate, then *x*, *y*, and *z* are returned as `NaN`.

You can create a point cloud from these returned points by using point cloud functions in a MATLAB Function block. For a list of point cloud processing functions, see “Lidar Processing”. For an example that uses these functions, see “Design Lidar SLAM Algorithm Using Unreal Engine Simulation Environment”.

Data Types: `single`

`Distance` — Distance to object points

m-by-*n* positive real-valued matrix

Distance to object points measured by the lidar sensor, returned as an *m*-by-*n* positive real-valued matrix. Each *m*-by-*n* value in the matrix corresponds to an [*x*, *y*, *z*] coordinate point returned by the **Point cloud** output port.

Dependencies

To enable this port, on the **Parameters** tab, select **Distance outport**.

Data Types: `single`

`Location` — Sensor location

real-valued 1-by-3 vector

Sensor location along the *X*-axis, *Y*-axis, and *Z*-axis of the scene. The **Location** values are in the world coordinates of the scene. In this coordinate system, the *Z*-axis points up from the ground. Units are in meters.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

`Orientation` — Sensor orientation

real-valued 1-by-3 vector

Roll, pitch, and yaw sensor orientation about the *X*-axis, *Y*-axis, and *Z*-axis of the scene. The **Orientation** values are in the world coordinates of the scene. These values are positive in the clockwise direction when looking in the positive directions of these axes. Units are in radians.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

Parameters

Mounting

`Sensor identifier` — Unique sensor identifier
`1` (default) | positive integer

Unique sensor identifier, specified as a positive integer. In a multisensor system, the sensor identifier distinguishes between sensors. When you add a new sensor block to your model, the **Sensor identifier** of that block is *N* + 1. *N* is the highest **Sensor identifier** value among existing sensor blocks in the model.

Example: `2`

`Parent name` — Name of parent to which sensor is mounted
`Scene Origin` (default) | vehicle name

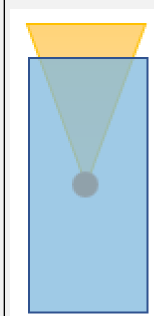
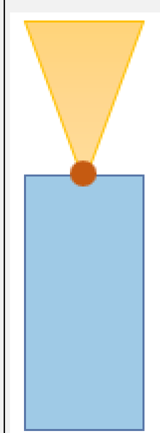
Name of the parent to which the sensor is mounted, specified as `Scene Origin` or as the name of a vehicle in your model. The vehicle names that you can select correspond to the **Name** parameters of the Simulation 3D Vehicle with Ground Following blocks in your model. If you select `Scene Origin`, the block places a sensor at the scene origin.

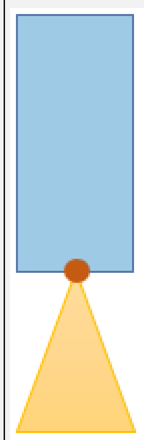
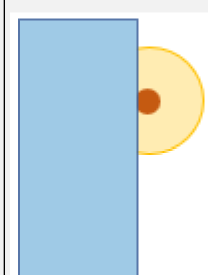
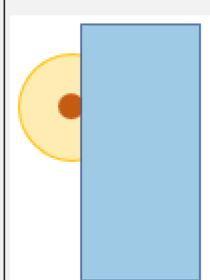
Example: `SimulinkVehicle1`

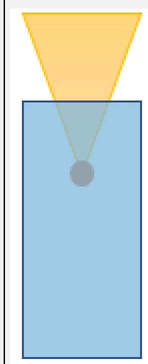
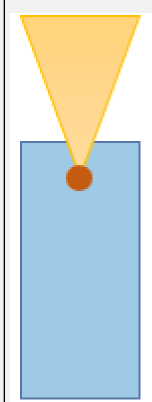
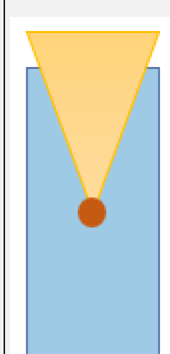
`Mounting location` — Sensor mounting location
`Origin` (default) | `Front bumper` | `Rear bumper` | `Right mirror` | `Left mirror` | `Rearview mirror` | `Hood center` | `Roof center`

Sensor mounting location.

- When **Parent name** is `Scene Origin`, the block mounts the sensor to the origin of the scene. You can set the **Mounting location** to `Origin` only. During simulation, the sensor remains stationary.
- When **Parent name** is the name of a vehicle (for example, `SimulinkVehicle1`) the block mounts the sensor to one of the predefined mounting locations described in the table. During simulation, the sensor travels with the vehicle.

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Origin`	Forward-facing sensor mounted to the vehicle origin, which is on the ground, at the geometric center of the vehicle (see “Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox”) 	[0, 0, 0]
`Front bumper`	Forward-facing sensor mounted to the front bumper 	[0, 0, 0]

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Rear bumper`	Backward-facing sensor mounted to the rear bumper 	[0, 0, 180]
`Right mirror`	Downward-facing sensor mounted to the right side-view mirror 	[0, –90, 0]
`Left mirror`	Downward-facing sensor mounted to the left side-view mirror 	[0, –90, 0]

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Rearview mirror`	Forward-facing sensor mounted to the rearview mirror, inside the vehicle 	[0, 0, 0]
`Hood center`	Forward-facing sensor mounted to the center of the hood 	[0, 0, 0]
`Roof center`	Forward-facing sensor mounted to the center of the roof 	[0, 0, 0]

Roll, pitch, and yaw are clockwise-positive when looking in the positive direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. When looking at a vehicle from the top down, the yaw angle (that is, the

orientation angle) is counterclockwise-positive, because you are looking in the negative direction of the axis.

The (*X*, *Y*, *Z*) mounting location of the sensor relative to the vehicle depends on the vehicle type. To specify the vehicle type, use the **Type** parameter of the Simulation 3D Vehicle with Ground Following block to which you are mounting the sensor. To obtain the (*X*, *Y*, *Z*) mounting locations for a vehicle type, see the reference page for that vehicle.

To determine the location of the sensor in world coordinates, open the sensor block. Then, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)** and inspect the data from the **Location** output port.

`Specify offset` — Specify offset from mounting location
`off` (default) | `on`

Select this parameter to specify an offset from the mounting location by using the **Relative translation [X, Y, Z] (m)** and **Relative rotation [Roll, Pitch, Yaw] (deg)** parameters.

`Relative translation [X, Y, Z] (m)` — Translation offset relative to mounting location
`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Translation offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*X*, *Y*, *Z*]. Units are in meters.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see “Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox”.

Example: `[0,0,0.01]`

Dependencies

To enable this parameter, select **Specify offset**.

`Relative rotation [Roll, Pitch, Yaw] (deg)` — Rotational offset relative to mounting location
`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Rotational offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*Roll*, *Pitch*, *Yaw*] . Roll, pitch, and yaw are the angles of rotation about the *X*-, *Y*-, and *Z*-axes, respectively. Units are in degrees.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.
- Roll, pitch, and yaw are clockwise-positive when looking in the forward direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. If you view a scene from a 2D top-down perspective, then the yaw angle (also called the orientation angle) is counterclockwise-positive because you are viewing the scene in the negative direction of the *Z*-axis.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see "Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox".

Example: `[0,0,10]`

Dependencies

To enable this parameter, select **Specify offset**.

`Sample time` — Sample time

`-1` (default) | positive scalar

Sample time of the block in seconds, specified as a positive scalar. The 3D simulation environment frame rate is the inverse of the sample time.

If you set the sample time to `-1`, the block inherits its sample time from the Simulation 3D Scene Configuration block.

Parameters

`Detection range (m)` — Maximum distance measured by lidar sensor

`120` (default) | positive scalar

Maximum distance measured by the lidar sensor, specified as a positive scalar. Points outside this range are ignored. Units are in meters.

`Range resolution (m)` — Resolution of lidar sensor range

`0.002` (default) | positive real scalar

Resolution of the lidar sensor range, in meters, specified as a positive real scalar. The range resolution is also known as the quantization factor. The minimal value of this factor is D_{range} / 2^{24}, where D_{range} is the maximum distance measured by the lidar sensor, as specified in the **Detection range (m)** parameter.

`Vertical field of view (deg)` — Vertical field of view

`40` (default) | positive scalar

Vertical field of view of the lidar sensor, specified as a positive scalar. Units are in degrees.

`Vertical resolution (deg)` — Vertical angular resolution
`1.25` (default) | positive scalar

Vertical angular resolution of the lidar sensor, specified as a positive scalar. Units are in degrees.

`Horizontal field of view (deg)` — Horizontal field of view
`360` (default) | positive scalar

Horizontal field of view of the lidar sensor, specified as a positive scalar. Units are in degrees.

`Horizontal resolution (deg)` — Horizontal angular (azimuth) resolution
`0.16` (default) | positive scalar

Horizontal angular (azimuth) resolution of the lidar sensor, specified as a positive scalar. Units are in degrees.

`Distance outport` — Output distance to measured object points
`off` (default) | `on`

Select this parameter to output the distance to measured object points at the **Distance** port.

Ground Truth

`Output location (m) and orientation (rad)` — Output location and orientation of sensor
`off` (default) | `on`

Select this parameter to output the location and orientation of the sensor at the **Location** and **Orientation** ports, respectively.

Tips

- To visualize point clouds that are output by the **Point cloud** port, you can either:
  - Use a `pcplayer` object in a MATLAB Function block. For an example of this visualization setup, see “Design Lidar SLAM Algorithm Using Unreal Engine Simulation Environment”.
  - Use the **Bird's-Eye Scope**. For more details, see “Visualize Sensor Data from Unreal Engine Simulation Environment”.
- The Unreal Engine can take a long time to start up between simulations, consider logging the signals that the sensors output. You can then use this data to develop perception algorithms in MATLAB. See “Configure a Signal for Logging” (Simulink).

See Also

Apps
Bird's-Eye Scope

Objects
`pcplayer` | `pointCloud`

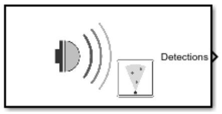
Topics
“Unreal Engine Simulation for Automated Driving”

“Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox”
“Choose a Sensor for Unreal Engine Simulation”
“Lidar Processing”

Introduced in R2019b

Simulation 3D Probabilistic Radar

Probabilistic radar sensor model in 3D simulation environment
Library: Automated Driving Toolbox / Simulation 3D



Description

The Simulation 3D Probabilistic Radar block provides an interface to the probabilistic radar sensor in a 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. You can specify the radar model and accuracy, bias, and detection parameters. The block uses the sample time to capture the radar detections and outputs a list of object detection reports. To configure the probabilistic radar signatures of actors in the 3D environment across all radars in your model, use a Simulation 3D Probabilistic Radar Configuration block.

If you set **Sample time** to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block. To use this sensor, you must include a Simulation 3D Scene Configuration block in your model.

Note The Simulation 3D Scene Configuration block must execute before the Simulation 3D Probabilistic Radar block. That way, the Unreal Engine 3D visualization environment prepares the data before the Simulation 3D Probabilistic Radar block receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Probabilistic Radar — `1`

For more information about execution order, see “How Unreal Engine Simulation for Automated Driving Works”.

Ports

Output

`Detections` — Object detections

Simulink bus containing MATLAB structure

Object detections, returned as a Simulink bus containing a MATLAB structure. For more details about buses, “Create Nonvirtual Buses” (Simulink). The structure has this form.

Field	Description	Type
`NumDetections`	Number of detections	integer

Field	Description	Type
`IsValidTime`	False when updates are requested at times that are between block invocation intervals	Boolean
`Detections`	Object detections	Array of object detection structures of length set by the **Maximum reported** parameter. Only `NumDetections` of these detections are actual detections.

Each object detection structure contains these properties.

Property	Definition
`Time`	Measurement time
`Measurement`	Object measurements
`MeasurementNoise`	Measurement noise covariance matrix
`SensorIndex`	Unique ID of the sensor
`ObjectClassID`	Object classification
`ObjectAttributes`	Additional information passed to tracker
`MeasurementParameters`	Parameters used by initialization functions of nonlinear Kalman tracking filters

- For Cartesian coordinates, `Measurement` and `MeasurementNoise` are reported in the coordinate system specified by the **Coordinate system** parameter.
- For spherical coordinates, `Measurement` and `MeasurementNoise` are reported in the spherical coordinate system based on the sensor Cartesian coordinate system. `MeasurementParameters` is reported in sensor Cartesian coordinates.

Measurement and MeasurementNoise

<table>
<tr><th>Coordinate System Used to Report Detections</th><th>Measurement and MeasurementNoise Coordinates</th></tr>
<tr><td><code>'Ego Cartesian'</code></td><td rowspan="2">This table shows the coordinate dependence when you enable or disable range rate measurements using the Enable range rate measurements parameter.
<table>
<tr><th>Range rate measurements</th><th>Coordinates</th></tr>
<tr><td>Enabled</td><td><code>[x;y;z;vx;vy;vz]</code></td></tr>
<tr><td>Disabled</td><td><code>[x;y;z]</code></td></tr>
</table></td></tr>
<tr><td><code>'Sensor Cartesian'</code></td></tr>
<tr><td><code>'Sensor spherical'</code></td><td>This table shows the coordinate dependence when you enable or disable the range rate and elevation angle measurements, by using the Enable range rate measurements and Enable elevation angle measurements parameters, respectively.
<table>
<tr><th>Range rate measurements</th><th>Elevation angle measurements</th><th>Coordinates</th></tr>
<tr><td>Enabled</td><td>Enabled</td><td><code>[az;el;rng;rr]</code></td></tr>
<tr><td>Enabled</td><td>Disabled</td><td><code>[az;rng;rr]</code></td></tr>
<tr><td>Disabled</td><td>Enabled</td><td><code>[az;el;rng]</code></td></tr>
<tr><td>Disabled</td><td>Disabled</td><td><code>[az;rng]</code></td></tr>
</table></td></tr>
</table>

Measurement Parameters

Parameter	Definition
`Frame`	Enumerated type that indicates the frame used to report measurements. When `Frame` is set to `'rectangular'`, detections are reported in Cartesian coordinates. When `Frame` is set to `'spherical'`, detections are reported in spherical coordinates.
`OriginPosition`	3D vector offset of the sensor origin from the ego vehicle origin. The vector is derived from the location and height of the sensor, as specified by the **Mounting location** parameter and the *Z* value of the **Relative translation [X, Y, Z] (m)** parameter, respectively.
`Orientation`	Orientation of the radar sensor coordinate system with respect to the ego vehicle coordinate system. The orientation is derived from the roll, pitch, and yaw values specified in the **Relative rotation [Roll, Pitch, Yaw] (deg)** parameter.
`HasVelocity`	Indicates whether measurements contain velocity or range rate components.
`HasElevation`	Indicates whether measurements contain elevation components.

The `ObjectAttributes` property of each detection is a structure with these fields.

Field	Definition
`TargetIndex`	Identifier of the actor, `ActorID`, that generated the detection. For false alarms, this value is negative.
`SNR`	Signal-to-noise ratio of the detection. Units are in decibels.

The `ObjectClassID` property of each detection has a value that corresponds to an object ID. The table shows the object IDs used in the default scenes that are selectable from the Simulation 3D Scene Configuration block. If you are using a custom scene, in the Unreal Editor, you can assign new object types to unused IDs. If a scene contains an object that does not have an assigned ID, that object is assigned an ID of `0`. The detection of lane markings is not supported.

ID	Type
`0`	None/default
`1`	Building
`2`	*Not used*
`3`	Other
`4`	*Not used*
`5`	Pole

ID	Type
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*
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

ID	Type
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*

Parameters

Mounting

Sensor identifier — Unique sensor identifier

1 (default) | positive integer

Unique sensor identifier, specified as a positive integer. In a multisensor system, the sensor identifier distinguishes between sensors. When you add a new sensor block to your model, the **Sensor identifier** of that block is $N + 1$. N is the highest **Sensor identifier** value among existing sensor blocks in the model.

Example: 2

Parent name — Name of parent to which sensor is mounted

Scene Origin (default) | vehicle name

Name of the parent to which the sensor is mounted, specified as Scene Origin or as the name of a vehicle in your model. The vehicle names that you can select correspond to the **Name** parameters of the Simulation 3D Vehicle with Ground Following blocks in your model. If you select Scene Origin, the block places a sensor at the scene origin.

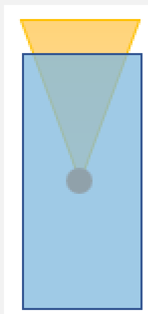
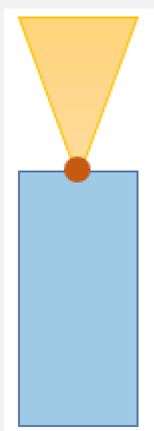
Example: SimulinkVehicle1

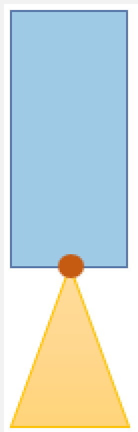
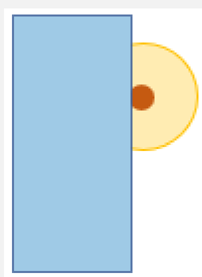
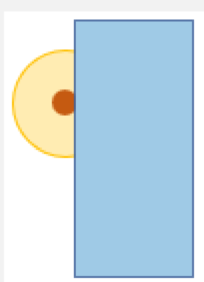
Mounting location — Sensor mounting location

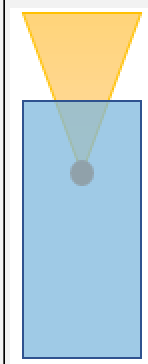
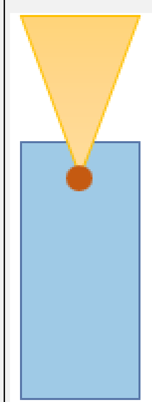
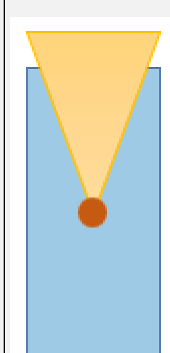
Origin (default) | Front bumper | Rear bumper | Right mirror | Left mirror | Rearview mirror | Hood center | Roof center

Sensor mounting location.

- When **Parent name** is Scene Origin, the block mounts the sensor to the origin of the scene. You can set the **Mounting location** to Origin only. During simulation, the sensor remains stationary.
- When **Parent name** is the name of a vehicle (for example, SimulinkVehicle1) the block mounts the sensor to one of the predefined mounting locations described in the table. During simulation, the sensor travels with the vehicle.

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Origin`	Forward-facing sensor mounted to the vehicle origin, which is on the ground, at the geometric center of the vehicle (see "Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox") 	[0, 0, 0]
`Front bumper`	Forward-facing sensor mounted to the front bumper 	[0, 0, 0]

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Rear bumper`	Backward-facing sensor mounted to the rear bumper 	[0, 0, 180]
`Right mirror`	Downward-facing sensor mounted to the right side-view mirror 	[0, –90, 0]
`Left mirror`	Downward-facing sensor mounted to the left side-view mirror 	[0, –90, 0]

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Rearview mirror`	Forward-facing sensor mounted to the rearview mirror, inside the vehicle 	[0, 0, 0]
`Hood center`	Forward-facing sensor mounted to the center of the hood 	[0, 0, 0]
`Roof center`	Forward-facing sensor mounted to the center of the roof 	[0, 0, 0]

Roll, pitch, and yaw are clockwise-positive when looking in the positive direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. When looking at a vehicle from the top down, the yaw angle (that is, the

orientation angle) is counterclockwise-positive, because you are looking in the negative direction of the axis.

The (*X*, *Y*, *Z*) mounting location of the sensor relative to the vehicle depends on the vehicle type. To specify the vehicle type, use the **Type** parameter of the Simulation 3D Vehicle with Ground Following block to which you are mounting the sensor. To obtain the (*X*, *Y*, *Z*) mounting locations for a vehicle type, see the reference page for that vehicle.

To determine the location of the sensor in world coordinates, open the sensor block. Then, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)** and inspect the data from the **Location** output port.

`Specify offset` — Specify offset from mounting location
`off` (default) | `on`

Select this parameter to specify an offset from the mounting location by using the **Relative translation [X, Y, Z] (m)** and **Relative rotation [Roll, Pitch, Yaw] (deg)** parameters.

`Relative translation [X, Y, Z] (m)` — Translation offset relative to mounting location
`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Translation offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*X*, *Y*, *Z*]. Units are in meters.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see “Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox”.

Example: `[0,0,0.01]`

Dependencies

To enable this parameter, select **Specify offset**.

`Relative rotation [Roll, Pitch, Yaw] (deg)` — Rotational offset relative to mounting location
`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Rotational offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*Roll*, *Pitch*, *Yaw*] . Roll, pitch, and yaw are the angles of rotation about the *X*-, *Y*-, and *Z*-axes, respectively. Units are in degrees.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.
- Roll, pitch, and yaw are clockwise-positive when looking in the forward direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. If you view a scene from a 2D top-down perspective, then the yaw angle (also called the orientation angle) is counterclockwise-positive because you are viewing the scene in the negative direction of the *Z*-axis.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see "Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox".

Example: `[0,0,10]`

Dependencies

To enable this parameter, select **Specify offset**.

`Sample time` — Sample time

`-1` (default) | positive scalar

Sample time of the block in seconds, specified as a positive scalar. The 3D simulation environment frame rate is the inverse of the sample time.

If you set the sample time to `-1`, the block inherits its sample time from the Simulation 3D Scene Configuration block.

Parameters

Accuracy Settings

`Azimuthal resolution of radar (deg)` — Azimuth resolution of radar

`4` (default) | positive real scalar

Azimuth resolution of the radar, specified as a positive real scalar. The azimuth resolution defines the minimum separation in azimuth angle at which the radar can distinguish between two targets. The azimuth resolution is typically the 3dB-downpoint in azimuth angle beamwidth of the radar. Units are in degrees.

Example: `6.5`

`Elevation resolution of radar (deg)` — Elevation resolution of radar

`10` (default) | positive real scalar

Elevation resolution of the radar, specified as a positive real scalar. The elevation resolution defines the minimum separation in elevation angle at which the radar can distinguish between two targets.

The elevation resolution is typically the 3dB-downpoint in elevation angle beamwidth of the radar. Units are in degrees.

Example: `3.5`

Dependencies

To enable this parameter, on the **Parameters** tab, in the **Radar model** section, select **Enable elevation angle measurements**.

`Range resolution of radar (m)` — Range resolution of radar

`2.5` (default) | positive real scalar

Range resolution of the radar, specified as a positive real scalar. The range resolution defines the minimum separation in range at which the radar can distinguish between two targets. Units are in meters.

Example: `5.0`

`Range rate resolution of radar (m/s)` — Range rate resolution of the radar

`0.5` (default) | positive real scalar

Range rate resolution of the radar, specified as a positive real scalar. The range rate resolution defines the minimum separation in range rate at which the radar can distinguish between two targets. Units are in meters per second.

Example: `0.75`

Dependencies

To enable this parameter, on the **Parameters** tab, in the **Radar model** section, select **Enable range rate measurements**.

Bias Settings

`Fractional azimuthal bias component` — Azimuth bias fraction

`0.1` (default) | nonnegative real scalar

Azimuth bias fraction of the radar, specified as a nonnegative real scalar. The azimuth bias is expressed as a fraction of the azimuth resolution specified in the **Azimuthal resolution of radar (deg)** parameter. Units are dimensionless.

Example: 0.3

`Fractional elevation bias component` — Elevation bias fraction

`0.1` (default) | nonnegative real scalar

Elevation bias fraction of the radar, specified as a nonnegative real scalar. The elevation bias is expressed as a fraction of the elevation resolution specified in the **Elevation resolution of radar (deg)** parameter. Units are dimensionless.

Example: 0.2

Dependencies

To enable this parameter, on the **Parameters** tab, in the **Radar model** section, select **Enable elevation angle measurements**.

Fractional range bias component — Range bias fraction
0.05 (default) | nonnegative real scalar

Range bias fraction of the radar, specified as a nonnegative real scalar. Range bias is expressed as a fraction of the range resolution specified in the **Range resolution of radar (m)** parameter. Units are dimensionless.

Example: 0.15

Fractional range rate bias component — Range rate bias fraction
0.05 (default) | nonnegative real scalar

Range rate bias fraction of the radar, specified as a nonnegative real scalar. Range rate bias is expressed as a fraction of the range rate resolution specified in the **Range rate resolution of radar (m/s)** parameter. Units are dimensionless.

Example: 0.2

Dependencies

To enable this parameter, on the **Parameters** tab, in the **Radar model** section, select **Enable range rate measurements**.

Detector Settings

Field of view (deg) — Field of view
[20, 5] (default) | positive real-valued 1-by-2 vector

Field of view of the radar, specified as a positive real-valued 1-by-2 vector of the form `[azfov, elfov]`. `azfov` is the azimuth angle field of view. *elfov* is the elevation angle field of view. The field of view defines the angular extent spanned by the sensor. Each component must lie in the interval (0,180]. Targets outside of the field of view of the radar are not detected. Units are in degrees.

Example: [14 7]

Detection ranges (m) — Detection range
[1, 150] (default) | positive real-valued 1-by-2 vector

Detection range, in meters, at which the radar can detect a target.

- To set only a maximum detection range, specify this parameter as a positive real scalar. By default, the minimum detection range is 0.
- To set both a minimum and maximum detection range, specify this parameter as a positive real-valued 1-by-2 vector of the form `[min, max]`.

Example: 250

Range rates (m/s) — Minimum and maximum detection range rates
[-100, 100] (default) | real-valued 1-by-2 vector

Minimum and maximum detection range rates, specified as a real-valued 1-by-2 vector. The radar can detect targets only within this range rate interval. Units are in meters per second.

Example: [-200 200]

Dependencies

To enable this parameter, on the **Parameters** tab, in the **Radar model** section, select **Enable range rate measurements**.

`Detection probability` — Probability that radar detects a target
`0.9` (default) | real scalar in the range (0, 1]

Probability that the radar detects a target, specified as a real scalar in the range (0, 1]. This quantity defines the probability of detecting a target that has a radar cross section specified by the **Reference radar cross section (dBsm)** parameter, at the reference detection range specified by the **Detection ranges (m)** parameter.

Example: `0.95`

`False alarm rate` — False alarm rate
`1e-6` (default) | positive real scalar in range [10^{-7}, 10^{-3}]

False alarm rate within a radar resolution cell, specified as a positive real scalar in the range [10^{-7}, 10^{-3}]. Units are dimensionless.

Example: `1e-5`

`Detection probability range (m):` — Reference range for given probability of detection
`100` (default) | positive real scalar

Reference range for a given probability of detection, specified as a positive real scalar. The reference range is the range at which the radar detects targets that have a radar cross section specified by **Reference radar cross section (dBsm)**, given a detection probability specified by **Detection probability**. Units are in meters.

Example: `150`

`Reference radar cross section (dBsm)` — Reference radar cross section for given probability of detection
`0` (default) | nonnegative real scalar

Reference radar cross section (RCS) for a given probability of detection, specified as a nonnegative real scalar. A radar with the detection probability specified by **Detection probability** detects targets at this reference RCS value. Units are in decibels per square meter.

Example: `2.0`

Radar Model

`Enable elevation angle measurements` — Enable radar to measure elevation
`on` (default) | `off`

Select this parameter to model a radar that can measure target elevation angles. This parameter enables the **Elevation resolution of radar (deg)** and **Fractional elevation bias component** parameters.

`Enable range rate measurements` — Enable radar to measure range rate
`on` (default) | `off`

Select this parameter to model a radar that can measure target range rates. This parameter enables the **Range rate resolution of radar (m/s)**, **Fractional range bias component**, and **Range rates (m/s)** parameters.

`Enable measurement noise` — Enable adding noise to radar sensor measurements

`on` (default) | `off`

Select this parameter to add noise to radar sensor measurements. Otherwise, the measurements are noise-free. The `MeasurementNoise` property of each detection is always computed and is not affected by the value you specify for the **Measurement noise** parameter. By not selecting this parameter, you can pass the sensor ground truth measurements into a Multi-Object Tracker block.

`Enable false detections` — Enable reporting false alarm radar detections

`on` (default) | `off`

Select this parameter to enable reporting false alarm radar measurements. Otherwise, only actual detections are reported.

`Random number generator method` — Method to set random number generator seed

`Repeatable` (default) | `Specify seed` | `Not repeatable`

Method to set the random number generator seed, specified as one of the options in the table.

Option	Description
`Repeatable`	The block generates a random initial seed for the first simulation and reuses this seed for all subsequent simulations. Select this parameter to generate repeatable results from the statistical sensor model. To change this initial seed, at the MATLAB command prompt, enter `clear all`.
`Specify seed`	Specify your own random initial seed for reproducible results by using the **Specify seed** parameter.
`Not repeatable`	The block generates a new random initial seed after each simulation run. Select this parameter to generate nonrepeatable results from the statistical sensor model.

`Initial seed` — Random number generator seed

`0` (default) | scalar in range $[0, 2^{32})$

Random number generator seed, specified as a scalar in the range $[0, 2^{32})$

Example: `2001`

Dependencies

To enable this parameter, set the **Random number generator method** parameter to `Specify seed`.

Detection Reporting

`Maximum reported` — Maximum number of reported detections

`50` (default) | positive integer

Maximum number of reported detections, specified as a positive integer. Units are dimensionless.

Example: 35

`Coordinate system` — Coordinate system of reported detections
`Ego Cartesian` (default) | `Sensor Cartesian` | `Sensor spherical`

Coordinate system of reported detections, specified as one of these values:

- `Ego Cartesian` — The radar reports detections in the ego vehicle Cartesian coordinate system.
- `Sensor Cartesian`— The radar reports detections in the sensor Cartesian coordinate system.
- `Sensor spherical` — The radar reports detections in the spherical coordinate system. This coordinate system is centered at the radar and aligned with the orientation of the radar on the ego vehicle.

`Specify output bus name` — Specify name of output bus
`off` (default) | `on`

Select this parameter to specify the name of the bus that the block outputs to the base workspace. Specify this name in the **Output bus name** parameter.

`Output bus name` — Name of output bus
`BusSimulation3DRadarTruthSensor` (default) | valid bus name

Name of the bus that the block outputs to the base workspace.

Dependencies

To enable this parameter, select the **Specify output bus name** parameter.

Tips

- To visualize detections and sensor coverage areas, use the **Bird's-Eye Scope**. For more details, see “Visualize Sensor Data from Unreal Engine Simulation Environment”.
- Because the Unreal Engine can take a long time to start between simulations, consider logging the signals that the sensors output. For more details, see “Configure a Signal for Logging” (Simulink).

References

[1] Blacksmith, P., R. E. Hiatt, and R. B. Mack. "Introduction to radar cross-section measurements." *Proceedings of the IEEE*. Volume 53, No. 8, August 1965, pp. 901–920. doi: 10.1109/PROC.1965.4069.

See Also

Apps
Bird's-Eye Scope

Blocks
Detection Concatenation | Multi-Object Tracker | Simulation 3D Probabilistic Radar Configuration | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following | Simulation 3D Vision Detection Generator

Topics

“Unreal Engine Simulation for Automated Driving”
“Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox”
“Choose a Sensor for Unreal Engine Simulation”
“Visualize Sensor Data and Tracks in Bird's-Eye Scope”

Introduced in R2019b

Simulation 3D Probabilistic Radar Configuration

Configure probabilistic radar signatures in 3D simulation environment

Library: Automated Driving Toolbox / Simulation 3D

Description

The Simulation 3D Probabilistic Radar Configuration block configures the probabilistic radar signatures for actors in a 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. To model the probabilistic radars, use Simulation 3D Probabilistic Radar blocks. The configured radar signatures apply to all Simulation 3D Probabilistic Radar blocks in your model.

Parameters

`Radar targets` — Identifiers corresponding to radar targets

[ ] (default) | positive integer | *L*-length vector of unique positive integers

Identifiers that correspond to radar targets, specified as a positive integer or *L*-length vector of unique positive integers. *L* equals the number of radar targets for which you want to specify a nondefault radar cross section (RCS).

This table provides the identifiers and corresponding object types that radars can detect in the default scenes that you can select from the Simulation 3D Scene Configuration block. For example, to specify a nondefault RCS for a building and a road, set **Radar targets** to `[1,7]`. If you are using a custom scene, in the Unreal Editor, you can assign new object types to unused IDs. If a scene contains an object that does not have an assigned ID, that object is assigned an ID of `0`. The detection of lane markings is not supported.

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

ID	Type
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*
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*

ID	Type
71	Barricade
72	Motorcycle
73-255	*Not used*

`Radar cross sections (dBsm)` — Radar cross sections
`{}` (default) | real-valued Q-by-P matrix | L-length cell array of real-valued Q_1-by-P_1, …, Q_L-by-P_L matrices

Radar cross sections of target actors, in decibels per square meter, specified as a matrix or cell array of matrices. Each matrix defines the RCS for the corresponding target actor specified by **Radar targets**.

If **Radar targets** is a scalar (that is, a single target actor), then specify **Radar cross sections (dBsm)** as a real-valued Q-by-P matrix, where:

- Q is the number of elevation angle samples for the actor.
- P is the number of azimuth angle samples for the actor.

If **Radar targets** is a vector (that is, multiple target actors), then specify **Radar cross sections (dBsm)** as a L-length cell array of real-valued Q_1-by-P_1, …, Q_L-by-P_L matrices, where:

- L is the number of actors.
- Q_1, …, Q_L are the number of elevation angle samples per actor.
- P_1, …, P_L are the number of azimuth angle samples per actor.

Q and P can vary for each actor. For each RCS matrix:

- The rows correspond to uniformly sampled elevation angles over the interval [0, 180].
- The columns correspond to uniformly sampled azimuth angles over the interval [0, 360].

For example, the number of elevation and azimuth samples for RCS matrix `RCS` are as follows:

```
el = linspace(0,180,size(RCS,1));
az = linspace(0,360,size(RCS,2));
```

`Default radar cross section (dBsm)` — Default radar cross section
`-20` (default) | real scalar

Default radar cross section, in decibels per square meter, specified as a real scalar. The block uses this RCS value for actors whose RCS is not specified by **Radar cross sections (dBsm)**.

Example: `-10`

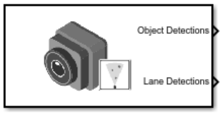
See Also
Simulation 3D Probabilistic Radar

Introduced in R2019b

Simulation 3D Vision Detection Generator

Detect objects and lanes from measurements in 3D simulation environment

Library: Automated Driving Toolbox / Simulation 3D



Description

The Simulation 3D Vision Detection Generator block generates detections from camera measurements taken by a vision sensor mounted on an ego vehicle in a 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The block derives detections from simulated actor poses that are based on cuboid (box-shaped) representations of the actors in the scenario. For more details, see "Algorithms" on page 2-167.

The block generates detections at intervals equal to the sensor update interval. Detections are referenced to the coordinate system of the sensor. The block can simulate real detections that have added random noise and also generate false positive detections. A statistical model generates the measurement noise, true detections, and false positives. To control the random numbers that the statistical model generates, use the random number generator settings on the **Measurements** tab of the block.

If you set **Sample time** to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block. To use this sensor, you must include a Simulation 3D Scene Configuration block in your model.

Note The Simulation 3D Scene Configuration block must execute before the Simulation 3D Vision Detection Generator block. That way, the Unreal Engine 3D visualization environment prepares the data before the Simulation 3D Vision Detection Generator block receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Vision Detection Generator — `1`

For more information about execution order, see "How Unreal Engine Simulation for Automated Driving Works".

Ports

Output

`Object Detections` — Object detections
Simulink bus containing MATLAB structure

Object detections, returned as a Simulink bus containing a MATLAB structure. For more details about buses, see "Create Nonvirtual Buses" (Simulink). The structure has the form shown in this table.

Field	Description	Type
NumDetections	Number of detections	Integer
IsValidTime	False when updates are requested at times that are between block invocation intervals	Boolean
Detections	Object detections	Array of object detection structures of length set by the **Maximum number of reported detections** parameter. Only NumDetections of these detections are actual detections.

The object detection structure contains these properties.

Property	Definition
Time	Measurement time
Measurement	Object measurements
MeasurementNoise	Measurement noise covariance matrix
SensorIndex	Unique ID of the sensor
ObjectClassID	Object classification
ObjectAttributes	Additional information passed to tracker
MeasurementParameters	Parameters used by initialization functions of nonlinear Kalman tracking filters

The Measurement field reports the position and velocity of a measurement in the coordinate system of the sensor. This field is a real-valued column vector of the form [*x*; *y*; *z*; *vx*; *vy*; *vz*]. The MeasurementNoise field is a 6-by-6 matrix that reports the measurement noise covariance for each coordinate in the Measurement field.

The MeasurementParameters field is a structure that has these fields.

Parameter	Definition
Frame	Enumerated type indicating the frame used to report measurements. The Simulation 3D Vision Detection Generator block reports detections in sensor Cartesian coordinates, which is a rectangular coordinate frame. Therefore, for this block, Frame is always set to 'rectangular'.
OriginPosition	Offset of the sensor origin from the ego vehicle origin, returned as a vector of the form [*x*, *y*, *z*]. The block derives these values from the *x*, *y*, and *z* mounting position of the sensor. For more details, see the **Mounting** parameters of this block.

Parameter	Definition
`Orientation`	Orientation of the sensor coordinate frame with respect to the ego vehicle coordinate frame, returned as a 3-by-3 real-valued orthonormal matrix. The block derives these values from the *yaw*, *pitch*, and *roll* mounting orientation of the sensor. For more details, see the **Mounting** parameters of this block.
`HasVelocity`	Indicates whether measurements contain velocity.

The `ObjectClassID` property of each detection has a value that corresponds to an object ID. The table shows the object IDs used in the default scenes that you can select from the Simulation 3D Scene Configuration block. If you are using a custom scene, in the Unreal Editor, you can assign new object types to unused IDs. If a scene contains an object that does not have an assigned ID, that object is assigned an ID of `0`. The block detects objects only of class `Vehicle`, such as vehicles created by using Simulation 3D Vehicle with Ground Following blocks, or of class `Road`.

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*

The `ObjectAttributes` property of each detection is a structure that has these fields.

Field	Definition
`TargetIndex`	Identifier of the actor, `ActorID`, that generated the detection. For false alarms, this value is negative.

Dependencies

To enable this output port, on the **Parameters** tab, set the **Types of detections generated by sensor** parameter to `Lanes and objects`, `Objects only`, or `Lanes with occlusion`.

`Lane Detections` — Lane boundary detections

Simulink bus containing MATLAB structure

Lane boundary detections, returned as a Simulink bus containing a MATLAB structure. The structure has these fields.

Field	Description	Type
`Time`	Lane detection time	Real scalar
`IsValidTime`	False when updates are requested at times that are between block invocation intervals	Boolean
`SensorIndex`	Unique identifier of sensor	Positive integer
`NumLaneBoundaries`	Number of lane boundary detections	Nonnegative integer
`LaneBoundaries`	Lane boundary detections	Array of `clothoidLaneBoundary` objects

Dependencies

To enable this output port, on the **Parameters** tab, set the **Types of detections generated by sensor** parameter to `Lanes and objects`, `Lanes only`, or `Lanes with occlusion`.

`Actor Truth` — Ground truth of actor poses

Simulink bus containing MATLAB structure

Ground truth of actor poses in the simulation environment, returned as a Simulink bus containing a MATLAB structure.

The structure has these fields.

Field	Description	Type
`NumActors`	Number of actors	Nonnegative integer
`Time`	Current simulation time	Real-valued scalar
`Actors`	Actor poses	`NumActors`-length array of actor pose structures

Each actor pose structure in `Actors` has these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
`Velocity`	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.

Field	Description
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (ω) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

The pose of the ego vehicle is excluded from the `Actors` array.

Dependencies

To enable this output port, on the **Ground Truth** tab, select the **Output actor truth** parameter.

Lane Truth — Ground truth of lane boundaries

Simulink bus containing MATLAB structure

Ground truth of lane boundaries in the simulation environment, returned as a Simulink bus containing a MATLAB structure.

The structure has these fields.

Field	Description	Type
`NumLaneBoundaries`	Number of lane boundaries	Nonnegative integer
`Time`	Current simulation time	Real scalar
`LaneBoundaries`	Lane boundaries	`NumLaneBoundaries`-length array of lane boundary structures

Each lane boundary structure in `LaneBoundaries` has these fields.

Field	Description

`Coordinates`	Lane boundary coordinates, specified as a real-valued *N*-by-3 matrix, where *N* is the number of lane boundary coordinates. Lane boundary coordinates define the position of points on the boundary at specified longitudinal distances away from the ego vehicle, along the center of the road. • In MATLAB, specify these distances by using the `'XDistance'` name-value pair argument of the `laneBoundaries` function. • In Simulink, specify these distances by using the **Distances from ego vehicle for computing boundaries (m)** parameter of the Scenario Reader block or the **Distance from parent for computing lane boundaries** parameter of the Simulation 3D Vision Detection Generator block. This matrix also includes the boundary coordinates at zero distance from the ego vehicle. These coordinates are to the left and right of the ego-vehicle origin, which is located under the center of the rear axle. Units are in meters.
`Curvature`	Lane boundary curvature at each row of the `Coordinates` matrix, specified as a real-valued *N*-by-1 vector. *N* is the number of lane boundary coordinates. Units are in radians per meter.
`CurvatureDerivative`	Derivative of lane boundary curvature at each row of the `Coordinates` matrix, specified as a real-valued *N*-by-1 vector. *N* is the number of lane boundary coordinates. Units are in radians per square meter.
`HeadingAngle`	Initial lane boundary heading angle, specified as a real scalar. The heading angle of the lane boundary is relative to the ego vehicle heading. Units are in degrees.
`LateralOffset`	Distance of the lane boundary from the ego vehicle position, specified as a real scalar. An offset to a lane boundary to the left of the ego vehicle is positive. An offset to the right of the ego vehicle is negative. Units are in meters.

`BoundaryType`	Type of lane boundary marking, specified as one of these values: • `'Unmarked'` — No physical lane marker exists • `'Solid'` — Single unbroken line • `'Dashed'` — Single line of dashed lane markers • `'DoubleSolid'` — Two unbroken lines • `'DoubleDashed'` — Two dashed lines • `'SolidDashed'` — Solid line on the left and a dashed line on the right • `'DashedSolid'` — Dashed line on the left and a solid line on the right
`Strength`	Saturation strength of the lane boundary marking, specified as a real scalar from 0 to 1. A value of `0` corresponds to a marking whose color is fully unsaturated. The marking is gray. A value of `1` corresponds to a marking whose color is fully saturated.
`Width`	Lane boundary width, specified as a positive real scalar. In a double-line lane marker, the same width is used for both lines and for the space between lines. Units are in meters.
`Length`	Length of dash in dashed lines, specified as a positive real scalar. In a double-line lane marker, the same length is used for both lines.
`Space`	Length of space between dashes in dashed lines, specified as a positive real scalar. In a dashed double-line lane marker, the same space is used for both lines.

The number of returned lane boundary structures depends on the **Maximum number of reported lanes** parameter value.

Dependencies

To enable this output port, on the **Ground Truth** tab, select the **Output lane truth** parameter.

Parameters

Mounting

`Sensor identifier` — Unique sensor identifier

`1` (default) | positive integer

Unique sensor identifier, specified as a positive integer. In a multisensor system, the sensor identifier distinguishes between sensors. When you add a new sensor block to your model, the **Sensor identifier** of that block is $N + 1$. N is the highest **Sensor identifier** value among existing sensor blocks in the model.

Example: 2

`Parent name` — Name of parent to which sensor is mounted
`Scene Origin` (default) | vehicle name

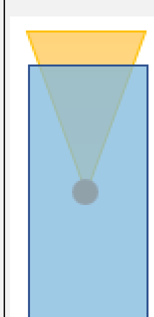
Name of the parent to which the sensor is mounted, specified as `Scene Origin` or as the name of a vehicle in your model. The vehicle names that you can select correspond to the **Name** parameters of the Simulation 3D Vehicle with Ground Following blocks in your model. If you select `Scene Origin`, the block places a sensor at the scene origin.

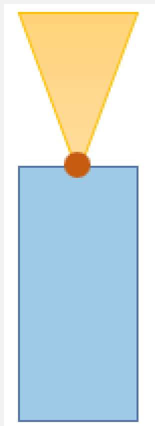
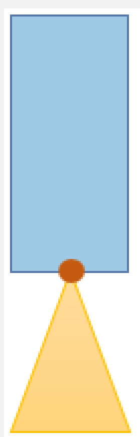
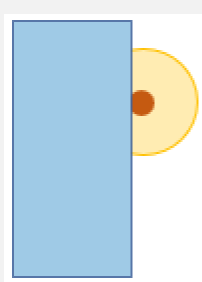
Example: `SimulinkVehicle1`

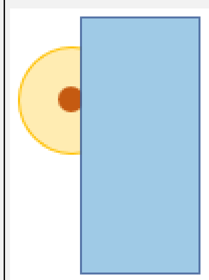
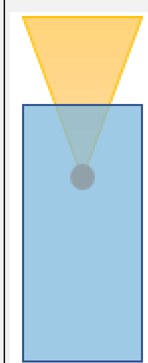
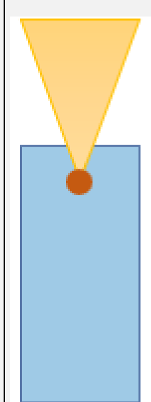
`Mounting location` — Sensor mounting location
`Origin` (default) | `Front bumper` | `Rear bumper` | `Right mirror` | `Left mirror` | `Rearview mirror` | `Hood center` | `Roof center`

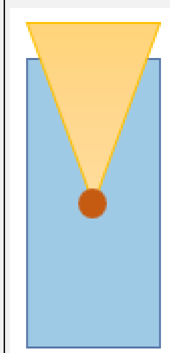
Sensor mounting location.

- When **Parent name** is `Scene Origin`, the block mounts the sensor to the origin of the scene. You can set the **Mounting location** to `Origin` only. During simulation, the sensor remains stationary.
- When **Parent name** is the name of a vehicle (for example, `SimulinkVehicle1`) the block mounts the sensor to one of the predefined mounting locations described in the table. During simulation, the sensor travels with the vehicle.

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Origin`	Forward-facing sensor mounted to the vehicle origin, which is on the ground, at the geometric center of the vehicle (see "Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox") 	[0, 0, 0]

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Front bumper`	Forward-facing sensor mounted to the front bumper 	[0, 0, 0]
`Rear bumper`	Backward-facing sensor mounted to the rear bumper 	[0, 0, 180]
`Right mirror`	Downward-facing sensor mounted to the right side-view mirror 	[0, –90, 0]

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Left mirror`	Downward-facing sensor mounted to the left side-view mirror 	[0, –90, 0]
`Rearview mirror`	Forward-facing sensor mounted to the rearview mirror, inside the vehicle 	[0, 0, 0]
`Hood center`	Forward-facing sensor mounted to the center of the hood 	[0, 0, 0]

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Roof center`	Forward-facing sensor mounted to the center of the roof 	[0, 0, 0]

Roll, pitch, and yaw are clockwise-positive when looking in the positive direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. When looking at a vehicle from the top down, the yaw angle (that is, the orientation angle) is counterclockwise-positive, because you are looking in the negative direction of the axis.

The (*X*, *Y*, *Z*) mounting location of the sensor relative to the vehicle depends on the vehicle type. To specify the vehicle type, use the **Type** parameter of the Simulation 3D Vehicle with Ground Following block to which you are mounting the sensor. To obtain the (*X*, *Y*, *Z*) mounting locations for a vehicle type, see the reference page for that vehicle.

To determine the location of the sensor in world coordinates, open the sensor block. Then, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)** and inspect the data from the **Location** output port.

`Specify offset` — Specify offset from mounting location
`off` (default) | `on`

Select this parameter to specify an offset from the mounting location by using the **Relative translation [X, Y, Z] (m)** and **Relative rotation [Roll, Pitch, Yaw] (deg)** parameters.

`Relative translation [X, Y, Z] (m)` — Translation offset relative to mounting location
`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Translation offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*X*, *Y*, *Z*]. Units are in meters.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see "Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox".

Example: `[0,0,0.01]`

Dependencies

To enable this parameter, select **Specify offset**.

`Relative rotation [Roll, Pitch, Yaw] (deg)` — Rotational offset relative to mounting location

`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Rotational offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*Roll*, *Pitch*, *Yaw*] . Roll, pitch, and yaw are the angles of rotation about the *X*-, *Y*-, and *Z*-axes, respectively. Units are in degrees.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.
- Roll, pitch, and yaw are clockwise-positive when looking in the forward direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. If you view a scene from a 2D top-down perspective, then the yaw angle (also called the orientation angle) is counterclockwise-positive because you are viewing the scene in the negative direction of the *Z*-axis.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see "Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox".

Example: `[0,0,10]`

Dependencies

To enable this parameter, select **Specify offset**.

`Sample time` — Sample time

`-1` (default) | positive scalar

Sample time of the block in seconds, specified as a positive scalar. The 3D simulation environment frame rate is the inverse of the sample time.

If you set the sample time to `-1`, the block inherits its sample time from the Simulation 3D Scene Configuration block.

Parameters

Detection Reporting

`Types of detections generated by sensor` — Types of detections generated by sensor
`Lanes and objects` (default) | `Objects only` | `Lanes only` | `Lanes with occlusion`

Types of detections generated by the sensor, specified as one of these options:

- `Lanes and objects` — Detect lanes and objects. No road information is used to occlude actors.
- `Objects only` — Detect objects only.
- `Lanes only` — Detection lanes only.
- `Lanes with occlusion` — Detect lane and objects. Objects in the camera field of view can impair the ability of the sensor to detect lanes.

`Maximum number of reported detections` — Maximum number of reported detections
`50` (default) | positive integer

Maximum number of detections reported by the sensor, specified as a positive integer. Detections are reported in order of increasing distance from the sensor until the maximum number is reached.

Example: `100`

`Maximum number of reported lanes` — Maximum number of reported lanes
`30` (default) | positive integer

Maximum number of reported lanes, specified as a positive integer.

Example: `100`

`Distance from parent for computing lane boundaries` — Distances from parent frame at which to compute lane boundaries
`0:0.5:9.5` (default) | *N*-element real-valued vector

Distances from the parent frame at which to compute the lane boundaries, specified as an *N*-element real-valued vector. *N* is the number of distance values. Units are in meters.

The parent is the frame to which the sensor is mounted, such as the ego vehicle. The **Parent name** parameter determines the parent frame. Distances are relative to the origin of the parent frame.

When detecting lanes from rear-facing cameras, specify negative distances. When detecting lanes from front-facing cameras, specify positive distances.

By default, the block computes a lane boundary every 0.5 meters over the range from 0 to 9.5 meters ahead of the parent.

Example: `1:0.1:10` computes a lane boundary every 0.1 meters over the range from 1 to 10 meters ahead of the parent.

Output Port Settings

`Source of object bus name` — Source of object bus name
`Auto` (default) | `Property`

Source of object bus name, specified as `Auto` or `Property`. If you select `Auto`, the block creates a bus name. If you select `Property`, specify the bus name by using the **Object bus name** parameter.

`Object bus name` — Object bus name
`BusObjectDetections` | valid bus name

Object bus name, specified as a valid bus name.

Dependencies

To enable this parameter, set the **Source of object bus name** parameter to `Property`.

`Source of output lane bus name` — Source of output lane bus name
`Auto` (default) | `Property`

Source of output lane bus name, specified as `Auto` or `Property`. If you select `Auto`, the block creates a bus name. If you select `Property`, specify the bus name by using the **Specify an output lane bus name** parameter.

`Specify an output lane bus name` — Lane bus name
`BusLaneDetections` (default) | valid bus name

Lane bus name, specified as a valid bus name.

Dependencies

To enable this parameter, set the **Source of output lane bus name** parameter to `Property`.

Measurements

`Maximum detection range (m)` — Maximum detection range
`150` (default) | positive real scalar

Maximum detection range, specified as a positive real scalar. The vision sensor cannot detect objects beyond this range. Units are in meters.

Example: `250`

Object Detector Settings

`Bounding box accuracy (pixels)` — Bounding box accuracy
`5` (default) | positive real scalar

Bounding box accuracy, specified as a positive real scalar. This quantity defines the accuracy with which the detector can match a bounding box to a target. Units are in pixels.

Example: `9`

`Smoothing filter noise intensity (m/s^2)` — Noise intensity used for filtering position and velocity measurements
`5` (default) | positive real scalar

Noise intensity used for filtering position and velocity measurements, specified as a positive real scalar. Noise intensity defines the standard deviation of the process noise of the internal constant-velocity Kalman filter used in a vision sensor. The filter models the process noise by using a piecewise-constant white noise acceleration model. Noise intensity is typically of the order of the maximum acceleration magnitude expected for a target. Units are in meters per second squared.

Example: 2

`Maximum detectable object speed (m/s)` — Maximum detectable object speed
`50` (default) | nonnegative real scalar

Maximum detectable object speed, specified as a nonnegative real scalar. Units are in meters per second.

Example: `20`

`Maximum allowed occlusion for detector` — Maximum allowed occlusion of an object
`0.5` (default) | real scalar in the range [0 1)

Maximum allowed occlusion of an object, specified as a real scalar in the range [0 1). Occlusion is the fraction of the total surface area of an object that is not visible to the sensor. A value of 1 indicates that the object is fully occluded. Units are dimensionless.

Example: `0.2`

`Minimum detectable image size of an object (pixels)` — Minimum height and width of an object
`[15,15]` (default) | 1-by-2 vector of positive values

Minimum height and width of an object that the vision sensor detects within an image, specified as a `[minHeight,minWidth]` vector of positive values. The 2-D projected height of an object must be greater than or equal to `minHeight`. The projected width of an object must be greater than or equal to `minWidth`. Units are in pixels.

Example: `[25 20]`

`Probability of detecting a target` — Probability of detection
`0.9` (default) | positive real scalar less than or equal to 1

Probability of detecting a target, specified as a positive real scalar less than or equal to 1. This quantity defines the probability that the sensor detects a detectable object. A detectable object is an object that satisfies the minimum detectable size, maximum range, maximum speed, and maximum allowed occlusion constraints.

Example: `0.95`

`Number of false positives per image` — Number of false detections generated by vision sensor per image
`0.1` (default) | nonnegative real scalar

Number of false detections generated by the vision sensor per image, specified as a nonnegative real scalar.

Example: `1.0`

Lane Detector Settings

`Minimum lane size in image (pixels)` — Maximum size of lane
`[20,3]` (default) | 1-by-2 real-valued vector

Minimum size of a projected lane marking in the camera image that the sensor can detect after accounting for curvature, specified as a 1-by-2 real-valued vector of the form `[minHeight, minWidth]`. Lane markings must exceed both of these values to be detected. Units are in pixels.

Accuracy of lane boundary (pixels) — Accuracy of lane boundary
3 (default) | positive real scalar

Accuracy of lane boundaries, specified as a positive real scalar. This parameter defines the accuracy with which the lane sensor can place a lane boundary. Units are in pixels.

Example: 2.5

Random Number Generator Settings

Add noise to measurements — Enable adding noise to vision sensor measurements
on (default) | off

Select this parameter to add noise to vision sensor measurements. Otherwise, the measurements are noise-free. The MeasurementNoise property of each detection is always computed and is not affected by the value you specify for the **Add noise to measurements** parameter.

Select method to specify initial seed — Method to specify random number generator seed
Repeatable (default) | Specify seed | Not repeatable

Method to set the random number generator seed, specified as one of the options in the table.

Option	Description
Repeatable	The block generates a random initial seed for the first simulation and reuses this seed for all subsequent simulations. Select this parameter to generate repeatable results from the statistical sensor model. To change this initial seed, at the MATLAB command prompt, enter: clear all.
Specify seed	Specify your own random initial seed for reproducible results by using the **Initial seed** parameter.
Not repeatable	The block generates a new random initial seed after each simulation run. Select this parameter to generate nonrepeatable results from the statistical sensor model.

Initial seed — Random number generator seed
1 (default) | nonnegative integer less than 2^{32}

Random number generator seed, specified as a nonnegative integer less than 2^{32}.

Example: 2001

Dependencies

To enable this parameter, set the **Select method to specify initial seed** parameter to Specify seed.

Camera Intrinsics

Focal length (pixels) — Camera focal length
[800,800] (default) | two-element real-valued vector

Camera focal length, in pixels, specified as a two-element real-valued vector. See also the `FocalLength` property of `cameraIntrinsics`.

Example: `[480,320]`

`Optical center (pixels)` — Optical center of camera
`[320,240]` (default) | two-element real-valued vector

Optical center of the camera, in pixels, specified as a two-element real-valued vector. See also the `PrincipalPoint` property of `cameraIntrinsics`.

Example: `[480,320]`

`Image size (pixels)` — Image size produced by camera
`[480,640]` (default) | two-element vector of positive integers

Image size produced by the camera, in pixels, specified as a two-element vector of positive integers. See also the `ImageSize` property of `cameraIntrinsics`.

Example: `[240,320]`

`Radial distortion coefficients` — Radial distortion coefficients
`[0,0]` (default) | two-element real-valued vector | three-element real-valued vector

Radial distortion coefficients, specified as a two-element or three-element real-valued vector. For details on setting these coefficients, see the `RadialDistortion` property of `cameraIntrinsics`.

Example: `[1,1]`

`Tangential distortion coefficients` — Tangential distortion coefficients
`[0,0]` (default) | two-element real-valued vector

Tangential distortion coefficients, specified as a two-element real-valued vector. For details on setting these coefficients, see the `TangentialDistortion` property of `cameraIntrinsics`.

Example: `[1,1]`

`Skew of the camera axes` — Skew angle of camera axes
`0` (default) | real scalar

Skew angle of the camera axes, specified as a real scalar. See also the `Skew` property of `cameraIntrinsics`.

Example: `0.1`

Ground Truth

`Output actor truth` — Output ground truth of actors
`off` (default) | `on`

Select this parameter to output the ground truth of actors on the **Actor Truth** output port.

`Output lane truth` — Output ground truth of lane boundaries
`off` (default) | `on`

Select this parameter to output the ground truth of lane boundaries on the **Lane Truth** output port.

Tips

- The sensor is unable to detect lanes and objects from vantage points too close to the ground. After mounting the sensor block to a vehicle by using the **Parent name** parameter, set the **Mounting location** parameter to one of the predefined mounting locations on the vehicle.

  If you leave **Mounting location** set to `Origin`, which mounts the sensor on the ground below the vehicle center, then specify an offset that is at least 0.1 meter above the ground. Select **Specify offset**, and in the **Relative translation [X, Y, Z] (m)** parameter, set a *Z* value of at least `0.1`.
- To visualize detections and sensor coverage areas, use the **Bird's-Eye Scope**. See “Visualize Sensor Data from Unreal Engine Simulation Environment”.
- Because the Unreal Engine can take a long time to start between simulations, consider logging the signals that the sensors output. See “Configure a Signal for Logging” (Simulink).

Algorithms

To generate detections, the Simulation 3D Vision Detection Generator block feeds the actor and lane ground truth data that is read from the Unreal Engine simulation environment to a Vision Detection Generator block. This block returns detections that are based on cuboid, or box-shaped, representations of the actors. The physical dimensions of detected actors are not based on their dimensions in the Unreal Engine environment. Instead, they are based on the default values set in the **Actor Profiles** parameter tab of the Vision Detection Generator block, where all actors are approximately the size of a sedan. If you return detections that have occlusions, then the occlusions are based on all actors being of this one size.

See Also

Apps
Bird's-Eye Scope

Blocks
Detection Concatenation | Multi-Object Tracker | Scenario Reader | Simulation 3D Probabilistic Radar | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following | Vision Detection Generator

Objects
`cameraIntrinsics` | `visionDetectionGenerator`

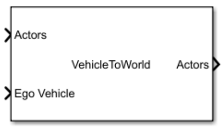
Topics
“Unreal Engine Simulation for Automated Driving”
“Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox”
“Choose a Sensor for Unreal Engine Simulation”
“Visualize Sensor Data and Tracks in Bird's-Eye Scope”

Introduced in R2020b

Vehicle To World

Convert actors from ego vehicle coordinates to world coordinates

Library: Automated Driving Toolbox / Driving Scenario and Sensor Modeling



Description

The Vehicle To World block converts actor poses from the vehicle coordinates of the input ego vehicle to world coordinates. Use this block to convert non-ego actor poses output by the Scenario Reader block into world coordinates for use with the 3D simulation environment. Before using these output poses to specify vehicle positions in the 3D environment, first convert them from the cuboid to the 3D simulation world coordinate system by using a Cuboid To 3D Simulation block. For an example of this workflow, see the “Visualize Sensor Data from Unreal Engine Simulation Environment” example.

Ports

Input

`Actors` — Actor poses in vehicle coordinates

Simulink bus containing MATLAB structure

Actor poses in vehicle coordinates, specified as a Simulink bus containing a MATLAB structure.

The structure must contain these fields.

Field	Description	Type
`NumActors`	Number of actors	Nonnegative integer
`Time`	Current simulation time	Real-valued scalar
`Actors`	Actor poses	`NumActors`-length array of actor pose structures

Each actor pose structure in `Actors` must contain these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
`Velocity`	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.

Field	Description
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (ω) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

`Ego Vehicle` — Ego vehicle pose

Simulink bus containing MATLAB structure

Ego vehicle pose, specified as a Simulink bus containing a MATLAB structure.

The structure must contain these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
`Velocity`	Velocity (v) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (ω) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

Output

`Actors` — Actor poses in world coordinates

Simulink bus containing MATLAB structure

Actor poses in world coordinates, returned as a Simulink bus containing a MATLAB structure.

The structure has these fields.

Field	Description	Type
`NumActors`	Number of actors	Nonnegative integer

Field	Description	Type
`Time`	Current simulation time	Real-valued scalar
`Actors`	Actor poses	`NumActors`-length array of actor pose structures

Each actor pose structure in `Actors` has these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
`Velocity`	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (*ω*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

Parameters

`Source of actors bus name` — Source of name for actor poses bus

`Auto` (default) | `Property`

Source of the name for the actor poses bus returned in the **Actors** output port, specified as one of these options:

- `Auto` — The block automatically creates an actor poses bus name.
- `Property` — Specify the actor poses bus name by using the **Actors bus name** parameter.

`Actors bus name` — Name of actor poses bus

valid bus name

Name of the actor poses bus returned in the **Actors** output port, specified as a valid bus name.

Dependencies

To enable this parameter, set **Source of actors bus name** to `Property`.

`Simulate using` — Type of simulation to run

`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.
- `Code generation` — Simulate the model using generated C/C++ code. The first time you run a simulation, Simulink generates C/C++ code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

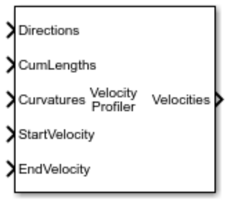
Cuboid To 3D Simulation | Scenario Reader | World To Vehicle

Topics
"Coordinate Systems in Automated Driving Toolbox"
"Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox"

Introduced in R2020a

Velocity Profiler

Generate velocity profile of vehicle path given kinematic constraints
Library: Automated Driving Toolbox



Description

The Velocity Profiler block generates a velocity profile of a driving path that satisfies this set of specified kinematic constraints:

- The maximum allowable speed of the vehicle
- The maximum longitudinal acceleration and deceleration of the vehicle
- The maximum longitudinal jerk on page 2-175 of the vehicle
- The maximum lateral acceleration on page 2-175 of the vehicle

Specify the cumulative lengths along the path and the driving directions and curvatures at each point along the path. You can obtain these values from the output of a Path Smoother Spline block. Also specify the longitudinal velocity of the vehicle at the start and end of the path.

Use the generated velocity profile as the input reference velocities of a longitudinal controller, as shown in the “Automated Parking Valet in Simulink” example.

Ports

Input

`Directions` — Driving directions along path
M-by-1 vector of 1s (forward motion) and –1s (reverse motion)

Driving directions of the vehicle along the length of the path, specified as an *M*-by-1 vector of 1s (forward motion) and –1s (reverse motion). Each vector element represents the driving direction of the vehicle at the corresponding cumulative path length specified by the **CumLengths** input port. *M* is the number of driving directions and must be equal to the lengths of the **CumLengths** and **Curvatures** inputs.

You can obtain **Directions** from the output of a Path Smoother Spline block.

`CumLengths` — Cumulative path lengths
M-by-1 vector of monotonically increasing real-valued elements

Cumulative path lengths, in meters, specified as an *M*-by-1 vector of monotonically increasing real-valued elements. Each vector element represents a point along the path. *M* is the number of cumulative path lengths and must be equal to the lengths of the **Directions** and **Curvatures** inputs.

You can obtain **CumLengths** from the output of a Path Smoother Spline block.

`Curvatures` — Signed path curvatures along path

M-by-1 real-valued vector

Signed path curvatures along the length of the path, in radians per meter, specified as an *M*-by-1 real-valued vector. Each vector element represents the curvature of the path at the corresponding cumulative path length specified by the **CumLengths** input port. *M* is the number of curvatures and must be equal to the lengths of the **Directions** and **CumLengths** inputs.

You can obtain **Curvatures** from the output of a Path Smoother Spline block.

`StartVelocity` — Longitudinal velocity of vehicle at start of path

real scalar

Longitudinal velocity of the vehicle at the start of the path, in meters per second, specified as a real scalar.

`EndVelocity` — Longitudinal velocity of vehicle at end of path

real scalar

Longitudinal velocity of the vehicle at the end of the path, in meters per second, specified as a real scalar.

Output

`Velocities` — Velocity profile along path

M-by-1 real-valued vector

Velocity profile along the length of the path, in meters per second, returned as an *M*-by-1 real-valued column vector. Each vector element represents a reference longitudinal velocity for the vehicle at the corresponding cumulative path length specified by the **CumLengths** input port. *M* is the number of velocities and is equal to the length of **CumLengths**.

The output velocity values satisfy the speed, acceleration, and jerk constraints specified in the parameters of the Velocity Profiler block. You can use this output as the reference velocity for a vehicle controller.

Velocities is a variable-size output with the limitations described in "Variable-Size Signal Limitations" (Simulink).

`Times` — Vehicle times of arrival for velocity profile

M-by-1 real-valued vector

Vehicle times of arrival for the velocity profile specified in **Velocities**, returned as an *M*-by-1 real-valued vector. *M* is the number of vehicle times of arrival and is equal to the length of **Velocities**. Units are in seconds.

Each vector element represents the time that a vehicle traveling at velocity *v* arrives at cumulative path length *p*, where:

- *v* is the corresponding velocity returned by the **Velocities** output port.
- *p* is the corresponding cumulative path length specified by the **CumLengths** input port.

Use **Times** to visualize the velocity profile over time, as shown in the “Velocity Profile of Straight Path” and “Velocity Profile of Path with Curve and Direction Change” examples.

Times is a variable-size output with the limitations described in “Variable-Size Signal Limitations” (Simulink).

Dependencies

To enable this port, select the **Show Times output port** parameter.

Parameters

`Maximum longitudinal acceleration (m/s^2)` — Maximum longitudinal acceleration of vehicle

`3` (default) | positive real scalar

Maximum longitudinal acceleration of the vehicle, in meters per second squared, specified as a positive real scalar.

When developing a longitudinal controller, this parameter must be equal to the corresponding parameter in the Longitudinal Controller Stanley block. Otherwise, the vehicle is unable to run the generated velocity profile.

`Maximum longitudinal deceleration (m/s^2)` — Maximum longitudinal deceleration of vehicle

`6` (default) | positive real scalar

Maximum longitudinal deceleration of the vehicle, in meters per second squared, specified as a positive real scalar.

When developing a longitudinal controller, this parameter must be equal to the corresponding parameter in the Longitudinal Controller Stanley block. Otherwise, the vehicle is unable to run the generated velocity profile.

`Maximum allowable speed (m/s)` — Maximum allowable speed along path

`10` (default) | positive real scalar

Maximum allowable speed of the vehicle along the path, in meters per second, specified as a positive real scalar. Use this parameter to constrain the speed of the vehicle based on passenger comfort or speed limit requirements.

When the path length is too short for the vehicle to reach this maximum speed, the block calculates a smaller maximum speed that satisfies the path length constraint.

In the output velocity profile, the speed of the vehicle is constrained to [$-V_{max}$, V_{max}], where V_{max} is the value of this parameter.

`Maximum longitudinal jerk (m/s^3)` — Maximum longitudinal jerk

`1` (default) | positive real scalar

Maximum longitudinal jerk of the vehicle along the path, in meters per second cubed, specified as a positive real scalar.

In the output velocity profile, the longitudinal jerk of the vehicle is constrained to [$-J_{max}$, J_{max}], where J_{max} is the value of this parameter.

`Maximum lateral acceleration (m/s^2)` — Maximum lateral acceleration
`1` (default) | positive real scalar

Maximum lateral acceleration of the vehicle along the path, in meters per second squared, specified as a positive real scalar.

In the output velocity profile, the lateral acceleration of the vehicle is constrained to [$-A_{max}$, A_{max}], where A_{max} is the value of this parameter.

`Show Times output port` — Output times of arrival for velocity profile
`off` (default) | `on`

Select this parameter to enable the **Times** output port.

`Sample time` — Sample time of block
`-1` (default) | positive real scalar

Sample time of the block, in seconds, specified as `-1` or as a positive real scalar. The default of `-1` means that the block inherits its sample time from upstream blocks.

Because the Velocity Profiler block outputs variable-size signals, the sample time of the block must be discrete (nonzero). If the block inherits its sample time from upstream blocks, those blocks must also have discrete sample times.

`Simulate using` — Type of simulation to run
`Code Generation` (default) | `Interpreted Execution`

- `Code generation` — Simulate the model using generated C/C++ code. The first time you run a simulation, Simulink generates C/C++ code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.
- `Interpreted execution` — Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.

More About

Jerk

Jerk is the rate of change of acceleration in a vehicle. Jerk minimization is a key comfort requirement for vehicle passengers. Rapid changes in acceleration or deceleration result in a "jerky" ride for passengers. Jerk is measured in units of meters per second cubed.

Lateral Acceleration

Lateral acceleration is defined as $a_{lat} = v^2\kappa$, where:

- v is the longitudinal velocity of the vehicle.
- κ is the curvature of the path. Units are in radians per meter.

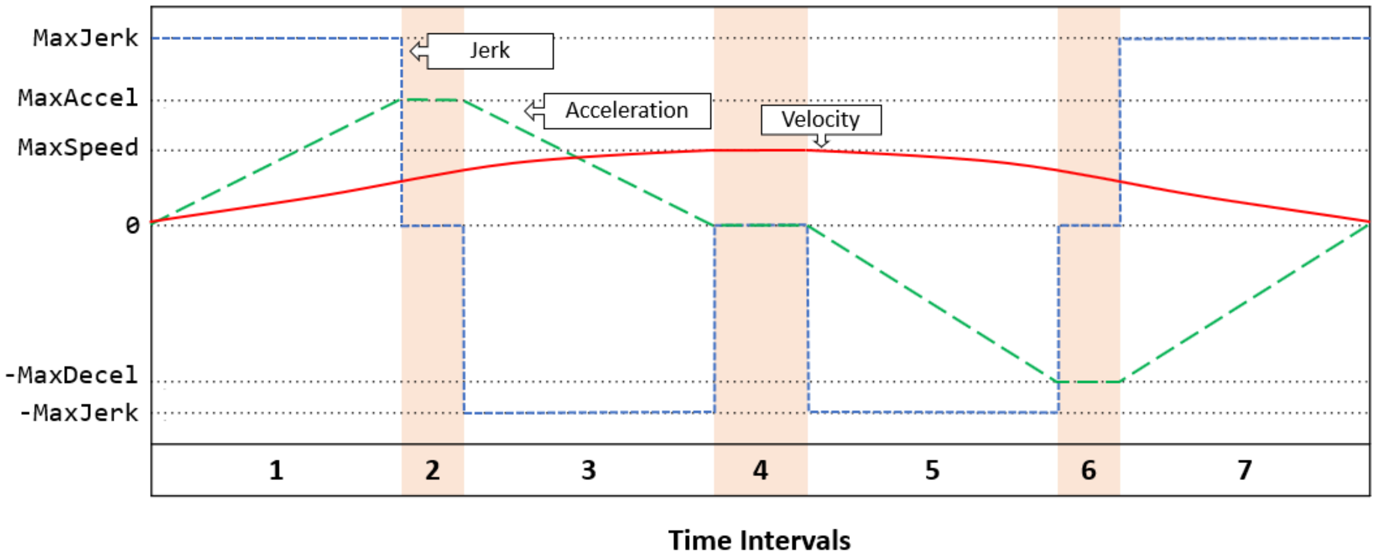
Lateral acceleration is measured in units of meters per second squared.

Algorithms

To generate the velocity profile for a reference path, the Velocity Profiler block performs these steps:

1. Generate a continuous velocity profile that satisfies all kinematic constraints (speed, acceleration, and jerk) specified by the block parameters.
2. Discretize the velocity profile by mapping poses in the reference path to velocity values, based on how far away the poses are from the starting pose. The cumulative path lengths specified in the **CumLengths** input port contain these distances. The Path Smoother Spline block returns these cumulative path lengths, along with the smooth path.

The generated velocity profile is a seven-interval curve. At each time interval within the curve, the jerk, acceleration, and velocity of the vehicle change to satisfy the specified constraints. The figure and table show how these values change for a vehicle traveling in forward motion along a path. For simplicity, the starting and ending velocity of the vehicle, as specified by the **StartVelocity** and **EndVelocity** input ports, are both `0`.



Time Interval	Jerk	Acceleration	Velocity	Notes
1	Set to `MaxJerk`	Increases from `0` to `MaxAccel`	Increases from starting velocity	-
2	Set to `0`	Held constant at `MaxAccel`	Keeps increasing	During the previous interval, if the vehicle cannot reach `MaxAccel` given the `MaxSpeed` constraint, then interval 2 does not occur.
3	Set to `-MaxJerk`	Decreases from `MaxAccel` to `0`	Increases to `MaxSpeed`	-
4	Set to `0`	Held constant at `0`	Held constant at `MaxSpeed`	-
5	Set to `-MaxJerk`	Decreases from `0` to `-MaxDecel`	Starts decreasing	-

Time Interval	Jerk	Acceleration	Velocity	Notes
6	Set to `0`	Held constant at `-MaxDecel`	Keeps decreasing	During the previous interval, if the vehicle cannot reach `-MaxDecel` given the `MaxSpeed` constraint, then interval 6 does not occur.
7	Set to `MaxJerk`	Increases from `-MaxDecel` to `0`	Decreases to ending velocity	-

In the figure and table:

- `MaxJerk` and `-MaxJerk` are set by the **Maximum longitudinal jerk (m/s^3)** parameter.
- `MaxAccel` and `-MaxDecel` are set by the **Maximum longitudinal acceleration (m/s^2)** and **Maximum longitudinal deceleration (m/s^2)** parameters, respectively. You can specify asymmetric values for these parameters.
- `MaxSpeed` is set by the **Maximum allowable speed (m/s)** parameter.

For a vehicle in reverse motion, the curves in the figure are reversed. The signs of the parameter values shown in the figure and table are also reversed.

If the vehicle includes multiple changes in direction, the block generates separate velocity profiles for each driving direction. Then the block concatenates these profiles in the final **Velocities** output. For an example, see “Velocity Profile of Path with Curve and Direction Change”.

References

[1] Villagra, Jorge, Vicente Milanés, Joshué Pérez, and Jorge Godoy. "Smooth path and speed planning for an automated public transport vehicle." *Robotics and Autonomous Systems*. Vol. 60, Number 2, February 2012, pp. 252–265.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

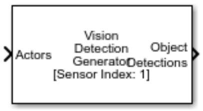
Lateral Controller Stanley | Longitudinal Controller Stanley | Path Smoother Spline

Introduced in R2019b

Vision Detection Generator

Detect objects and lanes from visual measurements

Library: Automated Driving Toolbox / Driving Scenario and Sensor Modeling



Description

The Vision Detection Generator block generates detections from camera measurements taken by a vision sensor mounted on an ego vehicle.

The block derives detections from simulated actor poses and generates these detections at intervals equal to the sensor update interval. By default, detections are referenced to the coordinate system of the ego vehicle. The block can simulate real detections with added random noise and also generate false positive detections. A statistical model generates the measurement noise, true detections, and false positives. To control the random numbers that the statistical model generates, use the random number generator settings on the **Measurements** tab of the block.

You can use the Vision Detection Generator to create input to a Multi-Object Tracker block. When building scenarios and sensor models using the **Driving Scenario Designer** app, the camera sensors exported to Simulink are output as Vision Detection Generator blocks.

Ports

Input

`Actors` — Scenario actor poses

Simulink bus containing MATLAB structure

Scenario actor poses in ego vehicle coordinates, specified as a Simulink bus containing a MATLAB structure.

The structure must contain these fields.

Field	Description	Type
`NumActors`	Number of actors	Nonnegative integer
`Time`	Current simulation time	Real-valued scalar
`Actors`	Actor poses	`NumActors`-length array of actor pose structures

Each actor pose structure in `Actors` must have these fields.

Field	Description
ActorID	Scenario-defined actor identifier, specified as a positive integer.
Position	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
Velocity	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
Roll	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
Pitch	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
Yaw	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
AngularVelocity	Angular velocity (*ω*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

Dependencies

To enable this input port, set the **Types of detections generated by sensor** parameter to `Objects only`, `Lanes with occlusion`, or `Lanes and objects`.

Lane Boundaries — Lane boundaries

Simulink bus containing MATLAB structure

Lane boundaries in ego vehicle coordinates, specified as a Simulink bus containing a MATLAB structure.

The structure must contain these fields.

Field	Description	Type
NumLaneBoundaries	Number of lane boundaries	Nonnegative integer
Time	Current simulation time	Real scalar
LaneBoundaries	Lane boundaries	`NumLaneBoundaries`-length array of lane boundary structures

Each lane boundary structure in `LaneBoundaries` must have these fields.

Field	Description

`Coordinates`	Lane boundary coordinates, specified as a real-valued *N*-by-3 matrix, where *N* is the number of lane boundary coordinates. Lane boundary coordinates define the position of points on the boundary at specified longitudinal distances away from the ego vehicle, along the center of the road. • In MATLAB, specify these distances by using the `'XDistance'` name-value pair argument of the `laneBoundaries` function. • In Simulink, specify these distances by using the **Distances from ego vehicle for computing boundaries (m)** parameter of the Scenario Reader block or the **Distance from parent for computing lane boundaries** parameter of the Simulation 3D Vision Detection Generator block. This matrix also includes the boundary coordinates at zero distance from the ego vehicle. These coordinates are to the left and right of the ego-vehicle origin, which is located under the center of the rear axle. Units are in meters.
`Curvature`	Lane boundary curvature at each row of the `Coordinates` matrix, specified as a real-valued *N*-by-1 vector. *N* is the number of lane boundary coordinates. Units are in radians per meter.
`CurvatureDerivative`	Derivative of lane boundary curvature at each row of the `Coordinates` matrix, specified as a real-valued *N*-by-1 vector. *N* is the number of lane boundary coordinates. Units are in radians per square meter.
`HeadingAngle`	Initial lane boundary heading angle, specified as a real scalar. The heading angle of the lane boundary is relative to the ego vehicle heading. Units are in degrees.
`LateralOffset`	Distance of the lane boundary from the ego vehicle position, specified as a real scalar. An offset to a lane boundary to the left of the ego vehicle is positive. An offset to the right of the ego vehicle is negative. Units are in meters.

`BoundaryType`	Type of lane boundary marking, specified as one of these values: • `'Unmarked'` — No physical lane marker exists • `'Solid'` — Single unbroken line • `'Dashed'` — Single line of dashed lane markers • `'DoubleSolid'` — Two unbroken lines • `'DoubleDashed'` — Two dashed lines • `'SolidDashed'` — Solid line on the left and a dashed line on the right • `'DashedSolid'` — Dashed line on the left and a solid line on the right
`Strength`	Saturation strength of the lane boundary marking, specified as a real scalar from 0 to 1. A value of `0` corresponds to a marking whose color is fully unsaturated. The marking is gray. A value of `1` corresponds to a marking whose color is fully saturated.
`Width`	Lane boundary width, specified as a positive real scalar. In a double-line lane marker, the same width is used for both lines and for the space between lines. Units are in meters.
`Length`	Length of dash in dashed lines, specified as a positive real scalar. In a double-line lane marker, the same length is used for both lines.
`Space`	Length of space between dashes in dashed lines, specified as a positive real scalar. In a dashed double-line lane marker, the same space is used for both lines.

Dependencies

To enable this input port, set the **Types of detections generated by sensor** parameter to `Lanes only`, `Lanes only`, `Lanes with occlusion`, or `Lanes and objects`.

Output

`Object Detections` — Object detections
Simulink bus containing MATLAB structure

Object detections, returned as a Simulink bus containing a MATLAB structure. For more details about buses, see “Create Nonvirtual Buses” (Simulink).

You can pass object detections from these sensors and other sensors to a tracker, such as a Multi-Object Tracker block, and generate tracks.

The detections structure has this form:

Field	Description	Type
`NumDetections`	Number of detections	Integer
`IsValidTime`	False when updates are requested at times that are between block invocation intervals	Boolean
`Detections`	Object detections	Array of object detection structures of length set by the **Maximum number of reported detections** parameter. Only `NumDetections` of these detections are actual detections.

The object detection structure contains these properties.

Property	Definition
`Time`	Measurement time
`Measurement`	Object measurements
`MeasurementNoise`	Measurement noise covariance matrix
`SensorIndex`	Unique ID of the sensor
`ObjectClassID`	Object classification
`ObjectAttributes`	Additional information passed to tracker
`MeasurementParameters`	Parameters used by initialization functions of nonlinear Kalman tracking filters

The `Measurement` field reports the position and velocity of a measurement in the coordinate system specified by **Coordinate system used to report detections**. This field is a real-valued column vector of the form [*x*; *y*; *z*; *vx*; *vy*; *vz*]. Units are in meters per second.

The `MeasurementNoise` field is a 6-by-6 matrix that reports the measurement noise covariance for each coordinate in the `Measurement` field.

The `MeasurementParameters` field is a structure with these fields.

Parameter	Definition
`Frame`	Enumerated type indicating the frame used to report measurements. The Vision Detection Generator block reports detections in either ego and sensor Cartesian coordinates, which are both rectangular coordinate frames. Therefore, for this block, `Frame` is always set to `'rectangular'`.
`OriginPosition`	3-D vector offset of the sensor origin from the ego vehicle origin. The vector is derived from the **Sensor's (x,y) position (m)** and **Sensor's height (m)** parameters of the block.

Parameter	Definition
Orientation	Orientation of the vision sensor coordinate system with respect to the ego vehicle coordinate system. The orientation is derived from the **Yaw angle of sensor mounted on ego vehicle (deg)**, **Pitch angle of sensor mounted on ego vehicle (deg)**, and **Roll angle of sensor mounted on ego vehicle (deg)** parameters of the block.
HasVelocity	Indicates whether measurements contain velocity.

The ObjectAttributes property of each detection is a structure with these fields.

Field	Definition
TargetIndex	Identifier of the actor, ActorID, that generated the detection. For false alarms, this value is negative.

Dependencies

To enable this output port, set the **Types of detections generated by sensor** parameter to Objects only, Lanes with occlusion, or Lanes and objects.

Lane Detections — Lane boundary detections

Simulink bus containing MATLAB structure

Lane boundary detections, returned as a Simulink bus containing a MATLAB structure. The structure had these fields:

Field	Description	Type
Time	Lane detection time	Real scalar
IsValidTime	False when updates are requested at times that are between block invocation intervals	Boolean
SensorIndex	Unique identifier of sensor	Positive integer
NumLaneBoundaries	Number of lane boundary detections	Nonnegative integer
LaneBoundaries	Lane boundary detections	Array of clothoidLaneBoundary objects

Dependencies

To enable this output port, set the **Types of detections generated by sensor** parameter to Lanes only, Lanes with occlusion, or Lanes and objects.

Parameters

Parameters

Sensor Identification

`Unique identifier of sensor` — Unique sensor identifier
`1` (default) | positive integer

Unique sensor identifier, specified as a positive integer. The sensor identifier distinguishes detections that come from different sensors in a multisensor system. If a model contains multiple sensor blocks with the same sensor identifier, the **Bird's-Eye Scope** displays an error.

Example: `5`

`Types of detections generated by sensor` — Select the types of detections
`Objects only` (default) | `Lanes only` | `Lanes with occlusion` | `Lanes and objects`

Types of detections generated by the sensor, specified as `Objects only`, `Lanes only`, `Lanes with occlusion`, or `Lanes and objects`.

- When set to `Objects only`, no road information is used to occlude actors.
- When set to `Lanes only`, no actor information is used to detect lanes.
- When set to `Lanes with occlusion`, actors in the camera field of view can impair the sensor ability to detect lanes.
- When set to `Lanes and objects`, the sensor generates object both object detections and occluded lane detections.

`Required interval between sensor updates (s)` — Required time interval
`0.1` (default) | positive real scalar

Required time interval between sensor updates, specified as a positive real scalar. The value of this parameter must be an integer multiple of the **Actors** input port data interval. Updates requested from the sensor between update intervals contain no detections. Units are in seconds.

`Required interval between lane detections updates (s)` — Time interval between lane detection updates
0.1 (default) | positive real scalar

Required time interval between lane detection updates, specified as a positive real scalar. The vision detection generator is called at regular time intervals. The vision detector generates new lane detections at intervals defined by this parameter which must be an integer multiple of the simulation time interval. Updates requested from the sensor between update intervals contain no lane detections. Units are in seconds.

Sensor Extrinsics

`Sensor's (x,y) position (m)` — Location of the vision sensor center
`[3.4 0]` (default) | real-valued 1-by-2 vector

Location of the vision sensor center, specified as a real-valued 1-by-2 vector. The **Sensor's (x,y) position (m)** and **Sensor's height (m)** parameters define the coordinates of the vision sensor with respect to the ego vehicle coordinate system. The default value corresponds to a forward-facing vision sensor mounted to a sedan dashboard. Units are in meters.

`Sensor's height (m)` — Vision sensor height above the ground plane
`0.2` (default) | positive real scalar

Vision sensor height above the ground plane, specified as a positive real scalar. The height is defined with respect to the vehicle ground plane. The **Sensor's (x,y) position (m)** and **Sensor's height (m)** parameters define the coordinates of the vision sensor with respect to the ego vehicle coordinate system. The default value corresponds to a forward-facing vision sensor mounted a sedan dashboard. Units are in meters.

Example: `0.25`

`Yaw angle of sensor mounted on ego vehicle (deg)` — Yaw angle of sensor
`0` (default) | real scalar

Yaw angle of vision sensor, specified as a real scalar. Yaw angle is the angle between the center line of the ego vehicle and the optical axis of the camera. A positive yaw angle corresponds to a clockwise rotation when looking in the positive direction of the *z*-axis of the ego vehicle coordinate system. Units are in degrees.

Example: `-4.0`

`Pitch angle of sensor mounted on ego vehicle (deg)` — Pitch angle of sensor
`0` (default) | real scalar

Pitch angle of sensor, specified as a real scalar. The pitch angle is the angle between the optical axis of the camera and the *x*-*y* plane of the ego vehicle coordinate system. A positive pitch angle corresponds to a clockwise rotation when looking in the positive direction of the *y*-axis of the ego vehicle coordinate system. Units are in degrees.

Example: `3.0`

`Roll angle of sensor mounted on ego vehicle (deg)` — Roll angle of sensor
`0` (default) | real scalar

Roll angle of the vision sensor, specified as a real scalar. The roll angle is the angle of rotation of the optical axis of the camera around the *x*-axis of the ego vehicle coordinate system. A positive roll angle corresponds to a clockwise rotation when looking in the positive direction of the *x*-axis of the coordinate system. Units are in degrees.

Output Port Settings

`Source of object bus name` — Source of object bus name
`Auto` (default) | `Property`

Source of object bus name, specified as `Auto` or `Property`. If you select `Auto`, the block automatically creates a bus name. If you select `Property`, specify the bus name using the **Specify an object bus name** parameter.

Example: `Property`

`Source of output lane bus name` — Source of lane bus name
`Auto` (default) | `Property`

Source of output lane bus name, specified as `Auto` or `Property`. If you choose `Auto`, the block will automatically create a bus name. If you choose `Property`, specify the bus name using the **Specify an object bus name** parameter.

Example: `Property`

`Object bus name` — Name of object bus
valid bus name

Name of object bus, specified as a valid bus name.

Example: `objectbus`

Dependencies

To enable this parameter, set the **Source of object bus name** parameter to `Property`.

`Specify an output lane bus name` — Name of output lane bus
valid bus name

Namer of output lane bus, specified as a valid bus name.

Example: `lanebus`

Dependencies

To enable this parameter, set the **Source of output lane bus name** parameter to `Property`.

Detection Reporting

`Maximum number of reported detections` — Maximum number of reported detections
`50` (default) | positive integer

Maximum number of detections reported by the sensor, specified as a positive integer. Detections are reported in order of increasing distance from the sensor until the maximum number is reached.

Example: `100`

Dependencies

To enable this parameter, set the **Types of detections generated by sensor** parameter to `Objects only` or `Lanes and objects`.

`Maximum number of reported lanes` — Maximum number of reported lanes
`30` (default) | positive integer

Maximum number of reported lanes, specified as a positive integer.

Example: `100`

Dependencies

To enable this parameter, set the **Types of detections generated by sensor** parameter to `Lanes only`, `Lanes with occlusion`, or `Lanes and objects`.

`Coordinate system used to report detections` — Coordinate system of reported detections
`Ego Cartesian` (default) | `Sensor Cartesian`

Coordinate system of reported detections, specified as one of these values:

- `Ego Cartesian` — Detections are reported in the ego vehicle Cartesian coordinate system.

- `Sensor Cartesian`— Detections are reported in the sensor Cartesian coordinate system.

Simulation

`Simulate using` — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.
- `Code generation` — Simulate the model using generated C/C++ code. The first time you run a simulation, Simulink generates C/C++ code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.

Measurements

Settings

`Maximum detection range (m)` — Maximum detection range
`150` (default) | positive real scalar

Maximum detection range, specified as a positive real scalar. The vision sensor cannot detect objects beyond this range. Units are in meters.

Example: `250`

Object Detector Settings

`Bounding box accuracy (pixels)` — Bounding box accuracy
`5` (default) | positive real scalar

Bounding box accuracy, specified as a positive real scalar. This quantity defines the accuracy with which the detector can match a bounding box to a target. Units are in pixels.

Example: `9`

`Smoothing filter noise intensity (m/s^2)` — Noise intensity used for filtering position and velocity measurements
`5` (default) | positive real scalar

Noise intensity used for filtering position and velocity measurements, specified as a positive real scalar. Noise intensity defines the standard deviation of the process noise of the internal constant-velocity Kalman filter used in a vision sensor. The filter models the process noise using a piecewise-constant white noise acceleration model. Noise intensity is typically of the order of the maximum acceleration magnitude expected for a target. Units are in meters per second squared.

Example: `2`

`Maximum detectable object speed (m/s)` — Maximum detectable object speed
`100` (default) | nonnegative real scalar

Maximum detectable object speed, specified as a nonnegative real scalar. Units are in meters per second.

Example: `20`

`Maximum allowed occlusion for detector` — Maximum allowed occlusion for detector
`0.5` (default) | real scalar in the range [0 1]

Maximum allowed occlusion of an object, specified as a real scalar in the range [0 1). Occlusion is the fraction of the total surface area of an object that is not visible to the sensor. A value of 1 indicates that the object is fully occluded. Units are dimensionless.

Example: `0.2`

`Minimum detectable image size of an object` — Minimum height and width of an object
`[15,15]` (default) | 1-by-2 vector of positive values

Minimum height and width of an object that the vision sensor detects within an image, specified as a `[minHeight,minWidth]` vector of positive values. The 2-D projected height of an object must be greater than or equal to `minHeight`. The projected width of an object must be greater than or equal to `minWidth`. Units are in pixels.

Example: `[25 20]`

`Probability of detecting a target` — Probability of detection
`0.9` (default) | positive real scalar less than or equal to 1

Probability of detecting a target, specified as a positive real scalar less than or equal to 1. This quantity defines the probability that the sensor detects a detectable object. A detectable object is an object that satisfies the minimum detectable size, maximum range, maximum speed, and maximum allowed occlusion constraints.

Example: `0.95`

`Number of false positives per image` — Number of false detections generated by vision sensor per image
`0.1` (default) | nonnegative real scalar

Number of false detections generated by the vision sensor per image, specified as a nonnegative real scalar.

Example: `1.0`

Lane Detector Settings

`Minimum lane size in image (pixels)` — Maximum size of lane
`[20,3]` (default) | 1-by-2 real-valued vector

Minimum size of a projected lane marking in the camera image that can be detected by the sensor after accounting for curvature, specified as a 1-by-2 real-valued vector, `[minHeight minWidth]`. Lane markings must exceed both of these values to be detected. Units are in pixels.

Dependencies

To enable this parameter, set the **Types of detections generated by sensor** parameter to `Lanes only`, `Lanes only`, or `Lanes and objects`.

`Accuracy of lane boundary (pixels)` — Accuracy of lane boundary
3 (default) | positive real scalar

Accuracy of lane boundaries, specified as a positive real scalar. This parameter defines the accuracy with which the lane sensor can place a lane boundary. Units are in pixels.

Example: `2.5`

Dependencies

To enable this parameter, set the **Types of detections generated by sensor** parameter to `Lanes only`, `Lanes only`, or `Lanes and objects`.

Random Number Generator Settings

`Add noise to measurements` — Enable adding noise to vision sensor measurements
`on` (default) | `off`

Select this parameter to add noise to vision sensor measurements. Otherwise, the measurements are noise-free. The `MeasurementNoise` property of each detection is always computed and is not affected by the value you specify for the **Add noise to measurements** parameter.

`Select method to specify initial seed` — Method to specify random number generator seed
`Repeatable` (default) | `Specify seed` | `Not repeatable`

Method to set the random number generator seed, specified as one of the options in the table.

Option	Description
`Repeatable`	The block generates a random initial seed for the first simulation and reuses this seed for all subsequent simulations. Select this parameter to generate repeatable results from the statistical sensor model. To change this initial seed, at the MATLAB command prompt, enter: `clear all`.
`Specify seed`	Specify your own random initial seed for reproducible results by using the **Specify seed** parameter.
`Not repeatable`	The block generates a new random initial seed after each simulation run. Select this parameter to generate nonrepeatable results from the statistical sensor model.

`Initial seed` — Random number generator seed
`0` (default) | nonnegative integer less than 2^{32}

Random number generator seed, specified as a nonnegative integer less than 2^{32}.

Example: `2001`

Dependencies

To enable this parameter, set the **Random Number Generator Settings** parameter to `Specify seed`.

Actor Profiles

`Select method to specify actor profiles` — Method to specify actor profiles
`Parameters` (default) | `MATLAB expression`

Method to specify actor profiles, specified as `Parameters` or `MATLAB expression`. When you select `Parameters`, set the actor profiles using the parameters in the **Actor Profiles** tab. When you select

`MATLAB expression`, set the actor profiles using the **MATLAB expression for actor profiles** parameter.

`MATLAB expression for actor profiles` — MATLAB expression for actor profiles
`struct('ClassID',0,'Length',4.7,'Width',1.8,'Height',1.4,'OriginOffset',[-1.35,0,0])` (default) | MATLAB structure | MATLAB structure array | valid MATLAB expression

MATLAB expression for actor profiles, specified as a MATLAB structure, a MATLAB structure array, or a valid MATLAB expression that produces such a structure or structure array.

If your Scenario Reader block reads data from a `drivingScenario` object, to obtain the actor profiles directly from this object, set this expression to call the `actorProfiles` function on the object. For example: `actorProfiles(scenario)`.

Example: `struct('ClassID',5,'Length',5.0,'Width',2,'Height',2,'OriginOffset',[-1.55,0,0])`

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `MATLAB expression`.

`Unique identifier for actors` — Scenario-defined actor identifier
`[]` (default) | positive integer | length-*L* vector of unique positive integers

Scenario-defined actor identifier, specified as a positive integer or length-*L* vector of unique positive integers. *L* must equal the number of actors input into the **Actors** input port. The vector elements must match `ActorID` values of the actors. You can specify **Unique identifier for actors** as `[]`. In this case, the same actor profile parameters apply to all actors.

Example: `[1,2]`

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `Parameters`.

`User-defined integer to classify actors` — User-defined classification identifier
`0` (default) | integer | length-*L* vector of integers

User-defined classification identifier, specified as an integer or length-*L* vector of integers. When **Unique identifier for actors** is a vector, this parameter is a vector of the same length with elements in one-to-one correspondence to the actors in **Unique identifier for actors**. When **Unique identifier for actors** is empty, `[]`, you must specify this parameter as a single integer whose value applies to all actors.

Example: 2

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `Parameters`.

`Length of actors cuboids (m)` — Length of cuboid
`4.7` (default) | positive real scalar | length-*L* vector of positive values

Length of cuboid, specified as a positive real scalar or length-*L* vector of positive values. When **Unique identifier for actors** is a vector, this parameter is a vector of the same length with elements in one-to-one correspondence to the actors in **Unique identifier for actors**. When **Unique identifier for actors** is empty, `[]`, you must specify this parameter as a positive real scalar whose value applies to all actors. Units are in meters.

Example: 6.3

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `Parameters`.

`Width of actors cuboids (m)` — Width of cuboid

`4.7` (default) | positive real scalar | length-*L* vector of positive values

Width of cuboid, specified as a positive real scalar or length-*L* vector of positive values. When **Unique identifier for actors** is a vector, this parameter is a vector of the same length with elements in one-to-one correspondence to the actors in **Unique identifier for actors**. When **Unique identifier for actors** is empty, `[]`, you must specify this parameter as a positive real scalar whose value applies to all actors. Units are in meters.

Example: 4.7

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `Parameters`.

`Height of actors cuboids (m)` — Height of cuboid

`4.7` (default) | positive real scalar | length-*L* vector of positive values

Height of cuboid, specified as a positive real scalar or length-*L* vector of positive values. When **Unique identifier for actors** is a vector, this parameter is a vector of the same length with elements in one-to-one correspondence to the actors in **Unique identifier for actors**. When **Unique identifier for actors** is empty, `[]`, you must specify this parameter as a positive real scalar whose value applies to all actors. Units are in meters.

Example: `2.0`

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `Parameters`.

`Rotational center of actors from bottom center (m)` — Rotational center of the actor

{ [-1.35, 0, 0] } (default) | length-*L* cell array of real-valued 1-by-3 vectors

Rotational center of the actor, specified as a length-*L* cell array of real-valued 1-by-3 vectors. Each vector represents the offset of the rotational center of the actor from the bottom-center of the actor. For vehicles, the offset corresponds to the point on the ground beneath the center of the rear axle. When **Unique identifier for actors** is a vector, this parameter is a cell array of vectors with cells in one-to-one correspondence to the actors in **Unique identifier for actors**. When **Unique identifier for actors** is empty, `[]`, you must specify this parameter as a cell array of one element containing the offset vector whose values apply to all actors. Units are in meters.

Example: [-1.35, .2, .3]

Dependencies

To enable this parameter, set the **Select method to specify actor profiles** parameter to `Parameters`.

Camera Intrinsics

`Focal length (pixels)` — Camera focal length
`[800,800]` (default) | two-element real-valued vector

Camera focal length, in pixels, specified as a two-element real-valued vector. See also the `FocalLength` property of `cameraIntrinsics`.

Example: `[480,320]`

`Optical center of the camera (pixels)` — Optical center of camera
`[320,240]` (default) | two-element real-valued vector

Optical center of the camera, in pixels, specified as a two-element real-valued vector. See also the `PrincipalPoint` property of `cameraIntrinsics`.

Example: `[480,320]`

`Image size produced by the camera (pixels)` — Image size produced by camera
`[480,640]` (default) | two-element vector of positive integers

Image size produced by the camera, in pixels, specified as a two-element vector of positive integers. See also the `ImageSize` property of `cameraIntrinsics`.

Example: `[240,320]`

`Radial distortion coefficients` — Radial distortion coefficients
`[0,0]` (default) | two-element real-valued vector | three-element real-valued vector

Radial distortion coefficients, specified as a two-element or three-element real-valued vector. For details on setting these coefficients, see the `RadialDistortion` property of `cameraIntrinsics`.

Example: `[1,1]`

`Tangential distortion coefficients` — Tangential distortion coefficients
`[0,0]` (default) | two-element real-valued vector

Tangential distortion coefficients, specified as a two-element real-valued vector. For details on setting these coefficients, see the `TangentialDistortion` property of `cameraIntrinsics`.

Example: `[1,1]`

`Skew of the camera axes` — Skew angle of camera axes
`0` (default) | real scalar

Skew angle of the camera axes, specified as a real scalar. See also the `Skew` property of `cameraIntrinsics`.

Example: `0.1`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Apps
Bird's-Eye Scope

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

Objects
`cameraIntrinsics` | `visionDetectionGenerator`

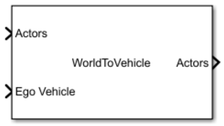
Topics
"Create Nonvirtual Buses" (Simulink)

Introduced in R2017b

World To Vehicle

Convert actors from world coordinates to ego vehicle coordinates

Library: Automated Driving Toolbox / Driving Scenario and Sensor Modeling



Description

The World To Vehicle block converts actor poses from world coordinates to the vehicle coordinates of the input ego vehicle.

Ports

Input

`Actors` — Actor poses in world coordinates

Simulink bus containing MATLAB structure

Actor poses in world coordinates, specified as a Simulink bus containing a MATLAB structure.

The structure must contain these fields.

Field	Description	Type
`NumActors`	Number of actors	Nonnegative integer
`Time`	Current simulation time	Real-valued scalar
`Actors`	Actor poses	`NumActors`-length array of actor pose structures

Each actor pose structure in `Actors` must contain these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
`Velocity`	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.

Field	Description
Yaw	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
AngularVelocity	Angular velocity (ω) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

Ego Vehicle — Ego vehicle pose
Simulink bus containing MATLAB structure

Ego vehicle pose, specified as a Simulink bus containing a MATLAB structure.

The structure must contain these fields.

Field	Description
ActorID	Scenario-defined actor identifier, specified as a positive integer.
Position	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
Velocity	Velocity (v) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
Roll	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
Pitch	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
Yaw	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
AngularVelocity	Angular velocity (ω) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

Output

Actors — Actor poses in vehicle coordinates
Simulink bus containing MATLAB structure

Actor poses in vehicle coordinates, returned as a Simulink bus containing a MATLAB structure.

The structure has these fields.

Field	Description	Type
NumActors	Number of actors	Nonnegative integer
Time	Current simulation time	Real-valued scalar
Actors	Actor poses	NumActors-length array of actor pose structures

Each actor pose structure in Actors has these fields.

Field	Description
ActorID	Scenario-defined actor identifier, specified as a positive integer.
Position	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
Velocity	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
Roll	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
Pitch	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
Yaw	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
AngularVelocity	Angular velocity (*ω*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

Parameters

Source of actors bus name — Source of name for actor poses bus
Auto (default) | Property

Source of the name for the actor poses bus returned in the **Actors** output port, specified as one of these options:

- Auto — The block automatically creates an actor poses bus name.
- Property — Specify the actor poses bus name by using the **Actors bus name** parameter.

Actors bus name — Name of actor poses bus
valid bus name

Name of the actor poses bus returned in the **Actors** output port, specified as a valid bus name.

Dependencies

To enable this parameter, set **Source of actors bus name** to Property.

Simulate using — Type of simulation to run
Interpreted execution (default) | Code generation

- Interpreted execution — Simulate the model using the MATLAB interpreter. This option shortens startup time. In Interpreted execution mode, you can debug the source code of the block.
- Code generation — Simulate the model using generated C/C++ code. The first time you run a simulation, Simulink generates C/C++ code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Cuboid To 3D Simulation | Scenario Reader | Vehicle To World

Topics
“Coordinate Systems in Automated Driving Toolbox”
“Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox”

Introduced in R2020a

Functions

addCustomBasemap

Add custom basemap

Syntax

```
addCustomBasemap(basemapName,URL)
addCustomBasemap( ___ ,Name,Value)
```

Description

`addCustomBasemap(basemapName,URL)` adds the custom basemap specified by `URL` to the list of basemaps available for use with mapping functions. `basemapName` is the name you choose to call the custom basemap. Added basemaps remain available for use in future MATLAB sessions.

You can use the custom basemap with the `geoplayer` object and with MATLAB geographic axes and charts.

`addCustomBasemap( ___ ,Name,Value)` specifies name-value pairs that set additional parameters of the basemap.

Examples

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.

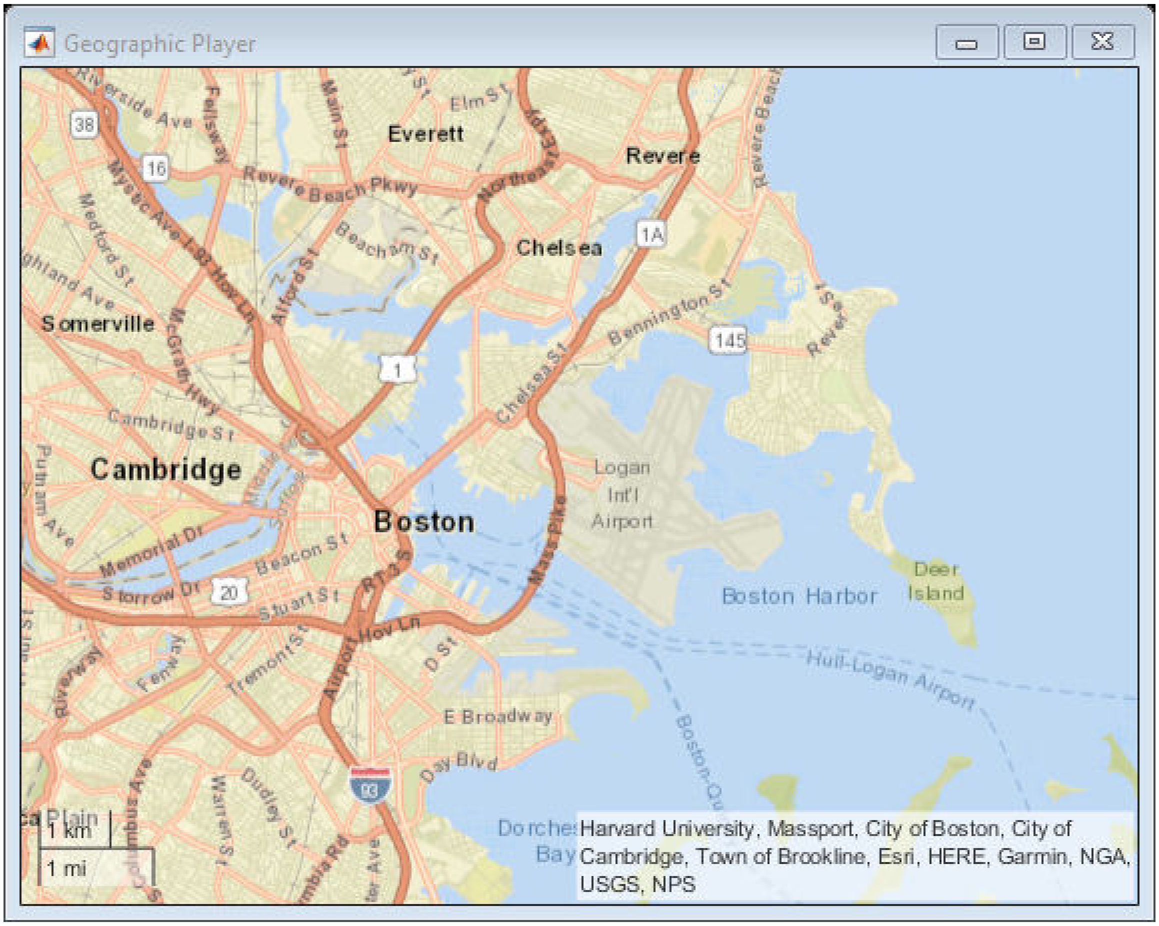
```
name = 'openstreetmap';
url = 'https://a.tile.openstreetmap.org/${z}/${x}/${y}.png';
copyright = char(uint8(169));
attribution = copyright + "OpenStreetMap contributors";
addCustomBasemap(name,url,'Attribution',attribution)
```

Load a sequence of latitude and longitude coordinates.

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

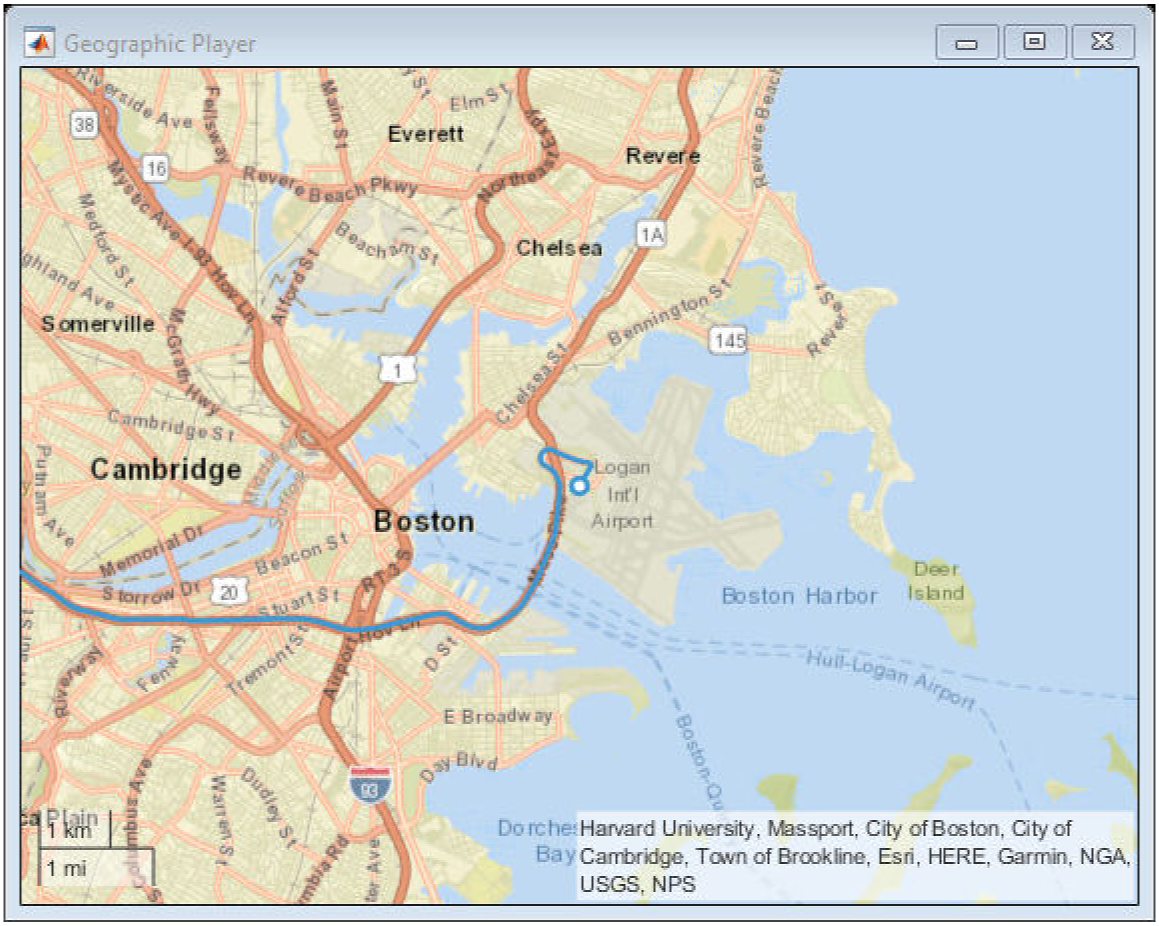
Create a geographic player. Center the geographic player on the first position of the driving route and set the zoom level to 12.

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



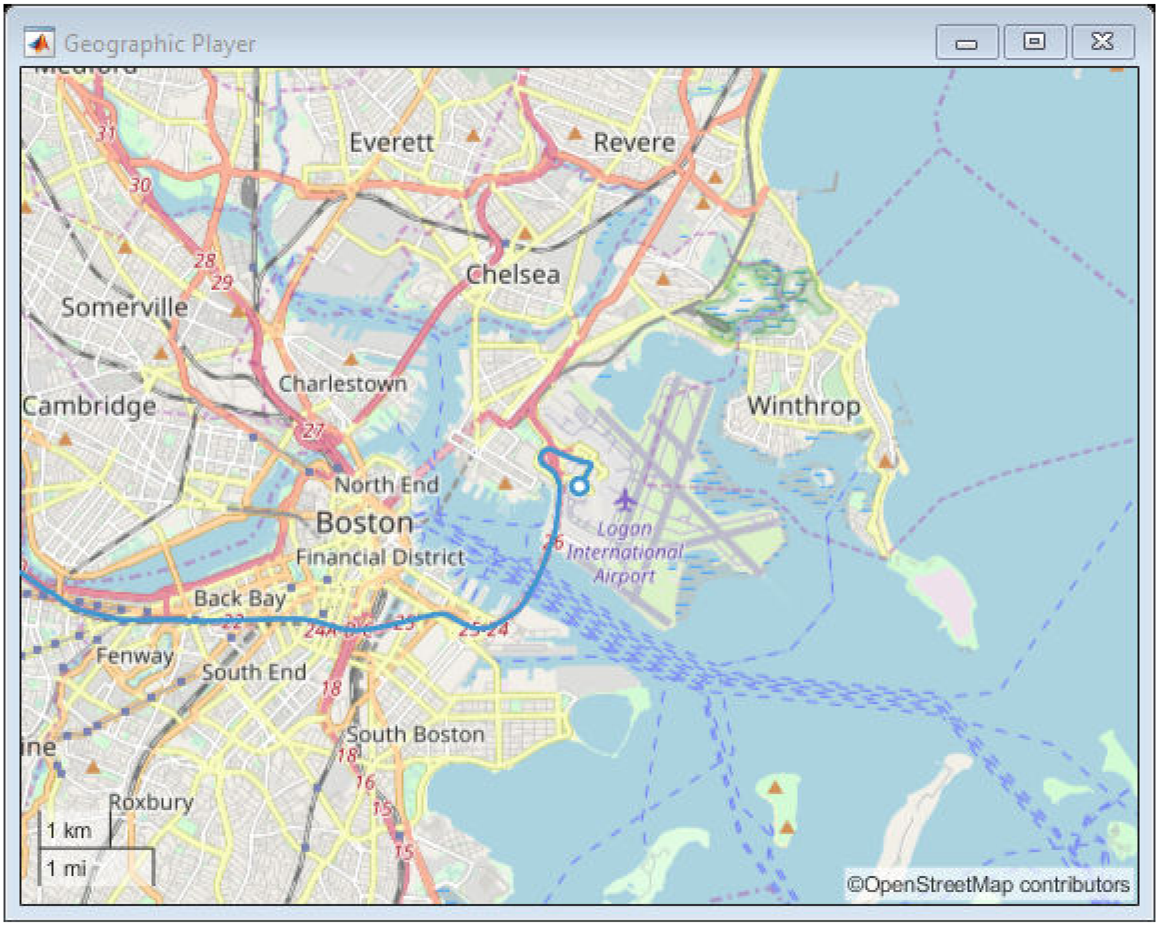
Display the full route.

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



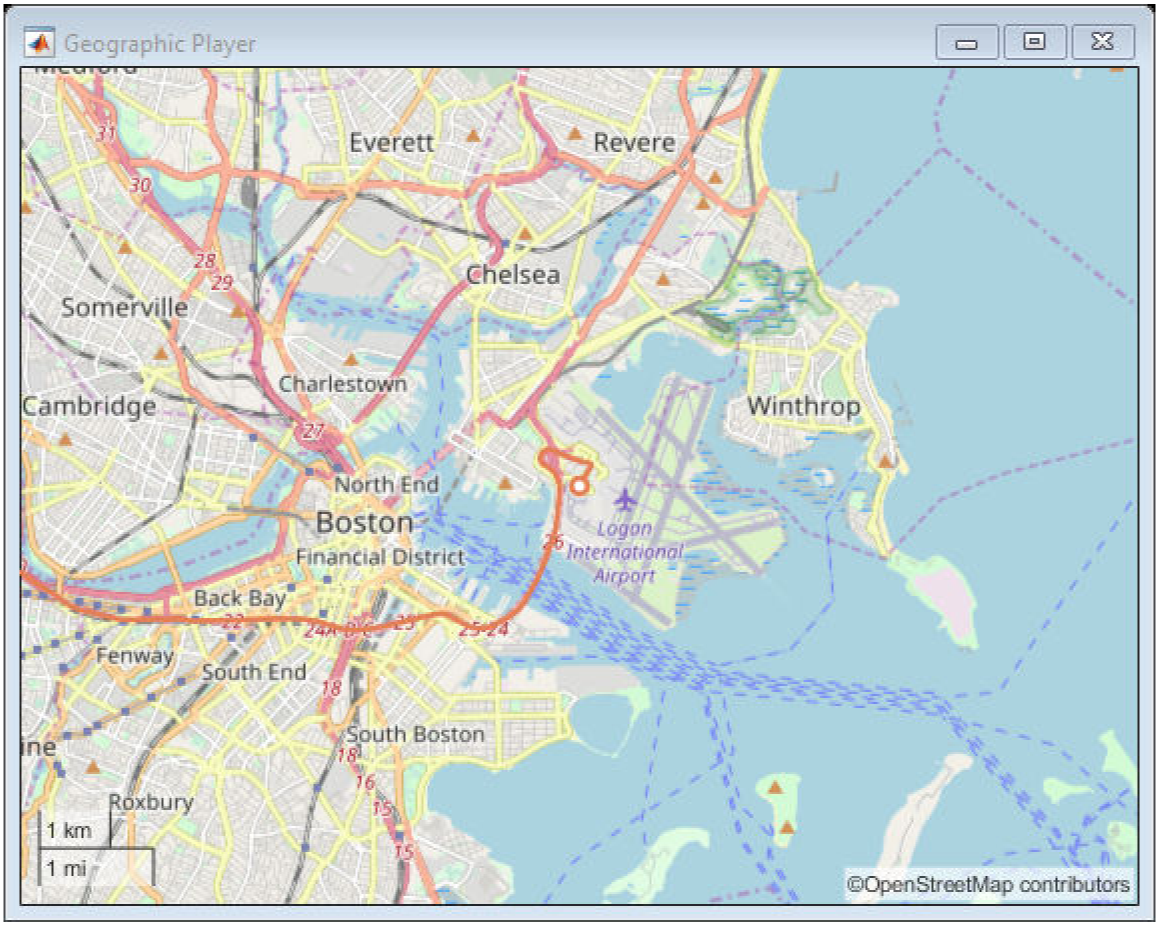
By default, the geographic player uses the World Street Map basemap (`'streets'`) provided by Esri®. Update the geographic player to use the added OpenStreetMap basemap instead.

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



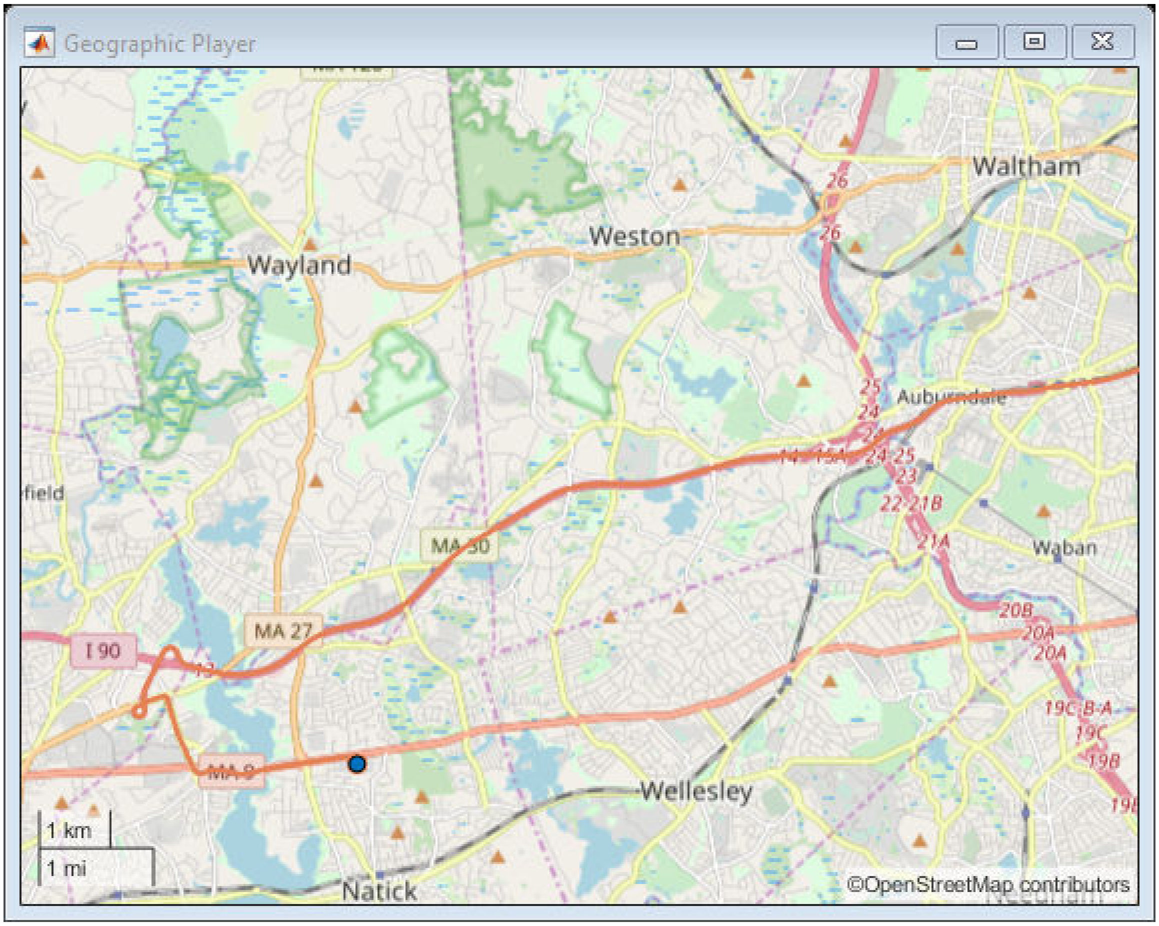
Display the route again.

```
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
```



Display Map Data on HERE Basemap

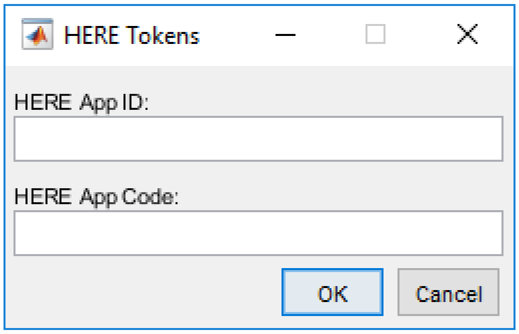
Display a driving route on a basemap provided by HERE Technologies. To use this example, you must have a valid license from HERE Technologies.

Specify the basemap name and map URL.

```
name = 'herestreets';
url  = ['https://2.base.maps.cit.api.here.com/maptile/2.1/maptile/', ...
        'newest/normal.day/${z}/${x}/${y}/256/png?app_id=%s&app_code=%s'];
```

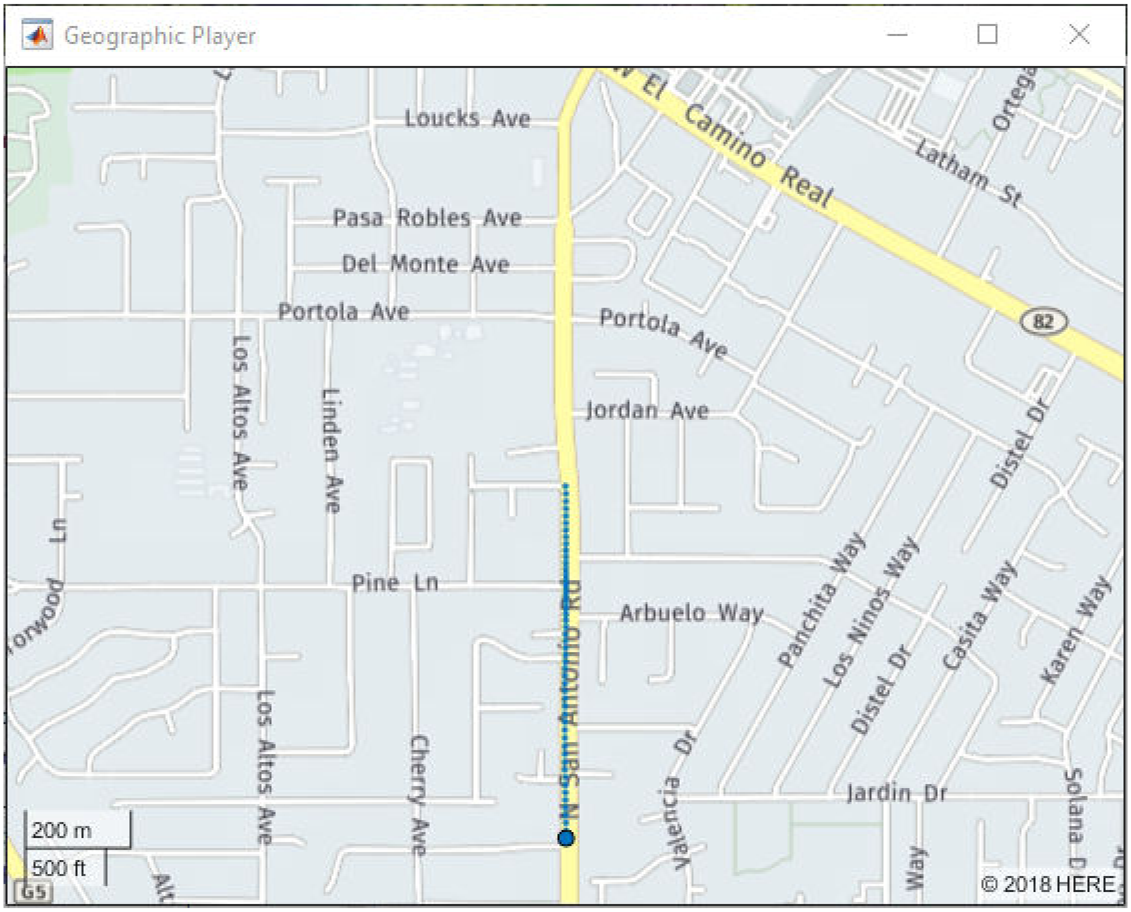
Maps from HERE Technologies require a valid license. Create a dialog box. In the dialog box, enter the App ID and App Code corresponding to your HERE license.

```
prompt = {'HERE App ID:','HERE App Code:'};
title = 'HERE Tokens';
dims = [1 40]; % Text edit field height and width
hereTokens = inputdlg(prompt,title,dims);
```



If the license is valid, specify the HERE credentials and a custom attribution, load coordinate data, and display the coordinates on the HERE basemap using a `geoplayer` object. If the license is not valid, display an error message.

```
if ~isempty(hereTokens)

    % Add HERE basemap with custom attribution.
    url = sprintf(url,hereTokens{1},hereTokens{2});
    copyrightSymbol = char(169); % Alt code
    attribution = [copyrightSymbol,' ',datestr(now,'yyyy'),' HERE'];
    addCustomBasemap(name,url,'Attribution',attribution);

    % Load sample lat,lon coordinates.
    data = load('geoSequence.mat');

    % Create geoplayer with HERE basemap.
    player = geoplayer(data.latitude(1),data.longitude(1), ...
        'Basemap','herestreets','HistoryDepth',Inf);

    % Display the coordinates in a sequence.
    for i = 1:length(data.latitude)
        plotPosition(player,data.latitude(i),data.longitude(i));
    end

else
    error('You must enter valid credentials to access maps from HERE Technologies');
end
```



Input Arguments

basemapName — Name used to identify basemap programmatically

string scalar | character vector

Name used to identify basemap programmatically, specified as a string scalar or character vector.

Example: `'openstreetmap'`

Data Types: `string` | `char`

URL — Parameterized map URL

string scalar | character vector

Parameterized map URL, specified as a string scalar or character vector. A parameterized URL is an index of the map tiles, formatted as `${z}/${x}/${y}.png` or `{z}/{x}/{y}.png`, where:

- `${z}` or `{z}` is the tile zoom level.
- `${x}` or `{x}` is the tile column index.
- `${y}` or `{y}` is the tile row index.

Example: `'https://hostname/${z}/${x}/${y}.png'`

Data Types: `string` | `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `addCustomBasemap(basemapName,URL,'Attribution',attribution)`

`Attribution` — Attribution of custom basemap
`'Tiles courtesy of` *`DOMAIN_NAME_OF_URL`*`'` (default) | string scalar | string array | character vector | cell array of character vectors

Attribution of custom basemap, specified as the comma-separated pair consisting of `'Attribution'` and a string scalar, string array, character vector, or cell array of character vectors. If the host is `'localhost'`, or if `URL` contains only IP numbers, specify an empty value (`''`). To create a multiline attribution, specify a string array or nonscalar cell array of character vectors.

If you do not specify an attribution, the default attribution is `'Tiles courtesy of` *`DOMAIN_NAME_OF_URL`*`'`, where the `addCustomBasemap` function obtains the domain name from the `URL` input argument.

Example: `'Credit: U.S. Geological Survey'`

Data Types: `string` | `char` | `cell`

`DisplayName` — Display name of custom basemap
string scalar | character vector

Display name of the custom basemap, specified as the comma-separated pair consisting of `'DisplayName'` and a string scalar or character vector.

Example: `'OpenStreetMap'`

Data Types: `string` | `char`

`MaxZoomLevel` — Maximum zoom level of basemap
`18` (default) | integer in the range [0, 25]

Maximum zoom level of the basemap, specified as the comma-separated pair consisting of `'MaxZoomLevel'` and an integer in the range [0, 25].

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`IsDeployable` — Map is deployable using MATLAB Compiler™
`false` (default) | `true`

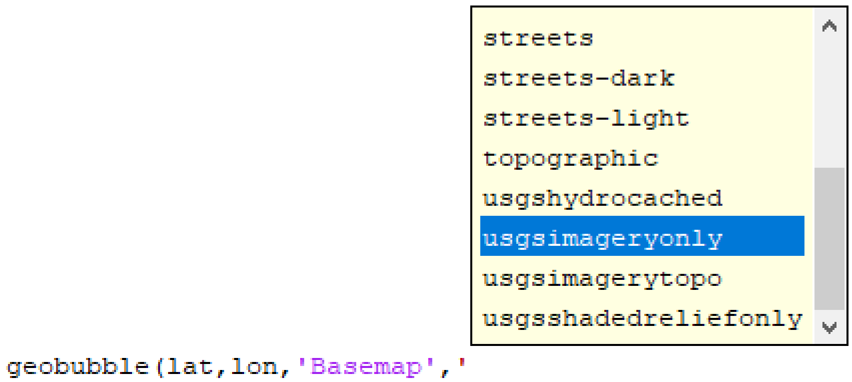
Map is deployable using MATLAB Compiler, specified as the comma-separated pair consisting of `'IsDeployable'` and `false` or `true`.

If you are deploying a map application and want users to have access to the added basemap, set `'IsDeployable'` to `true`. Maps in the `geoplayer` object are not deployable. If you are using a `geoplayer` object, leave `'IsDeployable'` set to `false`.

Data Types: `logical`

Tips

- You can find tiled web maps from various vendors, such as OpenStreetMap, the USGS National Map, Mapbox, DigitalGlobe, Esri® ArcGIS Online, the Geospatial Information Authority of Japan (GSI), and HERE Technologies. Abide by the map vendors terms-of-service agreement and include accurate attribution with the maps you use.
- To access a list of available basemaps, press **Tab** before specifying the basemap in your plotting function.



See Also

geoaxes | geobasemap | geobubble | geoplayer | removeCustomBasemap

Introduced in R2019a

cameas

Measurement function for constant-acceleration motion

Syntax

```
measurement = cameas(state)
measurement = cameas(state,frame)
measurement = cameas(state,frame,sensorpos)
measurement = cameas(state,frame,sensorpos,sensorvel)
measurement = cameas(state,frame,sensorpos,sensorvel,laxes)
measurement = cameas(state,measurementParameters)
```

Description

`measurement = cameas(state)` returns the measurement, for the constant-acceleration Kalman filter motion model in rectangular coordinates. The `state` argument specifies the current state of the filter.

`measurement = cameas(state,frame)` also specifies the measurement coordinate system, `frame`.

`measurement = cameas(state,frame,sensorpos)` also specifies the sensor position, `sensorpos`.

`measurement = cameas(state,frame,sensorpos,sensorvel)` also specifies the sensor velocity, `sensorvel`.

`measurement = cameas(state,frame,sensorpos,sensorvel,laxes)` also specifies the local sensor axes orientation, `laxes`.

`measurement = cameas(state,measurementParameters)` specifies the measurement parameters, `measurementParameters`.

Examples

Create Measurement from Accelerating Object in Rectangular Frame

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. The measurements are in rectangular coordinates.

```
state = [1,10,3,2,20,0.5].';
measurement = cameas(state)
```

```
measurement = 3×1

     1
     2
     0
```

The measurement is returned in three-dimensions with the *z*-component set to zero.

Create Measurement from Accelerating Object in Spherical Frame

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. The measurements are in spherical coordinates.

```
state = [1,10,3,2,20,5].';
measurement = cameas(state,'spherical')
```

```
measurement = 4×1

   63.4349
         0
    2.2361
   22.3607
```

The elevation of the measurement is zero and the range rate is positive. These results indicate that the object is moving away from the sensor.

Create Measurement from Accelerating Object in Translated Spherical Frame

Define the state of an object moving in 2-D constant-acceleration motion. The state consists of position, velocity, and acceleration in each dimension. The measurements are in spherical coordinates with respect to a frame located at *(20;40;0)* meters from the origin.

```
state = [1,10,3,2,20,5].';
measurement = cameas(state,'spherical',[20;40;0])
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -22.3607
```

The elevation of the measurement is zero and the range rate is negative indicating that the object is moving toward the sensor.

Create Measurement from Constant-Accelerating Object Using Measurement Parameters

Define the state of an object moving in 2-D constant-acceleration motion. The state consists of position, velocity, and acceleration in each dimension. The measurements are in spherical coordinates with respect to a frame located at *(20;40;0)* meters from the origin.

```
state2d = [1,10,3,2,20,5].';
```

The elevation of the measurement is zero and the range rate is negative indicating that the object is moving toward the sensor.

```
frame = 'spherical';
sensorpos = [20;40;0];
sensorvel = [0;5;0];
laxes = eye(3);
measurement = cameas(state2d,'spherical',sensorpos,sensorvel,laxes)
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -17.8885
```

The elevation of the measurement is zero and the range rate is negative. These results indicate that the object is moving toward the sensor.

Put the measurement parameters in a structure and use the alternative syntax.

```
measparm = struct('Frame',frame,'OriginPosition',sensorpos,'OriginVelocity',sensorvel, ...
    'Orientation',laxes);
measurement = cameas(state2d,measparm)
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -17.8885
```

Input Arguments

`state` — Kalman filter state vector

real-valued *3N*-element vector

Kalman filter state vector for constant-acceleration motion, specified as a real-valued *3N*-element vector. *N* is the number of spatial degrees of freedom of motion. For each spatial degree of motion, the state vector takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	`[x;vx;ax]`
2-D	`[x;vx;ax;y;vy;ay]`
3-D	`[x;vx;ax;y;vy;ay;z;vz;az]`

For example, `x` represents the *x*-coordinate, `vx` represents the velocity in the *x*-direction, and `ax` represents the acceleration in the *x*-direction. If the motion model is in one-dimensional space, the *y*- and *z*-axes are assumed to be zero. If the motion model is in two-dimensional space, values along the *z*-axis are assumed to be zero. Position coordinates are in meters. Velocity coordinates are in meters/second. Acceleration coordinates are in meters/second^2.

Example: `[5;0.1;0.01;0;-0.2;-0.01;-3;0.05;0]`

Data Types: `double`

`frame` — Measurement output frame
`'rectangular'` (default) | `'spherical'`

Measurement output frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of *x*, *y*, and *z* Cartesian coordinates. When specified as `'spherical'`, a measurement consists of azimuth, elevation, range, and range rate.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in meters.

Data Types: `double`

`sensorvel` — Sensor velocity
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor velocity with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in m/s.

Data Types: `double`

`laxes` — Local sensor coordinate axes
`[1,0,0;0,1,0;0,0,1]` (default) | 3-by-3 orthogonal matrix

Local sensor coordinate axes, specified as a 3-by-3 orthogonal matrix. Each column specifies the direction of the local *x*-, *y*-, and *z*-axes, respectively, with respect to the navigation frame. That is, the matrix is the rotation matrix from the global frame to the sensor frame.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure | array of structure

Measurement parameters, specified as a structure or an array of structures. The fields of the structure are:

Field	Description	Example
`Frame`	Frame used to report measurements, specified as one of these values: • `'rectangular'` — Detections are reported in rectangular coordinates. • `'spherical'` — Detections are reported in spherical coordinates.	`'spherical'`

Field	Description	Example
`OriginPosition`	Position offset of the origin of the frame relative to the parent frame, specified as an `[x y z]` real-valued vector.	`[0 0 0]`
`OriginVelocity`	Velocity offset of the origin of the frame relative to the parent frame, specified as a `[vx vy vz]` real-valued vector.	`[0 0 0]`
`Orientation`	Frame rotation matrix, specified as a 3-by-3 real-valued orthonormal matrix.	`[1 0 0; 0 1 0; 0 0 1]`
`HasAzimuth`	Logical scalar indicating if azimuth is included in the measurement.	`1`
`HasElevation`	Logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if `HasElevation` is false, the reported measurements assume 0 degrees of elevation.	`1`
`HasRange`	Logical scalar indicating if range is included in the measurement.	`1`
`HasVelocity`	Logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if `HasVelocity` is false, the measurements are reported as `[x y z]`. If `HasVelocity` is `true`, measurements are reported as `[x y z vx vy vz]`.	`1`
`IsParentToChild`	Logical scalar indicating if `Orientation` performs a frame rotation from the parent coordinate frame to the child coordinate frame. When `IsParentToChild` is `false`, then `Orientation` performs a frame rotation from the child coordinate frame to the parent coordinate frame.	`0`

Data Types: `struct`

Output Arguments

`measurement` — Measurement vector
N-by-1 column vector

Measurement vector, returned as an *N*-by-1 column vector. The form of the measurement depends upon which syntax you use.

- When the syntax does not use the `measurementParameters` argument, the measurement vector is `[x,y,z]` when the `frame` input argument is set to `'rectangular'` and `[az;el;r;rr]` when the `frame` is set to `'spherical'`.
- When the syntax uses the `measurementParameters` argument, the size of the measurement vector depends on the values of the `frame`, `HasVelocity`, and `HasElevation` fields in the `measurementParameters` structure.

<table>
<tr><th>frame</th><th>measurement</th></tr>
<tr><td>'spherical'</td><td>Specifies the azimuth angle, az, elevation angle, el, range, r, and range rate, rr, of the object with respect to the local ego vehicle coordinate system. Positive values for range rate indicate that an object is moving away from the sensor.

Spherical measurements
<table>
<tr><th></th><th></th><th>HasElevation</th><th></th></tr>
<tr><td></td><td></td><td>false</td><td>true</td></tr>
<tr><td>HasVelocity</td><td>false</td><td>[az;r]</td><td>[az;el;r]</td></tr>
<tr><td></td><td>true</td><td>[az;r;rr]</td><td>[az;el;r;rr]</td></tr>
</table>

Angle units are in degrees, range units are in meters, and range rate units are in m/s.</td></tr>
<tr><td>'rectangular</td><td>Specifies the Cartesian position and velocity coordinates of the tracked object with respect to the ego vehicle coordinate system.

Rectangular measurements
<table>
<tr><td>HasVelocity</td><td>false</td><td>[x;y;y]</td></tr>
<tr><td></td><td>true</td><td>[x;y;z;vx;vy;vz]</td></tr>
</table>

Position units are in meters and velocity units are in m/s.</td></tr>
</table>

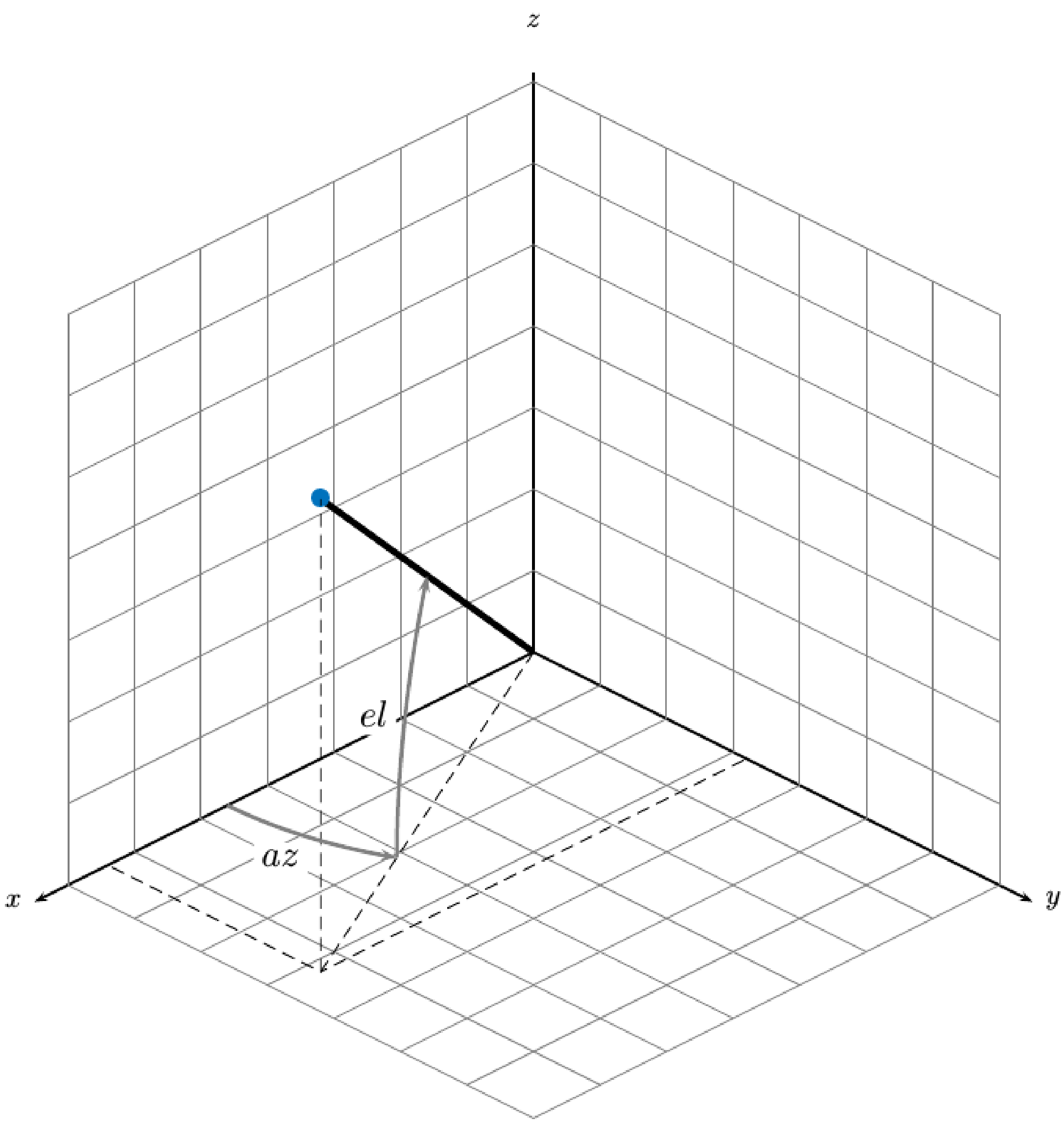
Data Types: `double`

More About

Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Automated Driving Toolbox.

The azimuth angle of a vector is the angle between the *x*-axis and its orthogonal projection onto the *xy* plane. The angle is positive in going from the *x* axis toward the *y* axis. Azimuth angles lie between –180 and 180 degrees. The elevation angle is the angle between the vector and its orthogonal projection onto the *xy*-plane. The angle is positive when going toward the positive *z*-axis from the *xy* plane.



Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`cameasjac` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `constveljac` | `ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac`

Objects
`trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

cameasjac

Jacobian of measurement function for constant-acceleration motion

Syntax

```
measurementjac = cameasjac(state)
measurementjac = cameasjac(state,frame)
measurementjac = cameasjac(state,frame,sensorpos)
measurementjac = cameasjac(state,frame,sensorpos,sensorvel)
measurementjac = cameasjac(state,frame,sensorpos,sensorvel,laxes)
measurementjac = cameasjac(state,measurementParameters)
```

Description

`measurementjac = cameasjac(state)` returns the measurement Jacobian, for constant-acceleration Kalman filter motion model in rectangular coordinates. The `state` argument specifies the current state of the filter.

`measurementjac = cameasjac(state,frame)` also specifies the measurement coordinate system, `frame`.

`measurementjac = cameasjac(state,frame,sensorpos)` also specifies the sensor position, `sensorpos`.

`measurementjac = cameasjac(state,frame,sensorpos,sensorvel)` also specifies the sensor velocity, `sensorvel`.

`measurementjac = cameasjac(state,frame,sensorpos,sensorvel,laxes)` also specifies the local sensor axes orientation, `laxes`.

`measurementjac = cameasjac(state,measurementParameters)` specifies the measurement parameters, `measurementParameters`.

Examples

Measurement Jacobian of Accelerating Object in Rectangular Frame

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. Construct the measurement Jacobian in rectangular coordinates.

```
state = [1,10,3,2,20,5].';
jacobian = cameasjac(state)
```

```
jacobian = 3×6

     1     0     0     0     0     0
     0     0     0     1     0     0
     0     0     0     0     0     0
```

Measurement Jacobian of Accelerating Object in Spherical Frame

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. Compute the measurement Jacobian in spherical coordinates.

```
state = [1;10;3;2;20;5];
measurementjac = cameasjac(state,'spherical')
```

```
measurementjac = 4×6

  -22.9183         0         0   11.4592         0         0
         0         0         0         0         0         0
    0.4472         0         0    0.8944         0         0
    0.0000    0.4472         0    0.0000    0.8944         0
```

Measurement Jacobian of Accelerating Object in Translated Spherical Frame

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. Compute the measurement Jacobian in spherical coordinates with respect to an origin at *(5;-20;0)* meters.

```
state = [1,10,3,2,20,5].';
sensorpos = [5,-20,0].';
measurementjac = cameasjac(state,'spherical',sensorpos)
```

```
measurementjac = 4×6

   -2.5210         0         0   -0.4584         0         0
         0         0         0         0         0         0
   -0.1789         0         0    0.9839         0         0
    0.5903   -0.1789         0    0.1073    0.9839         0
```

Create Measurement Jacobian of Accelerating Object Using Measurement Parameters

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. Compute the measurement Jacobian in spherical coordinates with respect to an origin at *(5;-20;0)* meters.

```
state2d = [1,10,3,2,20,5].';
sensorpos = [5,-20,0].';
frame = 'spherical';
sensorvel = [0;8;0];
laxes = eye(3);
measurementjac = cameasjac(state2d,frame,sensorpos,sensorvel,laxes)
```

```
measurementjac = 4×6

   -2.5210         0         0   -0.4584         0         0
```

```
         0         0         0         0         0         0
   -0.1789         0         0    0.9839         0         0
    0.5274   -0.1789         0    0.0959    0.9839         0
```

Put the measurement parameters in a structure and use the alternative syntax.

```
measparm = struct('Frame',frame,'OriginPosition',sensorpos,'OriginVelocity',sensorvel, ...
    'Orientation',laxes);
measurementjac = cameasjac(state2d,measparm)
```

```
measurementjac = 4×6

   -2.5210         0         0   -0.4584         0         0
         0         0         0         0         0         0
   -0.1789         0         0    0.9839         0         0
    0.5274   -0.1789         0    0.0959    0.9839         0
```

Input Arguments

`state` — Kalman filter state vector

real-valued *3N*-element vector

Kalman filter state vector for constant-acceleration motion, specified as a real-valued *3N*-element vector. *N* is the number of spatial degrees of freedom of motion. For each spatial degree of motion, the state vector takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	`[x;vx;ax]`
2-D	`[x;vx;ax;y;vy;ay]`
3-D	`[x;vx;ax;y;vy;ay;z;vz;az]`

For example, `x` represents the *x*-coordinate, `vx` represents the velocity in the *x*-direction, and `ax` represents the acceleration in the *x*-direction. If the motion model is in one-dimensional space, the *y*- and *z*-axes are assumed to be zero. If the motion model is in two-dimensional space, values along the *z*-axis are assumed to be zero. Position coordinates are in meters. Velocity coordinates are in meters/second. Acceleration coordinates are in meters/second^2.

Example: `[5;0.1;0.01;0;-0.2;-0.01;-3;0.05;0]`

Data Types: `double`

`frame` — Measurement output frame

`'rectangular'` (default) | `'spherical'`

Measurement output frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of *x*, *y*, and *z* Cartesian coordinates. When specified as `'spherical'`, a measurement consists of azimuth, elevation, range, and range rate.

Data Types: `char`

`sensorpos` — Sensor position

`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in meters.

Data Types: `double`

`sensorvel` — Sensor velocity
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor velocity with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in m/s.

Data Types: `double`

`laxes` — Local sensor coordinate axes
`[1,0,0;0,1,0;0,0,1]` (default) | 3-by-3 orthogonal matrix

Local sensor coordinate axes, specified as a 3-by-3 orthogonal matrix. Each column specifies the direction of the local *x*-, *y*-, and *z*-axes, respectively, with respect to the navigation frame. That is, the matrix is the rotation matrix from the global frame to the sensor frame.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure | array of structure

Measurement parameters, specified as a structure or an array of structures. The fields of the structure are:

Field	Description	Example
`Frame`	Frame used to report measurements, specified as one of these values: • `'rectangular'` — Detections are reported in rectangular coordinates. • `'spherical'` — Detections are reported in spherical coordinates.	`'spherical'`
`OriginPosition`	Position offset of the origin of the frame relative to the parent frame, specified as an `[x y z]` real-valued vector.	`[0 0 0]`
`OriginVelocity`	Velocity offset of the origin of the frame relative to the parent frame, specified as a `[vx vy vz]` real-valued vector.	`[0 0 0]`
`Orientation`	Frame rotation matrix, specified as a 3-by-3 real-valued orthonormal matrix.	`[1 0 0; 0 1 0; 0 0 1]`

Field	Description	Example
`HasAzimuth`	Logical scalar indicating if azimuth is included in the measurement.	`1`
`HasElevation`	Logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if `HasElevation` is false, the reported measurements assume 0 degrees of elevation.	`1`
`HasRange`	Logical scalar indicating if range is included in the measurement.	`1`
`HasVelocity`	Logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if `HasVelocity` is false, the measurements are reported as `[x y z]`. If `HasVelocity` is `true`, measurements are reported as `[x y z vx vy vz]`.	`1`
`IsParentToChild`	Logical scalar indicating if `Orientation` performs a frame rotation from the parent coordinate frame to the child coordinate frame. When `IsParentToChild` is `false`, then `Orientation` performs a frame rotation from the child coordinate frame to the parent coordinate frame.	`0`

Data Types: `struct`

Output Arguments

`measurementjac` — Measurement Jacobian
real-valued 3-by-*N* matrix | real-valued 4-by-*N* matrix

Measurement Jacobian, specified as a real-valued 3-by-*N* or 4-by-*N* matrix. *N* is the dimension of the state vector. The interpretation of the rows and columns depends on the `frame` argument, as described in this table.

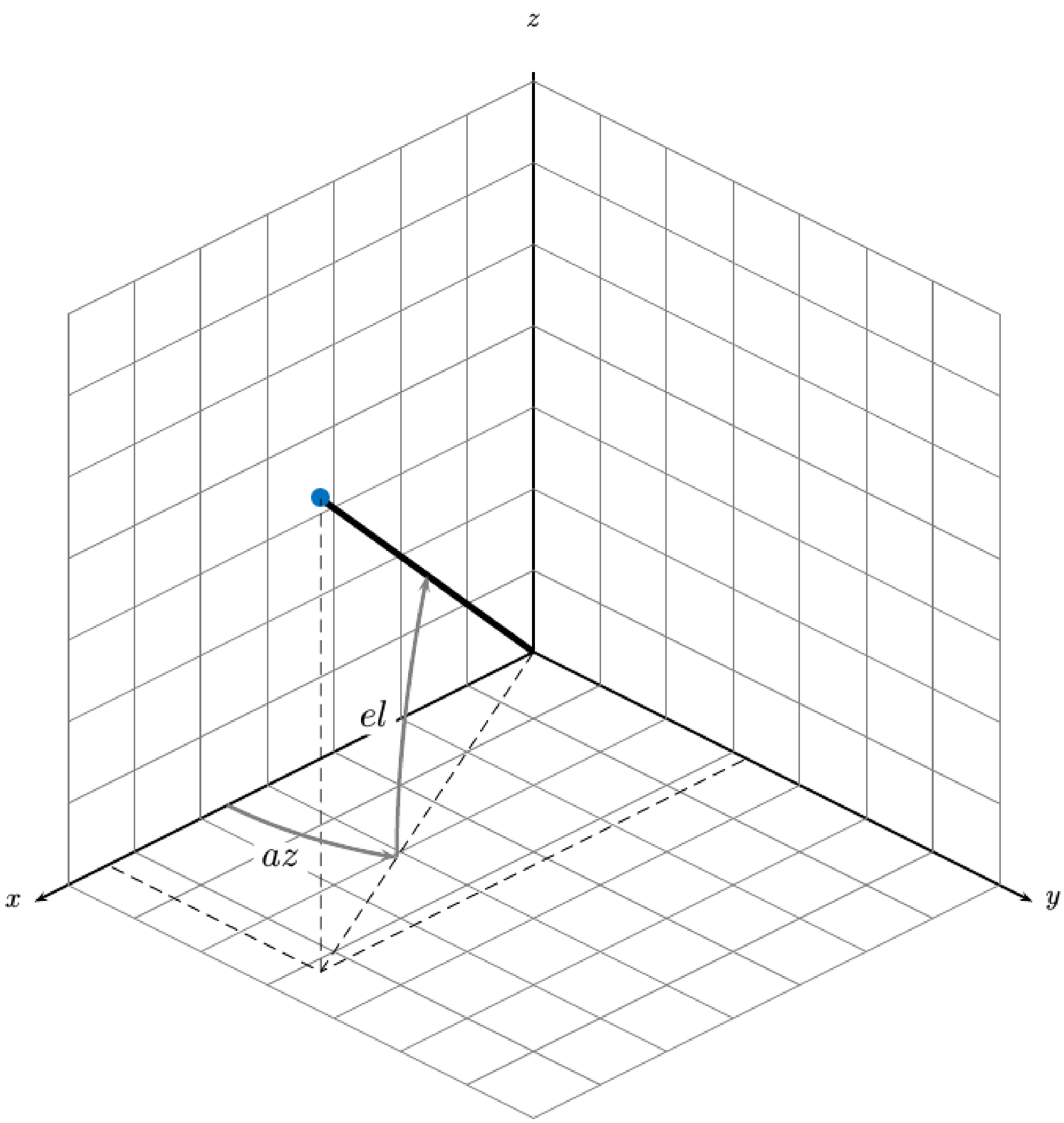
Frame	Measurement Jacobian
'rectangular'	Jacobian of the measurements [x;y;z] with respect to the state vector. The measurement vector is with respect to the local coordinate system. Coordinates are in meters.
'spherical'	Jacobian of the measurement vector [az;el;r;rr] with respect to the state vector. Measurement vector components specify the azimuth angle, elevation angle, range, and range rate of the object with respect to the local sensor coordinate system. Angle units are in degrees. Range units are in meters and range rate units are in meters/second.

More About

Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Automated Driving Toolbox.

The azimuth angle of a vector is the angle between the *x*-axis and its orthogonal projection onto the *xy* plane. The angle is positive in going from the *x* axis toward the *y* axis. Azimuth angles lie between –180 and 180 degrees. The elevation angle is the angle between the vector and its orthogonal projection onto the *xy*-plane. The angle is positive when going toward the positive *z*-axis from the *xy* plane.



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`cameas` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `constveljac` | `ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac`

Objects

`trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

checkPathValidity

Check validity of planned vehicle path

Syntax

```
isValid = checkPathValidity(refPath,costmap)
isValid = checkPathValidity(refPoses,costmap)
```

Description

`isValid = checkPathValidity(refPath,costmap)` checks the validity of a planned vehicle path, `refPath`, against the vehicle costmap. Use this function to test if a path is valid within a changing environment.

A path is valid if the following conditions are true:

- The path has at least one pose.
- The path is collision-free and within the limits of `costmap`.

`isValid = checkPathValidity(refPoses,costmap)` checks the validity of a sequence of vehicle poses, `refPoses`, against the vehicle costmap.

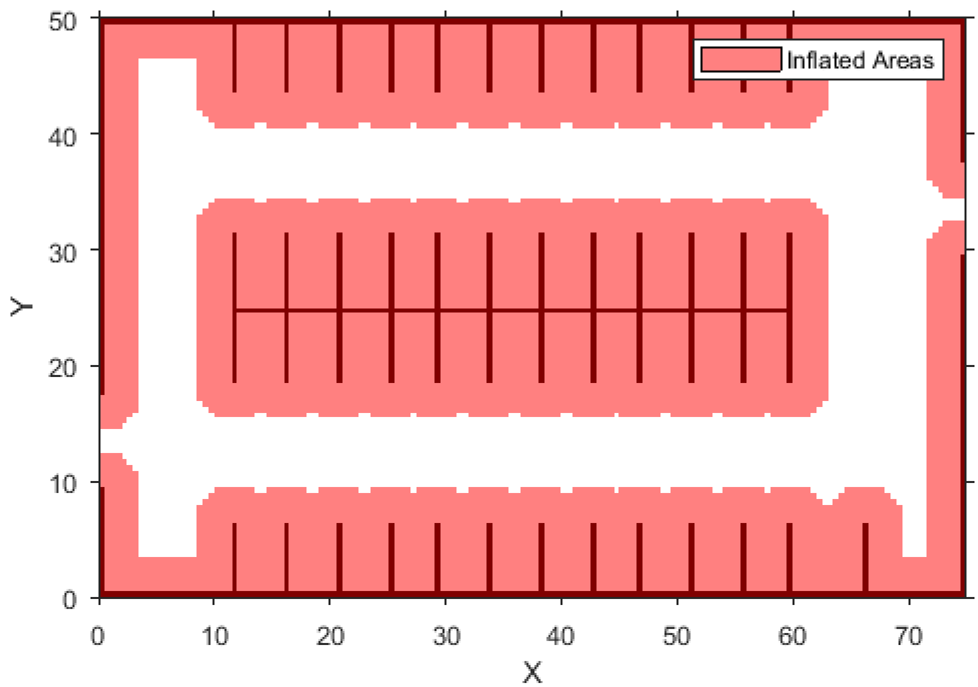
Examples

Plan Path and Check Its Validity

Plan a vehicle path through a parking lot by using the optimal rapidly exploring random tree (RRT*) algorithm. Check that the path is valid, and then plot the transition poses along the path.

Load a costmap of a parking lot. Plot the costmap to see the parking lot and inflated areas for the vehicle to avoid.

```
data = load('parkingLotCostmap.mat');
costmap = data.parkingLotCostmap;
plot(costmap)
```



Define start and goal poses for the vehicle as [*x*, *y*, *Θ*] vectors. World units for the (*x*,*y*) locations are in meters. World units for the *Θ* orientation angles are in degrees.

```
startPose = [4, 4, 90]; % [meters, meters, degrees]
goalPose = [30, 13, 0];
```

Use a `pathPlannerRRT` object to plan a path from the start pose to the goal pose.

```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Check that the path is valid.

```
isPathValid = checkPathValidity(refPath,costmap)
```

```
isPathValid = logical
   1
```

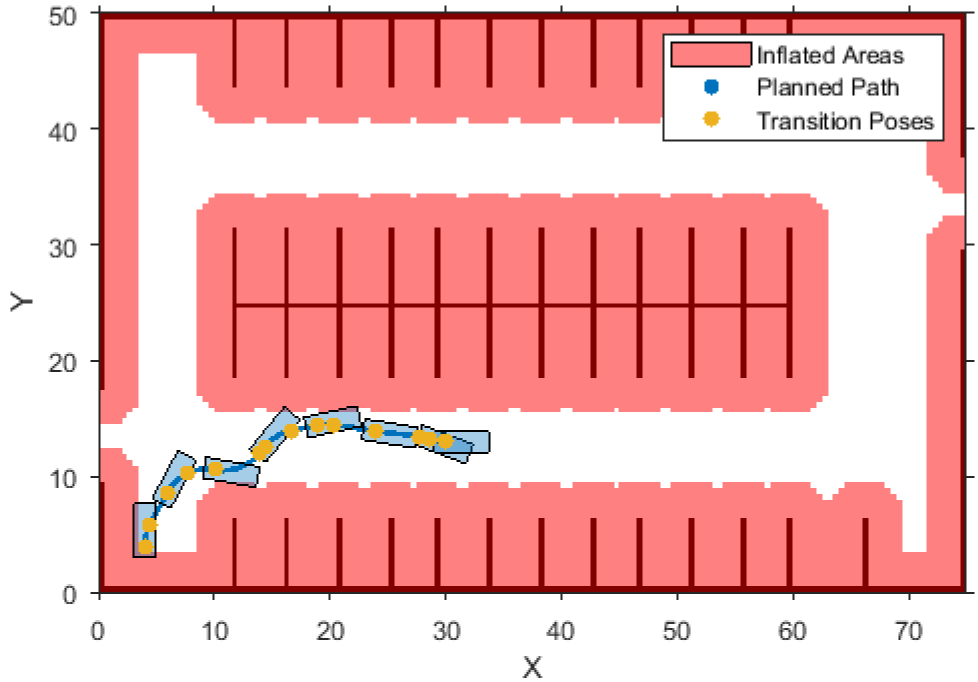
Interpolate the transition poses along the path.

```
transitionPoses = interpolate(refPath);
```

Plot the planned path and the transition poses on the costmap.

```
hold on
plot(refPath,'DisplayName','Planned Path')
scatter(transitionPoses(:,1),transitionPoses(:,2),[],'filled', ...
```

```
    'DisplayName','Transition Poses')
hold off
```



Input Arguments

`refPath` — Planned vehicle path
`driving.Path` object

Planned vehicle path, specified as a `driving.Path` object.

`costmap` — Costmap used for collision checking
`vehicleCostmap` object

Costmap used for collision checking, specified as a `vehicleCostmap` object.

`refPoses` — Sequence of vehicle poses
m-by-3 matrix of [*x*, *y*, *Θ*] vectors

Sequence of vehicle poses, specified as an *m*-by-3 matrix of [*x*, *y*, *Θ*] vectors. *m* is the number of specified poses.

x and *y* specify the location of the vehicle. These values must be in the same world units used by `costmap`.

Θ specifies the orientation angle of the vehicle in degrees.

Output Arguments

`isValid` — Indicates validity of path or poses
1 | 0

Indicates validity of the planned vehicle path, `refPath`, or the sequence of vehicle poses, `refPoses`, returned as a logical value of `1` or `0`.

A path or sequence of poses is valid (`1`) if the following conditions are true:

- The path or pose sequence has at least one pose.
- The path or pose sequence is collision-free and within the limits of `costmap`.

Algorithms

To check if a vehicle path is valid, the `checkPathValidity` function discretizes the path. Then, the function checks that the poses at the discretized points are collision-free. The threshold for a collision-free pose depends on the resolution at which `checkPathValidity` discretizes.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`plan` | `plot`

Objects
`driving.Path` | `pathPlannerRRT` | `vehicleCostmap`

Topics
"Automated Parking Valet"

Introduced in R2018a

configureDetectorMonoCamera

Configure object detector for using calibrated monocular camera

Syntax

`configuredDetector = configureDetectorMonoCamera(detector,sensor,objectSize)`

Description

`configuredDetector = configureDetectorMonoCamera(detector,sensor,objectSize)` configures any of these object detectors

- ACF (aggregate channel features)
- Faster R-CNN (regions with convolutional neural networks)
- Fast R-CNN
- YOLO v2 (you only look once v2)
- SSD (single shot detector),

to detect objects of a known size on a ground plane. Specify your trained object detector, `detector`, a camera configuration for transforming image coordinates to world coordinates, `sensor`, and the range of the object widths and lengths, `objectSize`.

Examples

Detect Vehicles Using Monocular Camera and ACF

Configure an ACF object detector for use with a monocular camera mounted on an ego vehicle. Use this detector to detect vehicles within video frames captured by the camera.

Load an `acfObjectDetector` object pretrained to detect vehicles.

```
detector = vehicleDetectorACF;
```

Model a monocular camera sensor by creating a `monoCamera` object. This object contains the camera intrinsics and the location of the camera on the ego vehicle.

```
focalLength = [309.4362 344.2161];    % [fx fy]
principalPoint = [318.9034 257.5352]; % [cx cy]
imageSize = [480 640];                % [mrows ncols]
height = 2.1798;                      % height of camera above ground, in meters
pitch = 14;                           % pitch of camera, in degrees
intrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);

monCam = monoCamera(intrinsics,height,'Pitch',pitch);
```

Configure the detector for use with the camera. Limit the width of detected objects to a typical range for vehicle widths: 1.5–2.5 meters. The configured detector is an `acfObjectDetectorMonoCamera` object.

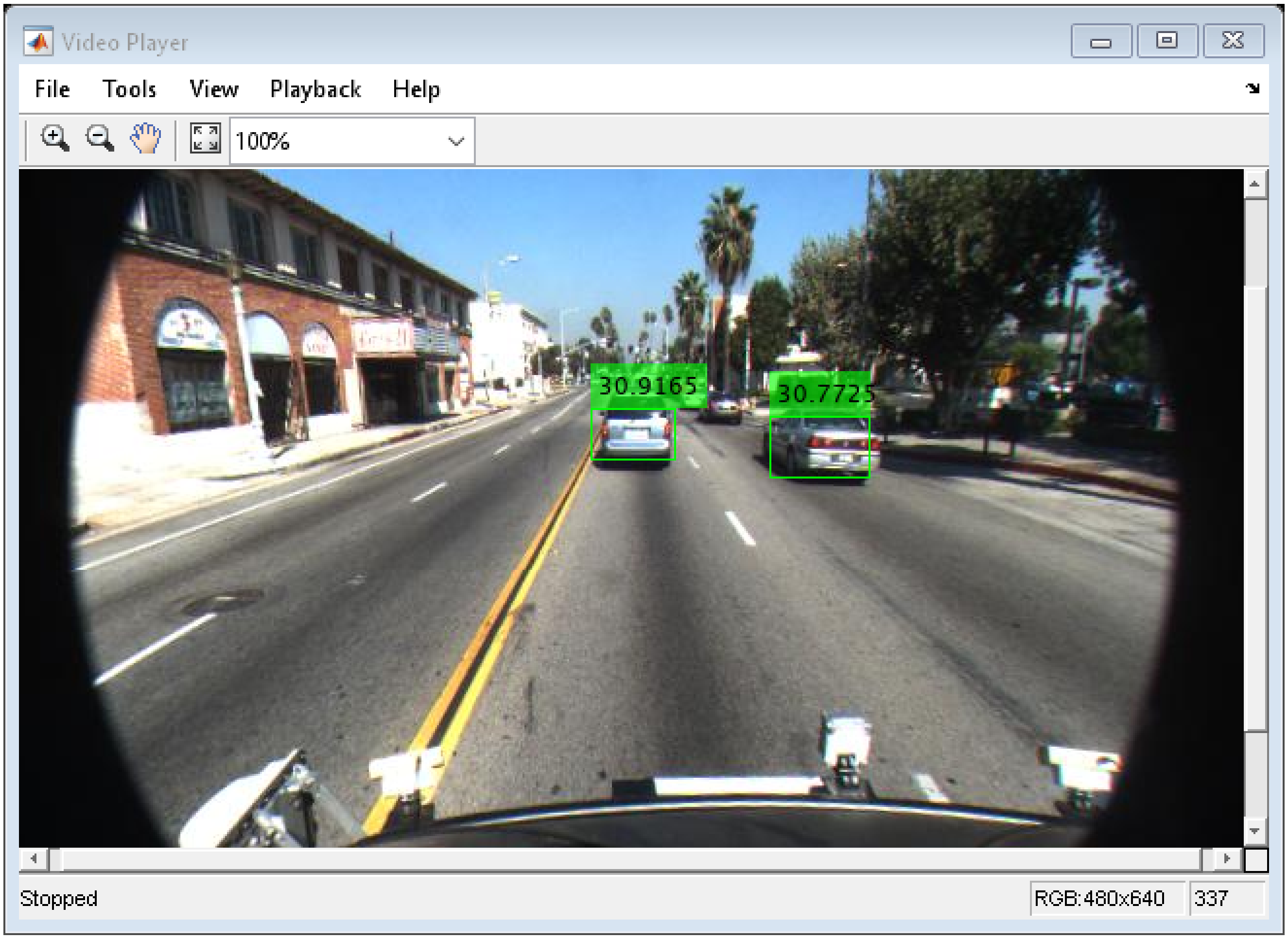
```
vehicleWidth = [1.5 2.5];
detectorMonoCam = configureDetectorMonoCamera(detector,monCam,vehicleWidth);
```

Load a video captured from the camera, and create a video reader and player.

```
videoFile = fullfile(toolboxdir('driving'),'drivingdata','caltech_washington1.avi');
reader = VideoReader(videoFile);
videoPlayer = vision.VideoPlayer('Position',[29 597 643 386]);
```

Run the detector in a loop over the video. Annotate the video with the bounding boxes for the detections and the detection confidence scores.

```
cont = hasFrame(reader);
while cont
   I = readFrame(reader);

   % Run the detector.
   [bboxes,scores] = detect(detectorMonoCam,I);
   if ~isempty(bboxes)
       I = insertObjectAnnotation(I, ...
                           'rectangle',bboxes, ...
                           scores, ...
                           'Color','g');
   end
   videoPlayer(I)
   % Exit the loop if the video player figure is closed.
   cont = hasFrame(reader) && isOpen(videoPlayer);
end

release(videoPlayer);
```



Input Arguments

detector — Object detector to configure
`acfObjectDetector` object | `fastRCNNObjectDetector` object | `fasterRCNNObjectDetector` object | `yolov2ObjectDetector` object | `ssdObjectDetector` object

Object detector to configure, specified as one of these object detector objects:

- `acfObjectDetector`
- `fastRCNNObjectDetector`
- `fasterRCNNObjectDetector`
- `yolov2ObjectDetector`
- `ssdObjectDetector`

Train the object detector before configuring them by using:

- `trainACFObjectDetector`
- `trainFastRCNNObjectDetector`
- `trainFasterRCNNObjectDetector`

- `trainYOLOv2ObjectDetector`
- `trainSSDObjectDetector`

`sensor` — Camera configuration
`monoCamera` object

Camera configuration, specified as a `monoCamera` object. The object contains the camera intrinsics, the location, the pitch, yaw, and roll placement, and the world units for the parameters. Use the intrinsics to transform the object points in the image to world coordinates, which you can then compare to the `WorldObjectSize` property for `detector`.

`objectSize` — Range of object widths and lengths
[*minWidth* *maxWidth*] vector | [*minWidth* *maxWidth*; *minLength* *maxLength*] vector

Range of object widths and lengths in world units, specified as a [*minWidth* *maxWidth*] vector or [*minWidth* *maxWidth*; *minLength* *maxLength*] vector. Specifying the range of object lengths is optional.

Output Arguments

`configuredDetector` — Configured object detector
`acfObjectDetectorMonoCamera` object | `fastRCNNObjectDetectorMonoCamera` object | `fasterRCNNObjectDetectorMonoCamera` object | `yolov2ObjectDetectorMonoCamera` | `ssdObjectDetectorMonoCamera`

Configured object detector, returned as one of these object detector objects:

- `acfObjectDetectorMonoCamera`
- `fastRCNNObjectDetectorMonoCamera`
- `fasterRCNNObjectDetectorMonoCamera`
- `yolov2ObjectDetectorMonoCamera`
- `ssdObjectDetectorMonoCamera`

See Also

`acfObjectDetector` | `acfObjectDetectorMonoCamera` | `fastRCNNObjectDetector` | `fastRCNNObjectDetectorMonoCamera` | `fasterRCNNObjectDetector` | `fasterRCNNObjectDetectorMonoCamera` | `monoCamera` | `ssdObjectDetectorMonoCamera` | `yolov2ObjectDetectorMonoCamera`

Introduced in R2017a

constacc

Constant-acceleration motion model

Syntax

```
updatedstate = constacc(state)
updatedstate = constacc(state,dt)
updatedstate = constacc(state,w,dt)
```

Description

`updatedstate = constacc(state)` returns the updated state, `state`, of a constant acceleration Kalman filter motion model for a step time of one second.

`updatedstate = constacc(state,dt)` specifies the time step, `dt`.

`updatedstate = constacc(state,w,dt)` also specifies the state noise, `w`.

Examples

Predict State for Constant-Acceleration Motion

Define an initial state for 2-D constant-acceleration motion.

```
state = [1;1;1;2;1;0];
```

Predict the state 1 second later.

```
state = constacc(state)
```

```
state = 6×1

    2.5000
    2.0000
    1.0000
    3.0000
    1.0000
         0
```

Predict State for Constant-Acceleration Motion With Specified Time Step

Define an initial state for 2-D constant-acceleration motion.

```
state = [1;1;1;2;1;0];
```

Predict the state 0.5 s later.

```
state = constacc(state,0.5)
```

```
state = 6×1

    1.6250
    1.5000
    1.0000
    2.5000
    1.0000
         0
```

Input Arguments

`state` — Kalman filter state vector

real-valued *3N*-element vector

Kalman filter state vector for constant-acceleration motion, specified as a real-valued *3N*-element vector. *N* is the number of spatial degrees of freedom of motion. For each spatial degree of motion, the state vector takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	`[x;vx;ax]`
2-D	`[x;vx;ax;y;vy;ay]`
3-D	`[x;vx;ax;y;vy;ay;z;vz;az]`

For example, `x` represents the *x*-coordinate, `vx` represents the velocity in the *x*-direction, and `ax` represents the acceleration in the *x*-direction. If the motion model is in one-dimensional space, the *y*- and *z*-axes are assumed to be zero. If the motion model is in two-dimensional space, values along the *z*-axis are assumed to be zero. Position coordinates are in meters. Velocity coordinates are in meters/second. Acceleration coordinates are in meters/second^2.

Example: `[5;0.1;0.01;0;-0.2;-0.01;-3;0.05;0]`

Data Types: `double`

`dt` — Time step interval of filter

`1.0` (default) | positive scalar

Time step interval of filter, specified as a positive scalar. Time units are in seconds.

Example: `0.5`

Data Types: `single` | `double`

`w` — State noise

scalar | real-valued *D*-by-*N* matrix

State noise, specified as a scalar or real-valued *D*-by-*N* matrix. *D* is the number of motion dimensions and *N* is the number of state vectors. If specified as a scalar, the scalar value is expanded to a *D*-by-*N* matrix.

Data Types: `single` | `double`

Output Arguments

`updatedstate` — Updated state vector
real-valued column or row vector | real-valued matrix

Updated state vector, returned as a real-valued vector or real-valued matrix with same number of elements and dimensions as the input state vector.

Algorithms

For a two-dimensional constant-acceleration process, the state transition matrix after a time step, *T*, is block diagonal:

$$\begin{bmatrix} x_{k+1} \\ vx_{k+1} \\ ax_{k+1} \\ y_{k+1} \\ vy_{k+1} \\ ay_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & T & \frac{1}{2}T^2 & 0 & 0 & 0 \\ 0 & 1 & T & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & T & \frac{1}{2}T^2 \\ 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_k \\ vx_k \\ ax_k \\ y_k \\ vy_k \\ ay_k \end{bmatrix}$$

The block for each spatial dimension has this form:

$$\begin{bmatrix} 1 & T & \frac{1}{2}T^2 \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix}$$

For each additional spatial dimension, add an identical block.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`cameas` | `cameasjac` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `constveljac` | `ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac`

Objects
`trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

constaccjac

Jacobian for constant-acceleration motion

Syntax

```
jacobian = constaccjac(state)
jacobian = constaccjac(state,dt)
[jacobian,noisejacobian] = constaccjac(state,w,dt)
```

Description

`jacobian = constaccjac(state)` returns the updated Jacobian , `jacobian`, for a constant-acceleration Kalman filter motion model. The step time is one second. The `state` argument specifies the current state of the filter.

`jacobian = constaccjac(state,dt)` also specifies the time step, `dt`.

`[jacobian,noisejacobian] = constaccjac(state,w,dt)` specifies the state noise, `w`, and returns the Jacobian, `noisejacobian`, of the state with respect to the noise.

Examples

Compute State Jacobian for Constant-Acceleration Motion

Compute the state Jacobian for two-dimensional constant-acceleration motion.

Define an initial state and compute the state Jacobian for a one second update time.

```
state = [1,1,1,2,1,0];
jacobian = constaccjac(state)
```

```
jacobian = 6×6

    1.0000    1.0000    0.5000         0         0         0
         0    1.0000    1.0000         0         0         0
         0         0    1.0000         0         0         0
         0         0         0    1.0000    1.0000    0.5000
         0         0         0         0    1.0000    1.0000
         0         0         0         0         0    1.0000
```

Compute State Jacobian for Constant-Acceleration Motion with Specified Time Step

Compute the state Jacobian for two-dimensional constant-acceleration motion. Set the step time to 0.5 seconds.

```
state = [1,1,1,2,1,0].';
jacobian = constaccjac(state,0.5)
```

```
jacobian = 6×6

    1.0000    0.5000    0.1250         0         0         0
         0    1.0000    0.5000         0         0         0
         0         0    1.0000         0         0         0
         0         0         0    1.0000    0.5000    0.1250
         0         0         0         0    1.0000    0.5000
         0         0         0         0         0    1.0000
```

Input Arguments

`state` — Kalman filter state vector
real-valued *3N*-element vector

Kalman filter state vector for constant-acceleration motion, specified as a real-valued *3N*-element vector. *N* is the number of spatial degrees of freedom of motion. For each spatial degree of motion, the state vector takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	`[x;vx;ax]`
2-D	`[x;vx;ax;y;vy;ay]`
3-D	`[x;vx;ax;y;vy;ay;z;vz;az]`

For example, `x` represents the *x*-coordinate, `vx` represents the velocity in the *x*-direction, and `ax` represents the acceleration in the *x*-direction. If the motion model is in one-dimensional space, the *y*- and *z*-axes are assumed to be zero. If the motion model is in two-dimensional space, values along the *z*-axis are assumed to be zero. Position coordinates are in meters. Velocity coordinates are in meters/second. Acceleration coordinates are in meters/second^2.

Example: `[5;0.1;0.01;0;-0.2;-0.01;-3;0.05;0]`

Data Types: `double`

`dt` — Time step interval of filter
`1.0` (default) | positive scalar

Time step interval of filter, specified as a positive scalar. Time units are in seconds.

Example: `0.5`

Data Types: `single` | `double`

`w` — State noise
scalar | real-valued *N*-by-1 vector

State noise, specified as a scalar or real-valued real valued *N*-by-1 vector. *N* is the number of motion dimensions. For example, *N* = 2 for the 2-D motion. If specified as a scalar, the scalar value is expanded to a *N*-by-1 vector.

Data Types: `single` | `double`

Output Arguments

`jacobian` — Constant-acceleration motion Jacobian
real-valued *3N*-by-*3N* matrix

Constant-acceleration motion Jacobian, returned as a real-valued *3N*-by-*3N* matrix.

`noisejacobian` — Constant acceleration motion noise Jacobian
real-valued *3N*-by-*N* matrix

Constant acceleration motion noise Jacobian, returned as a real-valued *3N*-by-*N* matrix. *N* is the number of spatial degrees of motion. For example, $N = 2$ for the 2-D motion. The Jacobian is constructed from the partial derivatives of the state at the updated time step with respect to the noise components.

Algorithms

For a two-dimensional constant-acceleration process, the Jacobian matrix after a time step, *T*, is block diagonal:

$$\begin{bmatrix} 1 & T & \frac{1}{2}T^2 & 0 & 0 & 0 \\ 0 & 1 & T & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & T & \frac{1}{2}T^2 \\ 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The block for each spatial dimension has this form:

$$\begin{bmatrix} 1 & T & \frac{1}{2}T^2 \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix}$$

For each additional spatial dimension, add an identical block.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`cameas` | `cameasjac` | `constacc` | `constturn` | `constturnjac` | `constvel` | `constveljac` | `ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac`

Objects
`trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

constturn

Constant turn-rate motion model

Syntax

```
updatedstate = constturn(state)
updatedstate = constturn(state,dt)
updatedstate = constturn(state,w,dt)
```

Description

`updatedstate = constturn(state)` returns the updated state, `updatedstate`, obtained from the previous state, `state`, after a one-second step time for motion modelled as constant turn rate. Constant turn rate means that motion in the *x*-*y* plane follows a constant angular velocity and motion in the vertical *z* directions follows a constant velocity model.

`updatedstate = constturn(state,dt)` also specifies the time step, `dt`.

`updatedstate = constturn(state,w,dt)` also specifies noise, `w`.

Examples

Update State for Constant Turn-Rate Motion

Define an initial state for 2-D constant turn-rate motion. The turn rate is 12 degrees per second. Update the state to one second later.

```
state = [500,0,0,100,12].';
state = constturn(state)
```

```
state = 5×1

  489.5662
  -20.7912
   99.2705
   97.8148
   12.0000
```

Update State for Constant Turn-Rate Motion with Specified Time Step

Define an initial state for 2-D constant turn-rate motion. The turn rate is 12 degrees per second. Update the state to 0.1 seconds later.

```
state = [500,0,0,100,12].';
state = constturn(state,0.1)
```

```
state = 5×1
```

```
 499.8953
  -2.0942
   9.9993
  99.9781
  12.0000
```

Input Arguments

`state` — State vector

real-valued 5-element vector | real-valued 7-element vector | 5-by-*N* real-valued matrix | 7-by-*N* real-valued matrix

State vector for a constant turn-rate motion model in two or three spatial dimensions, specified as a real-valued vector or matrix.

- When specified as a 5-element vector, the state vector describes 2-D motion in the *x*-*y* plane. You can specify the state vector as a row or column vector. The components of the state vector are `[x;vx;y;vy;omega]` where `x` represents the *x*-coordinate and `vx` represents the velocity in the *x*-direction. `y` represents the *y*-coordinate and `vy` represents the velocity in the *y*-direction. `omega` represents the turn rate.

  When specified as a 5-by-*N* matrix, each column represents a different state vector *N* represents the number of states.
- When specified as a 7-element vector, the state vector describes 3-D motion. You can specify the state vector as a row or column vector. The components of the state vector are `[x;vx;y;vy;omega;z;vz]` where `x` represents the *x*-coordinate and `vx` represents the velocity in the *x*-direction. `y` represents the *y*-coordinate and `vy` represents the velocity in the *y*-direction. `omega` represents the turn rate. `z` represents the *z*-coordinate and `vz` represents the velocity in the *z*-direction.

  When specified as a 7-by-*N* matrix, each column represents a different state vector. *N* represents the number of states.

Position coordinates are in meters. Velocity coordinates are in meters/second. Turn rate is in degrees/second.

Example: `[5;0.1;4;-0.2;0.01]`

Data Types: `double`

`dt` — Time step interval of filter

`1.0` (default) | positive scalar

Time step interval of filter, specified as a positive scalar. Time units are in seconds.

Example: `0.5`

Data Types: `single` | `double`

`w` — State noise

scalar | real-valued (*D*+1)-by-*N* matrix

State noise, specified as a scalar or real-valued (*D*+1)-length -by-*N* matrix. *D* is the number of motion dimensions and *N* is the number of state vectors. The components are each columns are [ax;ay;alpha] for 2-D motion or [ax;ay;alpha;az] for 3-D motion. ax, ay, and az are the linear acceleration noise values in the *x*-, *y*-, and *z*-axes, respectively, and alpha is the angular acceleration noise value. If specified as a scalar, the value expands to a (D+1)-by-N matrix.

Data Types: single | double

Output Arguments

updatedstate — Updated state vector
real-valued column or row vector | real-valued matrix

Updated state vector, returned as a real-valued vector or real-valued matrix with same number of elements and dimensions as the input state vector.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
cameas | cameasjac | constacc | constaccjac | constturnjac | constvel | constveljac | ctmeas | ctmeasjac | cvmeas | cvmeasjac | initctekf | initctukf

Objects
trackingEKF | trackingUKF

Introduced in R2017a

constturnjac

Jacobian for constant turn-rate motion

Syntax

```
jacobian = constturnjac(state)
jacobian = constturnjac(state,dt)
[jacobian,noisejacobian] = constturnjac(state,w,dt)
```

Description

`jacobian = constturnjac(state)` returns the updated Jacobian, `jacobian`, for constant turn-rate Kalman filter motion model for a one-second step time. The `state` argument specifies the current state of the filter. Constant turn rate means that motion in the *x*-*y* plane follows a constant angular velocity and motion in the vertical *z* directions follows a constant velocity model.

`jacobian = constturnjac(state,dt)` specifies the time step, `dt`.

`[jacobian,noisejacobian] = constturnjac(state,w,dt)` also specifies noise, `w`, and returns the Jacobian, `noisejacobian`, of the state with respect to the noise.

Examples

Compute State Jacobian for Constant Turn-Rate Motion

Compute the Jacobian for a constant turn-rate motion state. Assume the turn rate is 12 degrees/second. The time step is one second.

```
state = [500,0,0,100,12];
jacobian = constturnjac(state)
```

```
jacobian = 5×5

    1.0000    0.9927         0   -0.1043   -0.8631
         0    0.9781         0   -0.2079   -1.7072
         0    0.1043    1.0000    0.9927   -0.1213
         0    0.2079         0    0.9781   -0.3629
         0         0         0         0    1.0000
```

Compute State Jacobian for Constant Turn-Rate Motion with Specified Time Step

Compute the Jacobian for a constant turn-rate motion state. Assume the turn rate is 12 degrees/second. The time step is 0.1 second.

```
state = [500,0,0,100,12];
jacobian = constturnjac(state,0.1)
```

```
jacobian = 5×5

    1.0000    0.1000         0   -0.0010   -0.0087
         0    0.9998         0   -0.0209   -0.1745
         0    0.0010    1.0000    0.1000   -0.0001
         0    0.0209         0    0.9998   -0.0037
         0         0         0         0    1.0000
```

Input Arguments

`state` — State vector

real-valued 5-element vector | real-valued 7-element vector

State vector for a constant turn-rate motion model in two or three spatial dimensions, specified as a real-valued vector.

- When specified as a 5-element vector, the state vector describes 2-D motion in the *x*-*y* plane. You can specify the state vector as a row or column vector. The components of the state vector are `[x;vx;y;vy;omega]` where `x` represents the *x*-coordinate and `vx` represents the velocity in the *x*-direction. `y` represents the *y*-coordinate and `vy` represents the velocity in the *y*-direction. `omega` represents the turn rate.
- When specified as a 7-element vector, the state vector describes 3-D motion. You can specify the state vector as a row or column vector. The components of the state vector are `[x;vx;y;vy;omega;z;vz]` where `x` represents the *x*-coordinate and `vx` represents the velocity in the *x*-direction. `y` represents the *y*-coordinate and `vy` represents the velocity in the *y*-direction. `omega` represents the turn rate. `z` represents the *z*-coordinate and `vz` represents the velocity in the *z*-direction.

Position coordinates are in meters. Velocity coordinates are in meters/second. Turn rate is in degrees/second.

Example: `[5;0.1;4;-0.2;0.01]`

Data Types: `double`

`dt` — Time step interval of filter

`1.0` (default) | positive scalar

Time step interval of filter, specified as a positive scalar. Time units are in seconds.

Example: `0.5`

Data Types: `single` | `double`

`w` — State noise

scalar | real-valued (D+1) vector

State noise, specified as a scalar or real-valued M-by-(D+1)-length vector. D is the number of motion dimensions. D is two for 2-D motion and D is three for 3-D motion. The vector components are `[ax;ay;alpha]` for 2-D motion or `[ax;ay;alpha;az]` for 3-D motion. `ax`, `ay`, and `az` are the linear acceleration noise values in the *x*-, *y*-, and *z*-axes, respectively, and `alpha` is the angular acceleration noise value. If specified as a scalar, the value expands to a (D+1) vector.

Data Types: `single` | `double`

Output Arguments

`jacobian` — Constant turn-rate motion Jacobian
real-valued 5-by-5 matrix | real-valued 7-by-7 matrix

Constant turn-rate motion Jacobian, returned as a real-valued 5-by-5 matrix or 7-by-7 matrix depending on the size of the `state` vector. The Jacobian is constructed from the partial derivatives of the state at the updated time step with respect to the state at the previous time step.

`noisejacobian` — Constant turn-rate motion noise Jacobian
real-valued 5-by-5 matrix | real-valued 7-by-7 matrix

Constant turn-rate motion noise Jacobian, returned as a real-valued 5-by-(D+1) matrix where D is two for 2-D motion or a real-valued 7-by-(D+1) matrix where D is three for 3-D motion. The Jacobian is constructed from the partial derivatives of the state at the updated time step with respect to the noise components.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`cameas` | `cameasjac` | `constacc` | `constaccjac` | `constturn` | `constvel` | `constveljac` | `ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac` | `initctekf`

Objects
`trackingEKF`

Introduced in R2017a

constvel

Constant velocity state update

Syntax

```
updatedstate = constvel(state)
updatedstate = constvel(state,dt)
updatedstate = constvel(state,w,dt)
```

Description

`updatedstate = constvel(state)` returns the updated state, `state`, of a constant-velocity Kalman filter motion model after a one-second time step.

`updatedstate = constvel(state,dt)` specifies the time step, `dt`.

`updatedstate = constvel(state,w,dt)` also specifies state noise, `w`.

Examples

Update State for Constant-Velocity Motion

Update the state of two-dimensional constant-velocity motion for a time interval of one second.

```
state = [1;1;2;1];
state = constvel(state)
```

```
state = 4×1

     2
     1
     3
     1
```

Update State for Constant-Velocity Motion with Specified Time Step

Update the state of two-dimensional constant-velocity motion for a time interval of 1.5 seconds.

```
state = [1;1;2;1];
state = constvel(state,1.5)
```

```
state = 4×1

    2.5000
    1.0000
    3.5000
    1.0000
```

Input Arguments

`state` — Kalman filter state vector

real-valued 2*N*-element vector

Kalman filter state vector for constant-velocity motion, specified as a real-valued 2*N*-element column vector where *N* is the number of spatial degrees of freedom of motion. The `state` is expected to be Cartesian state. For each spatial degree of motion, the state vector takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	`[x;vx]`
2-D	`[x;vx;y;vy]`
3-D	`[x;vx;y;vy;z;vz]`

For example, `x` represents the *x*-coordinate and `vx` represents the velocity in the *x*-direction. If the motion model is 1-D, values along the *y* and *z* axes are assumed to be zero. If the motion model is 2-D, values along the *z* axis are assumed to be zero. Position coordinates are in meters and velocity coordinates are in meters/sec.

Example: `[5;.1;0;-.2;-3;.05]`

Data Types: `single` | `double`

`dt` — Time step interval of filter

`1.0` (default) | positive scalar

Time step interval of filter, specified as a positive scalar. Time units are in seconds.

Example: `0.5`

Data Types: `single` | `double`

`w` — State noise

scalar | real-valued *D*-by-*N* matrix

State noise, specified as a scalar or real-valued *D*-by-*N* matrix. *D* is the number of motion dimensions and *N* is the number of state vectors. For example, $D = 2$ for the 2-D motion. If specified as a scalar, the scalar value is expanded to a *D*-by-*N* matrix.

Data Types: `single` | `double`

Output Arguments

`updatedstate` — Updated state vector

real-valued column or row vector | real-valued matrix

Updated state vector, returned as a real-valued vector or real-valued matrix with same number of elements and dimensions as the input state vector.

Algorithms

For a two-dimensional constant-velocity process, the state transition matrix after a time step, *T*, is block diagonal as shown here.

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

The block for each spatial dimension is:

$$\begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix}$$

For each additional spatial dimension, add an identical block.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`cameas` | `cameasjac` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constveljac` | `ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac`

Objects

`trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

constveljac

Jacobian for constant-velocity motion

Syntax

```
jacobian = constveljac(state)
jacobian = constveljac(state,dt)
[jacobian,noisejacobian] = constveljac(state,w,dt)
```

Description

`jacobian = constveljac(state)` returns the updated Jacobian , `jacobian`, for a constant-velocity Kalman filter motion model for a step time of one second. The `state` argument specifies the current state of the filter.

`jacobian = constveljac(state,dt)` specifies the time step, `dt`.

`[jacobian,noisejacobian] = constveljac(state,w,dt)` specifies the state noise, `w`, and returns the Jacobian, `noisejacobian`, of the state with respect to the noise.

Examples

Compute State Jacobian for Constant-Velocity Motion

Compute the state Jacobian for a two-dimensional constant-velocity motion model for a one second update time.

```
state = [1,1,2,1].';
jacobian = constveljac(state)
```

```
jacobian = 4×4

     1     1     0     0
     0     1     0     0
     0     0     1     1
     0     0     0     1
```

Compute State Jacobian for Constant-Velocity Motion with Specified Time Step

Compute the state Jacobian for a two-dimensional constant-velocity motion model for a half-second update time.

```
state = [1;1;2;1];
```

Compute the state update Jacobian for 0.5 second.

```
jacobian = constveljac(state,0.5)
```

```
jacobian = 4×4

    1.0000    0.5000         0         0
         0    1.0000         0         0
         0         0    1.0000    0.5000
         0         0         0    1.0000
```

Input Arguments

`state` — Kalman filter state vector
real-valued *2N*-element vector

Kalman filter state vector for constant-velocity motion, specified as a real-valued *2N*-element column vector where *N* is the number of spatial degrees of freedom of motion. The `state` is expected to be Cartesian state. For each spatial degree of motion, the state vector takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	`[x;vx]`
2-D	`[x;vx;y;vy]`
3-D	`[x;vx;y;vy;z;vz]`

For example, `x` represents the *x*-coordinate and `vx` represents the velocity in the *x*-direction. If the motion model is 1-D, values along the *y* and *z* axes are assumed to be zero. If the motion model is 2-D, values along the *z* axis are assumed to be zero. Position coordinates are in meters and velocity coordinates are in meters/sec.

Example: `[5;.1;0;-.2;-3;.05]`

Data Types: `single` | `double`

`dt` — Time step interval of filter
`1.0` (default) | positive scalar

Time step interval of filter, specified as a positive scalar. Time units are in seconds.

Example: `0.5`

Data Types: `single` | `double`

`w` — State noise
scalar | real-valued *N*-by-1 vector

State noise, specified as a scalar or real-valued real valued *N*-by-1 vector. *N* is the number of motion dimensions. For example, *N* = 2 for the 2-D motion. If specified as a scalar, the scalar value is expanded to an *N*-by-1 vector.

Data Types: `single` | `double`

Output Arguments

`jacobian` — Constant-velocity motion Jacobian
real-valued *2N*-by-*2N* matrix

Constant-velocity motion Jacobian, returned as a real-valued 2*N*-by-2*N* matrix. *N* is the number of spatial degrees of motion.

`noisejacobian` — Constant velocity motion noise Jacobian
real-valued 2*N*-by-*N* matrix

Constant velocity motion noise Jacobian, returned as a real-valued 2*N*-by-*N* matrix. *N* is the number of spatial degrees of motion. The Jacobian is constructed from the partial derivatives of the state at the updated time step with respect to the noise components.

Algorithms

For a two-dimensional constant-velocity motion, the Jacobian matrix for a time step, *T*, is block diagonal:

$$\begin{bmatrix} 1 & T & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & T \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The block for each spatial dimension has this form:

$$\begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix}$$

For each additional spatial dimension, add an identical block.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`cameas` | `cameasjac` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac`

Objects
`trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

ctmeas

Measurement function for constant turn-rate motion

Syntax

```
measurement = ctmeas(state)
measurement = ctmeas(state,frame)
measurement = ctmeas(state,frame,sensorpos)
measurement = ctmeas(state,frame,sensorpos,sensorvel)
measurement = ctmeas(state,frame,sensorpos,sensorvel,laxes)
measurement = ctmeas(state,measurementParameters)
```

Description

`measurement = ctmeas(state)` returns the measurement for a constant turn-rate Kalman filter motion model in rectangular coordinates. The `state` argument specifies the current state of the filter.

`measurement = ctmeas(state,frame)` also specifies the measurement coordinate system, `frame`.

`measurement = ctmeas(state,frame,sensorpos)` also specifies the sensor position, `sensorpos`.

`measurement = ctmeas(state,frame,sensorpos,sensorvel)` also specifies the sensor velocity, `sensorvel`.

`measurement = ctmeas(state,frame,sensorpos,sensorvel,laxes)` also specifies the local sensor axes orientation, `laxes`.

`measurement = ctmeas(state,measurementParameters)` specifies the measurement parameters, `measurementParameters`.

Examples

Create Measurement from Constant Turn-Rate Motion in Rectangular Frame

Create a measurement from an object undergoing constant turn-rate motion. The state is the position and velocity in each dimension and the turn-rate. The measurements are in rectangular coordinates.

```
state = [1;10;2;20;5];
measurement = ctmeas(state)
```

```
measurement = 3×1

     1
     2
     0
```

The *z*-component of the measurement is zero.

Create Measurement from Constant Turn-Rate Motion in Spherical Frame

Define the state of an object in 2-D constant turn-rate motion. The state is the position and velocity in each dimension, and the turn rate. The measurements are in spherical coordinates.

```
state = [1;10;2;20;5];
measurement = ctmeas(state,'spherical')
```

```
measurement = 4×1

   63.4349
         0
    2.2361
   22.3607
```

The elevation of the measurement is zero and the range rate is positive indicating that the object is moving away from the sensor.

Create Measurement from Constant Turn-Rate Motion in Translated Spherical Frame

Define the state of an object moving in 2-D constant turn-rate motion. The state consists of position and velocity, and the turn rate. The measurements are in spherical coordinates with respect to a frame located at `[20;40;0]`.

```
state = [1;10;2;20;5];
measurement = ctmeas(state,'spherical',[20;40;0])
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -22.3607
```

The elevation of the measurement is zero and the range rate is negative indicating that the object is moving toward the sensor.

Create Measurement from Constant Turn-Rate Motion using Measurement Parameters

Define the state of an object moving in 2-D constant turn-rate motion. The state consists of position and velocity, and the turn rate. The measurements are in spherical coordinates with respect to a frame located at `[20;40;0]`.

```
state2d = [1;10;2;20;5];
frame = 'spherical';
```

```
sensorpos = [20;40;0];
sensorvel = [0;5;0];
laxes = eye(3);
measurement = ctmeas(state2d,frame,sensorpos,sensorvel,laxes)
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -17.8885
```

The elevation of the measurement is zero and the range rate is negative indicating that the object is moving toward the sensor.

Put the measurement parameters in a structure and use the alternative syntax.

```
measparm = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes);
measurement = ctmeas(state2d,measparm)
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -17.8885
```

Input Arguments

`state` — State vector

real-valued 5-element vector | real-valued 7-element vector | 5-by-*N* real-valued matrix | 7-by-*N* real-valued matrix

State vector for a constant turn-rate motion model in two or three spatial dimensions, specified as a real-valued vector or matrix.

- When specified as a 5-element vector, the state vector describes 2-D motion in the *x*-*y* plane. You can specify the state vector as a row or column vector. The components of the state vector are `[x;vx;y;vy;omega]` where `x` represents the *x*-coordinate and `vx` represents the velocity in the *x*-direction. `y` represents the *y*-coordinate and `vy` represents the velocity in the *y*-direction. `omega` represents the turn rate.

  When specified as a 5-by-*N* matrix, each column represents a different state vector *N* represents the number of states.
- When specified as a 7-element vector, the state vector describes 3-D motion. You can specify the state vector as a row or column vector. The components of the state vector are `[x;vx;y;vy;omega;z;vz]` where `x` represents the *x*-coordinate and `vx` represents the velocity in the *x*-direction. `y` represents the *y*-coordinate and `vy` represents the velocity in the *y*-direction. `omega` represents the turn rate. `z` represents the *z*-coordinate and `vz` represents the velocity in the *z*-direction.

When specified as a 7-by-*N* matrix, each column represents a different state vector. *N* represents the number of states.

Position coordinates are in meters. Velocity coordinates are in meters/second. Turn rate is in degrees/second.

Example: `[5;0.1;4;-0.2;0.01]`

Data Types: `double`

`frame` — Measurement output frame
`'rectangular'` (default) | `'spherical'`

Measurement output frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of *x*, *y*, and *z* Cartesian coordinates. When specified as `'spherical'`, a measurement consists of azimuth, elevation, range, and range rate.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in meters.

Data Types: `double`

`sensorvel` — Sensor velocity
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor velocity with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in m/s.

Data Types: `double`

`laxes` — Local sensor coordinate axes
`[1,0,0;0,1,0;0,0,1]` (default) | 3-by-3 orthogonal matrix

Local sensor coordinate axes, specified as a 3-by-3 orthogonal matrix. Each column specifies the direction of the local *x*-, *y*-, and *z*-axes, respectively, with respect to the navigation frame. That is, the matrix is the rotation matrix from the global frame to the sensor frame.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure | array of structure

Measurement parameters, specified as a structure or an array of structures. The fields of the structure are:

Field	Description	Example
Frame	Frame used to report measurements, specified as one of these values: • 'rectangular' — Detections are reported in rectangular coordinates. • 'spherical' — Detections are reported in spherical coordinates.	'spherical'
OriginPosition	Position offset of the origin of the frame relative to the parent frame, specified as an [x y z] real-valued vector.	[0 0 0]
OriginVelocity	Velocity offset of the origin of the frame relative to the parent frame, specified as a [vx vy vz] real-valued vector.	[0 0 0]
Orientation	Frame rotation matrix, specified as a 3-by-3 real-valued orthonormal matrix.	[1 0 0; 0 1 0; 0 0 1]
HasAzimuth	Logical scalar indicating if azimuth is included in the measurement.	1
HasElevation	Logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if HasElevation is false, the reported measurements assume 0 degrees of elevation.	1
HasRange	Logical scalar indicating if range is included in the measurement.	1
HasVelocity	Logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if HasVelocity is false, the measurements are reported as [x y z]. If HasVelocity is true, measurements are reported as [x y z vx vy vz].	1

Field	Description	Example
`IsParentToChild`	Logical scalar indicating if `Orientation` performs a frame rotation from the parent coordinate frame to the child coordinate frame. When `IsParentToChild` is `false`, then `Orientation` performs a frame rotation from the child coordinate frame to the parent coordinate frame.	`0`

Data Types: `struct`

Output Arguments

`measurement` — Measurement vector
N-by-1 column vector

Measurement vector, returned as an *N*-by-1 column vector. The form of the measurement depends upon which syntax you use.

- When the syntax does not use the `measurementParameters` argument, the measurement vector is `[x,y,z]` when the `frame` input argument is set to `'rectangular'` and `[az;el;r;rr]` when the `frame` is set to `'spherical'`.
- When the syntax uses the `measurementParameters` argument, the size of the measurement vector depends on the values of the `frame`, `HasVelocity`, and `HasElevation` fields in the `measurementParameters` structure.

<table>
<tr><th>frame</th><th>measurement</th></tr>
<tr><td><code>'spherical'</code></td><td>Specifies the azimuth angle, az, elevation angle, el, range, r, and range rate, rr, of the object with respect to the local ego vehicle coordinate system. Positive values for range rate indicate that an object is moving away from the sensor.

Spherical measurements
<table>
<tr><th></th><th></th><th colspan="2">HasElevation</th></tr>
<tr><td></td><td></td><td>false</td><td>true</td></tr>
<tr><td rowspan="2">HasVelocity</td><td>false</td><td><code>[az;r]</code></td><td><code>[az;el;r]</code></td></tr>
<tr><td>true</td><td><code>[az;r;rr]</code></td><td><code>[az;el;r;rr]</code></td></tr>
</table>

Angle units are in degrees, range units are in meters, and range rate units are in m/s.</td></tr>
</table>

<table>
<tr><th>frame</th><th>measurement</th></tr>
<tr><td><code>'rectangular</code></td><td>Specifies the Cartesian position and velocity coordinates of the tracked object with respect to the ego vehicle coordinate system.

Rectangular measurements
<table><tr><td rowspan="2">HasVelocity</td><td>false</td><td><code>[x;y;y]</code></td></tr><tr><td>true</td><td><code>[x;y;z;vx;vy;vz]</code></td></tr></table>
Position units are in meters and velocity units are in m/s.</td></tr>
</table>

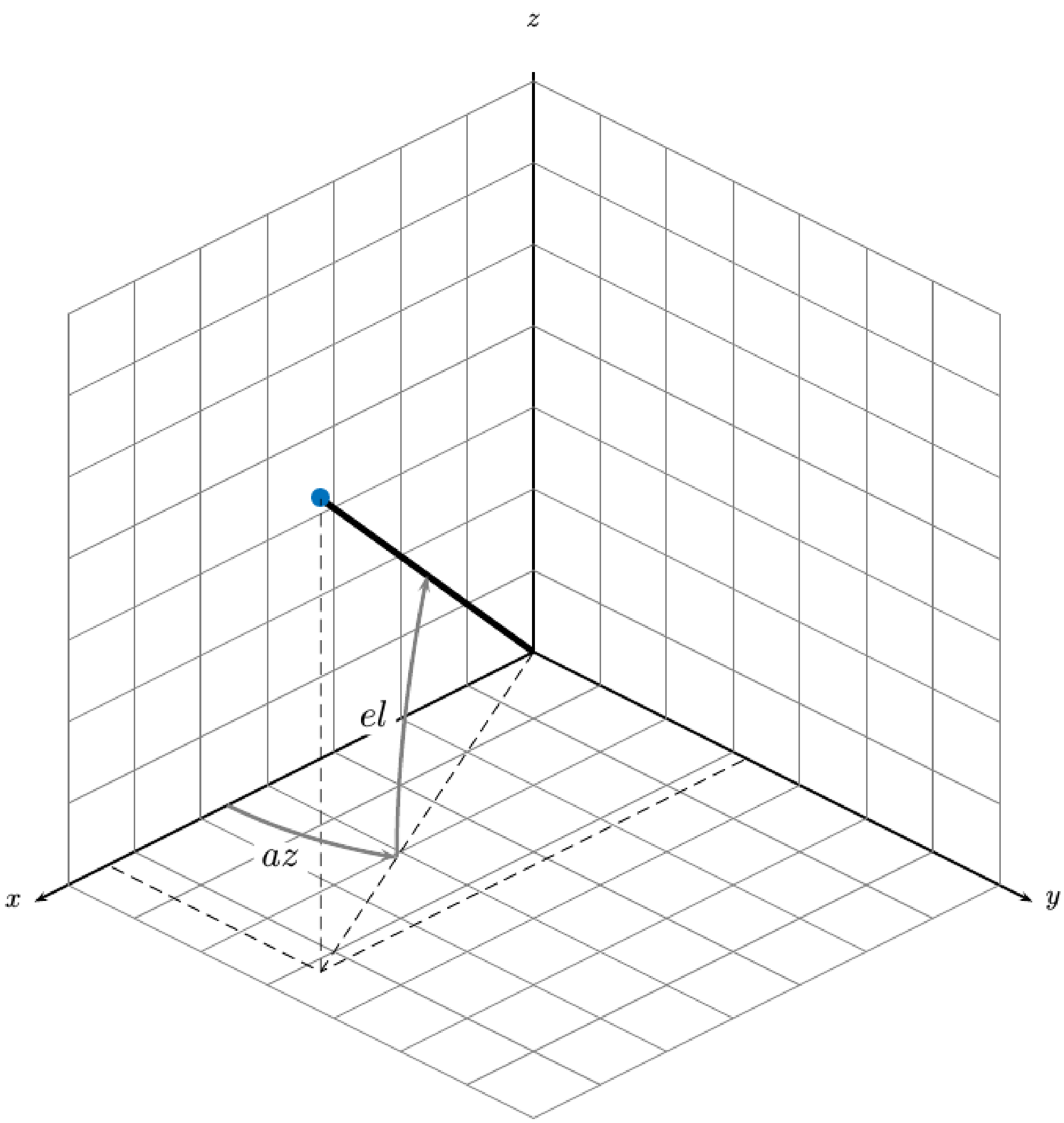
Data Types: `double`

More About

Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Automated Driving Toolbox.

The azimuth angle of a vector is the angle between the *x*-axis and its orthogonal projection onto the *xy* plane. The angle is positive in going from the *x* axis toward the *y* axis. Azimuth angles lie between –180 and 180 degrees. The elevation angle is the angle between the vector and its orthogonal projection onto the *xy*-plane. The angle is positive when going toward the positive *z*-axis from the *xy* plane.



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`cameas` | `cameasjac` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `constveljac` | `ctmeasjac` | `cvmeas` | `cvmeasjac`

Objects

`trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

ctmeasjac

Jacobian of measurement function for constant turn-rate motion

Syntax

```
measurementjac = ctmeasjac(state)
measurementjac = ctmeasjac(state,frame)
measurementjac = ctmeasjac(state,frame,sensorpos)
measurementjac = ctmeasjac(state,frame,sensorpos,sensorvel)
measurementjac = ctmeasjac(state,frame,sensorpos,sensorvel,laxes)
measurementjac = ctmeasjac(state,measurementParameters)
```

Description

`measurementjac = ctmeasjac(state)` returns the measurement Jacobian, `measurementjac`, for a constant turn-rate Kalman filter motion model in rectangular coordinates. `state` specifies the current state of the track.

`measurementjac = ctmeasjac(state,frame)` also specifies the measurement coordinate system, `frame`.

`measurementjac = ctmeasjac(state,frame,sensorpos)` also specifies the sensor position, `sensorpos`.

`measurementjac = ctmeasjac(state,frame,sensorpos,sensorvel)` also specifies the sensor velocity, `sensorvel`.

`measurementjac = ctmeasjac(state,frame,sensorpos,sensorvel,laxes)` also specifies the local sensor axes orientation, `laxes`.

`measurementjac = ctmeasjac(state,measurementParameters)` specifies the measurement parameters, `measurementParameters`.

Examples

Measurement Jacobian of Constant Turn-Rate Motion in Rectangular Frame

Define the state of an object in 2-D constant turn-rate motion. The state is the position and velocity in each dimension, and the turn rate. Construct the measurement Jacobian in rectangular coordinates.

```
state = [1;10;2;20;5];
jacobian = ctmeasjac(state)
```

```
jacobian = 3×5

     1     0     0     0     0
     0     0     1     0     0
     0     0     0     0     0
```

Measurement Jacobian of Constant Turn-Rate Motion in Spherical Frame

Define the state of an object in 2-D constant turn-rate motion. The state is the position and velocity in each dimension, and the turn rate. Compute the measurement Jacobian with respect to spherical coordinates.

```
state = [1;10;2;20;5];
measurementjac = ctmeasjac(state,'spherical')
```

```
measurementjac = 4×5

  -22.9183         0   11.4592         0         0
         0         0         0         0         0
    0.4472         0    0.8944         0         0
    0.0000    0.4472    0.0000    0.8944         0
```

Measurement Jacobian of Constant Turn-Rate Object in Translated Spherical Frame

Define the state of an object in 2-D constant turn-rate motion. The state is the position and velocity in each dimension, and the turn rate. Compute the measurement Jacobian with respect to spherical coordinates centered at `[5;-20;0]`.

```
state = [1;10;2;20;5];
sensorpos = [5;-20;0];
measurementjac = ctmeasjac(state,'spherical',sensorpos)
```

```
measurementjac = 4×5

   -2.5210         0   -0.4584         0         0
         0         0         0         0         0
   -0.1789         0    0.9839         0         0
    0.5903   -0.1789    0.1073    0.9839         0
```

Measurement Jacobian of Constant Turn-Rate Object Using Measurement Parameters

Define the state of an object in 2-D constant turn-rate motion. The state is the position and velocity in each dimension, and the turn rate. Compute the measurement Jacobian with respect to spherical coordinates centered at `[25;-40;0]`.

```
state2d = [1;10;2;20;5];
sensorpos = [25,-40,0].';
frame = 'spherical';
sensorvel = [0;5;0];
laxes = eye(3);
measurementjac = ctmeasjac(state2d,frame,sensorpos,sensorvel,laxes)
```

```
measurementjac = 4×5
```

```
   -1.0284         0   -0.5876         0         0
         0         0         0         0         0
   -0.4961         0    0.8682         0         0
    0.2894   -0.4961    0.1654    0.8682         0
```

Put the measurement parameters in a structure and use the alternative syntax.

```
measparm = struct('Frame',frame,'OriginPosition',sensorpos,'OriginVelocity',sensorvel, ...
    'Orientation',laxes);
measurementjac = ctmeasjac(state2d,measparm)
```

```
measurementjac = 4×5

   -1.0284         0   -0.5876         0         0
         0         0         0         0         0
   -0.4961         0    0.8682         0         0
    0.2894   -0.4961    0.1654    0.8682         0
```

Input Arguments

`state` — State vector

real-valued 5-element vector | real-valued 7-element vector | 5-by-*N* real-valued matrix | 7-by-*N* real-valued matrix

State vector for a constant turn-rate motion model in two or three spatial dimensions, specified as a real-valued vector or matrix.

- When specified as a 5-element vector, the state vector describes 2-D motion in the *x*-*y* plane. You can specify the state vector as a row or column vector. The components of the state vector are `[x;vx;y;vy;omega]` where `x` represents the *x*-coordinate and `vx` represents the velocity in the *x*-direction. `y` represents the *y*-coordinate and `vy` represents the velocity in the *y*-direction. `omega` represents the turn rate.

  When specified as a 5-by-*N* matrix, each column represents a different state vector *N* represents the number of states.
- When specified as a 7-element vector, the state vector describes 3-D motion. You can specify the state vector as a row or column vector. The components of the state vector are `[x;vx;y;vy;omega;z;vz]` where `x` represents the *x*-coordinate and `vx` represents the velocity in the *x*-direction. `y` represents the *y*-coordinate and `vy` represents the velocity in the *y*-direction. `omega` represents the turn rate. `z` represents the *z*-coordinate and `vz` represents the velocity in the *z*-direction.

  When specified as a 7-by-*N* matrix, each column represents a different state vector. *N* represents the number of states.

Position coordinates are in meters. Velocity coordinates are in meters/second. Turn rate is in degrees/second.

Example: `[5;0.1;4;-0.2;0.01]`

Data Types: `double`

`frame` — Measurement output frame
`'rectangular'` (default) | `'spherical'`

Measurement output frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of *x*, *y*, and *z* Cartesian coordinates. When specified as `'spherical'`, a measurement consists of azimuth, elevation, range, and range rate.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in meters.

Data Types: `double`

`sensorvel` — Sensor velocity
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor velocity with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in m/s.

Data Types: `double`

`laxes` — Local sensor coordinate axes
`[1,0,0;0,1,0;0,0,1]` (default) | 3-by-3 orthogonal matrix

Local sensor coordinate axes, specified as a 3-by-3 orthogonal matrix. Each column specifies the direction of the local *x*-, *y*-, and *z*-axes, respectively, with respect to the navigation frame. That is, the matrix is the rotation matrix from the global frame to the sensor frame.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure | array of structure

Measurement parameters, specified as a structure or an array of structures. The fields of the structure are:

Field	Description	Example
`Frame`	Frame used to report measurements, specified as one of these values: • `'rectangular'` — Detections are reported in rectangular coordinates. • `'spherical'` — Detections are reported in spherical coordinates.	`'spherical'`

Field	Description	Example
OriginPosition	Position offset of the origin of the frame relative to the parent frame, specified as an [x y z] real-valued vector.	[0 0 0]
OriginVelocity	Velocity offset of the origin of the frame relative to the parent frame, specified as a [vx vy vz] real-valued vector.	[0 0 0]
Orientation	Frame rotation matrix, specified as a 3-by-3 real-valued orthonormal matrix.	[1 0 0; 0 1 0; 0 0 1]
HasAzimuth	Logical scalar indicating if azimuth is included in the measurement.	1
HasElevation	Logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if HasElevation is false, the reported measurements assume 0 degrees of elevation.	1
HasRange	Logical scalar indicating if range is included in the measurement.	1
HasVelocity	Logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if HasVelocity is false, the measurements are reported as [x y z]. If HasVelocity is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if Orientation performs a frame rotation from the parent coordinate frame to the child coordinate frame. When IsParentToChild is false, then Orientation performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

Data Types: struct

Output Arguments

`measurementjac` — Measurement Jacobian
real-valued 3-by-5 matrix | real-valued 4-by-5 matrix

Measurement Jacobian, returned as a real-valued 3-by-5 or 4-by-5 matrix. The row dimension and interpretation depend on value of the `frame` argument.

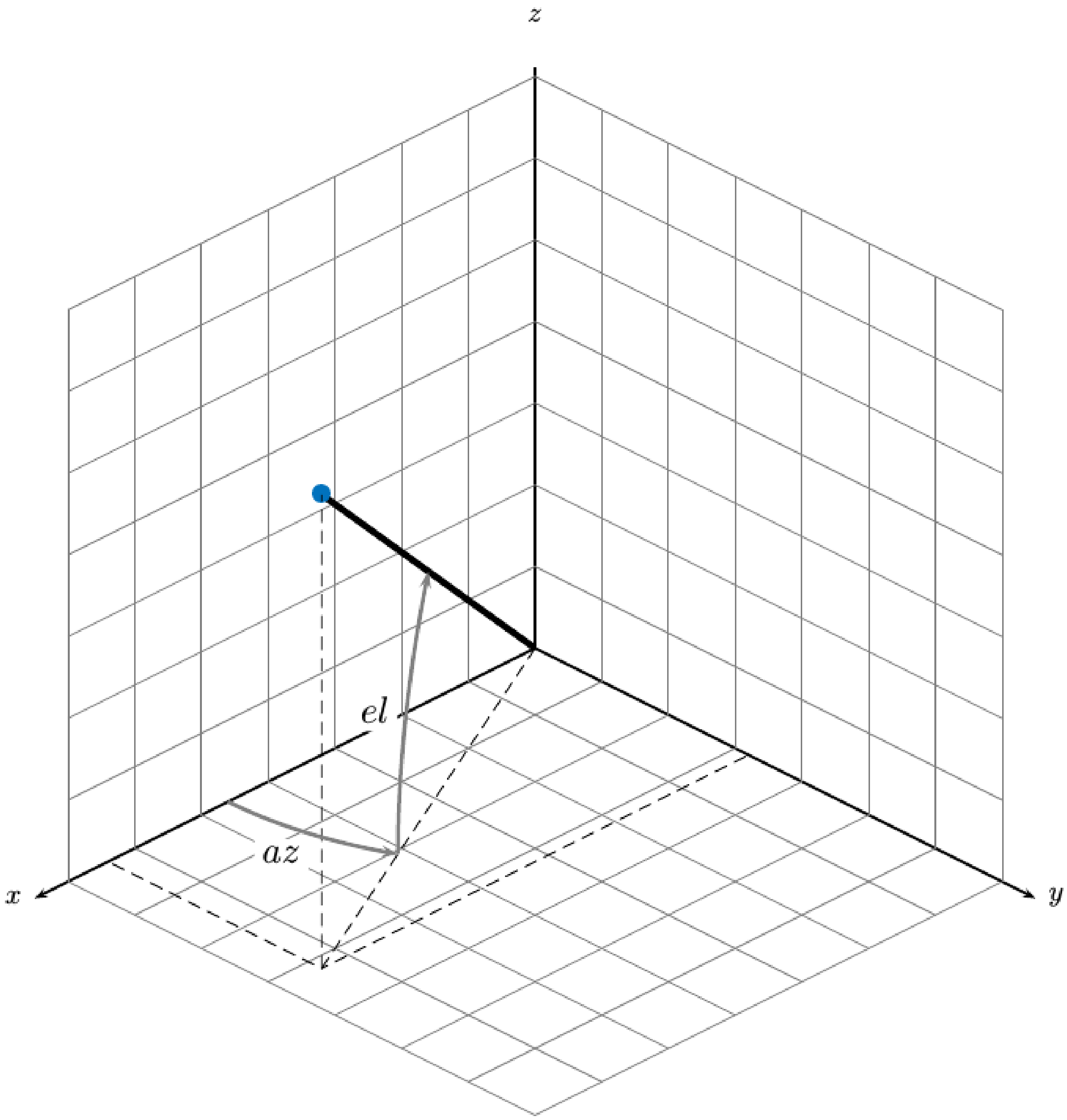
Frame	Measurement Jacobian
`'rectangular'`	Jacobian of the measurements `[x;y;z]` with respect to the state vector. The measurement vector is with respect to the local coordinate system. Coordinates are in meters.
`'spherical'`	Jacobian of the measurement vector `[az;el;r;rr]` with respect to the state vector. Measurement vector components specify the azimuth angle, elevation angle, range, and range rate of the object with respect to the local sensor coordinate system. Angle units are in degrees. Range units are in meters and range rate units are in meters/second.

More About

Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Automated Driving Toolbox.

The azimuth angle of a vector is the angle between the *x*-axis and its orthogonal projection onto the *xy* plane. The angle is positive in going from the *x* axis toward the *y* axis. Azimuth angles lie between –180 and 180 degrees. The elevation angle is the angle between the vector and its orthogonal projection onto the *xy*-plane. The angle is positive when going toward the positive *z*-axis from the *xy* plane.



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`cameas` | `cameasjac` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `constveljac` | `ctmeas` | `cvmeas` | `cvmeasjac`

Objects

`trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

cvmeas

Measurement function for constant velocity motion

Syntax

```
measurement = cvmeas(state)
measurement = cvmeas(state,frame)
measurement = cvmeas(state,frame,sensorpos)
measurement = cvmeas(state,frame,sensorpos,sensorvel)
measurement = cvmeas(state,frame,sensorpos,sensorvel,laxes)
measurement = cvmeas(state,measurementParameters)
```

Description

`measurement = cvmeas(state)` returns the measurement for a constant-velocity Kalman filter motion model in rectangular coordinates. The `state` argument specifies the current state of the tracking filter.

`measurement = cvmeas(state,frame)` also specifies the measurement coordinate system, `frame`.

`measurement = cvmeas(state,frame,sensorpos)` also specifies the sensor position, `sensorpos`.

`measurement = cvmeas(state,frame,sensorpos,sensorvel)` also specifies the sensor velocity, `sensorvel`.

`measurement = cvmeas(state,frame,sensorpos,sensorvel,laxes)` specifies the local sensor axes orientation, `laxes`.

`measurement = cvmeas(state,measurementParameters)` specifies the measurement parameters, `measurementParameters`.

Examples

Create Measurement from Constant-Velocity Object in Rectangular Frame

Define the state of an object in 2-D constant-velocity motion. The state is the position and velocity in both dimensions. The measurements are in rectangular coordinates.

```
state = [1;10;2;20];
measurement = cvmeas(state)
```

```
measurement = 3×1

     1
     2
     0
```

The z-component of the measurement is zero.

Create Measurement from Constant Velocity Object in Spherical Frame

Define the state of an object in 2-D constant-velocity motion. The state is the position and velocity in each spatial dimension. The measurements are in spherical coordinates.

```
state = [1;10;2;20];
measurement = cvmeas(state,'spherical')
```

```
measurement = 4×1

   63.4349
         0
    2.2361
   22.3607
```

The elevation of the measurement is zero and the range rate is positive. These results indicate that the object is moving away from the sensor.

Create Measurement from Constant-Velocity Object in Translated Spherical Frame

Define the state of an object in 2-D constant-velocity motion. The state consists of position and velocity in each spatial dimension. The measurements are in spherical coordinates with respect to a frame located at *(20;40;0)* meters.

```
state = [1;10;2;20];
measurement = cvmeas(state,'spherical',[20;40;0])
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -22.3607
```

The elevation of the measurement is zero and the range rate is negative. These results indicate that the object is moving toward the sensor.

Create Measurement from Constant-Velocity Object Using Measurement Parameters

Define the state of an object in 2-D constant-velocity motion. The state consists of position and velocity in each spatial dimension. The measurements are in spherical coordinates with respect to a frame located at *(20;40;0)* meters.

```
state2d = [1;10;2;20];
frame = 'spherical';
```

```
sensorpos = [20;40;0];
sensorvel = [0;5;0];
laxes = eye(3);
measurement = cvmeas(state2d,frame,sensorpos,sensorvel,laxes)
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -17.8885
```

The elevation of the measurement is zero and the range rate is negative. These results indicate that the object is moving toward the sensor.

Put the measurement parameters in a structure and use the alternative syntax.

```
measparm = struct('Frame',frame,'OriginPosition',sensorpos,'OriginVelocity',sensorvel, ...
    'Orientation',laxes);
measurement = cvmeas(state2d,measparm)
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -17.8885
```

Input Arguments

`state` — Kalman filter state vector

real-valued 2*N*-element vector

Kalman filter state vector for constant-velocity motion, specified as a real-valued 2*N*-element column vector where *N* is the number of spatial degrees of freedom of motion. The `state` is expected to be Cartesian state. For each spatial degree of motion, the state vector takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	`[x;vx]`
2-D	`[x;vx;y;vy]`
3-D	`[x;vx;y;vy;z;vz]`

For example, `x` represents the *x*-coordinate and `vx` represents the velocity in the *x*-direction. If the motion model is 1-D, values along the *y* and *z* axes are assumed to be zero. If the motion model is 2-D, values along the *z* axis are assumed to be zero. Position coordinates are in meters and velocity coordinates are in meters/sec.

Example: `[5;.1;0;-.2;-3;.05]`

Data Types: `single` | `double`

`frame` — Measurement output frame

`'rectangular'` (default) | `'spherical'`

Measurement output frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of *x*, *y*, and *z* Cartesian coordinates. When specified as `'spherical'`, a measurement consists of azimuth, elevation, range, and range rate.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in meters.

Data Types: `double`

`sensorvel` — Sensor velocity
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor velocity with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in m/s.

Data Types: `double`

`laxes` — Local sensor coordinate axes
`[1,0,0;0,1,0;0,0,1]` (default) | 3-by-3 orthogonal matrix

Local sensor coordinate axes, specified as a 3-by-3 orthogonal matrix. Each column specifies the direction of the local *x*-, *y*-, and *z*-axes, respectively, with respect to the navigation frame. That is, the matrix is the rotation matrix from the global frame to the sensor frame.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure | array of structure

Measurement parameters, specified as a structure or an array of structures. The fields of the structure are:

Field	Description	Example
`Frame`	Frame used to report measurements, specified as one of these values: • `'rectangular'` — Detections are reported in rectangular coordinates. • `'spherical'` — Detections are reported in spherical coordinates.	`'spherical'`
`OriginPosition`	Position offset of the origin of the frame relative to the parent frame, specified as an `[x y z]` real-valued vector.	`[0 0 0]`

Field	Description	Example
OriginVelocity	Velocity offset of the origin of the frame relative to the parent frame, specified as a [vx vy vz] real-valued vector.	[0 0 0]
Orientation	Frame rotation matrix, specified as a 3-by-3 real-valued orthonormal matrix.	[1 0 0; 0 1 0; 0 0 1]
HasAzimuth	Logical scalar indicating if azimuth is included in the measurement.	1
HasElevation	Logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if HasElevation is false, the reported measurements assume 0 degrees of elevation.	1
HasRange	Logical scalar indicating if range is included in the measurement.	1
HasVelocity	Logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if HasVelocity is false, the measurements are reported as [x y z]. If HasVelocity is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if Orientation performs a frame rotation from the parent coordinate frame to the child coordinate frame. When IsParentToChild is false, then Orientation performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

Data Types: struct

Output Arguments

`measurement` — Measurement vector
N-by-1 column vector

Measurement vector, returned as an *N*-by-1 column vector. The form of the measurement depends upon which syntax you use.

- When the syntax does not use the `measurementParameters` argument, the measurement vector is `[x,y,z]` when the `frame` input argument is set to `'rectangular'` and `[az;el;r;rr]` when the `frame` is set to `'spherical'`.
- When the syntax uses the `measurementParameters` argument, the size of the measurement vector depends on the values of the `frame`, `HasVelocity`, and `HasElevation` fields in the `measurementParameters` structure.

<table>
<tr><th>frame</th><th>measurement</th></tr>
<tr><td>'spherical'</td><td>Specifies the azimuth angle, az, elevation angle, el, range, r, and range rate, rr, of the object with respect to the local ego vehicle coordinate system. Positive values for range rate indicate that an object is moving away from the sensor.

Spherical measurements
<table>
<tr><th></th><th></th><th>HasElevation</th><th></th></tr>
<tr><td></td><td></td><td>false</td><td>true</td></tr>
<tr><td>HasVelocity</td><td>false</td><td>[az;r]</td><td>[az;el;r]</td></tr>
<tr><td></td><td>true</td><td>[az;r;rr]</td><td>[az;el;r;rr]</td></tr>
</table>
Angle units are in degrees, range units are in meters, and range rate units are in m/s.</td></tr>
<tr><td>'rectangular</td><td>Specifies the Cartesian position and velocity coordinates of the tracked object with respect to the ego vehicle coordinate system.

Rectangular measurements
<table>
<tr><td>HasVelocity</td><td>false</td><td>[x;y;y]</td></tr>
<tr><td></td><td>true</td><td>[x;y;z;vx;vy;vz]</td></tr>
</table>
Position units are in meters and velocity units are in m/s.</td></tr>
</table>

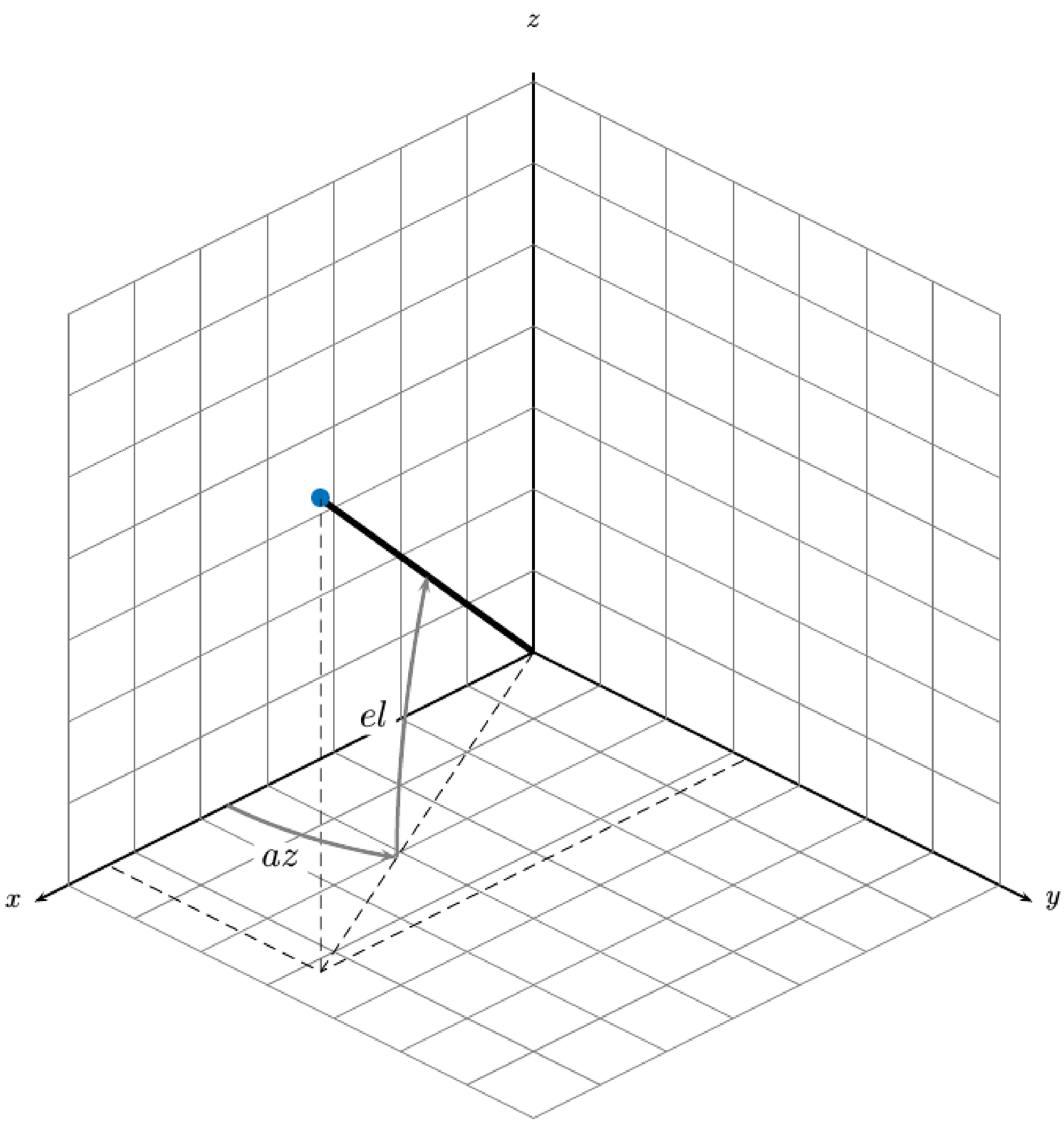
Data Types: `double`

More About

Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Automated Driving Toolbox.

The azimuth angle of a vector is the angle between the *x*-axis and its orthogonal projection onto the *xy* plane. The angle is positive in going from the *x* axis toward the *y* axis. Azimuth angles lie between –180 and 180 degrees. The elevation angle is the angle between the vector and its orthogonal projection onto the *xy*-plane. The angle is positive when going toward the positive *z*-axis from the *xy* plane.



Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`cameas` | `cameasjac` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `constveljac` | `ctmeas` | `ctmeasjac` | `cvmeasjac`

Objects
`trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

cvmeasjac

Jacobian of measurement function for constant velocity motion

Syntax

```
measurementjac = cvmeasjac(state)
measurementjac = cvmeasjac(state,frame)
measurementjac = cvmeasjac(state,frame,sensorpos)
measurementjac = cvmeasjac(state,frame,sensorpos,sensorvel)
measurementjac = cvmeasjac(state,frame,sensorpos,sensorvel,laxes)
measurementjac = cvmeasjac(state,measurementParameters)
```

Description

`measurementjac = cvmeasjac(state)` returns the measurement Jacobian for constant-velocity Kalman filter motion model in rectangular coordinates. `state` specifies the current state of the tracking filter.

`measurementjac = cvmeasjac(state,frame)` also specifies the measurement coordinate system, `frame`.

`measurementjac = cvmeasjac(state,frame,sensorpos)` also specifies the sensor position, `sensorpos`.

`measurementjac = cvmeasjac(state,frame,sensorpos,sensorvel)` also specifies the sensor velocity, `sensorvel`.

`measurementjac = cvmeasjac(state,frame,sensorpos,sensorvel,laxes)` also specifies the local sensor axes orientation, `laxes`.

`measurementjac = cvmeasjac(state,measurementParameters)` specifies the measurement parameters, `measurementParameters`.

Examples

Measurement Jacobian of Constant-Velocity Object in Rectangular Frame

Define the state of an object in 2-D constant-velocity motion. The state is the position and velocity in each spatial dimension. Construct the measurement Jacobian in rectangular coordinates.

```
state = [1;10;2;20];
jacobian = cvmeasjac(state)
```

```
jacobian = 3×4

     1     0     0     0
     0     0     1     0
     0     0     0     0
```

Measurement Jacobian of Constant-Velocity Motion in Spherical Frame

Define the state of an object in 2-D constant-velocity motion. The state is the position and velocity in each dimension. Compute the measurement Jacobian with respect to spherical coordinates.

```
state = [1;10;2;20];
measurementjac = cvmeasjac(state,'spherical')
```

```
measurementjac = 4×4

  -22.9183         0   11.4592         0
         0         0         0         0
    0.4472         0    0.8944         0
    0.0000    0.4472    0.0000    0.8944
```

Measurement Jacobian of Constant-Velocity Object in Translated Spherical Frame

Define the state of an object in 2-D constant-velocity motion. The state is the position and velocity in each spatial dimension. Compute the measurement Jacobian with respect to spherical coordinates centered at *(5;-20;0)* meters.

```
state = [1;10;2;20];
sensorpos = [5;-20;0];
measurementjac = cvmeasjac(state,'spherical',sensorpos)
```

```
measurementjac = 4×4

   -2.5210         0   -0.4584         0
         0         0         0         0
   -0.1789         0    0.9839         0
    0.5903   -0.1789    0.1073    0.9839
```

Create Measurement Jacobian for Constant-Velocity Object Using Measurement Parameters

Define the state of an object in 2-D constant-velocity motion. The state consists of position and velocity in each spatial dimension. The measurements are in spherical coordinates with respect to a frame located at *(20;40;0)* meters.

```
state2d = [1;10;2;20];
frame = 'spherical';
sensorpos = [20;40;0];
sensorvel = [0;5;0];
laxes = eye(3);
measurementjac = cvmeasjac(state2d,frame,sensorpos,sensorvel,laxes)
```

```
measurementjac = 4×4

    1.2062         0   -0.6031         0
         0         0         0         0
```

```
   -0.4472         0   -0.8944         0
    0.0471   -0.4472   -0.0235   -0.8944
```

Put the measurement parameters in a structure and use the alternative syntax.

```
measparm = struct('Frame',frame,'OriginPosition',sensorpos,'OriginVelocity',sensorvel, ...
    'Orientation',laxes);
measurementjac = cvmeasjac(state2d,measparm)
```

```
measurementjac = 4×4

    1.2062         0   -0.6031         0
         0         0         0         0
   -0.4472         0   -0.8944         0
    0.0471   -0.4472   -0.0235   -0.8944
```

Input Arguments

state — Kalman filter state vector

real-valued 2*N*-element vector

Kalman filter state vector for constant-velocity motion, specified as a real-valued 2*N*-element column vector where *N* is the number of spatial degrees of freedom of motion. The `state` is expected to be Cartesian state. For each spatial degree of motion, the state vector takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	`[x;vx]`
2-D	`[x;vx;y;vy]`
3-D	`[x;vx;y;vy;z;vz]`

For example, `x` represents the *x*-coordinate and `vx` represents the velocity in the *x*-direction. If the motion model is 1-D, values along the *y* and *z* axes are assumed to be zero. If the motion model is 2-D, values along the *z* axis are assumed to be zero. Position coordinates are in meters and velocity coordinates are in meters/sec.

Example: `[5;.1;0;-.2;-3;.05]`

Data Types: `single` | `double`

frame — Measurement output frame

`'rectangular'` (default) | `'spherical'`

Measurement output frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of *x*, *y*, and *z* Cartesian coordinates. When specified as `'spherical'`, a measurement consists of azimuth, elevation, range, and range rate.

Data Types: `char`

sensorpos — Sensor position

`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in meters.

Data Types: `double`

`sensorvel` — Sensor velocity
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor velocity with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in m/s.

Data Types: `double`

`laxes` — Local sensor coordinate axes
`[1,0,0;0,1,0;0,0,1]` (default) | 3-by-3 orthogonal matrix

Local sensor coordinate axes, specified as a 3-by-3 orthogonal matrix. Each column specifies the direction of the local *x*-, *y*-, and *z*-axes, respectively, with respect to the navigation frame. That is, the matrix is the rotation matrix from the global frame to the sensor frame.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure | array of structure

Measurement parameters, specified as a structure or an array of structures. The fields of the structure are:

Field	Description	Example
`Frame`	Frame used to report measurements, specified as one of these values: • `'rectangular'` — Detections are reported in rectangular coordinates. • `'spherical'` — Detections are reported in spherical coordinates.	`'spherical'`
`OriginPosition`	Position offset of the origin of the frame relative to the parent frame, specified as an `[x y z]` real-valued vector.	`[0 0 0]`
`OriginVelocity`	Velocity offset of the origin of the frame relative to the parent frame, specified as a `[vx vy vz]` real-valued vector.	`[0 0 0]`
`Orientation`	Frame rotation matrix, specified as a 3-by-3 real-valued orthonormal matrix.	`[1 0 0; 0 1 0; 0 0 1]`
`HasAzimuth`	Logical scalar indicating if azimuth is included in the measurement.	`1`

Field	Description	Example
HasElevation	Logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if HasElevation is false, the reported measurements assume 0 degrees of elevation.	1
HasRange	Logical scalar indicating if range is included in the measurement.	1
HasVelocity	Logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if HasVelocity is false, the measurements are reported as [x y z]. If HasVelocity is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if Orientation performs a frame rotation from the parent coordinate frame to the child coordinate frame. When IsParentToChild is false, then Orientation performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

Data Types: struct

Output Arguments

measurementjac — Measurement Jacobian

real-valued 3-by-*N* matrix | real-valued 4-by-*N* matrix

Measurement Jacobian, specified as a real-valued 3-by-*N* or 4-by-*N* matrix. *N* is the dimension of the state vector. The first dimension and meaning depend on value of the frame argument.

Frame	Measurement Jacobian
'rectangular'	Jacobian of the measurements [x;y;z] with respect to the state vector. The measurement vector is with respect to the local coordinate system. Coordinates are in meters.

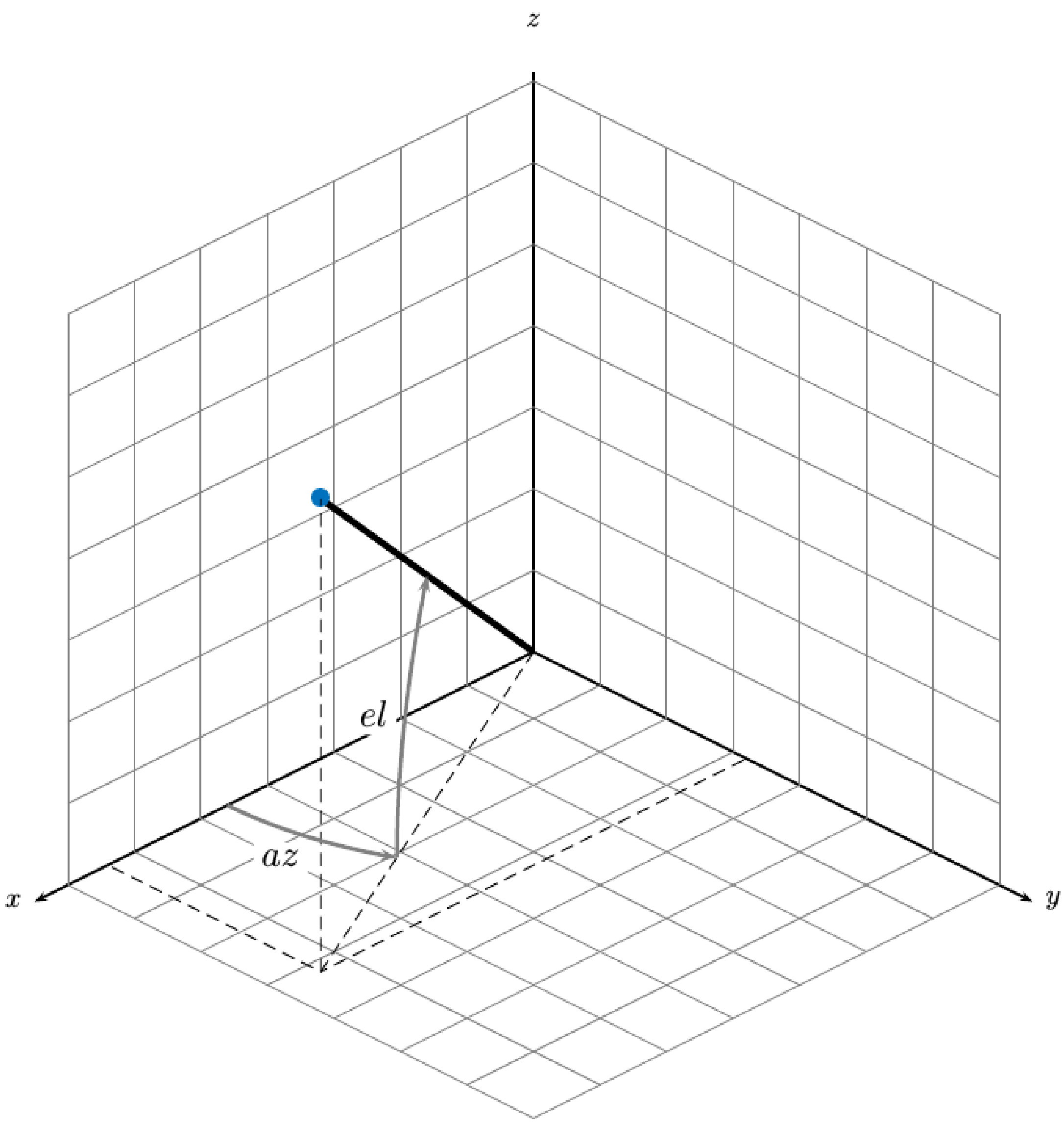
Frame	Measurement Jacobian
'spherical'	Jacobian of the measurement vector [az;el;r;rr] with respect to the state vector. Measurement vector components specify the azimuth angle, elevation angle, range, and range rate of the object with respect to the local sensor coordinate system. Angle units are in degrees. Range units are in meters and range rate units are in meters/second.

More About

Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Automated Driving Toolbox.

The azimuth angle of a vector is the angle between the *x*-axis and its orthogonal projection onto the *xy* plane. The angle is positive in going from the *x* axis toward the *y* axis. Azimuth angles lie between –180 and 180 degrees. The elevation angle is the angle between the vector and its orthogonal projection onto the *xy*-plane. The angle is positive when going toward the positive *z*-axis from the *xy* plane.



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`cameas` | `cameasjac` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `constveljac` | `ctmeas` | `ctmeasjac` | `cvmeas`

Objects

`trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

estimateMonoCameraParameters

Estimate extrinsic monocular camera parameters using checkerboard

Syntax

```
[pitch,yaw,roll,height] = estimateMonoCameraParameters(intrinsics,
imagePoints,worldPoints,patternOriginHeight)
[pitch,yaw,roll,height] = estimateMonoCameraParameters( ___ ,Name,Value)
```

Description

`[pitch,yaw,roll,height] = estimateMonoCameraParameters(intrinsics, imagePoints,worldPoints,patternOriginHeight)` estimates the extrinsic parameters of a monocular camera by using the intrinsic parameters of the camera and a checkerboard calibration pattern. The returned extrinsic parameters define the yaw, pitch, and roll rotation angles between the camera coordinate system (Computer Vision Toolbox) and vehicle coordinate system on page 3-94 axes. The function also returns the height of the camera above the ground. Specify the intrinsic parameters, the image and world coordinates of the checkerboard corner points, and the height of the checkerboard pattern's origin above the ground.

By default, the function assumes that the camera is facing forward and that the checkerboard pattern is parallel with the ground. For all possible camera and checkerboard placements, see “Calibrate a Monocular Camera”.

`[pitch,yaw,roll,height] = estimateMonoCameraParameters( ___ ,Name,Value)` specifies options using one or more name-value pairs, in addition to the inputs and outputs from the previous syntax. For example, you can specify the orientation or position of the checkerboard pattern.

Examples

Configure Monocular Camera Using Checkerboard Pattern

Configure a monocular fisheye camera by removing lens distortion and then estimating the camera's extrinsic parameters. Use an image of a checkerboard as the calibration pattern. For a more detailed look at how to configure a monocular camera that has a fisheye lens, see the “Configure Monocular Fisheye Camera” example.

Load the intrinsic parameters of a monocular camera that has a fisheye lens. `intrinsics` is a `fisheyeIntrinsics` (Computer Vision Toolbox) object.

```
ld = load('fisheyeCameraIntrinsics');
intrinsics = ld.intrinsics;
```

Load an image of a checkerboard pattern that is placed flat on the ground. This image is for illustrative purposes and was not taken from a camera mounted to the vehicle. In a camera mounted to the vehicle, the *X*-axis of the pattern points to the right of the vehicle, and the *Y*-axis of the pattern points to the camera. Display the image.

```
imageFileName = fullfile(toolboxdir('driving'),'drivingdata','checkerboard.png');
I = imread(imageFileName);
imshow(I)
```

Detect the coordinates of the checkerboard corners in the image.

```
[imagePoints,boardSize] = detectCheckerboardPoints(I);
```

Generate the corresponding world coordinates of the corners.

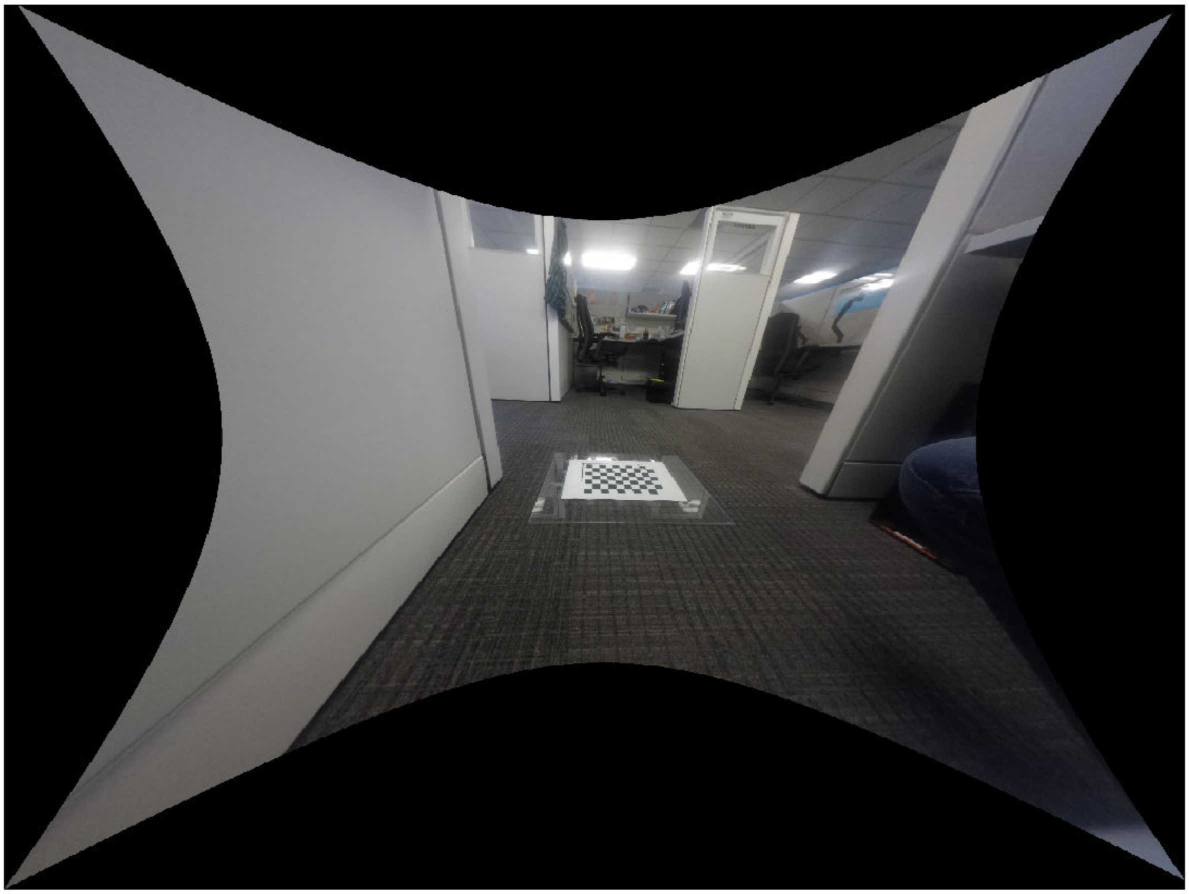
```
squareSize = 0.029; % Square size in meters
worldPoints = generateCheckerboardPoints(boardSize,squareSize);
```

Estimate the extrinsic parameters required to configure the `monoCamera` object. Because the checkerboard pattern is directly on the ground, set the height of the pattern's origin to 0.

```
patternOriginHeight = 0;
[pitch,yaw,roll,height] = estimateMonoCameraParameters(intrinsics, ...
                                imagePoints,worldPoints,patternOriginHeight);
```

Because `monoCamera` does not accept `fisheyeIntrinsics` objects, remove distortion from the image and compute new intrinsic parameters from the undistorted image. `camIntrinsics` is an cameraIntrinsics (Computer Vision Toolbox) object. Display the image to confirm distortion is removed.

```
[undistortedI,camIntrinsics] = undistortFisheyeImage(I,intrinsics,'Output','full');
imshow(undistortedI)
```



Configure the monocular camera using the estimated parameters.

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

```
monoCam = 
  monoCamera with properties:

        Intrinsics: [1x1 cameraIntrinsics]
        WorldUnits: 'meters'
            Height: 0.4447
             Pitch: 21.8459
               Yaw: -3.6130
              Roll: -3.1707
    SensorLocation: [0 0]
```

Input Arguments

intrinsics — Intrinsic camera parameters
cameraIntrinsics object | fisheyeIntrinsics object

Intrinsic camera parameters, specified as a cameraIntrinsics or fisheyeIntrinsics object.

Checkerboard pattern images produced by these cameras can include lens distortion, which can affect the accuracy of corner point detections. To remove lens distortion and compute new intrinsic parameters, use these functions:

- For cameraIntrinsics objects, use undistortImage.
- For fisheyeIntrinsics objects, use undistortFisheyeImage.

imagePoints — Image coordinates of checkerboard corner points
M-by-2 matrix

Image coordinates of checkerboard corner points, specified as an *M*-by-2 matrix of *M* number of [*x y*] vectors. These points must come from an image captured by a monocular camera. To detect these points in an image, use the detectCheckerboardPoints function.

estimateMonoCameraParameters assumes that all points in worldPoints are in the (X_P, Y_P) plane and that *M* is greater than or equal to 4. To specify the height of the (X_P, Y_P) plane above the ground, use patternOriginHeight.

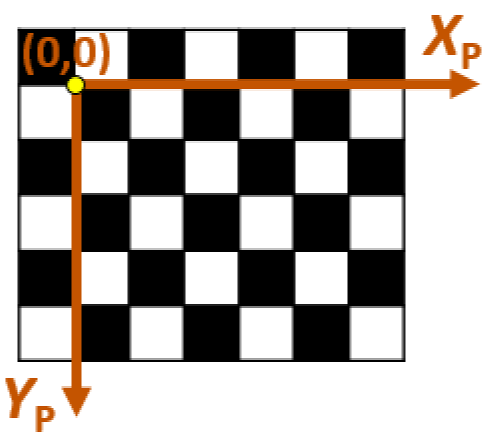
Data Types: single | double

worldPoints — World coordinates of corner points in checkerboard
M-by-2 matrix

World coordinates of the corner points in the checkerboard, specified as an *M*-by-2 matrix of *M* number of [*x y*] vectors.

estimateMonoCameraParameters assumes that all points in worldPoints are in the (X_P, Y_P) plane and that *M* is greater than or equal to 4. To specify the height of the (X_P, Y_P) plane above the ground, use patternOriginHeight.

Point (0,0) corresponds to the bottom-right corner of the top-left square of the checkerboard.



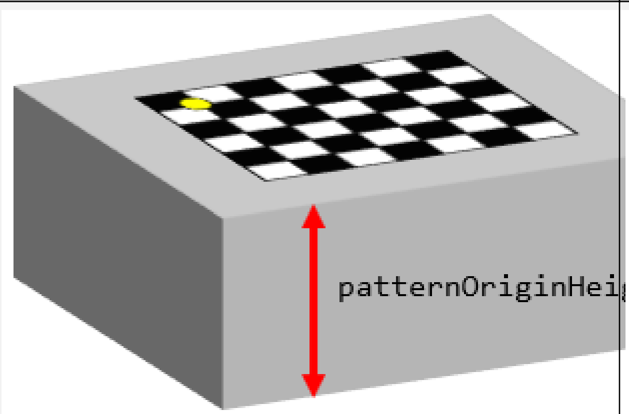
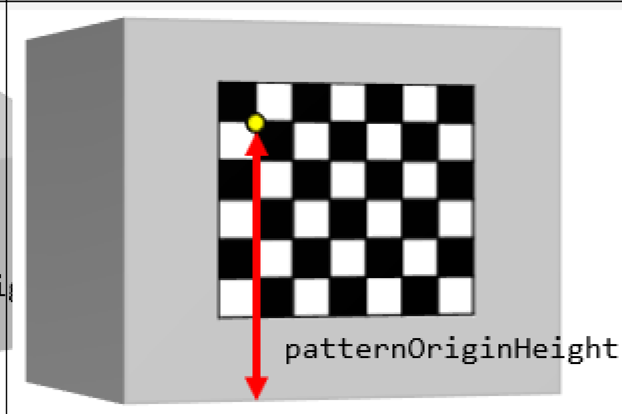
Data Types: single | double

patternOriginHeight — Height of checkerboard pattern's origin

nonnegative real scalar

Height of the checkerboard pattern's origin above the ground, specified as a nonnegative real scalar. The origin is the bottom-right corner of the top-left square of the checkerboard.

The measurement of `patternOriginHeight` depends on the orientation of the checkerboard pattern, as shown in these diagrams.

Horizontal Orientation	Vertical Orientation
	

To specify the pattern orientation, use the `'PatternOrientation'` name-value pair. If you set `'PatternOrientation'` to `'horizontal'` (default), and the pattern is on the ground, then set `patternOriginHeight` to `0`.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'PatternOrientation','vertical','PatternPosition','right'`

PatternOrientation — Orientation of checkerboard pattern

`'horizontal'` (default) | `'vertical'`

Orientation of the checkerboard pattern relative to the ground, specified as the comma-separated pair consisting of `'PatternOrientation'` and one of the following:

- `'horizontal'` — Checkerboard pattern is parallel to the ground.
- `'vertical'` — Checkerboard pattern is perpendicular to the ground.

PatternPosition — Position of checkerboard pattern

`'front'` (default) | `'back'` | `'left'` | `'right'`

Position of the checkerboard pattern relative to the ground, specified as the comma-separated pair consisting of `'PatternPosition'` and one of the following:

- `'front'` — Checkerboard pattern is in front of the vehicle.

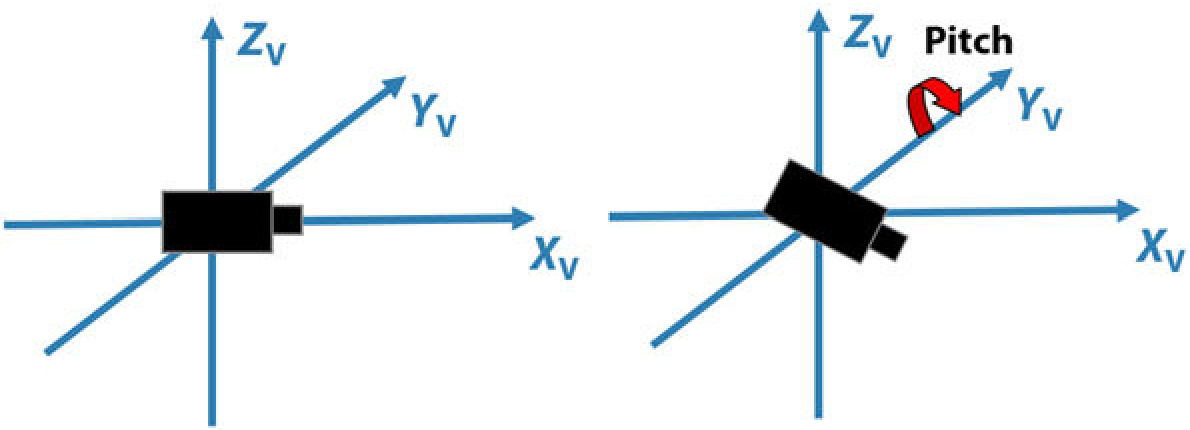
- `'back'` — Checkerboard pattern is behind the vehicle.
- `'left'` — Checkerboard pattern is to the left of the vehicle.
- `'right'` — Checkerboard pattern is to the right of the vehicle.

Output Arguments

`pitch` — Pitch angle

real scalar

Pitch angle between the horizontal plane of the vehicle and the optical axis of the camera, returned as a real scalar in degrees. `pitch` uses the ISO convention for rotation, with a clockwise positive angle direction when looking in the positive direction of the vehicle's Y_V-axis.

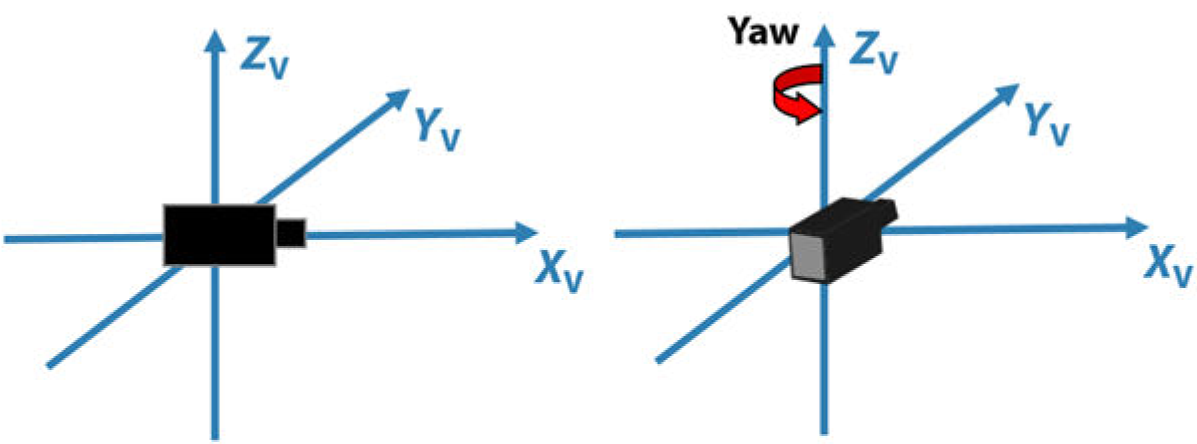


For more details, see "Angle Directions" on page 3-95.

`yaw` — Yaw angle

real scalar

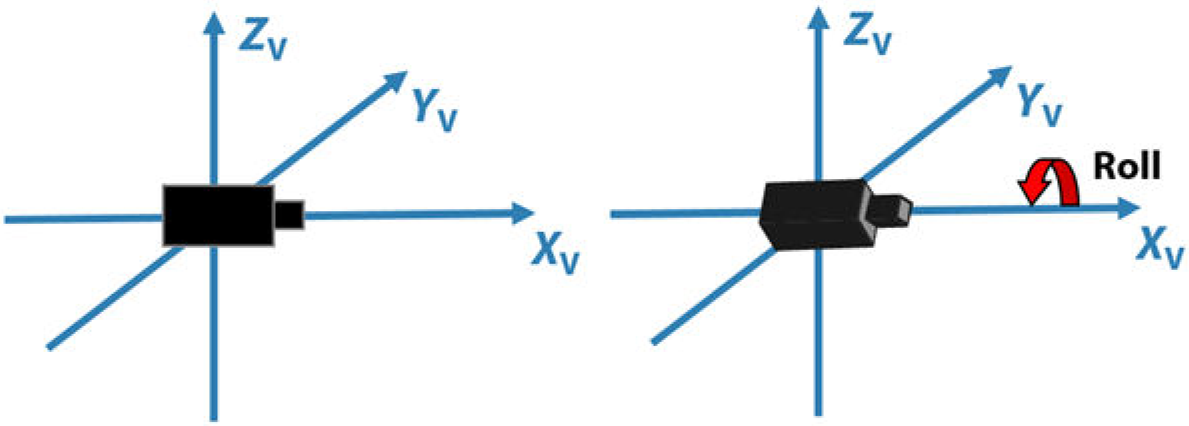
Yaw angle between the X_V-axis of the vehicle and the optical axis of the camera, returned as a real scalar in degrees. `yaw` uses the ISO convention for rotation, with a clockwise positive angle direction when looking in the positive direction of the vehicle's Z_V-axis.



For more details, see "Angle Directions" on page 3-95.

`roll` — Roll angle
real scalar

Roll angle of the camera around its optical axis, returned as a real scalar in degrees. `roll` uses the ISO convention for rotation, with a clockwise positive angle direction when looking in the positive direction of the vehicle's X_V-axis.



For more details, see "Angle Directions" on page 3-95.

`height` — Perpendicular height from ground to camera
nonnegative real scalar

Perpendicular height from the ground to the focal point of the camera, returned as a nonnegative real scalar in world units, such as meters.

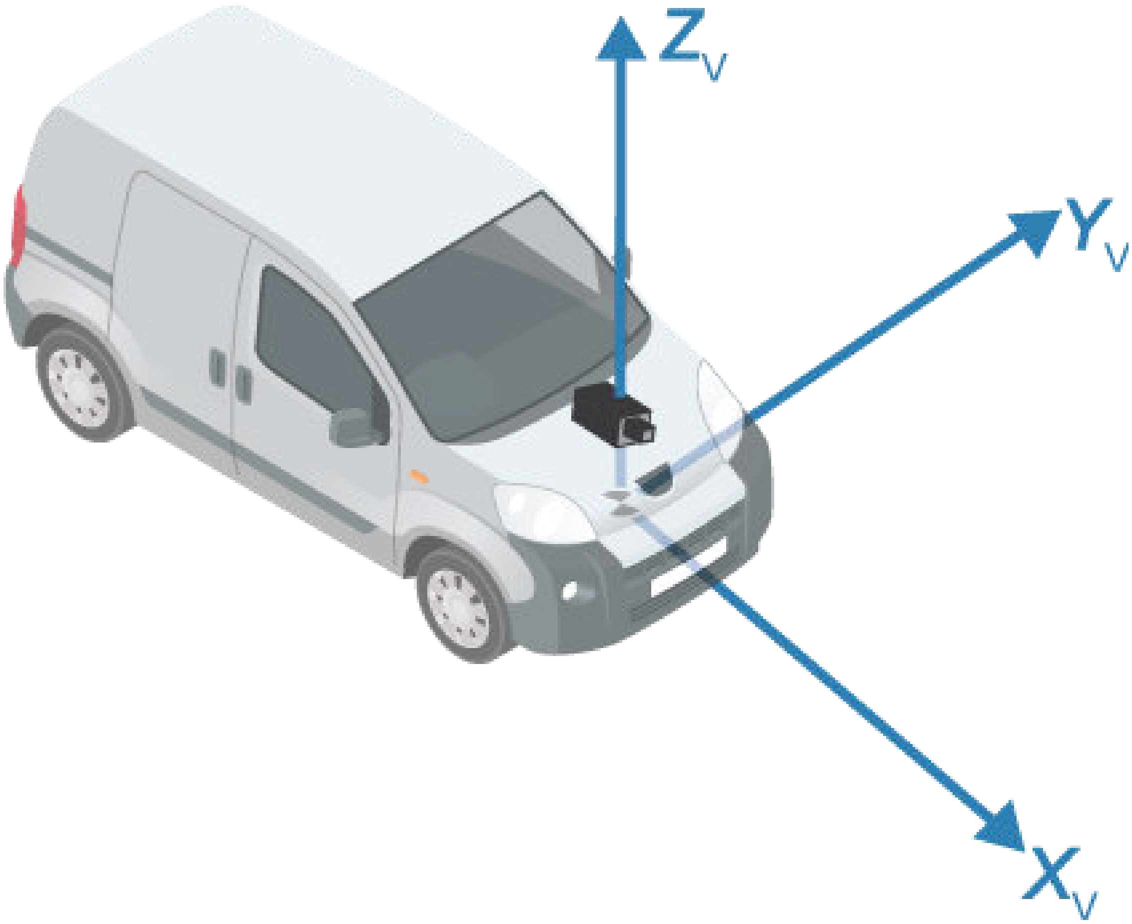
More About

Vehicle Coordinate System

In the vehicle coordinate system (X_V, Y_V, Z_V) defined by a `monoCamera` object:

- 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.

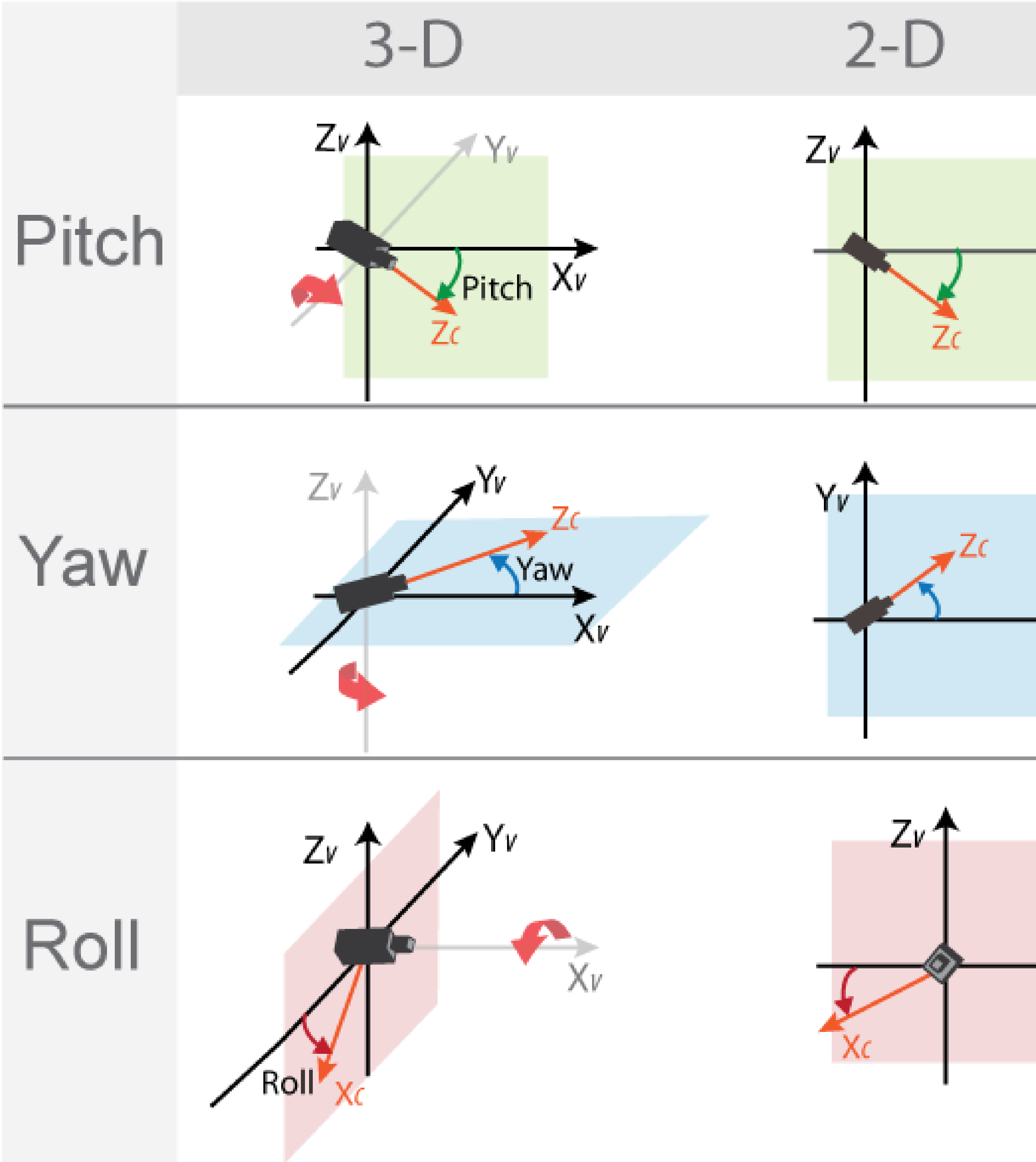
By default, the origin of this coordinate system is on the road surface, directly below the camera center (focal point of camera).



To obtain more reliable results from `estimateMonoCameraParameters`, the checkerboard pattern must be placed in precise locations relative to this coordinate system. For more details, see “Calibrate a Monocular Camera”.

Angle Directions

The monocular camera sensor uses clockwise positive angle directions when looking in the positive direction of the *Z*-, *Y*-, and *X*-axes, respectively.



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

See Also

Apps
Camera Calibrator

Functions
`detectCheckerboardPoints` | `estimateCameraParameters` | `estimateFisheyeParameters` | `extrinsics` | `generateCheckerboardPoints`

Objects
`cameraIntrinsics` | `fisheyeIntrinsics` | `monoCamera`

Topics
"Calibrate a Monocular Camera"
"Configure Monocular Fisheye Camera"
"Coordinate Systems in Automated Driving Toolbox"

Introduced in R2018b

evaluateLaneBoundaries

Evaluate lane boundary models against ground truth

Syntax

```
numMatches = evaluateLaneBoundaries(boundaries,worldGroundTruthPoints,
threshold)
[numMatches,numMissed,numFalsePositives] = evaluateLaneBoundaries( ___ )
[ ___ ] = evaluateLaneBoundaries( ___ ,xWorld)

[ ___ ] = evaluateLaneBoundaries(boundaries,groundTruthBoundaries,threshold)
[ ___ ,assignments] = evaluateLaneBoundaries( ___ )
```

Description

`numMatches = evaluateLaneBoundaries(boundaries,worldGroundTruthPoints, threshold)` returns the total number of lane boundary matches (true positives) within the lateral distance `threshold` by comparing the input boundary models, `boundaries`, against ground truth data.

`[numMatches,numMissed,numFalsePositives] = evaluateLaneBoundaries( ___ )` also returns the total number of misses (false negatives) and false positives, using the previous inputs.

`[ ___ ] = evaluateLaneBoundaries( ___ ,xWorld)` specifies the *x*-axis points at which to perform the comparisons. Points specified in `worldGroundTruthPoints` are linearly interpolated at the given *x*-axis locations.

`[ ___ ] = evaluateLaneBoundaries(boundaries,groundTruthBoundaries,threshold)` compares the boundaries against ground truth models that are specified in an array of lane boundary objects or a cell array of arrays.

`[ ___ ,assignments] = evaluateLaneBoundaries( ___ )` also returns the assignment indices that are specified in `groundTruthBoundaries`. Each boundary is matched to the corresponding class assignment in `groundTruthBoundaries`. The `k`th boundary in `boundaries` is matched to the `assignments(k)` element of `worldGroundTruthPoints`. Zero indicates a false positive (no match found).

Examples

Compare Lane Boundary Models

Create a set of ground truth points, add noise to simulate actual lane boundary points, and compare the simulated data to the model.

Create a set of points representing ground truth by using parabolic parameters.

```
parabolaParams1 = [-0.001 0.01 0.5];
parabolaParams2 = [0.001 0.02 0.52];
x = (0:0.1:20)';
```

```
y1 = polyval(parabolaParams1,x);
y2 = polyval(parabolaParams1,x);
```

Add noise relative to the offset parameter.

```
y1 = y1 + 0.10*parabolaParams1(3)*(rand(length(y1),1)-0.5);
y2 = y2 + 0.10*parabolaParams2(3)*(rand(length(y2),1)-0.5);
```

Create a set of test boundary models.

```
testlbs = parabolicLaneBoundary([-0.002 0.01 0.5;
                                 -0.001 0.02 0.45;
                                 -0.001 0.01 0.5;
                                  0.000 0.02 0.52;
                                 -0.001 0.01 0.51]);
```

Compare the boundary models to the ground truth points. Calculate the precision and sensitivity of the models based on the number of matches, misses, and false positives.

```
threshold = 0.1;
[numMatches,numMisses,numFalsePositives,~] = ...
        evaluateLaneBoundaries(testlbs,{[x y1],[x y2]},threshold);

disp('Precision:');

Precision:

disp(numMatches/(numMatches+numFalsePositives));

    0.4000

disp('Sensitivity/Recall:');

Sensitivity/Recall:

disp(numMatches/(numMatches+numMisses));

     1
```

Input Arguments

`worldGroundTruthPoints` — Ground truth points of lane boundaries

`[x y]` array | cell array of `[x y]` arrays

Ground truth points of lane boundaries, specified as an `[x y]` array or cell array of `[x y]` arrays. The *x*-axis points must be unique and in the same coordinate system as the boundary models. A lane boundary must contain at least two points, but for a robust comparison, four or more points are recommended. Each element of the cell array represents a separate lane boundary.

`threshold` — Maximum lateral distance from ground truth

real scalar

Maximum lateral distance between a model and ground truth point in order for that point to be considered a valid match (true positive), specified as a real scalar.

`boundaries` — Lane boundary models

array of `parabolicLaneBoundary` objects | array of `cubicLaneBoundary` objects

Lane boundary models, specified as an array of `parabolicLaneBoundary` objects or `cubicLaneBoundary` objects. Lane boundary models contain the following properties:

- `Parameters` — A vector corresponding to the coefficients of the boundary model. The size of the vector depends on the degree of polynomial for the model.

Lane Boundary Object	Parameters
`parabolicLaneBoundary`	`[A B C]`, corresponding to coefficients of a second-degree polynomial equation of the form $y = Ax^2 + Bx + C$
`cubicLaneBoundary`	`[A B C D]`, corresponding to coefficients of a third-degree polynomial equation of the form $y = Ax^3 + Bx^2 + Cx + D$

- `BoundaryType` — A `LaneBoundaryType` enumeration of supported lane boundaries:
  - `Unmarked`
  - `Solid`
  - `Dashed`
  - `BottsDots`
  - `DoubleSolid`

  Specify a lane boundary type as `LaneBoundaryType.`*`BoundaryType`*. For example:

  ```
  LaneBoundaryType.BottsDots
  ```
- `Strength` — The ratio of the number of unique *x*-axis locations on the boundary to the total number of points along the line based on the `XExtent` property.
- `XExtent` — A two-element vector describing the minimum and maximum *x*-axis locations for the boundary points.

`xWorld` — *x*-axis locations of boundary
real-valued vector

x-axis locations of boundary, specified as a real-valued vector. Points in `worldGroundTruthPoints` are linearly interpolated at the given *x*-axis locations. Boundaries outside of these locations are excluded and count as false negatives.

`groundTruthBoundaries` — Ground truth boundary models
array of `parabolicLaneBoundary` or `cubicLaneBoundary` objects | cell array of `parabolicLaneBoundary` or `cubicLaneBoundary` arrays

Ground truth boundary models, specified as an array of `parabolicLaneBoundary` or `cubicLaneBoundary` objects or cell array of `parabolicLaneBoundary` or `cubicLaneBoundary` arrays.

Output Arguments

`numMatches` — Number of matches (true positives)
real scalar

Number of matches (true positives), returned as a real scalar.

`numMissed` — Number of misses (false negatives)
real scalar

Number of misses (false negatives), returned as a real scalar.

`numFalsePositives` — Number of false positives
real scalar

Number of false positives, returned as a real scalar.

`assignments` — Assignment indices for ground truth boundaries
cell array of real-valued arrays

Assignment indices for ground truth boundaries, returned as a cell array of real-valued arrays. Each boundary is matched to the corresponding assignment in `groundTruthBoundaries`. The `k`th boundary in `boundaries` is matched to the `assignments(k)` element of `worldGroundTruthPoints`. Zero indicates a false positive (no match found).

See Also

Functions
`findCubicLaneBoundaries` | `findParabolicLaneBoundaries`

Objects
`cubicLaneBoundary` | `parabolicLaneBoundary`

Apps
Ground Truth Labeler

Introduced in R2017a

findCubicLaneBoundaries

Find boundaries using cubic model

Syntax

```
boundaries = findCubicLaneBoundaries(xyBoundaryPoints,approxBoundaryWidth)
[boundaries,boundaryPoints] = findCubicLaneBoundaries(xyBoundaryPoints,
approxBoundaryWidth)
[ ___ ] = findCubicLaneBoundaries( ___ ,Name,Value)
```

Description

`boundaries = findCubicLaneBoundaries(xyBoundaryPoints,approxBoundaryWidth)` uses the random sample consensus (RANSAC) algorithm to find cubic lane boundary models that fit a set of boundary points and an approximate width. Each model in the returned array of `cubicLaneBoundary` objects contains the `[A B C D]` coefficients of its third-degree polynomial equation and the strength of the boundary estimate.

`[boundaries,boundaryPoints] = findCubicLaneBoundaries(xyBoundaryPoints, approxBoundaryWidth)` also returns a cell array of inlier boundary points for each boundary model found, using the previous input arguments.

`[ ___ ] = findCubicLaneBoundaries( ___ ,Name,Value)` uses options specified by one or more `Name,Value` pair arguments, with any of the preceding syntaxes.

Examples

Find Cubic Lane Boundaries in Bird's-Eye-View Image

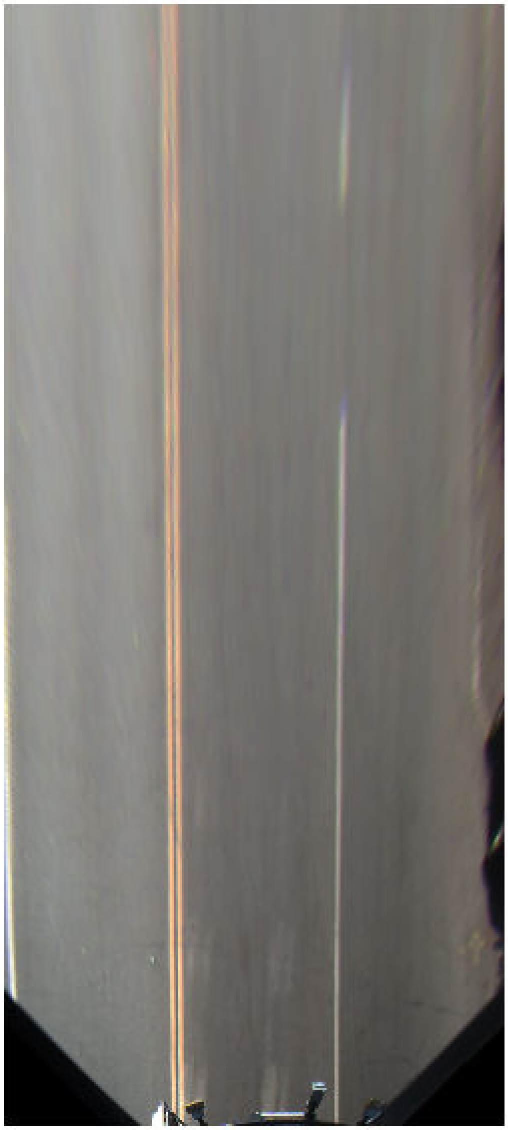
Find lanes in an image by using cubic lane boundary models. Overlay the identified lanes on the original image and on a bird's-eye-view transformation of the image.

Load an image of a road with lanes. The image was obtained from a camera sensor mounted on the front of a vehicle.

```
I = imread('road.png');
```

Transform the image into a bird's-eye-view image by using a preconfigured sensor object. This object models the sensor that captured the original image.

```
bevSensor = load('birdsEyeConfig');
birdsEyeImage = transformImage(bevSensor.birdsEyeConfig,I);
imshow(birdsEyeImage)
```

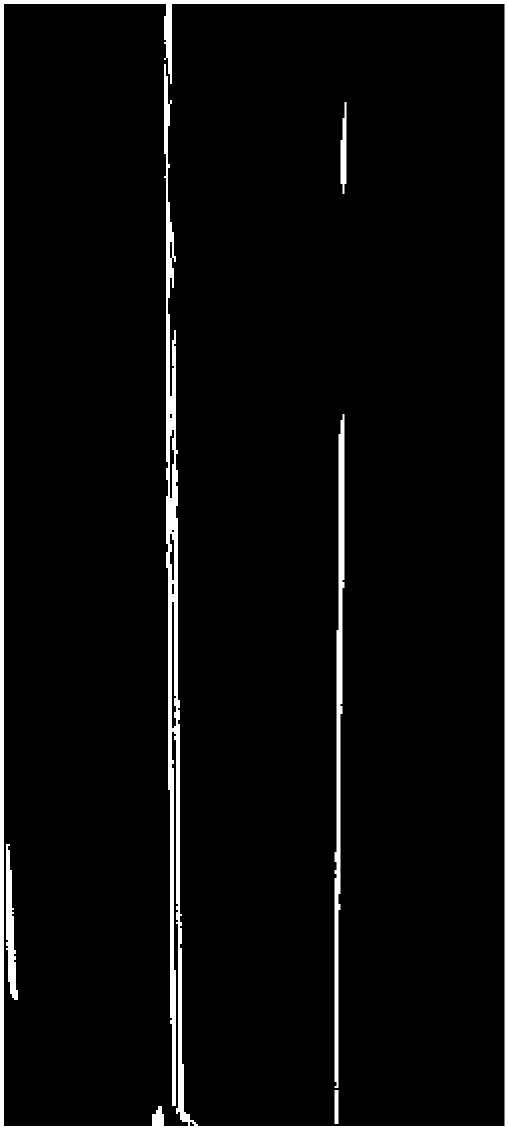


Set the approximate lane marker width in world units (meters).

```
approxBoundaryWidth = 0.25;
```

Detect lane features and display them as a black-and-white image.

```
birdsEyeBW = segmentLaneMarkerRidge(rgb2gray(birdsEyeImage), ...
    bevSensor.birdsEyeConfig,approxBoundaryWidth);
imshow(birdsEyeBW)
```



Obtain lane candidate points in world coordinates.

```
[imageX,imageY]  = find(birdsEyeBW);
xyBoundaryPoints = imageToVehicle(bevSensor.birdsEyeConfig,[imageY,imageX]);
```

Find lane boundaries in the image by using the `findCubicLaneBoundaries` function. By default, the function returns a maximum of two lane boundaries. The boundaries are stored in an array of `cubicLaneBoundary` objects.

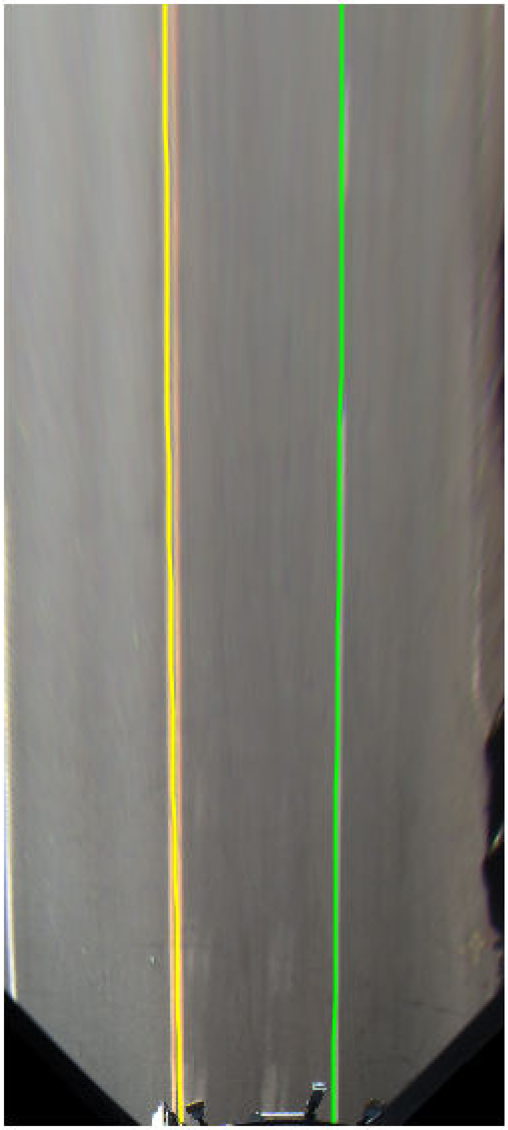
```
boundaries = findCubicLaneBoundaries(xyBoundaryPoints,approxBoundaryWidth);
```

Use `insertLaneBoundary` to overlay the lanes on the original image. The `XPoints` vector represents the lane points, in meters, that are within range of the ego vehicle's sensor. Specify the lanes in different colors. By default, lanes are yellow.

```
XPoints = 3:30;

figure
sensor = bevSensor.birdsEyeConfig.Sensor;
lanesI = insertLaneBoundary(I,boundaries(1),sensor,XPoints);
lanesI = insertLaneBoundary(lanesI,boundaries(2),sensor,XPoints,'Color','green');
imshow(lanesI)
```

View the lanes in the bird's-eye-view image.

```
figure
BEconfig = bevSensor.birdsEyeConfig;
lanesBEI = insertLaneBoundary(birdsEyeImage,boundaries(1),BEconfig,XPoints);
lanesBEI = insertLaneBoundary(lanesBEI,boundaries(2),BEconfig,XPoints,'Color','green');
imshow(lanesBEI)
```



Input Arguments

`xyBoundaryPoints` — Candidate boundary points
`[x y]` vector

Candidate boundary points, specified as an `[x y]` vector in vehicle coordinates. To obtain the vehicle coordinates for points in a `birdsEyeView` image, use the `imageToVehicle` function to convert the bird's-eye-view image coordinates to vehicle coordinates.

`approxBoundaryWidth` — Approximate boundary width
real scalar

Approximate boundary width, specified as a real scalar in world units. The width is a horizontal *y*-axis measurement.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MaxSamplingAttempts',200`

`MaxNumBoundaries` — Maximum number of lane boundaries
`2` (default) | positive integer

Maximum number of lane boundaries that the function attempts to find, specified as the comma-separated pair consisting of `'MaxNumBoundaries'` and a positive integer.

`ValidateBoundaryFcn` — Function to validate boundary model
function handle

Function to validate the boundary model, specified as the comma-separated pair consisting of `'ValidateBoundaryFcn'` and a function handle. The specified function returns logical `1` (true) if the boundary model is accepted and logical `0` (false) otherwise. Use this function to reject invalid boundaries. The function must be of the form:

```
isValid = validateBoundaryFcn(parameters)
```

`parameters` is a vector corresponding to the three parabolic parameters.

The default validation function always returns `1` (true).

`MaxSamplingAttempts` — Maximum number of sampling attempts
`100` (default) | positive integer

Maximum number of attempts to find a sample of points that yields a valid cubic boundary, specified as the comma-separated pair consisting of `'MaxSamplingAttempts'` and a function handle. `findCubicLaneBoundaries` uses the `fitPolynomialRANSAC` function to sample from the set of boundary points and fit a cubic boundary line.

Output Arguments

`boundaries` — Lane boundary models
array of `cubicLaneBoundary` objects

Lane boundary models, returned as an array of `cubicLaneBoundary` objects. Lane boundary objects contain the following properties:

- `Parameters` — A four-element vector, `[A B C D]`, that corresponds to the four coefficients of a third-degree polynomial equation in general form: $y = Ax^3 + Bx^2 + Cx + D$.
- `BoundaryType` — A `LaneBoundaryType` of supported lane boundaries. The supported lane boundary types are:

- `Unmarked`
- `Solid`
- `Dashed`
- `BottsDots`
- `DoubleSolid`

Specify a lane boundary type as `LaneBoundaryType.`*`BoundaryType`*. For example:

```
LaneBoundaryType.BottsDots
```

- `Strength` — A ratio of the number of unique *x*-axis locations on the boundary to the total number of points along the line, based on the `XExtent` property.
- `XExtent` — A two-element vector describing the minimum and maximum *x*-axis locations for the boundary points.

`boundaryPoints` — Inlier boundary points
cell array of `[x y]` values

Inlier boundary points, returned as a cell array of `[x y]` values. Each element of the cell array corresponds to the same element in the array of `cubicLaneBoundary` objects.

Tips

- To fit a single boundary model to a double lane marker, set the `approxBoundaryWidth` argument to be large enough to include the width spanning both lane markers.

Algorithms

- This function uses `fitPolynomialRANSAC` to find cubic models. Because this algorithm uses random sampling, the output can vary between runs.
- The `maxDistance` parameter of `fitPolynomialRANSAC` is set to half the width specified in the `approxBoundaryWidth` argument. Points are considered inliers if they are within the boundary width. The function obtains the final boundary model using a least-squares fit on the inlier points.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`birdsEyePlot` | `birdsEyeView` | `cubicLaneBoundary` | `fitPolynomialRANSAC` | `monoCamera` | `segmentLaneMarkerRidge`

Introduced in R2018a

findParabolicLaneBoundaries

Find boundaries using parabolic model

Syntax

```
boundaries = findParabolicLaneBoundaries(xyBoundaryPoints,
approxBoundaryWidth)
[boundaries,boundaryPoints] = findParabolicLaneBoundaries(xyBoundaryPoints,
approxBoundaryWidth)
[ ___ ] = findParabolicLaneBoundaries( ___ ,Name,Value)
```

Description

`boundaries = findParabolicLaneBoundaries(xyBoundaryPoints,
approxBoundaryWidth)` uses the random sample consensus (RANSAC) algorithm to find parabolic lane boundary models that fit a set of boundary points and an approximate width. Each model in the returned array of `parabolicLaneBoundary` objects contains the `[A B C]` coefficients of its second-degree polynomial equation and the strength of the boundary estimate.

`[boundaries,boundaryPoints] = findParabolicLaneBoundaries(xyBoundaryPoints,
approxBoundaryWidth)` also returns a cell array of inlier boundary points for each boundary model found.

`[ ___ ] = findParabolicLaneBoundaries( ___ ,Name,Value)` uses options specified by one or more `Name,Value` pair arguments, with any of the preceding syntaxes.

Examples

Find Parabolic Lane Boundaries in Bird's-Eye-View Image

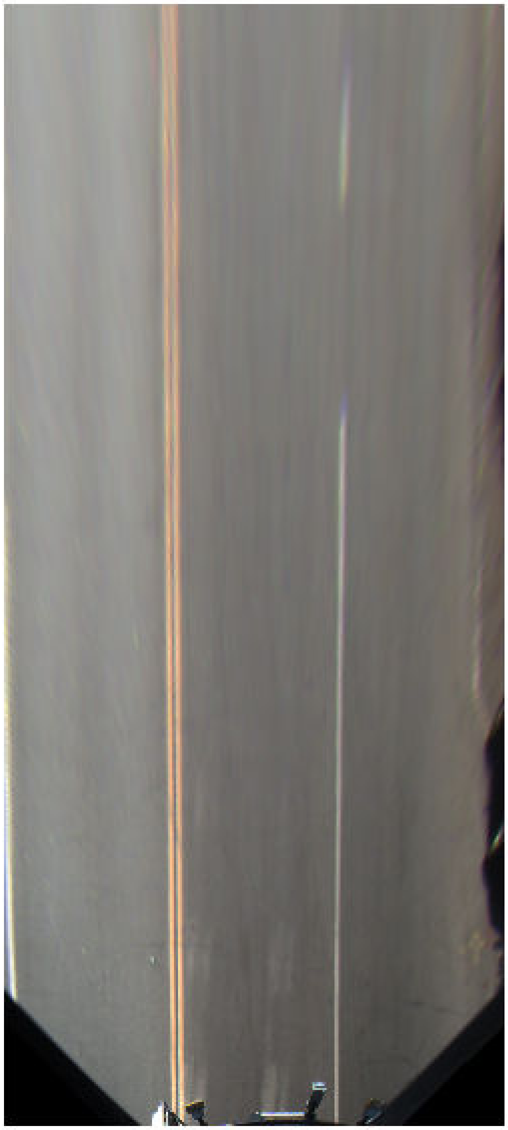
Find lanes in an image by using parabolic lane boundary models. Overlay the identified lanes on the original image and on a bird's-eye-view transformation of the image.

Load an image of a road with lanes. The image was obtained from a camera sensor mounted on the front of a vehicle.

```
I = imread('road.png');
```

Transform the image into a bird's-eye-view image by using a preconfigured sensor object. This object models the sensor that captured the original image.

```
bevSensor = load('birdsEyeConfig');
birdsEyeImage = transformImage(bevSensor.birdsEyeConfig,I);
imshow(birdsEyeImage)
```

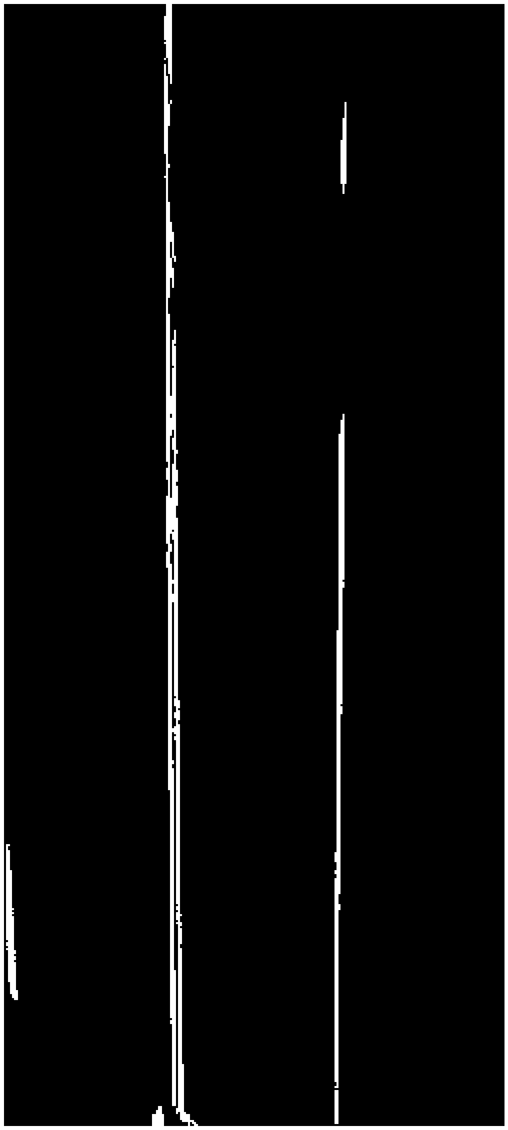


Set the approximate lane marker width in world units (meters).

```
approxBoundaryWidth = 0.25;
```

Detect lane features and display them as a black-and-white image.

```
birdsEyeBW = segmentLaneMarkerRidge(rgb2gray(birdsEyeImage), ...
    bevSensor.birdsEyeConfig,approxBoundaryWidth);
imshow(birdsEyeBW)
```



Obtain lane candidate points in world coordinates.

```
[imageX,imageY] = find(birdsEyeBW);
xyBoundaryPoints = imageToVehicle(bevSensor.birdsEyeConfig,[imageY,imageX]);
```

Find lane boundaries in the image by using the `findParabolicLaneBoundaries` function. By default, the function returns a maximum of two lane boundaries. The boundaries are stored in an array of `parabolicLaneBoundary` objects.

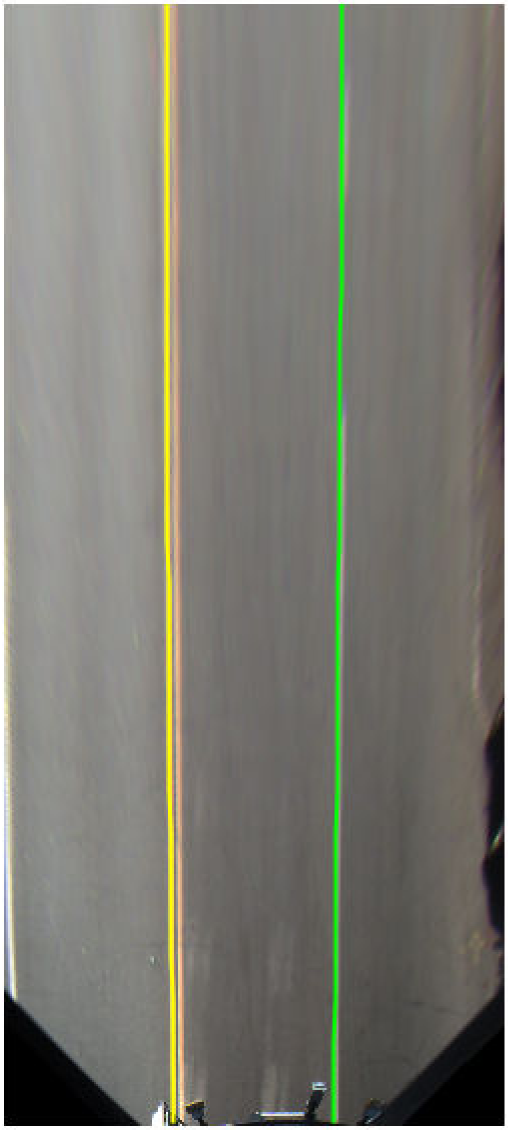
```
boundaries = findParabolicLaneBoundaries(xyBoundaryPoints,approxBoundaryWidth);
```

Use `insertLaneBoundary` to overlay the lanes on the original image. The `XPoints` vector represents the lane points, in meters, that are within range of the ego vehicle's sensor. Specify the lanes in different colors. By default, lanes are yellow.

```
XPoints = 3:30;

figure
sensor = bevSensor.birdsEyeConfig.Sensor;
lanesI = insertLaneBoundary(I,boundaries(1),sensor,XPoints);
lanesI = insertLaneBoundary(lanesI,boundaries(2),sensor,XPoints,'Color','green');
imshow(lanesI)
```

View the lanes in the bird's-eye-view image.

```
figure
BEconfig = bevSensor.birdsEyeConfig;
lanesBEI = insertLaneBoundary(birdsEyeImage,boundaries(1),BEconfig,XPoints);
lanesBEI = insertLaneBoundary(lanesBEI,boundaries(2),BEconfig,XPoints,'Color','green');
imshow(lanesBEI)
```



Input Arguments

`xyBoundaryPoints` — Candidate boundary points
`[x y]` vector

Candidate boundary points, specified as an `[x y]` vector in vehicle coordinates. To obtain the vehicle coordinates for points in a `birdsEyeView` image, use the `imageToVehicle` function to convert the bird's-eye-view image coordinates to vehicle coordinates.

approxBoundaryWidth — Approximate boundary width
real scalar

Approximate boundary width, specified as a real scalar in world units. The width is a horizontal *y*-axis measurement.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MaxSamplingAttempts',200`

MaxNumBoundaries — Maximum number of lane boundaries
`2` (default) | positive integer

Maximum number of lane boundaries that the function attempts to find, specified as the comma-separated pair consisting of `'MaxNumBoundaries'` and a positive integer.

ValidateBoundaryFcn — Function to validate boundary model
function handle

Function to validate the boundary model, specified as the comma-separated pair consisting of `'ValidateBoundaryFcn'` and a function handle. The specified function returns logical `1` (true) if the boundary model is accepted and logical `0` (false) otherwise. Use this function to reject invalid boundaries. The function must be of the form:

```
isValid = validateBoundaryFcn(parameters)
```

`parameters` is a vector corresponding to the three parabolic parameters.

The default validation function always returns `1` (true).

MaxSamplingAttempts — Maximum number of sampling attempts
`100` (default) | positive integer

Maximum number of attempts to find a sample of points that yields a valid parabolic boundary, specified as the comma-separated pair consisting of `'MaxSamplingAttempts'` and a function handle. `findParabolicLaneBoundaries` uses the `fitPolynomialRANSAC` function to sample from the set of boundary points and fit a parabolic boundary line.

Output Arguments

boundaries — Lane boundary models
array of `parabolicLaneBoundary` objects

Lane boundary models, returned as an array of `parabolicLaneBoundary` objects. Lane boundary objects contain the following properties:

- `Parameters` — A three-element vector, `[A B C]`, that corresponds to the three coefficients of a second-degree polynomial equation in general form: $y = Ax^2 + Bx + C$.
- `BoundaryType` — A `LaneBoundaryType` of supported lane boundaries. The supported lane boundary types are:

    - Unmarked
    - Solid
    - Dashed
    - BottsDots
    - DoubleSolid

  Specify a lane boundary type as LaneBoundaryType.*BoundaryType*. For example:

  LaneBoundaryType.BottsDots

- Strength — A ratio of the number of unique *x*-axis locations on the boundary to the total number of points along the line, based on the XExtent property.
- XExtent — A two-element vector describing the minimum and maximum *x*-axis locations for the boundary points.

boundaryPoints — Inlier boundary points
cell array of [x y] values

Inlier boundary points, returned as a cell array of [x y] values. Each element of the cell array corresponds to the same element in the array of parabolicLaneBoundary objects.

Tips

- To fit a single boundary model to a double lane marker, set the approxBoundaryWidth argument to be large enough to include the width spanning both lane markers.

Algorithms

- This function uses fitPolynomialRANSAC to find parabolic models. Because this algorithm uses random sampling, the output can vary between runs.
- The maxDistance parameter of fitPolynomialRANSAC is set to half the width specified in the approxBoundaryWidth argument. Points are considered inliers if they are within the boundary width. The function obtains the final boundary model using a least-squares fit on the inlier points.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

birdsEyePlot | birdsEyeView | fitPolynomialRANSAC | monoCamera | parabolicLaneBoundary | segmentLaneMarkerRidge

Introduced in R2017a

getTrackPositions

Returns updated track positions and position covariance matrix

Syntax

```
position = getTrackPositions(tracks,positionSelector)
[position,positionCovariances] = getTrackPositions(tracks,positionSelector)
```

Description

`position = getTrackPositions(tracks,positionSelector)` returns a matrix of track positions. Each row contains the position of a tracked object.

`[position,positionCovariances] = getTrackPositions(tracks,positionSelector)` returns a matrix of track positions.

Examples

Find Position of 3-D Constant-Acceleration Object

Create an extended Kalman filter tracker for 3-D constant-acceleration motion.

```
tracker = trackerTOMHT('FilterInitializationFcn',@initcaekf);
```

Update the tracker with a single detection and get the tracks output.

```
detection = objectDetection(0,[10;-20;4],'ObjectClassID',3);
tracks = step(tracker,detection,0)
```

```
tracks = 
  objectTrack with properties:

              TrackID: 1
             BranchID: 1
          SourceIndex: 0
           UpdateTime: 0
                  Age: 1
                State: [9x1 double]
      StateCovariance: [9x9 double]
      StateParameters: [1x1 struct]
        ObjectClassID: 3
           TrackLogic: 'Score'
      TrackLogicState: [13.7102 13.7102]
          IsConfirmed: 1
            IsCoasted: 0
       IsSelfReported: 1
     ObjectAttributes: [1x1 struct]
```

Obtain the position vector from the track state.

```
positionSelector = [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];
position = getTrackPositions(tracks, positionSelector)
```

```
position = 1×3

   10.0000  -20.0000    4.0000
```

Find Position and Covariance of 3-D Constant-Velocity Object

Create an extended Kalman filter tracker for 3-D constant-velocity motion.

```
tracker = trackerTOMHT('FilterInitializationFcn',@initcvekf);
```

Update the tracker with a single detection and get the tracks output.

```
detection = objectDetection(0,[10;3;-7],'ObjectClassID',3);
tracks = step(tracker,detection,0)
```

```
tracks = 
  objectTrack with properties:

             TrackID: 1
            BranchID: 1
         SourceIndex: 0
          UpdateTime: 0
                 Age: 1
               State: [6x1 double]
     StateCovariance: [6x6 double]
     StateParameters: [1x1 struct]
       ObjectClassID: 3
          TrackLogic: 'Score'
     TrackLogicState: [13.7102 13.7102]
         IsConfirmed: 1
           IsCoasted: 0
      IsSelfReported: 1
    ObjectAttributes: [1x1 struct]
```

Obtain the position vector and position covariance for that track

```
positionSelector = [1 0 0 0 0 0; 0 0 1 0 0 0; 0 0 0 0 1 0];
[position,positionCovariance] = getTrackPositions(tracks,positionSelector)
```

```
position = 1×3

   10.0000    3.0000   -7.0000


positionCovariance = 3×3

    1.0000   -0.0000         0
   -0.0000    1.0000   -0.0000
         0   -0.0000    1.0000
```

Input Arguments

tracks — Track data structure
struct array

Tracked object, specified as a `struct` array. A track `struct` array is an array of MATLAB `struct` types containing sufficient information to obtain the track position vector and, optionally, the position covariance matrix. At a minimum, the `struct` must contain a `State` column vector field and a positive-definite `StateCovariance` matrix field. For an example of a track `struct` used by Automated Driving Toolbox, examine the output argument, `tracks`, returned by the `updateTracks` function when used with a `multiObjectTracker` System object.

positionSelector — Position selection matrix
D-by-*N* real-valued matrix.

Position selector, specified as a *D*-by-*N* real-valued matrix of ones and zeros. *D* is the number of dimensions of the tracker. *N* is the size of the state vector. Using this matrix, the function extracts track positions from the state vector. Multiply the state vector by position selector matrix returns positions. The same selector is applied to all object tracks.

Output Arguments

position — Positions of tracked objects
real-valued *M*-by-*D* matrix

Positions of tracked objects at last update time, returned as a real-valued *M*-by-*D* matrix. *D* represents the number of position elements. *M* represents the number of tracks.

positionCovariances — Position covariance matrices of tracked objects
real-valued *D*-by-*D*-*M* array

Position covariance matrices of tracked objects, returned as a real-valued *D*-by-*D*-*M* array. *D* represents the number of position elements. *M* represents the number of tracks. Each *D*-by-*D* submatrix is a position covariance matrix for a track.

More About

Position Selector for 2-Dimensional Motion

Show the position selection matrix for two-dimensional motion when the state consists of the position and velocity.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Position Selector for 3-Dimensional Motion

Show the position selection matrix for three-dimensional motion when the state consists of the position and velocity.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Position Selector for 3-Dimensional Motion with Acceleration

Show the position selection matrix for three-dimensional motion when the state consists of the position, velocity, and acceleration.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`getTrackVelocities` | `initcaekf` | `initcakf` | `initcaukf` | `initctekf` | `initctukf` | `initcvkf` | `initcvukf`

Objects
`multiObjectTracker` | `objectDetection`

Introduced in R2017a

getTrackVelocities

Obtain updated track velocities and velocity covariance matrix

Syntax

```
velocity = getTrackVelocities(tracks,velocitySelector)
[velocity,velocityCovariances] = getTrackVelocities(tracks,velocitySelector)
```

Description

`velocity = getTrackVelocities(tracks,velocitySelector)` returns velocities of tracked objects.

`[velocity,velocityCovariances] = getTrackVelocities(tracks,velocitySelector)` also returns the track velocity covariance matrices.

Examples

Find Velocity of 3-D Constant-Acceleration Object

Create an extended Kalman filter tracker for 3-D constant-acceleration motion.

```
tracker = trackerGNN('FilterInitializationFcn',@initcaekf);
```

Initialize the tracker with one detection.

```
detection = objectDetection(0,[10;-20;4],'ObjectClassID',3);
tracks = step(tracker,detection,0);
```

Add a second detection at a later time and at a different position.

```
detection = objectDetection(0.1,[10.3;-20.2;4],'ObjectClassID',3);
tracks = step(tracker,detection,0.2);
```

Obtain the velocity vector from the track state.

```
velocitySelector = [0 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];
velocity = getTrackVelocities(tracks,velocitySelector)
```

```
velocity = 1×3

    1.0093   -0.6728         0
```

Velocity and Covariance of 3-D Constant-Acceleration Object

Create an extended Kalman filter tracker for 3-D constant-acceleration motion.

```
tracker = trackerGNN('FilterInitializationFcn',@initcaekf);
```

Initialize the tracker with one detection.

```
detection = objectDetection(0,[10;-20;4],'ObjectClassID',3);
tracks = step(tracker,detection,0);
```

Add a second detection at a later time and at a different position.

```
detection = objectDetection(0.1,[10.3;-20.2;4.3],'ObjectClassID',3);
tracks = step(tracker,detection,0.2);
```

Obtain the velocity vector from the track state.

```
velocitySelector = [0 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];
[velocity,velocityCovariance] = getTrackVelocities(tracks,velocitySelector)
```

```
velocity = 1×3

    1.0093   -0.6728    1.0093


velocityCovariance = 3×3

   70.0685         0         0
         0   70.0685         0
         0         0   70.0685
```

Input Arguments

`tracks` — Track data structure

struct array

Tracked object, specified as a `struct` array. A track `struct` array is an array of MATLAB `struct` types containing sufficient information to obtain the track position vector and, optionally, the position covariance matrix. At a minimum, the `struct` must contain a `State` column vector field and a positive-definite `StateCovariance` matrix field. For an example of a track `struct` used by Automated Driving Toolbox, examine the output argument, `tracks`, returned by the `updateTracks` function when used with a `multiObjectTracker` System object.

`velocitySelector` — Velocity selection matrix

D-by-*N* real-valued matrix.

Velocity selector, specified as a *D*-by-*N* real-valued matrix of ones and zeros. *D* is the number of dimensions of the tracker. *N* is the size of the state vector. Using this matrix, the function extracts track velocities from the state vector. Multiply the state vector by velocity selector matrix returns velocities. The same selector is applied to all object tracks.

Output Arguments

`velocity` — Velocities of tracked objects

real-valued *1*-by-*D* vector | real-valued *M*-by-*D* matrix

Velocities of tracked objects at last update time, returned as a *1*-by-*D* vector or a real-valued *M*-by-*D* matrix. *D* represents the number of velocity elements. *M* represents the number of tracks.

`velocityCovariances` — Velocity covariance matrices of tracked objects
real-valued *D*-by-*D*-matrix | real-valued *D*-by-*D*-by-*M* array

Velocity covariance matrices of tracked objects, returned as a real-valued *D*-by-*D*-matrix or a real-valued *D*-by-*D*-by-*M* array. *D* represents the number of velocity elements. *M* represents the number of tracks. Each *D*-by-*D* submatrix is a velocity covariance matrix for a track.

More About

Velocity Selector for 2-Dimensional Motion

Show the velocity selection matrix for two-dimensional motion when the state consists of the position and velocity.

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Velocity Selector for 3-Dimensional Motion

Show the velocity selection matrix for three-dimensional motion when the state consists of the position and velocity.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Velocity Selector for 3-Dimensional Motion with Acceleration

Show the velocity selection matrix for three-dimensional motion when the state consists of the position, velocity, and acceleration.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`getTrackPositions` | `initcaekf` | `initcakf` | `initcaukf` | `initctekf` | `initctukf` | `initcvkf` | `initcvukf`

Objects
`multiObjectTracker` | `objectDetection`

Introduced in R2017a

hereHDLMCredentials

Set up or delete HERE HD Live Map credentials

Syntax

```
hereHDLMCredentials('setup')
hereHDLMCredentials('delete')
```

Description

`hereHDLMCredentials('setup')` opens a dialog box for specifying the credentials required to access the HERE HD Live Map[2] (HERE HDLM) web service. By default, credentials last for the duration of a MATLAB session. To save credentials between sessions, in the HERE HD Live Map Credentials dialog box, select the **Save my credentials between MATLAB sessions** check box .

Simplified form: `hereHDLMCredentials setup`

`hereHDLMCredentials('delete')` deletes saved HERE HDLM credentials. Any subsequent use of HERE HDLM functions and objects, such as the `hereHDLMConfiguration` or `hereHDLMReader` object, requires entering new credentials.

Simplified form: `hereHDLMCredentials delete`

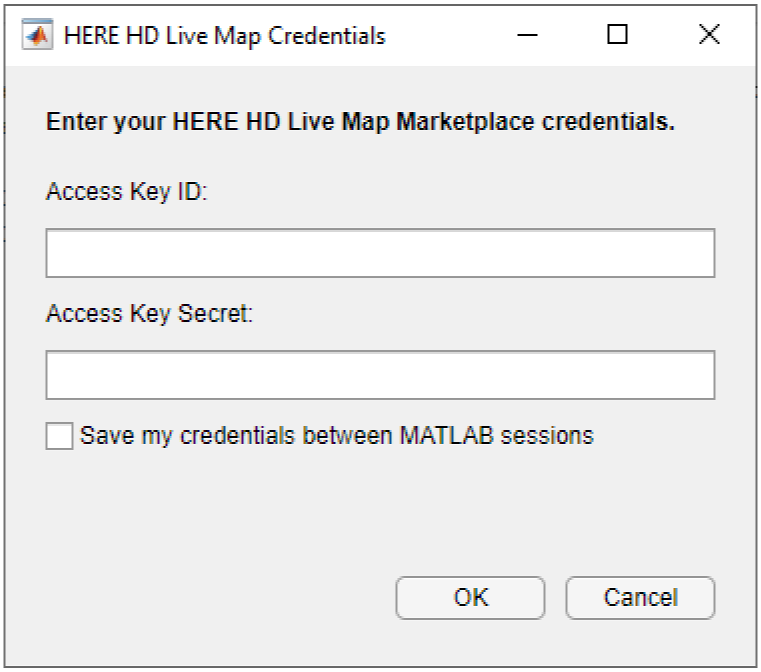
Examples

Manage HERE HD Live Map Credentials

Set up HERE HD Live Map (HERE HDLM) credentials.

```
hereHDLMCredentials setup
```

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.



Enter a valid **Access Key ID** and **Access Key Secret**. You can obtain these credentials by entering into a separate agreement with HERE Technologies. Optionally, select **Save my credentials between MATLAB sessions** to save your HERE HDLM credentials between MATLAB sessions. Click **OK**.

Load a driving route, and create a HERE HDLM reader using the route coordinates. The HERE HD Live Map Credentials dialog box does not open, because the credentials have already been set up.

```
data = load('geoSequence.mat');
reader = hereHDLMReader(data.latitude,data.longitude);
```

Delete the HERE HDLM credentials you previously entered. The next time you use `hereHDLMReader`, you must enter your credentials again.

```
hereHDLMCredentials delete
```

See Also

`hereHDLMConfiguration` | `hereHDLMReader`

Topics

"Read and Visualize HERE HD Live Map Data"

Introduced in R2019a

initcaabf

Create constant acceleration alpha-beta tracking filter from detection report

Syntax

`abf = initcaabf(detection)`

Description

`abf = initcaabf(detection)` initializes a constant acceleration alpha-beta tracking filter for object tracking based on information provided in `detection`.

Examples

Creating Constant Acceleration trackingABF Object from Detection

Create an objectDetection with a position measurement at x=1, y=3 and a measurement noise of [1 0.2; 0.2 2];

```
detection = objectDetection(0,[1;3],'MeasurementNoise',[1 0.2;0.2 2]);
```

Use `initccabf` to create a `trackingABF` filter initialized at the provided position and using the measurement noise defined above.

```
ABF = initcaabf(detection);
```

Check the values of the state and measurement noise. Verify that the filter state, `ABF.State`, has the same position components as the `Detection.Measurement`. Verify that the filter measurement noise, `ABF.MeasurementNoise`, is the same as the `Detection.MeasurementNoise` values.

```
ABF.State
```

```
ans = 6×1

     1
     0
     0
     3
     0
     0
```

```
ABF.MeasurementNoise
```

```
ans = 2×2

    1.0000    0.2000
    0.2000    2.0000
```

Input Arguments

`detection` — Detection report
`objectDetection` object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

Output Arguments

`abf` — Constant velocity alpha-beta filter
`trackingABF` object

Constant acceleration alpha-beta tracking filter for object tracking, returned as a `trackingABF` object.

Algorithms

- The function computes the process noise matrix assuming a unit standard deviation for the acceleration change rate.
- You can use this function as the `FilterInitializationFcn` property of trackers.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`objectDetection` | `trackingABF` | `trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2020a

initcvabf

Create constant velocity tracking alpha-beta filter from detection report

Syntax

```
abf = initcvabf(detection)
```

Description

`abf = initcvabf(detection)` initializes a constant velocity alpha-beta filter for object tracking based on information provided in `detection`.

Examples

Creating trackingABF Object from Detection

Create an objectDetection with a position measurement at x=1, y=3 and a measurement noise of [1 0.2; 0.2 2];

```
detection = objectDetection(0,[1;3],'MeasurementNoise',[1 0.2;0.2 2]);
```

Use `initcvabf` to create a `trackingABF` filter initialized at the provided position and using the measurement noise defined above.

```
ABF = initcvabf(detection);
```

Check the values of the state and measurement noise. Verify that the filter state, `ABF.State`, has the same position components as the `Detection.Measurement`. Verify that the filter measurement noise, `ABF.MeasurementNoise`, is the same as the `Detection.MeasurementNoise` values.

```
ABF.State
```

```
ans = 4×1

     1
     0
     3
     0
```

```
ABF.MeasurementNoise
```

```
ans = 2×2

    1.0000    0.2000
    0.2000    2.0000
```

Input Arguments

`detection` — Detection report
`objectDetection` object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

Output Arguments

`abf` — Constant velocity alpha-beta filter
`trackingABF` object

Constant velocity alpha-beta tracking filter for object tracking, returned as a `trackingABF` object.

Algorithms

- The function computes the process noise matrix assuming a unit acceleration standard deviation.
- You can use this function as the `FilterInitializationFcn` property of trackers.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`objectDetection` | `trackingABF` | `trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2020a

initcaekf

Create constant-acceleration extended Kalman filter from detection report

Syntax

```
filter = initcaekf(detection)
```

Description

`filter = initcaekf(detection)` creates and initializes a constant-acceleration extended Kalman `filter` from information contained in a `detection` report. For more information about the extended Kalman filter, see `trackingEKF`.

Examples

Initialize 3-D Constant-Acceleration Extended Kalman Filter

Create and initialize a 3-D constant-acceleration extended Kalman filter object from an initial detection report.

Create the detection report from an initial 3-D measurement, *(-200;30;0)* , of the object position. Assume uncorrelated measurement noise.

```
detection = objectDetection(0,[-200;-30;0],'MeasurementNoise',2.1*eye(3), ...
    'SensorIndex',1,'ObjectClassID',1,'ObjectAttributes',{'Car',2});
```

Create the new filter from the detection report and display its properties.

```
filter = initcaekf(detection)
```

```
filter = 
  trackingEKF with properties:

                          State: [9x1 double]
                StateCovariance: [9x9 double]

             StateTransitionFcn: @constacc
     StateTransitionJacobianFcn: @constaccjac
                   ProcessNoise: [3x3 double]
        HasAdditiveProcessNoise: 0

                 MeasurementFcn: @cameas
         MeasurementJacobianFcn: @cameasjac
               MeasurementNoise: [3x3 double]
    HasAdditiveMeasurementNoise: 1
```

Show the filter state.

```
filter.State
```

```
ans = 9×1

  -200
     0
     0
   -30
     0
     0
     0
     0
     0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 9×9

    2.1000         0         0         0         0         0         0         0         0
         0  100.0000         0         0         0         0         0         0         0
         0         0  100.0000         0         0         0         0         0         0
         0         0         0    2.1000         0         0         0         0         0
         0         0         0         0  100.0000         0         0         0         0
         0         0         0         0         0  100.0000         0         0         0
         0         0         0         0         0         0    2.1000         0         0
         0         0         0         0         0         0         0  100.0000         0
         0         0         0         0         0         0         0         0  100.0000
```

Create 3D Constant Acceleration EKF from Spherical Measurement

Initialize a 3D constant-acceleration extended Kalman filter from an initial detection report made from an initial measurement in spherical coordinates. If you want to use spherical coordinates, then you must supply a measurement parameter structure as part of the detection report with the `Frame` field set to `'spherical'`. Set the azimuth angle of the target to 45˚, the elevation to 22˚, the range to 1000 meters, and the range rate to -4.0 m/s.

```
frame = 'spherical';
sensorpos = [25,-40,-10].';
sensorvel = [0;5;0];
laxes = eye(3);
```

Create the measurement parameters structure. Set `'HasVelocity'` and `'HasElevation'` to `true`. Then, the measurement vector consists of azimuth, elevation, range, and range rate.

```
measparms = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes,'HasVelocity',true, ...
    'HasElevation',true);
meas = [45;22;1000;-4];
measnoise = diag([3.0,2.5,2,1.0].^2);
detection = objectDetection(0,meas,'MeasurementNoise', ...
    measnoise,'MeasurementParameters',measparms)
```

```
detection = 
  objectDetection with properties:
```

```
                    Time: 0
             Measurement: [4x1 double]
        MeasurementNoise: [4x4 double]
             SensorIndex: 1
           ObjectClassID: 0
   MeasurementParameters: [1x1 struct]
        ObjectAttributes: {}
```

```
filter = initcaekf(detection);
```

Display the state vector.

```
disp(filter.State)
```

```
  680.6180
   -2.6225
         0
  615.6180
    2.3775
         0
  364.6066
   -1.4984
         0
```

Input Arguments

detection — Detection report
objectDetection object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

Output Arguments

filter — Extended Kalman filter
trackingEKF object

Extended Kalman filter, returned as a `trackingEKF` object.

Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration-rate standard deviation of 1 m/s^3.
- You can use this function as the `FilterInitializationFcn` property of a `multiObjectTracker` object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`initcakf` | `initcaukf` | `initctekf` | `initctukf` | `initcvekf` | `initcvkf` | `initcvukf`

Objects
`multiObjectTracker` | `objectDetection` | `trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

initcakf

Create constant-acceleration linear Kalman filter from detection report

Syntax

```
filter = initcakf(detection)
```

Description

`filter = initcakf(detection)` creates and initializes a constant-acceleration linear Kalman `filter` from information contained in a `detection` report. For more information about the linear Kalman filter, see `trackingKF`.

Examples

Initialize 2-D Constant-Acceleration Linear Kalman Filter

Create and initialize a 2-D constant-acceleration linear Kalman filter object from an initial detection report.

Create the detection report from an initial 2-D measurement, (10,−5), of the object position. Assume uncorrelated measurement noise.

```
detection = objectDetection(0,[10;-5],'MeasurementNoise',eye(2), ...
    'SensorIndex',1,'ObjectClassID',1,'ObjectAttributes',{'Car',5});
```

Create the new filter from the detection report.

```
filter = initcakf(detection);
```

Show the filter state.

```
filter.State
```

```
ans = 6×1

    10
     0
     0
    -5
     0
     0
```

Show the state transition model.

```
filter.StateTransitionModel
```

```
ans = 6×6

    1.0000    1.0000    0.5000         0         0         0
```

```
     0    1.0000    1.0000         0         0         0
     0         0    1.0000         0         0         0
     0         0         0    1.0000    1.0000    0.5000
     0         0         0         0    1.0000    1.0000
     0         0         0         0         0    1.0000
```

Input Arguments

`detection` — Detection report
`objectDetection` object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

Output Arguments

`filter` — Linear Kalman filter
`trackingKF` object

Linear Kalman filter, returned as a `trackingKF` object.

Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration rate standard deviation of 1 m/s^3.
- You can use this function as the `FilterInitializationFcn` property of a `multiObjectTracker` object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`initcaekf` | `initcaukf` | `initctekf` | `initctukf` | `initcvekf` | `initcvkf` | `initcvukf`

Objects
`multiObjectTracker` | `objectDetection` | `trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

initcaukf

Create constant-acceleration unscented Kalman filter from detection report

Syntax

```
filter = initcaukf(detection)
```

Description

`filter = initcaukf(detection)` creates and initializes a constant-acceleration unscented Kalman `filter` from information contained in a `detection` report. For more information about the unscented Kalman filter, see `trackingUKF`.

Examples

Initialize 3-D Constant-Acceleration Unscented Kalman Filter

Create and initialize a 3-D constant-acceleration unscented Kalman filter object from an initial detection report.

Create the detection report from an initial 3-D measurement, (-200,-30,5), of the object position. Assume uncorrelated measurement noise.

```
detection = objectDetection(0,[-200;-30;5],'MeasurementNoise',2.0*eye(3), ...
    'SensorIndex',1,'ObjectClassID',1,'ObjectAttributes',{'Car',2});
```

Create the new filter from the detection report and display the filter properties.

```
filter = initcaukf(detection)
```

```
filter = 
  trackingUKF with properties:

                          State: [9x1 double]
                StateCovariance: [9x9 double]

             StateTransitionFcn: @constacc
                   ProcessNoise: [3x3 double]
        HasAdditiveProcessNoise: 0

                 MeasurementFcn: @cameas
               MeasurementNoise: [3x3 double]
    HasAdditiveMeasurementNoise: 1

                          Alpha: 1.0000e-03
                           Beta: 2
                          Kappa: 0
```

Show the state.

```
filter.State
```

```
ans = 9×1

  -200
     0
     0
   -30
     0
     0
     5
     0
     0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 9×9

    2.0000         0         0         0         0         0         0         0         0
         0  100.0000         0         0         0         0         0         0         0
         0         0  100.0000         0         0         0         0         0         0
         0         0         0    2.0000         0         0         0         0         0
         0         0         0         0  100.0000         0         0         0         0
         0         0         0         0         0  100.0000         0         0         0
         0         0         0         0         0         0    2.0000         0         0
         0         0         0         0         0         0         0  100.0000         0
         0         0         0         0         0         0         0         0  100.0000
```

Create 3D Constant Acceleration UKF from Spherical Measurement

Initialize a 3D constant-acceleration unscented Kalman filter from an initial detection report made from a measurement in spherical coordinates. If you want to use spherical coordinates, then you must supply a measurement parameter structure as part of the detection report with the `Frame` field set to `'spherical'`. Set the azimuth angle of the target to 45˚, and the range to 1000 meters.

```
frame = 'spherical';
sensorpos = [25,-40,-10].';
sensorvel = [0;5;0];
laxes = eye(3);
```

Create the measurement structure. Set `'HasVelocity'` and `'HasElevation'` to `false`. Then, the measurement vector consists of azimuth angle and range.

```
measparms = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes,'HasVelocity',false, ...
    'HasElevation',false);
meas = [45;1000];
measnoise = diag([3.0,2.0].^2);
detection = objectDetection(0,meas,'MeasurementNoise', ...
    measnoise,'MeasurementParameters',measparms)
```

```
detection = 
  objectDetection with properties:

                     Time: 0
              Measurement: [2x1 double]
         MeasurementNoise: [2x2 double]
              SensorIndex: 1
            ObjectClassID: 0
    MeasurementParameters: [1x1 struct]
         ObjectAttributes: {}
```

```
filter = initcaukf(detection);
```

Display the state vector.

```
disp(filter.State)
```

```
  732.1068
         0
         0
  667.1068
         0
         0
  -10.0000
         0
         0
```

Input Arguments

`detection` — Detection report
`objectDetection` object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

Output Arguments

`filter` — Unscented Kalman filter
`trackingUKF` object

Unscented Kalman filter, returned as a `trackingUKF` object.

Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration rate standard deviation of 1 m/s^3.
- You can use this function as the `FilterInitializationFcn` property of a `multiObjectTracker` object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`initcaekf` | `initcakf` | `initctekf` | `initctukf` | `initcvekf` | `initcvkf` | `initcvukf`

Objects
`multiObjectTracker` | `objectDetection` | `trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

initctekf

Create constant turn-rate extended Kalman filter from detection report

Syntax

```
filter = initctekf(detection)
```

Description

`filter = initctekf(detection)` creates and initializes a constant-turn-rate extended Kalman `filter` from information contained in a `detection` report. For more information about the extended Kalman filter, see `trackingEKF`.

Examples

Initialize 2-D Constant Turn-Rate Extended Kalman Filter

Create and initialize a 2-D constant turn-rate extended Kalman filter object from an initial detection report.

Create the detection report from an initial 2-D measurement, (-250,-40), of the object position. Assume uncorrelated measurement noise.

Extend the measurement to three dimensions by adding a *z*-component of zero.

```
detection = objectDetection(0,[-250;-40;0],'MeasurementNoise',2.0*eye(3), ...
    'SensorIndex',1,'ObjectClassID',1,'ObjectAttributes',{'Car',2});
```

Create the new filter from the detection report and display the filter properties.

```
filter = initctekf(detection)
```

```
filter = 
  trackingEKF with properties:

                          State: [7x1 double]
                StateCovariance: [7x7 double]

             StateTransitionFcn: @constturn
     StateTransitionJacobianFcn: @constturnjac
                   ProcessNoise: [4x4 double]
        HasAdditiveProcessNoise: 0

                 MeasurementFcn: @ctmeas
         MeasurementJacobianFcn: @ctmeasjac
               MeasurementNoise: [3x3 double]
    HasAdditiveMeasurementNoise: 1
```

Show the state.

```
filter.State
```

```
ans = 7×1

  -250
     0
   -40
     0
     0
     0
     0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 7×7

    2.0000         0         0         0         0         0         0
         0  100.0000         0         0         0         0         0
         0         0    2.0000         0         0         0         0
         0         0         0  100.0000         0         0         0
         0         0         0         0  100.0000         0         0
         0         0         0         0         0    2.0000         0
         0         0         0         0         0         0  100.0000
```

Create 2-D Constant Turnrate EKF from Spherical Measurement

Initialize a 2-D constant-turnrate extended Kalman filter from an initial detection report made from an initial measurement in spherical coordinates. If you want to use spherical coordinates, then you must supply a measurement parameter structure as part of the detection report with the `Frame` field set to `'spherical'`. Set the azimuth angle of the target to 45 degrees, the range to 1000 meters, and the range rate to -4.0 m/s.

```
frame = 'spherical';
sensorpos = [25,-40,-10].';
sensorvel = [0;5;0];
laxes = eye(3);
```

Create the measurement parameters structure. Set `'HasElevation'` to `false`. Then, the measurement consists of azimuth, range, and range rate.

```
measparms = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes,'HasVelocity',true, ...
    'HasElevation',false);
meas = [45;1000;-4];
measnoise = diag([3.0,2,1.0].^2);
detection = objectDetection(0,meas,'MeasurementNoise', ...
    measnoise,'MeasurementParameters',measparms)
```

```
detection = 
  objectDetection with properties:

                     Time: 0
```

```
            Measurement: [3x1 double]
       MeasurementNoise: [3x3 double]
            SensorIndex: 1
          ObjectClassID: 0
  MeasurementParameters: [1x1 struct]
       ObjectAttributes: {}
```

```
filter = initctekf(detection);
```

Filter state vector.

```
disp(filter.State)
```

```
  732.1068
   -2.8284
  667.1068
    2.1716
         0
  -10.0000
         0
```

Input Arguments

`detection` — Detection report

`objectDetection` object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

Output Arguments

`filter` — Extended Kalman filter

`trackingEKF` object

Extended Kalman filter, returned as a `trackingEKF` object.

Algorithms

- The function computes the process noise matrix assuming a one-second time step. The function assumes an acceleration standard deviation of 1 m/s^2, and a turn-rate acceleration standard deviation of 1°/s^2.
- You can use this function as the `FilterInitializationFcn` property of a `multiObjectTracker` object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
initcaekf | initcakf | initcaukf | initctukf | initcvekf | initcvkf | initcvukf

Objects
multiObjectTracker | objectDetection | trackingEKF | trackingKF | trackingUKF

Introduced in R2017a

initctukf

Create constant turn-rate unscented Kalman filter from detection report

Syntax

```
filter = initctukf(detection)
```

Description

`filter = initctukf(detection)` creates and initializes a constant-turn-rate unscented Kalman `filter` from information contained in a `detection` report. For more information about the unscented Kalman filter, see `trackingUKF`.

Examples

Initialize 2-D Constant Turn-Rate Unscented Kalman Filter

Create and initialize a 2-D constant turn-rate unscented Kalman filter object from an initial detection report.

Create the detection report from an initial 2D measurement, (-250,-40), of the object position. Assume uncorrelated measurement noise.

Extend the measurement to three dimensions by adding a z-component of zero.

```
detection = objectDetection(0,[-250;-40;0],'MeasurementNoise',2.0*eye(3), ...
    'SensorIndex',1,'ObjectClassID',1,'ObjectAttributes',{'Car',2});
```

Create the new filter from the detection report and display the filter properties.

```
filter = initctukf(detection)
```

```
filter = 
  trackingUKF with properties:

                          State: [7x1 double]
                StateCovariance: [7x7 double]

             StateTransitionFcn: @constturn
                   ProcessNoise: [4x4 double]
        HasAdditiveProcessNoise: 0

                 MeasurementFcn: @ctmeas
               MeasurementNoise: [3x3 double]
    HasAdditiveMeasurementNoise: 1

                          Alpha: 1.0000e-03
                           Beta: 2
                          Kappa: 0
```

Show the filter state.

```
filter.State
```

```
ans = 7×1

  -250
     0
   -40
     0
     0
     0
     0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 7×7

    2.0000         0         0         0         0         0         0
         0  100.0000         0         0         0         0         0
         0         0    2.0000         0         0         0         0
         0         0         0  100.0000         0         0         0
         0         0         0         0  100.0000         0         0
         0         0         0         0         0    2.0000         0
         0         0         0         0         0         0  100.0000
```

Create 2-D Constant Turn-rate UKF from Spherical Measurement

Initialize a 2-D constant turn-rate extended Kalman filter from an initial detection report made from an initial measurement in spherical coordinates. If you want to use spherical coordinates, then you must supply a measurement parameter structure as part of the detection report with the `Frame` field set to `'spherical'`. Set the azimuth angle of the target to 45 degrees and the range to 1000 meters.

```
frame = 'spherical';
sensorpos = [25,-40,-10].';
sensorvel = [0;5;0];
laxes = eye(3);
```

Create the measurement parameters structure. Set `'HasVelocity'` and `'HasElevation'` to `false`. Then, the measurement consists of azimuth and range.

```
measparms = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes,'HasVelocity',false, ...
    'HasElevation',false);
meas = [45;1000];
measnoise = diag([3.0,2].^2);
detection = objectDetection(0,meas,'MeasurementNoise', ...
    measnoise,'MeasurementParameters',measparms)
```

```
detection = 
  objectDetection with properties:
```

```
                    Time: 0
             Measurement: [2x1 double]
        MeasurementNoise: [2x2 double]
             SensorIndex: 1
           ObjectClassID: 0
   MeasurementParameters: [1x1 struct]
        ObjectAttributes: {}
```

```
filter = initctukf(detection);
```

Filter state vector.

```
disp(filter.State)
```

```
  732.1068
         0
  667.1068
         0
         0
  -10.0000
         0
```

Input Arguments

`detection` — Detection report
`objectDetection` object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

Output Arguments

`filter` — Unscented Kalman filter
`trackingUKF` object

Unscented Kalman filter, returned as a `trackingUKF` object.

Algorithms

- The function computes the process noise matrix assuming a one-second time step. The function assumes an acceleration standard deviation of 1 m/s^2, and a turn-rate acceleration standard deviation of 1°/s^2.
- You can use this function as the `FilterInitializationFcn` property of a `multiObjectTracker` object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`initcaekf` | `initcakf` | `initcaukf` | `initctekf` | `initcvekf` | `initcvkf` | `initcvukf`

Objects
`multiObjectTracker` | `objectDetection` | `trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

initcvekf

Create constant-velocity extended Kalman filter from detection report

Syntax

`filter = initcvekf(detection)`

Description

`filter = initcvekf(detection)` creates and initializes a constant-velocity extended Kalman `filter` from information contained in a `detection` report. For more information about the extended Kalman filter, see `trackingEKF`.

Examples

Initialize 3-D Constant-Velocity Extended Kalman Filter

Create and initialize a 3-D constant-velocity extended Kalman filter object from an initial detection report.

Create the detection report from an initial 3-D measurement, (10,20,−5), of the object position.

```
detection = objectDetection(0,[10;20;-5],'MeasurementNoise',1.5*eye(3), ...
    'SensorIndex',1,'ObjectClassID',1,'ObjectAttributes',{'Sports Car',5});
```

Create the new filter from the detection report.

```
filter = initcvekf(detection)
```

```
filter = 
  trackingEKF with properties:

                          State: [6x1 double]
                StateCovariance: [6x6 double]

             StateTransitionFcn: @constvel
     StateTransitionJacobianFcn: @constveljac
                   ProcessNoise: [3x3 double]
        HasAdditiveProcessNoise: 0

                 MeasurementFcn: @cvmeas
         MeasurementJacobianFcn: @cvmeasjac
               MeasurementNoise: [3x3 double]
    HasAdditiveMeasurementNoise: 1
```

Show the filter state.

```
filter.State
```

```
ans = 6×1
```

```
    10
     0
    20
     0
    -5
     0
```

Show the state covariance.

```
filter.StateCovariance
```

```
ans = 6×6

    1.5000         0         0         0         0         0
         0  100.0000         0         0         0         0
         0         0    1.5000         0         0         0
         0         0         0  100.0000         0         0
         0         0         0         0    1.5000         0
         0         0         0         0         0  100.0000
```

Create 3-D Constant Velocity EKF from Spherical Measurement

Initialize a 3-D constant-velocity extended Kalman filter from an initial detection report made from a 3-D measurement in spherical coordinates. If you want to use spherical coordinates, then you must supply a measurement parameter structure as part of the detection report with the `Frame` field set to `'spherical'`. Set the azimuth angle of the target to 45 degrees, the elevation to -10 degrees, the range to 1000 meters, and the range rate to -4.0 m/s.

```
frame = 'spherical';
sensorpos = [25,-40,0].';
sensorvel = [0;5;0];
laxes = eye(3);
measparms = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes,'HasVelocity',true, ...
    'HasElevation',true);
meas = [45;-10;1000;-4];
measnoise = diag([3.0,2.5,2,1.0].^2);
detection = objectDetection(0,meas,'MeasurementNoise', ...
    measnoise,'MeasurementParameters',measparms)
```

```
detection = 
  objectDetection with properties:

                     Time: 0
              Measurement: [4x1 double]
         MeasurementNoise: [4x4 double]
              SensorIndex: 1
            ObjectClassID: 0
    MeasurementParameters: [1x1 struct]
         ObjectAttributes: {}
```

```
filter = initcvekf(detection);
```

Filter state vector.

```
disp(filter.State)
```

```
  721.3642
   -2.7855
  656.3642
    2.2145
 -173.6482
    0.6946
```

Input Arguments

`detection` — Detection report
`objectDetection` object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

Output Arguments

`filter` — Extended Kalman filter
`trackingEKF` object

Extended Kalman filter, returned as a `trackingEKF` object.

Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration standard deviation of 1 m/s^2.
- You can use this function as the `FilterInitializationFcn` property of a `multiObjectTracker` object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`initcaekf` | `initcakf` | `initcaukf` | `initctekf` | `initctukf` | `initcvkf` | `initcvukf`

Objects
`multiObjectTracker` | `objectDetection` | `trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

initcvkf

Create constant-velocity linear Kalman filter from detection report

Syntax

```
filter = initcvkf(detection)
```

Description

`filter = initcvkf(detection)` creates and initializes a constant-velocity linear Kalman `filter` from information contained in a `detection` report. For more information about the linear Kalman filter, see `trackingKF`.

Examples

Initialize 2-D Constant-Velocity Linear Kalman Filter

Create and initialize a 2-D linear Kalman filter object from an initial detection report.

Create the detection report from an initial 2-D measurement, (10,20), of the object position.

```
detection = objectDetection(0,[10;20],'MeasurementNoise',[1 0.2; 0.2 2], ...
    'SensorIndex',1,'ObjectClassID',1,'ObjectAttributes',{'Yellow Car',5});
```

Create the new track from the detection report.

```
filter = initcvkf(detection)
```

```
filter = 
  trackingKF with properties:

               State: [4x1 double]
     StateCovariance: [4x4 double]

         MotionModel: '2D Constant Velocity'
        ControlModel: []
        ProcessNoise: [4x4 double]

    MeasurementModel: [2x4 double]
    MeasurementNoise: [2x2 double]
```

Show the state.

```
filter.State
```

```
ans = 4×1

    10
     0
    20
```

```
     0
```

Show the state transition model.

```
filter.StateTransitionModel
```

```
ans = 4×4

     1     1     0     0
     0     1     0     0
     0     0     1     1
     0     0     0     1
```

Initialize 3-D Constant-Velocity Linear Kalman Filter

Create and initialize a 3-D linear Kalman filter object from an initial detection report.

Create the detection report from an initial 3-D measurement, (10,20,−5), of the object position.

```
detection = objectDetection(0,[10;20;-5],'MeasurementNoise',eye(3), ...
    'SensorIndex', 1,'ObjectClassID',1,'ObjectAttributes',{'Green Car', 5});
```

Create the new filter from the detection report and display its properties.

```
filter = initcvkf(detection)
```

```
filter = 
  trackingKF with properties:

               State: [6x1 double]
     StateCovariance: [6x6 double]

         MotionModel: '3D Constant Velocity'
        ControlModel: []
        ProcessNoise: [6x6 double]

    MeasurementModel: [3x6 double]
    MeasurementNoise: [3x3 double]
```

Show the state.

```
filter.State
```

```
ans = 6×1

    10
     0
    20
     0
    -5
     0
```

Show the state transition model.

```
filter.StateTransitionModel

ans = 6×6

     1     1     0     0     0     0
     0     1     0     0     0     0
     0     0     1     1     0     0
     0     0     0     1     0     0
     0     0     0     0     1     1
     0     0     0     0     0     1
```

Input Arguments

`detection` — Detection report
`objectDetection` object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

Output Arguments

`filter` — Linear Kalman filter
`trackingKF` object

Linear Kalman filter, returned as a `trackingKF` object.

Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration standard deviation of 1 m/s^2.
- You can use this function as the `FilterInitializationFcn` property of a `multiObjectTracker` object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`initcaekf` | `initcakf` | `initcaukf` | `initctekf` | `initctukf` | `initcvekf` | `initcvukf`

Objects
`multiObjectTracker` | `objectDetection` | `trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

initcvukf

Create constant-velocity unscented Kalman filter from detection report

Syntax

```
filter = initcvukf(detection)
```

Description

`filter = initcvukf(detection)` creates and initializes a constant-velocity unscented Kalman `filter` from information contained in a `detection` report. For more information about the unscented Kalman filter, see `trackingUKF`.

Examples

Initialize 3-D Constant-Velocity Unscented Kalman Filter

Create and initialize a 3-D constant-velocity unscented Kalman filter object from an initial detection report.

Create the detection report from an initial 3-D measurement, (10,200,−5), of the object position.

```
detection = objectDetection(0,[10;200;-5],'MeasurementNoise',1.5*eye(3), ...
    'SensorIndex',1,'ObjectClassID',1,'ObjectAttributes',{'Sports Car',5});
```

Create the new filter from the detection report and display the filter properties.

```
filter = initcvukf(detection)
```

```
filter = 
  trackingUKF with properties:

                          State: [6x1 double]
                StateCovariance: [6x6 double]

             StateTransitionFcn: @constvel
                   ProcessNoise: [3x3 double]
        HasAdditiveProcessNoise: 0

                 MeasurementFcn: @cvmeas
               MeasurementNoise: [3x3 double]
    HasAdditiveMeasurementNoise: 1

                          Alpha: 1.0000e-03
                           Beta: 2
                          Kappa: 0
```

Display the state.

```
filter.State
```

```
ans = 6×1

    10
     0
   200
     0
    -5
     0
```

Show the state covariance.

```
filter.StateCovariance
```

```
ans = 6×6

    1.5000         0         0         0         0         0
         0  100.0000         0         0         0         0
         0         0    1.5000         0         0         0
         0         0         0  100.0000         0         0
         0         0         0         0    1.5000         0
         0         0         0         0         0  100.0000
```

Create Constant Velocity UKF from Spherical Measurement

Initialize a constant-velocity unscented Kalman filter from an initial detection report made from an initial measurement in spherical coordinates. Because the object lies in the *x*-*y* plane, no elevation measurement is made. If you want to use spherical coordinates, then you must supply a measurement parameter structure as part of the detection report with the `Frame` field set to `'spherical'`. Set the azimuth angle of the target to 45 degrees, the range to 1000 meters, and the range rate to -4.0 m/s.

```
frame = 'spherical';
sensorpos = [25,-40,0].';
sensorvel = [0;5;0];
laxes = eye(3);
```

Create the measurement parameters structure. Set `'HasElevation'` to `false`. Then, the measurement consists of azimuth, range, and range rate.

```
measparms = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes,'HasVelocity',true, ...
    'HasElevation',false);
meas = [45;1000;-4];
measnoise = diag([3.0,2,1.0].^2);
detection = objectDetection(0,meas,'MeasurementNoise', ...
    measnoise,'MeasurementParameters',measparms)
```

```
detection = 
  objectDetection with properties:

                     Time: 0
              Measurement: [3x1 double]
         MeasurementNoise: [3x3 double]
              SensorIndex: 1
```

```
           ObjectClassID: 0
    MeasurementParameters: [1x1 struct]
         ObjectAttributes: {}
```

```
filter = initcvukf(detection);
```

Display filter state vector.

```
disp(filter.State)
```

```
  732.1068
   -2.8284
  667.1068
    2.1716
         0
         0
```

Input Arguments

detection — Detection report
objectDetection object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

Output Arguments

filter — Unscented Kalman filter
trackingUKF object

Unscented Kalman filter, returned as a `trackingUKF` object.

Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration standard deviation of 1 m/s^2.
- You can use this function as the `FilterInitializationFcn` property of a `multiObjectTracker` object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`initcaekf` | `initcakf` | `initcaukf` | `initctekf` | `initctukf` | `initcvekf` | `initcvkf`

Objects
`multiObjectTracker` | `objectDetection` | `trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

insertLaneBoundary

Insert lane boundary into image

Syntax

```
rgb = insertLaneBoundary(I,boundaries,sensor,xVehicle)
rgb = insertLaneBoundary( ___ ,Name,Value)
```

Description

`rgb = insertLaneBoundary(I,boundaries,sensor,xVehicle)` inserts lane boundary markings into a truecolor image. The lanes are overlaid on the input road image, `I`. This image comes from the sensor specified in the `sensor` object. `xVehicle` specifies the *x*-coordinates at which to draw the lane markers. The *y*-coordinates are calculated based on the parameters of the boundary models in `boundaries`.

`rgb = insertLaneBoundary( ___ ,Name,Value)` inserts lane boundary markings with additional options specified by one or more `Name,Value` pair arguments, using the previous input arguments.

Examples

Find Parabolic Lane Boundaries in Bird's-Eye-View Image

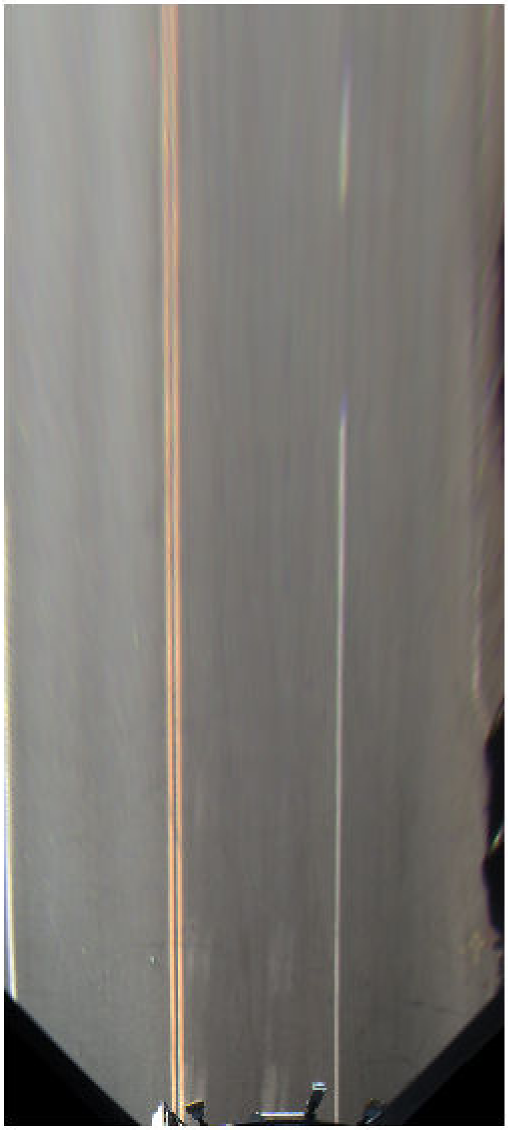
Find lanes in an image by using parabolic lane boundary models. Overlay the identified lanes on the original image and on a bird's-eye-view transformation of the image.

Load an image of a road with lanes. The image was obtained from a camera sensor mounted on the front of a vehicle.

```
I = imread('road.png');
```

Transform the image into a bird's-eye-view image by using a preconfigured sensor object. This object models the sensor that captured the original image.

```
bevSensor = load('birdsEyeConfig');
birdsEyeImage = transformImage(bevSensor.birdsEyeConfig,I);
imshow(birdsEyeImage)
```

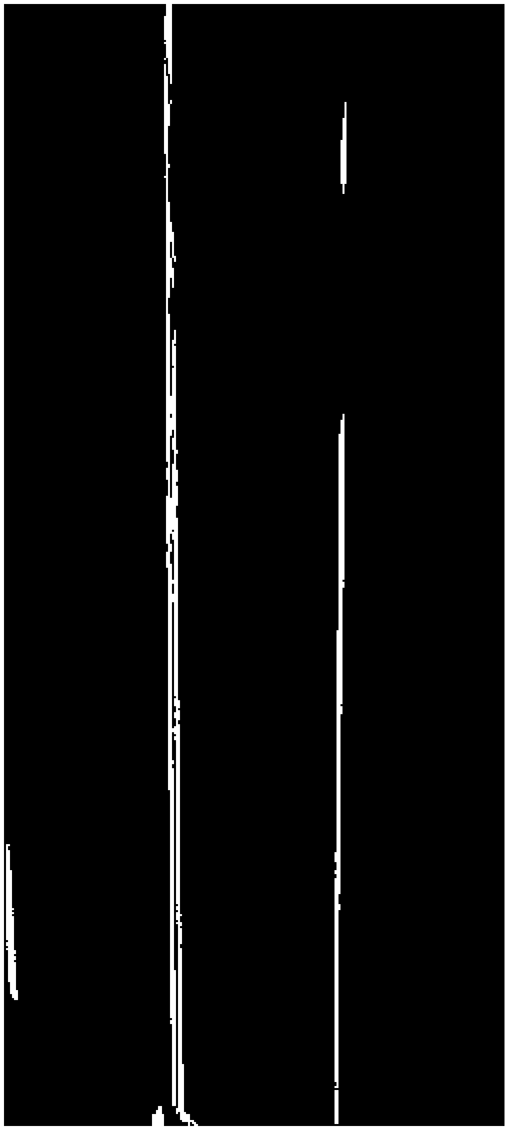


Set the approximate lane marker width in world units (meters).

```
approxBoundaryWidth = 0.25;
```

Detect lane features and display them as a black-and-white image.

```
birdsEyeBW = segmentLaneMarkerRidge(rgb2gray(birdsEyeImage), ...
    bevSensor.birdsEyeConfig,approxBoundaryWidth);
imshow(birdsEyeBW)
```



Obtain lane candidate points in world coordinates.

```
[imageX,imageY] = find(birdsEyeBW);
xyBoundaryPoints = imageToVehicle(bevSensor.birdsEyeConfig,[imageY,imageX]);
```

Find lane boundaries in the image by using the `findParabolicLaneBoundaries` function. By default, the function returns a maximum of two lane boundaries. The boundaries are stored in an array of `parabolicLaneBoundary` objects.

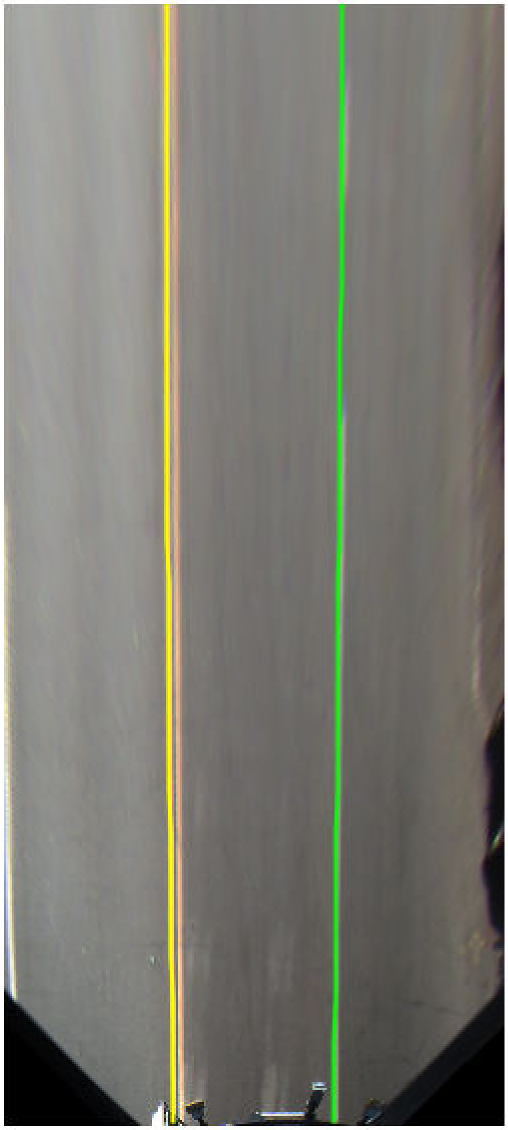
```
boundaries = findParabolicLaneBoundaries(xyBoundaryPoints,approxBoundaryWidth);
```

Use `insertLaneBoundary` to overlay the lanes on the original image. The `XPoints` vector represents the lane points, in meters, that are within range of the ego vehicle's sensor. Specify the lanes in different colors. By default, lanes are yellow.

```
XPoints = 3:30;

figure
sensor = bevSensor.birdsEyeConfig.Sensor;
lanesI = insertLaneBoundary(I,boundaries(1),sensor,XPoints);
lanesI = insertLaneBoundary(lanesI,boundaries(2),sensor,XPoints,'Color','green');
imshow(lanesI)
```

View the lanes in the bird's-eye-view image.

```
figure
BEconfig = bevSensor.birdsEyeConfig;
lanesBEI = insertLaneBoundary(birdsEyeImage,boundaries(1),BEconfig,XPoints);
lanesBEI = insertLaneBoundary(lanesBEI,boundaries(2),BEconfig,XPoints,'Color','green');
imshow(lanesBEI)
```



Input Arguments

I — Input road image
truecolor image | grayscale image

Input road image, specified as a truecolor or grayscale image.

Data Types: `single` | `double` | `int8` | `int16` | `uint8` | `uint16`

boundaries — Lane boundary models
array of parabolicLaneBoundary objects | array of cubicLaneBoundary objects

Lane boundary models, specified as an array of parabolicLaneBoundary objects or cubicLaneBoundary objects. Lane boundary models contain the following properties:

- Parameters — A vector corresponding to the coefficients of the boundary model. The size of the vector depends on the degree of polynomial for the model.

Lane Boundary Object	Parameters
parabolicLaneBoundary	[A B C], corresponding to coefficients of a second-degree polynomial equation of the form $y = Ax^2 + Bx + C$
cubicLaneBoundary	[A B C D], corresponding to coefficients of a third-degree polynomial equation of the form $y = Ax^3 + Bx^2 + Cx + D$

- BoundaryType — A LaneBoundaryType enumeration of supported lane boundaries:
  - Unmarked
  - Solid
  - Dashed
  - BottsDots
  - DoubleSolid

  Specify a lane boundary type as LaneBoundaryType.*BoundaryType*. For example:

  ```
  LaneBoundaryType.BottsDots
  ```
- Strength — The ratio of the number of unique *x*-axis locations on the boundary to the total number of points along the line based on the XExtent property.
- XExtent — A two-element vector describing the minimum and maximum *x*-axis locations for the boundary points.

sensor — Sensor that collects images
birdsEyeView object | monoCamera object

Sensor that collects images, specified as either a birdsEyeView or monoCamera object.

xVehicle — *x*-axis locations of boundary
real-valued vector

x-axis locations at which to display the lane boundaries, specified as a real-valued vector in vehicle coordinates. The spacing between points controls the spacing between dashes and dots for the corresponding types of boundaries. To show dashed boundaries clearly, specify at least four points in xVehicle. If you specify fewer than four points, the function draws a solid boundary.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Color',[0 1 0]

`Color` — Color of lane boundaries
`'yellow'` (default) | character vector | string scalar | `[R,G,B]` vector of RGB values | cell array of character vectors | string array | *m*-by-3 matrix of RGB values

Color of lane boundaries, specified as a character vector, string scalar, or `[R,G,B]` vector of RGB values. You can specify specific colors for each boundary in `boundaries` with a cell array of character vectors, a string array, or an *m*-by-3 matrix of RGB values. The colors correspond to the order of the boundary lanes.

RGB values must be in the range of the image data type.

Supported color values are `'blue'`, `'green'`, `'red'`, `'cyan'`, `'magenta'`, `'yellow'`, `'black'`, and `'white'`.

Example: `'red'`

Example: `[1,0,0]`

`LineWidth` — Line width for boundary lanes
`3` (default) | positive integer

Line width for boundary lanes, specified as a positive integer in pixels.

Output Arguments

`rgb` — Image with boundary lanes
RGB truecolor image

Image with boundary lanes overlaid, returned as an RGB truecolor image. The output image class matches the input image, `I`.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`birdsEyeView` | `cubicLaneBoundary` | `fitPolynomialRANSAC` | `monoCamera` | `parabolicLaneBoundary`

Introduced in R2017a

lateralControllerStanley

Compute steering angle command for path following by using Stanley method

Syntax

```
steerCmd = lateralControllerStanley(refPose,currPose,currVelocity)
steerCmd = lateralControllerStanley(refPose,currPose,currVelocity,Name,Value)
```

Description

`steerCmd = lateralControllerStanley(refPose,currPose,currVelocity)` computes the steering angle command, in degrees, that adjusts the current pose of a vehicle to match a reference pose, given the current velocity of the vehicle. By default, the function assumes that the vehicle is in forward motion.

The controller computes the steering angle command using the Stanley method [1], whose control law is based on a kinematic bicycle model. Use this controller for path following in low-speed environments, where inertial effects are minimal.

`steerCmd = lateralControllerStanley(refPose,currPose,currVelocity,Name,Value)` specifies options using one or more name-value pairs. For example, `lateralControllerStanley(refPose,currPose,currVelocity,'Direction',-1)` computes the steering angle command for a vehicle in reverse motion.

Examples

Steering Angle Command for Vehicle in Forward Motion

Compute the steering angle command that adjusts the current pose of a vehicle to a reference pose along a driving path. The vehicle is in forward motion.

In this example, you compute a single steering angle command. In path-following algorithms, compute the steering angle continuously as the pose and velocity of the vehicle change.

Set a reference pose on the path. The pose is at position (4.8 m, 6.5 m) and has an orientation angle of 2 degrees.

```
refPose = [4.8, 6.5, 2]; % [meters, meters, degrees]
```

Set the current pose of the vehicle. The pose is at position (2 m, 6.5 m) and has an orientation angle of 0 degrees. Set the current velocity of the vehicle to 2 meters per second.

```
currPose = [2, 6.5, 0]; % [meters, meters, degrees]
currVelocity = 2; % meters per second
```

Compute the steering angle command. For the vehicle to match the reference pose, the steering wheel must turn 2 degrees counterclockwise.

```
steerCmd = lateralControllerStanley(refPose,currPose,currVelocity)
```

```
steerCmd = 2.0000
```

Steering Angle Command for Vehicle in Reverse Motion

Compute the steering angle command that adjusts the current pose of a vehicle to a reference pose along a driving path. The vehicle is in reverse motion.

In this example, you compute a single steering angle command. In path-following algorithms, compute the steering angle continuously as the pose and velocity of the vehicle change.

Set a reference pose on the path. The pose is at position (5 m, 9 m) and has an orientation angle of 90 degrees.

```
refPose = [5, 9, 90]; % [meters, meters, degrees]
```

Set the current pose of the vehicle. The pose is at position (5 m, 10 m) and has an orientation angle of 75 degrees.

```
currPose = [5, 10, 75]; % [meters, meters, degrees]
```

Set the current velocity of the vehicle to –2 meters per second. Because the vehicle is in reverse motion, the velocity must be negative.

```
currVelocity = -2; % meters per second
```

Compute the steering angle command. For the vehicle to match the reference pose, the steering wheel must turn 15 degrees clockwise.

```
steerCmd = lateralControllerStanley(refPose,currPose,currVelocity,'Direction',-1)
```

```
steerCmd = -15.0000
```

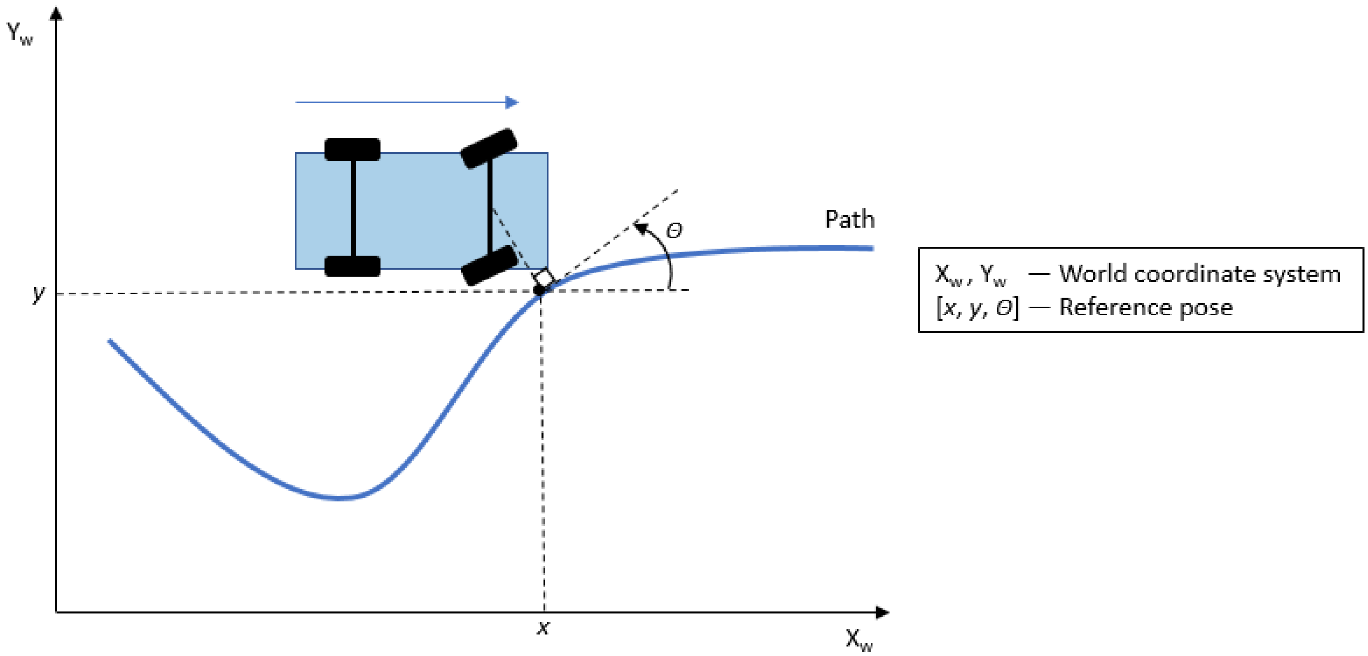
Input Arguments

`refPose` — Reference pose
[*x*, *y*, *Θ*] vector

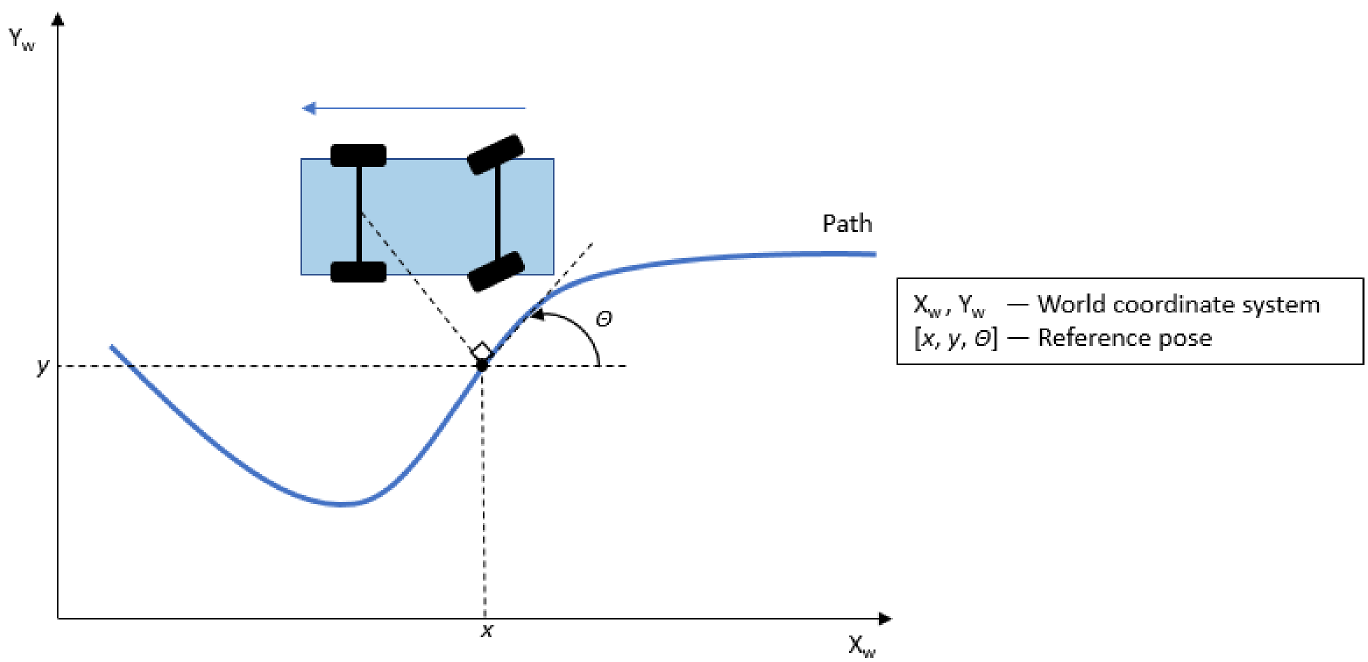
Reference pose, specified as an [*x*, *y*, *Θ*] vector. *x* and *y* are in meters, and *Θ* is in degrees.

x and *y* specify the reference point to steer the vehicle toward. *Θ* specifies the orientation angle of the path at this reference point and is positive in the counterclockwise direction.

- For a vehicle in forward motion, the reference point is the point on the path that is closest to the center of the vehicle's front axle.



- For a vehicle in reverse motion, the reference point is the point on the path that is closest to the center of the vehicle's rear axle.



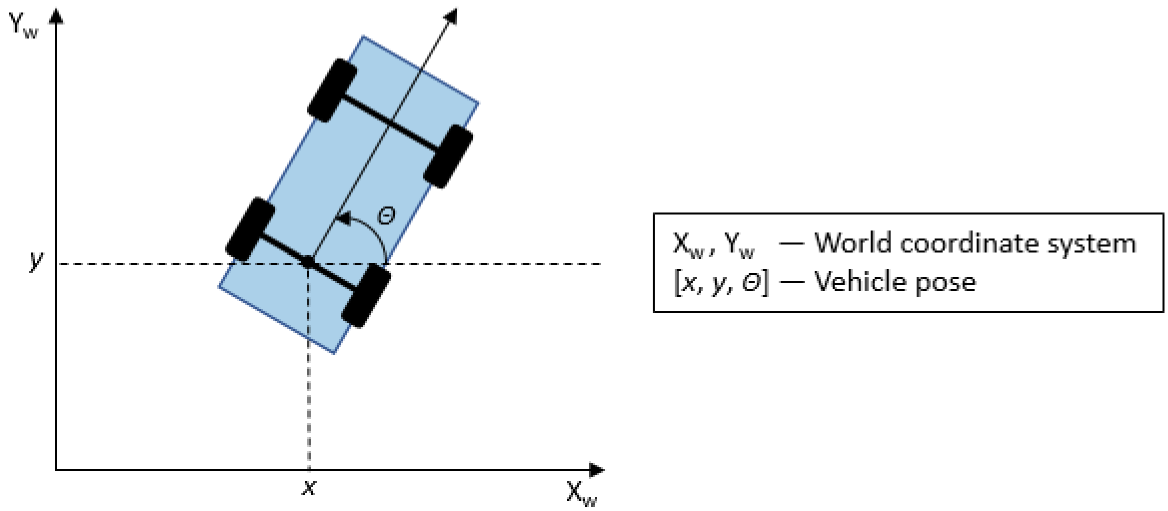
Data Types: `single` | `double`

`currPose` — Current pose
[x, y, Θ] vector

Current pose of the vehicle, specified as an [*x*, *y*, *Θ*] vector. *x* and *y* are in meters, and *Θ* is in degrees.

x and *y* specify the location of the vehicle, which is defined as the center of the vehicle's rear axle.

Θ specifies the orientation angle of the vehicle at location (*x*,*y*) and is positive in the counterclockwise direction.



For more details on vehicle pose, see “Coordinate Systems in Automated Driving Toolbox”.

Data Types: `single` | `double`

`currVelocity` — Current longitudinal velocity
real scalar

Current longitudinal velocity of the vehicle, specified as a real scalar. Units are in meters per second.

- If the vehicle is in forward motion, then this value must be greater than 0.
- If the vehicle is in reverse motion, then this value must be less than 0.
- A value of 0 represents a vehicle that is not in motion.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MaxSteeringAngle',25`

`Direction` — Driving direction of vehicle
`1` (forward motion) (default) | `-1` (reverse motion)

Driving direction of the vehicle, specified as the comma-separated pair consisting of `'Direction'` and either `1` for forward motion or `-1` for reverse motion. The driving direction determines the position error and angle error used to compute the steering angle command. For more details, see “Algorithms” on page 3-169.

PositionGain — Position gain
2.5 (default) | positive real scalar

Position gain of the vehicle, specified as the comma-separated pair consisting of 'PositionGain' and a positive real scalar. This value determines how much the position error affects the steering angle. Typical values are in the range [1, 5]. Increase this value to increase the magnitude of the steering angle.

Wheelbase — Distance between front and rear axles of vehicle
2.8 (default) | real scalar

Distance between the front and rear axles of the vehicle, in meters, specified as the comma-separated pair consisting of 'Wheelbase' and a real scalar. This value applies only when the vehicle is in forward motion.

MaxSteeringAngle — Maximum allowed steering angle
35 (default) | real scalar in the range (0, 180)

Maximum allowed steering angle of the vehicle, in degrees, specified as the comma-separated pair consisting of 'MaxSteeringAngle' and a real scalar in the range (0, 180).

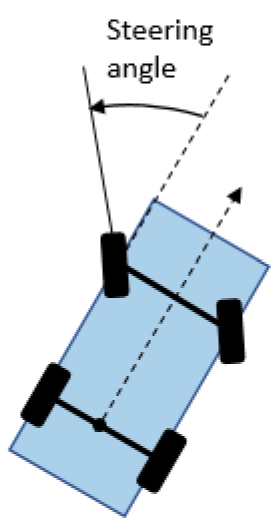
The steerCmd value is saturated to the range [-MaxSteeringAngle, MaxSteeringAngle].

- Values below -MaxSteeringAngle are set to -MaxSteeringAngle.
- Values above MaxSteeringAngle are set to MaxSteeringAngle.

Output Arguments

steerCmd — Steering angle command
real scalar

Steering angle command, in degrees, returned as a real scalar. This value is positive in the counterclockwise direction.



For more details, see "Coordinate Systems in Automated Driving Toolbox".

Algorithms

To compute the steering angle command, the controller minimizes the position error and the angle error of the current pose with respect to the reference pose. The driving direction of the vehicle determines these error values.

When the vehicle is in forward motion (`'Direction'` name-value pair is `1`):

- The position error is the lateral distance from the center of the front axle to the reference point on the path.
- The angle error is the angle of the front wheel with respect to reference path.

When the vehicle is in reverse motion (`'Direction'` name-value pair is `-1`):

- The position error is the lateral distance from the center of the rear axle to the reference point on the path.
- The angle error is the angle of the rear wheel with respect to reference path.

For details on how the controller minimizes these errors, see [1].

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. doi:10.1109/ACC.2007.4282788

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Blocks
Lateral Controller Stanley | Longitudinal Controller Stanley

Objects
`pathPlannerRRT`

Topics
"Automated Parking Valet"
"Coordinate Systems in Automated Driving Toolbox"

Introduced in R2018b

removeCustomBasemap

Remove custom basemap

Syntax

```
removeCustomBasemap(basemapName)
```

Description

`removeCustomBasemap(basemapName)` removes the custom basemap specified by `basemapName` from the list of available basemaps.

If the custom basemap specified by `basemapName` has not been previously added using the `addCustomBasemap` function, the `removeCustomBasemap` function returns an error.

Examples

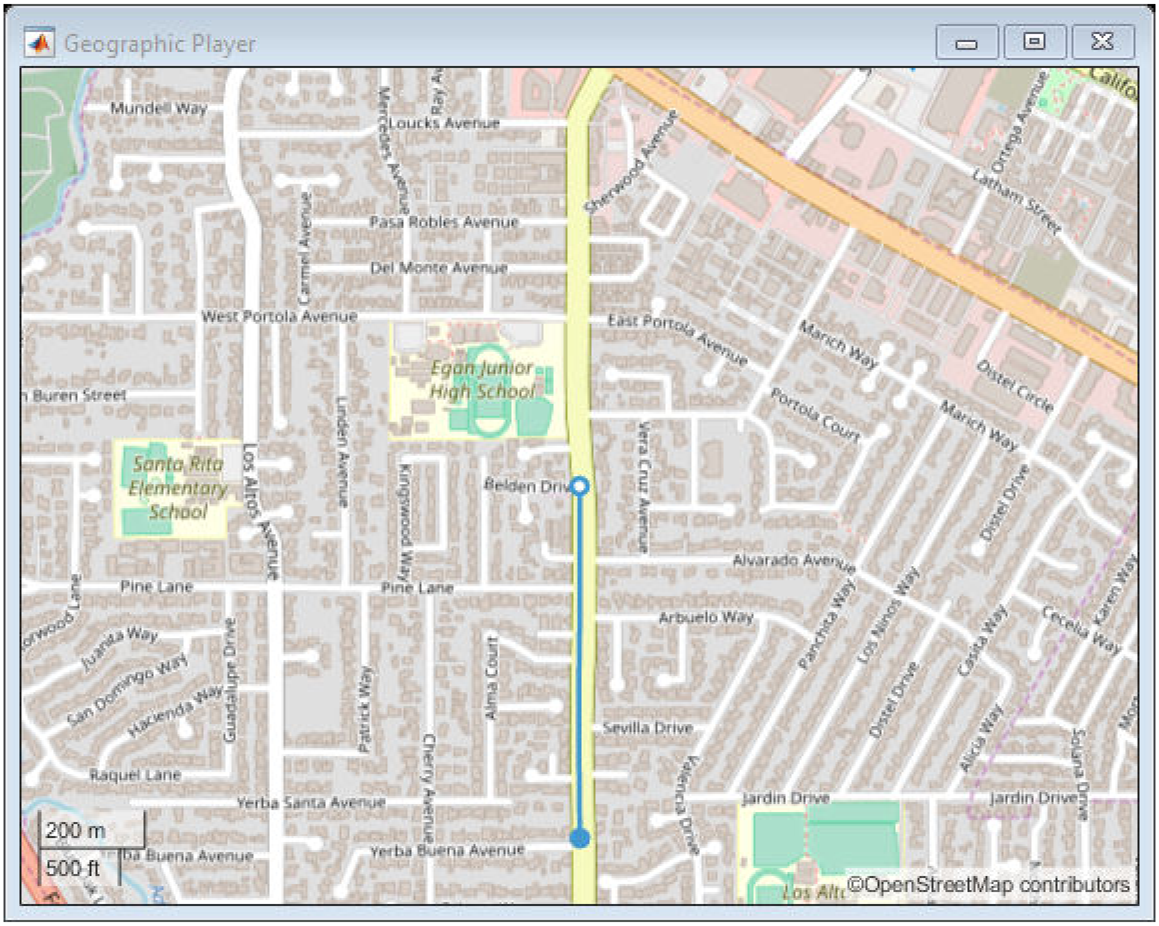
Remove Custom Basemap

Add a custom basemap to view locations on an OpenStreetMap® basemap.

```
name = 'openstreetmap';
url = 'a.tile.openstreetmap.org';
copyright = char(uint8(169));
attribution = copyright + "OpenStreetMap contributors";
addCustomBasemap(name,url,'Attribution',attribution)
```

Use the custom basemap with a geographic player.

```
data = load('geoSequence.mat');
player = geoplayer(data.latitude(1),data.longitude(1),'Basemap',name);
plotRoute(player,data.latitude,data.longitude);
```



Remove the custom basemap. The custom basemap associated with the specified name remains stored in this geographic player. However, this basemap is no longer available for use with new players.

```
removeCustomBasemap(name)
```

Input Arguments

basemapName — Name of custom basemap
string scalar | character vector

Name of the custom basemap to remove, specified as a string scalar or character vector. You define the basemap name when you add the basemap using the `addCustomBasemap` function.

Data Types: `string` | `char`

See Also

`addCustomBasemap` | `geoaxes` | `geobasemap` | `geobubble` | `geodensityplot` | `geoplayer` | `geoplot` | `geoscatter`

Introduced in R2019a

segmentLaneMarkerRidge

Detect lanes in a grayscale intensity image

Syntax

```
birdsEyeBW = segmentLaneMarkerRidge(birdsEyeImage,birdsEyeConfig,
approxMarkerWidth)
birdsEyeBW = segmentLaneMarkerRidge( ___ ,Name,Value)
```

Description

`birdsEyeBW = segmentLaneMarkerRidge(birdsEyeImage,birdsEyeConfig, approxMarkerWidth)` returns a binary image that represents lane features. The function segments the input grayscale intensity image, `birdsEyeImage`, using a lane ridge detector. `birdsEyeConfig` transforms point locations from vehicle coordinates to image coordinates. The `approxMarkerWidth` argument is in world units, and specifies the approximate width of the lane-like features that are detected.

`birdsEyeBW = segmentLaneMarkerRidge( ___ ,Name,Value)` returns a binary image with additional options specified by one or more `Name,Value` pair arguments.

Examples

Detect Lanes in Road Image

Load a bird's-eye-view configuration object.

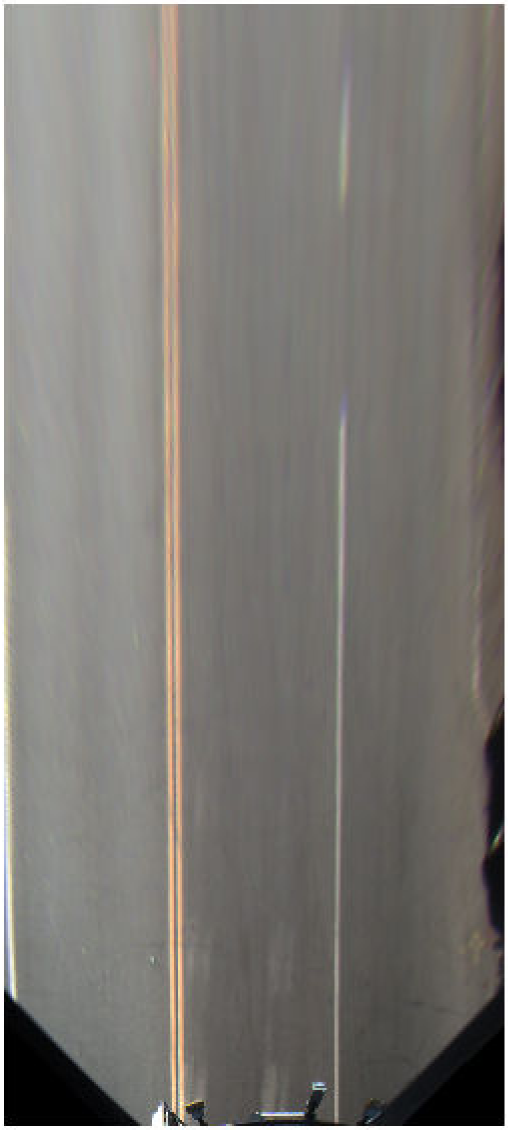
```
load birdsEyeConfig
```

Load the image captured from the sensor that is defined in the bird's-eye-view configuration object.

```
I = imread('road.png');
figure
imshow(I)
title('Original Image')
```

Create a bird's-eye-view image.

```
birdsEyeImage = transformImage(birdsEyeConfig,I);
imshow(birdsEyeImage)
```



Convert bird's-eye-view image to grayscale.

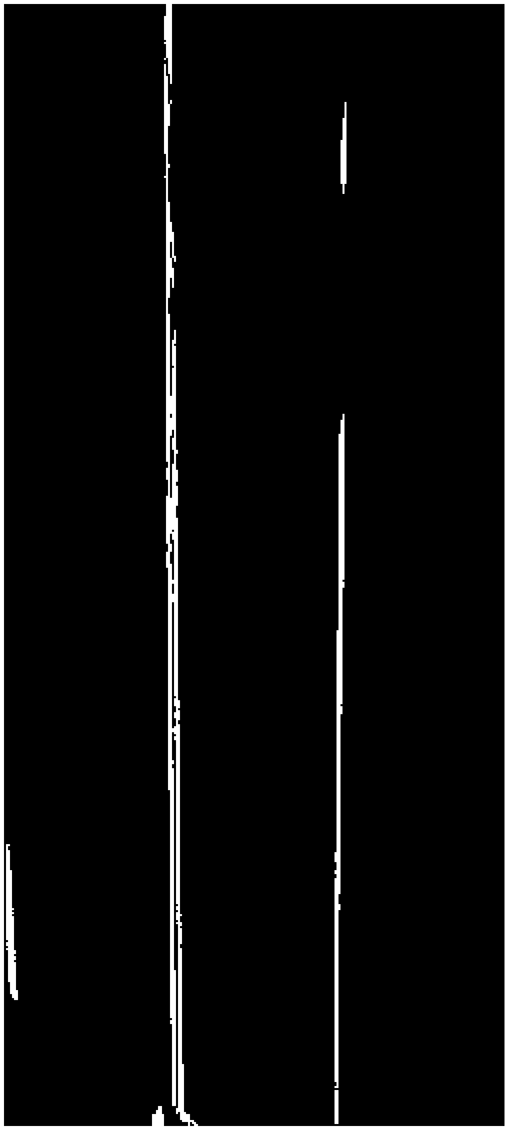
```
birdsEyeImage = rgb2gray(birdsEyeImage);
```

Set the approximate lane marker width to 25 cm, which is in world units.

```
approxMarkerWidth = 0.25;
```

Detect lane features.

```
birdsEyeBW = segmentLaneMarkerRidge(birdsEyeImage,birdsEyeConfig,approxMarkerWidth);
imshow(birdsEyeBW)
```



Input Arguments

`birdsEyeImage` — Bird’s-eye-view image
matrix

Bird’s-eye-view image, specified as a nonsparse matrix.

Data Types: `single` | `int16` | `uint16` | `uint8`

`birdsEyeConfig` — Object to transform point locations
`birdsEyeView` object

Object to transform point locations from vehicle to image coordinates, specified as a `birdsEyeView` object.

`approxMarkerWidth` — Approximate width of lane-like features
real scalar in world units

Approximate width of lane-like features for the function to detect in the bird's-eye-view image, specified as a real scalar in world units, such as meters.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'ROI' []`

`ROI` — Region of interest
`[]` (default) | world units

Region of interest in world units, specified as the comma-separated pair consisting of `'ROI'` and a 1-by-4 vector in the format [*xmin*,*xmax*,*ymin*,*ymax*]. The function searches for lane-like features only within this region of interest. If you do not specify `ROI`, the function searches the entire image.

`Sensitivity` — Sensitivity factor
`0.25` (default) | real scalar in the range [0, 1]

Sensitivity factor, specified as the comma-separated pair consisting of `'Sensitivity'` and a real scalar in the range [0, 1]. You can increase this value to detect more lane-like features. However, the higher sensitivity can increase the risk of false detections.

Output Arguments

`birdsEyeBW` — Bird's-eye-view image
binary image

Bird's-eye-view image, returned as a binary image that represents lane features.

More About

Vehicle Coordinate System

This function uses a vehicle coordinate system to define point locations, as defined by the sensor in the `birdsEyeView` object. It uses the same world units as defined by the `birdsEyeConfig.Sensor.WorldUnits` property. See "Coordinate Systems in Automated Driving Toolbox".

Algorithms

`segmentLaneMarkerRidge` selects lanes by searching for pixels that are lane-like. Lane-like pixels are groups of pixels with high-intensity contrast compared to neighboring pixels on either side. The function chooses the filter used to threshold the intensity contrast based on the approxMarkerWidth value. The filter has high responses for pixels with intensity values higher than those of the left and right neighboring pixels that have a similar intensity at a distance of approxMarkerWidth. The function retains only certain values from the filtered image based on the Sensitivity factor.

References

[1] Nieto, M., J. A. Laborda, and L. Salgado. “Road Environment Modeling Using Robust Perspective Analysis and Recursive Bayesian Segmentation.” *Machine Vision and Applications.* Volume 22, Issue 6, 2011, pp. 927–945.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
birdsEyeView

Introduced in R2017a

smoothPathSpline

Smooth vehicle path using cubic spline interpolation

Syntax

```
[poses,directions] = smoothPathSpline(refPoses,refDirections,numSmoothPoses)
[poses,directions] = smoothPathSpline(refPoses,refDirections,numSmoothPoses,
minSeparation)
[ ___ ,cumLengths,curvatures] = smoothPathSpline( ___ )
```

Description

`[poses,directions] = smoothPathSpline(refPoses,refDirections,numSmoothPoses)` generates a smooth vehicle path, consisting of `numSmoothPoses` discretized poses, by fitting the input reference path poses to a cubic spline. Given the input reference path directions, `smoothPathSpline` also returns the directions that correspond to each pose.

Use this function to convert a C^1-continuous vehicle path to a C^2-continuous path. C^1-continuous paths include the `driving.DubinsPathSegment` or `driving.ReedsSheppPathSegment` paths that you can plan using a `pathPlannerRRT` object. For more details on these path types, see “C1-Continuous and C2-Continuous Paths” on page 3-183.

You can use the returned poses and directions with a vehicle controller, such as the `lateralControllerStanley` function.

`[poses,directions] = smoothPathSpline(refPoses,refDirections,numSmoothPoses, minSeparation)` specifies a minimum separation threshold between poses. If the distance between two poses is smaller than `minSeparation`, the function uses only one of the poses for interpolation.

`[ ___ ,cumLengths,curvatures] = smoothPathSpline( ___ )` also returns the cumulative path length and signed path curvature at each returned pose, using any of the previous syntaxes. Use these values to generate a velocity profile along the path.

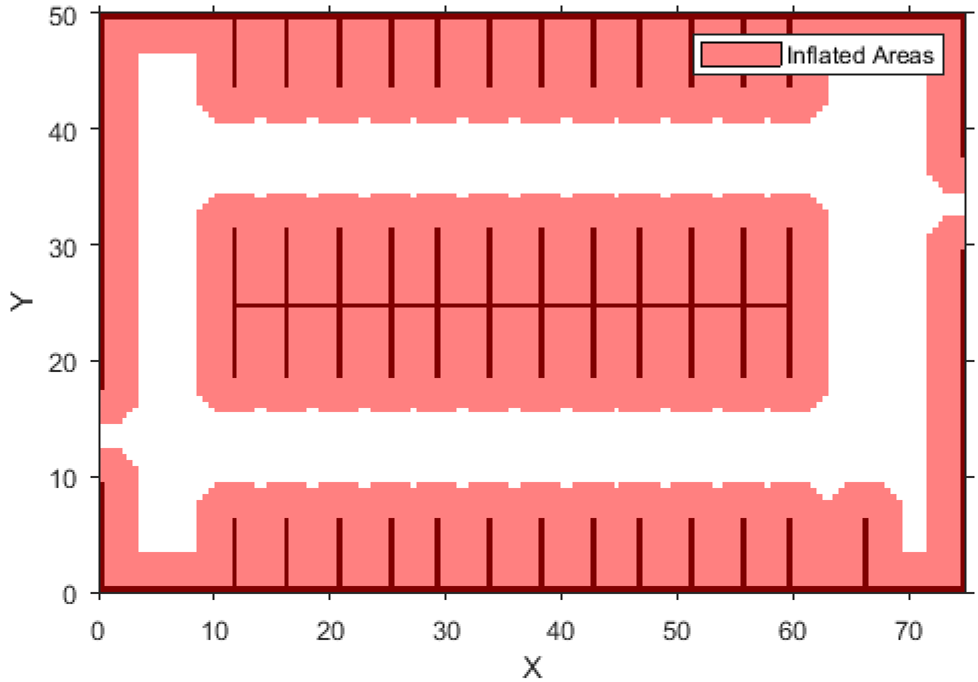
Examples

Smooth a Planned Path

Smooth a path that was planned by an RRT* path planner.

Load and plot a costmap of a parking lot.

```
data = load('parkingLotCostmap.mat');
costmap = data.parkingLotCostmap;
plot(costmap)
```



Define start and goal poses for the vehicle as [*x*, *y*, *Θ*] vectors. World units for the (*x*,*y*) locations are in meters. World units for the *Θ* orientation angles are in degrees.

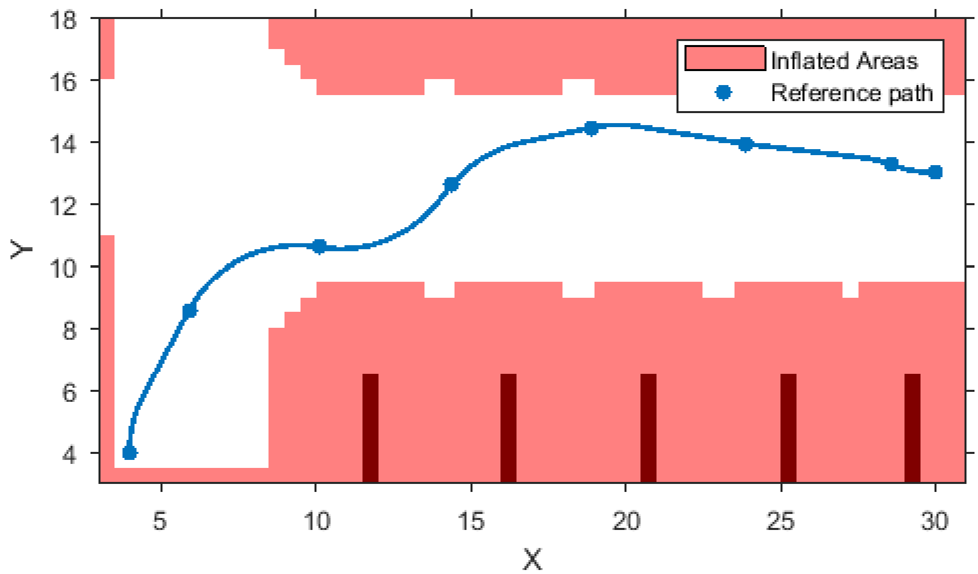
```
startPose = [4,4,90]; % [meters, meters, degrees]
goalPose = [30,13,0];
```

Use a `pathPlannerRRT` object to plan a path from the start pose to the goal pose.

```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Plot and zoom in on the planned path. The path is composed of a sequence of Dubins curves. These curves include abrupt changes in curvature that are not suitable for driving with passengers.

```
hold on
plot(refPath,'Vehicle','off','DisplayName','Reference path')
xlim([3 31])
ylim([3 18])
```



Interpolate the transition poses of the path. Use these poses as the reference poses for interpolating the smooth path. Also return the motion directions at each pose.

```
[refPoses,refDirections] = interpolate(refPath);
```

Specify the number of poses to return in the smooth path. Return poses spaced about 0.1 meters apart, along the entire length of the path.

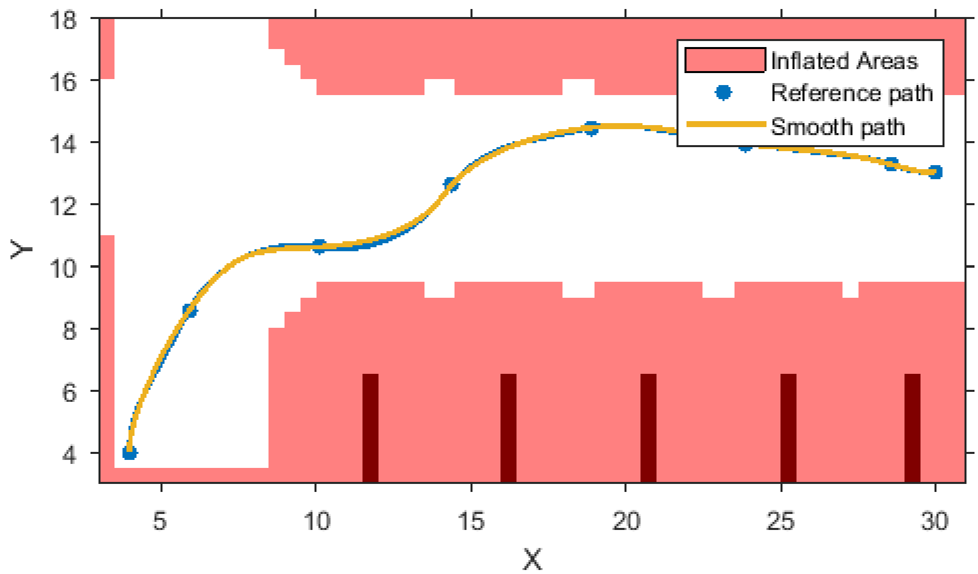
```
approxSeparation = 0.1; % meters
numSmoothPoses = round(refPath.Length / approxSeparation);
```

Generate the smooth path by fitting a cubic spline to the reference poses. `smoothPathSpline` returns the specified number of discretized poses along the smooth path.

```
[poses,directions] = smoothPathSpline(refPoses,refDirections,numSmoothPoses);
```

Plot the smooth path. The more abrupt changes in curvature that were present in the reference path are now smoothed out.

```
plot(poses(:,1),poses(:,2),'LineWidth',2,'DisplayName','Smooth path')
hold off
```



Input Arguments

refPoses — Reference poses
M-by-3 matrix of [*x*, *y*, *Θ*] vectors

Reference poses of the vehicle along the path, specified as an *M*-by-3 matrix of [*x*, *y*, *Θ*] vectors, where *M* is the number of poses.

x and *y* specify the location of the vehicle in meters. *Θ* specifies the orientation angle of the vehicle in degrees.

Data Types: `single` | `double`

refDirections — Reference directions
M-by-1 column vector of 1s (forward motion) and –1s (reverse motion)

Reference directions of the vehicle along the path, specified as an *M*-by-1 column vector of 1s (forward motion) and –1s (reverse motion). *M* is the number of reference directions. Each element of `refDirections` corresponds to a pose in the `refPoses` input argument.

Data Types: `single` | `double`

numSmoothPoses — Number of smooth poses to return
positive integer

Number of smooth poses to return in the `poses` output argument, specified as a positive integer. To increase the granularity of the returned poses, increase `numSmoothPoses`.

`minSeparation` — Minimum separation between poses

`1e-3` (default) | positive real scalar

Minimum separation between poses, in meters, specified as a positive real scalar. If the Euclidean (*x*, *y*) distance between two poses is less than this value, then the function uses only one of these poses for interpolation.

Output Arguments

`poses` — Discretized poses of smoothed path

`numSmoothPoses`-by-3 matrix of [*x*, *y*, *Θ*] vectors

Discretized poses of the smoothed path, returned as a `numSmoothPoses`-by-3 matrix of [*x*, *y*, *Θ*] vectors.

x and *y* specify the location of the vehicle in meters. *Θ* specifies the orientation angle of the vehicle in degrees.

The values in `poses` are of the same data type as the values in the `refPoses` input argument.

`directions` — Motion directions at each output pose

`numSmoothPoses`-by-1 column vector of 1s (forward motion) and –1s (reverse motion)

Motion directions at each output pose in `poses`, returned as a `numSmoothPoses`-by-1 column vector of 1s (forward motion) and –1s (reverse motion).

The values in `directions` are of the same data type as the values in the `refDirections` input argument.

`cumLengths` — Cumulative path lengths

`numSmoothPoses`-by-1 real-valued column vector

Cumulative path length at each output pose in `poses`, returned as a `numSmoothPoses`-by-1 real-valued column vector. Units are in meters.

You can use the `cumLengths` and `curvatures` outputs to generate a velocity profile of the vehicle along the smooth path. For more details, see the “Automated Parking Valet” example.

`curvatures` — Signed path curvatures

`numSmoothPoses`-by-1 real-valued column vector

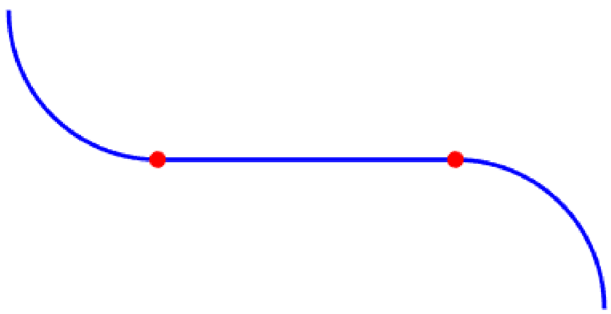
Signed path curvatures at each output pose in `poses`, returned as a `numSmoothPoses`-by-1 real-valued column vector. Units are in radians per meter.

You can use the `curvatures` and `cumLengths` outputs to generate a velocity profile of the vehicle along the smooth path. For more details, see the “Automated Parking Valet” example.

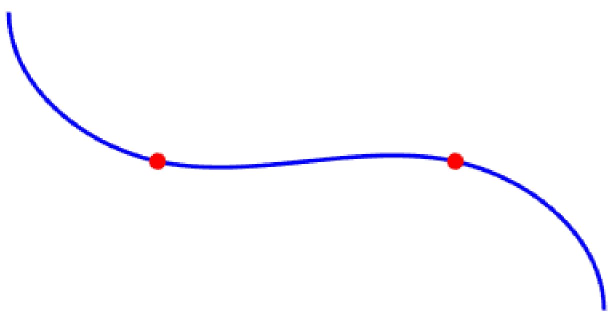
More About

C^1-Continuous and C^2-Continuous Paths

A path is C^1-continuous if its derivative exists and is continuous. Paths that are only C^1-continuous have discontinuities in their curvature. For example, a path composed of Dubins or Reeds-Sheep path segments has discontinuities in curvature at the points where the segments join. These discontinuities result in changes in direction that are not smooth enough for driving with passengers.



A path is also C^2-continuous if its second derivative exists and is continuous. C^2-continuous paths have continuous curvature and are smooth enough for driving with passengers.



Tips

- To check if a smooth path is collision-free, specify the smooth poses as an input to the `checkPathValidity` function.

Algorithms

- The path-smoothing algorithm interpolates a parametric cubic spline that passes through all input reference pose points. The parameter of the spline is the cumulative chord length at these points. [1]
- The tangent direction of the smoothed output path approximately matches the orientation angle of the vehicle at the starting and goal poses.

References

[1] Floater, Michael S. "On the Deviation of a Parametric Cubic Spline Interpolant from Its Data Polygon." *Computer Aided Geometric Design*. Vol. 25, Number 3, 2008, pp. 148–156.

[2] Lepetic, Marko, Gregor Klancar, Igor Skrjanc, Drago Matko, and Bostjan Potocnik. "Time Optimal Path Planning Considering Acceleration Limits." *Robotics and Autonomous Systems*. Vol. 45, Numbers 3–4, 2003, pp. 199–210.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`checkPathValidity` | `driving.Path` | `interpolate` | `lateralControllerStanley` | `pathPlannerRRT` | `spline`

Blocks
Path Smoother Spline

Topics
"Automated Parking Valet"

Introduced in R2019a

vehicleDetectorACF

Load vehicle detector using aggregate channel features

Syntax

```
detector = vehicleDetectorACF
detector = vehicleDetectorACF(modelName)
```

Description

`detector = vehicleDetectorACF` returns a pretrained vehicle detector using aggregate channel features (ACF). The returned `acfObjectDetector` object is trained using unoccluded images of the front, rear, left, and right sides of the vehicles.

`detector = vehicleDetectorACF(modelName)` returns a pretrained vehicle detector based on the model specified in `modelName`. A `'full-view'` model uses training images that are unoccluded views from the front, rear, left, and right sides of vehicles. A `'front-rear-view'` model uses images only from the front and rear sides of the vehicle.

Examples

Detect Vehicles in Image

Load the pre-trained detector for vehicles

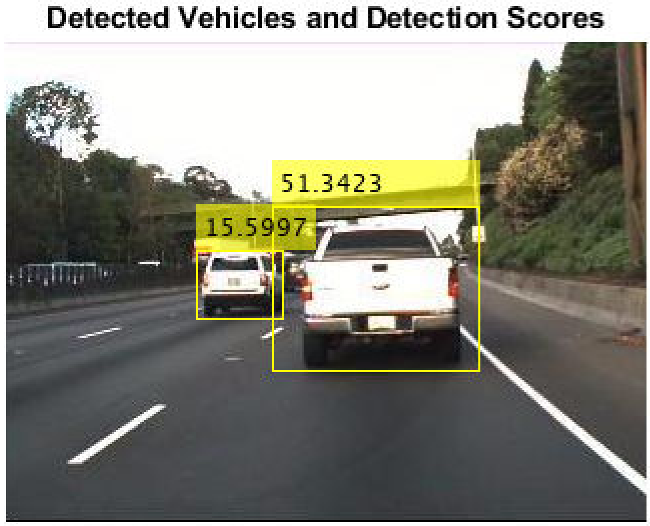
```
detector = vehicleDetectorACF('front-rear-view');
```

Load an image and run the detector.

```
I = imread('highway.png');
[bboxes,scores] = detect(detector,I);
```

Overlay bounding boxes and scores for vehicles detected in the image.

```
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I)
title('Detected Vehicles and Detection Scores')
```



Input Arguments

modelName — Type of vehicle detector model
'full-view' (default) | 'front-rear-view'

Type of vehicle detector model, specified as either 'front-rear-view' or 'full-view'. A 'full-view' model uses training images that are unoccluded views from the front, rear, left, and right sides of vehicles. A 'front-rear-view' model uses images only from the front and rear sides of the vehicle.

Data Types: char | string

Output Arguments

detector — Trained ACF-based object detector
acfObjectDetector object

Trained ACF-based object detector, returned as an acfObjectDetector object.

See Also

acfObjectDetector | trainACFObjectDetector

Introduced in R2017a

vehicleDetectorFasterRCNN

Detect vehicles using Faster R-CNN

Syntax

`detector = vehicleDetectorFasterRCNN`

Description

`detector = vehicleDetectorFasterRCNN` returns a trained Faster R-CNN (regions with convolution neural networks) object detector for detecting vehicles. Faster R-CNN is a deep learning object detection framework that uses a convolutional neural network (CNN) for detection.

The detector is trained using unoccluded images of the front, rear, left, and right sides of vehicles. The CNN used with the vehicle detector uses a modified version of the MobileNet-v2 network architecture.

Use of this function requires Deep Learning Toolbox™.

Note The detector is trained using `uint8` images. Before using this detector, rescale the input images to the range [0, 255] by using `im2uint8` or `rescale`.

Examples

Detect Vehicles on Highway

Detect cars in a single image and annotate the image with the detection scores. To detect cars, use a Faster R-CNN object detector that was trained using images of vehicles.

Load the pretrained detector.

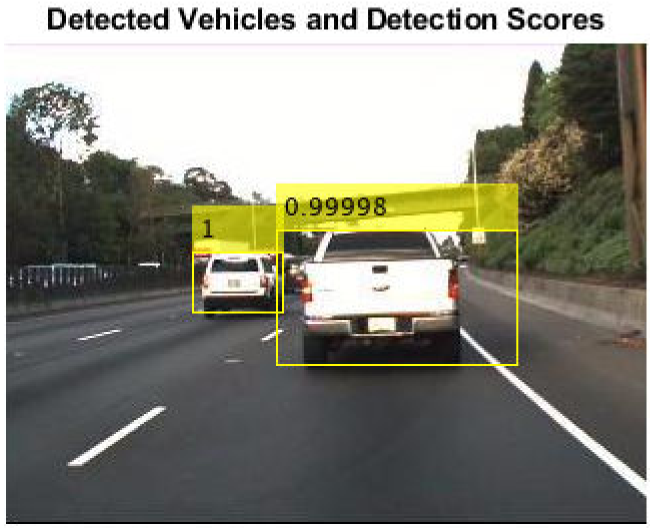
```
fasterRCNN = vehicleDetectorFasterRCNN('full-view');
```

Use the detector on a loaded image. Store the locations of the bounding boxes and their detection scores.

```
I = imread('highway.png');
[bboxes,scores] = detect(fasterRCNN,I);
```

Annotate the image with the detections and their scores.

```
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I)
title('Detected Vehicles and Detection Scores')
```



Output Arguments

`detector` — Trained Faster R-CNN-based object detector
`fasterRCNNObjectDetector` object

Trained Faster R-CNN-based object detector, returned as an `fasterRCNNObjectDetector` object.

Compatibility Considerations

`modelName` input argument is not recommended
Behavior change in future release

`modelName` input argument is not recommended. To update your code, remove all instances of `modelName`.

Discouraged Usage	Recommended Replacement
`modelName = 'front-rear-view'` `detector = vehicleDetectorFasterRCNN(modelName);`	`detector = vehicleDetectorFasterRCNN;`

See Also

`fasterRCNNObjectDetector` | `trainFasterRCNNObjectDetector` | `vehicleDetectorACF` | `vehicleDetectorYOLOv2`

Introduced in R2017a

vehicleDetectorYOLOv2

Detect vehicles using YOLO v2 Network

Syntax

```
detector = vehicleDetectorYOLOv2
```

Description

`detector = vehicleDetectorYOLOv2` returns a trained you only look once (YOLO) v2 object detector for detecting vehicles. YOLO v2 is a deep learning object detection framework that uses a convolutional neural network (CNN) for detection.

The detector is trained using unoccluded RGB images of the front, rear, left, and right sides of cars on a highway scene. The CNN used with the vehicle detector uses a modified version of the MobileNet-v2 network architecture. You can also fine tune the vehicle detector with additional training data by using the `trainYOLOv2ObjectDetector`.

For information about creating a YOLO v2 object detector, see "Create YOLO v2 Object Detection Network" (Computer Vision Toolbox). Use of this function requires Deep Learning Toolbox.

Examples

Detect Vehicles Using YOLO v2 Detector

This example shows how to detect cars in an image and annotate the image with the detection scores. To detect the cars, use a YOLO v2 detector that is trained to detect vehicles in an image.

Load the pretrained detector.

```
detector = vehicleDetectorYOLOv2();
```

Read a test image into the workspace.

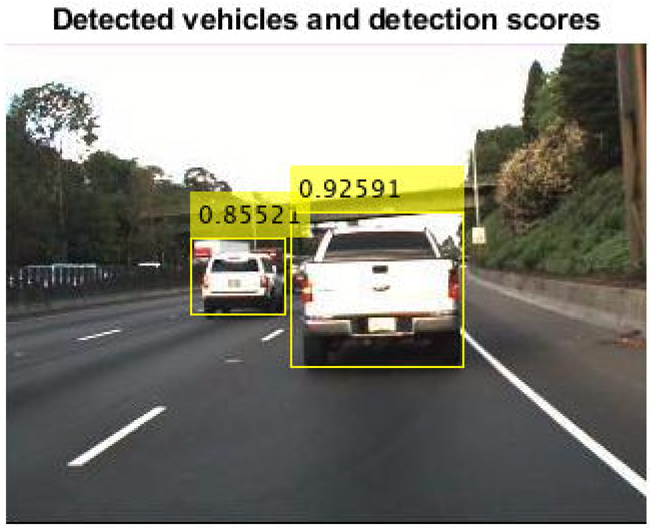
```
I = imread('highway.png');
```

Detect vehicles in the test image by using the trained YOLO v2 detector. Pass the test image and the detector as input to the `detect` (Computer Vision Toolbox) function. The `detect` function returns the bounding boxes and the detection scores.

```
[bboxes,scores] = detect(detector,I);
```

Annotate the image with the bounding boxes and the detection scores. Display the detection results. The bounding boxes localize vehicles in the test image.

```
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I)
title('Detected vehicles and detection scores');
```



Output Arguments

detector — Trained YOLO v2 network for vehicle detection
yolov2ObjectDetector

Trained YOLO v2 network for vehicle detection, returned as an yolov2ObjectDetector object. To detect vehicles in a test image, pass the YOLO v2 vehicle detector as input to the detect function.

See Also

Objects
yolov2ObjectDetector

Functions
configureDetectorMonoCamera | detect | trainYOLOv2ObjectDetector | vehicleDetectorFasterRCNN

Topics
"Create YOLO v2 Object Detection Network" (Computer Vision Toolbox)
"Train YOLO v2 Network for Vehicle Detection" (Computer Vision Toolbox)
"Object Detection Using YOLO v2 Deep Learning" (Computer Vision Toolbox)

Introduced in R2020a

latlon2local

Convert geographic coordinates to local Cartesian coordinates

Syntax

```
[xEast,yNorth,zUp] = latlon2local(lat,lon,alt,origin)
```

Description

`[xEast,yNorth,zUp] = latlon2local(lat,lon,alt,origin)` converts point locations given by `lat`, `lon`, and `alt` from geographic coordinates to local Cartesian coordinates returned as `xEast`, `yNorth`, and `zUp`. `origin` specifies the anchor of the local coordinate system as a vector of the form [*latOrigin*,*lonOrigin*,*altOrigin*]. Local *x*, *y*, *z* coordinates align with east, north and up directions, respectively. `alt` and `altOrigin` are altitudes as returned by a typical GPS sensor.

Examples

Convert Route to Cartesian Coordinates

Load a GPS route.

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

Define the origin in geographic coordinates, latitude and longitude.

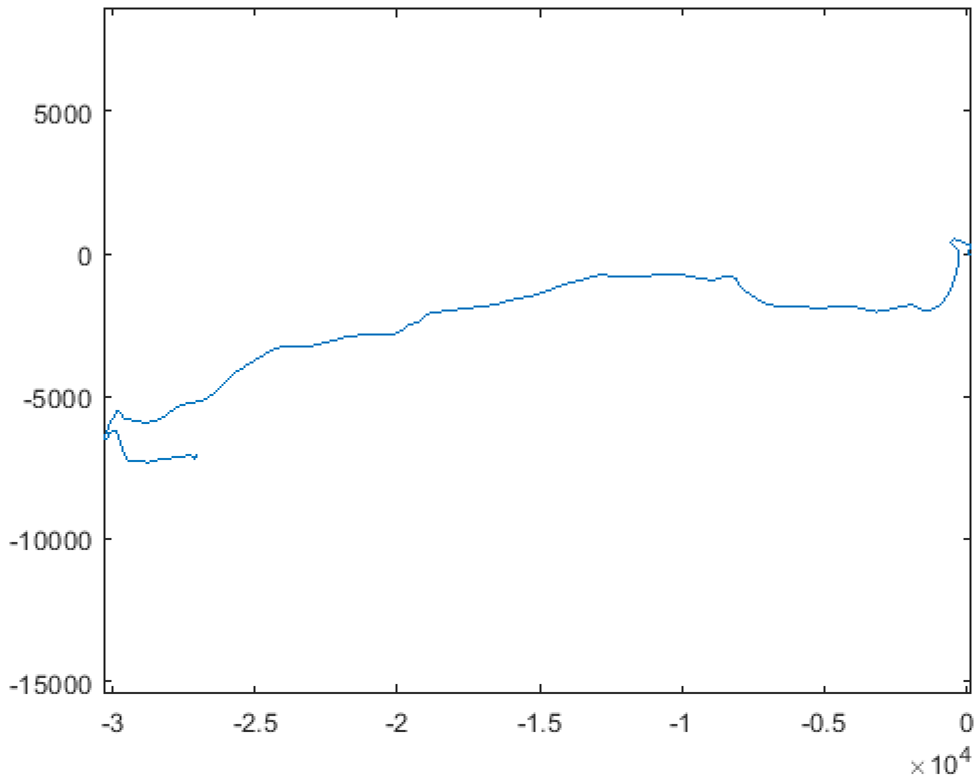
```
alt = 10;  % 10 meters is an approximate altitude in Boston, MA
origin = [d.latitude(1), d.longitude(1), alt];
```

Convert the route from geographic coordinates to Cartesian coordinates, *x* and *y*.

```
[xEast,yNorth] = latlon2local(d.latitude,d.longitude,alt,origin);
```

Plot the route in Cartesian coordinates.

```
figure;
plot(xEast,yNorth)
axis('equal'); % set 1:1 aspect ratio to see real-world shape
```



Input Arguments

lat — Latitude coordinates
numeric scalar | numeric vector

Latitude coordinates, in degrees, specified as a numeric scalar or vector. Value must be in the range [–90, 90]. `lat` must be the same length as `lon`.

Example: `lat = 42.3648`

Data Types: `single` | `double`

lon — Longitude coordinates
real scalar or vector in the range [–180, 180]

Longitude coordinates, in degrees, specified as a numeric scalar or vector. Value must be in the range [–180, 180]. `lon` must be the same length as `lat`.

Example: `lon = -71.0214`

Data Types: `single` | `double`

alt — Altitude
numeric scalar | numeric vector

Altitude, in meters, specified as a numeric scalar or vector.

Example: `10`

Data Types: `single` | `double`

`origin` — Anchor of local coordinate system
three-element vector

Anchor of local coordinate system, specified as a three-element vector of the form [*latOrigin*,*lonOrigin*,*altOrigin*].

Example: `[42.3648, -71.0214, 10.0];`

Data Types: `single` | `double`

Output Arguments

`xEast` — *x*-coordinates
numeric scalar | numeric vector

x-coordinates, returned as a numeric scalar or vector, in meters.

`xEast` is the same class as `lat`. However, if any of the input arguments is of class `single`, then `xEast` is of class `single`.

`yNorth` — *y*-coordinates
numeric scalar | numeric vector

y-coordinates, returned as a numeric scalar or vector, in meters.

`yNorth` is the same class as `lon`. However, if any of the input arguments is of class `single`, then `yNorth` is of class `single`.

`zUp` — Altitude
numeric scalar or vector

Altitude, returned as a numeric scalar or vector, in meters.

`zUp` is the same class as `alt`. However, if any of the input arguments is of class `single`, then `zUp` is of class `single`.

Tips

- The latitude and longitude of the geographic coordinate system use the WGS84 standard that is commonly used by GPS receivers.
- This function defines altitude as the height, in meters, above the WGS84 reference ellipsoid.
- Some GPS receivers use standards other than WGS84. Conversions using other ellipsoids are available in the Mapping Toolbox. This function addresses the most common conversion between geographic locations and Cartesian coordinates used by the on-board sensors of a vehicle.

See Also

`geoplayer` | `geoplot` | `local2latlon`

Introduced in R2020a

local2latlon

Convert local Cartesian coordinates to geographic coordinates

Syntax

`[lat,lon,alt] = local2latlon(xEast,yNorth,zUp,origin)`

Description

`[lat,lon,alt] = local2latlon(xEast,yNorth,zUp,origin)` converts point locations given by `xEast`, `yNorth`, and `zUp` from local Cartesian coordinates to geographic coordinates returned as `lat`, `lon`, and `alt`. `origin` specifies the anchor of the local coordinate system as a three-element vector of the form [*latOrigin*,*lonOrigin*,*altOrigin*]. Local coordinates `xEast`, `yNorth`, and `zUp` align with the east, north, and up directions, respectively. `alt` and *altOrigin* are altitudes as typically returned by GPS sensors.

Examples

Convert Route to Geographic Coordinates

Establish an anchor point in the geographic coordinate system. These latitude and longitude coordinates specify Boston, MA.

```
origin = [42.3648, -71.0214, 10.0];
```

Generate local route in Cartesian coordinates, *x*, *y*, and *z*.

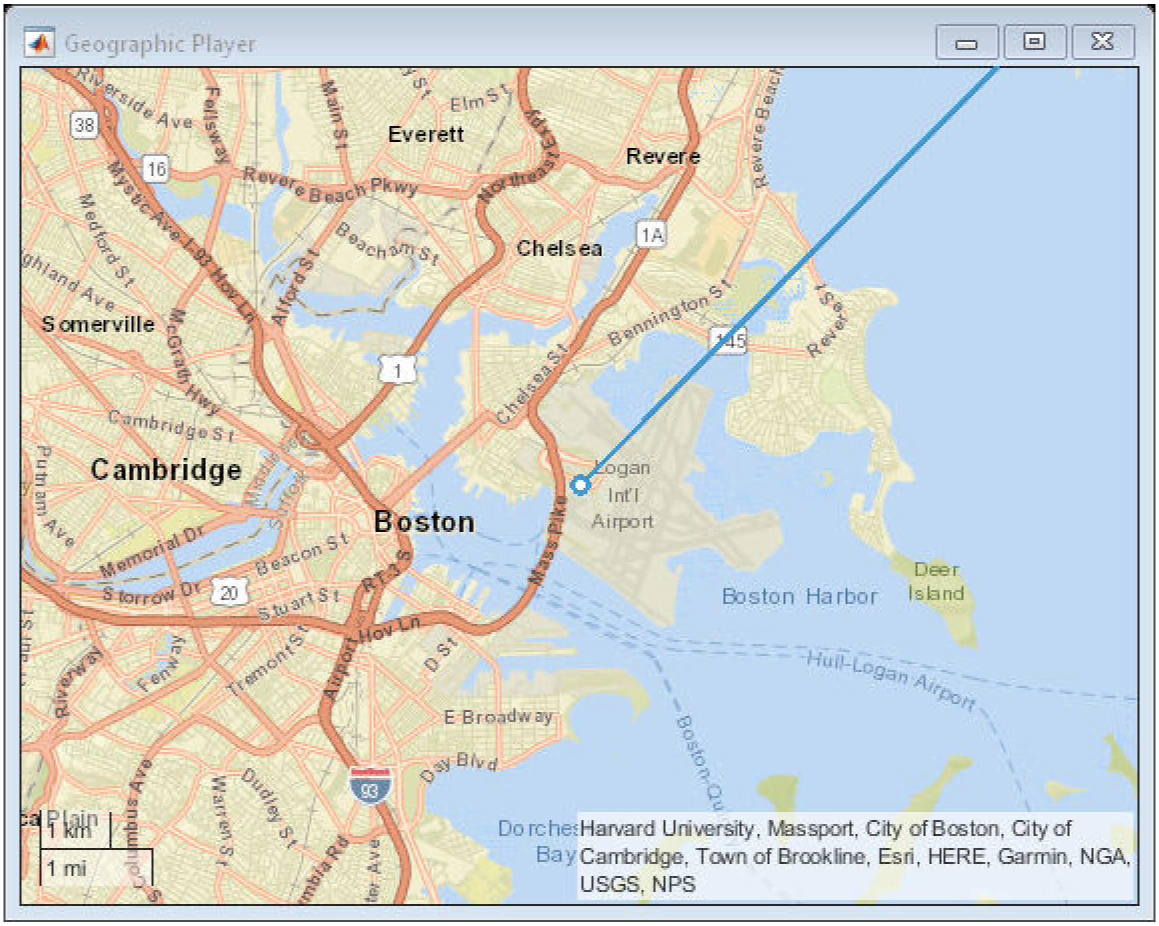
```
z = zeros(1,101);    % maintain height of 0 m
x = 0:1000:100000;   % 100 km in 1 km increments
y = x;               % move 100 km northeast
```

Convert the local route coordinates to geographic coordinates, latitude and longitude.

```
[lat,lon] = local2latlon(x,y,z,origin);
```

Visualize the route on a map.

```
zoomLevel = 12;
player = geoplayer(lat(1),lon(1),zoomLevel);
plotRoute(player,lat,lon);
```



Input Arguments

`xEast` — *x*-coordinates
numeric scalar | numeric vector

x-coordinates in the local Cartesian coordinate system, specified as a numeric scalar or vector, in meters.

Example: `-2.7119`

Data Types: `single` | `double`

`yNorth` — *y*-coordinates
numeric scalar | numeric vector

y-coordinates in the local Cartesian coordinate system, specified as a numeric scalar or vector, in meters.

Example: `-7.0681`

Data Types: `single` | `double`

`zUp` — *z*-coordinate
numeric scalar | numeric vector

y coordinate in local Cartesian coordinate system, specified as a numeric scalar or vector, in meters.

Example: `-0.2569`

Data Types: `single` | `double`

`origin` — Anchor of local coordinate system
three-element vector

Anchor of local coordinate system, specified as a three-element vector of the form [*latOrigin*,*lonOrigin*,*altOrigin*].

Example: `[42.3648 -71.0214 10]`

Data Types: `single` | `double`

Output Arguments

`lat` — Latitude
numeric scalar or vector

Latitude, in degrees, returned as a numeric scalar or vector.

`lat` is the same class as `xEast`. However, if any of the input arguments is of class `single`, then `lat` is of class `single`.

`lon` — Longitude
numeric scalar or vector

Longitude, in degrees, returned as a numeric scalar or vector.

`lon` is the same class as `yNorth`. However, if any of the input arguments is of class `single`, then `lon` is of class `single`.

`alt` — Altitude
numeric scalar or vector

Altitude, in meters, returned as a numeric scalar or vector, the same class as `zUp`.

`alt` is the same class as `zUp`. However, if any of the input arguments is of class `single`, then `alt` is of class `single`.

Tips

- The latitude and longitude of the geographic coordinate system use the WGS84 standard that is commonly used by GPS receivers.
- This function defines altitude as the height, in meters, above the WGS84 reference ellipsoid.
- Some GPS receivers use standards other than WGS84. Conversions using other ellipsoids are available in the Mapping Toolbox. This function addresses the most common conversion between geographic locations and Cartesian coordinates used by the on-board sensors of a vehicle.

See Also
`geoplayer` | `geoplot` | `latlon2local`

Introduced in R2020a

4

Objects

birdsEyePlot

Plot detections, tracks, and sensor coverages around vehicle

Description

The `birdsEyePlot` object displays a bird's-eye plot of a 2-D driving scenario in the immediate vicinity of an ego vehicle. You can use this plot with sensors capable of detecting objects and lanes.

To display aspects of a driving scenario on a bird’s-eye plot:

1. Create a `birdsEyePlot` object.
2. Create plotters for the aspects of the driving scenario that you want to plot.
3. Use the plotters with their corresponding plot functions to display those aspects on the bird's-eye plot.

This table shows the plotter functions to use based on the driving scenario aspect that you want to plot.

Driving Scenario Aspect to Plot	Plotter Creation Function	Plotter Display Function
Sensor coverage areas	`coverageAreaPlotter`	`plotCoverageArea`
Sensor detections	`detectionPlotter`	`plotDetection`
Lane boundaries	`laneBoundaryPlotter`	`plotLaneBoundary`
Lane markings	`laneMarkingPlotter`	`plotLaneMarking`
Object meshes	`meshPlotter`	`plotMesh`
Object outlines	`outlinePlotter`	`plotOutline`
Ego vehicle path	`pathPlotter`	`plotPath`
Point cloud	`pointCloudPlotter`	`plotPointCloud`
Object tracking results	`trackPlotter`	`plotTrack`

For an example of how to configure and use a bird's-eye plot, see “Visualize Sensor Coverage, Detections, and Tracks”.

Creation

Syntax

```
bep = birdsEyePlot
bep = birdsEyePlot(Name,Value)
```

Description

`bep = birdsEyePlot` creates a bird’s-eye plot in a new figure.

`bep = birdsEyePlot(Name,Value)` sets properties on page 4-3 using one or more `Name,Value` pair arguments. For example, `birdsEyePlot('XLimits',[0 60],'YLimits',[-20 20])` displays the area that is 60 meters in front of the ego vehicle and 20 meters to either side of the ego vehicle. Enclose each property name in quotes.

Properties

`Parent` — Axes on which to plot
axes handle

Axes on which to plot, specified as an axes handle. By default, the `birdsEyePlot` object uses the current axes handle, which is returned by the `gca` function.

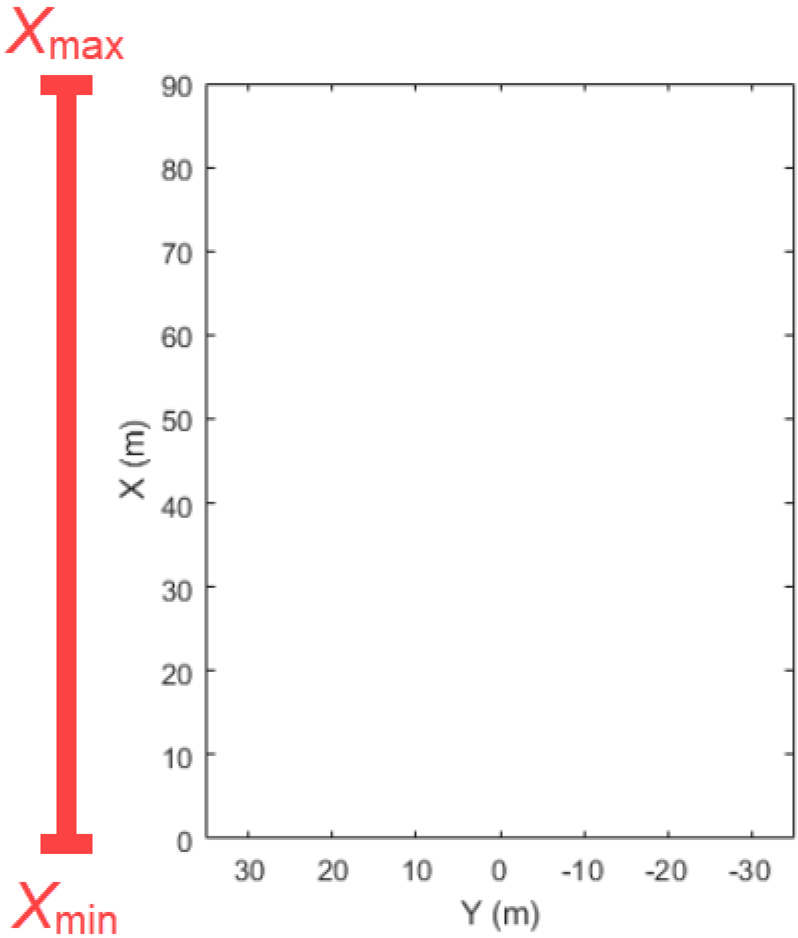
`Plotters` — Plotters created for bird's-eye plot
array of plotter objects

Plotters created for the bird's-eye plot, specified as an array of plotter objects.

`XLimits` — *X*-axis range
real-valued vector of the form [X_{min} X_{max}]

X-axis range of the bird's-eye plot, in vehicle coordinates, specified as a real-valued vector of the form [X_{min} X_{max}]. Units are in meters. If you do not specify `XLimits`, then the plot uses the default values for the parent axes.

The *X*-axis is vertical and positive in the forward direction of the ego vehicle. The origin is at the center of the rear axle of the ego vehicle.

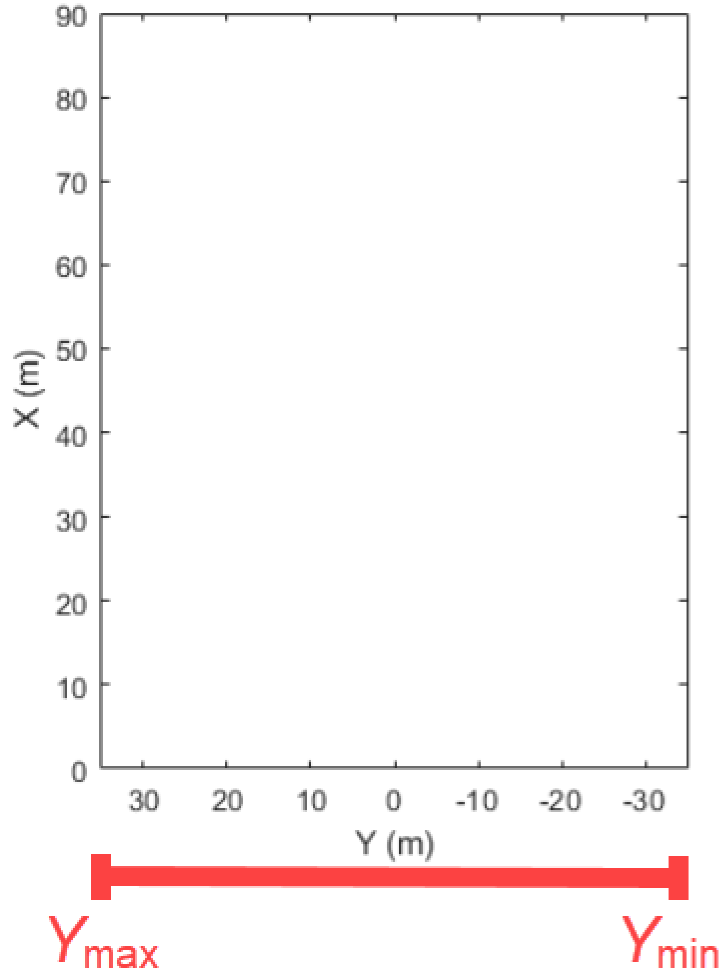


For more details on the coordinate system used in the bird's-eye plot, see “Vehicle Coordinate System” on page 4-11.

YLimits — *Y*-axis range
real-valued vector of the form [Y_{min} Y_{max}]

Y-axis range of the bird's-eye plot, in vehicle coordinates, specified as a real-valued vector of the form [Y_{min} Y_{max}]. Units are in meters. If you do not specify `YLimits`, then the plot uses the default values for the parent axes.

The *Y*-axis runs horizontally and is positive to the left of the ego vehicle, as viewed when facing forward. The origin is at the center of the rear axle of the ego vehicle.



For more details on the coordinate system used in the `birdsEyePlot` object, see "Vehicle Coordinate System" on page 4-11.

Object Functions

Plotter Creation

coverageAreaPlotter	Coverage area plotter for bird's-eye plot
detectionPlotter	Detection plotter for bird's-eye plot
laneBoundaryPlotter	Lane boundary plotter for bird's-eye plot
laneMarkingPlotter	Lane marking plotter for bird's-eye plot
meshPlotter	Mesh plotter for bird's-eye plot
outlinePlotter	Outline plotter for bird's-eye plot
pathPlotter	Path plotter for bird's-eye plot
pointCloudPlotter	Point cloud plotter for bird's-eye plot
trackPlotter	Track plotter for bird's-eye plot

Plotter Display

plotCoverageArea	Display sensor coverage area on bird's-eye plot
plotDetection	Display object detections on bird's-eye plot
plotLaneBoundary	Display lane boundaries on bird’s-eye plot
plotLaneMarking	Display lane markings on bird’s-eye plot
plotMesh	Display object meshes on bird's-eye plot
plotOutline	Display object outlines on bird's-eye plot
plotPath	Display actor paths on bird’s-eye plot
plotPointCloud	Display generated point cloud on bird's-eye plot
plotTrack	Display object tracks on bird's-eye plot

Plotter Utilities

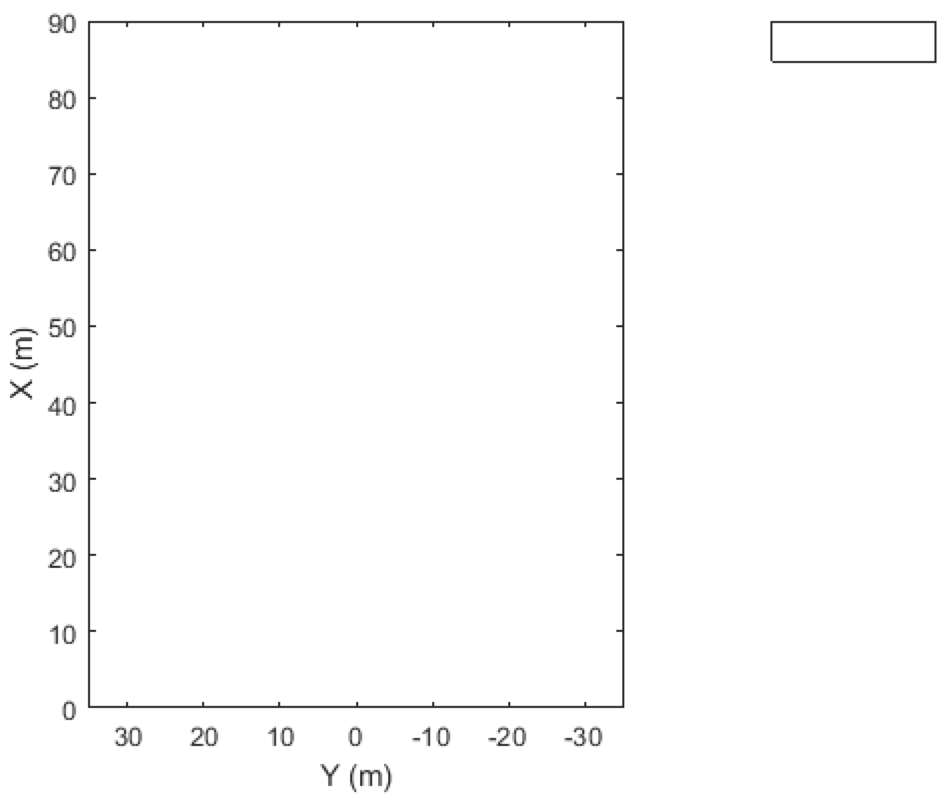
clearData	Clear data from specific plotter of bird’s-eye plot
clearPlotterData	Clear data from bird’s-eye plot
findPlotter	Find plotters associated with bird’s-eye plot

Examples

Create and Display a Bird's-Eye Plot

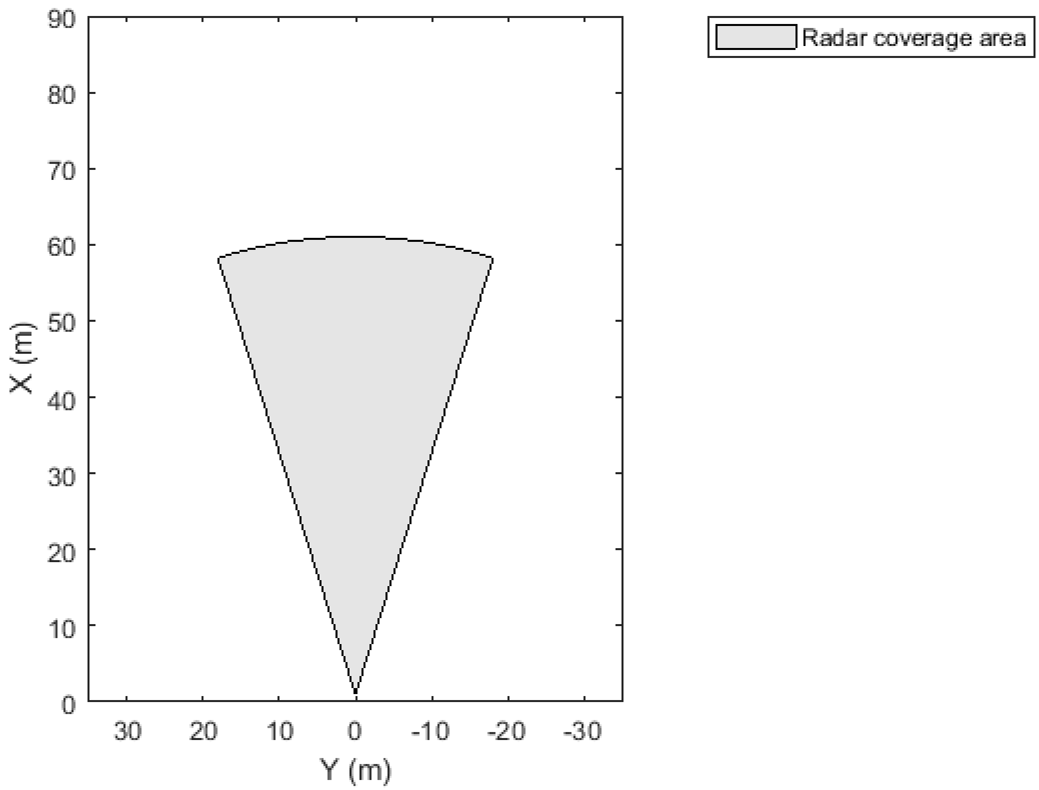
Create a bird's-eye plot with an *x*-axis range from 0 to 90 meters and a *y*-axis range from –35 to 35 meters.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
```



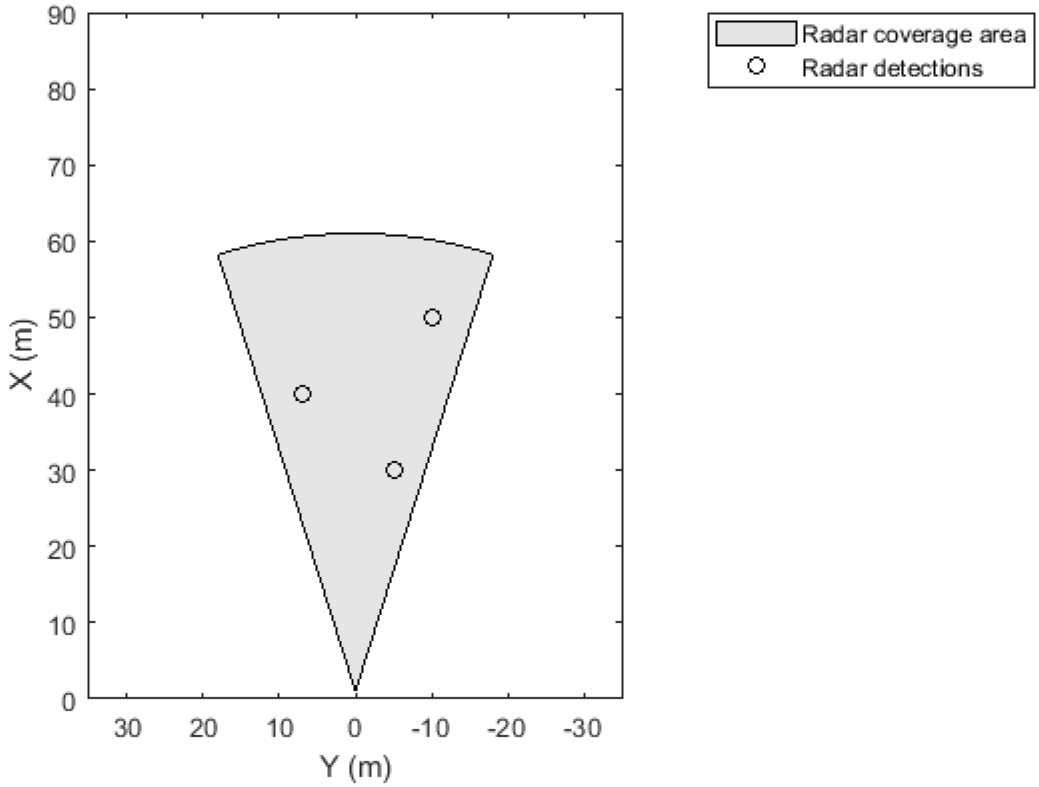
Display a coverage area with a 35-degree field of view and a 60-meter range.

```
caPlotter = coverageAreaPlotter(bep,'DisplayName','Radar coverage area');
mountPosition = [1 0];
range = 60;
orientation = 0;
fieldOfView = 35;
plotCoverageArea(caPlotter,mountPosition,range,orientation,fieldOfView);
```



Display radar detections with coordinates at (30, –5), (50, –10), and (40, 7).

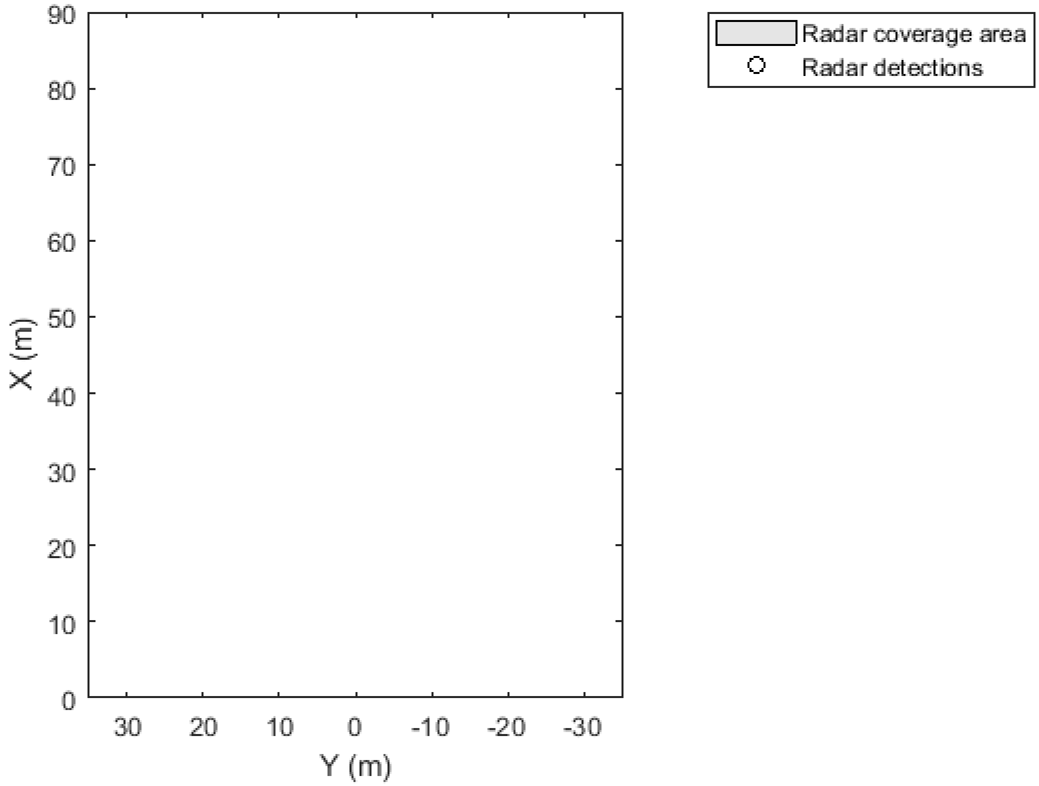
```
radarPlotter = detectionPlotter(bep,'DisplayName','Radar detections');
plotDetection(radarPlotter,[30 -5; 50 -10; 40 7]);
```



Create Bird's-Eye Plot with Coverage Area and Detection Plotters

Create a bird's-eye plot with an *x*-axis range of 0 to 90 meters and a *y*-axis range from –35 to 35 meters. Configure the plot to include a radar coverage area plotter and a detection plotter. Set the display names of these plotters.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
coverageAreaPlotter(bep,'DisplayName','Radar coverage area');
detectionPlotter(bep,'DisplayName','Radar detections');
```

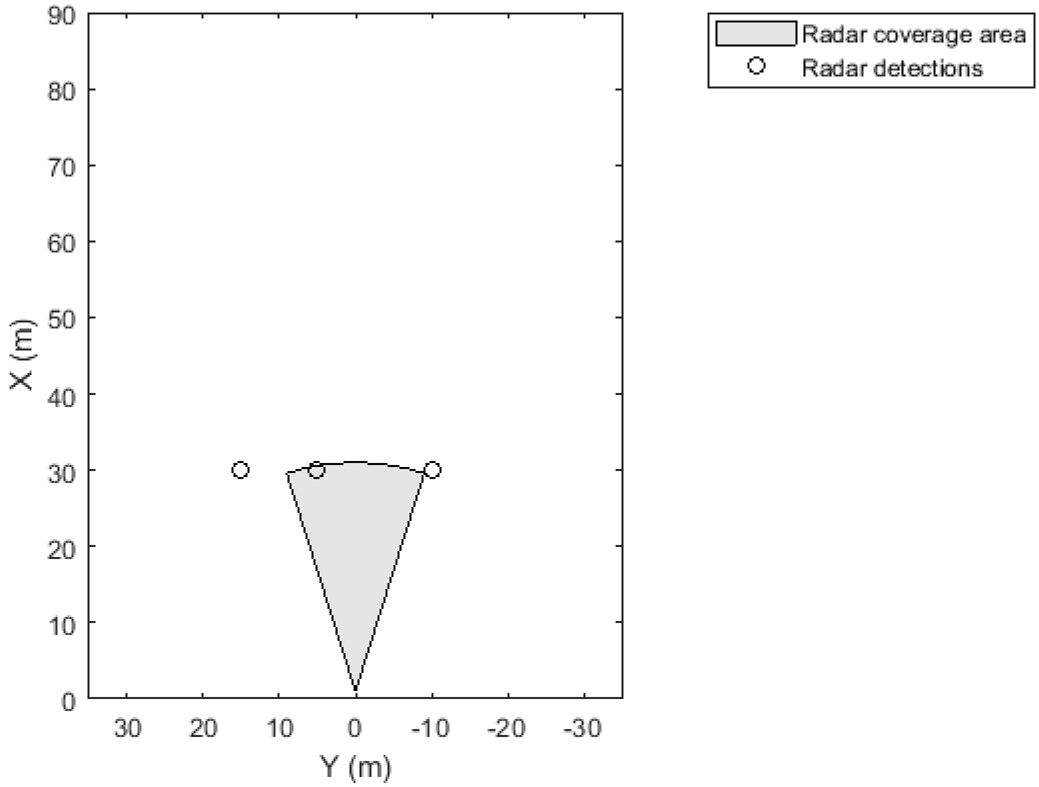


Use `findPlotter` to locate the plotters by their display names.

```
caPlotter = findPlotter(bep,'DisplayName','Radar coverage area');
radarPlotter = findPlotter(bep,'DisplayName','Radar detections');
```

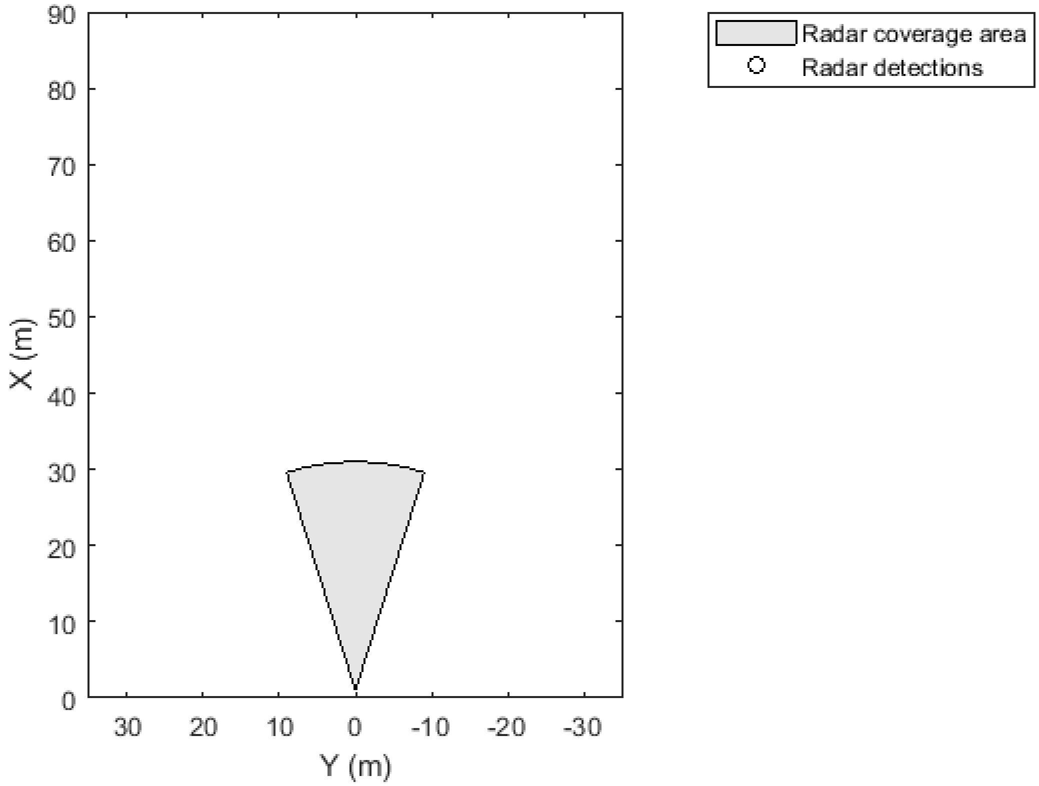
Plot the coverage area and detected objects.

```
plotCoverageArea(caPlotter,[1 0],30,0,35);
plotDetection(radarPlotter,[30 5; 30 -10; 30 15]);
```



Clear data from the plot.

```
clearPlotterData(bep);
```



Limitations

The rectangle-zoom feature, where you draw a rectangle to zoom in on a section of a figure, does not work in bird's-eye plot figures.

More About

Vehicle Coordinate System

The `birdsEyePlot` uses the vehicle coordinate system (X_V, Y_V), where:

- The X_V-axis points forward from the ego vehicle.
- The Y_V-axis points to the left, as viewed when facing forward.

The origin is at the center of rotation of the ego vehicle. This point is on the road surface, beneath the center of the rear axle of the ego vehicle.

For more details about the vehicle coordinate system, see “Coordinate Systems in Automated Driving Toolbox”.

See Also

Bird's-Eye Scope | `drivingScenario`

Topics

“Visualize Sensor Coverage, Detections, and Tracks”
“Coordinate Systems in Automated Driving Toolbox”

Introduced in R2017a

clearData

Clear data from specific plotter of bird’s-eye plot

Syntax

`clearData(pl)`

Description

`clearData(pl)` clears data belonging to the plotter `pl` associated with a bird’s-eye plot. This function can clear data from these plotters:

- `detectionPlotter`
- `laneBoundaryPlotter`
- `laneMarkingPlotter`
- `outlinePlotter`
- `meshPlotter`
- `pathPlotter`
- `pointCloudPlotter`
- `trackPlotter`

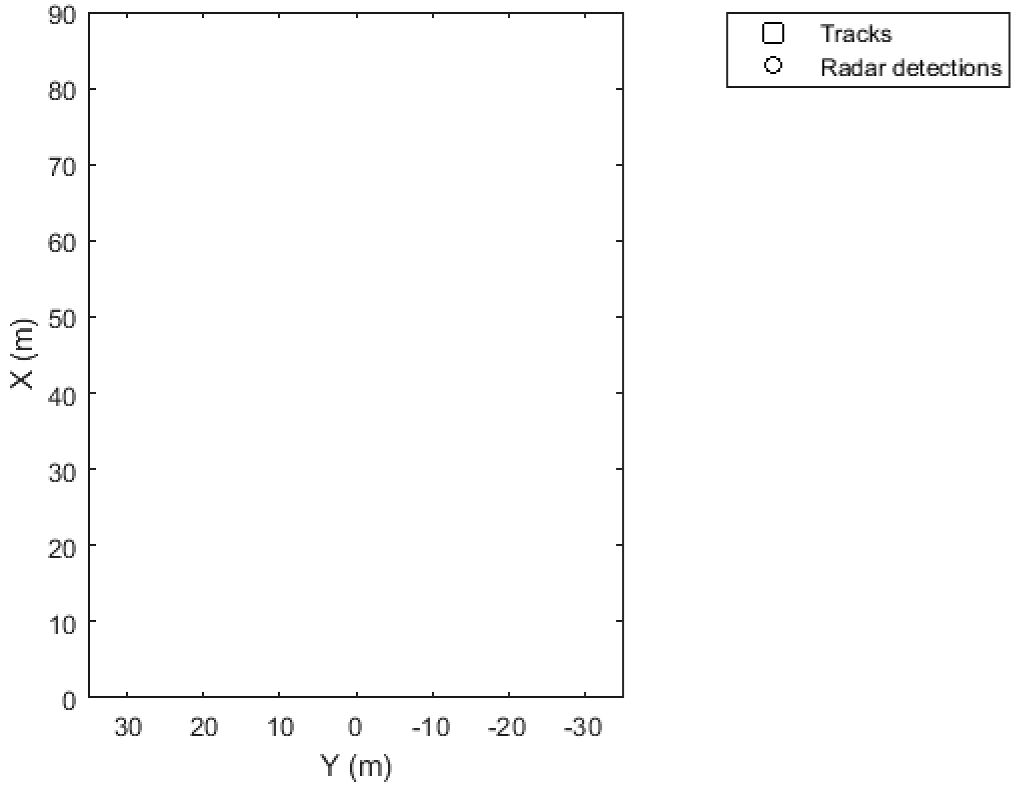
To clear data from all plotters belonging to a bird's-eye plot, use the `clearPlotterData` function.

Examples

Clear Specific Plotter Data from Bird's-Eye Plot

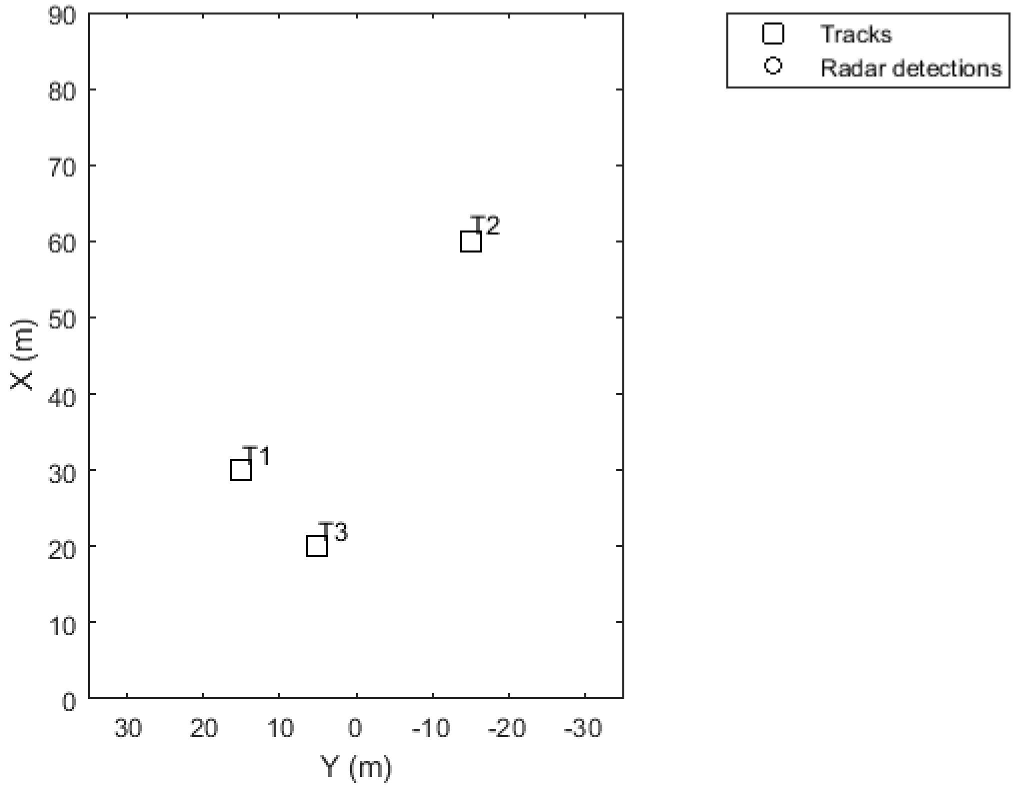
Create a bird's-eye plot. Add a track plotter and detection plotter to the bird's-eye plot.

```
bep = birdsEyePlot('XLim',[0,90],'YLim',[-35,35]);
tPlotter = trackPlotter(bep,'DisplayName','Tracks');
detPlotter = detectionPlotter(bep,'DisplayName','Radar detections');
```



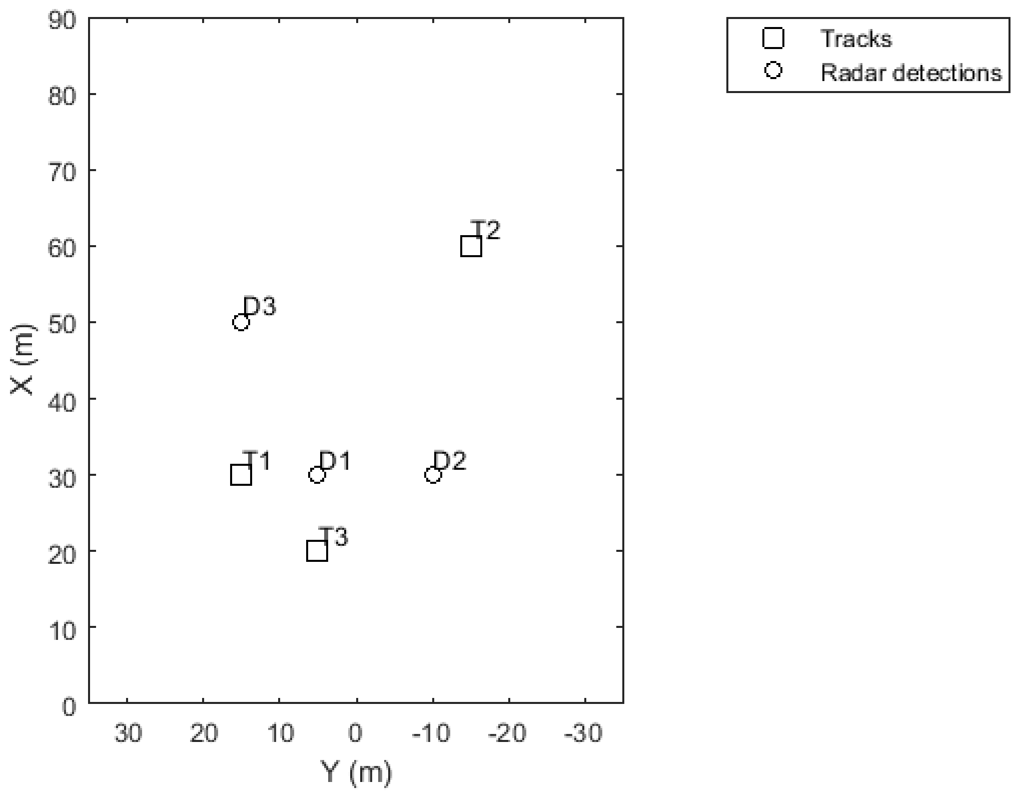
Create and display a set of tracks on the bird's-eye plot.

```
trackPos = [30 15 1; 60 -15 1; 20 5 1];
trackLabels = {'T1','T2','T3'};
plotTrack(tPlotter,trackPos,trackLabels)
```



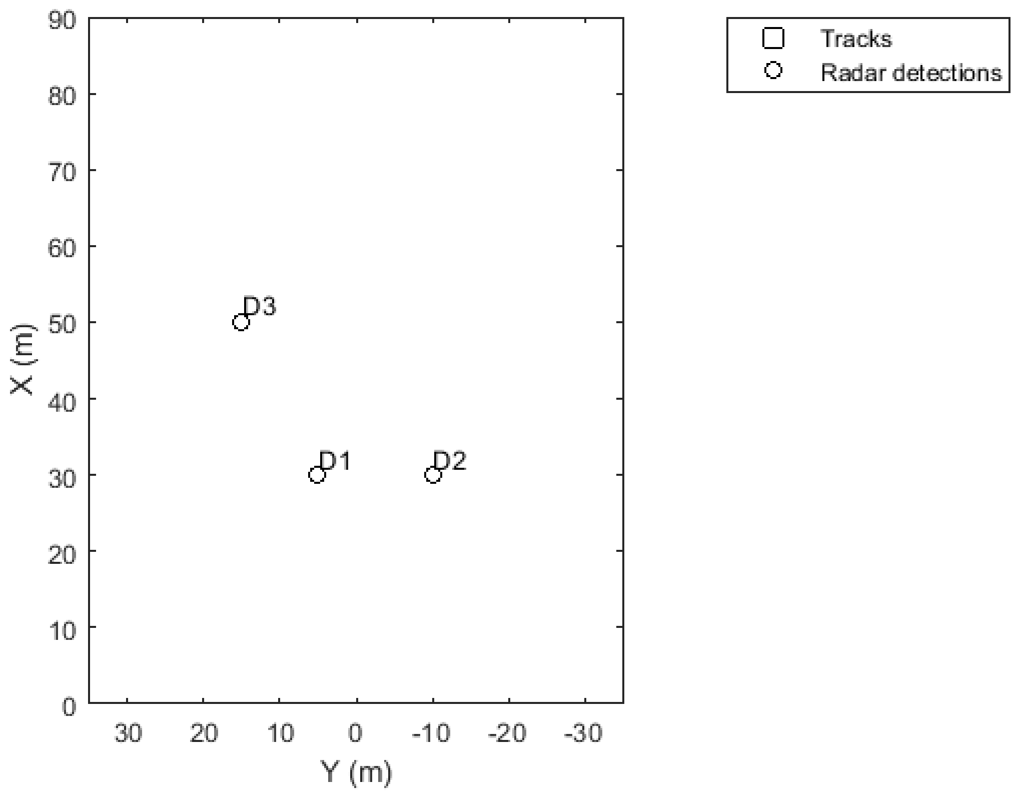
Create and display a set of detections on the bird's-eye plot.

```
detPos = [30 5 4; 30 -10 2; 50 15 1];
detLabels = {'D1','D2','D3'};
plotDetection(detPlotter,detPos,detLabels)
```



Clear the track plotter data from the bird's-eye plot.

```
clearData(tPlotter)
```



Input Arguments

pl — Plotter belonging to bird's-eye plot
plotter object

Plotter belonging to a `birdsEyePlot` object, specified as a plotter object. You can clear data from any plotter except `coverageAreaPlotter`.

See Also

Objects
`birdsEyePlot` | `clearPlotterData` | `findPlotter`

Introduced in R2017a

clearPlotterData

Clear data from bird’s-eye plot

Syntax

`clearPlotterData(bep)`

Description

`clearPlotterData(bep)` clears all plotter data displayed in the specified bird’s-eye plot. Legend entries and coverage areas are not cleared from the plot.

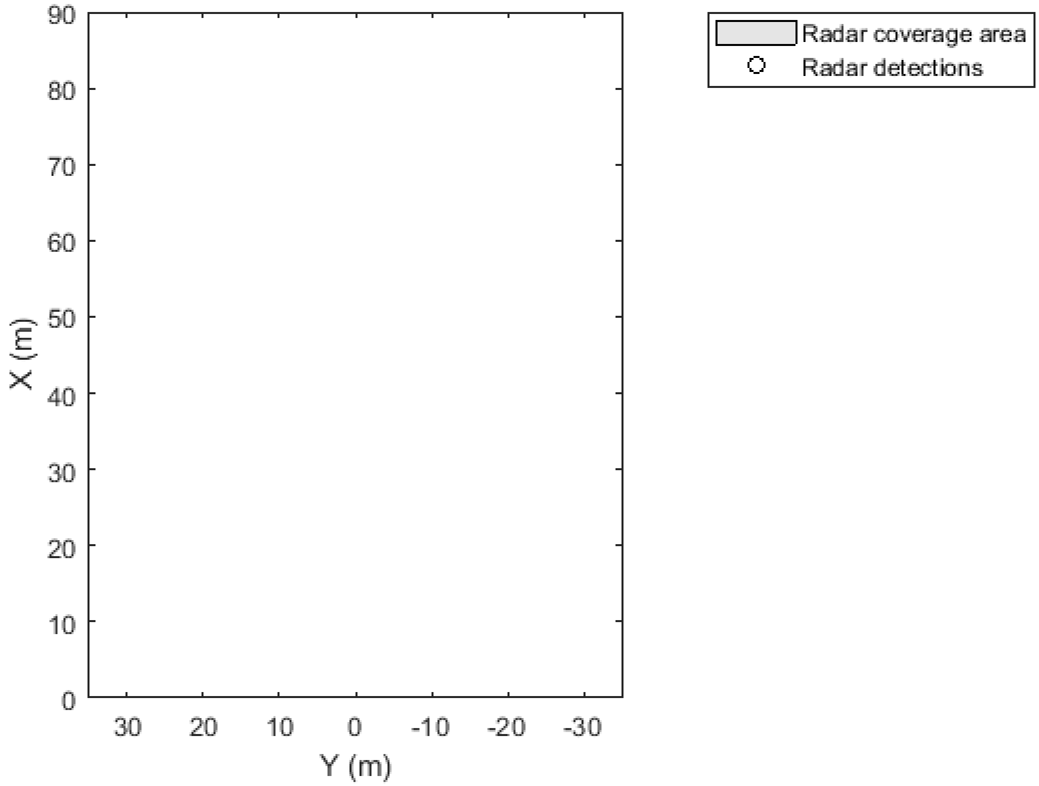
To clear data from a specific plotter, use the `clearData` function.

Examples

Create Bird's-Eye Plot with Coverage Area and Detection Plotters

Create a bird's-eye plot with an *x*-axis range of 0 to 90 meters and a *y*-axis range from –35 to 35 meters. Configure the plot to include a radar coverage area plotter and a detection plotter. Set the display names of these plotters.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
coverageAreaPlotter(bep,'DisplayName','Radar coverage area');
detectionPlotter(bep,'DisplayName','Radar detections');
```

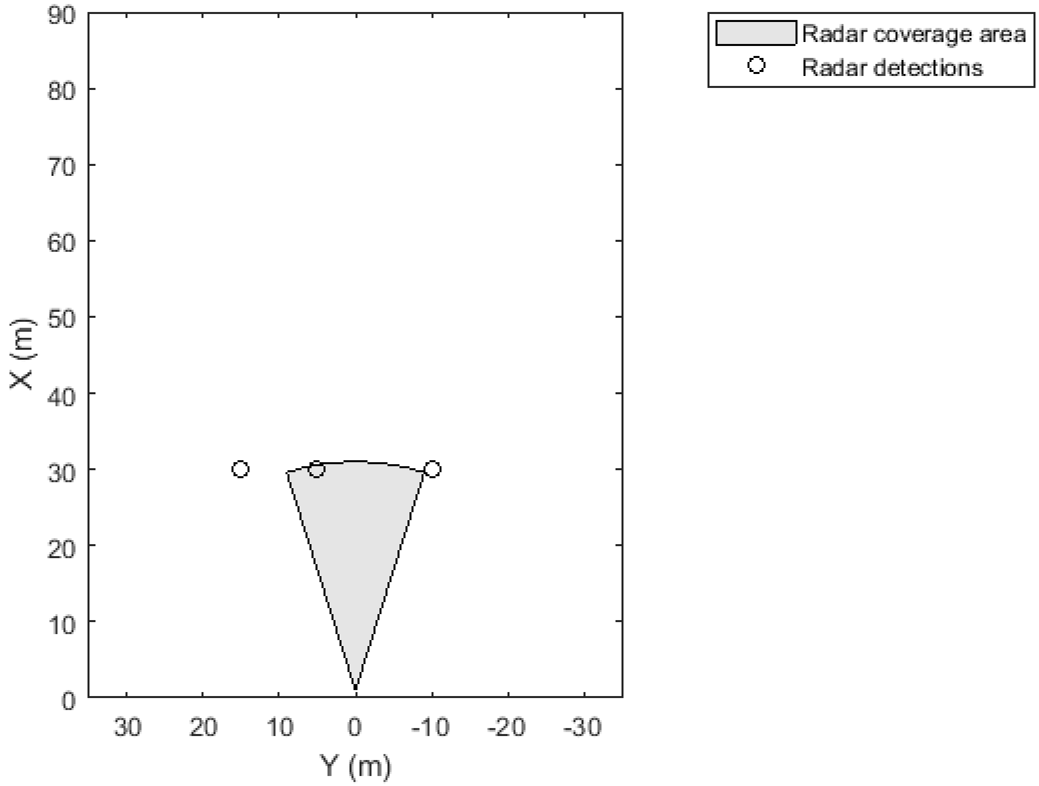


Use `findPlotter` to locate the plotters by their display names.

```
caPlotter = findPlotter(bep,'DisplayName','Radar coverage area');
radarPlotter = findPlotter(bep,'DisplayName','Radar detections');
```

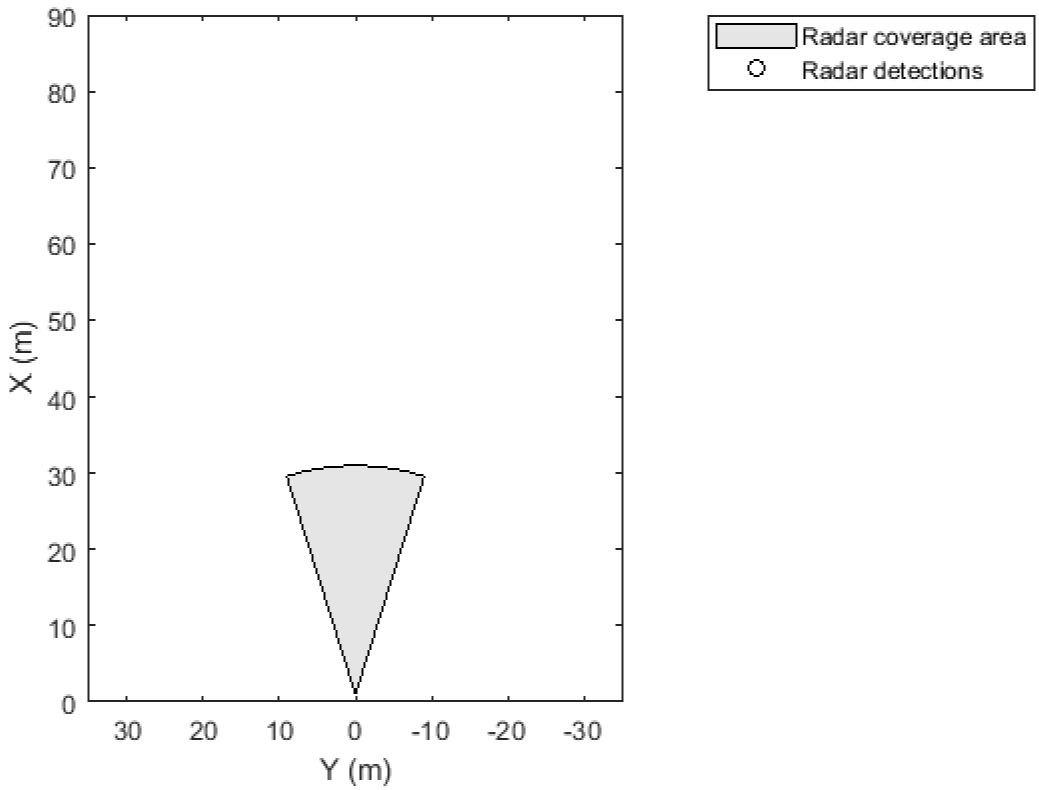
Plot the coverage area and detected objects.

```
plotCoverageArea(caPlotter,[1 0],30,0,35);
plotDetection(radarPlotter,[30 5; 30 -10; 30 15]);
```



Clear data from the plot.

```
clearPlotterData(bep);
```



Input Arguments

bep — Bird's-eye plot
`birdsEyePlot` object

Bird's-eye plot, specified as a `birdsEyePlot` object.

See Also

Functions
`birdsEyePlot` | `clearData` | `findPlotter`

Introduced in R2017a

coverageAreaPlotter

Package:

Coverage area plotter for bird's-eye plot

Syntax

```
caPlotter = coverageAreaPlotter(bep)
caPlotter = coverageAreaPlotter(bep,Name,Value)
```

Description

`caPlotter = coverageAreaPlotter(bep)` creates a `CoverageAreaPlotter` object that configures the display of sensor coverage areas on a bird's-eye plot. The `CoverageAreaPlotter` object is stored in the `Plotters` property of the input `birdsEyePlot` object, `bep`. To display the sensor coverage areas, use the `plotCoverageArea` function.

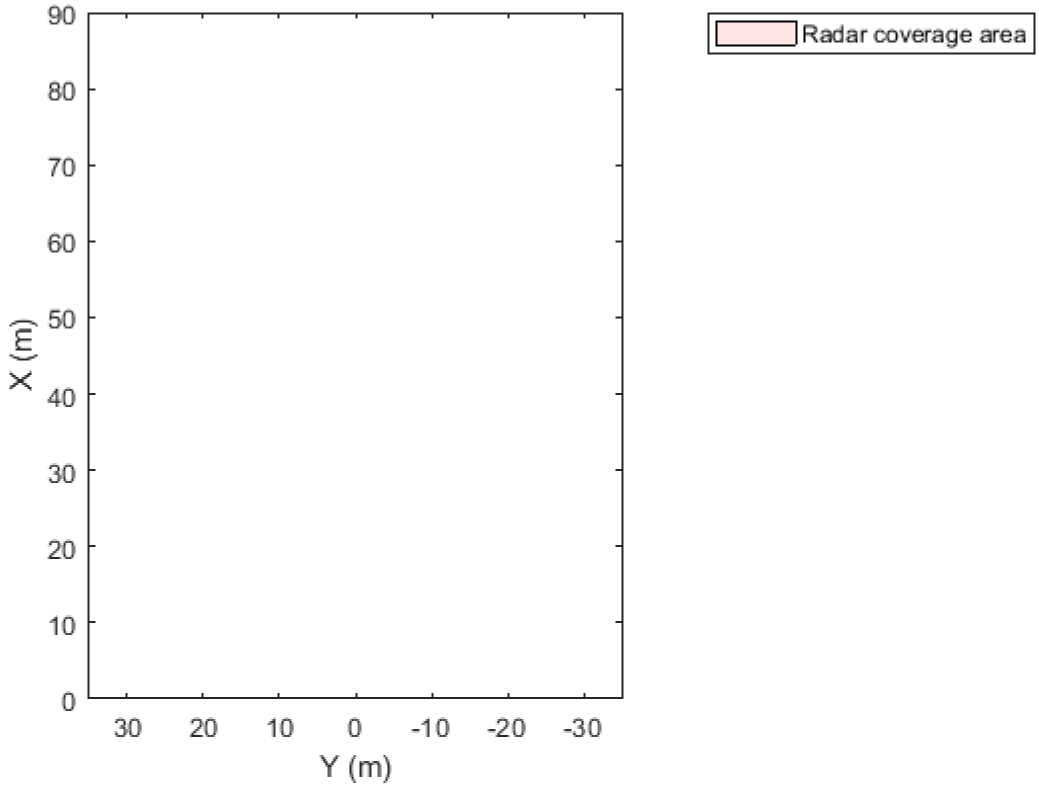
`caPlotter = coverageAreaPlotter(bep,Name,Value)` sets properties using one or more `Name,Value` pair arguments. For example, `coverageAreaPlotter(bep,'DisplayName','Coverage area')` sets the display name that appears in the bird's-eye-plot legend.

Examples

Create and Display Coverage Area on Bird's-Eye Plot

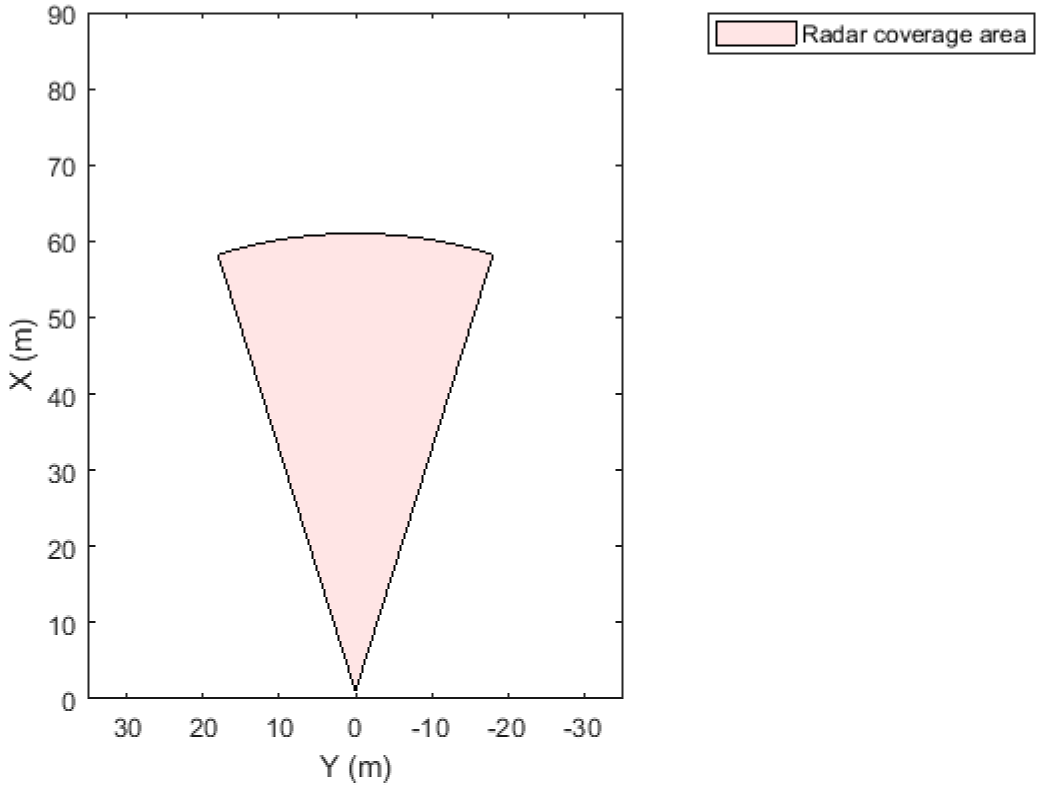
Create a bird's-eye plot with an *x*-axis range from 0 to 90 meters and a *y*-axis range from –35 to 35 meters. Create a coverage area plotter that displays coverage areas in red.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
caPlotter = coverageAreaPlotter(bep,'DisplayName','Radar coverage area','FaceColor','r');
```



Display a coverage area that has a 35-degree field of view and a 60-meter range. Mount the coverage area sensor 1 meter in front of the origin. Set the orientation angle of the sensor to 0 degrees.

```
mountPosition = [1 0];
range = 60;
orientation = 0;
fieldOfView = 35;
plotCoverageArea(caPlotter,mountPosition,range,orientation,fieldOfView);
```



Input Arguments

bep — Bird's-eye plot
birdsEyePlot object

Bird's-eye plot, specified as a `birdsEyePlot` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `coverageAreaPlotter('FaceColor','red')` sets the fill color of sensor coverage areas to red.

`DisplayName` — Plotter name to display in legend
`''` (default) | character vector | string scalar

Plotter name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and character vector or string scalar. If you do not specify a name, the bird's-eye plot does not display a legend entry for the plotter.

`FaceColor` — Fill color of coverage areas
`[0 0 0]` (black) (default) | RGB triplet | hexadecimal color code | color name | short color name

Fill color of coverage areas, specified as the comma-separated pair consisting of 'FaceColor' and an RGB triplet, a hexadecimal color code, a color name, or a short color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

EdgeColor — Border color of coverage areas

[0 0 0] (black) (default) | RGB triplet | hexadecimal color code | color name | short color name

Border color of coverage areas, specified as the comma-separated pair consisting of 'EdgeColor' and an RGB triplet, a hexadecimal color code, a color name, or a short color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

FaceAlpha — Transparency of coverage areas
0.1 (default) | scalar in the range [0, 1]

Transparency of coverage areas, specified as the comma-separated pair consisting of 'FaceAlpha' and a scalar in the range [0, 1]. A value of 0 makes the coverage area fully transparent. A value of 1 makes the coverage area fully opaque.

Tag — Tag associated with plotter object
'Plotter*N*' (default) | character vector | string scalar

Tag associated with the plotter object, specified as the comma-separated pair consisting of `'Tag'` and a character vector or string scalar. The default value is `'Plotter`*`N`*`'`, where *N* is an integer that corresponds to the *N*th plotter associated with the input `birdsEyePlot` object.

Output Arguments

`caPlotter` — Coverage area plotter
`CoverageAreaPlotter` object

Coverage area plotter, returned as a `CoverageAreaPlotter` object. You can modify this object by changing its property values. The property names correspond to the name-value pair arguments of the `coverageAreaPlotter` function.

`caPlotter` is stored in the `Plotters` property of the input `birdsEyePlot` object, `bep`. To plot the coverage areas, use the `plotCoverageArea` function.

See Also

`birdsEyePlot` | `findPlotter` | `plotCoverageArea`

Introduced in R2017a

detectionPlotter

Package:

Detection plotter for bird's-eye plot

Syntax

```
detPlotter = detectionPlotter(bep)
detPlotter = detectionPlotter(bep,Name,Value)
```

Description

`detPlotter = detectionPlotter(bep)` creates a `DetectionPlotter` object that configures the display of object detections on a bird's-eye plot. The `DetectionPlotter` object is stored in the `Plotters` property of the input `birdsEyePlot` object, `bep`. To plot the object detections, use the `plotDetection` function.

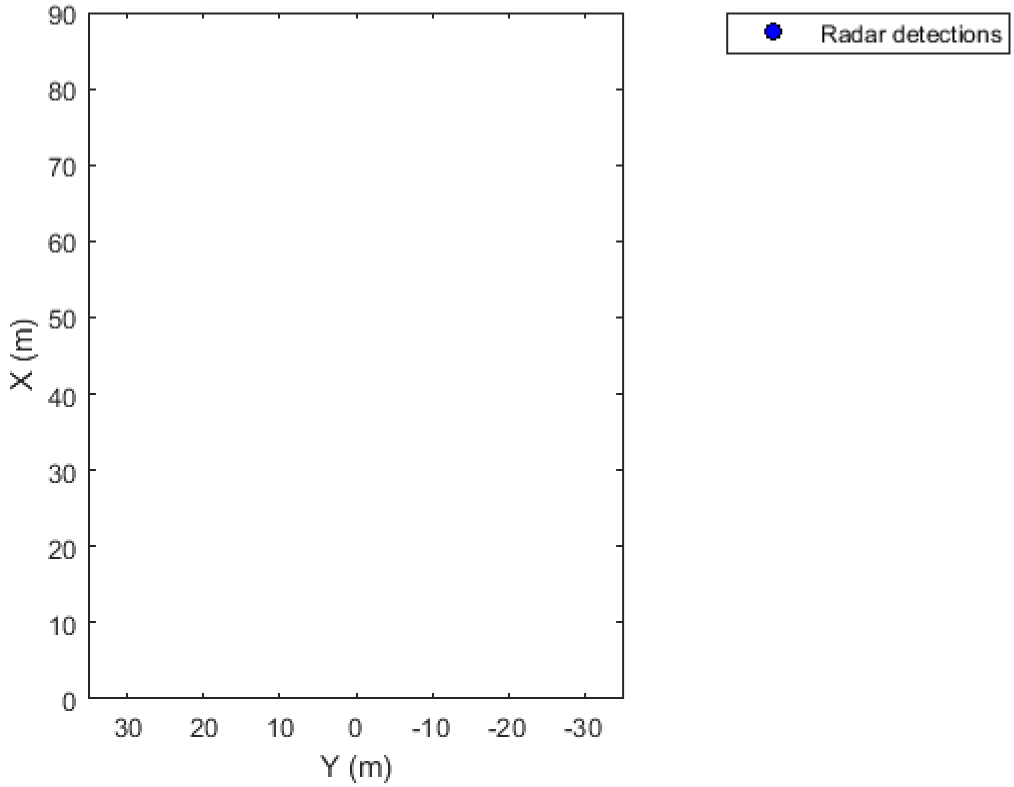
`detPlotter = detectionPlotter(bep,Name,Value)` sets properties using one or more `Name,Value` pair arguments. For example, `detectionPlotter(bep,'DisplayName','Detections')` sets the display name that appears in the bird's-eye-plot legend.

Examples

Create and Display Labeled Detections on Bird's-Eye Plot

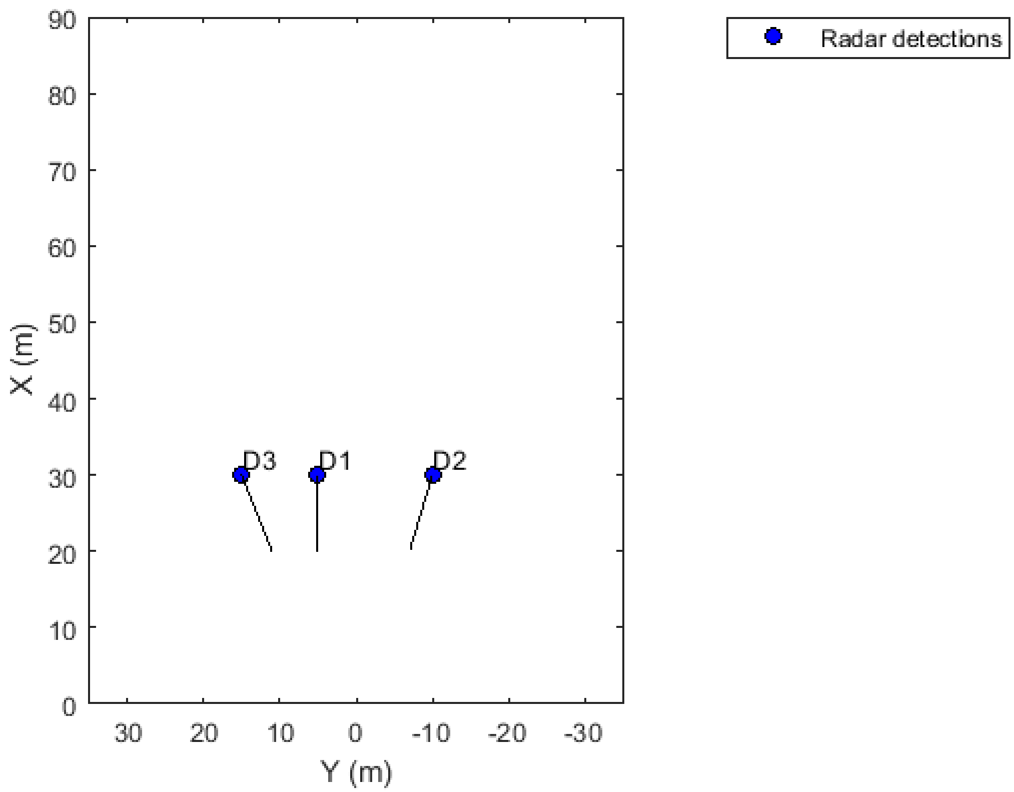
Create a bird's-eye plot with an *x*-axis range from 0 to 90 meters and a *y*-axis range from –35 to 35 meters. Create a radar detection plotter that displays detections in blue.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
detPlotter = detectionPlotter(bep,'DisplayName','Radar detections', ...
    'MarkerFaceColor','b');
```



Display the positions and velocities of three labeled detections.

```
positions = [30 5; 30 -10; 30 15];
velocities = [-10 0; -10 3; -10 -4];
labels = {'D1','D2','D3'};
plotDetection(detPlotter,positions,velocities,labels);
```



Input Arguments

bep — Bird’s-eye plot
`birdsEyePlot` object

Bird’s-eye plot, specified as a `birdsEyePlot` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `detectionPlotter('Marker','+')` sets the marker symbol for detections to a plus sign.

`DisplayName` — Plotter name to display in legend
`''` (default) | character vector | string scalar

Plotter name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and character vector or string scalar. If you do not specify a name, the bird's-eye plot does not display a legend entry for the plotter.

`Marker` — Marker symbol for detections
`'o'` (default) | `'+'` | `'*'` | `'.'` | `'x'` | ...

Marker symbol for detections, specified as the comma-separated pair consisting of 'Marker' and one of the markers in this table.

Value	Description
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross
'_'	Horizontal line
'|'	Vertical line
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No markers

MarkerSize — Size of marker for detections

6 (default) | positive integer

Size of marker, specified as the comma-separated pair consisting of 'MarkerSize' and a positive integer in points.

MarkerEdgeColor — Marker outline color for detections

[0 0 0] (black) (default) | RGB triplet | hexadecimal color code | color name | short color name

Marker outline color for detections, specified as the comma-separated pair consisting of 'MarkerEdgeColor' and an RGB triplet, a hexadecimal color code, a color name, or a short color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | RGB triplet | hexadecimal color code | color name | short color name

Marker fill color, specified as the comma-separated pair consisting of `'MarkerFaceColor'` and an RGB triplet, a hexadecimal color code, a color name, or a short color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

FontSize — Font size for labeling detections
10 points (default) | positive integer

Font size for labeling detections, specified as the comma-separated pair consisting of 'FontSize' and a positive integer in font points.

LabelOffset — Gap between label and positional point
[0 0] (default) | real-valued vector of the form [*x* *y*]

Gap between label and positional point, specified as the comma-separated pair consisting of 'LabelOffset' and a real-valued vector of the form [*x* *y*]. Units are in meters.

VelocityScaling — Scale factor for magnitude length of velocity vectors
1 (default) | positive real scalar

Scale factor for magnitude length of velocity vectors, specified as the comma-separated pair consisting of 'VelocityScaling' and a positive real scalar. The bird's-eye plot renders the magnitude vector value as M × VelocityScaling, where M is the magnitude of velocity.

Tag — Tag associated with plotter object
'Plotter*N*' (default) | character vector | string scalar

Tag associated with the plotter object, specified as the comma-separated pair consisting of `'Tag'` and a character vector or string scalar. The default value is `'Plotter`*N*`'`, where *N* is an integer that corresponds to the *N*th plotter associated with the input `birdsEyePlot` object.

Output Arguments

`detPlotter` — Detection plotter
`DetectionPlotter` object

Detection plotter, returned as a `DetectionPlotter` object. You can modify this object by changing its property values. The property names correspond to the name-value pair arguments of the `detectionPlotter` function.

`detPlotter` is stored in the `Plotters` property of the input `birdsEyePlot` object, `bep`. To plot the detections, use the `plotDetection` function.

See Also

`birdsEyePlot` | `clearData` | `clearPlotterData` | `findPlotter` | `plotDetection`

Introduced in R2017a

findPlotter

Find plotters associated with bird's-eye plot

Syntax

```
p = findPlotter(bep)
p = findPlotter(bep,Name,Value)
```

Description

`p = findPlotter(bep)` returns an array of plotters associated with a bird's-eye plot.

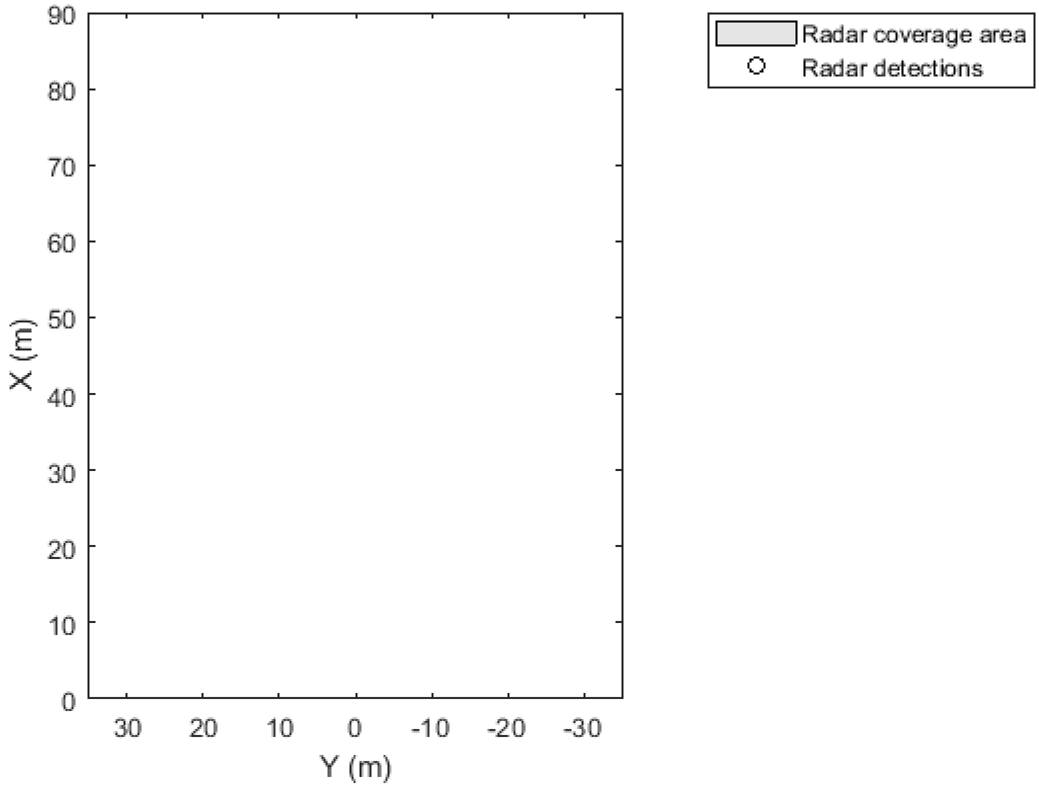
`p = findPlotter(bep,Name,Value)` specifies options using one or more `Name,Value` pair arguments. For example, `findPlotter(bep,'Tag','Plotter1')` returns the plotter object whose `Tag` property value is `'Plotter1'`.

Examples

Create Bird's-Eye Plot with Coverage Area and Detection Plotters

Create a bird's-eye plot with an *x*-axis range of 0 to 90 meters and a *y*-axis range from –35 to 35 meters. Configure the plot to include a radar coverage area plotter and a detection plotter. Set the display names of these plotters.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
coverageAreaPlotter(bep,'DisplayName','Radar coverage area');
detectionPlotter(bep,'DisplayName','Radar detections');
```

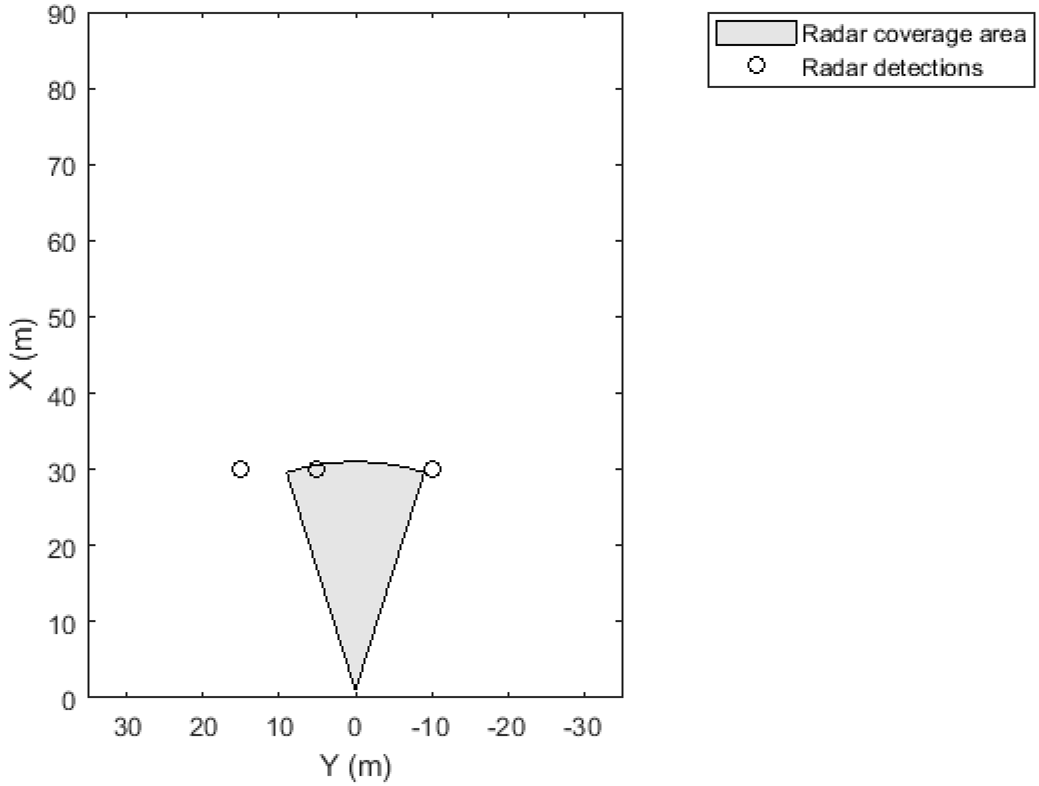


Use `findPlotter` to locate the plotters by their display names.

```
caPlotter = findPlotter(bep,'DisplayName','Radar coverage area');
radarPlotter = findPlotter(bep,'DisplayName','Radar detections');
```

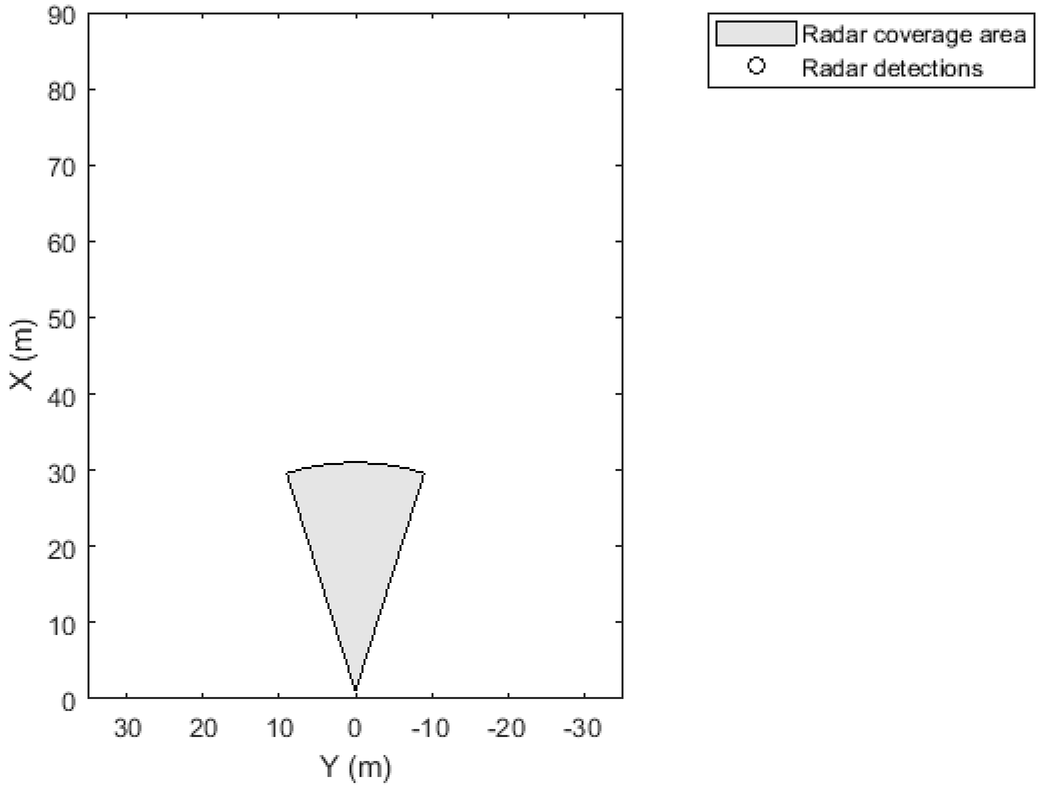
Plot the coverage area and detected objects.

```
plotCoverageArea(caPlotter,[1 0],30,0,35);
plotDetection(radarPlotter,[30 5; 30 -10; 30 15]);
```



Clear data from the plot.

```
clearPlotterData(bep);
```



Input Arguments

bep — Bird’s-eye plot
`birdsEyePlot` object

Bird’s-eye plot, specified as a `birdsEyePlot` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'DisplayName'`,`'MyBirdsEyePlots'`

`DisplayName` — Display name of plotter to find
character vector | string scalar

Display name of the plotter to find, specified as the comma-separated pair consisting of `'DisplayName'` and a character vector or string scalar. `DisplayName` is the plotter name that appears in the legend of the bird's-eye plot. To match missing legend entries, specify `DisplayName` as `''`.

Tag — Tag of plotter to find
`'Plotter*N*'` (default) | character vector | string scalar

Tag of plotter to find, specified as the comma-separated pair consisting of `'Tag'` and a character vector or string scalar. By default, plotter objects have a `Tag` property with a default value of `'Plotter`*`N`*`'`. *N* is an integer that corresponds to the *N*th plotter associated with the specified `birdsEyePlot` object, `bep`.

Output Arguments

p — Plotters associated with input bird's-eye plot
array of plotter objects

Plotters associated with the input bird's-eye plot, returned as an array of plotter objects.

See Also

Functions
`birdsEyePlot` | `clearData` | `clearPlotterData`

Introduced in R2017a

laneBoundaryPlotter

Package:

Lane boundary plotter for bird's-eye plot

Syntax

```
lbPlotter = laneBoundaryPlotter(bep)
lbPlotter = laneBoundaryPlotter(bep,Name,Value)
```

Description

`lbPlotter = laneBoundaryPlotter(bep)` creates a `LaneBoundaryPlotter` object that configures the display of lane boundaries on a bird's-eye plot. The `LaneBoundaryPlotter` object is stored in the `Plotters` property of the input `birdsEyePlot` object, `bep`. To display the lane boundaries, use the `plotLaneBoundary` function.

`lbPlotter = laneBoundaryPlotter(bep,Name,Value)` sets properties using one or more `Name,Value` pair arguments. For example, `laneBoundaryPlotter(bep,'DisplayName','Lane boundaries')` sets the display name that appears in the bird's-eye-plot legend.

Examples

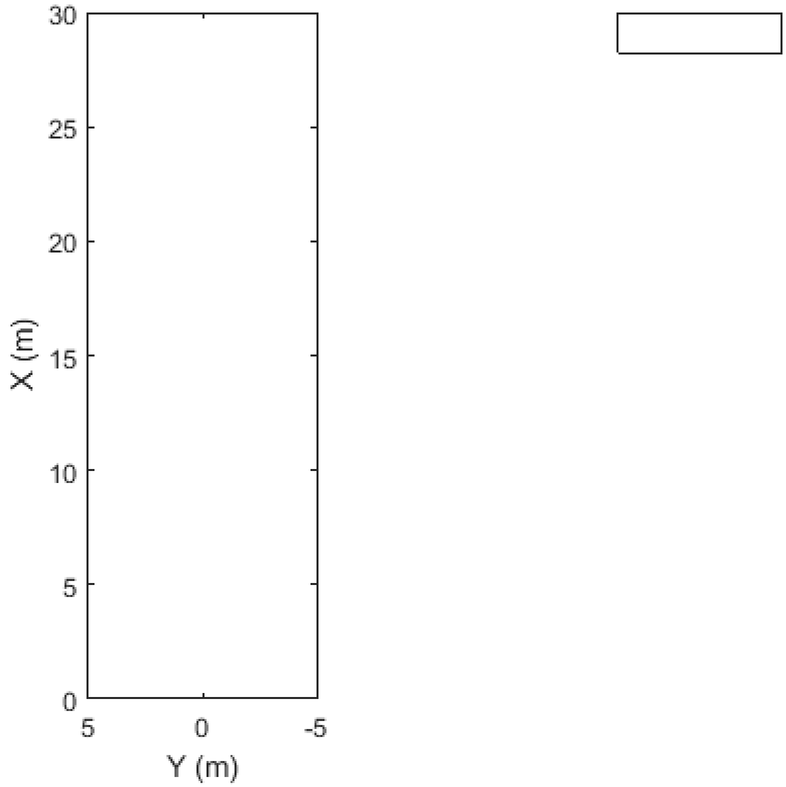
Create and Display Lane Boundaries on Bird's-Eye Plot

Create left-lane and right-lane boundaries.

```
leftlb = parabolicLaneBoundary([-0.001,0.01,-1.8]);
rightlb = parabolicLaneBoundary([-0.001,0.01,1.8]);
```

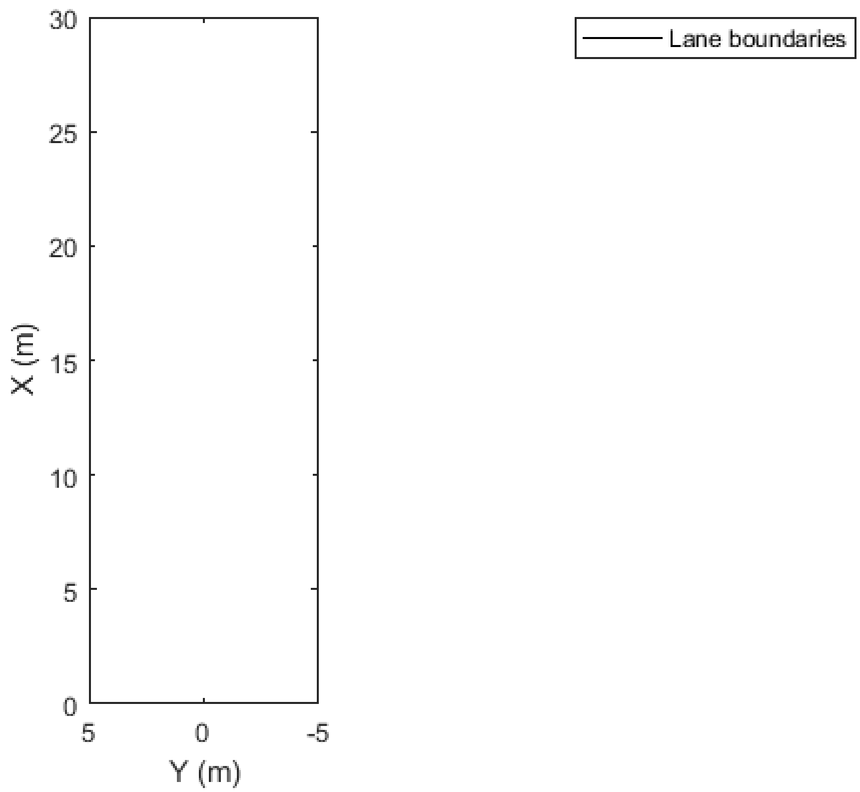
Create a bird's-eye plot with an *x*-axis range from 0 to 30 meters and a *y*-axis range from –5 to 5 meters.

```
bep = birdsEyePlot('XLimits',[0 30],'YLimits',[-5 5]);
```



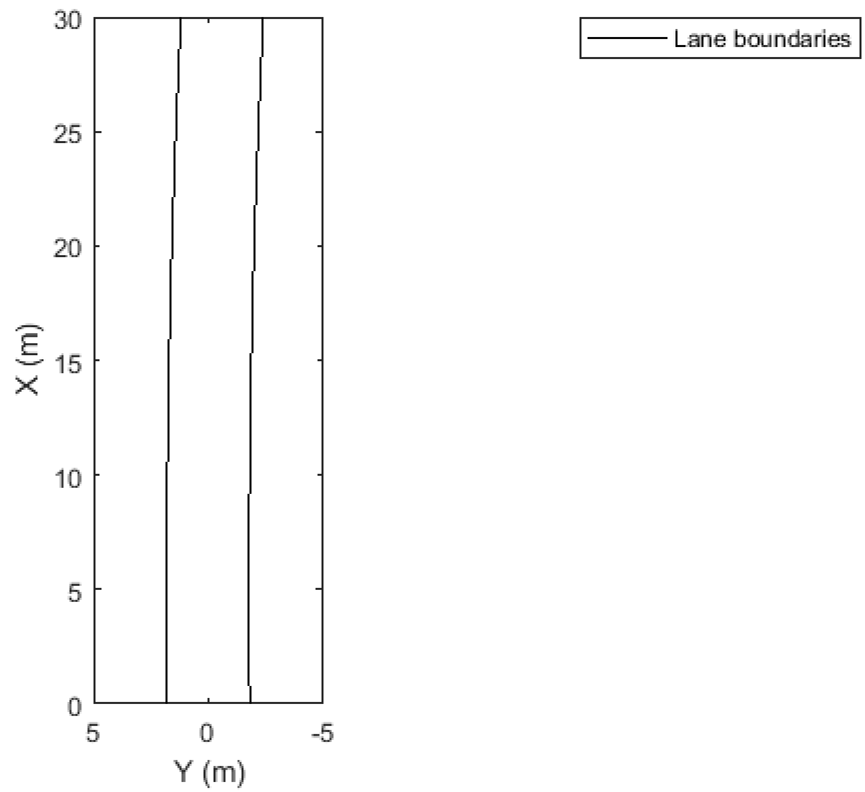
Create a lane boundary plotter.

```
lbPlotter = laneBoundaryPlotter(bep,'DisplayName','Lane boundaries');
```



Display the lane boundaries on the bird's-eye plot.

```
plotLaneBoundary(lbPlotter,[leftlb rightlb]);
```



Input Arguments

bep — Bird's-eye plot
`birdsEyePlot` object

Bird's-eye plot, specified as a `birdsEyePlot` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `laneBoundaryPlotter('Color','red')` sets the color of lane boundaries to red.

`DisplayName` — Plotter name to display in legend
`''` (default) | character vector | string scalar

Plotter name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and character vector or string scalar. If you do not specify a name, the bird's-eye plot does not display a legend entry for the plotter.

`Color` — Lane boundary color
`[0 0 0]` (black) (default) | RGB triplet | hexadecimal color code | color name | short color name

Lane boundary color, specified as the comma-separated pair consisting of 'Color' and an RGB triplet, a hexadecimal color code, a color name, or a short color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

LineStyle — Lane boundary line style
'-' (default) | '--' | ':' | '-.' | 'none'

Lane boundary line style, specified as the comma-separated pair consisting of 'LineStyle' and one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

Tag — Tag associated with plotter object
`'PlotterN'` (default) | character vector | string scalar

Tag associated with the plotter object, specified as the comma-separated pair consisting of `'Tag'` and a character vector or string scalar. The default value is `'PlotterN'`, where *N* is an integer that corresponds to the *N*th plotter associated with the input `birdsEyePlot` object.

Output Arguments

`lbPlotter` — Lane boundary plotter
`LaneBoundaryPlotter` object

Lane boundary plotter, returned as a `LaneBoundaryPlotter` object. You can modify this object by changing its property values. The property names correspond to the name-value pair arguments of the `laneBoundaryPlotter` function.

`lbPlotter` is stored in the `Plotters` property of the input `birdsEyePlot` object, `bep`. To plot the lane boundaries, use the `plotLaneBoundary` function.

See Also
`birdsEyePlot` | `clearData` | `clearPlotterData` | `findPlotter` | `plotLaneBoundary`

Introduced in R2017a

laneMarkingPlotter

Package:

Lane marking plotter for bird's-eye plot

Syntax

```
lmPlotter = laneMarkingPlotter(bep)
lmPlotter = laneMarkingPlotter(bep,Name,Value)
```

Description

`lmPlotter = laneMarkingPlotter(bep)` creates a `LaneMarkingPlotter` object that configures the display of lane markings on a bird's-eye plot. The `LaneMarkingPlotter` object is stored in the `Plotters` property of the input `birdsEyePlot` object, `bep`. To display the lane markings, use the `plotLaneMarking` function.

`lmPlotter = laneMarkingPlotter(bep,Name,Value)` sets properties using one or more `Name,Value` pair arguments. For example, `laneMarkingPlotter(bep,'DisplayName','Lane markings')` sets the display name that appears in the bird's-eye-plot legend.

Examples

Generate Object and Lane Boundary Detections

Create a driving scenario containing an ego vehicle and a target vehicle traveling along a three-lane road. Detect the lane boundaries by using a vision detection generator.

```
scenario = drivingScenario;
```

Create a three-lane road by using lane specifications.

```
roadCenters = [0 0 0; 60 0 0; 120 30 0];
lspc = lanespec(3);
road(scenario,roadCenters,'Lanes',lspc);
```

Specify that the ego vehicle follows the center lane at 30 m/s.

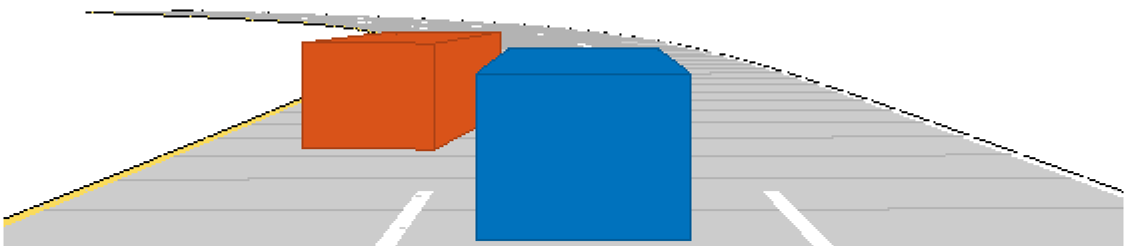
```
egovehicle = vehicle(scenario,'ClassID',1);
egopath = [1.5 0 0; 60 0 0; 111 25 0];
egospeed = 30;
trajectory(egovehicle,egopath,egospeed);
```

Specify that the target vehicle travels ahead of the ego vehicle at 40 m/s and changes lanes close to the ego vehicle.

```
targetcar = vehicle(scenario,'ClassID',1);
targetpath = [8 2; 60 -3.2; 120 33];
targetspeed = 40;
trajectory(targetcar,targetpath,targetspeed);
```

Display a chase plot for a 3-D view of the scenario from behind the ego vehicle.

```
chasePlot(egovehicle)
```



Create a vision detection generator that detects lanes and objects. The pitch of the sensor points one degree downward.

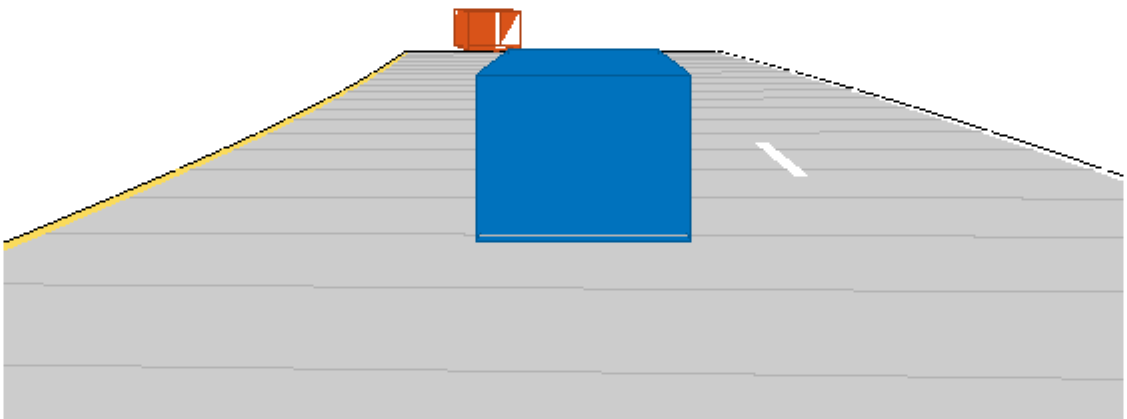
```
visionSensor = visionDetectionGenerator('Pitch',1.0);
visionSensor.DetectorOutput = 'Lanes and objects';
visionSensor.ActorProfiles = actorProfiles(scenario);
```

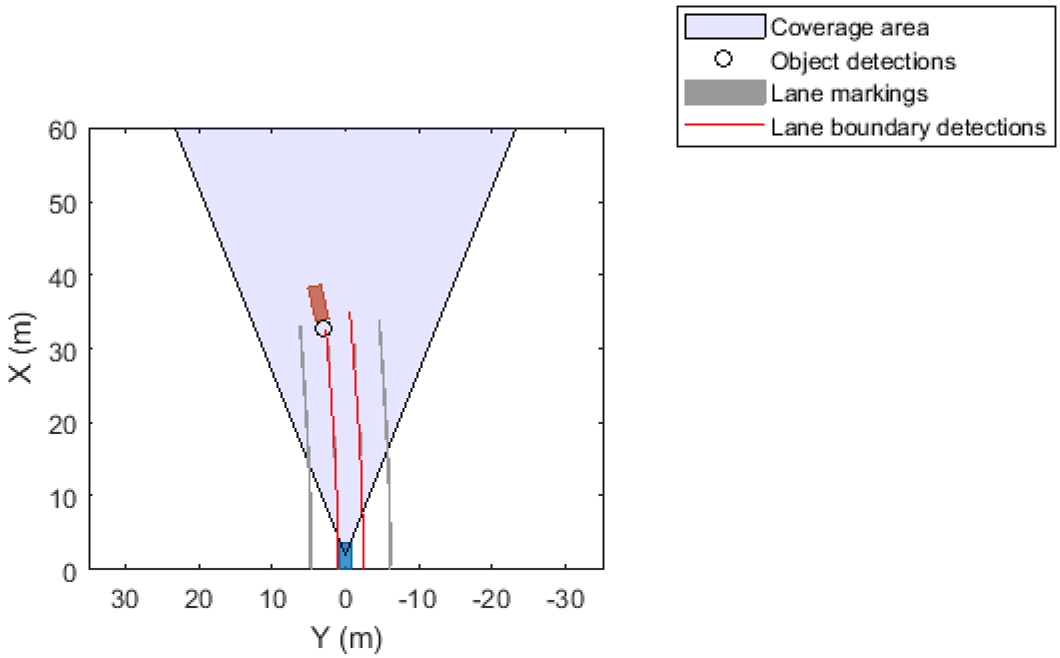
Run the simulation.

1 Create a bird's-eye plot and the associated plotters.
2 Display the sensor coverage area.
3 Display the lane markings.
4 Obtain ground truth poses of targets on the road.
5 Obtain ideal lane boundary points up to 60 m ahead.
6 Generate detections from the ideal target poses and lane boundaries.
7 Display the outline of the target.
8 Display object detections when the object detection is valid.
9 Display the lane boundary when the lane detection is valid.

```
bep = birdsEyePlot('XLim',[0 60],'YLim',[-35 35]);
caPlotter = coverageAreaPlotter(bep,'DisplayName','Coverage area', ...
```

```
    'FaceColor','blue');
detPlotter = detectionPlotter(bep,'DisplayName','Object detections');
lmPlotter = laneMarkingPlotter(bep,'DisplayName','Lane markings');
lbPlotter = laneBoundaryPlotter(bep,'DisplayName', ...
    'Lane boundary detections','Color','red');
olPlotter = outlinePlotter(bep);
plotCoverageArea(caPlotter,visionSensor.SensorLocation,...
    visionSensor.MaxRange,visionSensor.Yaw, ...
    visionSensor.FieldOfView(1));
while advance(scenario)
    [lmv,lmf] = laneMarkingVertices(egovehicle);
    plotLaneMarking(lmPlotter,lmv,lmf)
    tgtpose = targetPoses(egovehicle);
    lookaheadDistance = 0:0.5:60;
    lb = laneBoundaries(egovehicle,'XDistance',lookaheadDistance,'LocationType','inner');
    [obdets,nobdets,obValid,lb_dets,nlb_dets,lbValid] = ...
        visionSensor(tgtpose,lb,scenario.SimulationTime);
    [objposition,objyaw,objlength,objwidth,objoriginOffset,color] = targetOutlines(egovehicle);
    plotOutline(olPlotter,objposition,objyaw,objlength,objwidth, ...
        'OriginOffset',objoriginOffset,'Color',color)
    if obValid
        detPos = cellfun(@(d)d.Measurement(1:2),obdets,'UniformOutput',false);
        detPos = vertcat(zeros(0,2),cell2mat(detPos')');
        plotDetection(detPlotter,detPos)
    end
    if lbValid
        plotLaneBoundary(lbPlotter,vertcat(lb_dets.LaneBoundaries))
    end
end
```





Input Arguments

bep — Bird’s-eye plot
`birdsEyePlot` object

Bird’s-eye plot, specified as a `birdsEyePlot` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `laneBoundaryPlotter('Color','red')` sets the color of lane markings to red.

`DisplayName` — Plotter name to display in legend
`''` (default) | character vector | string scalar

Plotter name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and character vector or string scalar. If you do not specify a name, the bird's-eye plot does not display a legend entry for the plotter.

`FaceColor` — Face color of lane marking patches
`[0.6 0.6 0.6]` (gray) (default) | RGB triplet | color name

Face color of lane marking patches, specified as the comma-separated pair consisting of 'FaceColor' and an RGB triplet or one of the color names listed in the table.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0, 1]; for example, [0.4 0.6 0.7]. Alternatively, you can specify some common colors by name. This table lists the named color options and the equivalent RGB triplet values.

Color Name	RGB Triplet	Appearance
'red'	[1 0 0]	
'green'	[0 1 0]	
'blue'	[0 0 1]	
'cyan'	[0 1 1]	
'magenta'	[1 0 1]	
'yellow'	[1 1 0]	
'black'	[0 0 0]	
'white'	[1 1 1]	

Tag — Tag associated with plotter object
'Plotter*N*' (default) | character vector | string scalar

Tag associated with the plotter object, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. The default value is 'Plotter*N*', where *N* is an integer that corresponds to the *N*th plotter associated with the input birdsEyePlot object.

Output Arguments

lmPlotter — Lane marking plotter
LaneMarkingPlotter object

Lane marking plotter, returned as a LaneMarkingPlotter object. You can modify this object by changing its property values. The property names correspond to the name-value pair arguments of the laneMarkingPlotter function.

lmPlotter is stored in the Plotters property of the input birdsEyePlot object, bep. To plot the lane markings, use the plotLaneMarking function.

See Also

birdsEyePlot | clearData | clearPlotterData | findPlotter | plotLaneMarking

Introduced in R2018a

meshPlotter

Package:

Mesh plotter for bird's-eye plot

Syntax

```
mPlotter = meshPlotter(bep)
mPlotter = meshPlotter(bep,Name,Value)
```

Description

`mPlotter = meshPlotter(bep)` creates a `MeshPlotter` object that configures the display of meshes on page 4-56 on a bird's-eye plot. The `MeshPlotter` object is stored in the `Plotters` property of the input `birdsEyePlot` object, `bep`. To display the mesh representations of objects, use the `plotMesh` function.

`mPlotter = meshPlotter(bep,Name,Value)` sets properties using one or more `Name,Value` pair arguments. For example, `meshPlotter(bep,'FaceAlpha',1)` sets the mesh faces to be fully opaque.

Examples

Display Actor Meshes in Driving Scenario

Display actors in a driving scenario by using their mesh representations instead of their cuboid representations.

Create a driving scenario, and add a 25-meter straight road to the scenario.

```
scenario = drivingScenario;
roadcenters = [0 0 0; 25 0 0];
road(scenario,roadcenters);
```

Add a pedestrian and a vehicle to the scenario. Specify the mesh dimensions of the actors using prebuilt meshes.

- Specify the pedestrian mesh as a `driving.scenario.pedestrianMesh` object.
- Specify the vehicle mesh as a `driving.scenario.carMesh` object.

```
p = actor(scenario,'ClassID',4, ...
            'Length',0.2,'Width',0.4, ...
            'Height',1.7,'Mesh',driving.scenario.pedestrianMesh);

v = vehicle(scenario,'ClassID',1, ...
            'Mesh',driving.scenario.carMesh);
```

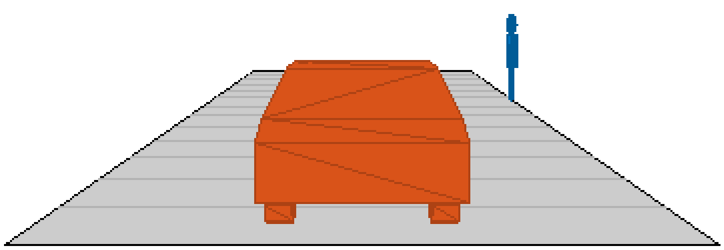
Add trajectories for the pedestrian and vehicle.

- Specify for the pedestrian to cross the road at 1 meter per second.
- Specify for the vehicle to follow the road at 10 meters per second.

```
waypointsP = [15 -3 0; 15 3 0];
speedP = 1;
trajectory(p,waypointsP,speedP);

wayPointsV = [v.RearOverhang 0 0; (25 - v.Length + v.RearOverhang) 0 0];
speedV = 10;
trajectory(v,wayPointsV,speedV)
```

Add an egocentric plot for the vehicle. Turn the display of meshes on.

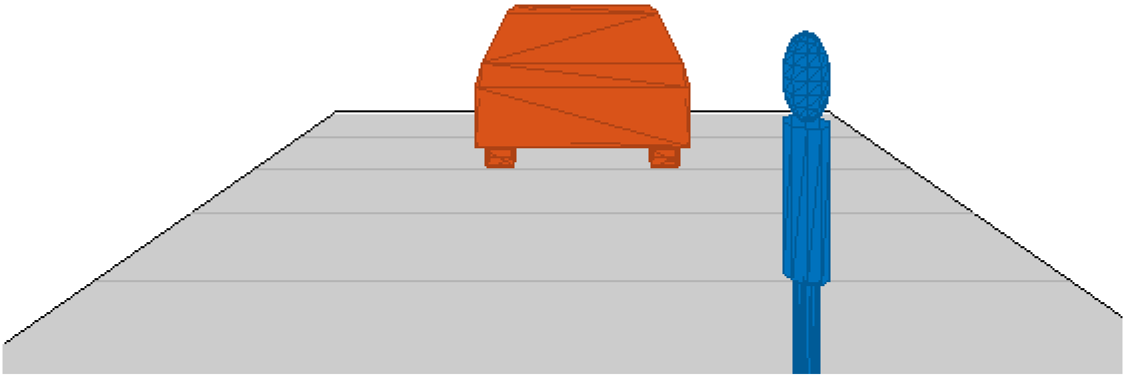
```
chasePlot(v,'Meshes','on')
```

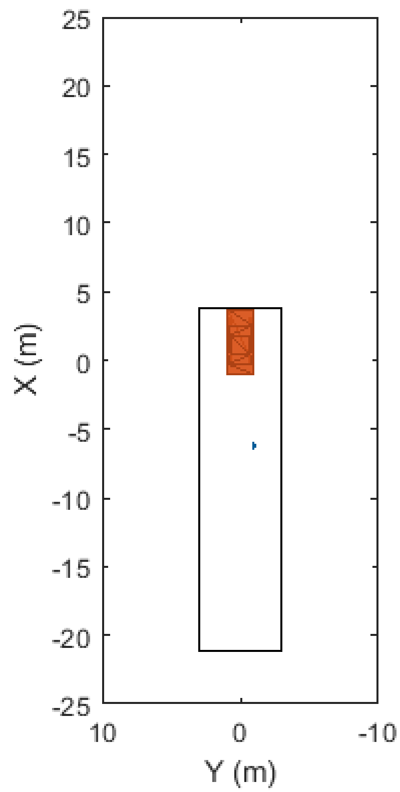


Create a bird's-eye plot in which to display the meshes. Also create a mesh plotter and lane boundary plotter. Then run the simulation loop.

1. Obtain the road boundaries of the road the vehicle is on.
2. Obtain the mesh vertices, faces, and colors of the actor meshes, with positions relative to the vehicle.
3. Plot the road boundaries and actor meshes on the bird's-eye plot.
4. Pause the scenario to allow time for the plots to update. The chase plot updates every time you advance the scenario.

```
bep = birdsEyePlot('XLim',[-25 25],'YLim',[-10 10]);
mPlotter = meshPlotter(bep);
lbPlotter = laneBoundaryPlotter(bep);
legend('off')

while advance(scenario)

   rb = roadBoundaries(v);

   [vertices,faces,colors] = targetMeshes(v);

   plotLaneBoundary(lbPlotter,rb)
   plotMesh(mPlotter,vertices,faces,'Color',colors)

   pause(0.01)
end
```





Input Arguments

bep — Bird’s-eye plot
`birdsEyePlot` object

Bird’s-eye plot, specified as a `birdsEyePlot` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `meshPlotter('FaceAlpha',0.5)` sets the mesh faces to be 50% transparent.

FaceAlpha — Transparency of mesh faces
`0.75` (default) | scalar in the range [0, 1]

Transparency of mesh faces, specified as the comma-separated pair consisting of `'FaceAlpha'` and a scalar in the range [0, 1]. A value of `0` makes the mesh faces fully transparent. A value of `1` makes the mesh faces fully opaque.

Tag — Tag associated with plotter object
`'Plotter`*`N`*`'` (default) | character vector | string scalar

Tag associated with the plotter object, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. The default value is 'Plotter*N*', where *N* is an integer that corresponds to the *N*th plotter associated with the input birdsEyePlot object.

Output Arguments

mPlotter — Mesh plotter
MeshPlotter object

Mesh plotter, returned as a MeshPlotter object. You can modify this object by changing its property values. The property names correspond to the name-value pair arguments of the meshPlotter function.

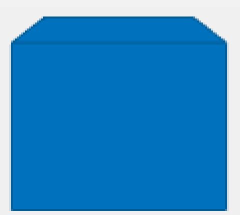
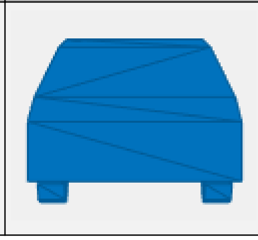
mPlotter is stored in the Plotters property of the input birdsEyePlot object, bep. To plot the meshes, use the plotMesh function.

More About

Meshes

In driving scenarios, a mesh is a triangle-based 3-D representation of an object. Mesh representations of objects are more detailed than the default cuboid (box-shaped) representations of objects. Meshes are useful for generating synthetic point cloud data from a driving scenario.

This table shows the difference between a cuboid representation and a mesh representation of a vehicle in a driving scenario.

Cuboid	Mesh
	

See Also

birdsEyePlot | clearData | clearPlotterData | plotMesh | targetMeshes

Introduced in R2020b

outlinePlotter

Package:

Outline plotter for bird's-eye plot

Syntax

```
olPlotter = outlinePlotter(bep)
olPlotter = outlinePlotter(bep,Name,Value)
```

Description

`olPlotter = outlinePlotter(bep)` creates an `OutlinePlotter` object that configures the display of object outlines on a bird's-eye plot. The `OutlinePlotter` object is stored in the `Plotters` property of the `birdsEyePlot` object, `bep`. To display the outlines of actors that are in a driving scenario, first use `targetOutlines` to get the dimensions of the actors. Then, after creating an outline plotter object, use the `plotOutline` function to display the outlines of all the actors in the bird's-eye plot.

`olPlotter = outlinePlotter(bep,Name,Value)` sets properties using one or more `Name,Value` pair arguments. For example, `outlinePlotter(bep,'FaceAlpha',0)` sets the areas within each outline to be fully transparent.

Examples

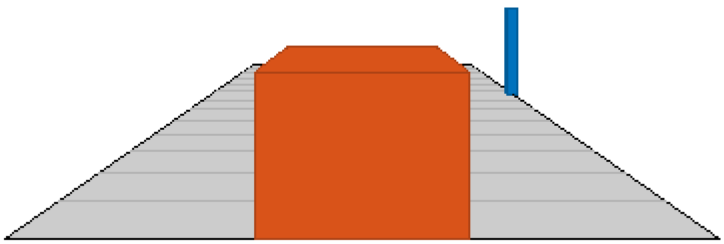
Plot Outlines of Targets on Bird's-Eye Plot

Create a driving scenario. Create a 25 m road segment, add a pedestrian and a vehicle, and specify their trajectories to follow. The pedestrian crosses the road at 1 m/s. The vehicle drives along the road at 10 m/s.

```
scenario = drivingScenario;

road(scenario,[0 0 0; 25 0 0]);

p = actor(scenario,'ClassID',4,'Length',0.2,'Width',0.4,'Height',1.7);
v = vehicle(scenario,'ClassID',1);

trajectory(p,[15 -3 0; 15 3 0],1);
trajectory(v,[v.RearOverhang 0 0; 25-v.Length+v.RearOverhang 0 0], 10);
```

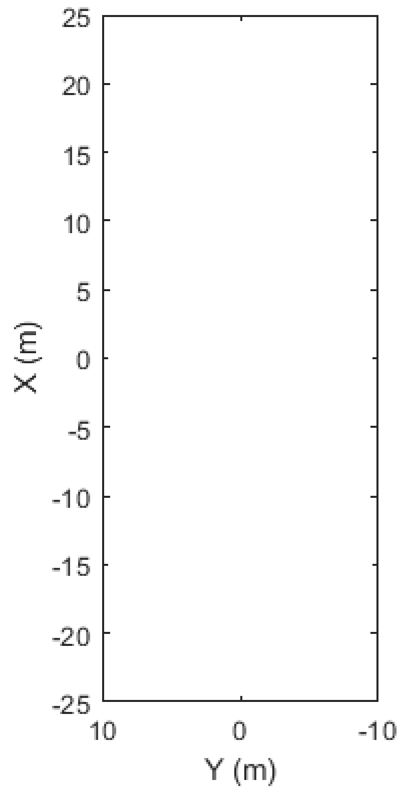
Use a chase plot to display the scenario from the perspective of the vehicle.

```
chasePlot(v,'Centerline','on')
```



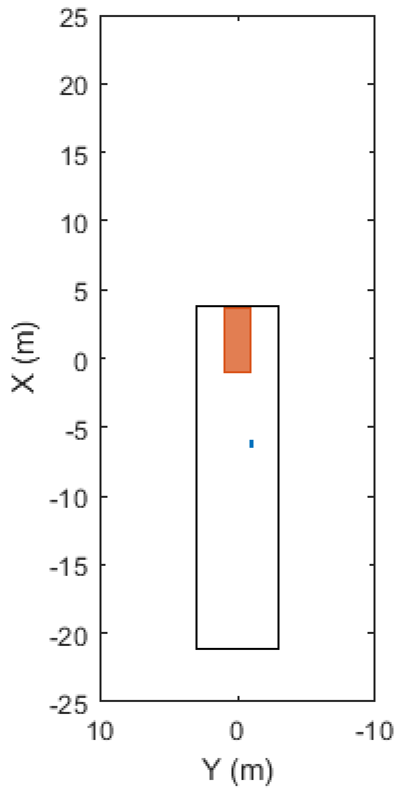
Create a bird's-eye plot, outline plotter, and lane boundary plotter.

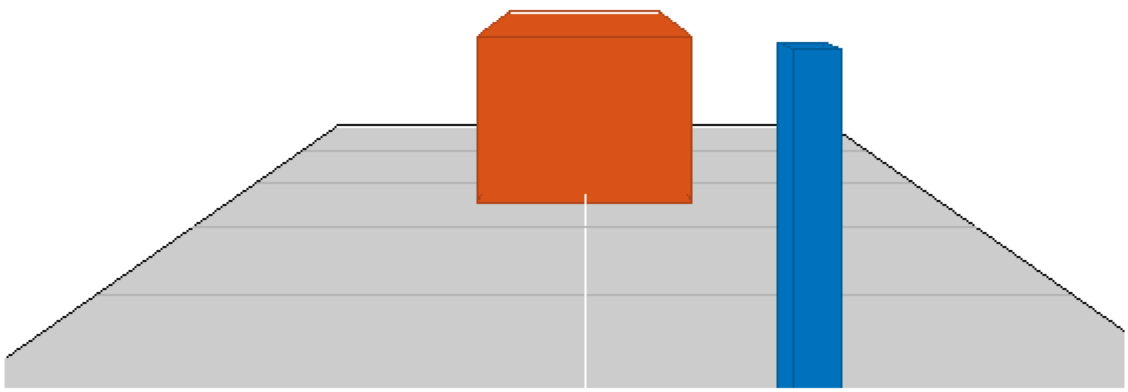
```
bep = birdsEyePlot('XLim',[-25 25],'YLim',[-10 10]);
olPlotter = outlinePlotter(bep);
lbPlotter = laneBoundaryPlotter(bep);
legend('off')
```



Run the simulation loop. Update the plotter with outlines for the targets.

```
while advance(scenario)
   % Obtain the road boundaries and rectangular outlines.
   rb = roadBoundaries(v);
   [position,yaw,length,width,originOffset,color] = targetOutlines(v);

   % Update the bird's-eye plotters with the road and actors.
   plotLaneBoundary(lbPlotter,rb);
   plotOutline(olPlotter,position,yaw,length,width, ...
               'OriginOffset',originOffset,'Color',color);

   % Allow time for plot to update.
   pause(0.01)
end
```


25
20
15
10
5
0
-5
-10
-15
-20
-25
X (m)
10
0
-10
Y (m)



Input Arguments

bep — Bird's-eye plot
`birdsEyePlot` object

Bird's-eye plot, specified as a `birdsEyePlot` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `outlinePlotter('FaceAlpha',1)` sets the areas within each outline to be fully opaque.

FaceAlpha — Transparency of area within each outline
`0.75` (default) | real scalar

Transparency of the area within each outline, specified as the comma-separated pair consisting of `'FaceAlpha'` and a real scalar in the range [0, 1]. A value of `0` makes the areas fully transparent. A value of `1` makes the areas fully opaque.

Tag — Tag associated with plotter object
`'Plotter`*`N`*`'` (default) | character vector | string scalar

Tag associated with the plotter object, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. The default value is 'Plotter*N*', where *N* is an integer that corresponds to the *N*th plotter associated with the input birdsEyePlot object.

Output Arguments

olPlotter — Outline plotter
OutlinePlotter object

Outline plotter, returned as an OutlinePlotter object. You can modify this object by changing its property values. The property names correspond to the name-value pair arguments of the outlinePlotter function.

olPlotter is stored in the Plotters property of a birdsEyePlot object. To plot the outlines of actors that are in a driving scenario, first use targetOutlines to get the dimensions of the actors. Then, after calling outlinePlotter to create a plotter object, use plotOutline to plot the outlines of all the actors in a bird's-eye plot.

See Also

birdsEyePlot | clearData | clearPlotterData | findPlotter | plotOutline

Introduced in R2017b

pathPlotter

Package:

Path plotter for bird's-eye plot

Syntax

```
pPlotter = pathPlotter(bep)
pPlotter = pathPlotter(bep,Name,Value)
```

Description

`pPlotter = pathPlotter(bep)` creates a `PathPlotter` object that configures the display of actor paths on a bird's-eye plot. The `PathPlotter` object is stored in the `Plotters` property of the input `birdsEyePlot` object, `bep`. To display the paths, use the `plotPath` function.

`pPlotter = pathPlotter(bep,Name,Value)` sets properties using one or more `Name,Value` pair arguments. For example, `pathPlotter(bep,'DisplayName','Actor paths')` sets the display name that appears in the bird's-eye-plot legend.

Examples

Plot Path of Ego Vehicle

Create a 3-meter-wide lane.

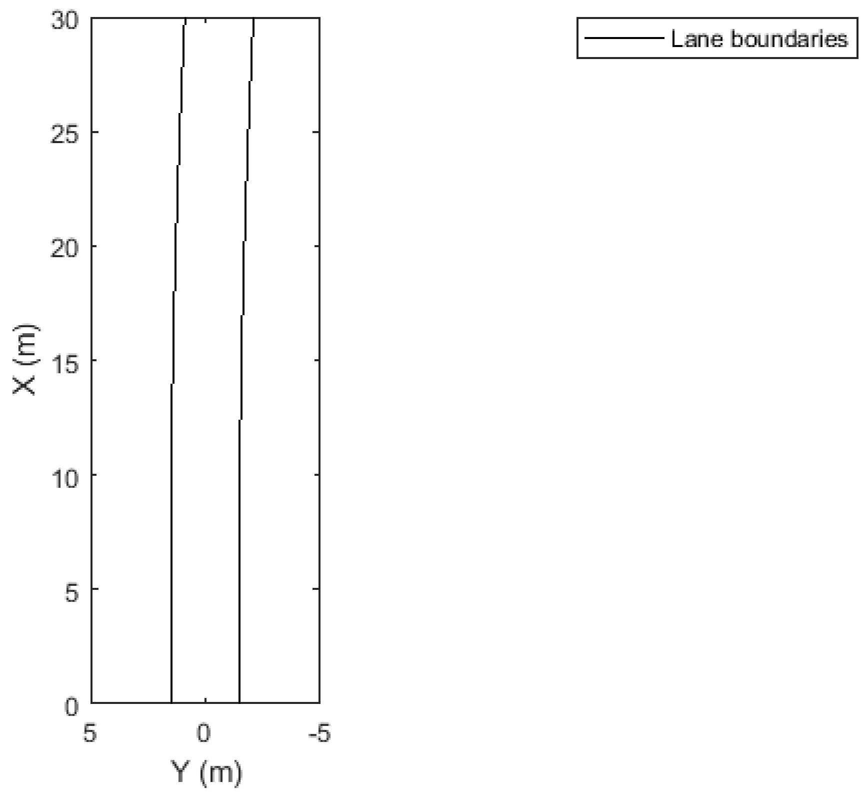
```
lb = parabolicLaneBoundary([-0.001,0.01,1.5]);
rb = parabolicLaneBoundary([-0.001,0.01,-1.5]);
```

Compute the lane boundary model manually from 0 to 30 meters along the *x*-axis.

```
xWorld = (0:30)';
yLeft = computeBoundaryModel(lb,xWorld);
yRight = computeBoundaryModel(rb,xWorld);
```

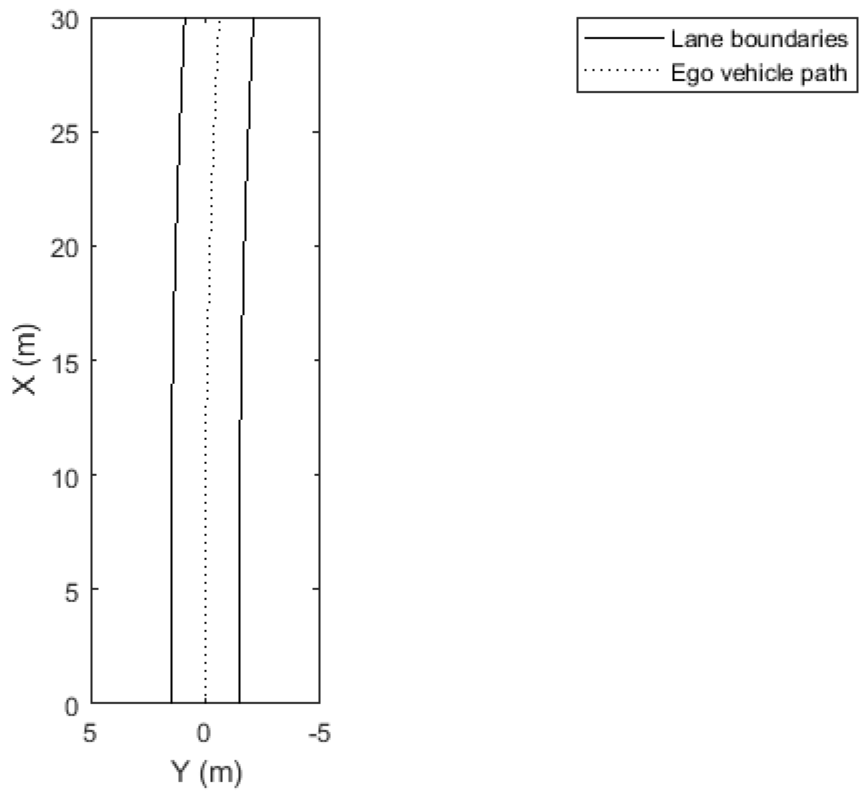
Create a bird's-eye plot and lane boundary plotter. Display the lane information on the bird's-eye plot.

```
bep = birdsEyePlot('XLimits',[0 30],'YLimits',[-5 5]);
lanePlotter = laneBoundaryPlotter(bep,'DisplayName','Lane boundaries');
plotLaneBoundary(lanePlotter,{[xWorld,yLeft],[xWorld,yRight]});
```



Create a path plotter. Create and display the path of an ego vehicle that travels through the center of the lane.

```
yCenter = (yLeft + yRight)/2;
egoPathPlotter = pathPlotter(bep,'DisplayName','Ego vehicle path');
plotPath(egoPathPlotter,{[xWorld,yCenter]});
```



Input Arguments

bep — Bird’s-eye plot
`birdsEyePlot` object

Bird’s-eye plot, specified as a `birdsEyePlot` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `pathPlotter('Color','red')` sets the color of the path to red.

`DisplayName` — Plotter name to display in legend
`''` (default) | character vector | string scalar

Plotter name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and character vector or string scalar. If you do not specify a name, the bird's-eye plot does not display a legend entry for the plotter.

`Color` — Path color
`[0 0 0]` (black) (default) | RGB triplet | hexadecimal color code | color name | short color name

Path color, specified as the comma-separated pair consisting of `'Color'` and an RGB triplet, a hexadecimal color code, a color name, or a short color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

LineStyle — Path line style
`':'` (default) | `'-'` | `'--'` | `'-.'` | `'none'`

Path line style, specified as the comma-separated pair consisting of `'LineStyle'` and one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

Tag — Tag associated with plotter object
`'Plotter`*`N`*`'` (default) | character vector | string scalar

Tag associated with the plotter object, specified as the comma-separated pair consisting of `'Tag'` and a character vector or string scalar. The default value is `'Plotter`*`N`*`'`, where *N* is an integer that corresponds to the *N*th plotter associated with the input `birdsEyePlot` object.

Output Arguments

`pPlotter` — Path plotter
`PathPlotter` object

Path plotter, returned as a `PathPlotter` object. You can modify this object by changing its property values. The property names correspond to the name-value pair arguments of the `pathPlotter` function.

`pPlotter` is stored in the `Plotters` property of the input `birdsEyePlot` object, `bep`. To plot the paths, use the `plotPath` function.

See Also
`birdsEyePlot` | `clearData` | `clearPlotterData` | `findPlotter` | `plotPath`

Introduced in R2017a

pointCloudPlotter

Package:

Point cloud plotter for bird's-eye plot

Syntax

```
pcPlotter = pointCloudPlotter(bep)
pcPlotter = pointCloudPlotter(bep,Name,Value)
```

Description

`pcPlotter = pointCloudPlotter(bep)` creates a point cloud plotter object that configures the display of lidar point cloud data on a bird's-eye plot. The point cloud plotter object is stored in the `Plotters` property of the input bird's-eye plot object, `bep`. To plot the lidar point cloud data, use the `plotPointCloud` function.

`pcPlotter = pointCloudPlotter(bep,Name,Value)` specifies options using one or more name-value pair arguments. For example, `'DisplayName','Point Cloud'` sets the display name that appears in the bird's-eye-plot legend to "Point Cloud".

Examples

Generate Lidar Point Cloud Data of Multiple Actors

Generate lidar point cloud data for a driving scenario with multiple actors by using the `lidarPointCloudGenerator` System object. Create the driving scenario by using `drivingScenario` object. It contains an ego-vehicle, pedestrian and two other vehicles.

Create and plot a driving scenario with multiple vehicles

Create a driving scenario.

```
scenario = drivingScenario;
```

Add a straight road to the driving scenario. The road has one lane in each direction.

```
roadCenters = [0 0 0; 70 0 0];
laneSpecification = lanespec([1 1]);
road(scenario,roadCenters,'Lanes',laneSpecification);
```

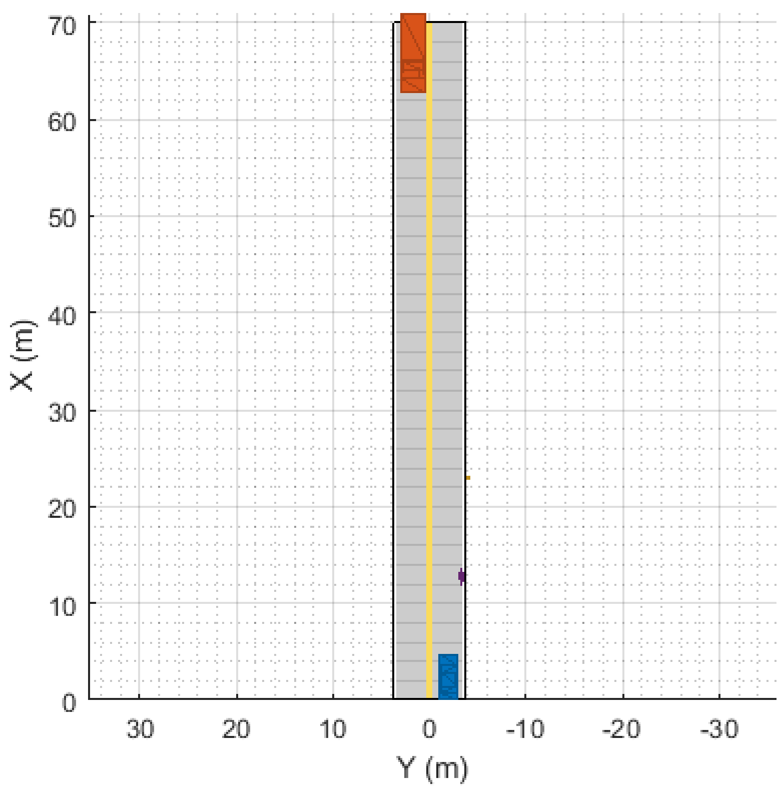
Add an ego vehicle to the driving scenario.

```
egoVehicle = vehicle(scenario,'ClassID',1,'Mesh',driving.scenario.carMesh);
waypoints = [1 -2 0; 35 -2 0];
trajectory(egoVehicle,waypoints,10);
```

Add a truck, pedestrian, and bicycle to the driving scenario and plot the scenario.

```
truck = vehicle(scenario,'ClassID',2,'Length', 8.2,'Width',2.5,'Height',3.5, ...
  'Mesh',driving.scenario.truckMesh);
```

```
waypoints = [70 1.7 0; 20 1.9 0];
trajectory(truck,waypoints,15);
pedestrian = actor(scenario,'ClassID',4,'Length',0.24,'Width',0.45,'Height',1.7, ...
  'Mesh',driving.scenario.pedestrianMesh);
waypoints = [23 -4 0; 10.4 -4 0];
trajectory(pedestrian,waypoints,1.5);
bicycle = actor(scenario,'ClassID',3,'Length',1.7,'Width',0.45,'Height',1.7, ...
  'Mesh',driving.scenario.bicycleMesh);
waypoints = [12.7 -3.3 0; 49.3 -3.3 0];
trajectory(bicycle,waypoints,5);
plot(scenario,'Meshes','on')
```



Generate and plot lidar point cloud data

Create a `lidarPointCloudGenerator` System object.

```
lidar = lidarPointCloudGenerator;
```

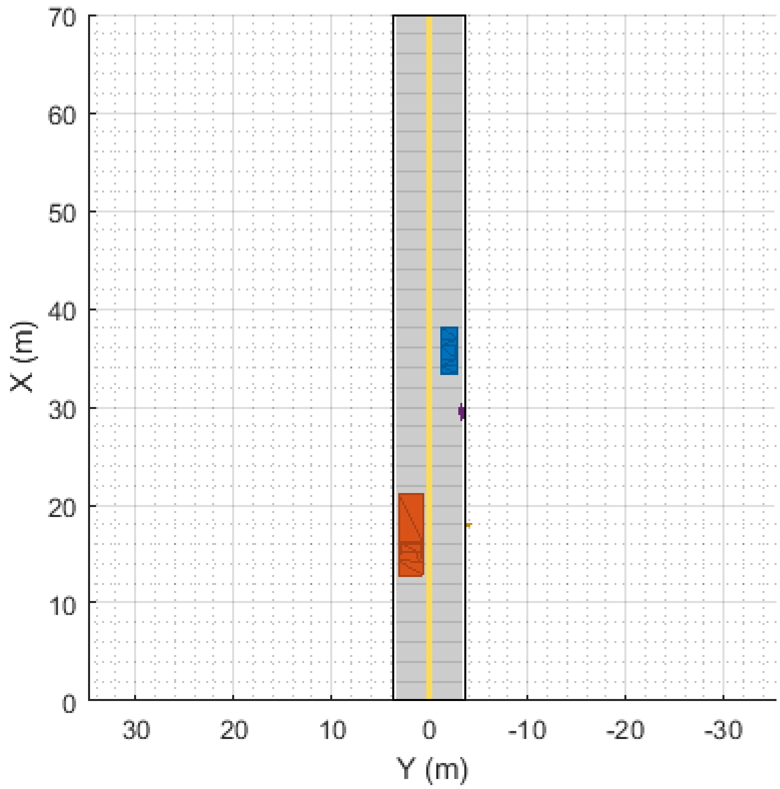
Add actor profiles and the ego vehicle actor ID from the driving scenario to the System object.

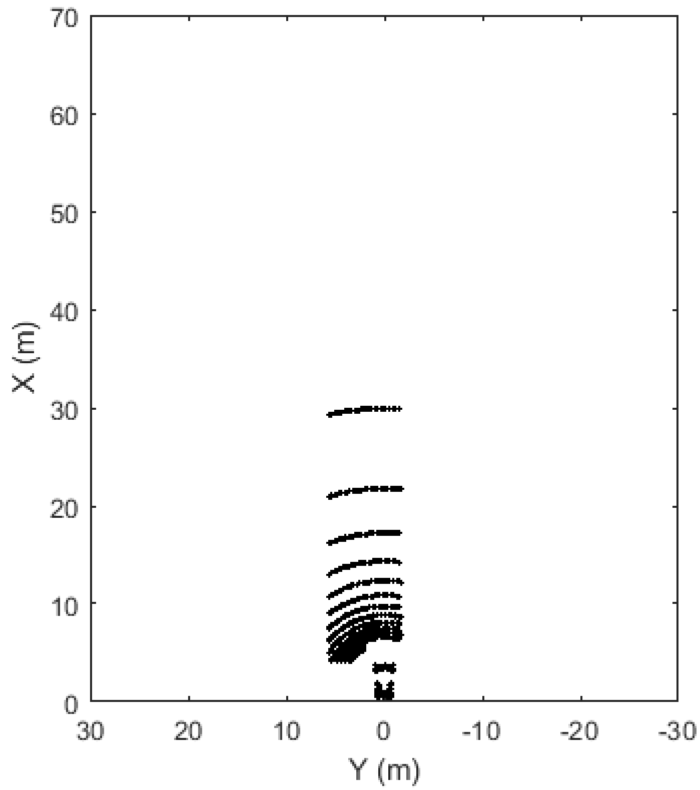
```
lidar.ActorProfiles = actorProfiles(scenario);
lidar.EgoVehicleActorID = egoVehicle.ActorID;
```

Plot the point cloud data.

```
bep = birdsEyePlot('Xlimits',[0 70],'YLimits',[-30 30]);
plotter = pointCloudPlotter(bep);
legend('off');
```

```
while advance(scenario)
    tgts = targetPoses(egoVehicle);
    rdmesh = roadMesh(egoVehicle);
    [ptCloud,isValidTime] = lidar(tgts,rdmesh,scenario.SimulationTime);
    if isValidTime
        plotPointCloud(plotter,ptCloud);
    end
end
```





Input Arguments

bep — Bird’s-eye plot
`birdsEyePlot` object

Bird’s-eye plot, specified as a `birdsEyePlot` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'DisplayName','Point Cloud'` sets the display name that appears in the bird's-eye-plot legend to "Point Cloud".

`DisplayName` — Plotter name to display in legend
`''` (default) | character vector | string scalar

Plotter name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and character vector or string scalar. If you do not specify a name, the bird's-eye plot does not display a legend entry for the plotter.

Data Types: `char` | `string`

`PointSize` — Size of marker for points in point cloud
`6` (default) | positive integer

Size of marker for points in a point cloud, specified as the comma-separated pair consisting of `'PointSize'` and a positive integer in points.

`Color` — Point fill color
`'none'` (default) | RGB triplet | hexadecimal color code | color name | short color name

Point fill color, specified as the comma-separated pair consisting of `'Color'` and an RGB triplet, a hexadecimal color code, a color name, or a short color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Tag — Tag associated with plotter object
'Plotter*N*' (default) | character vector | string scalar

Tag associated with the plotter object, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. The default value is 'Plotter*N*', where *N* is an integer that corresponds to the *N*th plotter associated with the input birdsEyePlot object.

Output Arguments

pcPlotter — Point cloud plotter
pointCloudPlotter object

Point cloud plotter, returned as a pointCloudPlotter object. You can modify this object by changing its property values.

pcPlotter is stored in the Plotters property of the input, bep. To plot the point cloud data, use the plotPointCloud function.

See Also

birdsEyePlot | clearData | clearPlotterData | findPlotter | plotPointCloud

Introduced in R2020a

plotCoverageArea

Display sensor coverage area on bird's-eye plot

Syntax

```
plotCoverageArea(caPlotter,position,range,orientation,fieldOfView)
```

Description

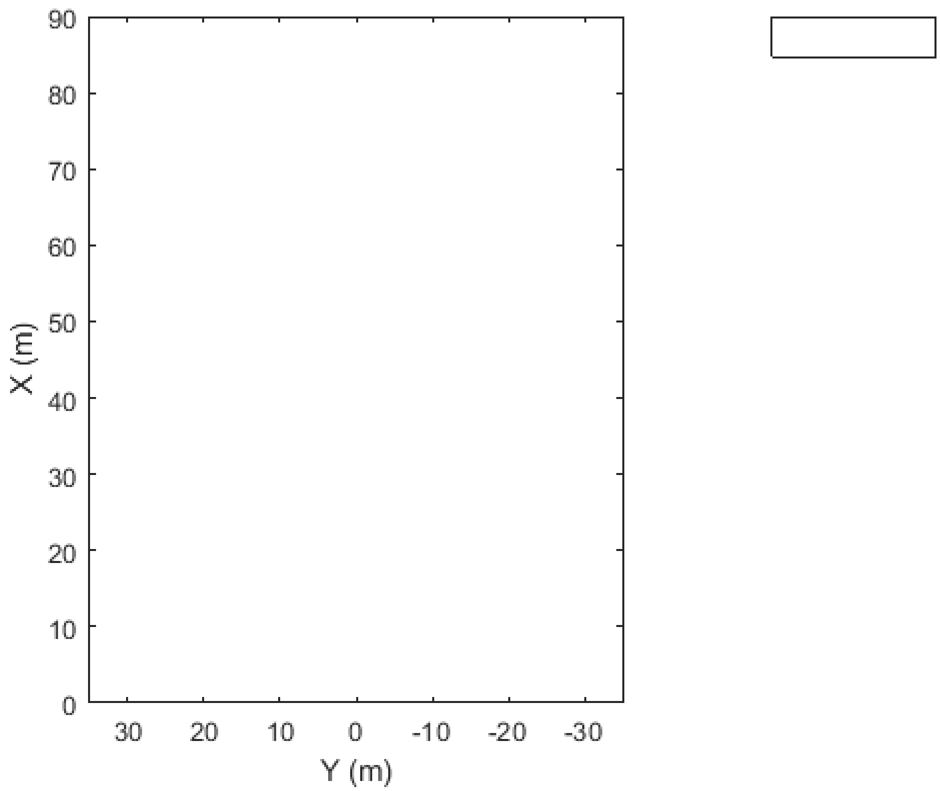
`plotCoverageArea(caPlotter,position,range,orientation,fieldOfView)` displays the coverage area of an ego vehicle sensor on a bird's-eye plot. Specify the position, range, orientation angle, and field of view of the sensor. The coverage area plotter, `caPlotter`, is associated with a `birdsEyePlot` object and configures the display of sensor coverage areas.

Examples

Display Coverage Area for Radar Sensor

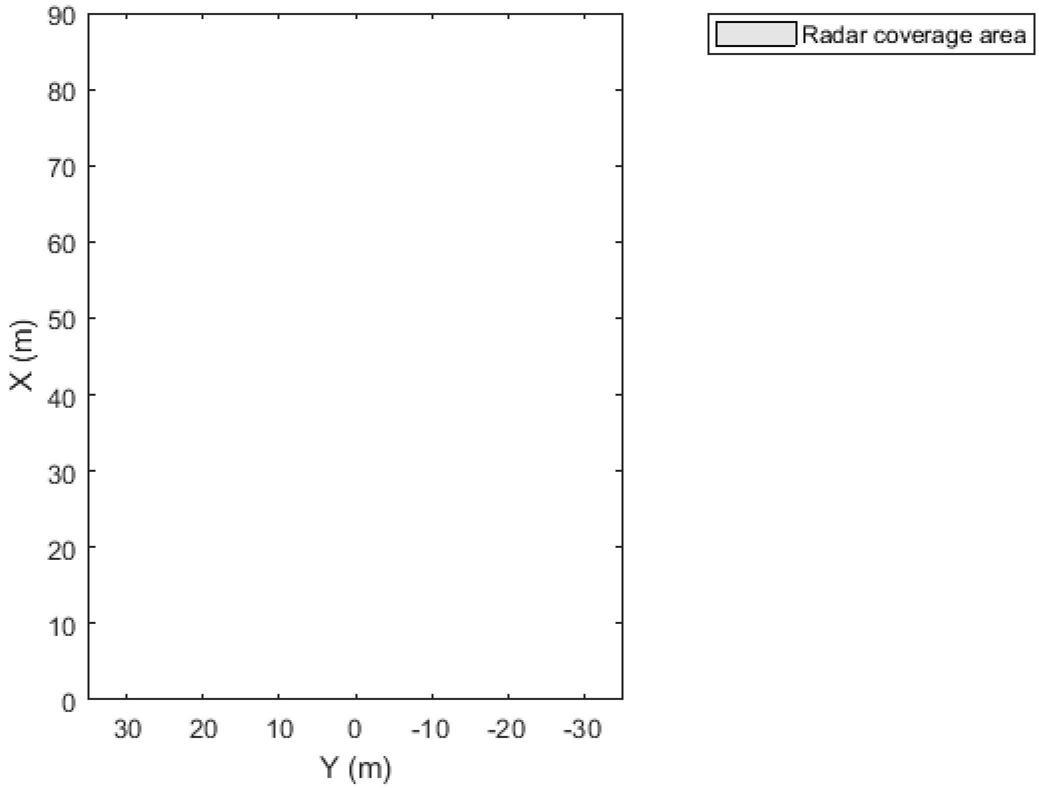
Create a bird's-eye plot with an *x*-axis range from 0 to 90 meters and a *y*-axis range from –35 to 35 meters.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
```



Create a coverage are plotter for the bird's-eye plot.

```
caPlotter = coverageAreaPlotter(bep,'DisplayName','Radar coverage area');
```

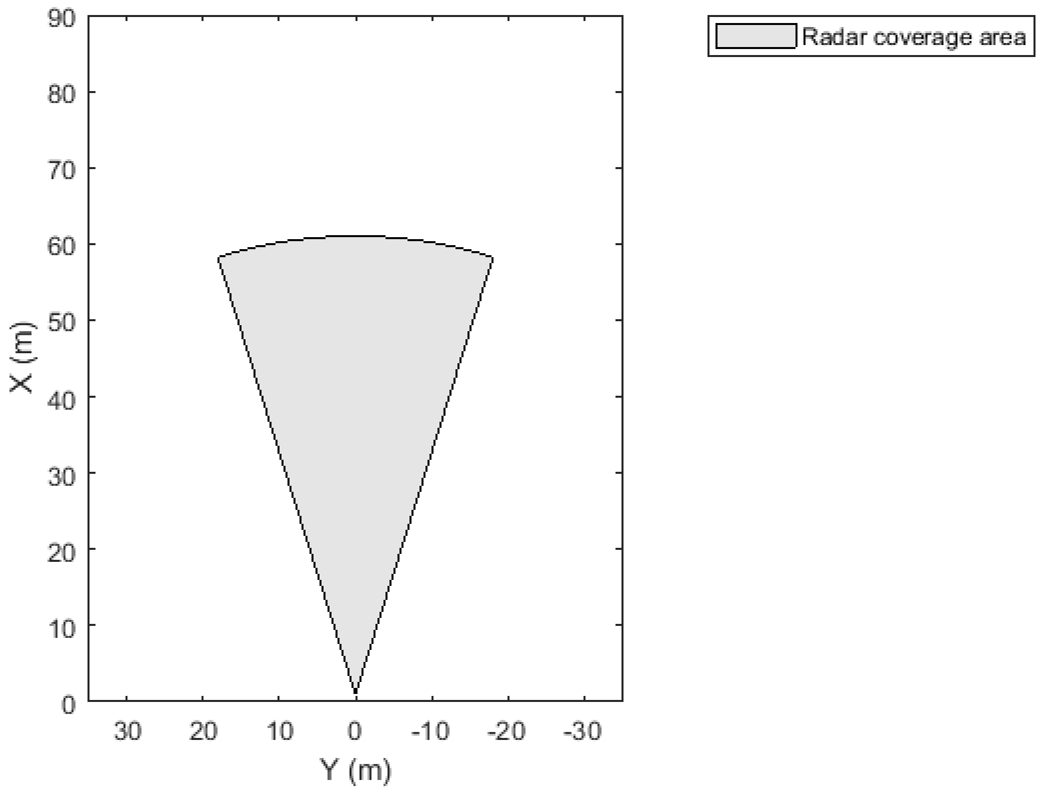


Display a coverage area that has a 35-degree field of view and a 60-meter range. Mount the coverage area sensor 1 meter in front of the origin. Set the orientation angle of the sensor to 0 degrees.

```
mountPosition = [1 0];
range = 60;
orientation = 0;
fieldOfView = 35;
```

Plot the coverage area.

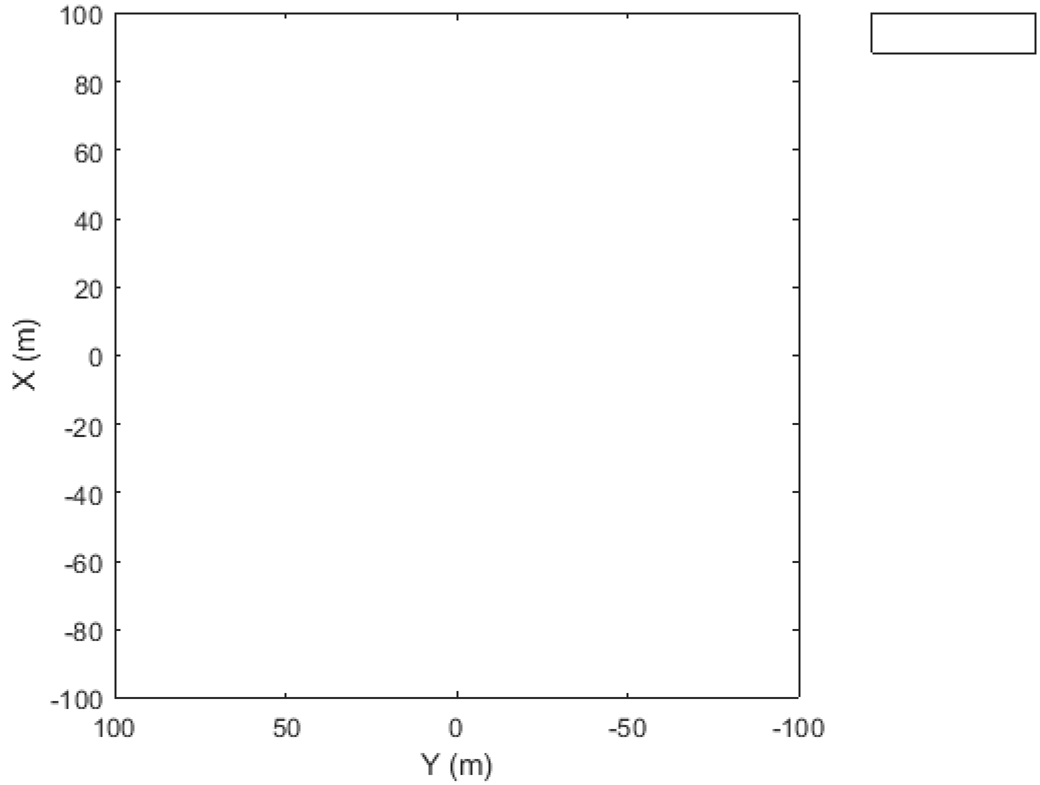
```
plotCoverageArea(caPlotter,mountPosition,range,orientation,fieldOfView);
```



Display Sensor Coverage Areas from Four Corners of Vehicle

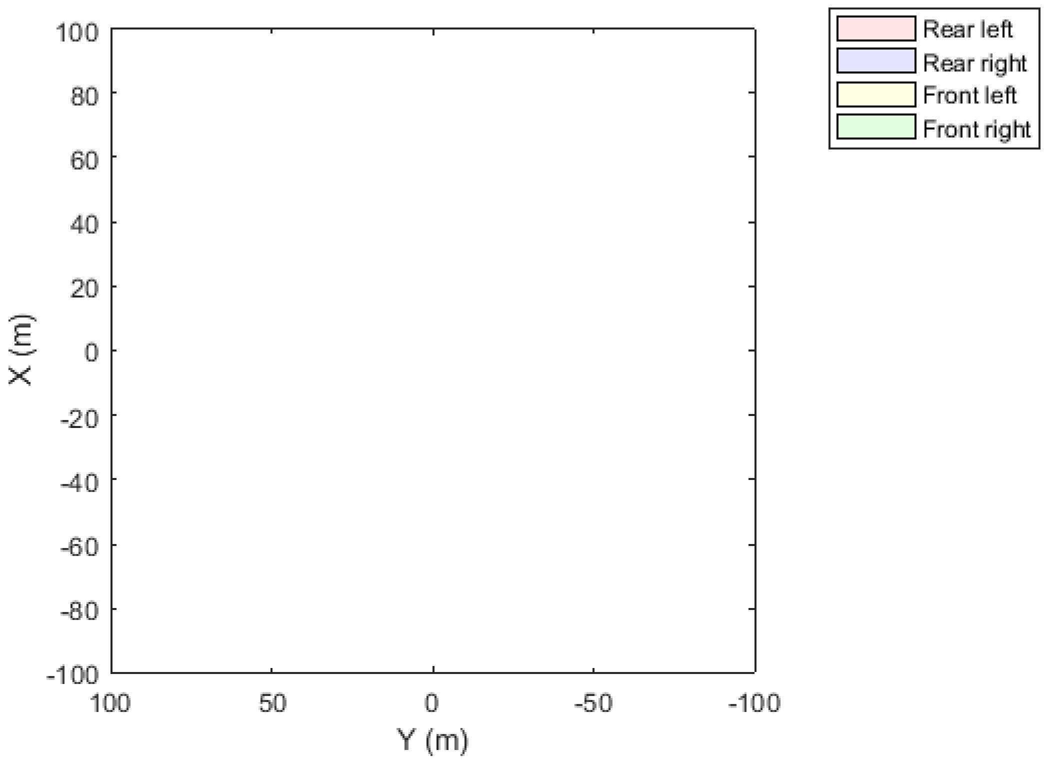
Create a bird's-eye plot with an *x*-axis range from –100 to 100 meters and a *y*-axis range from –100 to 100 meters

```
bep = birdsEyePlot('XLim',[-100 100],'YLim',[-100 100]);
```



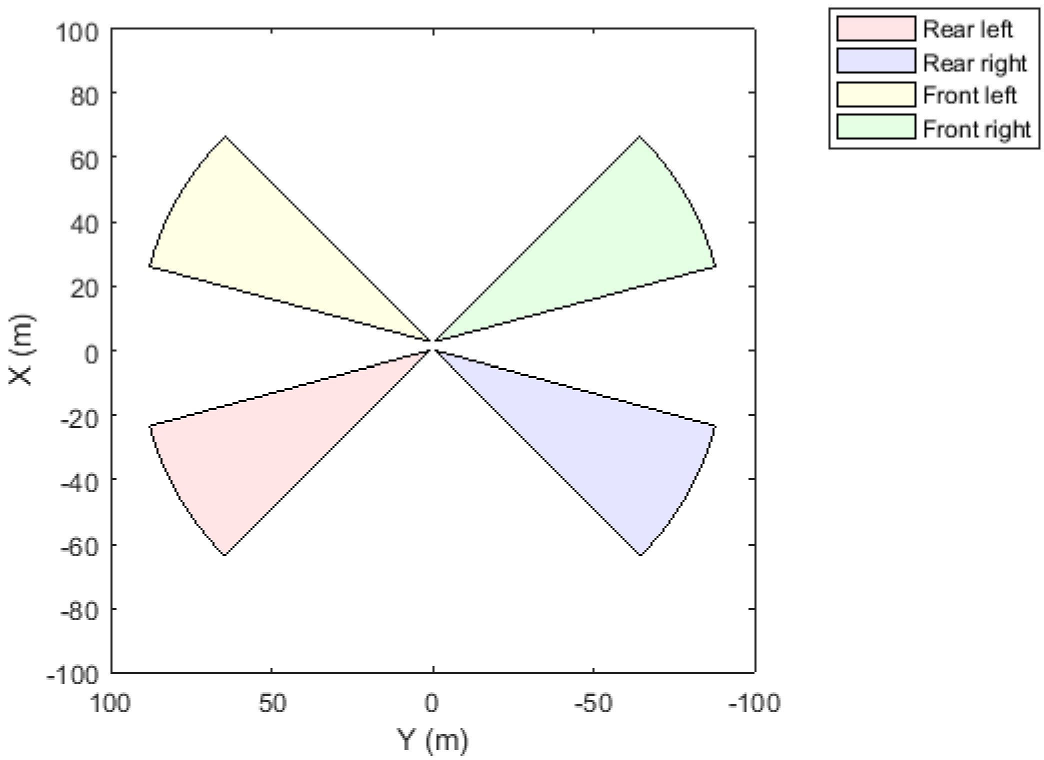
Create coverage area plotters with unique display names and fill colors for each sensor location on the vehicle.

```
rearLeftPlotter = coverageAreaPlotter(bep,'DisplayName','Rear left','FaceColor','r');
rearRightPlotter = coverageAreaPlotter(bep,'DisplayName','Rear right','FaceColor','b');
frontLeftPlotter = coverageAreaPlotter(bep,'DisplayName','Front left','FaceColor','y');
frontRightPlotter = coverageAreaPlotter(bep,'DisplayName','Front right','FaceColor','g');
```



Set the positions, ranges, orientations, and fields of view for the sensors. The sensors have a maximum range of 90 meters and a field of view of 30 degrees. Plot the coverage areas.

```
plotCoverageArea(rearLeftPlotter,[0 0.9],90,120,30);
plotCoverageArea(rearRightPlotter,[0 -0.9],90,-120,30);
plotCoverageArea(frontLeftPlotter,[2.8 0.9],90,60,30);
plotCoverageArea(frontRightPlotter,[2.8 -0.9],90,-60,30);
```



Input Arguments

caPlotter — Coverage area plotter
`CoverageAreaPlotter` object

Coverage area plotter, specified as a `CoverageAreaPlotter` object. This object is stored in the `Plotters` property of a `birdsEyePlot` object and configures the display of coverage areas in the bird's-eye plot. To create this object, use the `coverageAreaPlotter` function.

position — Position of sensor
real-valued vector of the form [$X_{OriginOffset}$ $Y_{OriginOffset}$]

Position of the sensor in vehicle coordinates, specified as a real-valued vector of the form [$X_{OriginOffset}$ $Y_{OriginOffset}$]. Units are in meters.

- $X_{OriginOffset}$ specifies the distance that the sensor is in front of the origin.
- $Y_{OriginOffset}$ specifies the distance that the sensor is to the left of the origin.

The origin is located at the center of the rear axle, as shown in this figure of the vehicle coordinate system.

range — Range of sensor
positive real scalar

Range of sensor, specified as a positive real scalar. Units are in meters.

orientation — Orientation angle of sensor
real scalar

Orientation angle of the sensor relative to the *X*-axis of the ego vehicle, specified as a real scalar. Units are in degrees. `orientation` is positive in the counterclockwise direction (to the left).

fieldOfView — Field of view of sensor
positive real scalar

Field of view of the sensor coverage area, specified as a positive real scalar. Units are in degrees.

See Also
`birdsEyePlot` | `coverageAreaPlotter`

Introduced in R2017a

plotDetection

Display object detections on bird's-eye plot

Syntax

```
plotDetection(detPlotter,positions)
plotDetection(detPlotter,positions,velocities)
plotDetection(detPlotter,positions,labels)
plotDetection(detPlotter,positions,velocities,labels)
```

Description

`plotDetection(detPlotter,positions)` displays object detections from a list of object positions on a bird's-eye plot. The detection plotter, `detPlotter`, is associated with a `birdsEyePlot` object and configures the display of the specified detections.

To remove all detections associated with detection plotter `detPlotter`, call the `clearData` function and specify `detPlotter` as the input argument.

`plotDetection(detPlotter,positions,velocities)` displays detections and their velocities on a bird's-eye plot.

`plotDetection(detPlotter,positions,labels)` displays detections and their labels on a bird's-eye plot.

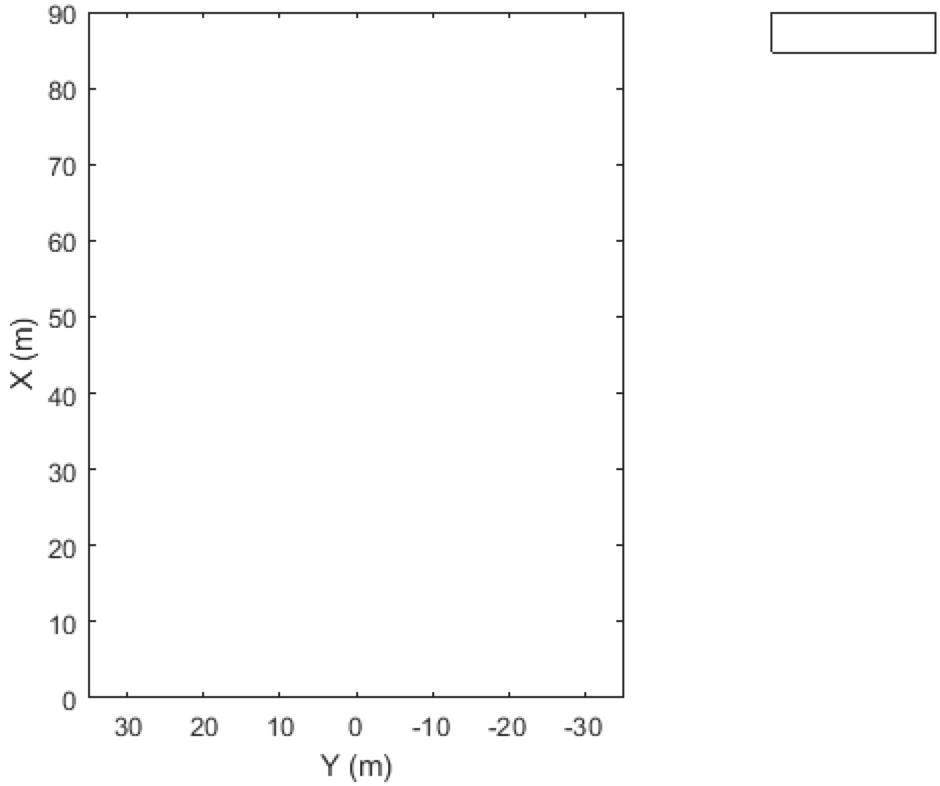
`plotDetection(detPlotter,positions,velocities,labels)` displays detections and their velocities and labels on a bird's-eye plot. `velocities` and `labels` can appear in either order but must come after `detPlotter` and `positions`.

Examples

Create and Display a Bird's-Eye Plot

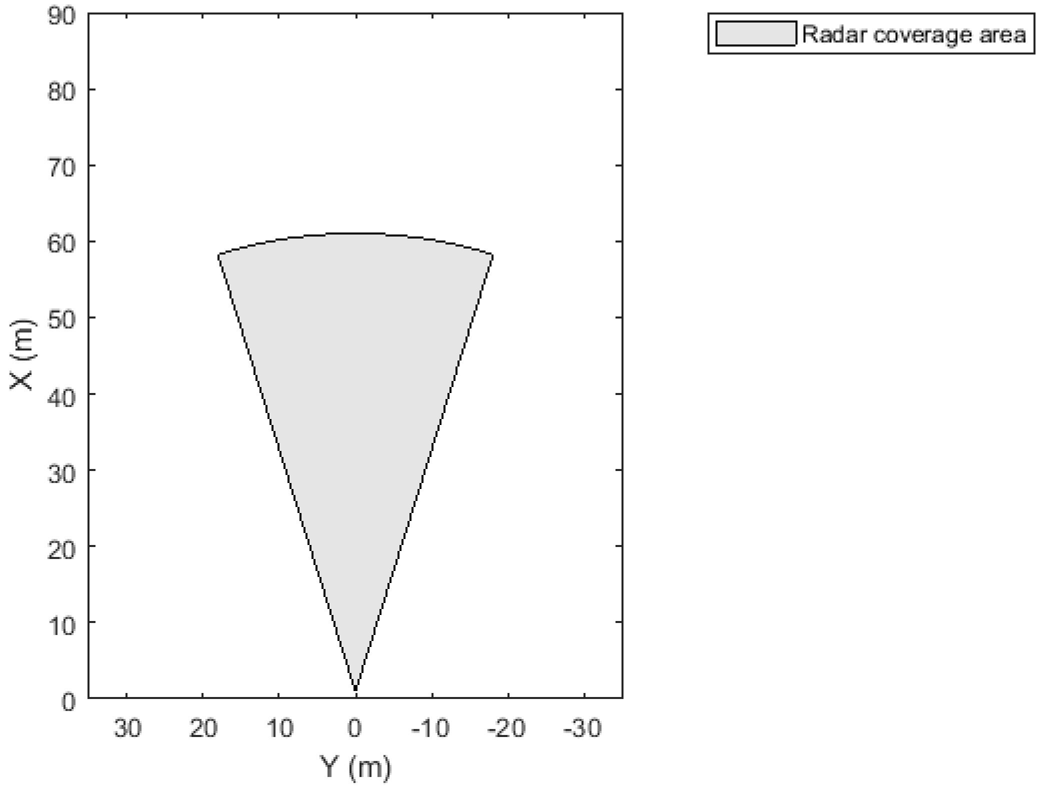
Create a bird's-eye plot with an *x*-axis range from 0 to 90 meters and a *y*-axis range from –35 to 35 meters.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
```



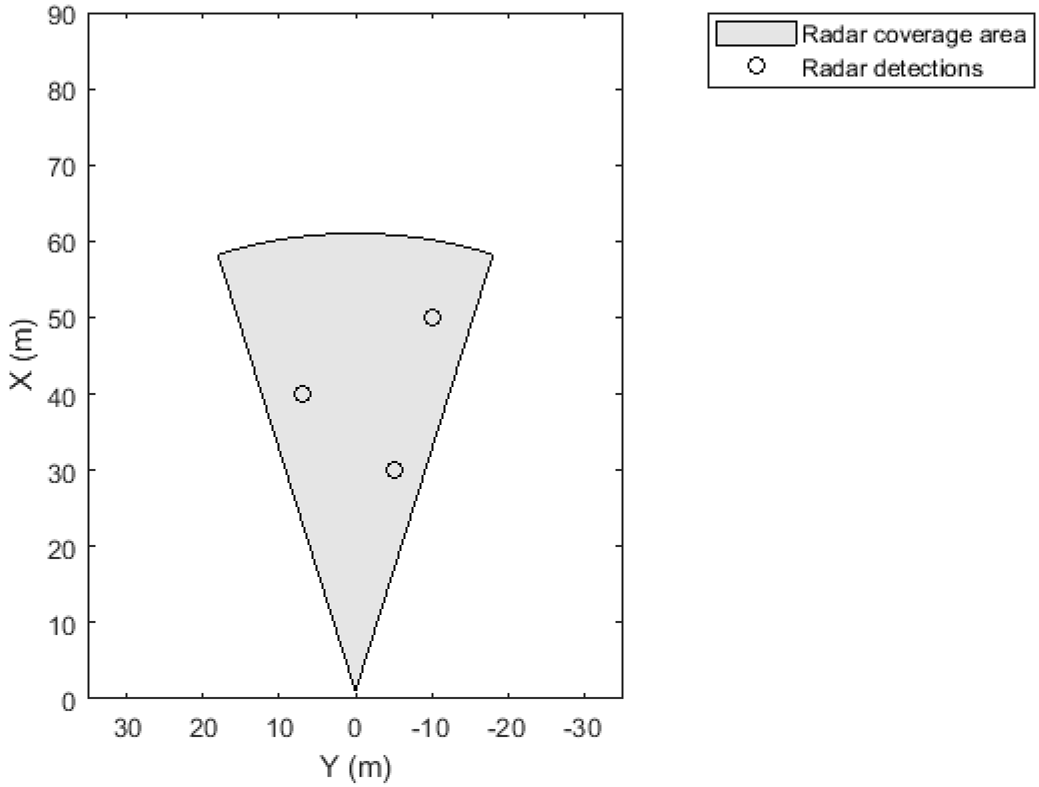
Display a coverage area with a 35-degree field of view and a 60-meter range.

```
caPlotter = coverageAreaPlotter(bep,'DisplayName','Radar coverage area');
mountPosition = [1 0];
range = 60;
orientation = 0;
fieldOfView = 35;
plotCoverageArea(caPlotter,mountPosition,range,orientation,fieldOfView);
```



Display radar detections with coordinates at (30, –5), (50, –10), and (40, 7).

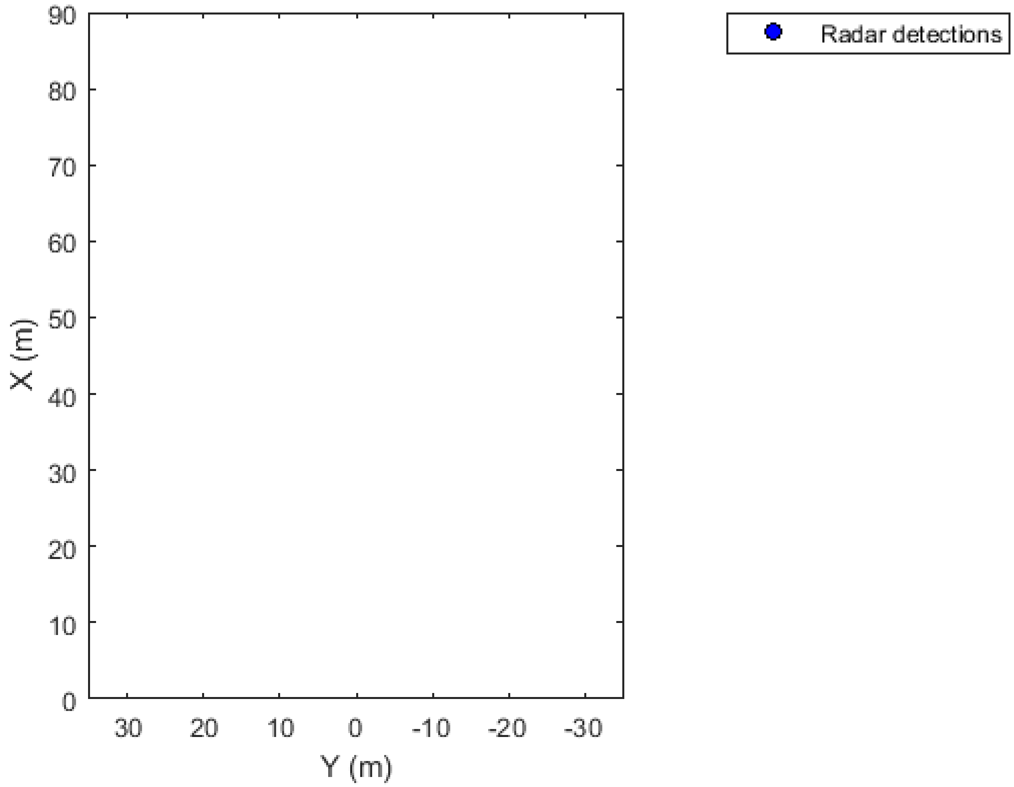
```
radarPlotter = detectionPlotter(bep,'DisplayName','Radar detections');
plotDetection(radarPlotter,[30 -5; 50 -10; 40 7]);
```



Create and Display Labeled Detections on Bird's-Eye Plot

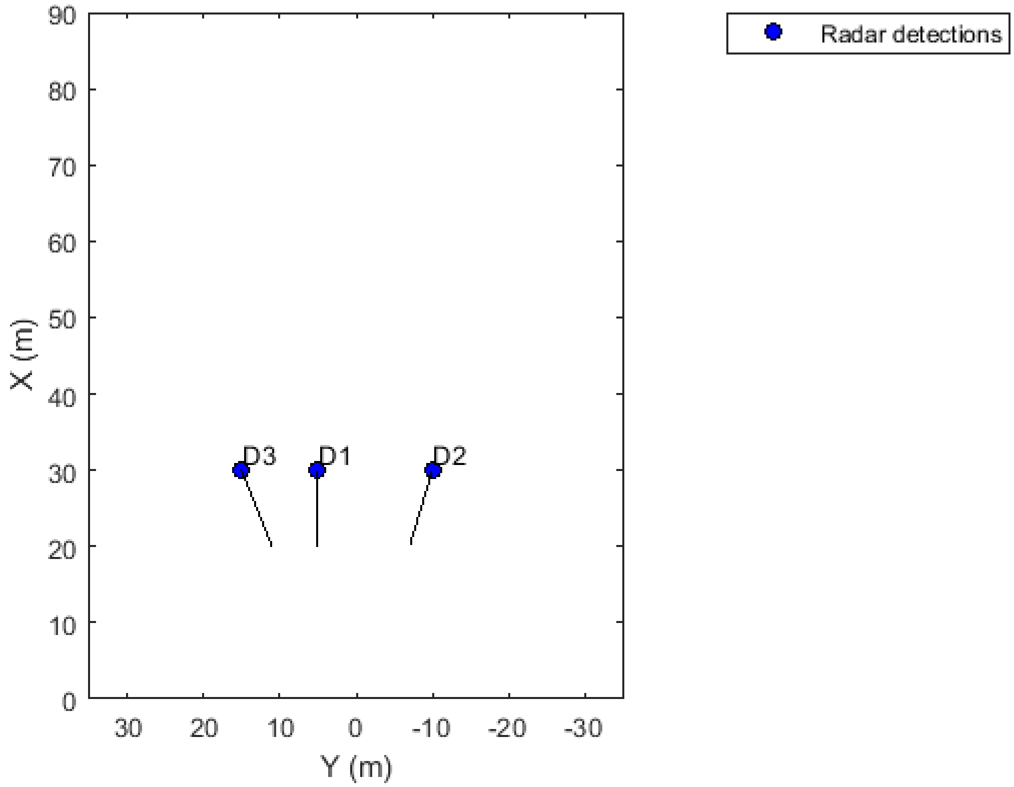
Create a bird's-eye plot with an *x*-axis range from 0 to 90 meters and a *y*-axis range from –35 to 35 meters. Create a radar detection plotter that displays detections in blue.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
detPlotter = detectionPlotter(bep,'DisplayName','Radar detections', ...
    'MarkerFaceColor','b');
```



Display the positions and velocities of three labeled detections.

```
positions = [30 5; 30 -10; 30 15];
velocities = [-10 0; -10 3; -10 -4];
labels = {'D1','D2','D3'};
plotDetection(detPlotter,positions,velocities,labels);
```



Input Arguments

`detPlotter` — Detection plotter
`DetectionPlotter` object

Detection plotter, specified as a `DetectionPlotter` object. This object is stored in the `Plotters` property of a `birdsEyePlot` object and configures the display of the specified detections in the bird's-eye plot. To create this object, use the `detectionPlotter` function.

`positions` — Positions of detected objects
M-by-2 real-valued matrix

Positions of detected objects in vehicle coordinates, specified as an *M*-by-2 real-valued matrix of (*X*, *Y*) positions. *M* is the number of detected objects. The positive *X*-direction points ahead of the center of the vehicle. The positive *Y*-direction points to the left of the origin of the vehicle, which is the center of the rear axle, as shown in this figure of the vehicle coordinate system.

`velocities` — Velocities of detected objects
M-by-2 real-valued matrix

Velocities of detected objects, specified as an *M*-by-2 real-valued matrix of velocities in the (*X*, *Y*) direction. *M* is the number of detected objects. The velocities are plotted as line vectors that originate from the center positions of the detections as they are tracked.

`labels` — Detection labels
M-length string array | *M*-length cell array of character vectors

Detection labels, specified as an *M*-length string array or *M*-length cell array of character vectors. *M* is the number of detected objects. The labels correspond to the locations in the `positions` matrix. By default, detections do not have labels. To remove all annotations and labels associated with the detection plotter, use the `clearData` function.

See Also
`birdsEyePlot` | `detectionPlotter`

Introduced in R2017a

plotLaneBoundary

Display lane boundaries on bird’s-eye plot

Syntax

```
plotLaneBoundary(lbPlotter,boundaryCoords)
plotLaneBoundary(lbPlotter,boundaries)
```

Description

`plotLaneBoundary(lbPlotter,boundaryCoords)` displays lane boundaries from a list of boundary coordinates on a bird’s-eye plot. The lane boundary plotter, `lbPlotter`, is associated with a `birdsEyePlot` object and configures the display of the specified lane boundaries.

To remove all lane boundaries associated with lane boundary plotter `lbPlotter`, call the `clearData` function and specify `lbPlotter` as the input argument.

`plotLaneBoundary(lbPlotter,boundaries)` displays lane boundaries from a lane boundary object or an array of lane boundary objects, `boundaries`.

Examples

Create and Display Road Boundaries

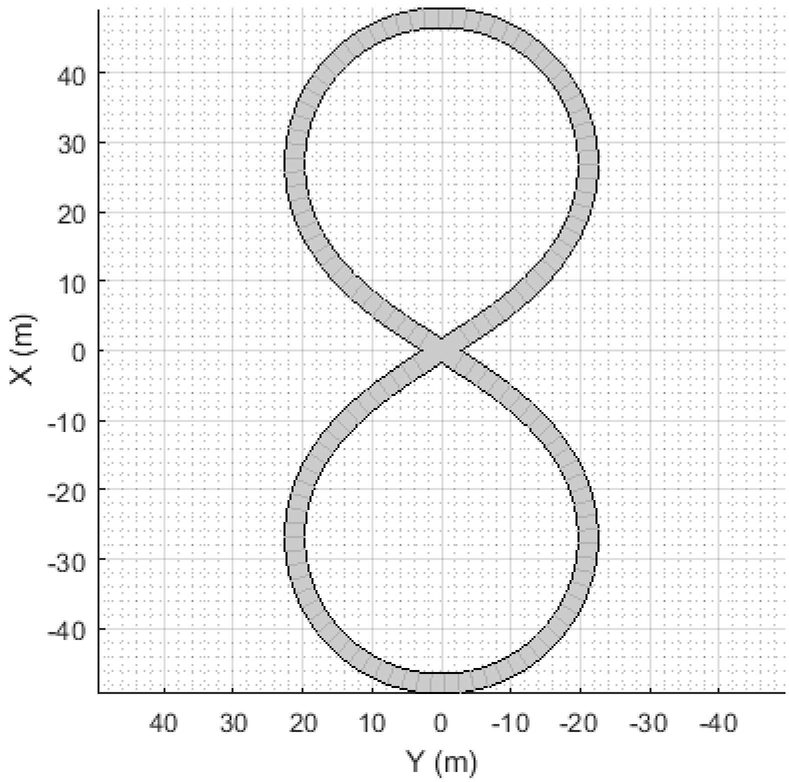
Create a driving scenario containing a figure-8 road specified in the world coordinates of the scenario. Convert the world coordinates of the scenario to the coordinate system of the ego vehicle.

Create an empty driving scenario.

```
scenario = drivingScenario;
```

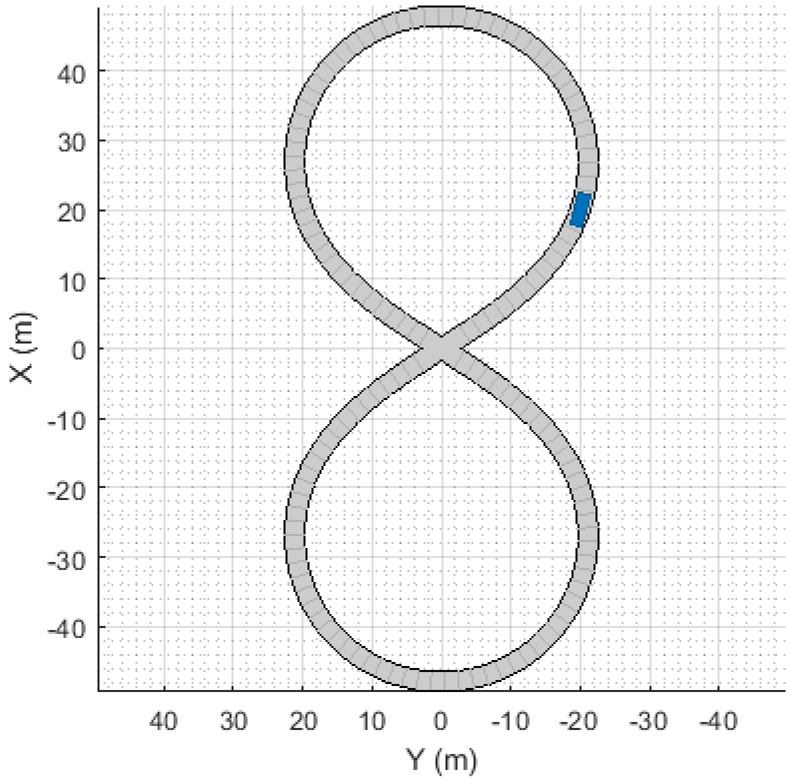
Add a figure-8 road to the scenario. Display the scenario.

```
roadCenters = [0  0  1
             20 -20  1
             20  20  1
            -20 -20  1
            -20  20  1
              0   0  1];

roadWidth = 3;
bankAngle = [0 15 15 -15 -15 0];
road(scenario,roadCenters,roadWidth,bankAngle);
plot(scenario)
```



Add an ego vehicle to the scenario. Position the vehicle at world coordinates (20, –20) and orient it at a –15 degree yaw angle.

```
ego = actor(scenario,'ClassID',1,'Position',[20 -20 0],'Yaw',-15);
```

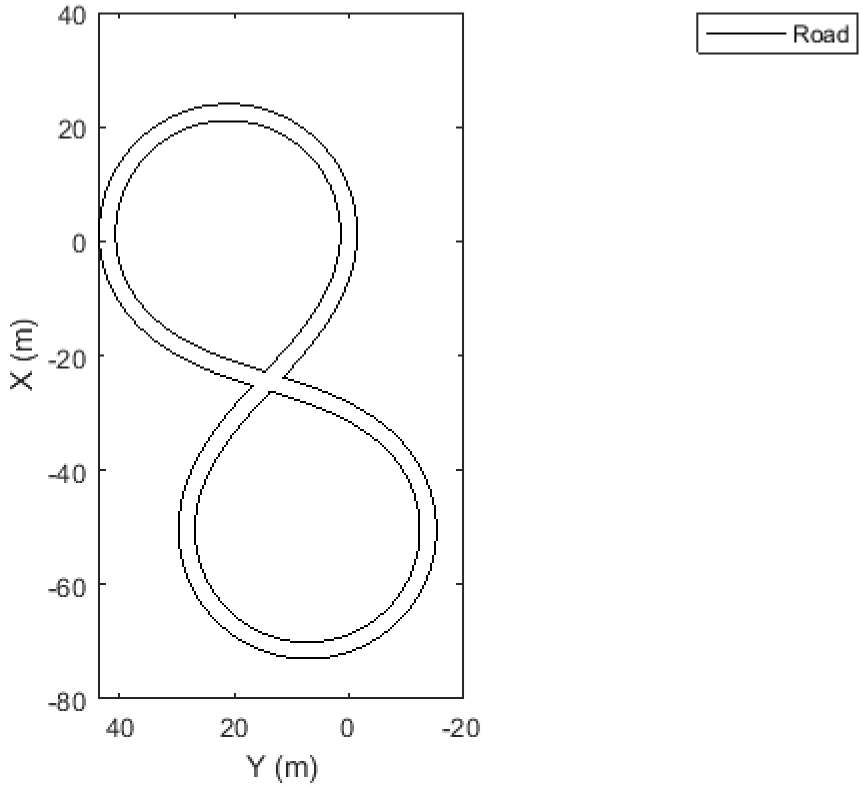


Obtain the road boundaries in ego vehicle coordinates by using the `roadBoundaries` function. Specify the ego vehicle as the input argument.

```
rbEgo1 = roadBoundaries(ego);
```

Display the result on a bird's-eye plot.

```
bep = birdsEyePlot;
lbp = laneBoundaryPlotter(bep,'DisplayName','Road');
plotLaneBoundary(lbp,rbEgo1)
```



Obtain the road boundaries in world coordinates by using the `roadBoundaries` function. Specify the scenario as the input argument.

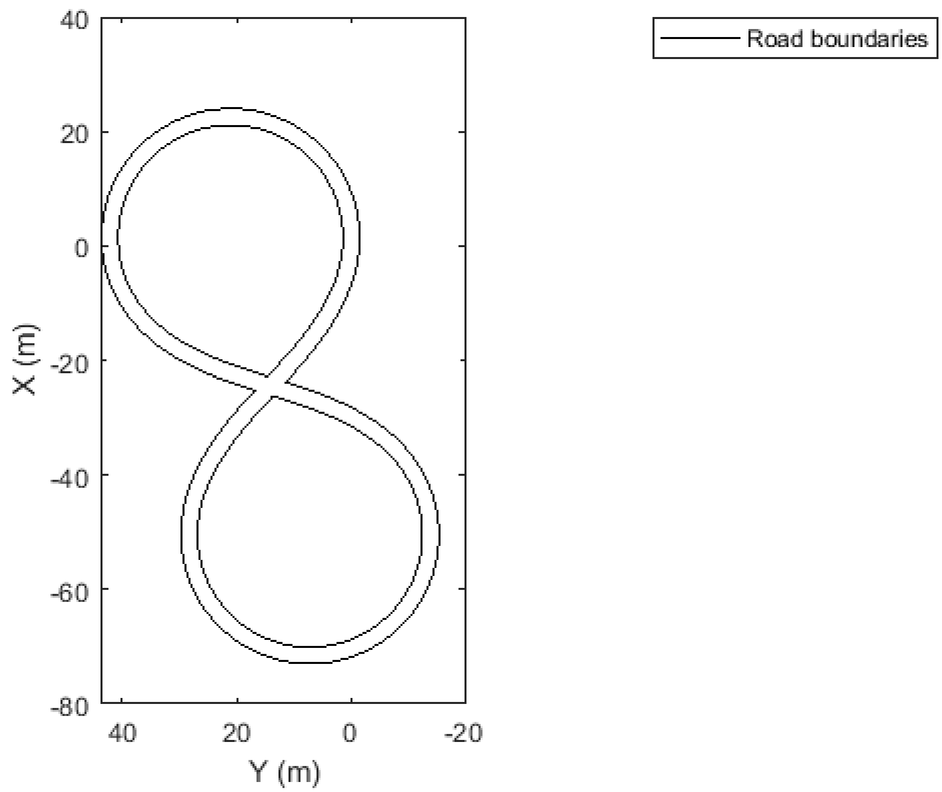
```
rbScenario = roadBoundaries(scenario);
```

Obtain the road boundaries in ego vehicle coordinates by using the `driving.scenario.roadBoundariesToEgo` function.

```
rbEgo2 = driving.scenario.roadBoundariesToEgo(rbScenario,ego);
```

Display the road boundaries on a bird's-eye plot.

```
bep = birdsEyePlot;
lbp = laneBoundaryPlotter(bep,'DisplayName','Road boundaries');
plotLaneBoundary(lbp,{rbEgo2})
```



Input Arguments

`lbPlotter` — Lane boundary plotter
`LaneBoundaryPlotter` object

Lane boundary plotter, specified as a `LaneBoundaryPlotter` object. This object is stored in the `Plotters` property of a `birdsEyePlot` object and configures the display of the specified lane boundaries in the bird's-eye plot. To create this object, use the `laneBoundaryPlotter` function.

`boundaryCoords` — Lane boundary coordinates
cell array of *M*-by-2 real-valued matrices

Lane boundary coordinates, specified as a cell array of *M*-by-2 real-valued matrices. Each matrix represents the coordinates for a different lane boundary. *M* is the number of coordinates in a lane boundary and can be different for each lane boundary. Each row represents the (*X*, *Y*) positions of a curve. The positive *X*-direction points ahead of the center of the vehicle. The positive *Y*-direction points to the left of the origin of the vehicle, which is the center of the rear axle, as shown in this figure of the vehicle coordinate system.

boundaries — Lane boundaries
lane boundary object | array of lane boundary objects

Lane boundaries, specified as a lane boundary object or an array of lane boundary objects. Valid lane boundary objects are `parabolicLaneBoundary`, `cubicLaneBoundary`, and `clothoidLaneBoundary`. If you specify an array of lane boundary objects, all objects must be of the same type. *Z*-data, which represents height, is ignored.

See Also

`birdsEyePlot` | `laneBoundaryPlotter`

Introduced in R2017a

plotLaneMarking

Display lane markings on bird’s-eye plot

Syntax

```
plotLaneMarking(lmPlotter,lmv,lmf)
```

Description

`plotLaneMarking(lmPlotter,lmv,lmf)` displays lane marking vertices, `lmv`, and lane marking faces, `lmf`, on a bird's-eye plot. The lane marking plotter, `lmPlotter`, is associated with a `birdsEyePlot` object and configures the display of the specified lane markings.

To remove all lane markings associated with the lane marking plotter `lmPlotter`, call the `clearData` function and specify `lmPlotter` as the input argument.

Examples

Display Lane Markings in Car and Pedestrian Scenario

Create a driving scenario containing a car and pedestrian on a straight road. Then, create and display the lane markings of the road on a bird's-eye plot.

Create an empty driving scenario.

```
scenario = drivingScenario;
```

Create a straight, 25-meter road segment with two travel lanes in one direction.

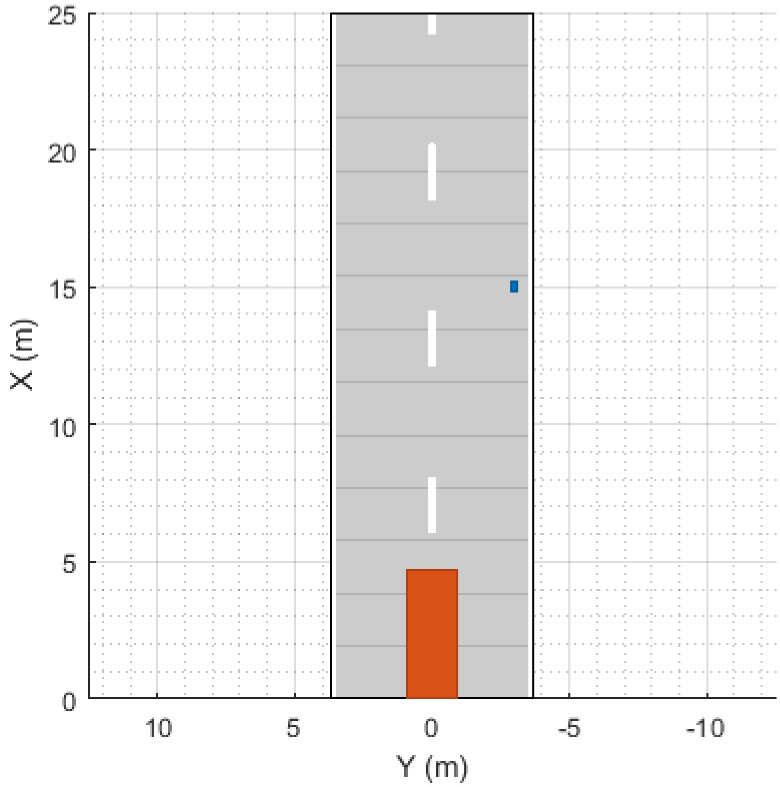
```
lm = [laneMarking('Solid')
      laneMarking('Dashed','Length',2,'Space',4)
      laneMarking('Solid')];
l = lanespec(2,'Marking',lm);
road(scenario,[0 0 0; 25 0 0],'Lanes',l);
```

Add to the driving scenario a pedestrian crossing the road at 1 meter per second and a car following the road at 10 meters per second.

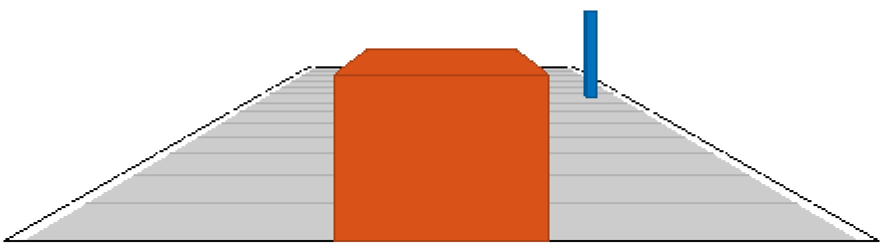
```
ped = actor(scenario,'ClassID',4,'Length',0.2,'Width',0.4,'Height',1.7);
car = vehicle(scenario,'ClassID',1);
trajectory(ped,[15 -3 0; 15 3 0],1);
trajectory(car,[car.RearOverhang 0 0; 25-car.Length+car.RearOverhang 0 0],10);
```

Display the scenario and corresponding chase plot.

```
plot(scenario)
```



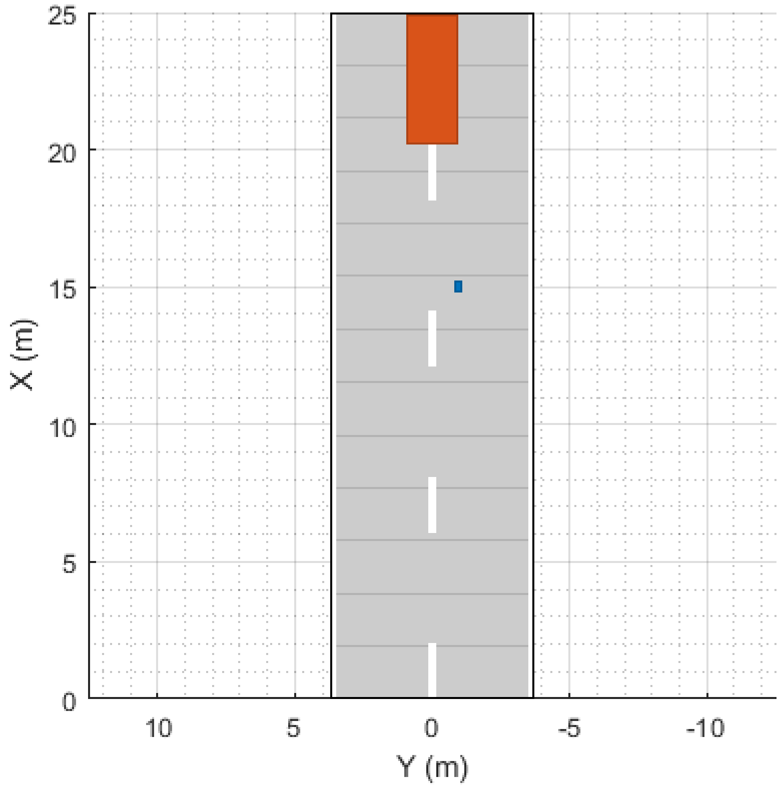

```
chasePlot(car)
```

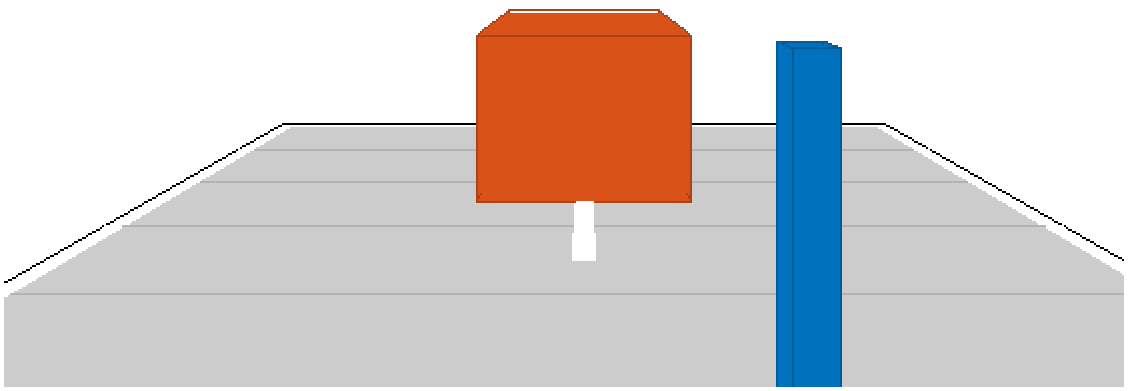


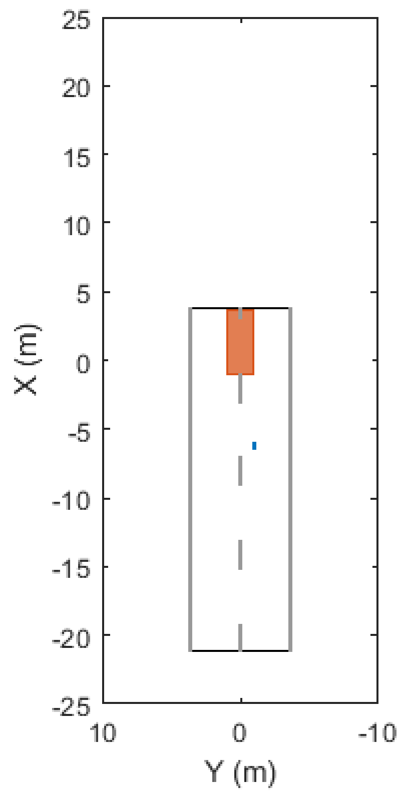
Run the simulation.

1. Create a bird's-eye plot.
2. Create an outline plotter, lane boundary plotter, and lane marking plotter for the bird's-eye plot.
3. Obtain the road boundaries and target outlines.
4. Obtain the lane marking vertices and faces.
5. Display the lane boundaries and lane markers.
6. Run the simulation loop.

```
bep = birdsEyePlot('XLim',[-25 25],'YLim',[-10 10]);
olPlotter = outlinePlotter(bep);
lbPlotter = laneBoundaryPlotter(bep);
lmPlotter = laneMarkingPlotter(bep,'DisplayName','Lanes');
legend('off');
while advance(scenario)
    rb = roadBoundaries(car);
    [position,yaw,length,width,originOffset,color] = targetOutlines(car);
    [lmv,lmf] = laneMarkingVertices(car);
    plotLaneBoundary(lbPlotter,rb);
    plotLaneMarking(lmPlotter,lmv,lmf);
    plotOutline(olPlotter,position,yaw,length,width, ...
        'OriginOffset',originOffset,'Color',color);
end
```


25
20
15
10
5
0
X (m)
10
5
0
-5
-10
Y (m)





Input Arguments

`lmPlotter` — Lane marking plotter
`LaneMarkingPlotter` object

Lane marking plotter, specified as a `LaneMarkingPlotter` object. This object is stored in the `Plotters` property of a `birdsEyePlot` object and configures the display of the specified lane markings in the bird's-eye plot. To create this object, use the `laneMarkingPlotter` function.

`lmv` — Lane marking vertices
L-by-3 real-valued matrix

Lane marking vertices, specified as an *L*-by-3 real-valued matrix. Each row of `lmv` represents the *x*, *y*, and *z* coordinates of one vertex. The plotter uses only the *x* and *y* coordinates. To obtain lane marking vertices and faces from a driving scenario, use the `laneMarkingVertices` function.

`lmf` — Lane marking faces
real-valued matrix

Lane marking faces, specified as a real-valued matrix. Each row of `lmf` is a face that defines the connection between vertices for one lane marking. To obtain lane marking vertices and faces from a driving scenario, use the `laneMarkingVertices` function.

See Also

`birdsEyePlot` | `laneMarkingPlotter` | `laneMarkingVertices`

Introduced in R2018a

plotMesh

Package:

Display object meshes on bird's-eye plot

Syntax

```
plotMesh(mPlotter,vertices,faces)
plotMesh(mPlotter,vertices,faces,'Color',colors)
```

Description

`plotMesh(mPlotter,vertices,faces)` displays meshes on page 4-105 composed of the specified vertices and faces on a bird's-eye plot. To obtain the mesh vertices and faces of an object in a driving scenario, use the `targetMeshes` function. The mesh plotter, `mPlotter`, is associated with a `birdsEyePlot` object and configures the display of the meshes.

The bird's-eye plot assigns a different color to each actor, based on the default color order of `Axes` objects. For more details, see the `ColorOrder` property for `Axes` objects.

To remove all meshes associated with mesh plotter `mPlotter`, call the `clearData` function and specify `mPlotter` as the input argument.

`plotMesh(mPlotter,vertices,faces,'Color',colors)` specifies the colors of the meshes.

Examples

Display Actor Meshes in Driving Scenario

Display actors in a driving scenario by using their mesh representations instead of their cuboid representations.

Create a driving scenario, and add a 25-meter straight road to the scenario.

```
scenario = drivingScenario;
roadcenters = [0 0 0; 25 0 0];
road(scenario,roadcenters);
```

Add a pedestrian and a vehicle to the scenario. Specify the mesh dimensions of the actors using prebuilt meshes.

- Specify the pedestrian mesh as a `driving.scenario.pedestrianMesh` object.
- Specify the vehicle mesh as a `driving.scenario.carMesh` object.

```
p = actor(scenario,'ClassID',4, ...
            'Length',0.2,'Width',0.4, ...
            'Height',1.7,'Mesh',driving.scenario.pedestrianMesh);

v = vehicle(scenario,'ClassID',1, ...
            'Mesh',driving.scenario.carMesh);
```

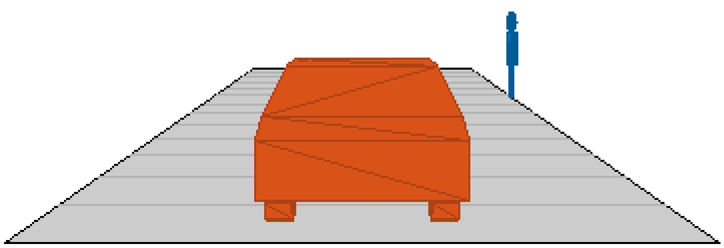
Add trajectories for the pedestrian and vehicle.

- Specify for the pedestrian to cross the road at 1 meter per second.
- Specify for the vehicle to follow the road at 10 meters per second.

```
waypointsP = [15 -3 0; 15 3 0];
speedP = 1;
trajectory(p,waypointsP,speedP);

wayPointsV = [v.RearOverhang 0 0; (25 - v.Length + v.RearOverhang) 0 0];
speedV = 10;
trajectory(v,wayPointsV,speedV)
```

Add an egocentric plot for the vehicle. Turn the display of meshes on.

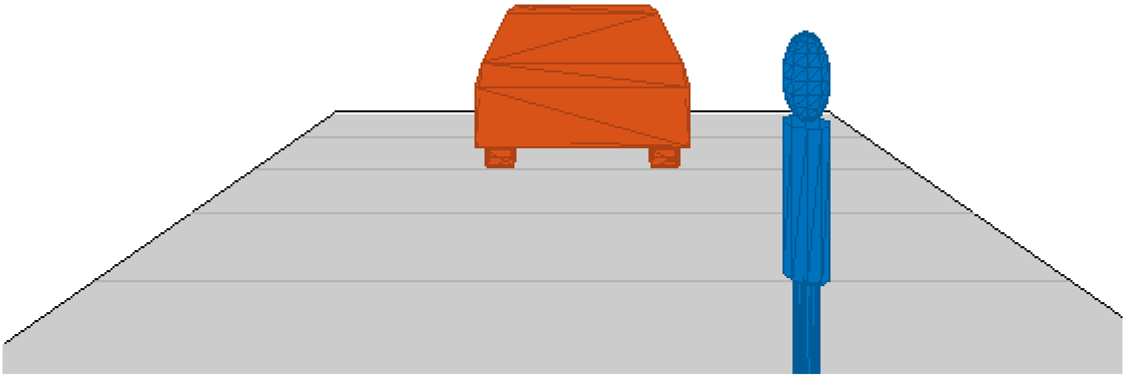
```
chasePlot(v,'Meshes','on')
```

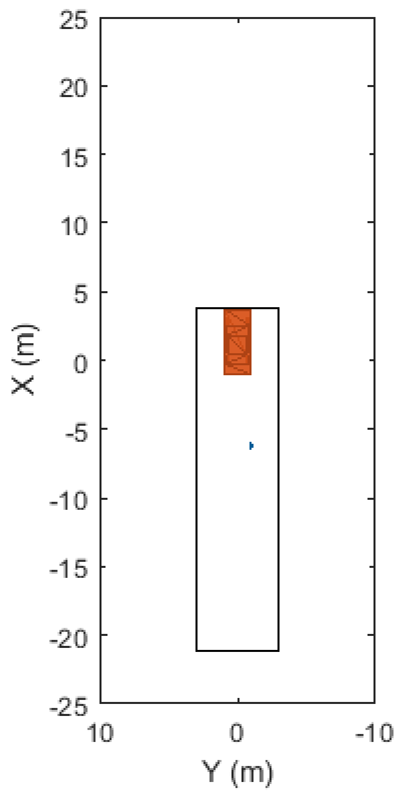


Create a bird's-eye plot in which to display the meshes. Also create a mesh plotter and lane boundary plotter. Then run the simulation loop.

1. Obtain the road boundaries of the road the vehicle is on.
2. Obtain the mesh vertices, faces, and colors of the actor meshes, with positions relative to the vehicle.
3. Plot the road boundaries and actor meshes on the bird's-eye plot.
4. Pause the scenario to allow time for the plots to update. The chase plot updates every time you advance the scenario.

```
bep = birdsEyePlot('XLim',[-25 25],'YLim',[-10 10]);
mPlotter = meshPlotter(bep);
lbPlotter = laneBoundaryPlotter(bep);
legend('off')

while advance(scenario)

   rb = roadBoundaries(v);

   [vertices,faces,colors] = targetMeshes(v);

   plotLaneBoundary(lbPlotter,rb)
   plotMesh(mPlotter,vertices,faces,'Color',colors)

   pause(0.01)
end
```





Input Arguments

mPlotter — Mesh plotter
MeshPlotter object

Mesh plotter, specified as a `MeshPlotter` object. This object is stored in the `Plotters` property of a `birdsEyePlot` object and configures the display of meshes in the bird's-eye plot. To create this object, use the `meshPlotter` function.

vertices — Mesh vertices of each actor
N-element cell array

Mesh vertices of each actor, specified as an *N*-element cell array, where *N* is the number of actors.

Each element in `vertices` must be a *V*-by-3 real-valued matrix containing the vertices of an actor, where:

- *V* is the number of vertices.
- Each row defines the 3-D (*x*,*y*,*z*) position of a vertex. When you use the `targetMeshes` function to obtain mesh vertices, the vertex positions are relative to the position of the actor that is input to that function. Units are in meters.

faces — Mesh faces of each actor
N-element cell array

Mesh faces of each actor, specified as an *N*-element cell array, where *N* is the number of actors.

Each element in `faces` must be an *F*-by-3 integer-valued matrix containing the faces of an actor, where:

- *F* is the number of faces.
- Each row defines a triangle of vertex IDs that make up the face. The vertex IDs correspond to row numbers within `vertices`.

Suppose the first face of the `i`th element of `faces` has these vertex IDs.

```
faces{i}(1,:)
```

```
ans =

     1     2     3
```

In the `i`th element of `vertices`, rows 1, 2, and 3 contain the (*x*, *y*, *z*) positions of the vertices that make up this face.

```
vertices{i}(1:3,:)
```

```
ans =

    3.7000    0.9000    0.8574
    3.7000   -0.9000    0.8574
    3.7000   -0.9000    0.3149
```

`colors` — Color of mesh faces for each actor
N-by-3 matrix of RGB triplets

Color of the mesh faces for each actor, specified as an *N*-by-3 matrix of RGB triplets. *N* is the number of actors and is equal to the number of elements in `vertices` and `faces`.

The `i`th row of `colors` is the RGB color value of the faces in the `i`th element of `faces`. The function applies the same color to all mesh faces of an actor.

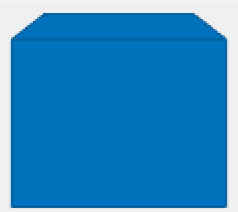
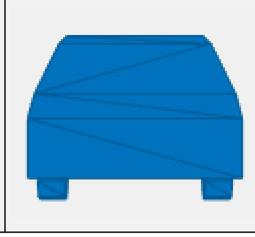
An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0, 1]. For example, `[0.4 0.6 0.7]`.

More About

Meshes

In driving scenarios, a mesh is a triangle-based 3-D representation of an object. Mesh representations of objects are more detailed than the default cuboid (box-shaped) representations of objects. Meshes are useful for generating synthetic point cloud data from a driving scenario.

This table shows the difference between a cuboid representation and a mesh representation of a vehicle in a driving scenario.

Cuboid	Mesh
	

See Also
birdsEyePlot | meshPlotter | targetMeshes

Introduced in R2020b

plotOutline

Display object outlines on bird's-eye plot

Syntax

```
plotOutline(olPlotter,positions,yaw,length,width)
plotOutline( ___ ,Name,Value)
```

Description

`plotOutline(olPlotter,positions,yaw,length,width)` displays the rectangular outlines of cuboid objects on a bird's-eye plot. Specify the position, yaw angle of rotation, length, and width of each cuboid. The outline plotter, `olPlotter`, is associated with a `birdsEyePlot` object and configures the display of the specified outlines.

To remove all outlines associated with outline plotter `olPlotter`, call the `clearData` function and specify `olPlotter` as the input argument.

To display the outlines of actors that are in a driving scenario, first use `targetOutlines` to get the dimensions of the actors. Then, after calling `outlinePlotter` to create a plotter object, use the `plotOutline` function to display the outlines of all the actors in a bird's-eye plot.

`plotOutline( ___ ,Name,Value)` specifies options using one or more `Name,Value` pair arguments and the input arguments from the previous syntax.

Examples

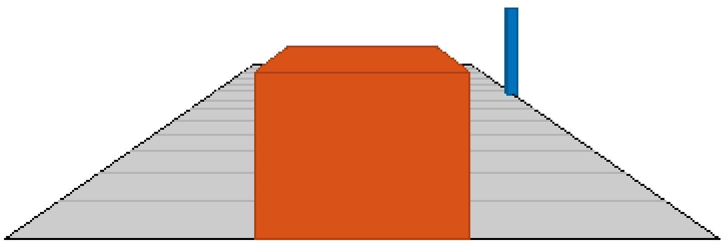
Plot Outlines of Targets on Bird's-Eye Plot

Create a driving scenario. Create a 25 m road segment, add a pedestrian and a vehicle, and specify their trajectories to follow. The pedestrian crosses the road at 1 m/s. The vehicle drives along the road at 10 m/s.

```
scenario = drivingScenario;

road(scenario,[0 0 0; 25 0 0]);

p = actor(scenario,'ClassID',4,'Length',0.2,'Width',0.4,'Height',1.7);
v = vehicle(scenario,'ClassID',1);

trajectory(p,[15 -3 0; 15 3 0],1);
trajectory(v,[v.RearOverhang 0 0; 25-v.Length+v.RearOverhang 0 0], 10);
```

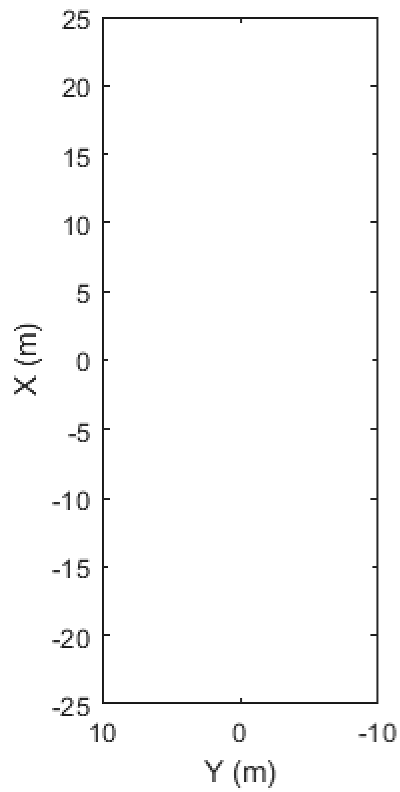
Use a chase plot to display the scenario from the perspective of the vehicle.

```
chasePlot(v,'Centerline','on')
```



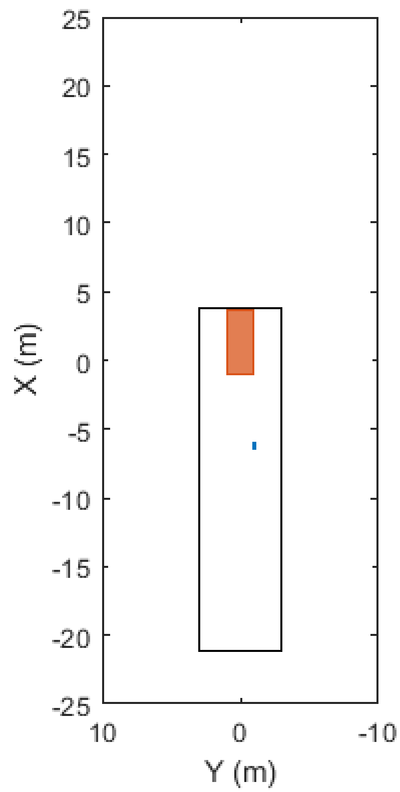
Create a bird's-eye plot, outline plotter, and lane boundary plotter.

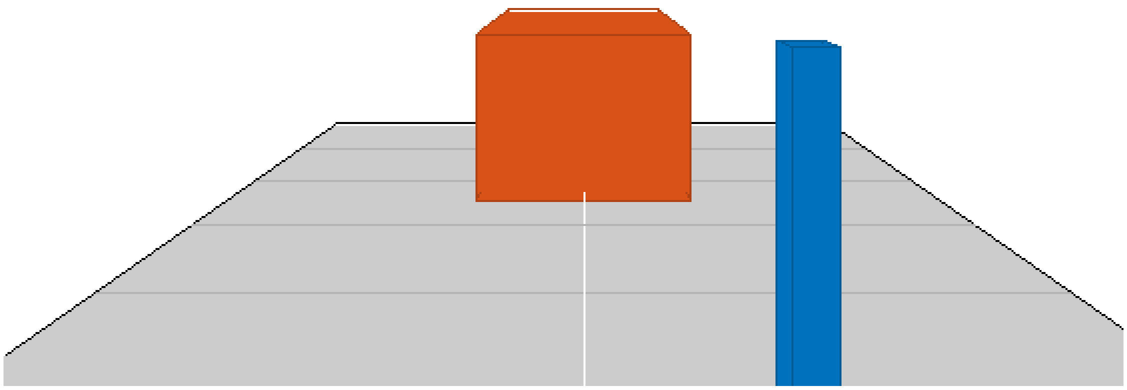
```
bep = birdsEyePlot('XLim',[-25 25],'YLim',[-10 10]);
olPlotter = outlinePlotter(bep);
lbPlotter = laneBoundaryPlotter(bep);
legend('off')
```



Run the simulation loop. Update the plotter with outlines for the targets.

```
while advance(scenario)
   % Obtain the road boundaries and rectangular outlines.
   rb = roadBoundaries(v);
   [position,yaw,length,width,originOffset,color] = targetOutlines(v);

   % Update the bird's-eye plotters with the road and actors.
   plotLaneBoundary(lbPlotter,rb);
   plotOutline(olPlotter,position,yaw,length,width, ...
               'OriginOffset',originOffset,'Color',color);

   % Allow time for plot to update.
   pause(0.01)
end
```


25
20
15
10
5
0
-5
-10
-15
-20
-25
X (m)
10
0
-10
Y (m)



Input Arguments

`olPlotter` — Outline plotter
`OutlinePlotter` object

Outline plotter, specified as an `OutlinePlotter` object. This object is stored in the `Plotters` property of a `birdsEyePlot` object and configures the display of the specified outlines in the bird's-eye plot. To create this object, use the `outlinePlotter` function.

`positions` — Positions of detected objects
M-by-2 real-valued matrix

Positions of detected objects in vehicle coordinates, specified as an *M*-by-2 real-valued matrix of (*X*, *Y*) positions. *M* is the number of detected objects. The positive *X*-direction points ahead of the center of the vehicle. The positive *Y*-direction points to the left of the origin of the vehicle, which is the center of the rear axle, as shown in this figure of the vehicle coordinate system.

yaw — Angles of rotation
M-element real-valued vector

Angles of rotation for object outlines, specified as an *M*-element real-valued vector, where *M* is the number of objects.

length — Lengths of outlines
M-element real-valued vector

Lengths of object outlines, specified as an *M*-element real-valued vector, where *M* is the number of objects.

width — Widths of outlines
M-element real-valued vector

Widths of object outlines, specified as an *M*-element real-valued vector, where *M* is the number of objects.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Marker','x'`

`OriginOffset` — Rotational centers of rectangles relative to origin
M-by-2 real-valued matrix

Rotational centers of rectangles relative to origin, specified as the comma-separated pair consisting of `'OriginOffset'` and an *M*-by-2 real-valued matrix. *M* is the number of objects. Each row corresponds to the rotational center about which to rotate the corresponding rectangle, specified as an (*X*,*Y*) displacement from the geometrical center of that rectangle.

`Color` — Outline color
M-by-3 matrix of RGB triplets

Outline color, specified as the comma-separated pair consisting of `'Color'` and an *M*-by-3 matrix of RGB triplets. *M* is the number of objects. If you do not specify this argument, the function uses the default colormap for each object.

Example: `'Color',[0 0.5 0.75; 0.8 0.3 0.1]`

See Also
`birdsEyePlot` | `outlinePlotter`

Introduced in R2017b

plotPointCloud

Package:

Display generated point cloud on bird's-eye plot

Syntax

```
plotPointCloud(pcPlotter,pcObject)
plotPointCloud(pcPlotter,pointCloudMatrix)
```

Description

`plotPointCloud(pcPlotter,pcObject)` displays a point cloud generated from a point cloud data object, `pcObject`. The point cloud plotter, `pcPlotter`, is associated with a `birdsEyePlot` object and configures the display of the specified point cloud.

To remove the point cloud associated with the point cloud plotter, use the `clearData` function with `pcPlotter` specified as the input argument.

`plotPointCloud(pcPlotter,pointCloudMatrix)` specifies the point cloud data as a matrix of 2-D or 3-D points, `pointCloudMatrix`.

Examples

Generate Lidar Point Cloud Data of Multiple Actors

Generate lidar point cloud data for a driving scenario with multiple actors by using the `lidarPointCloudGenerator` System object. Create the driving scenario by using `drivingScenario` object. It contains an ego-vehicle, pedestrian and two other vehicles.

Create and plot a driving scenario with multiple vehicles

Create a driving scenario.

```
scenario = drivingScenario;
```

Add a straight road to the driving scenario. The road has one lane in each direction.

```
roadCenters = [0 0 0; 70 0 0];
laneSpecification = lanespec([1 1]);
road(scenario,roadCenters,'Lanes',laneSpecification);
```

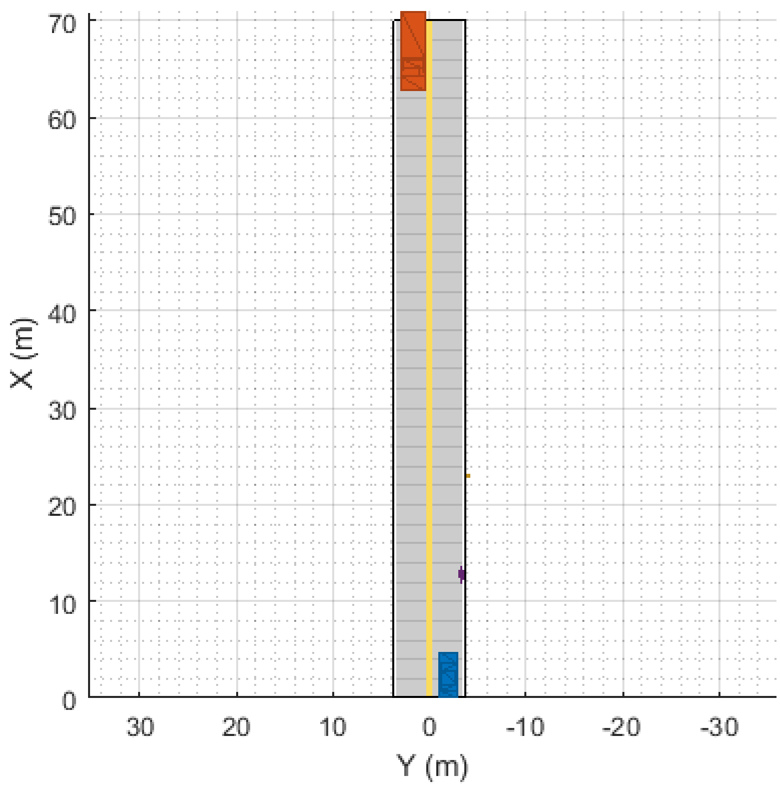
Add an ego vehicle to the driving scenario.

```
egoVehicle = vehicle(scenario,'ClassID',1,'Mesh',driving.scenario.carMesh);
waypoints = [1 -2 0; 35 -2 0];
trajectory(egoVehicle,waypoints,10);
```

Add a truck, pedestrian, and bicycle to the driving scenario and plot the scenario.

```
truck = vehicle(scenario,'ClassID',2,'Length', 8.2,'Width',2.5,'Height',3.5, ...
  'Mesh',driving.scenario.truckMesh);
```

```
waypoints = [70 1.7 0; 20 1.9 0];
trajectory(truck,waypoints,15);
pedestrian = actor(scenario,'ClassID',4,'Length',0.24,'Width',0.45,'Height',1.7, ...
  'Mesh',driving.scenario.pedestrianMesh);
waypoints = [23 -4 0; 10.4 -4 0];
trajectory(pedestrian,waypoints,1.5);
bicycle = actor(scenario,'ClassID',3,'Length',1.7,'Width',0.45,'Height',1.7, ...
  'Mesh',driving.scenario.bicycleMesh);
waypoints = [12.7 -3.3 0; 49.3 -3.3 0];
trajectory(bicycle,waypoints,5);
plot(scenario,'Meshes','on')
```



Generate and plot lidar point cloud data

Create a `lidarPointCloudGenerator` System object.

```
lidar = lidarPointCloudGenerator;
```

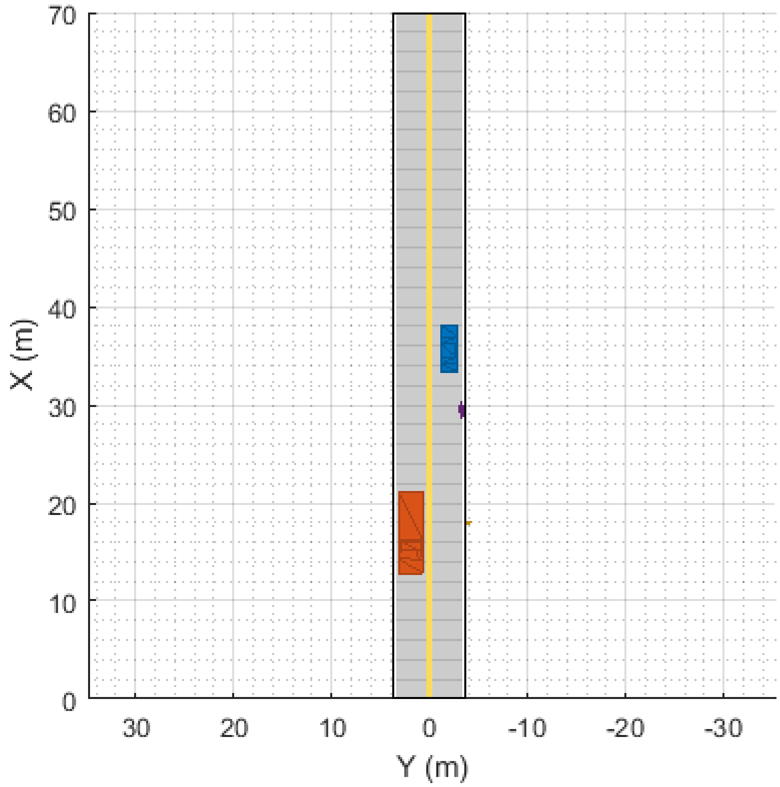
Add actor profiles and the ego vehicle actor ID from the driving scenario to the System object.

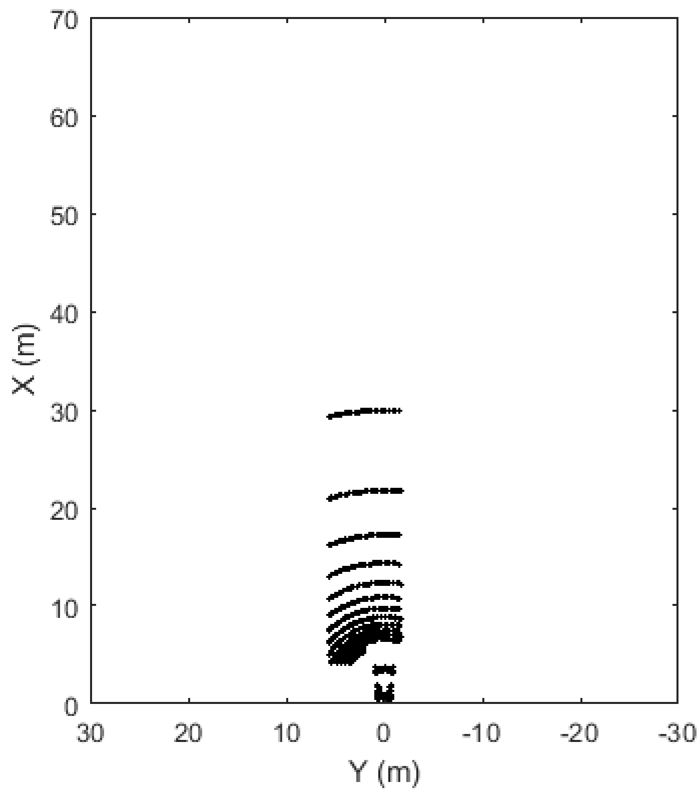
```
lidar.ActorProfiles = actorProfiles(scenario);
lidar.EgoVehicleActorID = egoVehicle.ActorID;
```

Plot the point cloud data.

```
bep = birdsEyePlot('Xlimits',[0 70],'YLimits',[-30 30]);
plotter = pointCloudPlotter(bep);
legend('off');
```

```
while advance(scenario)
    tgts = targetPoses(egoVehicle);
    rdmesh = roadMesh(egoVehicle);
    [ptCloud,isValidTime] = lidar(tgts,rdmesh,scenario.SimulationTime);
    if isValidTime
        plotPointCloud(plotter,ptCloud);
    end
end
```





Input Arguments

`pcPlotter` — Point cloud plotter
`pointCloudPlotter` object

Point cloud plotter, specified as a `pointCloudPlotter` object. This object is stored in the `Plotters` property of a `birdsEyePlot` object and configures the display of the specified point cloud in the bird's-eye plot. The property names correspond to the name-value pair arguments of the `pointCloudPlotter` function.

`pcObject` — Point cloud data object
`pointCloud` object

Point cloud data object, specified as a `pointCloud` object.

`pointCloudMatrix` — Point cloud data matrix
N-by-2 or *N*-by-3 real-valued matrix

Point cloud data matrix, specified as a *N*-by-2 or *N*-by-3 real-valued matrix. Each element in the first column of the matrix corresponds to the *x*- coordinates of a point in the point cloud. The elements in the second and third columns correspond to *y*- and *z*- coordinates.

Data Types: `single` | `double`

See Also
`birdsEyePlot` | `clearData` | `clearPlotterData` | `findPlotter` | `pointCloudPlotter`

Introduced in R2020a

plotPath

Display actor paths on bird’s-eye plot

Syntax

```
plotPath(pPlotter,pathCoords)
```

Description

`plotPath(pPlotter,pathCoords)` displays the paths of actors from a list of path coordinates on a bird's-eye plot. The path plotter object, `pPlotter`, is associated with a `birdsEyePlot` object and configures the display of the specified path.

To remove all paths associated with the path plotter `pPlotter`, call the `clearData` function and specify `pPlotter` as the input argument.

Examples

Plot Path of Ego Vehicle

Create a 3-meter-wide lane.

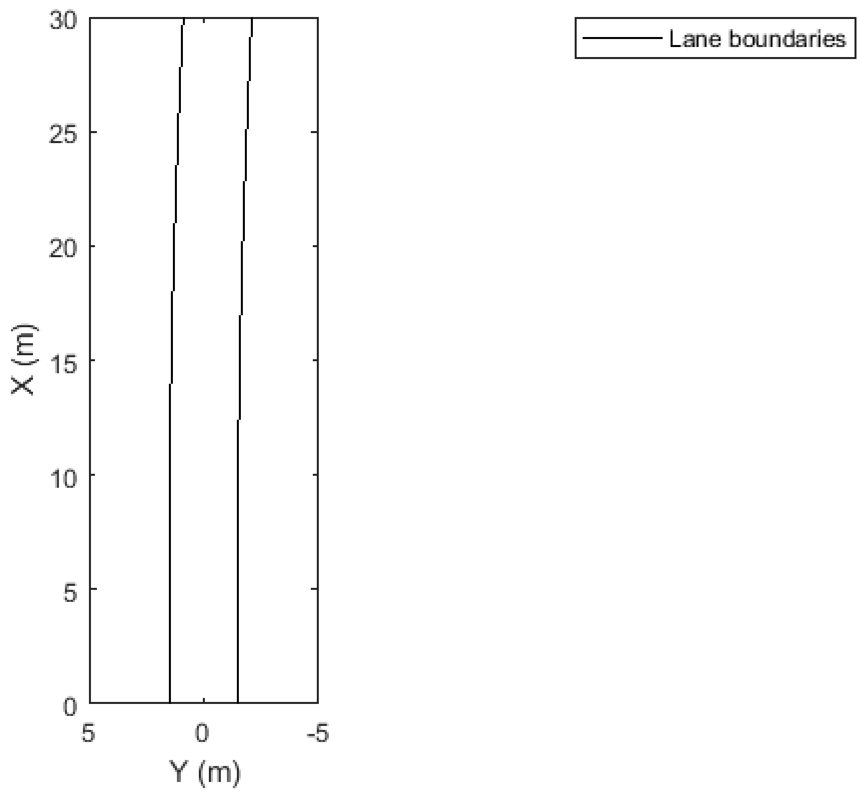
```
lb = parabolicLaneBoundary([-0.001,0.01,1.5]);
rb = parabolicLaneBoundary([-0.001,0.01,-1.5]);
```

Compute the lane boundary model manually from 0 to 30 meters along the *x*-axis.

```
xWorld = (0:30)';
yLeft = computeBoundaryModel(lb,xWorld);
yRight = computeBoundaryModel(rb,xWorld);
```

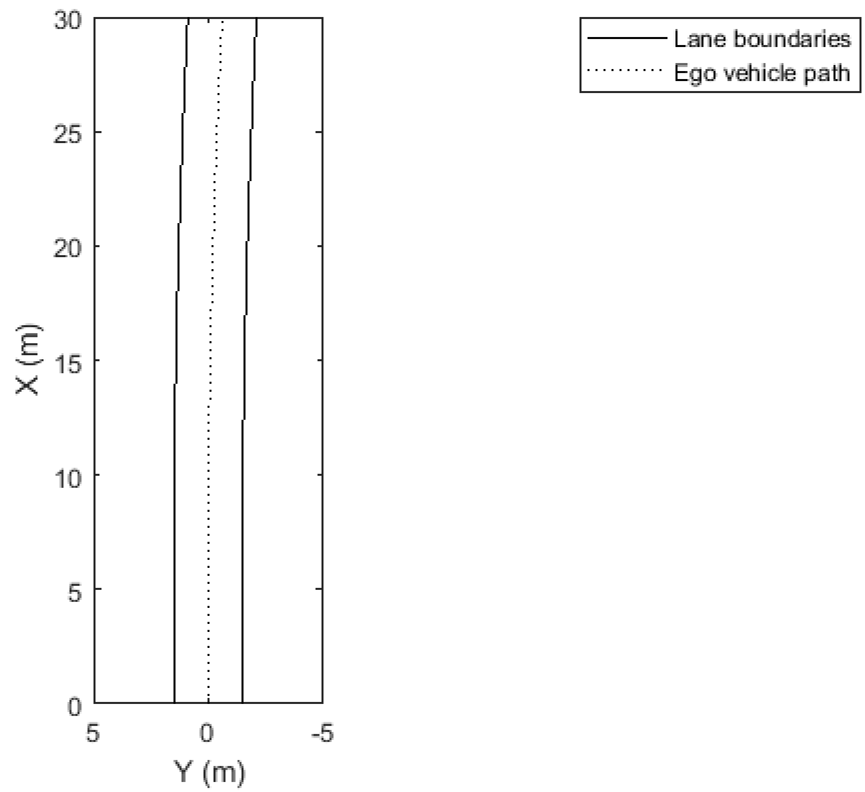
Create a bird's-eye plot and lane boundary plotter. Display the lane information on the bird's-eye plot.

```
bep = birdsEyePlot('XLimits',[0 30],'YLimits',[-5 5]);
lanePlotter = laneBoundaryPlotter(bep,'DisplayName','Lane boundaries');
plotLaneBoundary(lanePlotter,{[xWorld,yLeft],[xWorld,yRight]});
```



Create a path plotter. Create and display the path of an ego vehicle that travels through the center of the lane.

```
yCenter = (yLeft + yRight)/2;
egoPathPlotter = pathPlotter(bep,'DisplayName','Ego vehicle path');
plotPath(egoPathPlotter,{[xWorld,yCenter]});
```



Input Arguments

pPlotter — Path plotter
PathPlotter object

Path plotter, specified as a PathPlotter object. This object is stored in the Plotters property of a birdsEyePlot object and configures the display of the specified actor paths in the bird's-eye plot. To create this object, use the pathPlotter function.

pathCoords — Path coordinates
cell array of *M*-by-2 real-valued matrices

Path coordinates, specified as a cell array of *M*-by-2 real-valued matrices. Each matrix represents the coordinates for a different path. *M* is the number of coordinates in a path and can be different for each path. The first and second columns of each matrix represent the (*X*, *Y*) positions of the path curve. The positive *X*-direction points ahead of the center of the vehicle. The positive *Y*-direction points to the left of the origin of the vehicle, which is the center of the rear axle, as shown in this figure of the vehicle coordinate system..

Path coordinates are relative to the ego vehicle.

See Also

`birdsEyePlot` | `pathPlotter`

Introduced in R2017a

plotTrack

Display object tracks on bird's-eye plot

Syntax

```
plotTrack(tPlotter,positions)
plotTrack(tPlotter,positions,velocities)
plotTrack(tPlotter,positions,labels)
plotTrack(tPlotter,positions,covariances)
plotTrack(tPlotter,positions,velocities,labels,covariances)
```

Description

`plotTrack(tPlotter,positions)` displays object tracks from a list of object positions on a bird's-eye plot. The track plotter, `tPlotter`, is associated with a `birdsEyePlot` object and configures the display of the object tracks.

To remove all tracks associated with track plotter `tPlotter`, call the `clearData` function and specify `tPlotter` as the input argument.

`plotTrack(tPlotter,positions,velocities)` displays tracks and their velocities on a bird's-eye plot.

`plotTrack(tPlotter,positions,labels)` displays tracks and their labels on a bird's-eye plot.

`plotTrack(tPlotter,positions,covariances)` displays tracks and the covariances of track uncertainties on a bird's-eye plot.

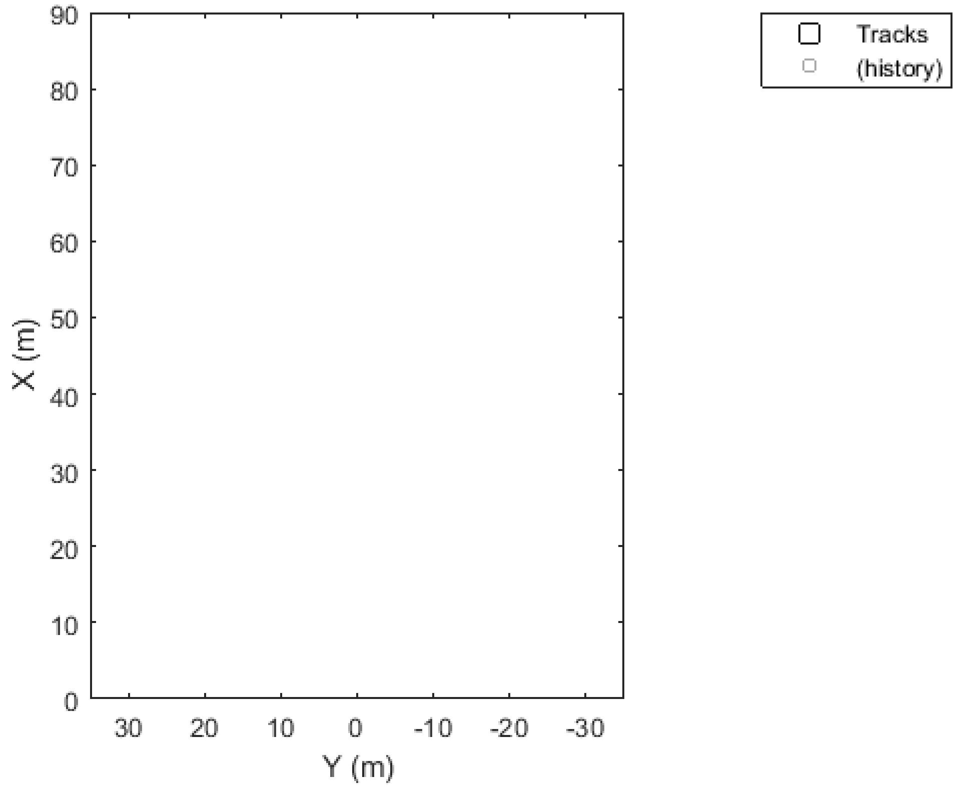
`plotTrack(tPlotter,positions,velocities,labels,covariances)` displays tracks and their velocities, labels, and covariances on a bird's-eye plot. You can specify one or more of `velocities`, `labels`, and `covariances`. These arguments can appear in any order but they must come after `tPlotter` and `positions`.

Examples

Create and Display Labeled Tracks on Bird's-Eye Plot

Create a bird's-eye plot with an *x*-axis range from 0 to 90 meters and a *y*-axis range from –35 to 35 meters. Create a track plotter that displays up to seven history values for each track and offsets labels by 3 meters in front of the tracks.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
tPlotter = trackPlotter(bep,'DisplayName','Tracks','HistoryDepth',7,'LabelOffset',[3 0]);
```

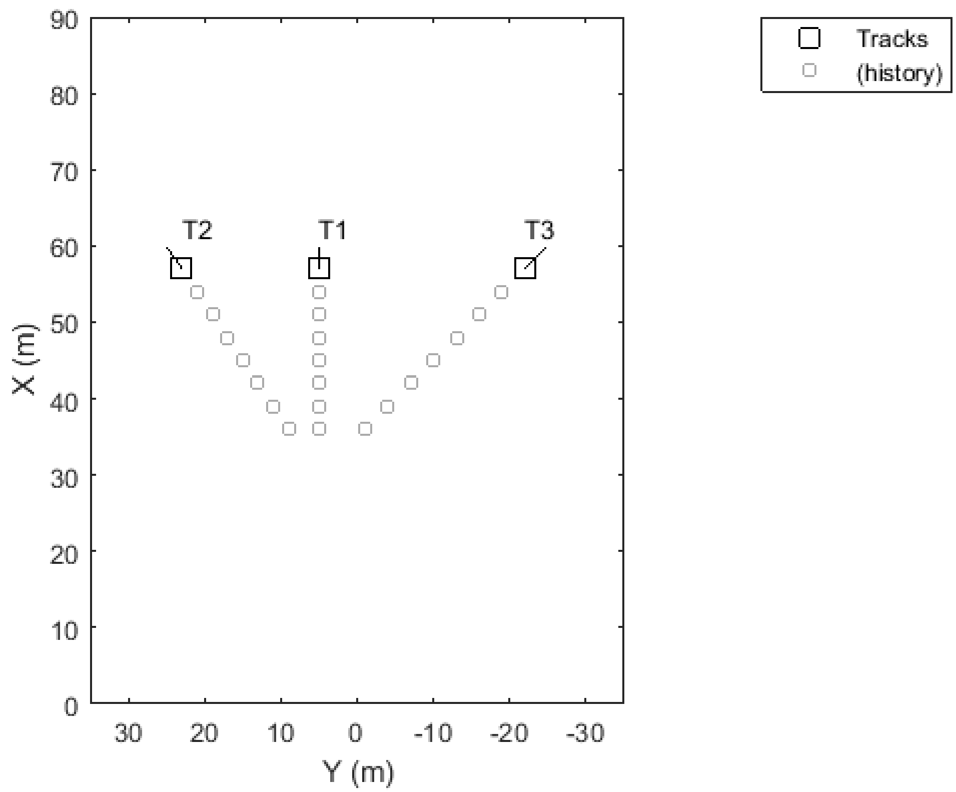


Set the positions and velocities of three labeled tracks.

```
positions = [30, 5; 30, 5; 30, 5];
velocities = [3, 0; 3, 2; 3, -3];
labels = {'T1','T2','T3'};
```

Display the tracks for 10 trials. The bird's-eye plot shows the seven history values specified previously.

```
for i=1:10
   plotTrack(tPlotter,positions,velocities,labels);
   positions = positions + velocities;
end
```



Input Arguments

`tPlotter` — Track plotter
`TrackPlotter` object

Track plotter, specified as a `TrackPlotter` object. This object is stored in the `Plotters` property of a `birdsEyePlot` object and configures the display of the specified tracks in the bird's-eye plot. To create this object, use the `trackPlotter` function.

`positions` — Positions of tracked objects
M-by-2 real-valued matrix

Positions of tracked objects in vehicle coordinates, specified as an *M*-by-2 real-valued matrix of (*X*, *Y*) positions. *M* is the number of tracked objects. The positive *X*-direction points ahead of the center of the vehicle. The positive *y*-direction points to the left of the origin of the vehicle, which is the center of the rear axle, as shown in this figure of the vehicle coordinate system.

`velocities` — Velocities of tracked objects
M-by-2 real-valued matrix

Velocities of tracked objects, specified as an *M*-by-2 real-valued matrix of velocities in the (*X*, *Y*) direction. *M* is the number of tracked objects. The velocities are plotted as line vectors that originate from the center positions of the tracked objects.

`labels` — Track labels
M-length string array | *M*-length cell array of character vectors

Track labels, specified as an *M*-length string array or an *M*-length cell array of character vectors. *M* is the number of tracked objects. The labels correspond to the locations in the `positions` matrix. By default, tracks do not have labels. To remove all annotations and labels associated with the track plotter, use the `clearData` function.

`covariances` — Covariances of track uncertainties
2-by-2-by-*M* real-valued array

Covariances of track uncertainties centered at the track positions, specified as a 2-by-2-by-*M* real-valued array. The uncertainties are plotted as an ellipse.

See Also
`birdsEyePlot` | `trackPlotter`

Introduced in R2017a

trackPlotter

Package:

Track plotter for bird's-eye plot

Syntax

```
tPlotter = trackPlotter(bep)
tPlotter = trackPlotter(bep,Name,Value)
```

Description

`tPlotter = trackPlotter(bep)` creates a `TrackPlotter` object that configures the display of tracks on a bird's-eye plot. The `TrackPlotter` object is stored in the `Plotters` property of the input `birdsEyePlot` object, `bep`. To display the tracks, use the `plotTrack` function.

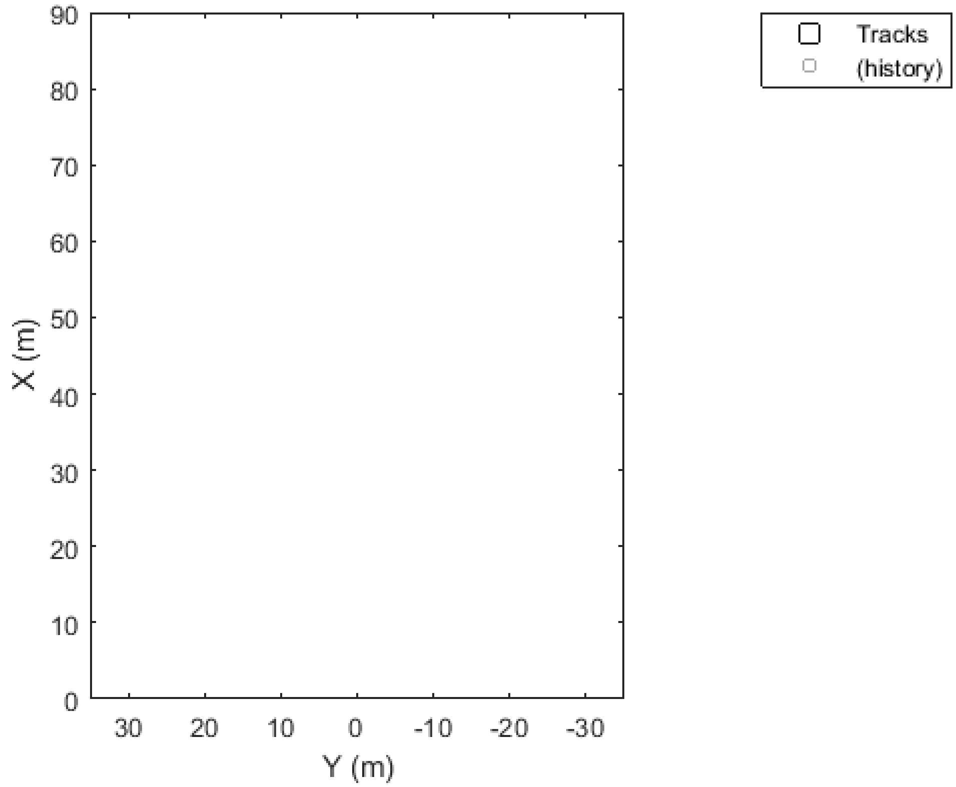
`tPlotter = trackPlotter(bep,Name,Value)` sets properties using one or more `Name,Value` pair arguments. For example, `trackPlotter(bep,'DisplayName','Tracks')` sets the display name that appears in the bird's-eye-plot legend.

Examples

Create and Display Labeled Tracks on Bird's-Eye Plot

Create a bird's-eye plot with an *x*-axis range from 0 to 90 meters and a *y*-axis range from –35 to 35 meters. Create a track plotter that displays up to seven history values for each track and offsets labels by 3 meters in front of the tracks.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
tPlotter = trackPlotter(bep,'DisplayName','Tracks','HistoryDepth',7,'LabelOffset',[3 0]);
```

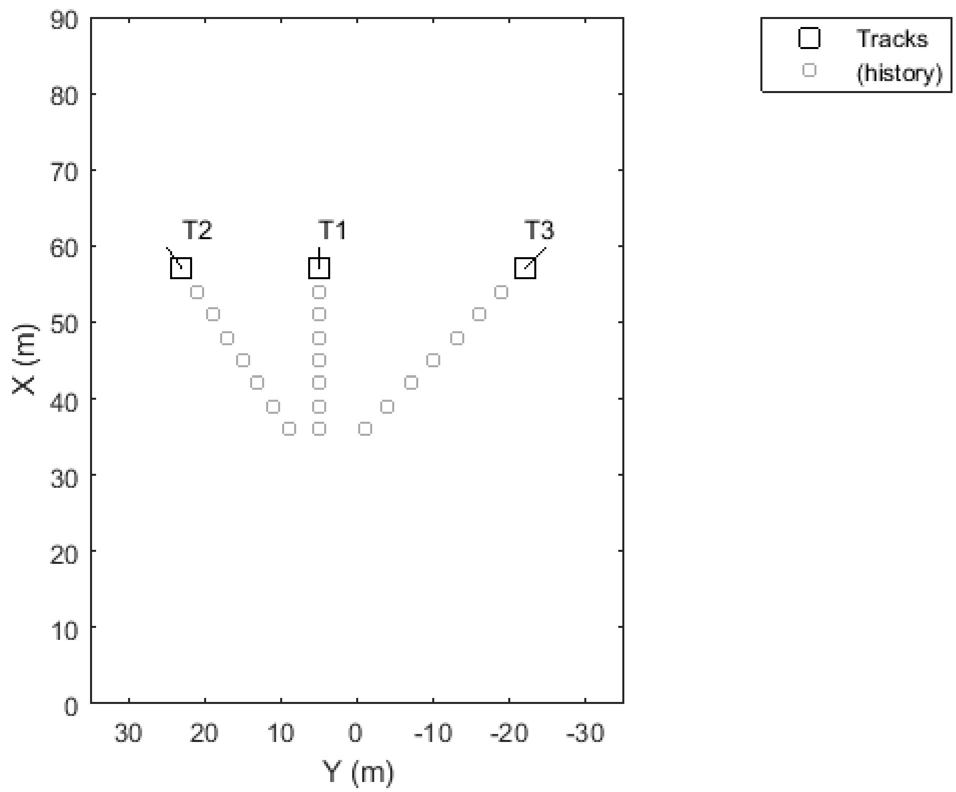


Set the positions and velocities of three labeled tracks.

```
positions = [30, 5; 30, 5; 30, 5];
velocities = [3, 0; 3, 2; 3, -3];
labels = {'T1','T2','T3'};
```

Display the tracks for 10 trials. The bird's-eye plot shows the seven history values specified previously.

```
for i=1:10
   plotTrack(tPlotter,positions,velocities,labels);
   positions = positions + velocities;
end
```



Input Arguments

bep — Bird's-eye plot
birdsEyePlot object

Bird's-eye plot, specified as a `birdsEyePlot` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `trackPlotter('Marker','*')` sets the marker symbol for tracks to an asterisk.

`DisplayName` — Plotter name to display in legend
`''` (default) | character vector | string scalar

Plotter name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and character vector or string scalar. If you do not specify a name, the bird's-eye plot does not display a legend entry for the plotter.

`HistoryDepth` — Number of previous track updates to display
`0` (default) | integer in the range [0, 100]

Number of previous track updates to display, specified as the comma-separated pair consisting of `'HistoryDepth'` and an integer in the range [0, 100]. When you set this value to `0`, the bird's-eye plot displays no previous updates.

`Marker` — Marker symbol for tracks
`'square'` (default) | `'+'` | `'*'` | `'.'` | `'x'` | ...

Marker symbol for tracks, specified as the comma-separated pair consisting of `'Marker'` and one of the markers in this table.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'_'`	Horizontal line
`'|'`	Vertical line
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerSize` — Size of marker for tracks
`10` (default) | positive integer

Size of marker for tracks, specified as the comma-separated pair consisting of `'MarkerSize'` and a positive integer in points.

`MarkerEdgeColor` — Marker outline color for tracks
`[0 0 0]` (black) (default) | RGB triplet | hexadecimal color code | color name | short color name

Marker outline color for tracks, specified as the comma-separated pair consisting of `'MarkerEdgeColor'` and an RGB triplet, a hexadecimal color code, a color name, or a short color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`MarkerFaceColor` — Marker fill color for tracks

`'none'` (default) | RGB triplet | hexadecimal color code | color name | short color name

Marker fill color for tracks, specified as the comma-separated pair consisting of `'MarkerFaceColor'` and an RGB triplet, a hexadecimal color code, a color name, or a short color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

FontSize — Font size for labeling tracks
10 points (default) | positive integer

Font size for labeling tracks, specified as the comma-separated pair consisting of 'FontSize' and a positive integer in font points.

LabelOffset — Gap between label and positional point
[0 0] (default) | real-valued vector of the form [*x* *y*]

Gap between label and positional point, specified as the comma-separated pair consisting of 'LabelOffset' and a real-valued vector of the form [*x* *y*]. Units are in meters.

VelocityScaling — Scale factor for magnitude length of velocity vectors
1 (default) | positive real scalar

Scale factor for magnitude length of velocity vectors, specified as the comma-separated pair consisting of 'VelocityScaling' and a positive real scalar. The bird's-eye plot renders the magnitude vector value as M × VelocityScaling, where M is the magnitude of velocity.

Tag — Tag associated with plotter object
'Plotter*N*' (default) | character vector | string scalar

Tag associated with the plotter object, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. The default value is 'Plotter*N*', where *N* is an integer that corresponds to the *N*th plotter associated with the input birdsEyePlot object.

Output Arguments

tPlotter — Track plotter
TrackPlotter object

Track plotter, returned as a TrackPlotter object. You can modify this object by changing its property values. The property names correspond to the name-value pair arguments of the trackPlotter function.

tPlotter is stored in the Plotters property of the input birdsEyePlot object, bep. To plot the tracks, use the plotTrack function.

See Also

birdsEyePlot | clearData | clearPlotterData | findPlotter | plotTrack

Introduced in R2017a

birdsEyeView

Create bird's-eye view using inverse perspective mapping

Description

Use the `birdsEyeView` object to create a bird's-eye view of a 2-D scene using inverse perspective mapping. To transform an image into a bird's-eye view, pass a `birdsEyeView` object and that image to the `transformImage` function. To convert the bird’s-eye-view image coordinates to or from vehicle coordinates, use the `imageToVehicle` and `vehicleToImage` functions. All of these functions assume that the input image does not have lens distortion. To remove lens distortion, use the `undistortImage` function.

Creation

Syntax

```
birdsEye = birdsEyeView(sensor,outView,outImageSize)
```

Description

`birdsEye = birdsEyeView(sensor,outView,outImageSize)` creates a `birdsEyeView` object for transforming an image to a bird’s-eye-view.

- `sensor` is a `monoCamera` object that defines the configuration of the camera sensor. This input sets the `Sensor` property.
- `outView` defines the portion of the camera view, in vehicle coordinates, that is transformed into a bird's-eye view. This input sets the `OutputView` property.
- `outImageSize` defines the size, in pixels, of the output bird's-eye-view image. This input sets the `ImageSize` property.

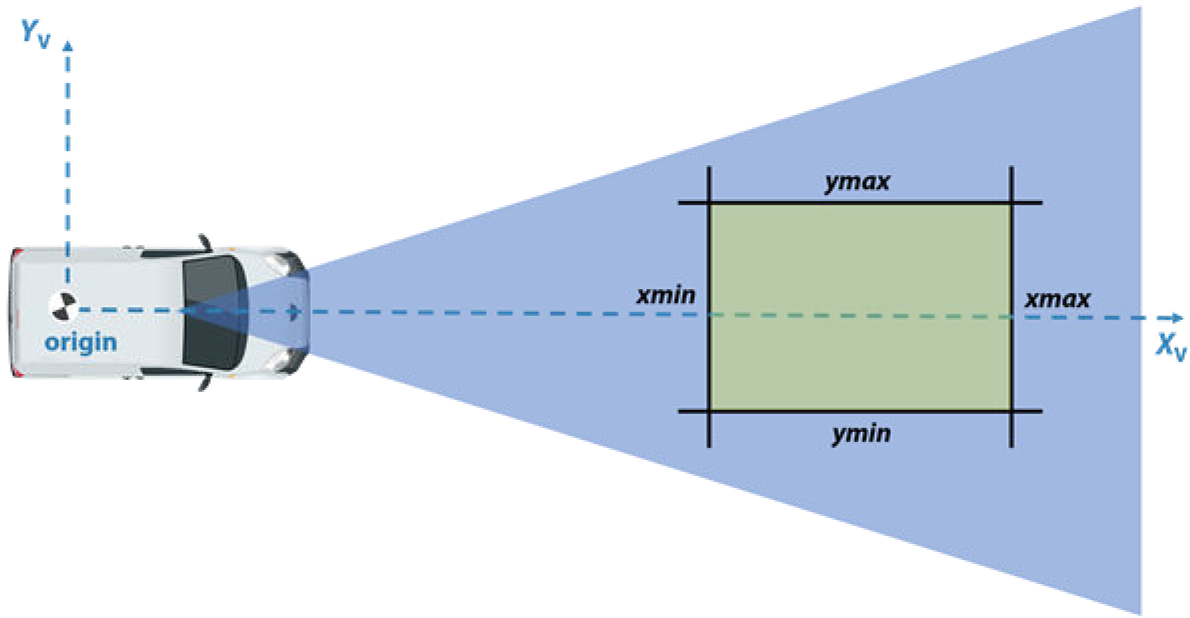
Properties

`Sensor` — Camera sensor configuration
`monoCamera` object

Camera sensor configuration, specified as a `monoCamera` object. The object contains the intrinsic camera parameters, the mounting height, and the camera mounting angles. This configuration defines the vehicle coordinate system of the `birdsEyeView` object. For more details, see “Vehicle Coordinate System” on page 4-140.

`OutputView` — Coordinates of region to transform
four-element vector of form [*xmin xmax ymin ymax*]

Coordinates of the region to transform into a bird's-eye-view image, specified as a four-element vector of the form [*xmin xmax ymin ymax*]. The units are in world coordinates, such as meters or feet, as determined by the `Sensor` property. The four coordinates define the output space in the vehicle coordinate system (X_V,Y_V).



You can set this property when you create the object. After you create the object, this property is read-only.

ImageSize — Size of output bird's-eye-view images
two-element vector

Size of output bird's-eye-view images, in pixels, specified as a two-element vector of the form [*m n*], where *m* and *n* specify the number of rows and columns of pixels for the output image, respectively. If you specify a value for one dimension, you can set the other dimension to `NaN` and `birdsEyeView` calculates this value automatically. Setting one dimension to `NaN` maintains the same pixel to world-unit ratio along the X_V-axis and Y_V-axis.

You can set this property when you create the object. After you create the object, this property is read-only.

Object Functions

transformImage	Transform image to bird's-eye view
imageToVehicle	Convert bird's-eye-view image coordinates to vehicle coordinates
vehicleToImage	Convert vehicle coordinates to bird's-eye-view image coordinates

Examples

Transform Road Image to Bird's-Eye-View Image

Create a bird's-eye-view image from an image obtained by a front-facing camera mounted on a vehicle. Display points within the bird's-eye view using the vehicle and image coordinate systems.

Define the camera intrinsics and create an object containing these intrinsics.

```
focalLength = [309.4362 344.2161];
principalPoint = [318.9034 257.5352];
```

```
imageSize = [480 640];

camIntrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);
```

Set the height of the camera to be about 2 meters above the ground. Set the pitch of the camera to 14 degrees toward the ground.

```
height = 2.1798;
pitch = 14;
```

Create an object containing the camera configuration.

```
sensor = monoCamera(camIntrinsics,height,'Pitch',pitch);
```

Define the area in front of the camera that you want to transform into a bird's-eye view. Set an area from 3 to 30 meters in front of the camera, with 6 meters to either side of the camera.

```
distAhead = 30;
spaceToOneSide = 6;
bottomOffset = 3;

outView = [bottomOffset,distAhead,-spaceToOneSide,spaceToOneSide];
```

Set the output image width to 250 pixels. Compute the output length automatically from the width by setting the length to `NaN`.

```
outImageSize = [NaN,250];
```

Create an object for performing bird's-eye-view transforms, using the previously defined parameters.

```
birdsEye = birdsEyeView(sensor,outView,outImageSize);
```

Load an image that was captured by the sensor.

```
I = imread('road.png');
figure
imshow(I)
title('Original Image')
```

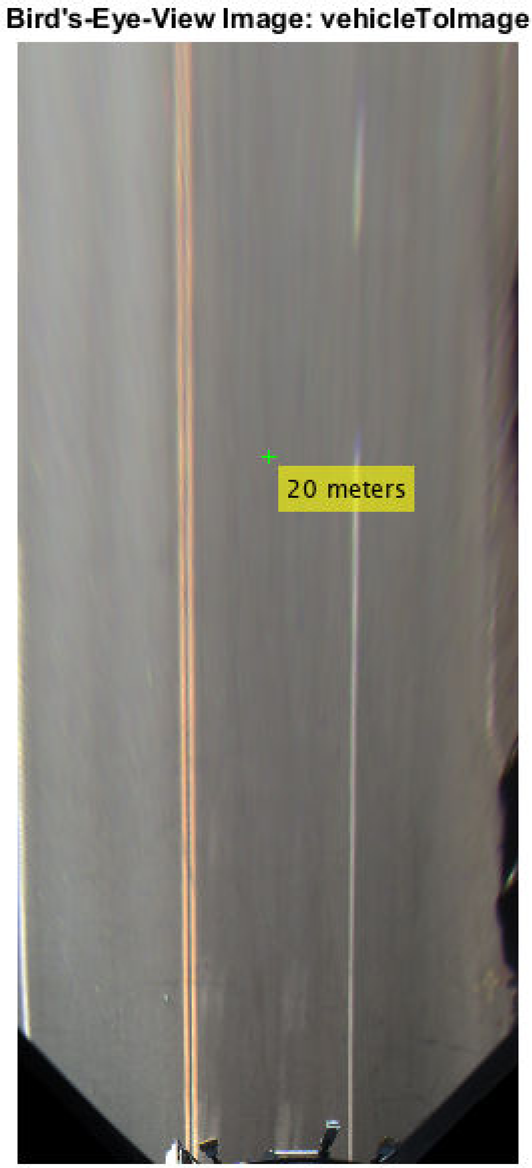
Original Image

Transform the input image into a bird's-eye-view image.

```
BEV = transformImage(birdsEye,I);
```

In the bird's-eye-view image, place a 20-meter marker directly in front of the sensor. Use the `vehicleToImage` function to specify the location of the marker in vehicle coordinates. Display the marker on the bird's-eye-view image.

```
imagePoint = vehicleToImage(birdsEye,[20 0]);
annotatedBEV = insertMarker(BEV,imagePoint);
annotatedBEV = insertText(annotatedBEV,imagePoint + 5,'20 meters');

figure
imshow(annotatedBEV)
title('Bird''s-Eye-View Image: vehicleToImage')
```



Define a location in the original bird's-eye-view image, this time in image coordinates. Use the `imageToVehicle` function to convert the image coordinates to vehicle coordinates. Display the distance between the marker and the front of the vehicle.

```
imagePoint2 = [120 400];
annotatedBEV = insertMarker(BEV,imagePoint2);

vehiclePoint = imageToVehicle(birdsEye,imagePoint2);
xAhead = vehiclePoint(1);
displayText = sprintf('%.2f meters',xAhead);
annotatedBEV = insertText(annotatedBEV,imagePoint2 + 5,displayText);
```

```
figure
imshow(annotatedBEV)
title('Bird''s-Eye-View Image: imageToVehicle')
```

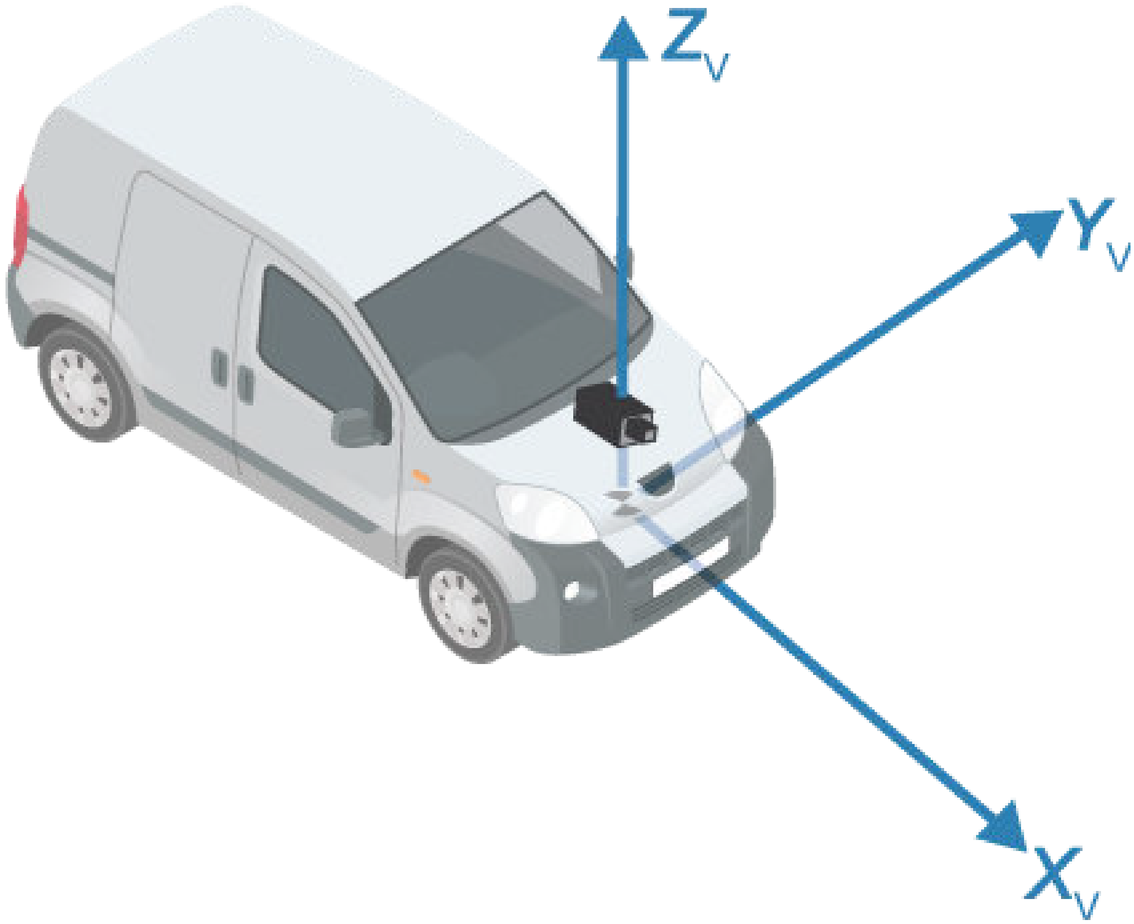
More About

Vehicle Coordinate System

In the vehicle coordinate system (X_V, Y_V, Z_V) defined by the input `monoCamera` object:

- 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 default origin of this coordinate system is on the road surface, directly below the camera center. The focal point of the camera defines this center point.



To change the placement of the origin within the vehicle coordinate system, update the `SensorLocation` property of the input `monoCamera` object.

For more details about the vehicle coordinate system, see "Coordinate Systems in Automated Driving Toolbox".

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`monoCamera`

Topics
"Coordinate Systems in Automated Driving Toolbox"

Introduced in R2017a

imageToVehicle

Convert bird's-eye-view image coordinates to vehicle coordinates

Syntax

```
vehiclePoints = imageToVehicle(birdsEye,imagePoints)
```

Description

`vehiclePoints = imageToVehicle(birdsEye,imagePoints)` converts bird’s-eye-view image coordinates to [*x y*] vehicle coordinates.

Examples

Transform Road Image to Bird's-Eye-View Image

Create a bird's-eye-view image from an image obtained by a front-facing camera mounted on a vehicle. Display points within the bird's-eye view using the vehicle and image coordinate systems.

Define the camera intrinsics and create an object containing these intrinsics.

```
focalLength = [309.4362 344.2161];
principalPoint = [318.9034 257.5352];
imageSize = [480 640];

camIntrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);
```

Set the height of the camera to be about 2 meters above the ground. Set the pitch of the camera to 14 degrees toward the ground.

```
height = 2.1798;
pitch = 14;
```

Create an object containing the camera configuration.

```
sensor = monoCamera(camIntrinsics,height,'Pitch',pitch);
```

Define the area in front of the camera that you want to transform into a bird's-eye view. Set an area from 3 to 30 meters in front of the camera, with 6 meters to either side of the camera.

```
distAhead = 30;
spaceToOneSide = 6;
bottomOffset = 3;

outView = [bottomOffset,distAhead,-spaceToOneSide,spaceToOneSide];
```

Set the output image width to 250 pixels. Compute the output length automatically from the width by setting the length to `NaN`.

```
outImageSize = [NaN,250];
```

Create an object for performing bird's-eye-view transforms, using the previously defined parameters.

```
birdsEye = birdsEyeView(sensor,outView,outImageSize);
```

Load an image that was captured by the sensor.

```
I = imread('road.png');
figure
imshow(I)
title('Original Image')
```

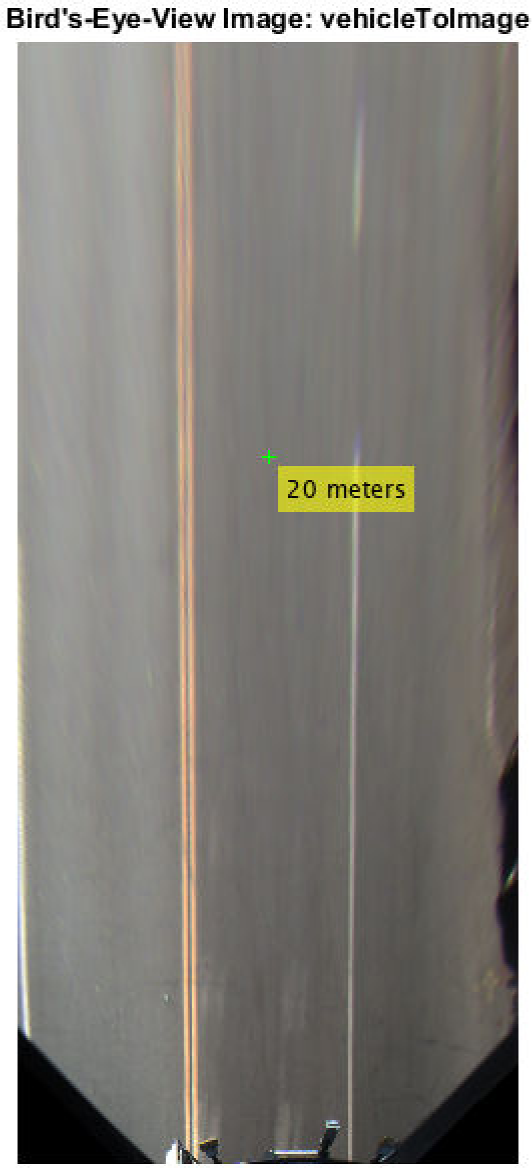
Original Image

Transform the input image into a bird's-eye-view image.

```
BEV = transformImage(birdsEye,I);
```

In the bird's-eye-view image, place a 20-meter marker directly in front of the sensor. Use the `vehicleToImage` function to specify the location of the marker in vehicle coordinates. Display the marker on the bird's-eye-view image.

```
imagePoint = vehicleToImage(birdsEye,[20 0]);
annotatedBEV = insertMarker(BEV,imagePoint);
annotatedBEV = insertText(annotatedBEV,imagePoint + 5,'20 meters');

figure
imshow(annotatedBEV)
title('Bird''s-Eye-View Image: vehicleToImage')
```



Define a location in the original bird's-eye-view image, this time in image coordinates. Use the `imageToVehicle` function to convert the image coordinates to vehicle coordinates. Display the distance between the marker and the front of the vehicle.

```
imagePoint2 = [120 400];
annotatedBEV = insertMarker(BEV,imagePoint2);

vehiclePoint = imageToVehicle(birdsEye,imagePoint2);
xAhead = vehiclePoint(1);
displayText = sprintf('%.2f meters',xAhead);
annotatedBEV = insertText(annotatedBEV,imagePoint2 + 5,displayText);
```

```
figure
imshow(annotatedBEV)
title('Bird''s-Eye-View Image: imageToVehicle')
```

Input Arguments

birdsEye — Object for transforming image to bird's-eye view
birdsEyeView object

Object for transforming image to bird's-eye view, specified as a `birdsEyeView` object.

`imagePoints` — Image points
M-by-2 matrix

Image points, specified as an *M*-by-2 matrix containing *M* number of [*x* *y*] image coordinates.

Output Arguments

`vehiclePoints` — Vehicle points
M-by-2 matrix

Vehicle points, returned as an *M*-by-2 matrix containing *M* number of [*x* *y*] vehicle coordinates.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`birdsEyeView`

Functions
`vehicleToImage`

Topics
"Coordinate Systems in Automated Driving Toolbox"

Introduced in R2017a

transformImage

Transform image to bird's-eye view

Syntax

```
J = transformImage(birdsEye,I)
```

Description

`J = transformImage(birdsEye,I)` transforms the input image, `I`, to a bird’s-eye-view image, `J`. The `OutputView` and `ImageSize` properties of the `birdsEyeView` object, `birdsEye`, determine the portion of `I` to transform and the size of `J`, respectively.

Examples

Transform Road Image to Bird's-Eye-View Image

Create a bird's-eye-view image from an image obtained by a front-facing camera mounted on a vehicle. Display points within the bird's-eye view using the vehicle and image coordinate systems.

Define the camera intrinsics and create an object containing these intrinsics.

```
focalLength = [309.4362 344.2161];
principalPoint = [318.9034 257.5352];
imageSize = [480 640];

camIntrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);
```

Set the height of the camera to be about 2 meters above the ground. Set the pitch of the camera to 14 degrees toward the ground.

```
height = 2.1798;
pitch = 14;
```

Create an object containing the camera configuration.

```
sensor = monoCamera(camIntrinsics,height,'Pitch',pitch);
```

Define the area in front of the camera that you want to transform into a bird's-eye view. Set an area from 3 to 30 meters in front of the camera, with 6 meters to either side of the camera.

```
distAhead = 30;
spaceToOneSide = 6;
bottomOffset = 3;

outView = [bottomOffset,distAhead,-spaceToOneSide,spaceToOneSide];
```

Set the output image width to 250 pixels. Compute the output length automatically from the width by setting the length to `NaN`.

```
outImageSize = [NaN,250];
```

Create an object for performing bird's-eye-view transforms, using the previously defined parameters.

```
birdsEye = birdsEyeView(sensor,outView,outImageSize);
```

Load an image that was captured by the sensor.

```
I = imread('road.png');
figure
imshow(I)
title('Original Image')
```

Original Image

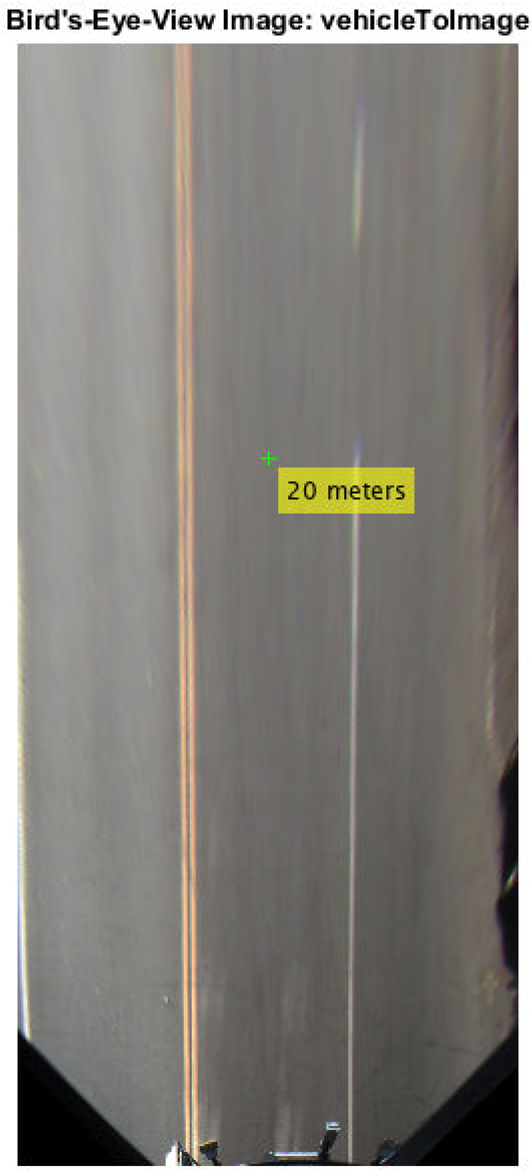
Transform the input image into a bird's-eye-view image.

```
BEV = transformImage(birdsEye,I);
```

In the bird's-eye-view image, place a 20-meter marker directly in front of the sensor. Use the `vehicleToImage` function to specify the location of the marker in vehicle coordinates. Display the marker on the bird's-eye-view image.

```
imagePoint = vehicleToImage(birdsEye,[20 0]);
annotatedBEV = insertMarker(BEV,imagePoint);
annotatedBEV = insertText(annotatedBEV,imagePoint + 5,'20 meters');

figure
```

```
imshow(annotatedBEV)
title('Bird''s-Eye-View Image: vehicleToImage')
```



Define a location in the original bird's-eye-view image, this time in image coordinates. Use the `imageToVehicle` function to convert the image coordinates to vehicle coordinates. Display the distance between the marker and the front of the vehicle.

```
imagePoint2 = [120 400];
annotatedBEV = insertMarker(BEV,imagePoint2);

vehiclePoint = imageToVehicle(birdsEye,imagePoint2);
```

```
xAhead = vehiclePoint(1);
displayText = sprintf('%.2f meters',xAhead);
annotatedBEV = insertText(annotatedBEV,imagePoint2 + 5,displayText);

figure
imshow(annotatedBEV)
title('Bird''s-Eye-View Image: imageToVehicle')
```

Input Arguments

birdsEye — Object for transforming image to bird's-eye view
birdsEyeView object

Object for transforming image to bird's-eye view, specified as a birdsEyeView object.

I — Input image
truecolor image | grayscale image

Input image, specified as a truecolor or grayscale image. The OutputView property of birdsEye determines the portion of I to transform to a bird's-eye view.

I must not contain lens distortion. You can remove lens distortion by using the undistortImage function. In high-end optics, you can ignore distortion.

Output Arguments

J — Bird's-eye-view image
truecolor image | grayscale image

Bird's-eye-view image, returned as a truecolor or grayscale image. The ImageSize property of birdsEye determines the size of J.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
birdsEyeView

Functions
imageToVehicle | vehicleToImage

Introduced in R2017a

vehicleToImage

Convert vehicle coordinates to bird's-eye-view image coordinates

Syntax

`imagePoints = vehicleToImage(birdsEye,vehiclePoints)`

Description

`imagePoints = vehicleToImage(birdsEye,vehiclePoints)` converts vehicle coordinates to [*x y*] bird's-eye-view image coordinates.

Examples

Transform Road Image to Bird's-Eye-View Image

Create a bird's-eye-view image from an image obtained by a front-facing camera mounted on a vehicle. Display points within the bird's-eye view using the vehicle and image coordinate systems.

Define the camera intrinsics and create an object containing these intrinsics.

```
focalLength = [309.4362 344.2161];
principalPoint = [318.9034 257.5352];
imageSize = [480 640];

camIntrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);
```

Set the height of the camera to be about 2 meters above the ground. Set the pitch of the camera to 14 degrees toward the ground.

```
height = 2.1798;
pitch = 14;
```

Create an object containing the camera configuration.

```
sensor = monoCamera(camIntrinsics,height,'Pitch',pitch);
```

Define the area in front of the camera that you want to transform into a bird's-eye view. Set an area from 3 to 30 meters in front of the camera, with 6 meters to either side of the camera.

```
distAhead = 30;
spaceToOneSide = 6;
bottomOffset = 3;

outView = [bottomOffset,distAhead,-spaceToOneSide,spaceToOneSide];
```

Set the output image width to 250 pixels. Compute the output length automatically from the width by setting the length to `NaN`.

```
outImageSize = [NaN,250];
```

Create an object for performing bird's-eye-view transforms, using the previously defined parameters.

```
birdsEye = birdsEyeView(sensor,outView,outImageSize);
```

Load an image that was captured by the sensor.

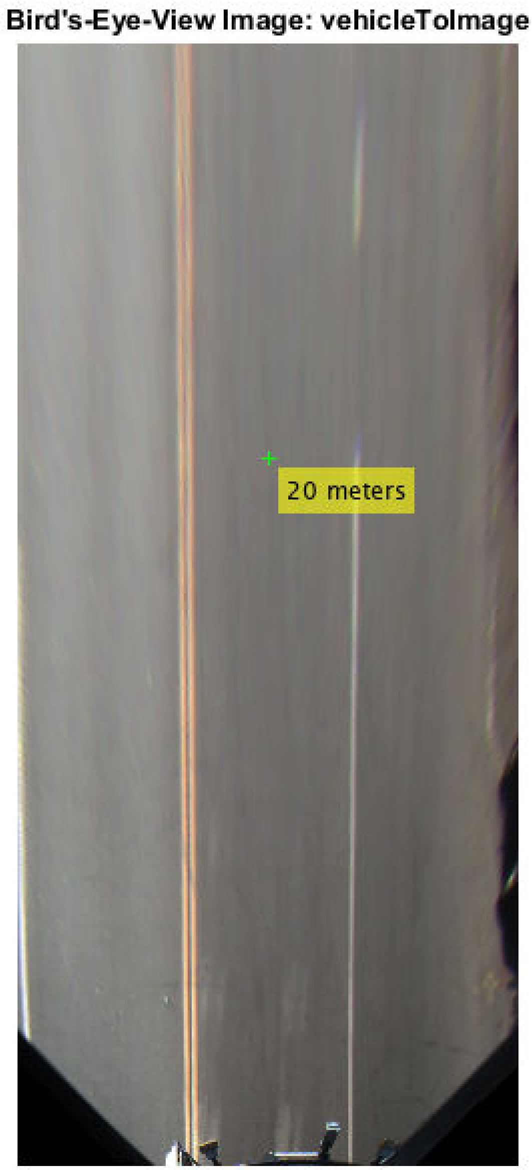
```
I = imread('road.png');
figure
imshow(I)
title('Original Image')
```

Transform the input image into a bird's-eye-view image.

```
BEV = transformImage(birdsEye,I);
```

In the bird's-eye-view image, place a 20-meter marker directly in front of the sensor. Use the `vehicleToImage` function to specify the location of the marker in vehicle coordinates. Display the marker on the bird's-eye-view image.

```
imagePoint = vehicleToImage(birdsEye,[20 0]);
annotatedBEV = insertMarker(BEV,imagePoint);
annotatedBEV = insertText(annotatedBEV,imagePoint + 5,'20 meters');

figure
imshow(annotatedBEV)
title('Bird''s-Eye-View Image: vehicleToImage')
```



Define a location in the original bird's-eye-view image, this time in image coordinates. Use the `imageToVehicle` function to convert the image coordinates to vehicle coordinates. Display the distance between the marker and the front of the vehicle.

```
imagePoint2 = [120 400];
annotatedBEV = insertMarker(BEV,imagePoint2);

vehiclePoint = imageToVehicle(birdsEye,imagePoint2);
xAhead = vehiclePoint(1);
displayText = sprintf('%.2f meters',xAhead);
annotatedBEV = insertText(annotatedBEV,imagePoint2 + 5,displayText);
```

```
figure
imshow(annotatedBEV)
title('Bird''s-Eye-View Image: imageToVehicle')
```

Input Arguments

birdsEye — Object for transforming image to bird's-eye view
birdsEyeView object

Object for transforming image to bird's-eye view, specified as a `birdsEyeView` object.

`vehiclePoints` — Vehicle points
M-by-2 matrix

Vehicle points, specified as an *M*-by-2 matrix containing *M* number of [*x* *y*] vehicle coordinates.

Output Arguments

`imagePoints` — Image points
M-by-2 matrix

Image points, returned as an *M*-by-2 matrix containing *M* number of [*x* *y*] image coordinates.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`birdsEyeView`

Functions
`imageToVehicle`

Topics
"Coordinate Systems in Automated Driving Toolbox"

Introduced in R2017a

driving.connector.Connector class

Package: `driving.connector`

Interface to connect external tool to Ground Truth Labeler app

Description

The `driving.connector.Connector` class creates an interface between a custom visualization or analysis tool and a signal in the **Ground Truth Labeler** app. You can use the connector with video and image sequence signals only.

The `driving.connector.Connector` class is a `handle` class.

Creation

The `Connector` class that is inherited from the `Connector` interface is called a client.

The client can:

- Sync an external tool to each frame change event for a specific signal loaded into the **Ground Truth Labeler**. Syncing allows you to control the external tool through the range slider and playback controls of the app.
- Control the current time in the external tool and the corresponding display in the app.
- Export custom labeled data from an external tool via the app.

To connect an external tool to the **Ground Truth Labeler** app, follow these steps:

1. Define a client class that inherits from `driving.connector.Connector`. You can use the `Connector` class template to define a class and implement your custom visualization or analysis tool. At the MATLAB command prompt, enter:

   ```
   driving.connector.Connector.openTemplateInEditor
   ```

   Follow the steps found in the template.
2. Save the file to any folder on the MATLAB path. Alternatively, save the file to a folder and add the folder to MATLAB path by using the `addpath` function.

Properties

`VideoStartTime` — Start time of signal
real scalar in seconds

Start time of the signal, specified as a real scalar in seconds.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

`VideoEndTime` — End time of signal
real scalar in seconds

End time of the signal, specified as a real scalar in seconds.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

`StartTime` — Start of time interval in app
real scalar in seconds

Start of the time interval in the app, specified as a real scalar in seconds. To set the start time, use the start flag interval in the app.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

`CurrentTime` — Time of frame currently displaying in app
real scalar in seconds

Time of the frame currently displaying in the app for the connected signal, specified as a real scalar in seconds. If the slider is between two timestamps, then the currently displaying frame is the frame that is at the previous timestamp. For more details, see “Control Playback of Signal Frames for Labeling”.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

`EndTime` — End of time interval in app
real scalar in seconds

End of the time interval in the app, specified as a real scalar in seconds. To set the end time, use the end flag interval in the app.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

`TimeVector` — Timestamps for connected signal
`duration` vector

Timestamps for the connected signal, specified as a `duration` vector. This signal must be the master signal. If you change the master signal, the `TimeVector` property updates to the timestamps for new master signal.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

LabelData — Label data imported from external tool

two-column table

Label data imported from the external tool, specified as a two-column table. The first column contains the timestamps of the connected signal and the second column contains the label information that you specify for the corresponding timestamp.

Attributes:

GetAccess	public
SetAccess	private

LabelName — Names of labels

character vector | string scalar | cell array of character vectors | string array

Names of labels, specified as a character vector, a string scalar, a cell array of character vectors, or a string array. These names must be valid MATLAB variables that correspond to the label names specified in the second column of `LabelData`.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

LabelDescription — Descriptions of labels

' ' (default) | character vector | string scalar | cell array of character vectors | string array

Descriptions of labels, specified as a character vector, a string scalar, a cell array of character vectors, or a string array. Each description of `LabelDescription` corresponds to a label specified in `LabelName`.

Attributes:

GetAccess	public
SetAccess	public

Methods

Public Methods

frameChangeListener	Update external tool when new frame is displayed in Ground Truth Labeler app
labelDefinitionLoadListener	Update external tool for new label definitions in Ground Truth Labeler app
labelLoadListener	Update external tool for new label data in Ground Truth Labeler app
addLabelData	Add custom label data at current time
queryLabelData	Query for custom label data at current time
updateLabelerCurrentTime	Update current time in Ground Truth Labeler app
close	Close external tool connected to Ground Truth Labeler app
disconnect	Disconnect external tool from Ground Truth Labeler app
dataSourceChangeListener	Update external tool when connecting to signal being loaded into Ground Truth Labeler app

Examples

Connect Lidar Display to Ground Truth Labeler

Connect a lidar display tool to the **Ground Truth Labeler** app. Use the app and tool to display synchronized lidar and video data.

Specify the name of a video signal to load into the app.

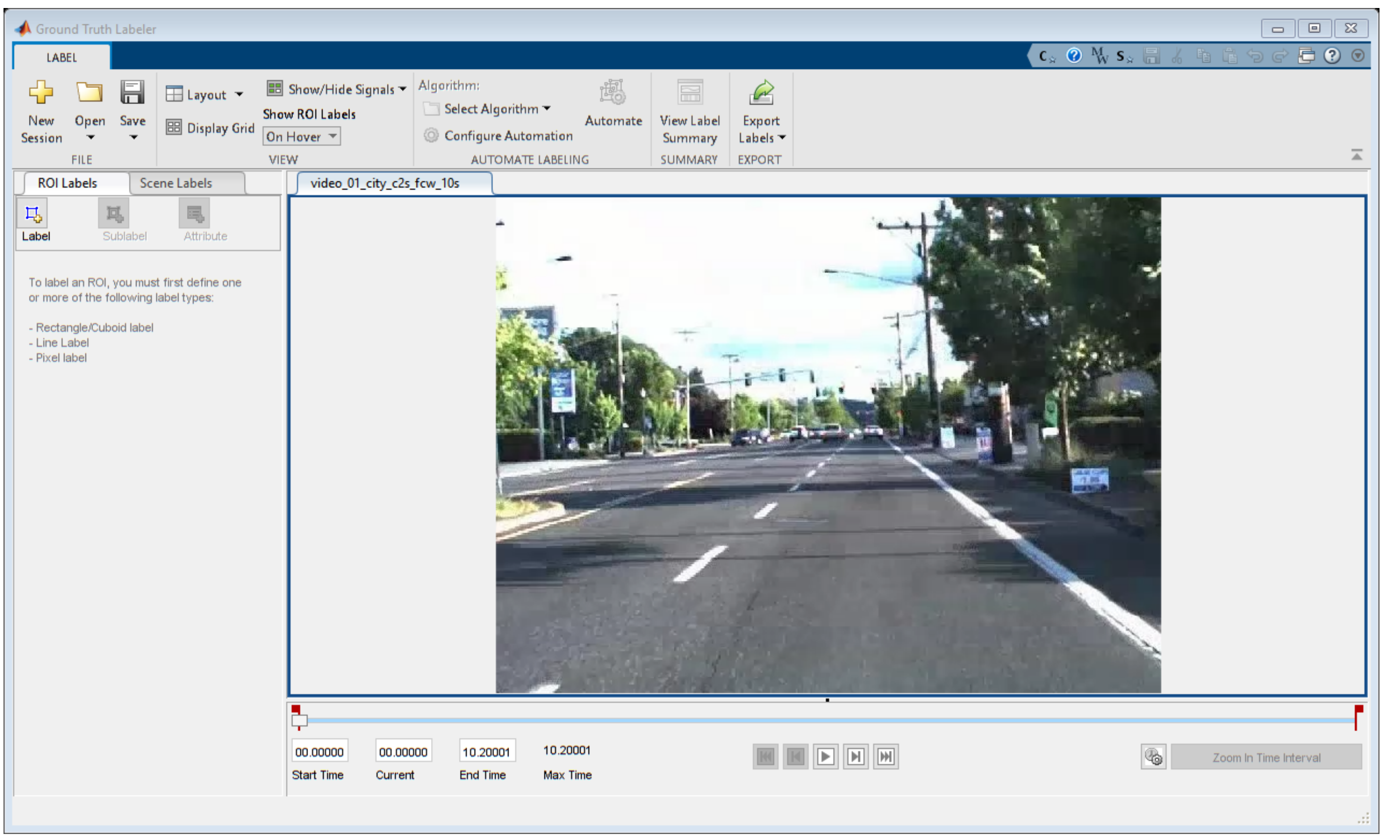
```
signalName = '01_city_c2s_fcw_10s.mp4';
```

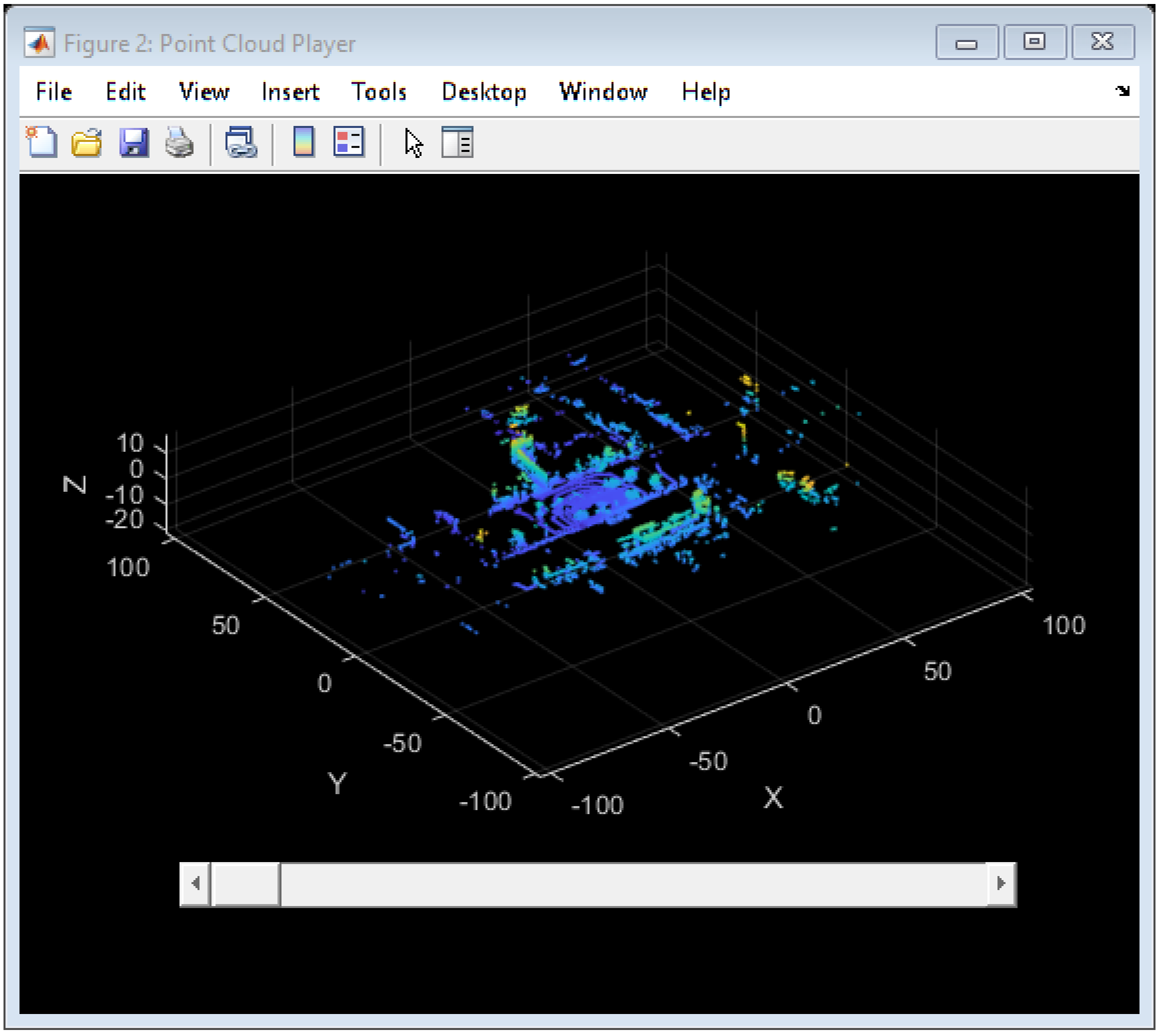
Add the path to the function handle for the lidar display tool.

```
path = fullfile(toolboxdir('driving'),'drivingdemos');
addpath(path)
```

Connect the lidar display to the app.

```
groundTruthLabeler(signalName,'ConnectorTargetHandle',@LidarDisplay);
```





After the app loads the video and lidar display tool, remove the path to the function handle.

```
rmpath(path)
```

Tips

- For an example of an external tool, see this `driving.connector.Connector` class implementation. This class implements a lidar visualization tool. You can use this code as a starting point for creating your own tools.

  ```
  edit LidarDisplay
  ```
- To keep an external tool synchronized with the app, specify timestamps that are at the same frame rate as the signals loaded in the app. If the tool visualizes data at a timestamp that is between two frames, then the app displays the frame that is at the previous timestamp. For more details, see "Control Playback of Signal Frames for Labeling".

See Also

Apps
Ground Truth Labeler

Introduced in R2017a

addLabelData

Class: `driving.connector.Connector`
Package: `driving.connector`

Add custom label data at current time

Syntax

`addLabelData(connectorObj,labelData)`

Description

`addLabelData(connectorObj,labelData)` adds the custom label data related to the current time that is shown in the **Ground Truth Labeler** app. The client calls this method using the `connectorObj` object.

Note The client class can call this method.

Input Arguments

`connectorObj` — Connector object
`driving.connector.Connector` object

Connector object, specified as a `driving.connector.Connector` object.

`labelData` — Label data
cell array of character vectors | string array

Label data, specified as a cell array of character vectors or as a string array. Each element of `labelData` must correspond to a label stored in the `labelData` property of the input `driving.connector.Connector` object, `connectorObj`. The app appends label data only to the signal to which the external tool is connected.

See Also

Ground Truth Labeler | `driving.connector.Connector`

Introduced in R2017a

close

Class: `driving.connector.Connector`
Package: `driving.connector`

Close external tool connected to Ground Truth Labeler app

Syntax

`close(connectorObj)`

Description

`close(connectorObj)` provides the option to close the external tool that is connected to the **Ground Truth Labeler** app when the app closes. The app calls this method using the `connectorObj` object.

Note The client class can optionally implement this method.

Input Arguments

`connectorObj` — Connector object
`driving.connector.Connector` object

Connector object, specified as a `driving.connector.Connector` object.

See Also

Ground Truth Labeler | `driving.connector.Connector`

Introduced in R2017a

dataSourceChangeListener

Class: `driving.connector.Connector`
Package: `driving.connector`

Update external tool when connecting to signal being loaded into Ground Truth Labeler app

Syntax

`dataSourceChangeListener(connectorObj)`

Description

`dataSourceChangeListener(connectorObj)` provides the option to update the external tool when a new data source is loaded into the **Ground Truth Labeler** app. The app calls this method using the `connectorObj` object.

You can optionally use this method to react to when the tool connects to a signal that is being loaded into the app.

Note The client class can optionally implement this method.

Input Arguments

`connectorObj` — Connector object
`driving.connector.Connector` object

Connector object, specified as a `driving.connector.Connector` object.

See Also

Ground Truth Labeler | `driving.connector.Connector`

Introduced in R2017a

disconnect

Class: `driving.connector.Connector`
Package: `driving.connector`

Disconnect external tool from Ground Truth Labeler app

Syntax

`disconnect(connectorObj)`

Description

`disconnect(connectorObj)` disconnects the interface between an external tool and the **Ground Truth Labeler** app. The client calls this method using the `connectorObj` object. After the external tool is disconnected, the **Ground Truth Labeler** app no longer calls the `frameChangeListener` method in the client class.

Note The client class can call this method.

Input Arguments

`connectorObj` — Connector object
`driving.connector.Connector` object

Connector object, specified as a `driving.connector.Connector` object.

See Also

Ground Truth Labeler | `driving.connector.Connector`

Introduced in R2017a

frameChangeListener

Class: `driving.connector.Connector`
Package: `driving.connector`

Update external tool when new frame is displayed in Ground Truth Labeler app

Syntax

`frameChangeListener(connectorObj)`

Description

`frameChangeListener(connectorObj)` provides an option to synchronize an external tool with the frame changes in the **Ground Truth Labeler** app. The app calls this method when a new frame is displayed in the app. If the slider is between two timestamps, then the app displays the frame that is at the previous timestamp. For more details, see "Control Playback of Signal Frames for Labeling".

Note The client class must implement this method.

Input Arguments

`connectorObj` — Connector object
`driving.connector.Connector` object

Connector object, specified as a `driving.connector.Connector` object.

See Also

Ground Truth Labeler | `driving.connector.Connector`

Introduced in R2017a

labelDefinitionLoadListener

Class: `driving.connector.Connector`
Package: `driving.connector`

Update external tool for new label definitions in Ground Truth Labeler app

Syntax

`labelDefinitionLoadListener(connectorObj)`

Description

`labelDefinitionLoadListener(connectorObj)` provides an option to update the external tool that is connected to the **Ground Truth Labeler** app when new set of label definitions is imported into the app. The app calls this method using the `connectorObj` object. You can optionally use this method to react to the event of connecting a new data source to the app.

Note The client class can optionally implement this method.

Input Arguments

`connectorObj` — Connector object
`driving.connector.Connector` object

Connector object, specified as a `driving.connector.Connector` object.

See Also

Ground Truth Labeler | `driving.connector.Connector`

Introduced in R2017a

labelLoadListener

Class: `driving.connector.Connector`
Package: `driving.connector`

Update external tool for new label data in Ground Truth Labeler app

Syntax

`labelLoadListener(connectorObj)`

Description

`labelLoadListener(connectorObj)` provides the option to update the external tool that is connected to the **Ground Truth Labeler** app when a new set of label data or new session with label data is imported into the app. The app calls this method using the `connectorObj` object. Use this method to react to the event of loading a new label data into the app.

Note The client class can optionally implement this method.

Input Arguments

`connectorObj` — Connector object
`driving.connector.Connector` object

Connector object, specified as a `driving.connector.Connector` object.

See Also

Ground Truth Labeler | `driving.connector.Connector`

Introduced in R2017a

queryLabelData

Class: `driving.connector.Connector`
Package: `driving.connector`

Query for custom label data at current time

Syntax

`queryLabelData(connectorObj)`

Description

`queryLabelData(connectorObj)` queries for label data related to the current time in the **Ground Truth Labeler** app. The client calls this method using the `connectorObj`.

Note The client class can call this method.

Input Arguments

`connectorObj` — Connector object
`driving.connector.Connector` object

Connector object, specified as a `driving.connector.Connector` object.

See Also

Ground Truth Labeler | `driving.connector.Connector`

Introduced in R2017a

updateLabelerCurrentTime

Class: `driving.connector.Connector`
Package: `driving.connector`

Update current time in Ground Truth Labeler app

Syntax

`updateLabelerCurrentTime(connectorObj,newTime)`

Description

`updateLabelerCurrentTime(connectorObj,newTime)` updates the current time in the **Ground Truth Labeler** app to `newTime`. The client calls this method using the `connectorObj` object.

Note The client class can call this method.

Input Arguments

`connectorObj` — Connector object
`driving.connector.Connector` object

Connector object, specified as a `driving.connector.Connector` object.

`newTime` — Current time for app
real scalar in seconds

Current time for app, specified as a real scalar in seconds. The `newTime` value sets the current time in the **Ground Truth Labeler** app.

See Also

Ground Truth Labeler | `driving.connector.Connector`

Introduced in R2017a

radarDetectionGenerator

Generate radar detections for driving scenario

Description

The `radarDetectionGenerator` System object generates detections from a radar sensor mounted on an ego vehicle. All detections are referenced to the coordinate system of the ego vehicle. You can use the `radarDetectionGenerator` object in a scenario containing actors and trajectories, which you can create by using a `drivingScenario` object. The object can simulate real detections with added random noise and also generate false alarm detections. In addition, you can use the `radarDetectionGenerator` object to create input to a `multiObjectTracker`. When building scenarios using the **Driving Scenario Designer** app, the radar sensors mounted on the ego vehicle are output as `radarDetectionGenerator` objects.

To generate radar detections:

1. Create the `radarDetectionGenerator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
sensor = radarDetectionGenerator
sensor = radarDetectionGenerator(Name,Value)
```

Description

`sensor = radarDetectionGenerator` creates a radar detection generator object with default property values.

`sensor = radarDetectionGenerator(Name,Value)` sets properties on page 4-173 using one or more name-value pairs. For example, `radarDetectionGenerator('DetectionCoordinates','Sensor Cartesian','MaxRange',200)` creates a radar detection generator that reports detections in the sensor Cartesian coordinate system and has a maximum detection range of 200 meters. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

SensorIndex — Unique sensor identifier
positive integer

Unique sensor identifier, specified as a positive integer. This property distinguishes detections that come from different sensors in a multisensor system.

Example: 5

Data Types: double

UpdateInterval — Required time interval between sensor updates
0.1 (default) | positive real scalar

Required time interval between sensor updates, specified as a positive real scalar. The `drivingScenario` object calls the radar detection generator at regular time intervals. The radar detector generates new detections at intervals defined by the `UpdateInterval` property. The value of the `UpdateInterval` property must be an integer multiple of the simulation time interval. Updates requested from the sensor between update intervals contain no detections. Units are in seconds.

Example: 5

Data Types: double

SensorLocation — Sensor location
[3.4 0] (default) | [x y] vector

Location of the radar sensor center, specified as an `[x y]` vector. The `SensorLocation` and `Height` properties define the coordinates of the radar sensor with respect to the ego vehicle coordinate system. The default value corresponds to a radar mounted at the center of the front grill of a sedan. Units are in meters.

Example: [4 0.1]

Data Types: double

Height — Radar sensor height above ground plane
0.2 (default) | positive real scalar

Radar sensor height above the ground plane, specified as a positive real scalar. The height is defined with respect to the vehicle ground plane. The `SensorLocation` and `Height` properties define the coordinates of the radar sensor with respect to the ego vehicle coordinate system. The default value corresponds to a radar mounted at the center of the front grill of a sedan. Units are in meters.

Example: 0.3

Data Types: double

Yaw — Yaw angle of sensor
0 (default) | real scalar

Yaw angle of radar sensor, specified as a real scalar. The yaw angle is the angle between the center line of the ego vehicle and the downrange axis of the radar sensor. A positive yaw angle corresponds to a clockwise rotation when looking in the positive direction of the *z*-axis of the ego vehicle coordinate system. Units are in degrees.

Example: -4

Data Types: double

Pitch — Pitch angle of sensor
0 (default) | real scalar

Pitch angle of sensor, specified as a real scalar. The pitch angle is the angle between the downrange axis of the radar sensor and the *x*-*y* plane of the ego vehicle coordinate system. A positive pitch angle corresponds to a clockwise rotation when looking in the positive direction of the *y*-axis of the ego vehicle coordinate system. Units are in degrees.

Example: 3

Data Types: double

Roll — Roll angle of sensor
0 (default) | real scalar

Roll angle of the radar sensor, specified as a real scalar. The roll angle is the angle of rotation of the downrange axis of the radar around the *x*-axis of the ego vehicle coordinate system. A positive roll angle corresponds to a clockwise rotation when looking in the positive direction of the *x*-axis of the coordinate system. Units are in degrees.

Example: -4

Data Types: double

FieldOfView — Azimuth and elevation fields of view of radar sensor
[20 5] | real-valued 1-by-2 vector of positive values

Azimuth and elevation fields of view of radar sensor, specified as a real-valued 1-by-2 vector of positive values, [azfov elfov]. The field of view defines the angular extent spanned by the sensor. Each component must lie in the interval (0,180]. Targets outside of the field of view of the radar are not detected. Units are in degrees.

Example: [14 7]

Data Types: double

MaxRange — Maximum detection range
150 | positive real scalar

Maximum detection range, specified as a positive real scalar. The radar cannot detect a target beyond this range. Units are in meters.

Example: 200

Data Types: double

RangeRateLimits — Minimum and maximum detection range rates
[-100 100] | real-valued 1-by-2 vector

Minimum and maximum detection range rates, specified as a real-valued 1-by-2 vector. The radar cannot detect a target out this range rate interval. Units are in meters per second.

Example: [-20 100]

Dependencies

To enable this property, set the `HasRangeRate` property to `true`.

Data Types: `double`

`DetectionProbability` — Probability of detecting a target
`0.9` | positive real scalar less than or equal to 1

Probability of detecting a target, specified as a positive real scalar less than or equal to one. This quantity defines the probability of detecting target that has a radar cross-section, `ReferenceRCS`, at the reference detection range, `ReferenceRange`.

`FalseAlarmRate` — False alarm rate
`1e-6` (default) | positive real scalar

False alarm rate within a radar resolution cell, specified as a positive real scalar in the range $[10^{-7},10^{-3}]$. Units are dimensionless.

Example: `1e-5`

Data Types: `double`

`ReferenceRange` — Reference range for given probability of detection
`100` (default) | positive real scalar

Reference range for a given probability of detection, specified as a positive real scalar. The reference range is the range when a target having a radar cross-section specified by `ReferenceRCS` is detected with a probability of specified by `DetectionProbability`. Units are in meters.

Data Types: `double`

`ReferenceRCS` — Reference radar cross-section for given probability of detection
`0` (default) | nonnegative real scalar

Reference radar cross-section (RCS) for given probability of detection, specified as a nonnegative real scalar. The reference RCS is the value at which a target is detected with probability specified by `DetectionProbability`. Units are in dBsm.

Data Types: `double`

`RadarLoopGain` — Radar loop gain
real scalar

This property is read-only.

Radar loop gain, specified as a real scalar. Radar loop gain is related to the reported signal-to-noise ratio of the radar, *SNR*, the target radar cross section, *RCS*, and target range, *R* by

SNR = RadarLoopGain + RCS - 40*log10(R)

SNR and *RCS* units are in dB and dBsm, respectively and range units are in meters. `RadarLoopGain` depends on the `DetectionProbability`, `ReferenceRange`, `ReferenceRCS`, and `FalseAlarmRate` property values. Units are in dB.

Data Types: `double`

AzimuthResolution — Azimuth resolution of radar
4 (default) | positive real scalar

Azimuth resolution of the radar, specified as a positive real scalar. The azimuth resolution defines the minimum separation in azimuth angle at which the radar can distinguish two targets. The azimuth resolution is typically the 3dB-downpoint in azimuth angle beamwidth of the radar. Units are in degrees.

Data Types: double

ElevationResolution — Elevation resolution of radar
10 (default) | positive real scalar

Elevation resolution of the radar, specified as a positive real scalar. The elevation resolution defines the minimum separation in elevation angle at which the radar can distinguish two targets. The elevation resolution is typically the 3dB-downpoint in elevation angle beamwidth of the radar. Units are in degrees.

Dependencies

To enable this property, set the HasElevation property to true.

Data Types: double

RangeResolution — Range resolution of radar
2.5 (default) | positive real scalar

Range resolution of the radar, specified as a positive real scalar. The range resolution defines the minimum separation in range at which the radar can distinguish between two targets. Units are in meters.

Data Types: double

RangeRateResolution — Range rate resolution of radar
0.5 (default) | positive real scalar

Range rate resolution of the radar, specified as a positive real scalar. The range rate resolution defines the minimum separation in range rate at which the radar can distinguish between two targets. Units are in meters per second.

Dependencies

To enable this property, set the HasRangeRate property to true.

Data Types: double

AzimuthBiasFraction — Azimuth bias fraction
0.1 (default) | nonnegative real scalar

Azimuth bias fraction of the radar, specified as a nonnegative real scalar. The azimuth bias is expressed as a fraction of the azimuth resolution specified in AzimuthResolution. Units are dimensionless.

Data Types: double

ElevationBiasFraction — Elevation bias fraction
0.1 (default) | nonnegative real scalar

Elevation bias fraction of the radar, specified as a nonnegative real scalar. Elevation bias is expressed as a fraction of the elevation resolution specified in `ElevationResolution`. Units are dimensionless.

Dependencies

To enable this property, set the `HasElevation` property to `true`.

Data Types: `double`

`RangeBiasFraction` — Range bias fraction
`0.05` (default) | nonnegative real scalar

Range bias fraction of the radar, specified as a nonnegative real scalar. Range bias is expressed as a fraction of the range resolution specified in `RangeResolution`. Units are dimensionless.

Data Types: `double`

`RangeRateBiasFraction` — Range rate bias fraction
`0.05` (default) | nonnegative real scalar

Range rate bias fraction of the radar, specified as a nonnegative real scalar. Range rate bias is expressed as a fraction of the range rate resolution specified in `RangeRateResolution`. Units are dimensionless.

Dependencies

To enable this property, set the `HasRangeRate` property to `true`.

Data Types: `double`

`HasElevation` — Enable radar to measure elevation
`false` (default) | `true`

Enable the radar to measure target elevation angles, specified as `false` or `true`. Set this property to `true` to model a radar sensor that can estimate target elevation. Set this property to `false` to model a radar sensor that cannot measure elevation.

Data Types: `logical`

`HasRangeRate` — Enable radar to measure range rate
`false` (default) | `true`

Enable the radar to measure target range rates, specified as `false` or `true`. Set this property to `true` to model a radar sensor which can estimate target range rate. Set this property to `false` to model a radar sensor that cannot measure range rate.

Data Types: `logical`

`HasNoise` — Enable adding noise to radar sensor measurements
`true` (default) | `false`

Enable adding noise to radar sensor measurements, specified as `true` or `false`. Set this property to `true` to add noise to the radar measurements. Otherwise, the measurements have no noise. Even if you set `HasNoise` to `false`, the object still computes the `MeasurementNoise` property of each detection.

Data Types: `logical`

HasFalseAlarms — Enable creating false alarm radar detections
true (default) | false

Enable reporting false alarm radar measurements, specified as true or false. Set this property to true to report false alarms. Otherwise, only actual detections are reported.

Data Types: logical

HasOcclusion — Enable line-of-sight occlusion
true (default) | false

Enable line-of-sight occlusion, specified as true or false. To generate detections only from objects for which the radar has a direct line of sight, set this property to true. For example, with this property enabled, the radar does not generate a detection for a vehicle that is behind another vehicle and blocked from view.

Data Types: logical

MaxNumDetectionsSource — Source of maximum number of detections reported
'Auto' (default) | 'Property'

Source of maximum number of detections reported by the sensor, specified as 'Auto' or 'Property'. When this property is set to 'Auto', the sensor reports all detections. When this property is set to 'Property', the sensor reports no more than the number of detections specified by the MaxNumDetections property.

Data Types: char | string

MaxNumDetections — Maximum number of reported detections
50 (default) | positive integer

Maximum number of detections reported by the sensor, specified as a positive integer. Detections are reported in order of distance to the sensor until the maximum number is reached.

Dependencies

To enable this property, set the MaxNumDetectionsSource property to 'Property'.

Data Types: double

DetectionCoordinates — Coordinate system of reported detections
'Ego Cartesian' (default) | 'Sensor Cartesian' | 'Sensor Spherical'

Coordinate system of reported detections, specified as one of these values:

- 'Ego Cartesian' — Detections are reported in the ego vehicle Cartesian coordinate system.
- 'Sensor Cartesian' — Detections are reported in the sensor Cartesian coordinate system.
- 'Sensor Spherical' — Detections are reported in a spherical coordinate system. This coordinate system is centered at the radar and aligned with the orientation of the radar on the ego vehicle.

Data Types: char | string

ActorProfiles — Actor profiles
structure | array of structures

Actor profiles, specified as structure or as an array of structures. Each structure contains the physical and radar characteristics of an actor.

- If `ActorProfiles` is a single structure, all actors passed into the `radarDetectionGenerator` object use this profile.
- If `ActorProfiles` is an array, each actor passed into the object must have a unique actor profile.

To generate an array of structures for your driving scenario, use the `actorProfiles` function. The table shows the valid structure fields. If you do not specify a field, the fields are set to their default values. If no actors are passed into the object, then the `ActorID` field is not included.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`ClassID`	Classification identifier, specified as a nonnegative integer. `0` is reserved for an object of an unknown or unassigned class.
`Length`	Length of actor, specified as a positive real scalar. The default is `4.7`. Units are in meters.
`Width`	Width of actor, specified as a positive real scalar. The default is `1.8`. Units are in meters.
`Height`	Height of actor, specified as a positive real scalar. The default is `1.4`. Units are in meters.
`OriginOffset`	Offset of actor's rotational center from its geometric center, specified as an [*x*,*y*, *z*] real-valued vector. The rotational center, or origin, is located at the bottom center of the actor. For vehicles, the rotational center is the point on the ground beneath the center of the rear axle. The default is `[0 0 0]`. Units are in meters.
`RCSPattern`	Radar cross-section pattern of actor, specified as a `numel(RCSElevationAngles)`-by-`numel(RCSAzimuthAngles)` real-valued matrix. The default is `[10 10; 10 10]`. Units are in decibels per square meter.
`RCSAzimuthAngles`	Azimuth angles corresponding to rows of `RCSPattern`, specified as a vector of real values in the range [–180, 180]. The default is `[-180 180]`. Units are in degrees.
`RCSElevationAngles`	Elevation angles corresponding to rows of `RCSPattern`, specified as a vector of real values in the range [–90, 90]. The default is `[-90 90]`. Units are in degrees.

For full definitions of the structure fields, see the `actor` and `vehicle` functions.

Usage

Syntax

```
dets = sensor(actors,time)
[dets,numValidDets] = sensor(actors,time)
[dets,numValidDets,isValidTime] = sensor(actors,time)
```

Description

`dets = sensor(actors,time)` creates radar detections, `dets`, from sensor measurements taken of `actors` at the current simulation `time`. The object can generate sensor detections for multiple actors simultaneously. Do not include the ego vehicle as one of the actors.

`[dets,numValidDets] = sensor(actors,time)` also returns the number of valid detections reported, `numValidDets`.

`[dets,numValidDets,isValidTime] = sensor(actors,time)` also returns a logical value, `isValidTime`, indicating that the `UpdateInterval` time has elapsed.

Input Arguments

`actors` — Scenario actor poses
structure | structure array

Scenario actor poses, specified as a structure or structure array. Each structure corresponds to an actor. You can generate these structures using the `actorPoses` function. You can also create these structures manually.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
`Velocity`	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (*ω*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

For full definitions of the structure fields, see the `actor` and `vehicle` functions.

`time` — Current simulation time
nonnegative real scalar

Current simulation time, specified as a nonnegative real scalar. The `drivingScenario` object calls the radar detection generator at regular time intervals. The radar detector generates new detections at intervals defined by the `UpdateInterval` property. The value of the `UpdateInterval` property must be an integer multiple of the simulation time interval. Updates requested from the sensor between update intervals contain no detections. Units are in seconds.

Example: `10.5`

Data Types: `double`

Output Arguments

`dets` — Radar sensor detections

cell array of `objectDetection` objects

Radar sensor detections, returned as a cell array of `objectDetection` objects. Each object contains these fields:

Property	Definition
`Time`	Measurement time
`Measurement`	Object measurements
`MeasurementNoise`	Measurement noise covariance matrix
`SensorIndex`	Unique ID of the sensor
`ObjectClassID`	Object classification
`ObjectAttributes`	Additional information passed to tracker
`MeasurementParameters`	Parameters used by initialization functions of nonlinear Kalman tracking filters

For Cartesian coordinates, `Measurement`, `MeasurementNoise`, and `MeasurementParameters` are reported in the coordinate system specified by the `DetectionCoordinates` property of the `radarDetectionGenerator`.

For spherical coordinates, `Measurement` and `MeasurementNoise` are reported in the spherical coordinate system based on the sensor Cartesian coordinate system. `MeasurementParameters` are reported in sensor Cartesian coordinates.

Measurement

<table>
<tr><th>DetectionCoordinates Property</th><th>Measurement and Measurement Noise Coordinates</th></tr>
<tr><td>'Ego Cartesian'</td><td rowspan="2">Coordinate Dependence on HasRangeRate
<table>
<tr><th>HasRangeRate</th><th>Coordinates</th></tr>
<tr><td>true</td><td>[x;y;z;vx;vy;vz]</td></tr>
<tr><td>false</td><td>[x;y;z]</td></tr>
</table></td></tr>
<tr><td>'Sensor Cartesian'</td></tr>
<tr><td>'Sensor Spherical'</td><td>Coordinate Dependence on HasRangeRate and HasElevation
<table>
<tr><th>HasRangeRate</th><th>HasElevation</th><th>Coordinates</th></tr>
<tr><td>true</td><td>true</td><td>[az;el;rng;rr]</td></tr>
<tr><td>true</td><td>false</td><td>[az;rng;rr]</td></tr>
<tr><td>false</td><td>true</td><td>[az;el;rng]</td></tr>
<tr><td>false</td><td>false</td><td>[az;rng]</td></tr>
</table></td></tr>
</table>

MeasurementParameters

Parameter	Definition
`Frame`	Enumerated type indicating the frame used to report measurements. When `Frame` is set to `'rectangular'`, detections are reported in Cartesian coordinates. When `Frame` is set `'spherical'`, detections are reported in spherical coordinates.
`OriginPosition`	3-D vector offset of the sensor origin from the ego vehicle origin. The vector is derived from the `SensorLocation` and `Height` properties specified in the `radarDetectionGenerator`.
`Orientation`	Orientation of the vision sensor coordinate system with respect to the ego vehicle coordinate system. The orientation is derived from the `Yaw`, `Pitch`, and `Roll` properties of the `radarDetectionGenerator`.
`HasVelocity`	Indicates whether measurements contain velocity or range rate components.
`HasElevation`	Indicates whether measurements contain elevation components.

ObjectAttributes

Attribute	Definition
TargetIndex	Identifier of the actor, ActorID, that generated the detection. For false alarms, this value is negative.
SNR	Detection signal-to-noise ratio in dB.

numValidDets — Number of detections
nonnegative integer

Number of detections, returned as a nonnegative integer.

- When the MaxNumDetectionsSource property is set to 'Auto', numValidDets is set to the length of dets.
- When the MaxNumDetectionsSource property is set to 'Property', dets is a cell array with length determined by the MaxNumDetections property. No more than MaxNumDetections number of detections are returned. If the number of detections is fewer than MaxNumDetections, the first numValidDets elements of dets hold valid detections. The remaining elements of dets are set to the default value.

Data Types: double

isValidTime — Valid detection time
0 | 1

Valid detection time, returned as 0 or 1. isValidTime is 0 when detection updates are requested at times that are between update intervals specified by UpdateInterval.

Data Types: logical

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named obj, use this syntax:

```
release(obj)
```

Specific to radarDetectionGenerator

isLocked	Determine if System object is in use

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Generate Radar Detections of Multiple Vehicles

Generate detections using a forward-facing automotive radar mounted on an ego vehicle. Assume that there are three targets:

- Vehicle 1 is in the center lane, directly in front of the ego vehicle, and driving at the same speed.
- Vehicle 2 is in the left lane and driving faster than the ego vehicle by 12 kilometers per hour.
- Vehicle 3 is in the right lane and driving slower than the ego vehicle by 5 kilometers per hour.

All positions, velocities, and measurements are relative to the ego vehicle. Run the simulation for ten steps.

```
dt = 0.1;
pos1 = [150 0 0];
pos2 = [160 10 0];
pos3 = [130 -10 0];
vel1 = [0 0 0];
vel2 = [12*1000/3600 0 0];
vel3 = [-5*1000/3600 0 0];
car1 = struct('ActorID',1,'Position',pos1,'Velocity',vel1);
car2 = struct('ActorID',2,'Position',pos2,'Velocity',vel2);
car3 = struct('ActorID',3,'Position',pos3,'Velocity',vel3);
```

Create an automotive radar sensor that is offset from the ego vehicle. By default, the sensor location is at (3.4,0) meters from the vehicle center and 0.2 meters above the ground plane. Turn off the range rate computation so that the radar sensor measures position only.

```
radar = radarDetectionGenerator('DetectionCoordinates','Sensor Cartesian', ...
    'MaxRange',200,'RangeResolution',10,'AzimuthResolution',10, ...
    'FieldOfView',[40 15],'UpdateInterval',dt,'HasRangeRate',false);
tracker = multiObjectTracker('FilterInitializationFcn',@initcvkf, ...
    'ConfirmationThreshold',[3 4],'DeletionThreshold',[6 6]);
```

Generate detections with the radar from the non-ego vehicles. The output detections form a cell array and can be passed directly in to the `multiObjectTracker`.

```
simTime = 0;
nsteps = 10;
for k = 1:nsteps
    dets = radar([car1 car2 car3],simTime);
    [confirmedTracks,tentativeTracks,allTracks] = updateTracks(tracker,dets,simTime);
```

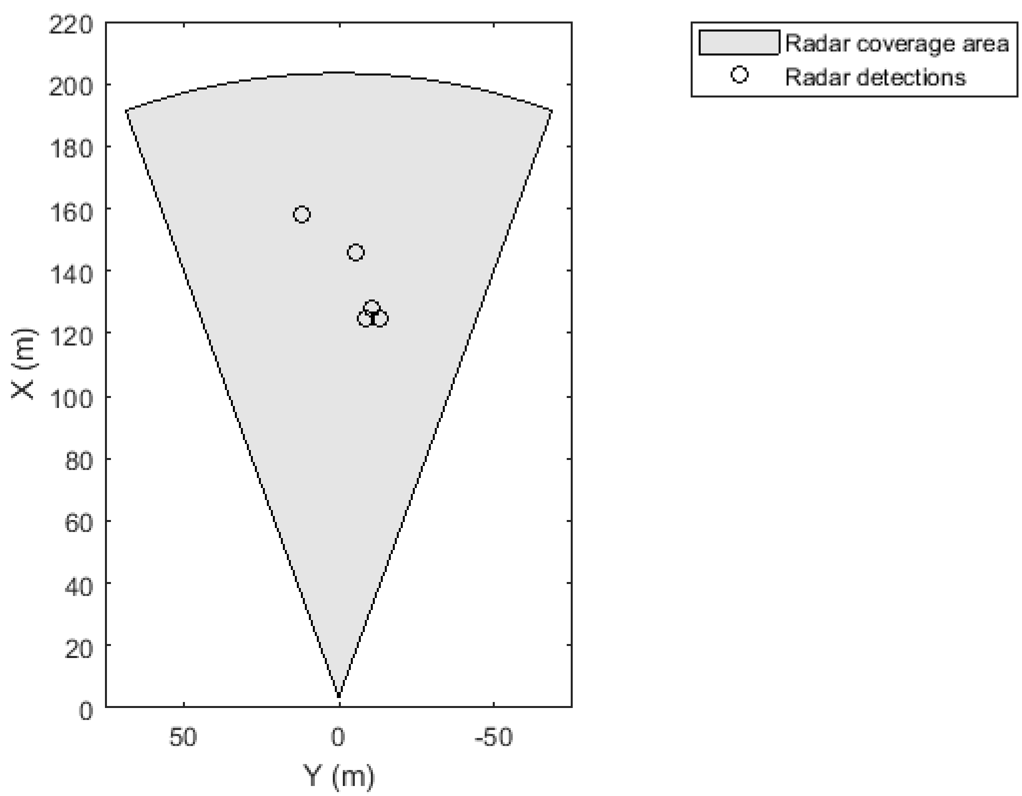
Move the cars one time step and update the multi-object tracker.

```
    simTime = simTime + dt;
    car1.Position = car1.Position + dt*car1.Velocity;
    car2.Position = car2.Position + dt*car2.Velocity;
    car3.Position = car3.Position + dt*car3.Velocity;
end
```

Use `birdsEyePlot` to create an overhead view of the detections. Plot the sensor coverage area. Extract the *X* and *Y* positions of the targets by converting the `Measurement` fields of the cell array into a MATLAB array. Display the detections on the bird's-eye plot.

```
BEplot = birdsEyePlot('XLim',[0 220],'YLim',[-75 75]);
caPlotter = coverageAreaPlotter(BEplot,'DisplayName','Radar coverage area');
plotCoverageArea(caPlotter,radar.SensorLocation,radar.MaxRange, ...
```

```
    radar.Yaw,radar.FieldOfView(1))
detPlotter = detectionPlotter(BEplot,'DisplayName','Radar detections');
detPos = cellfun(@(d)d.Measurement(1:2),dets,'UniformOutput',false);
detPos = cell2mat(detPos')';
if ~isempty(detPos)
    plotDetection(detPlotter,detPos)
end
```



Generate Radar Detections of Occluded Targets

Model the effects of occlusion when generating radar detections from a `radarDetectionGenerator` System object™.

Create two cars. Position the first car 40 meters away from the sensor. Position the second car 10 meters directly behind the first car.

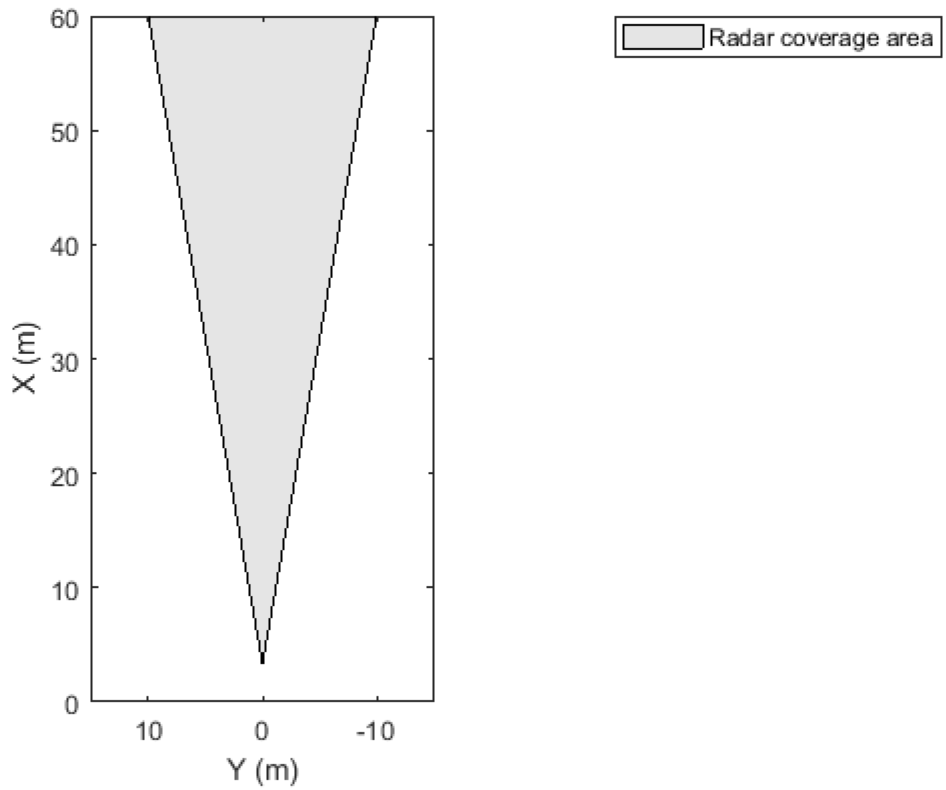
```
car1 = struct('ActorID',1,'Position',[40 0 0]);
car2 = struct('ActorID',2,'Position',[50 0 0]);
```

Create a radar detection generator System object, `radarSensor`, with default values. Use the System object to generate detections.

```
radarSensor = radarDetectionGenerator;
simTime = 0; % start of simulation
[dets,numValidDets] = radarSensor([car1 car2],simTime);
```

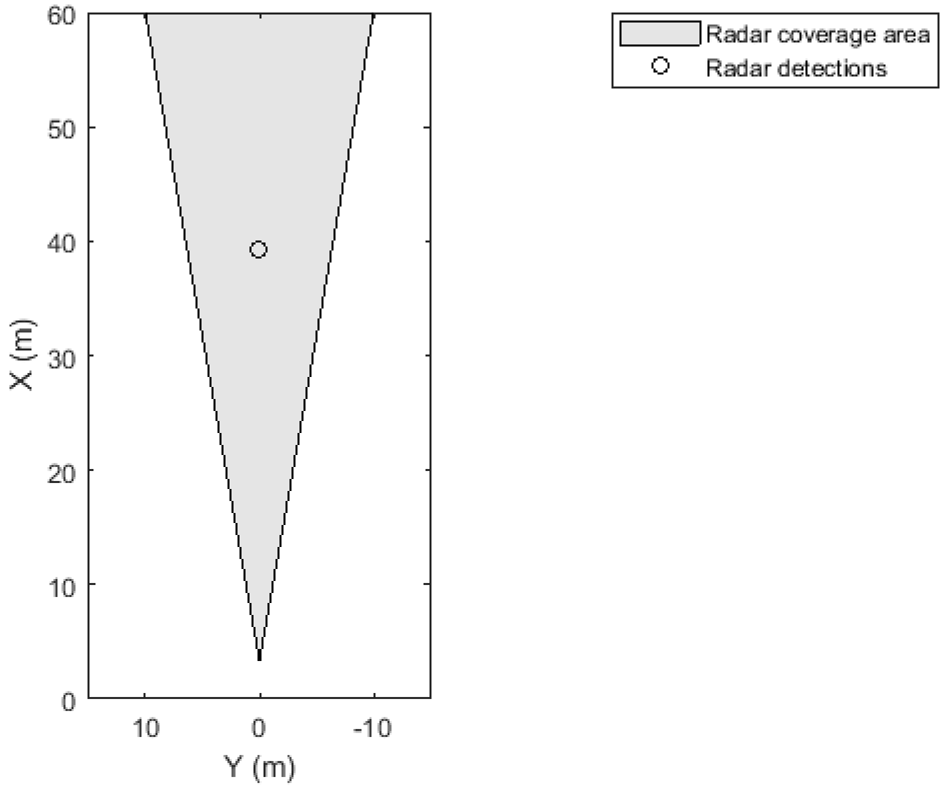
Display the coverage area of the radar detection generator on a bird's-eye plot.

```
bep = birdsEyePlot('XLim',[0 60],'YLim',[-15 15]);
caPlotter = coverageAreaPlotter(bep,'DisplayName', ...
    'Radar coverage area');
plotCoverageArea(caPlotter,radarSensor.SensorLocation, ...
    radarSensor.MaxRange,radarSensor.Yaw, ...
    radarSensor.FieldOfView(1));
```



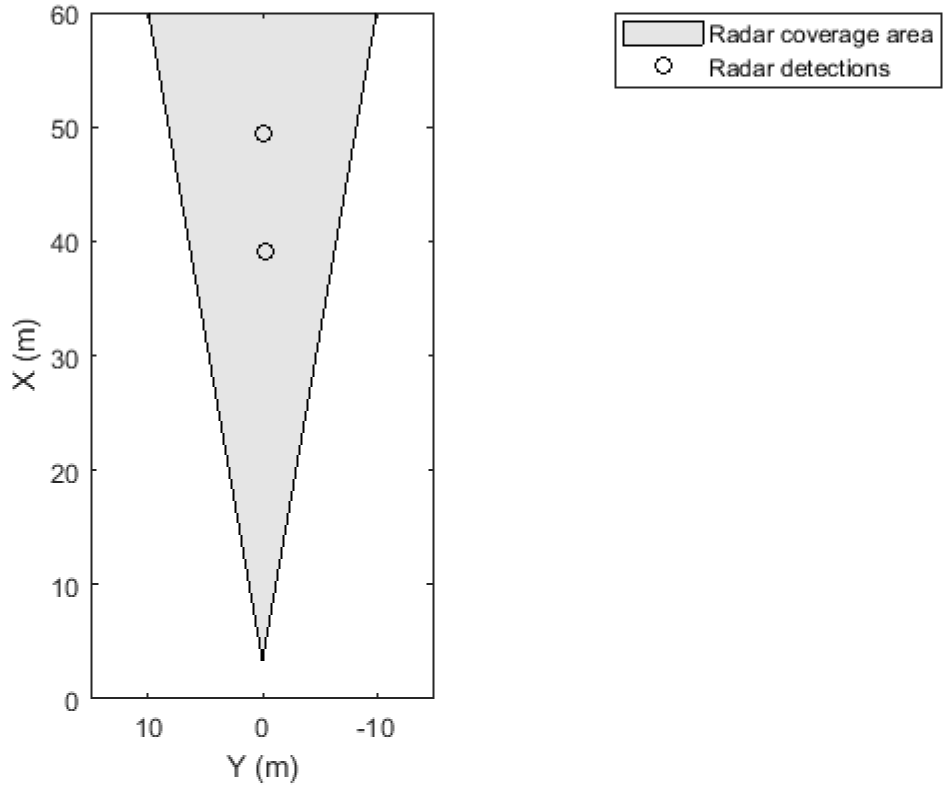
Extract the (*X*,*Y*) positions of the targets by converting the (*X*,*Y*) values of the `Measurement` field of the cell array into a MATLAB array. Then, display the detections.

```
if numValidDets > 0
    detPlotter = detectionPlotter(bep,'DisplayName','Radar detections');
    detPos = cellfun(@(d)d.Measurement(1:2),dets,'UniformOutput',false);
    detPos = cell2mat(detPos')';
    plotDetection(detPlotter,detPos)
end
```



By default, the radar detection generator excludes targets that are occluded by other objects. Therefore, the radar detects the nearest target but not the target directly behind it. To include the occluded target in the detections, release the radar detection generator, disable line-of-sight occlusion, and generate detections again. Display the detections.

```
release(radarSensor)
radarSensor.HasOcclusion = false;
[detsNoOcclusion,numValidDets] = radarSensor([car1 car2],simTime);
if numValidDets > 0
    detPos = cellfun(@(d)d.Measurement(1:2),detsNoOcclusion,'UniformOutput',false);
    detPos = cell2mat(detPos')';
    plotDetection(detPlotter, detPos)
end
```



Release the radar detection generator.

```
release(radarSensor)
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`drivingScenario` | `lidarPointCloudGenerator` | `multiObjectTracker` | `objectDetection` | `visionDetectionGenerator`

Functions

`actorPoses` | `actorProfiles`

Apps

Driving Scenario Designer

Topics
"Model Radar Sensor Detections"
"Track-Level Fusion of Radar and Lidar Data"
"Coordinate Systems in Automated Driving Toolbox"

Introduced in R2017a

visionDetectionGenerator

Generate vision detections for driving scenario

Description

The `visionDetectionGenerator` System object generates detections from a monocular camera sensor mounted on an ego vehicle. All detections are referenced to the coordinate system of the ego vehicle or the vehicle-mounted sensor. You can use the `visionDetectionGenerator` object in a scenario containing actors and trajectories, which you can create by using a `drivingScenario` object. Using a statistical mode, the generator can simulate real detections with added random noise and also generate false alarm detections. In addition, you can use the `visionDetectionGenerator` object to create input to a `multiObjectTracker`. When building scenarios using the **Driving Scenario Designer** app, the camera sensors mounted on the ego vehicle are output as `visionDetectionGenerator` objects.

To generate visual detections:

1. Create the `visionDetectionGenerator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
sensor = visionDetectionGenerator
sensor = visionDetectionGenerator(cameraConfig)
sensor = visionDetectionGenerator(Name,Value)
```

Description

`sensor = visionDetectionGenerator` creates a vision detection generator object with default property values.

`sensor = visionDetectionGenerator(cameraConfig)` creates a vision detection generator object using the `monoCamera` configuration object, `cameraConfig`.

`sensor = visionDetectionGenerator(Name,Value)` sets properties on page 4-192 using one or more name-value pairs. For example, `visionDetectionGenerator('DetectionCoordinates','Sensor Cartesian','MaxRange',200)` creates a vision detection generator that reports detections in the sensor Cartesian coordinate system and has a maximum detection range of 200 meters. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`DetectorOutput` — Types of detections generated by sensor
`'Objects only'` (default) | `'Lanes only'` | `'Lanes with occlusion'` | `'Lanes and objects'`

Types of detections generated by the sensor, specified as `'Objects only'`, `'Lanes only'`, `'Lanes with occlusion'`, or `'Lanes and objects'`.

- When set to `'Objects only'`, only actors are detected.
- When set to `'Lanes only'`, only lanes are detected.
- When set to `'Lanes with occlusion'`, only lanes are detected but actors in the camera field of view can impair the sensor ability to detect lanes.
- When set to `'Lanes and objects'`, the sensor generates both object detections and occluded lane detections.

Example: `'Lanes with occlusion'`

Data Types: `char` | `string`

`SensorIndex` — Unique sensor identifier
positive integer

Unique sensor identifier, specified as a positive integer. This property distinguishes detections that come from different sensors in a multi-sensor system.

Example: `5`

Data Types: `double`

`UpdateInterval` — Required time interval between sensor updates
`0.1` | positive real scalar

Required time interval between sensor updates, specified as a positive real scalar. The `drivingScenario` object calls the vision detection generator at regular time intervals. The vision detector generates new detections at intervals defined by the `UpdateInterval` property. The value of the `UpdateInterval` property must be an integer multiple of the simulation time interval. Updates requested from the sensor between update intervals contain no detections. Units are in seconds.

Example: `5`

Data Types: `double`

`SensorLocation` — Sensor location
`[3.4 0]` | `[x y]` vector

Location of the vision sensor center, specified as an `[x y]`. The `SensorLocation` and `Height` properties define the coordinates of the vision sensor with respect to the ego vehicle coordinate system. The default value corresponds to a forward-facing sensor mounted on a vehicle dashboard. Units are in meters.

Example: `[4 0.1]`

Data Types: `double`

`Height` — Sensor height above ground plane
`1.1` | positive real scalar

Sensor height above the vehicle ground plane, specified as a positive real scalar. The default value corresponds to a forward-facing vision sensor mounted on the dashboard of a sedan. Units are in meters.

Example: `1.5`

Data Types: `double`

`Yaw` — Yaw angle of vision sensor
`0` | real scalar

Yaw angle of vision sensor, specified as a real scalar. The yaw angle is the angle between the center line of the ego vehicle and the down-range axis of the vision sensor. A positive yaw angle corresponds to a clockwise rotation when looking in the positive direction of the *z*-axis of the ego vehicle coordinate system. Units are in degrees.

Example: `-4`

Data Types: `double`

`Pitch` — Pitch angle of vision sensor
`0` | real scalar

Pitch angle of vision sensor, specified as a real scalar. The pitch angle is the angle between the down-range axis of the vision sensor and the *x*-*y* plane of the ego vehicle coordinate system. A positive pitch angle corresponds to a clockwise rotation when looking in the positive direction of the *y*-axis of the ego vehicle coordinate system. Units are in degrees.

Example: `3`

Data Types: `double`

`Roll` — Roll angle of vision sensor
`0` | real scalar

Roll angle of the vision sensor, specified as a real scalar. The roll angle is the angle of rotation of the down-range axis of the vision sensor around the *x*-axis of the ego vehicle coordinate system. A positive roll angle corresponds to a clockwise rotation when looking in the positive direction of the *x*-axis of the coordinate system. Units are in degrees.

Example: `-4`

Data Types: `double`

`Intrinsics` — Intrinsic calibration parameters of vision sensor
`cameraIntrinsics([800 800],[320 240],[480 640])` (default) | `cameraIntrinsics` object

Intrinsic calibration parameters of vision sensor, specified as a `cameraIntrinsics` object.

`FieldOfView` — Angular field of view of vision sensor
real-valued 1-by-2 vector of positive values

This property is read-only.

Angular field of view of vision sensor, specified as a real-valued 1-by-2 vector of positive values, `[azfov,elfov]`. The field of view defines the azimuth and elevation extents of the sensor image. Each component must lie in the interval from 0 degrees to 180 degrees. The field of view is derived from the intrinsic parameters of the vision sensor. Targets outside of the angular field of view of the sensor are not detected. Units are in degrees.

Data Types: `double`

`MaxRange` — Maximum detection range
`150` | positive real scalar

Maximum detection range, specified as a positive real scalar. The sensor cannot detect a target beyond this range. Units are in meters.

Example: `200`

Data Types: `double`

`MaxSpeed` — Maximum detectable object speed
`100` (default) | nonnegative real scalar

Maximum detectable object speed, specified as a nonnegative real scalar. Units are in meters per second.

Example: `10.0`

Data Types: `double`

`MaxAllowedOcclusion` — Maximum allowed occlusion of an object
`0.5` (default) | real scalar in the range (0 1]

Maximum allowed occlusion of an object, specified as a real scalar in the range [0 1]. Occlusion is the fraction of the total surface area of an object not visible to the sensor. A value of one indicates that the object is fully occluded. Units are dimensionless.

Example: `0.2`

Data Types: `double`

`DetectionProbability` — Probability of detection
`0.9` (default) | positive real scalar less than or equal to 1

Probability of detecting a target, specified as a positive real scalar less than or equal to 1. This quantity defines the probability that the sensor detects a detectable object. A detectable object is an object that satisfies the minimum detectable size, maximum range, maximum speed, and maximum allowed occlusion constraints.

Example: `0.95`

Data Types: `double`

`FalsePositivesPerImage` — Number of false detections per image
`0.1` (default) | nonnegative real scalar

Number of false detections that the vision sensor generates for each image, specified as a nonnegative real scalar.

Example: 2

Data Types: double

MinObjectImageSize — Minimum image size of detectable object
[15 15] (default) | 1-by-2 vector of positive values

Minimum height and width of an object that the vision sensor detects within an image, specified as a [minHeight,minWidth] vector of positive values. The 2-D projected height of an object must be greater than or equal to minHeight. The projected width of an object must be greater than or equal to minWidth. Units are in pixels.

Example: [30 20]

Data Types: double

BoundingBoxAccuracy — Bounding box accuracy
5 (default) | positive real scalar

Bounding box accuracy, specified as a positive real scalar. This quantity defines the accuracy with which the detector can match a bounding box to a target. Units are in pixels.

Example: 4

Data Types: double

ProcessNoiseIntensity — Noise intensity used for filtering position and velocity measurements
5 (default) | positive real scalar

Noise intensity used for filtering position and velocity measurements, specified as a positive real scalar. Noise intensity defines the standard deviation of the process noise of the internal constant-velocity Kalman filter used in a vision sensor. The filter models the process noise using a piecewise-constant white noise acceleration model. Noise intensity is typically of the order of the maximum acceleration magnitude expected for a target. Units are in m/s^2.

Example: 2.5

Data Types: double

HasNoise — Enable adding noise to vision sensor measurements
true (default) | false

Enable adding noise to vision sensor measurements, specified as true or false. Set this property to true to add noise to the sensor measurements. Otherwise, the measurements have no noise. Even if you set HasNoise to false, the object still computes the MeasurementNoise property of each detection.

Data Types: logical

MaxNumDetectionsSource — Source of maximum number of detections reported
'Auto' (default) | 'Property'

Source of maximum number of detections reported by the sensor, specified as 'Auto' or 'Property'. When this property is set to 'Auto', the sensor reports all detections. When this property is set to `'Property'`, the sensor reports no more than the number of detections specified by the `MaxNumDetections` property.

Data Types: `char` | `string`

`MaxNumDetections` — Maximum number of reported detections
`50` (default) | positive integer

Maximum number of detections reported by the sensor, specified as a positive integer. The detections closest to the sensor are reported.

Dependencies

To enable this property, set the `MaxNumDetectionsSource` property to `'Property'`.

Data Types: `double`

`DetectionCoordinates` — Coordinate system of reported detections
`'Ego Cartesian'` (default) | `'Sensor Cartesian'`

Coordinate system of reported detections, specified as one of these values:

- `'Ego Cartesian'` — Detections are reported in the ego vehicle Cartesian coordinate system.
- `'Sensor Cartesian'` — Detections are reported in the sensor Cartesian coordinate system.

Data Types: `char` | `string`

`LaneUpdateInterval` — Required time interval between lane detection updates
`0.1` (default) | positive real scalar

Required time interval between lane detection updates, specified as a positive real scalar. The `drivingScenario` object calls the vision detection generator at regular time intervals. The vision detector generates new lane detections at intervals defined by this property which must be an integer multiple of the simulation time interval. Updates requested from the sensor between update intervals contain no lane detections. Units are in seconds.

Example: `0.4`

Data Types: `double`

`MinLaneImageSize` — Minimum lane size in image
`[20 5]` (default) | 1-by-2 real-valued vector

Minimum size of a projected lane marking that can be detected by the sensor after accounting for curvature, specified as a 1-by-2 real-valued vector, `[minHeight minWidth]`. Lane markings must exceed both of these values to be detected. This property is used only when detecting lanes. Units are in pixels.

Example: `[5,7]`

Data Types: `double`

`LaneBoundaryAccuracy` — Accuracy of lane boundaries
`3` | positive real scalar

Accuracy of lane boundaries, specified as a positive real scalar. This property defines the accuracy with which the lane sensor can place a lane boundary. Units are in pixels. This property is used only when detecting lanes.

MaxNumLanesSource — Source of maximum number of reported lanes
'Property' (default) | 'Auto'

Source of maximum number of reported lanes, specified as 'Auto' or 'Property'. When specified as 'Auto', the maximum number of lanes is computed automatically. When specified as 'Property', use the MaxNumLanes property to set the maximum number or lanes.

Data Types: char | string

MaxNumLanes — Maximum number of reported lanes
30 (default) | positive integer

Maximum number of reported lanes, specified as a positive integer.

Dependencies

To enable this property, set the MaxNumLanesSource property to 'Property'.

Data Types: char | string

ActorProfiles — Actor profiles
structure | array of structures

Actor profiles, specified as structure or as an array of structures. Each structure contains the physical and radar characteristics of an actor.

- If ActorProfiles is a single structure, all actors passed into the visionDetectionGenerator object use this profile.
- If ActorProfiles is an array, each actor passed into the object must have a unique actor profile.

To generate an array of structures for your driving scenario, use the actorProfiles function. The table shows the valid structure fields. If you do not specify a field, the fields are set to their default values. If no actors are passed into the object, then the ActorID field is not included.

Field	Description
ActorID	Scenario-defined actor identifier, specified as a positive integer.
ClassID	Classification identifier, specified as a nonnegative integer. 0 is reserved for an object of an unknown or unassigned class.
Length	Length of actor, specified as a positive real scalar. The default is 4.7. Units are in meters.
Width	Width of actor, specified as a positive real scalar. The default is 1.8. Units are in meters.
Height	Height of actor, specified as a positive real scalar. The default is 1.4. Units are in meters.

Field	Description
`OriginOffset`	Offset of actor's rotational center from its geometric center, specified as an [*x*,*y*, *z*] real-valued vector. The rotational center, or origin, is located at the bottom center of the actor. For vehicles, the rotational center is the point on the ground beneath the center of the rear axle. The default is `[0 0 0]`. Units are in meters.
`RCSPattern`	Radar cross-section pattern of actor, specified as a `numel(RCSElevationAngles)`-by-`numel(RCSAzimuthAngles)` real-valued matrix. The default is `[10 10; 10 10]`. Units are in decibels per square meter.
`RCSAzimuthAngles`	Azimuth angles corresponding to rows of `RCSPattern`, specified as a vector of real values in the range [–180, 180]. The default is `[-180 180]`. Units are in degrees.
`RCSElevationAngles`	Elevation angles corresponding to rows of `RCSPattern`, specified as a vector of real values in the range [–90, 90]. The default is `[-90 90]`. Units are in degrees.

For full definitions of the structure fields, see the `actor` and `vehicle` functions.

Usage

Syntax

```
dets = sensor(actors,time)
lanedets = sensor(laneboundaries,time)
lanedets = sensor(actors,laneboundaries,time)
[ ___ ,numValidDets] = sensor( ___ )
[ ___ ,numValidDetsisValidTime] = sensor( ___ )
[dets,numValidDets,isValidTime,lanedets,numValidLaneDets,isValidLaneTime] =
sensor(actors,laneboundaries,time)
```

Description

`dets = sensor(actors,time)` creates visual detections, `dets`, from sensor measurements taken of `actors` at the current simulation `time`. The object can generate sensor detections for multiple actors simultaneously. Do not include the ego vehicle as one of the actors.

To enable this syntax, set `DetectionOutput` to `'Objects only'`.

`lanedets = sensor(laneboundaries,time)` generates lane detections, `lanedets`, from lane boundary structures, `laneboundaries`.

To enable this syntax set `DetectionOutput` to `'Lanes only'`. The lane detector generates lane boundaries at intervals specified by the `LaneUpdateInterval` property.

`lanedets = sensor(actors,laneboundaries,time)` generates lane detections, `lanedets`, from lane boundary structures, `laneboundaries`.

To enable this syntax, set `DetectionOutput` to `'Lanes with occlusion'`. The lane detector generates lane boundaries at intervals specified by the `LaneUpdateInterval` property.

`[ ___ ,numValidDets] = sensor( ___ )` also returns the number of valid detections reported, `numValidDets`.

`[ ___ ,numValidDetsisValidTime] = sensor( ___ )` also returns a logical value, `isValidTime`, indicating that the `UpdateInterval` time to generate detections has elapsed.

`[dets,numValidDets,isValidTime,lanedets,numValidLaneDets,isValidLaneTime] = sensor(actors,laneboundaries,time)` returns both object detections, `dets`, and lane detections `lanedets`. This syntax also returns the number of valid lane detections reported, `numValidLaneDets`, and a flag, `isValidLaneTime`, indicating whether the required simulation time to generate lane detections has elapsed.

To enable this syntax, set `DetectionOutput` to `'Lanes and objects'`.

Input Arguments

`actors` — Scenario actor poses

structure | structure array

Scenario actor poses, specified as a structure or structure array. Each structure corresponds to an actor. You can generate this structure using the `actorPoses` function. You can also create these structures manually. The table shows the fields that the object uses to generate detections. All other fields are ignored.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [x, y, z]. Units are in meters.
`Velocity`	Velocity (v) of actor in the x-, y-, and z-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (ω) of actor in the x-, y-, and z-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

For full definitions of the structure fields, see the `actor` and `vehicle` functions.

Dependencies

To enable this argument, set the `DetectorOutput` property to `'Objects only'`, `'Lanes with occlusion'`, or `'Lanes and objects'`.

`laneboundaries` — Lane boundaries

array of lane boundary structures

Lane boundaries, specified as an array of lane boundary structures. The table shows the fields for each structure.

Field	Description
`Coordinates`	Lane boundary coordinates, specified as a real-valued *N*-by-3 matrix, where *N* is the number of lane boundary coordinates. Lane boundary coordinates define the position of points on the boundary at specified longitudinal distances away from the ego vehicle, along the center of the road. • In MATLAB, specify these distances by using the `'XDistance'` name-value pair argument of the `laneBoundaries` function. • In Simulink, specify these distances by using the **Distances from ego vehicle for computing boundaries (m)** parameter of the Scenario Reader block or the **Distance from parent for computing lane boundaries** parameter of the Simulation 3D Vision Detection Generator block. This matrix also includes the boundary coordinates at zero distance from the ego vehicle. These coordinates are to the left and right of the ego-vehicle origin, which is located under the center of the rear axle. Units are in meters.
`Curvature`	Lane boundary curvature at each row of the `Coordinates` matrix, specified as a real-valued *N*-by-1 vector. *N* is the number of lane boundary coordinates. Units are in radians per meter.
`CurvatureDerivative`	Derivative of lane boundary curvature at each row of the `Coordinates` matrix, specified as a real-valued *N*-by-1 vector. *N* is the number of lane boundary coordinates. Units are in radians per square meter.
`HeadingAngle`	Initial lane boundary heading angle, specified as a real scalar. The heading angle of the lane boundary is relative to the ego vehicle heading. Units are in degrees.

`LateralOffset`	Distance of the lane boundary from the ego vehicle position, specified as a real scalar. An offset to a lane boundary to the left of the ego vehicle is positive. An offset to the right of the ego vehicle is negative. Units are in meters.
`BoundaryType`	Type of lane boundary marking, specified as one of these values: • `'Unmarked'` — No physical lane marker exists • `'Solid'` — Single unbroken line • `'Dashed'` — Single line of dashed lane markers • `'DoubleSolid'` — Two unbroken lines • `'DoubleDashed'` — Two dashed lines • `'SolidDashed'` — Solid line on the left and a dashed line on the right • `'DashedSolid'` — Dashed line on the left and a solid line on the right
`Strength`	Saturation strength of the lane boundary marking, specified as a real scalar from 0 to 1. A value of `0` corresponds to a marking whose color is fully unsaturated. The marking is gray. A value of `1` corresponds to a marking whose color is fully saturated.
`Width`	Lane boundary width, specified as a positive real scalar. In a double-line lane marker, the same width is used for both lines and for the space between lines. Units are in meters.
`Length`	Length of dash in dashed lines, specified as a positive real scalar. In a double-line lane marker, the same length is used for both lines.
`Space`	Length of space between dashes in dashed lines, specified as a positive real scalar. In a dashed double-line lane marker, the same space is used for both lines.

Dependencies

To enable this argument, set the `DetectorOutput` property to `'Lanes only'`, `'Lanes with occlusion'`, or `'Lanes and objects'`.

Data Types: `struct`

`time` — Current simulation time
positive real scalar

Current simulation time, specified as a positive real scalar. The `drivingScenario` object calls the vision detection generator at regular time intervals. The vision detector generates new detections at intervals defined by the `UpdateInterval` property. The values of the `UpdateInterval` and

LanesUpdateInterval properties must be an integer multiple of the simulation time interval. Updates requested from the sensor between update intervals contain no detections. Units are in seconds.

Example: 10.5

Data Types: double

Output Arguments

dets — Object detections

cell array of objectDetection objects

Object detections, returned as a cell array of objectDetection objects. Each object contains these fields:

Property	Definition
Time	Measurement time
Measurement	Object measurements
MeasurementNoise	Measurement noise covariance matrix
SensorIndex	Unique ID of the sensor
ObjectClassID	Object classification
ObjectAttributes	Additional information passed to tracker
MeasurementParameters	Parameters used by initialization functions of nonlinear Kalman tracking filters

Measurement, MeasurementNoise, and MeasurementParameters are reported in the coordinate system specified by the DetectionCoordinates property of the visionDetectionGenerator.

Measurement

DetectionCoordinates Property	Measurement and Measurement Noise Coordinates
'Ego Cartesian'	[x;y;z;vx;vy;vz]
'Sensor Cartesian'	

MeasurementParameters

Parameter	Definition
Frame	Enumerated type indicating the frame used to report measurements. When Frame is set to 'rectangular', detections are reported in Cartesian coordinates. When Frame is set 'spherical', detections are reported in spherical coordinates.
OriginPosition	3-D vector offset of the sensor origin from the ego vehicle origin. The vector is derived from the SensorLocation and Height properties specified in the visionDetectionGenerator.
Orientation	Orientation of the vision sensor coordinate system with respect to the ego vehicle coordinate system. The orientation is derived from the Yaw, Pitch, and Roll properties of the visionDetectionGenerator.
HasVelocity	Indicates whether measurements contain velocity or range rate components.

ObjectAttributes

Attribute	Definition
TargetIndex	Identifier of the actor, ActorID, that generated the detection. For false alarms, this value is negative.

numValidDets — Number of detections
nonnegative integer

Number of detections returned, defined as a nonnegative integer.

- When the MaxNumDetectionsSource property is set to 'Auto', numValidDets is set to the length of dets.
- When the MaxNumDetectionsSource is set to 'Property', dets is a cell array with length determined by the MaxNumDetections property. No more than MaxNumDetections number of detections are returned. If the number of detections is fewer than MaxNumDetections, the first numValidDets elements of dets hold valid detections. The remaining elements of dets are set to the default value.

.

Data Types: double

isValidTime — Valid detection time
0 | 1

Valid detection time, returned as 0 or 1. isValidTime is 0 when detection updates are requested at times that are between update intervals specified by UpdateInterval.

Data Types: logical

lanedets — Lane boundary detections
lane boundary detection structure

Lane boundary detections, returned as an array structures. The fields of the structure are:

Lane Boundary Detection Structure

Field	Description
Time	Lane detection time
SensorIndex	Unique identifier of sensor
LaneBoundaries	Array of clothoidLaneBoundary objects.

numValidLaneDets — Number of detections
nonnegative integer

Number of lane detections returned, defined as a nonnegative integer.

- When the MaxNumLanesSource property is set to 'Auto', numValidLaneDets is set to the length of lanedets.
- When the MaxNumLanesSource is set to 'Property', lanedets is a cell array with length determined by the MaxNumLanes property. No more than MaxNumLanes number of lane detections are returned. If the number of detections is fewer than MaxNumLanes, the first numValidLaneDetections elements of lanedets hold valid lane detections. The remaining elements of lanedets are set to the default value.

.

Data Types: double

isValidLaneTime — Valid lane detection time
0 | 1

Valid lane detection time, returned as 0 or 1. isValidLaneTime is 0 when lane detection updates are requested at times that are between update intervals specified by LaneUpdateInterval.

Data Types: logical

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named obj, use this syntax:

```
release(obj)
```

Specific to visionDetectionGenerator

isLocked	Determine if System object is in use

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Generate Visual Detections of Multiple Vehicles

Generate detections using a forward-facing automotive vision sensor mounted on an ego vehicle. Assume that there are two target vehicles:

- Vehicle 1 is directly in front of the ego vehicle and moving at the same speed.
- Vehicle 2 vehicle is driving faster than the ego vehicle by 12 kph in the left lane.

All positions, velocities, and measurements are relative to the ego vehicle. Run the simulation for ten steps.

```
dt = 0.1;
car1 = struct('ActorID',1,'Position',[100 0 0],'Velocity', [5*1000/3600 0 0]);
car2 = struct('ActorID',2,'Position',[150 10 0],'Velocity',[12*1000/3600 0 0]);
```

Create an automotive vision sensor having a location offset from the ego vehicle. By default, the sensor location is at (3.4,0) meters from the vehicle center and 1.1 meters above the ground plane..

```
sensor = visionDetectionGenerator('DetectionProbability',1, ...
    'MinObjectImageSize',[5 5],'MaxRange',200,'DetectionCoordinates','Sensor Cartesian');
tracker = multiObjectTracker('FilterInitializationFcn',@initcvkf, ...
    'ConfirmationParameters',[3 4],'NumCoastingUpdates',6);
```

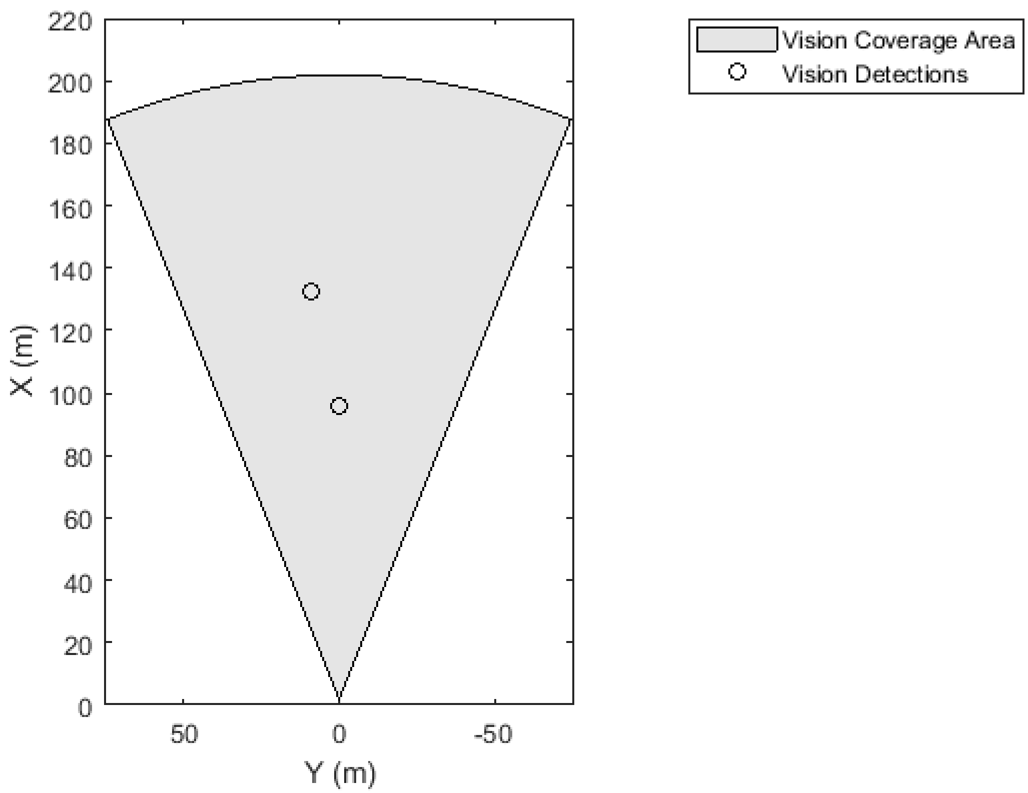
Generate visual detections for the non-ego actors as they move. The output detections form a cell array. Extract only position information from the detections to pass to the `multiObjectTracker`, which expects only position information. The Update the tracker for each new set of detections.

```
simTime = 0;
nsteps = 10;
for k = 1:nsteps
    dets = sensor([car1 car2],simTime);
    n = size(dets,1);
    for k = 1:n
        meas = dets{k}.Measurement(1:3);
        dets{k}.Measurement = meas;
        measmtx = dets{k}.MeasurementNoise(1:3,1:3);
        dets{k}.MeasurementNoise = measmtx;
    end
    [confirmedTracks,tentativeTracks,allTracks] = updateTracks(tracker,dets,simTime);
    simTime = simTime + dt;
    car1.Position = car1.Position + dt*car1.Velocity;
    car2.Position = car2.Position + dt*car2.Velocity;
end
```

Use `birdsEyePlot` to create an overhead view of the detections. Plot the sensor coverage area. Extract the *x* and *y* positions of the targets by converting the `Measurement` fields of the cell into a MATLAB® array. Then, plot the detections using `birdsEyePlot` functions.

```
BEplot = birdsEyePlot('XLim',[0 220],'YLim',[-75 75]);
caPlotter = coverageAreaPlotter(BEplot,'DisplayName','Vision Coverage Area');
plotCoverageArea(caPlotter,sensor.SensorLocation,sensor.MaxRange, ...
    sensor.Yaw,sensor.FieldOfView(1))
detPlotter = detectionPlotter(BEplot,'DisplayName','Vision Detections');
detPos = cellfun(@(d)d.Measurement(1:2),dets,'UniformOutput',false);
```

```
detPos = cell2mat(detPos')';
if ~isempty(detPos)
    plotDetection(detPlotter,detPos)
end
```



Generate Visual Detections from Monocular Camera

Create a vision sensor by using a monocular camera configuration, and generate detections from that sensor.

Specify the intrinsic parameters of the camera and create a `monoCamera` object from these parameters. The camera is mounted on top of an ego vehicle at a height of 1.5 meters above the ground and a pitch of 1 degree toward the ground.

```
focalLength = [800 800];
principalPoint = [320 240];
imageSize = [480 640];
intrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);

height = 1.5;
pitch = 1;
monoCamConfig = monoCamera(intrinsics,height,'Pitch',pitch);
```

Create a vision detection generator using the monocular camera configuration.

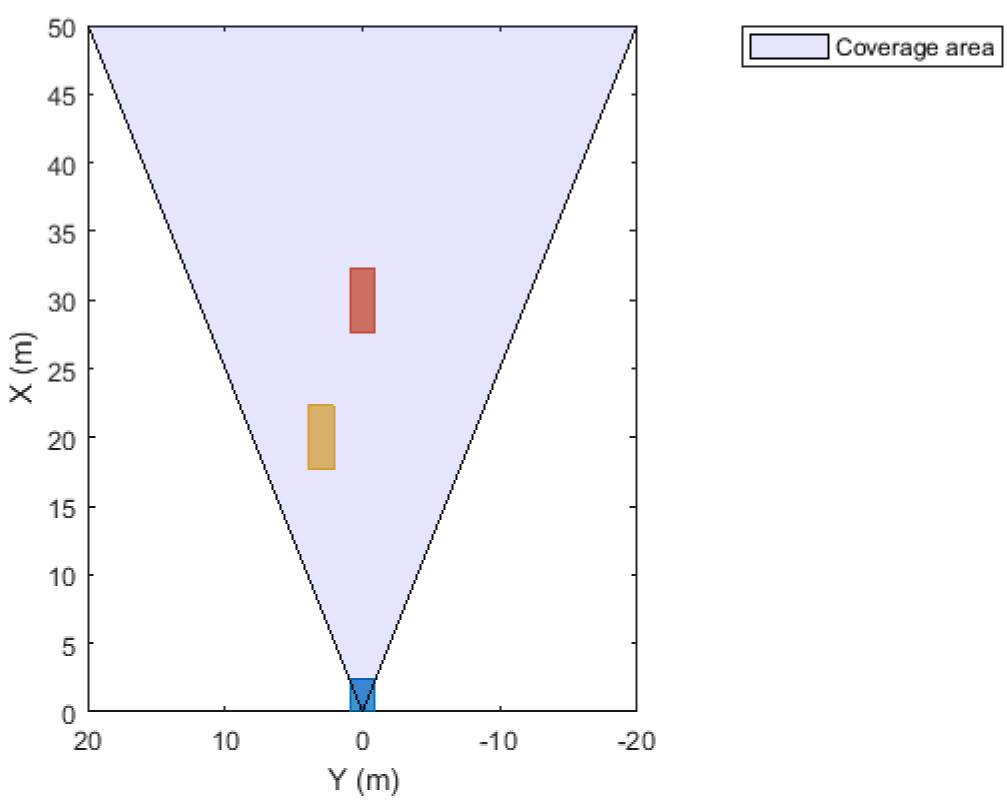
```
visionSensor = visionDetectionGenerator(monoCamConfig);
```

Generate a driving scenario with an ego vehicle and two target cars. Position the first target car 30 meters directly in front of the ego vehicle. Position the second target car 20 meters in front of the ego vehicle but offset to the left by 3 meters.

```
scenario = drivingScenario;
egoVehicle = vehicle(scenario,'ClassID',1);
targetCar1 = vehicle(scenario,'ClassID',1,'Position',[30 0 0]);
targetCar2 = vehicle(scenario,'ClassID',1,'Position',[20 3 0]);
```

Use a bird's-eye plot to display the vehicle outlines and sensor coverage area.

```
figure
bep = birdsEyePlot('XLim',[0 50],'YLim',[-20 20]);

olPlotter = outlinePlotter(bep);
[position,yaw,length,width,originOffset,color] = targetOutlines(egoVehicle);
plotOutline(olPlotter,position,yaw,length,width);

caPlotter = coverageAreaPlotter(bep,'DisplayName','Coverage area','FaceColor','blue');
plotCoverageArea(caPlotter,visionSensor.SensorLocation,visionSensor.MaxRange, ...
    visionSensor.Yaw,visionSensor.FieldOfView(1))
```



Obtain the poses of the target cars from the perspective of the ego vehicle. Use these poses to generate detections from the sensor.

```
poses = targetPoses(egoVehicle);
[dets,numValidDets] = visionSensor(poses,scenario.SimulationTime);
```

Display the (*X*,*Y*) positions of the valid detections. For each detection, the (*X*,*Y*) positions are the first two values of the `Measurement` field.

```
for i = 1:numValidDets
    XY = dets{i}.Measurement(1:2);
    detXY = sprintf('Detection %d: X = %.2f meters, Y = %.2f meters',i,XY);
    disp(detXY)
end
```

```
Detection 1: X = 19.09 meters, Y = 2.79 meters
Detection 2: X = 27.81 meters, Y = 0.08 meters
```

Generate Object and Lane Boundary Detections

Create a driving scenario containing an ego vehicle and a target vehicle traveling along a three-lane road. Detect the lane boundaries by using a vision detection generator.

```
scenario = drivingScenario;
```

Create a three-lane road by using lane specifications.

```
roadCenters = [0 0 0; 60 0 0; 120 30 0];
lspc = lanespec(3);
road(scenario,roadCenters,'Lanes',lspc);
```

Specify that the ego vehicle follows the center lane at 30 m/s.

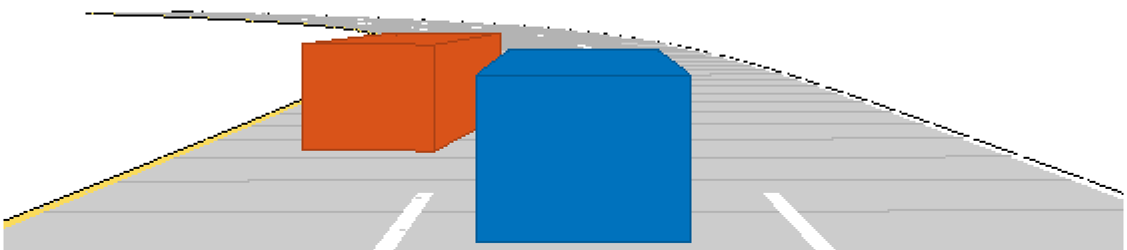
```
egovehicle = vehicle(scenario,'ClassID',1);
egopath = [1.5 0 0; 60 0 0; 111 25 0];
egospeed = 30;
trajectory(egovehicle,egopath,egospeed);
```

Specify that the target vehicle travels ahead of the ego vehicle at 40 m/s and changes lanes close to the ego vehicle.

```
targetcar = vehicle(scenario,'ClassID',1);
targetpath = [8 2; 60 -3.2; 120 33];
targetspeed = 40;
trajectory(targetcar,targetpath,targetspeed);
```

Display a chase plot for a 3-D view of the scenario from behind the ego vehicle.

```
chasePlot(egovehicle)
```



Create a vision detection generator that detects lanes and objects. The pitch of the sensor points one degree downward.

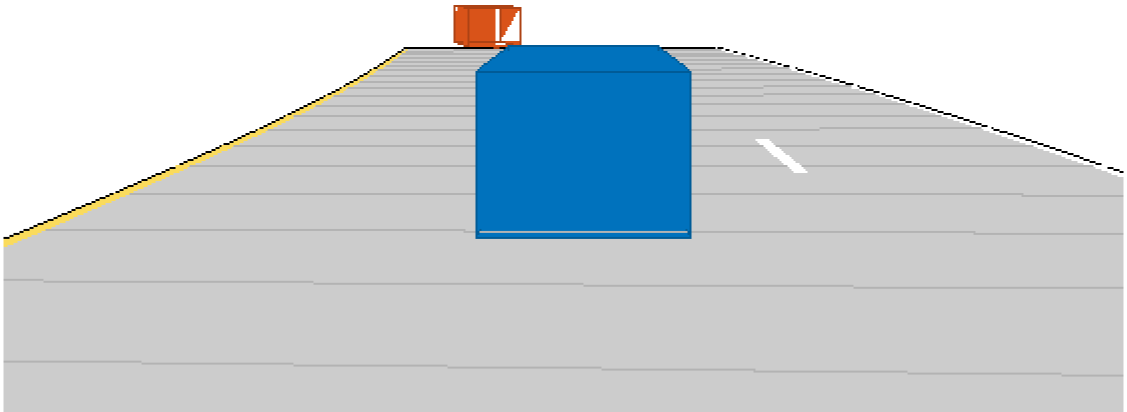
```
visionSensor = visionDetectionGenerator('Pitch',1.0);
visionSensor.DetectorOutput = 'Lanes and objects';
visionSensor.ActorProfiles = actorProfiles(scenario);
```

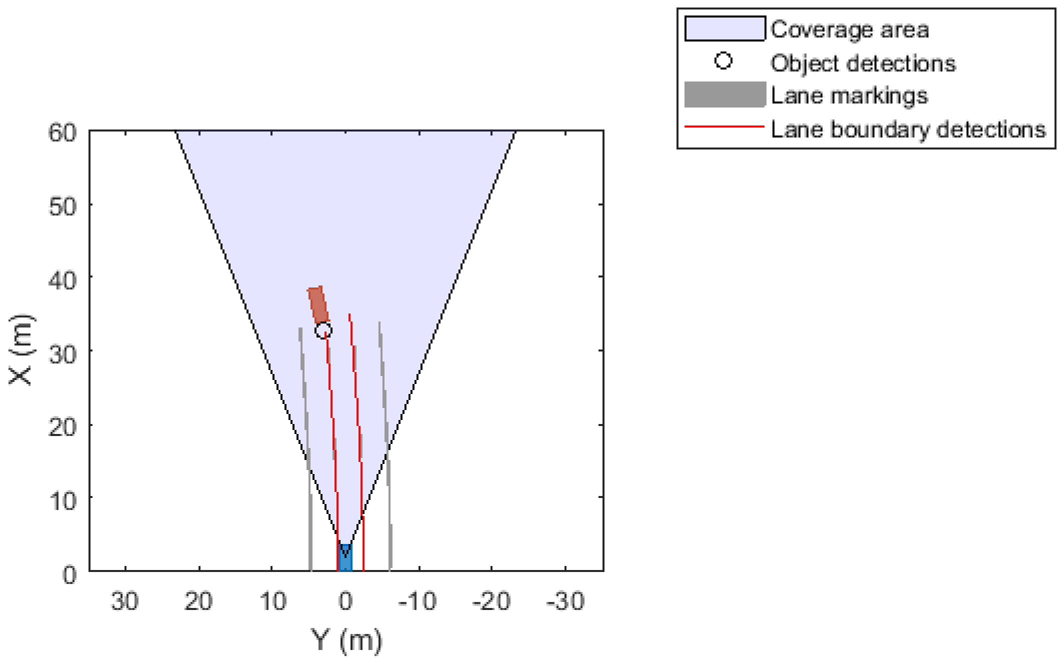
Run the simulation.

1. Create a bird's-eye plot and the associated plotters.
2. Display the sensor coverage area.
3. Display the lane markings.
4. Obtain ground truth poses of targets on the road.
5. Obtain ideal lane boundary points up to 60 m ahead.
6. Generate detections from the ideal target poses and lane boundaries.
7. Display the outline of the target.
8. Display object detections when the object detection is valid.
9. Display the lane boundary when the lane detection is valid.

```
bep = birdsEyePlot('XLim',[0 60],'YLim',[-35 35]);
caPlotter = coverageAreaPlotter(bep,'DisplayName','Coverage area', ...
    'FaceColor','blue');
detPlotter = detectionPlotter(bep,'DisplayName','Object detections');
lmPlotter = laneMarkingPlotter(bep,'DisplayName','Lane markings');
```

```
lbPlotter = laneBoundaryPlotter(bep,'DisplayName', ...
    'Lane boundary detections','Color','red');
olPlotter = outlinePlotter(bep);
plotCoverageArea(caPlotter,visionSensor.SensorLocation,...
    visionSensor.MaxRange,visionSensor.Yaw, ...
    visionSensor.FieldOfView(1));
while advance(scenario)
    [lmv,lmf] = laneMarkingVertices(egovehicle);
    plotLaneMarking(lmPlotter,lmv,lmf)
    tgtpose = targetPoses(egovehicle);
    lookaheadDistance = 0:0.5:60;
    lb = laneBoundaries(egovehicle,'XDistance',lookaheadDistance,'LocationType','inner');
    [obdets,nobdets,obValid,lb_dets,nlb_dets,lbValid] = ...
        visionSensor(tgtpose,lb,scenario.SimulationTime);
    [objposition,objyaw,objlength,objwidth,objoriginOffset,color] = targetOutlines(egovehicle);
    plotOutline(olPlotter,objposition,objyaw,objlength,objwidth, ...
        'OriginOffset',objoriginOffset,'Color',color)
    if obValid
        detPos = cellfun(@(d)d.Measurement(1:2),obdets,'UniformOutput',false);
        detPos = vertcat(zeros(0,2),cell2mat(detPos')');
        plotDetection(detPlotter,detPos)
    end
    if lbValid
        plotLaneBoundary(lbPlotter,vertcat(lb_dets.LaneBoundaries))
    end
end
```





Configure Ideal Vision Sensor

Generate detections from an ideal vision sensor and compare these detections to ones from a noisy sensor. An *ideal sensor* is one that always generates detections, with no false positives and no added random noise.

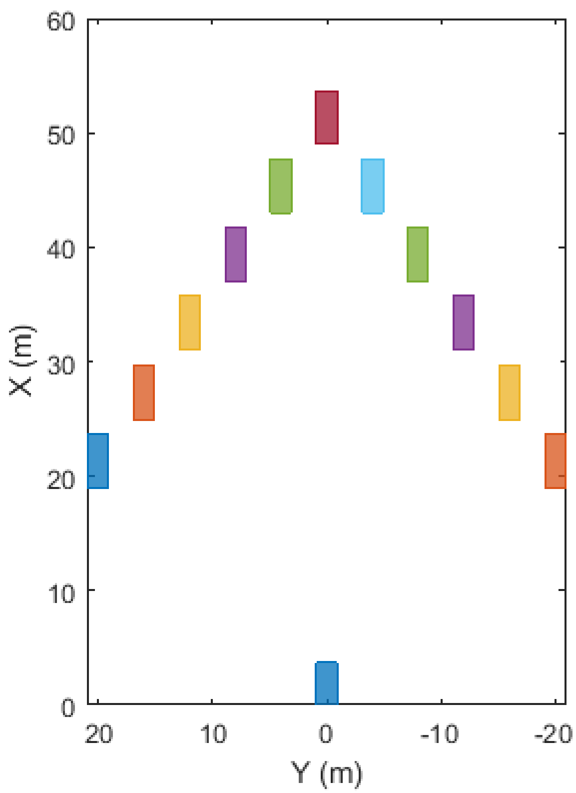
Create a Driving Scenario

Create a driving scenario in which the ego vehicle is positioned in front of a diagonal array of target cars. With this configuration, you can later plot the measurement noise covariances of the detected targets without having the target cars occlude one another.

```
scenario = drivingScenario;
egoVehicle = vehicle(scenario,'ClassID',1);

numTgts = 6;
x = linspace(20,50,numTgts)';
y = linspace(-20,0,numTgts)';
x = [x;x(1:end-1)];
y = [y;-y(1:end-1)];
numTgts = numel(x);

for m = 1:numTgts
    vehicle(scenario,'ClassID',1,'Position',[x(m) y(m) 0]);
end
```

Plot the driving scenario in a bird's-eye plot.

```
bep = birdsEyePlot('XLim',[0 60]);
legend('hide')

olPlotter = outlinePlotter(bep);
[position,yaw,length,width,originOffset,color] = targetOutlines(egoVehicle);
plotOutline(olPlotter,position,yaw,length,width, ...
    'OriginOffset',originOffset,'Color',color)
```



Create an Ideal Vision Sensor

Create a vision sensor by using the `visionDetectionGenerator` System object™. To generate ideal detections, set `DetectionProbability` to `1`, `FalsePositivesPerImage` to `0`, and `HasNoise` to `false`.

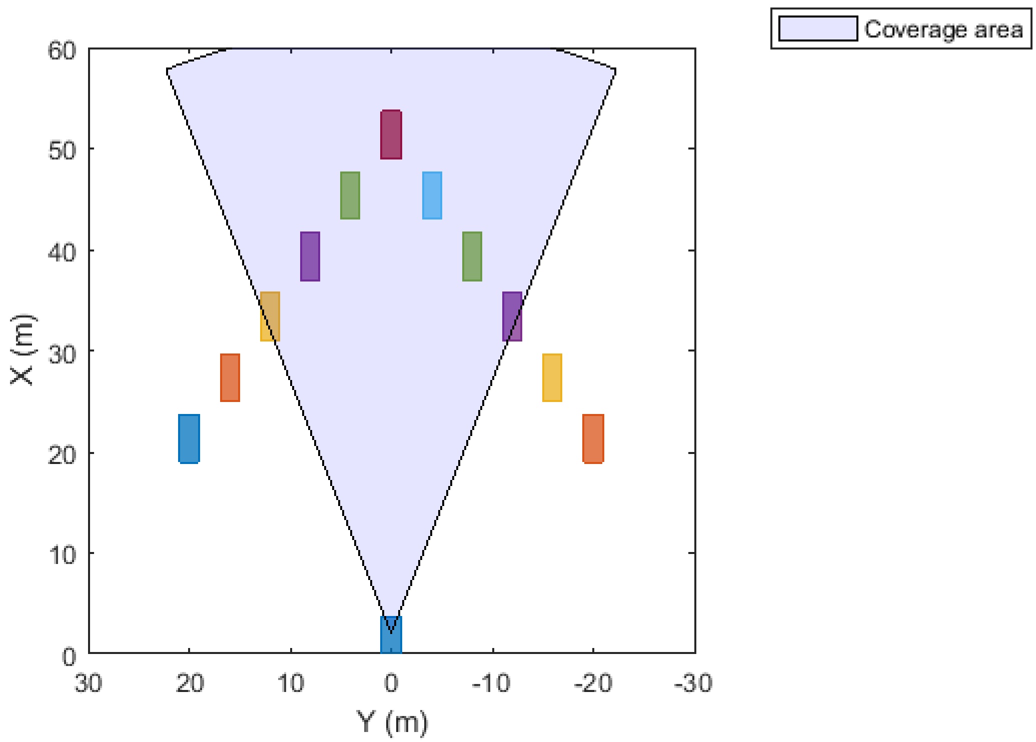
- `DetectionProbability = 1` — The sensor always generates detections for a target, as long as the target is not occluded and meets the range, speed, and image size constraints.
- `FalsePositivesPerImage = 0` — The sensor generates detections from only real targets in the driving scenario.
- `HasNoise = false` — The sensor does not add random noise to the reported position and velocity of the target. However, the `objectDetection` objects returned by the sensor have measurement noise values set to the noise variance that would have been added if `HasNoise` were `true`. With these noise values, you can process ideal detections using the `multiObjectTracker`. This technique is useful for analyzing maneuver lag without needing to run time-consuming Monte Carlo simulations.

```
idealSensor = visionDetectionGenerator( ...
    'SensorIndex',1, ...
    'UpdateInterval',scenario.SampleTime, ...
    'SensorLocation',[0.75*egoVehicle.Wheelbase 0], ...
    'Height',1.1, ...
    'Pitch',0, ...
    'Intrinsics',cameraIntrinsics(800,[320 240],[480 640]), ...
    'BoundingBoxAccuracy',50, ... % Make the noise large for illustrative purposes
    'ProcessNoiseIntensity',5, ...
    'MaxRange',60, ...
    'DetectionProbability',1, ...
    'FalsePositivesPerImage',0, ...
    'HasNoise',false, ...
    'ActorProfiles',actorProfiles(scenario))
```

```
idealSensor = 
  visionDetectionGenerator with properties:

               SensorIndex: 1
            UpdateInterval: 0.0100

            SensorLocation: [2.1000 0]
                    Height: 1.1000
                       Yaw: 0
                     Pitch: 0
                      Roll: 0
                Intrinsics: [1x1 cameraIntrinsics]

            DetectorOutput: 'Objects only'
               FieldOfView: [43.6028 33.3985]
                  MaxRange: 60
                  MaxSpeed: 100
       MaxAllowedOcclusion: 0.5000
        MinObjectImageSize: [15 15]

      DetectionProbability: 1
    FalsePositivesPerImage: 0

  Show all properties
```

Plot the coverage area of the ideal vision sensor.

```
legend('show')
caPlotter = coverageAreaPlotter(bep,'DisplayName','Coverage area','FaceColor','blue');
mountPosition = idealSensor.SensorLocation;
range = idealSensor.MaxRange;
orientation = idealSensor.Yaw;
fieldOfView = idealSensor.FieldOfView(1);
plotCoverageArea(caPlotter,mountPosition,range,orientation,fieldOfView);
```



Simulate Ideal Vision Detections

Obtain the positions of the targets. The positions are in ego vehicle coordinates.

```
gTruth = targetPoses(egoVehicle);
```

Generate timestamped vision detections. These detections are returned as a cell array of `objectDetection` objects.

```
time = scenario.SimulationTime;
dets = idealSensor(gTruth,time);
```

Inspect the measurement and measurement noise variance of the first (leftmost) detection. Even though the detection is ideal and therefore has no added random noise, the `MeasurementNoise` property shows the values as if the detection did have noise.

```
dets{1}.Measurement
```

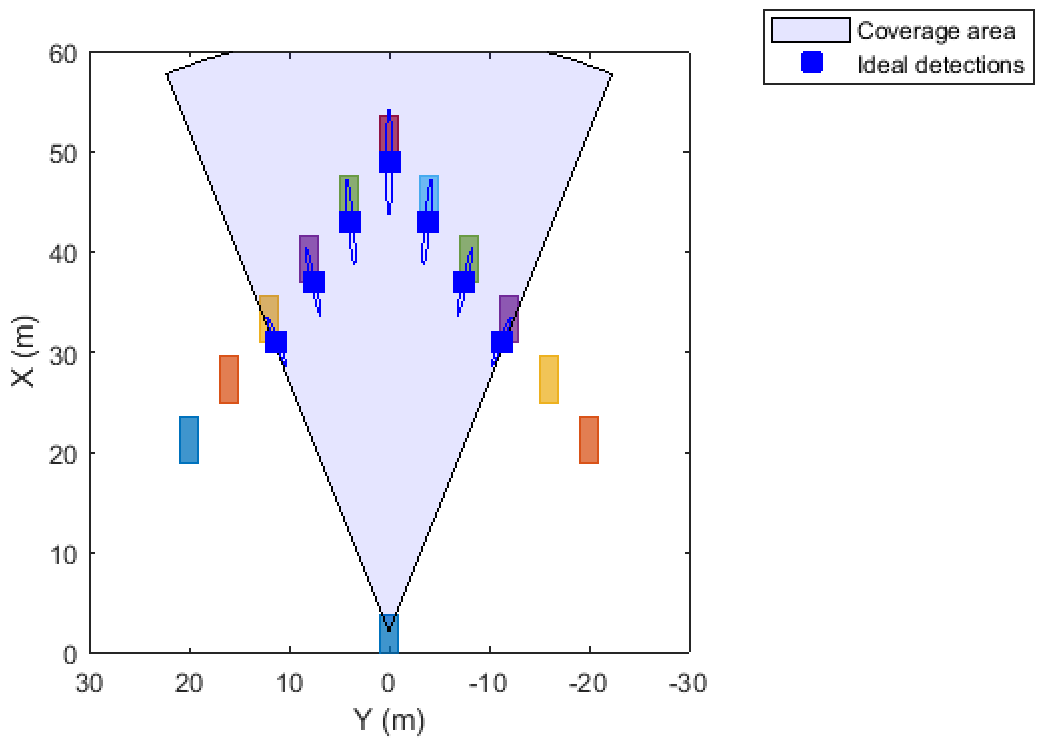
```
ans = 6×1

   31.0000
  -11.2237
         0
         0
         0
         0
```

```
dets{1}.MeasurementNoise
```

```
ans = 6×6

    1.5427   -0.5958         0         0         0         0
   -0.5958    0.2422         0         0         0         0
         0         0  100.0000         0         0         0
         0         0         0    0.5398   -0.1675         0
         0         0         0   -0.1675    0.1741         0
         0         0         0         0         0  100.0000
```

Plot the ideal detections and ellipses for the 2-sigma contour of the measurement noise covariance.

```
pos = cell2mat(cellfun(@(d)d.Measurement(1:2)',dets, ...
    'UniformOutput',false));
cov = reshape(cell2mat(cellfun(@(d)d.MeasurementNoise(1:2,1:2),dets, ...
    'UniformOutput',false))',2,2,[]);
plotter = trackPlotter(bep,'DisplayName','Ideal detections', ...
    'MarkerEdgeColor','blue','MarkerFaceColor','blue');
sigma = 2;
plotTrack(plotter,pos,sigma^2*cov)
```



Simulate Noisy Detections for Comparison

Create a noisy sensor based on the properties of the ideal sensor.

```
noisySensor = clone(idealSensor);
release(noisySensor)
noisySensor.HasNoise = true;
```

Reset the driving scenario back to its original state.

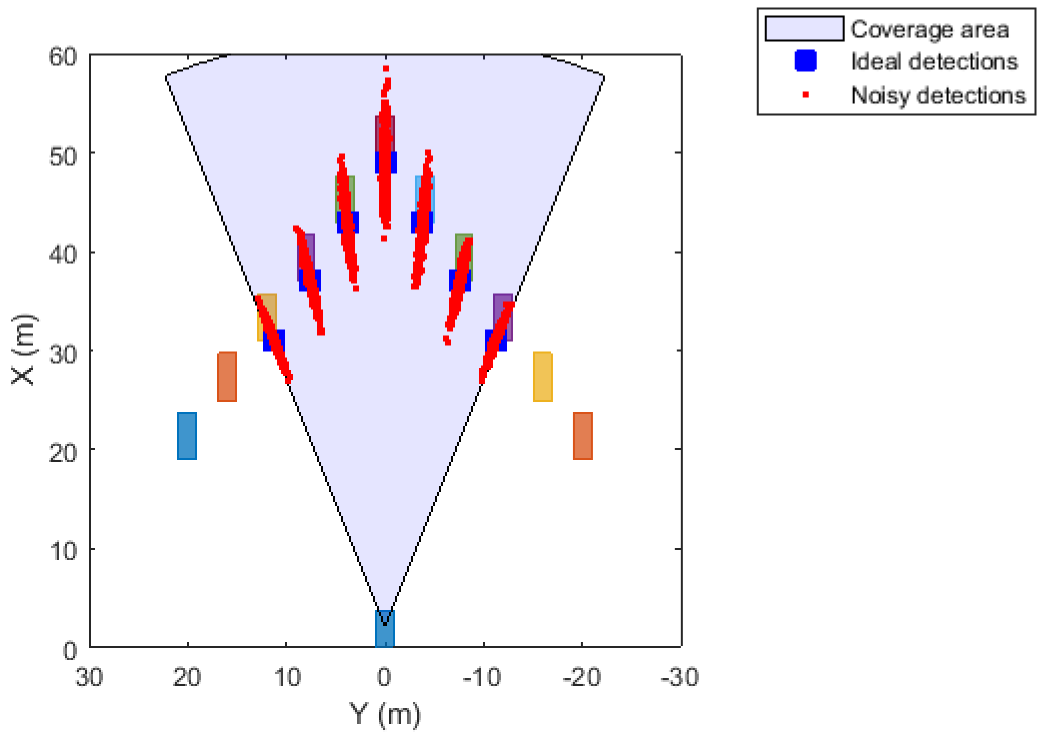
```
restart(scenario)
```

Collect statistics from the noisy detections.

```
numMonte = 1e3;
pos = [];
for itr = 1:numMonte
    time = scenario.SimulationTime;
    dets = noisySensor(gTruth,time);

    % Save noisy measurements
    pos = [pos;cell2mat(cellfun(@(d)d.Measurement(1:2)',dets,'UniformOutput',false))];

    advance(scenario);
end
```

Plot the noisy detections.

```
plotter = detectionPlotter(bep,'DisplayName','Noisy detections', ...
    'Marker','.','MarkerEdgeColor','red','MarkerFaceColor','red');
plotDetection(plotter,pos)
```



Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

Objects
`drivingScenario` | `laneMarking` | `lanespec` | `lidarPointCloudGenerator` | `monoCamera` | `multiObjectTracker` | `objectDetection` | `radarDetectionGenerator`

Functions
`actorPoses` | `actorProfiles` | `laneBoundaries` | `road`

Apps
Driving Scenario Designer

Topics
"Model Vision Sensor Detections"
"Coordinate Systems in Automated Driving Toolbox"

Introduced in R2017a

lidarPointCloudGenerator

Generate lidar point cloud data for driving scenario

Description

The `lidarPointCloudGenerator` System object generates detections from a lidar sensor mounted on an ego vehicle. All detections are referenced to the coordinate system of the ego vehicle or the vehicle-mounted sensor. You can use the `lidarPointCloudGenerator` object in a scenario containing actors and trajectories, which you can create by using a `drivingScenario` object. Using a statistical sensor model, `lidarPointCloudGenerator` object can simulate real detections with added random noise.

To generate lidar point clouds:

1. Create the `lidarPointCloudGenerator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
lidar = lidarPointCloudGenerator
lidar = lidarPointCloudGenerator(Name,Value)
```

Description

`lidar = lidarPointCloudGenerator` creates a `lidarPointCloudGenerator` object with default property values to generate a point cloud for a lidar sensor.

`lidar = lidarPointCloudGenerator(Name,Value)` sets properties on page 4-219 using one or more name-value pairs. For example, `lidarPointCloudGenerator('DetectionCoordinates','Sensor Cartesian','MaxRange',200)` creates a lidar point cloud generator that reports detections in the sensor Cartesian coordinate system and has a maximum detection range of 200 meters. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

SensorLocation — Sensor location
[1.5 0] (default) | [x y] vector

Location of the lidar sensor center, specified as a [x y] vector. The SensorLocation and Height properties define the coordinates of the lidar sensor with respect to the ego vehicle coordinate system. The default value corresponds to a lidar sensor mounted on a sedan, at the center of the roof's front edge. Units are in meters.

Example: [4 0.1]

Data Types: double

SensorIndex — Unique sensor identifier
1 (default) | positive integer

Unique sensor identifier, specified as a positive integer. This property distinguishes detections that come from different sensors in a multisensor system.

Example: 5

Data Types: double

UpdateInterval — Required time interval between sensor updates
0.1 (default) | positive real scalar

Required time interval between sensor updates, specified as a positive real scalar. The drivingScenario object calls the lidar point cloud generator at regular time intervals. lidarPointCloudGenerator object generates new detections at intervals defined by the UpdateInterval property. The value of the UpdateInterval property must be an integer multiple of the simulation time interval. Updates requested from the sensor between update intervals contain no detections. Units are in seconds.

Example: 5

Data Types: double

Height — Sensor height above ground plane
1.6 (default) | positive real scalar

Sensor height above the vehicle ground plane, specified as a positive real scalar. The default value corresponds to a lidar sensor mounted on a sedan, at the center of the roof's front edge. Units are in meters.

Example: 1.5

Data Types: double

Yaw — Yaw angle of lidar sensor
0 (default) | real scalar

Yaw angle of the lidar sensor, specified as a real scalar. The yaw angle is the angle between the center line of the ego vehicle and the downrange axis of the lidar sensor. A positive yaw angle corresponds to a clockwise rotation when looking in the positive direction of the *z*-axis of the ego vehicle coordinate system. Units are in degrees.

Example: -4

Data Types: double

`Pitch` — Pitch angle of lidar sensor
`0` (default) | real scalar

Pitch angle of the lidar sensor, specified as a real scalar. The pitch angle is the angle between the downrange axis of the lidar sensor and the *x*-*y* plane of the ego vehicle coordinate system. A positive pitch angle corresponds to a clockwise rotation when looking in the positive direction of the *y*-axis of the ego vehicle coordinate system. Units are in degrees.

Example: `3`

Data Types: `double`

`Roll` — Roll angle of lidar sensor
`0` (default) | real scalar

Roll angle of the lidar sensor, specified as a real scalar. The roll angle is the angle of rotation of the downrange axis of the lidar sensor around the *x*-axis of the ego vehicle coordinate system. A positive roll angle corresponds to a clockwise rotation when looking in the positive direction of the *x*-axis of the coordinate system. Units are in degrees.

Example: `-4`

Data Types: `double`

`MaxRange` — Maximum detection range
`120` (default) | positive real scalar

Maximum detection range, specified as a positive real scalar. The sensor cannot detect roads and actors beyond this range. Units are in meters.

Example: `200`

Data Types: `double`

`RangeAccuracy` — Accuracy of range measurements
`0.002` (default) | positive real scalar

Accuracy of range measurements, specified as a positive real scalar. Units are in meters.

Example: `0.01`

Data Types: `single` | `double`

`AzimuthResolution` — Azimuth resolution of lidar
`0.16` (default) | positive real scalar

Azimuth resolution of the lidar, specified as a positive real scalar. The azimuth resolution defines the minimum separation in azimuth angle at which the lidar can distinguish two targets. Units are in degrees.

Example: `0.5`

Data Types: `single` | `double`

`ElevationResolution` — Elevation resolution of lidar
`1.25` (default) | positive real scalar

Elevation resolution of the lidar, specified as a positive real scalar. The elevation resolution defines the minimum separation in elevation angle at which the lidar can distinguish two targets. Units are in degrees.

Example: 0.5

Data Types: single | double

AzimuthLimits — Azimuth limits of lidar
[-180 180] (default) | 1-by-2 real-valued vector

Azimuth limits of lidar, specified as a 1-by-2 real-valued vector of the form [min, max]. Units are in degrees.

Example: [-100 50]

Data Types: single | double

ElevationLimits — Elevation limits of lidar
[-20 20] (default) | 1-by-2 real-valued vector

Elevation limits of lidar, specified as a 1-by-2 real-valued vector of the form [min, max]. Units are in degrees.

Example: [-10 10]

Data Types: single | double

HasNoise — Enable adding noise to lidar sensor measurements
true (default) | false

Enable adding noise to lidar sensor measurements, specified as true or false. Set this property to true to add noise to the sensor measurements. Otherwise, the measurements have no noise.

Data Types: logical

HasOrganizedOutput — Output organized point cloud
true (default) | false

Output the generated data as an organized point cloud, specified as true or false. Set this property to true to output an organized point cloud. Otherwise, the output is unorganized.

Data Types: logical

HasEgoVehicle — Include ego vehicle in point cloud
true (default) | false

Include ego vehicle in the generated point cloud, specified as true or false. Set this property to true to include the ego vehicle in the output. Otherwise, the output point cloud has no ego vehicle.

Data Types: logical

HasRoadInputPort — Add roads to point cloud
true (default) | false

Add roads in the generated point cloud, specified as true or false. Set this property to true to add roads in the output. Otherwise, the output point cloud has no roads.

Data Types: logical

EgoVehicleActorID — ActorID of ego vehicle
1 (default) | positive integer

`ActorID` of ego vehicle, specified as a positive integer scalar. `ActorID` is the unique identifier for an actor.

Example: 4

Data Types: `single` | `double`

`DetectionCoordinates` — Coordinate system of reported detections
`'Ego Cartesian'` (default) | `'Sensor Cartesian'`

Coordinate system of reported detections, specified as one of these values:

- `'Ego Cartesian'` — Detections are reported in the ego vehicle Cartesian coordinate system.
- `'Sensor Cartesian'` — Detections are reported in the sensor Cartesian coordinate system.

Data Types: `char` | `string`

`ActorProfiles` — Actor profiles
structure | array of structures

Actor profiles, specified as a structure or as an array of structures. Each structure contains the physical and radar characteristics of an actor.

- If `ActorProfiles` is a single structure, all actors passed into the `lidarPointCloudGenerator` object use this profile.
- If `ActorProfiles` is an array, each actor passed into the object must have a unique actor profile.

To generate an array of structures for your driving scenario, use the `actorProfiles` function. The table shows the valid structure fields. If you do not specify a field, the fields are set to their default values. If no actors are passed into the object, then the `ActorID` field is not included.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`ClassID`	Classification identifier, specified as a nonnegative integer. `0` represents an object of an unknown or unassigned class.
`Length`	Length of actor, specified as a positive real-valued scalar. Units are in meters.
`Width`	Width of actor, specified as a positive real-valued scalar. Units are in meters.
`Height`	Height of actor, specified as a positive real-valued scalar. Units are in meters.
`OriginOffset`	Offset of actor's rotational center from its geometric center, specified as a real-valued vector of the form [*x*, *y*, *z*]. The rotational center, or origin, is located at the bottom center of the actor. For vehicles, the rotational center is the point on the ground beneath the center of the rear axle. Units are in meters.

Field	Description
MeshVertices	Mesh vertices of actor, specified as an *n*-by-*3* real-valued matrix of vertices. Each row in the matrix defines a point in 3-D space.
MeshFaces	Mesh faces of actor, specified as an *m*-by-*3* matrix of integers. Each row of MeshFaces represents a triangle defined by the vertex IDs, which are the row numbers of vertices.
RCSPattern	Radar cross-section (RCS) pattern of actor, specified as a numel(RCSElevationAngles)-by-numel(RCSAzimuthAngles) real-valued matrix. Units are in decibels per square meter.
RCSAzimuthAngles	Azimuth angles corresponding to rows of RCSPattern, specified as a vector of values in the range [–180, 180]. Units are in degrees.
RCSElevationAngles	Elevation angles corresponding to rows of RCSPattern, specified as a vector of values in the range [–90, 90]. Units are in degrees.

For full definitions of the structure fields, see the `actor` and `vehicle` functions.

Usage

Syntax

```
ptCloud = step(lidar,actors,simTime)
ptCloud = step(lidar,actors,simTime,rdMesh)
[ptCloud,isValidTime,clusters] = step(lidar,actors,simTime,rdMesh)
```

Description

`ptCloud = step(lidar,actors,simTime)` creates a statistical sensor model to generate a lidar point cloud, `ptCloud`, from sensor measurements taken of `actors` at the current simulation `simTime`. An `extendedObjectMesh` object, `rdMesh`, contains road data around the ego vehicle.

`ptCloud = step(lidar,actors,simTime,rdMesh)` additionally inputs `rdMesh`, an `extendedObjectMesh` object, which contains road data around the ego vehicle.

`[ptCloud,isValidTime,clusters] = step(lidar,actors,simTime,rdMesh)` additionally returns `isValidTime` and `clusters`. `isValidTime` specifies if the point cloud has been generated. `clusters` specifies the classification data of the generated point cloud.

Input Arguments

`actors` — Scenario actor poses
structure | structure array

Scenario actor poses, specified as a structure or structure array. Each structure corresponds to an actor. You can generate this structure using the `actorPoses` function. You can also create these structures manually. The table shows the properties that the object uses to generate detections. All other `actor` properties are ignored.

Field	Description
ActorID	Scenario-defined actor identifier, specified as a positive integer.
Position	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
Velocity	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
Roll	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
Pitch	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
Yaw	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
AngularVelocity	Angular velocity (*ω*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

For full definitions of the structure fields, see the `actor` and `vehicle` functions.

Data Types: `struct`

`rdMesh` — Mesh representation of roads near to actor
`extendedObjectMesh` object

Mesh representation of roads near to the actor, specified as an `extendedObjectMesh` object.

`simTime` — Current simulation time
positive real scalar

Current simulation time, specified as a positive real scalar. The `drivingScenario` object calls the lidar point cloud generator at regular time intervals to generate new point clouds at intervals defined by the `UpdateInterval` property. The value of the `UpdateInterval` property must be an integer multiple of the simulation time interval. Updates requested from the sensor between update intervals do not generate a point cloud. Units are in seconds.

Example: `10.5`

Data Types: `double`

Output Arguments

`ptCloud` — Point cloud data
`pointCloud` object

Point cloud data, returned as a `pointCloud` object.

`isValidTime` — Valid time to generate point cloud
`0` | `1`

Valid time to generate point cloud, returned as `0` or `1`. `isValidTime` is `0` when updates are requested at times that are between update intervals specified by `UpdateInterval`.

Data Types: `logical`

`clusters` — Classification data of generated point cloud
N-by-2 vector

Classification data of the generated point cloud, returned as an *N*-by-2 vector. The vector defines the IDs of the target from which the point cloud was generated. *N* is equal to the `Count` property of the `pointCloud` object. The vector contains `ActorID` in the first column and `ClassID` in the second column.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `lidarPointCloudGenerator`

isLocked   Determine if System object is in use

Common to All System Objects

step   Run System object algorithm
release   Release resources and allow changes to System object property values and input characteristics
reset   Reset internal states of System object

Examples

Generate Lidar Point Cloud Data of Multiple Actors

Generate lidar point cloud data for a driving scenario with multiple actors by using the `lidarPointCloudGenerator` System object. Create the driving scenario by using `drivingScenario` object. It contains an ego-vehicle, pedestrian and two other vehicles.

Create and plot a driving scenario with multiple vehicles

Create a driving scenario.

```
scenario = drivingScenario;
```

Add a straight road to the driving scenario. The road has one lane in each direction.

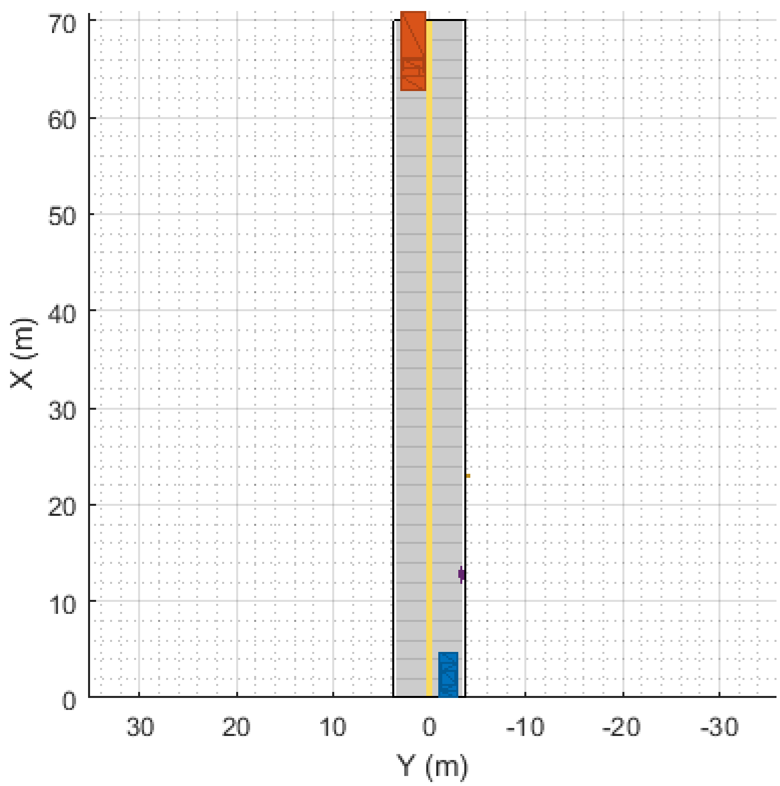
```
roadCenters = [0 0 0; 70 0 0];
laneSpecification = lanespec([1 1]);
road(scenario,roadCenters,'Lanes',laneSpecification);
```

Add an ego vehicle to the driving scenario.

```
egoVehicle = vehicle(scenario,'ClassID',1,'Mesh',driving.scenario.carMesh);
waypoints = [1 -2 0; 35 -2 0];
trajectory(egoVehicle,waypoints,10);
```

Add a truck, pedestrian, and bicycle to the driving scenario and plot the scenario.

```
truck = vehicle(scenario,'ClassID',2,'Length', 8.2,'Width',2.5,'Height',3.5, ...
  'Mesh',driving.scenario.truckMesh);
waypoints = [70 1.7 0; 20 1.9 0];
trajectory(truck,waypoints,15);
pedestrian = actor(scenario,'ClassID',4,'Length',0.24,'Width',0.45,'Height',1.7, ...
  'Mesh',driving.scenario.pedestrianMesh);
waypoints = [23 -4 0; 10.4 -4 0];
trajectory(pedestrian,waypoints,1.5);
bicycle = actor(scenario,'ClassID',3,'Length',1.7,'Width',0.45,'Height',1.7, ...
  'Mesh',driving.scenario.bicycleMesh);
waypoints = [12.7 -3.3 0; 49.3 -3.3 0];
trajectory(bicycle,waypoints,5);
plot(scenario,'Meshes','on')
```



Generate and plot lidar point cloud data

Create a `lidarPointCloudGenerator` System object.

```
lidar = lidarPointCloudGenerator;
```

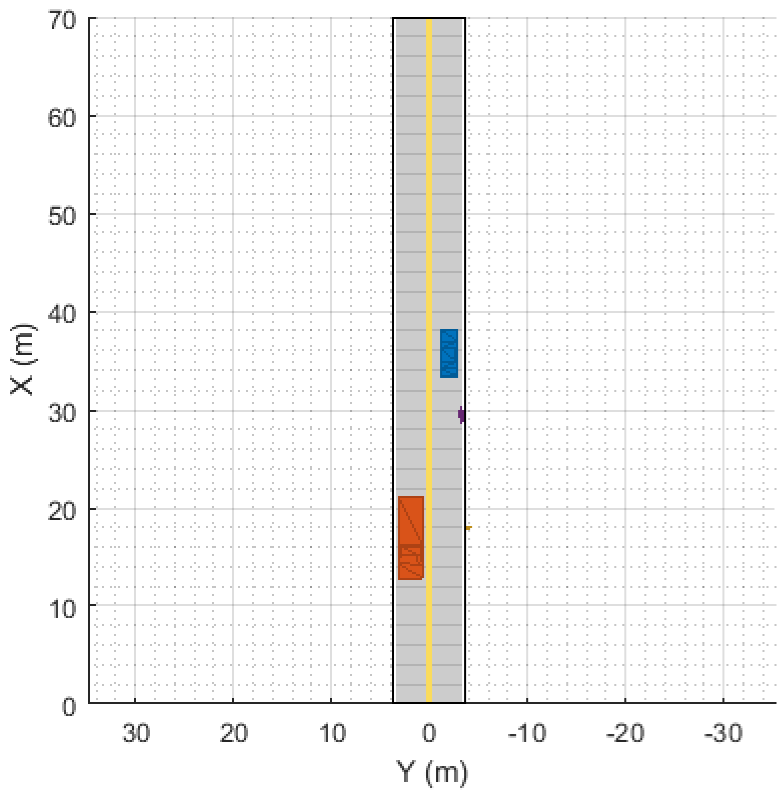
Add actor profiles and the ego vehicle actor ID from the driving scenario to the System object.

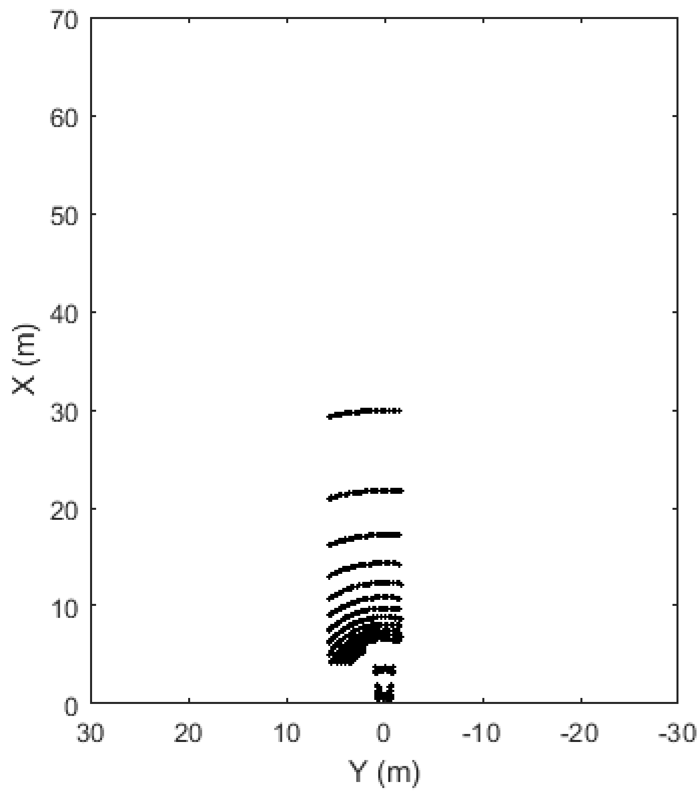
```
lidar.ActorProfiles = actorProfiles(scenario);
lidar.EgoVehicleActorID = egoVehicle.ActorID;
```

Plot the point cloud data.

```
bep = birdsEyePlot('Xlimits',[0 70],'YLimits',[-30 30]);
plotter = pointCloudPlotter(bep);
```

```
legend('off');
while advance(scenario)
    tgts = targetPoses(egoVehicle);
    rdmesh = roadMesh(egoVehicle);
    [ptCloud,isValidTime] = lidar(tgts,rdmesh,scenario.SimulationTime);
    if isValidTime
        plotPointCloud(plotter,ptCloud);
    end
end
```





See Also

Objects
drivingScenario | extendedObjectMesh | laneMarking | lanespec | monoCamera | multiObjectTracker | objectDetection | radarDetectionGenerator | visionDetectionGenerator

Functions
actorPoses | actorProfiles | laneBoundaries | road | roadMesh

Blocks
Lidar Point Cloud Generator

Apps
Driving Scenario Designer

Topics
"Track-Level Fusion of Radar and Lidar Data"
"Coordinate Systems in Automated Driving Toolbox"

Introduced in R2020a

roadMesh

Mesh representation of roads near actor

Syntax

```
mesh = roadMesh(ac)
mesh = roadMesh(ac,maxRadius)
```

Description

`mesh = roadMesh(ac)` creates a mesh representation of roads nearest to the specified actor `ac`. The function returns the mesh as an `extendedObjectMesh` object. By default, the function searches for roads within a 120 m radius of the input actor.

`mesh = roadMesh(ac,maxRadius)`specifies the maximum search radius.

Examples

Generate Lidar Point Cloud Data of Multiple Actors

Generate lidar point cloud data for a driving scenario with multiple actors by using the `lidarPointCloudGenerator` System object. Create the driving scenario by using `drivingScenario` object. It contains an ego-vehicle, pedestrian and two other vehicles.

Create and plot a driving scenario with multiple vehicles

Create a driving scenario.

```
scenario = drivingScenario;
```

Add a straight road to the driving scenario. The road has one lane in each direction.

```
roadCenters = [0 0 0; 70 0 0];
laneSpecification = lanespec([1 1]);
road(scenario,roadCenters,'Lanes',laneSpecification);
```

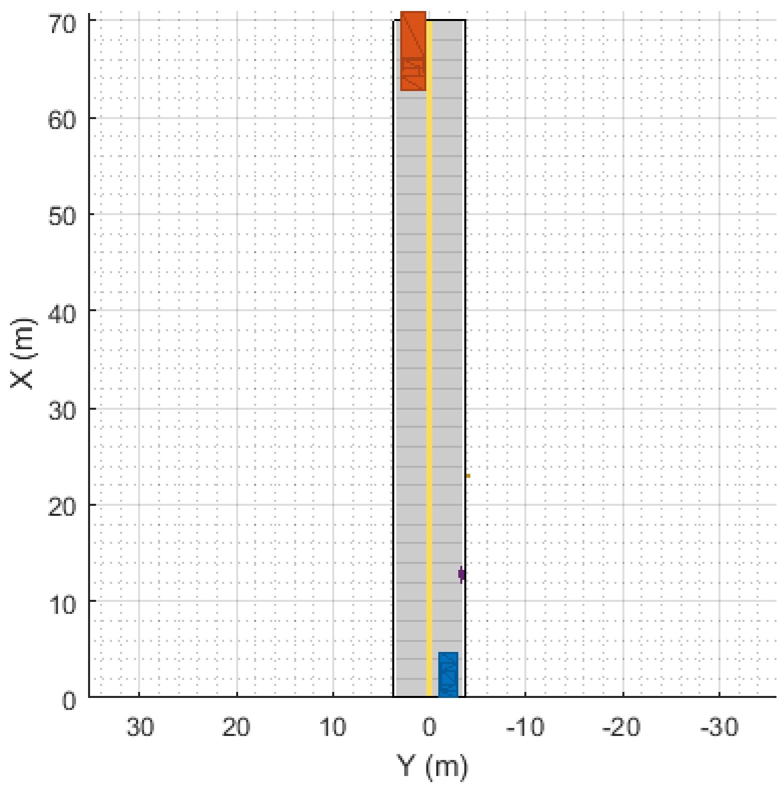
Add an ego vehicle to the driving scenario.

```
egoVehicle = vehicle(scenario,'ClassID',1,'Mesh',driving.scenario.carMesh);
waypoints = [1 -2 0; 35 -2 0];
trajectory(egoVehicle,waypoints,10);
```

Add a truck, pedestrian, and bicycle to the driving scenario and plot the scenario.

```
truck = vehicle(scenario,'ClassID',2,'Length', 8.2,'Width',2.5,'Height',3.5, ...
  'Mesh',driving.scenario.truckMesh);
waypoints = [70 1.7 0; 20 1.9 0];
trajectory(truck,waypoints,15);
pedestrian = actor(scenario,'ClassID',4,'Length',0.24,'Width',0.45,'Height',1.7, ...
  'Mesh',driving.scenario.pedestrianMesh);
waypoints = [23 -4 0; 10.4 -4 0];
```

```
trajectory(pedestrian,waypoints,1.5);
bicycle = actor(scenario,'ClassID',3,'Length',1.7,'Width',0.45,'Height',1.7, ...
  'Mesh',driving.scenario.bicycleMesh);
waypoints = [12.7 -3.3 0; 49.3 -3.3 0];
trajectory(bicycle,waypoints,5);
plot(scenario,'Meshes','on')
```



Generate and plot lidar point cloud data

Create a `lidarPointCloudGenerator` System object.

```
lidar = lidarPointCloudGenerator;
```

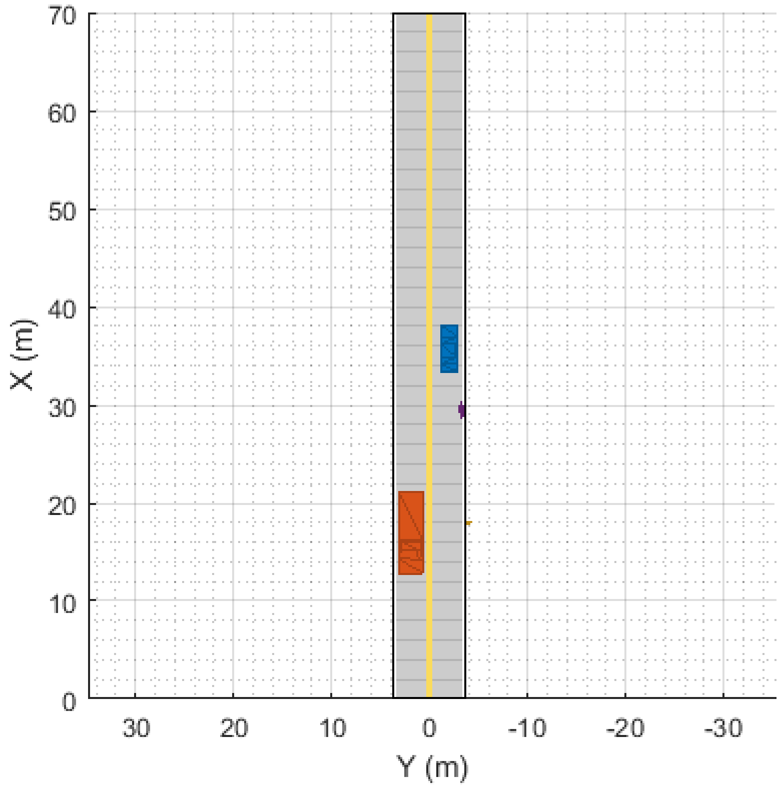
Add actor profiles and the ego vehicle actor ID from the driving scenario to the System object.

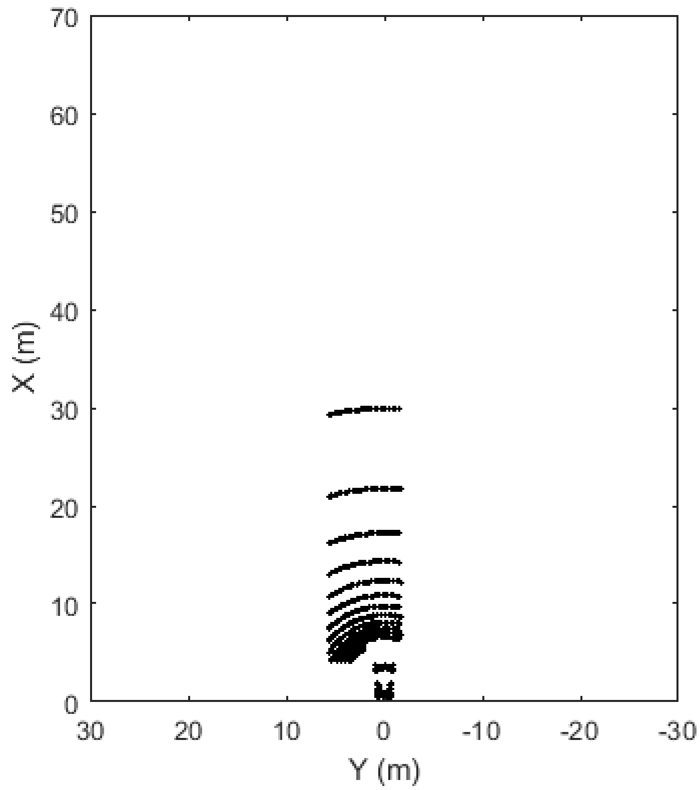
```
lidar.ActorProfiles = actorProfiles(scenario);
lidar.EgoVehicleActorID = egoVehicle.ActorID;
```

Plot the point cloud data.

```
bep = birdsEyePlot('Xlimits',[0 70],'YLimits',[-30 30]);
plotter = pointCloudPlotter(bep);
legend('off');
while advance(scenario)
    tgts = targetPoses(egoVehicle);
    rdmesh = roadMesh(egoVehicle);
    [ptCloud,isValidTime] = lidar(tgts,rdmesh,scenario.SimulationTime);
    if isValidTime
```

```
            plotPointCloud(plotter,ptCloud);
        end
    end
```





Input Arguments

ac — Actor
Actor object | Vehicle object

Actor belonging to a drivingScenario object, specified as an Actor or Vehicle object. To create these objects, use the actor and vehicle functions, respectively.

maxRadius — Maximum radius of search area
120 (default) | integer in the range [1, 500]

Maximum radius of search area, specified as an integer in the range [1, 500].

Data Types: single | double | int16 | int32 | int64 | uint16 | uint32 | uint64

Output Arguments

mesh — Mesh representation of roads near to actor
extendedObjectMesh object

Mesh representation of roads near to the actor, returned as an extendedObjectMesh object.

See Also

Objects
`extendedObjectMesh` | `lidarPointCloudGenerator`

Functions
`driving.scenario.bicycleMesh` | `driving.scenario.carMesh` | `driving.scenario.pedestrianMesh` | `driving.scenario.truckMesh`

Introduced in R2020a

driving.Path

Planned vehicle path

Description

The `driving.Path` object represents a vehicle path composed of a sequence of path segments. These segments can be either `driving.DubinsPathSegment` objects or `driving.ReedsSheppPathSegment` objects and are stored in the `PathSegments` property of `driving.Path`.

To check the validity of the path against a `vehicleCostmap` object, use the `checkPathValidity` function. To interpolate poses along the length of the path, use the `interpolate` function.

Creation

To create a `driving.Path` object, use the `plan` function, specifying a `pathPlannerRRT` object as input.

Properties

`StartPose` — Initial pose of vehicle
[*x*, *y*, *Θ*] vector

This property is read-only.

Initial pose of the vehicle, specified as an [*x*, *y*, *Θ*] vector. *x* and *y* are in world units, such as meters. *Θ* is in degrees.

`GoalPose` — Goal pose of vehicle
[*x*, *y*, *Θ*] vector

This property is read-only.

Goal pose of the vehicle, specified as an [*x*, *y*, *Θ*] vector. *x* and *y* are in world units, such as meters. *Θ* is in degrees.

`PathSegments` — Segments along path
array of `driving.DubinsPathSegment` objects | array of `driving.ReedsSheppPathSegment` objects

This property is read-only.

Segments along the path, specified as an array of `driving.DubinsPathSegment` objects or `driving.ReedsSheppPathSegment` objects.

`Length` — Length of path
positive real scalar

This property is read-only.

Length of the path, in world units, specified as a positive real scalar.

Object Functions

interpolate	Interpolate poses along planned vehicle path
plot	Plot planned vehicle path

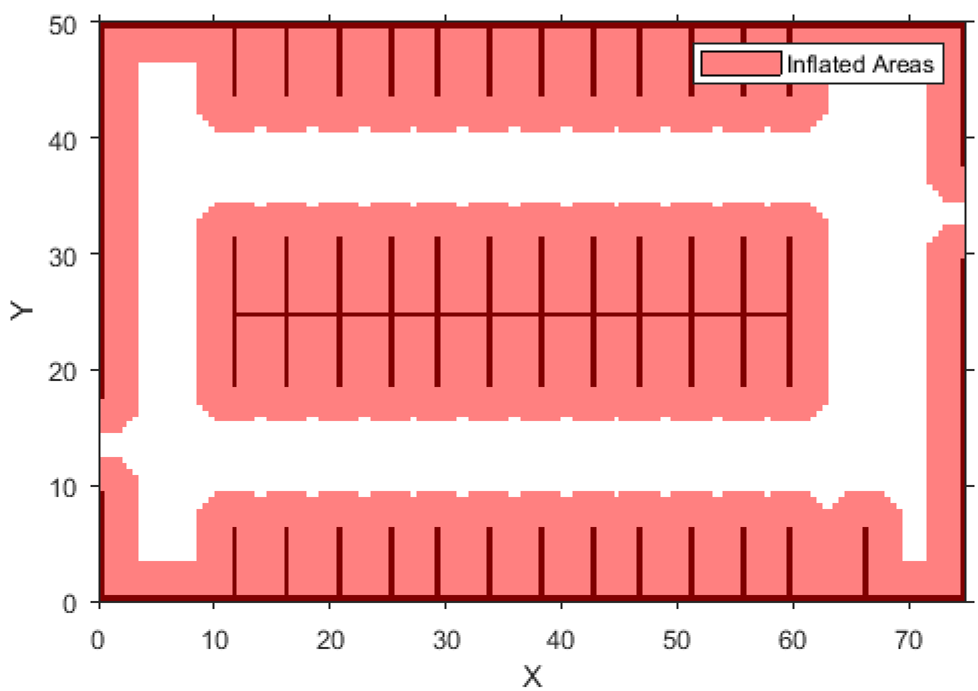
Examples

Plan Path and Check Its Validity

Plan a vehicle path through a parking lot by using the optimal rapidly exploring random tree (RRT*) algorithm. Check that the path is valid, and then plot the transition poses along the path.

Load a costmap of a parking lot. Plot the costmap to see the parking lot and inflated areas for the vehicle to avoid.

```
data = load('parkingLotCostmap.mat');
costmap = data.parkingLotCostmap;
plot(costmap)
```



Define start and goal poses for the vehicle as [*x*, *y*, *Θ*] vectors. World units for the (*x*,*y*) locations are in meters. World units for the *Θ* orientation angles are in degrees.

```
startPose = [4, 4, 90]; % [meters, meters, degrees]
goalPose = [30, 13, 0];
```

Use a `pathPlannerRRT` object to plan a path from the start pose to the goal pose.

```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Check that the path is valid.

```
isPathValid = checkPathValidity(refPath,costmap)
```

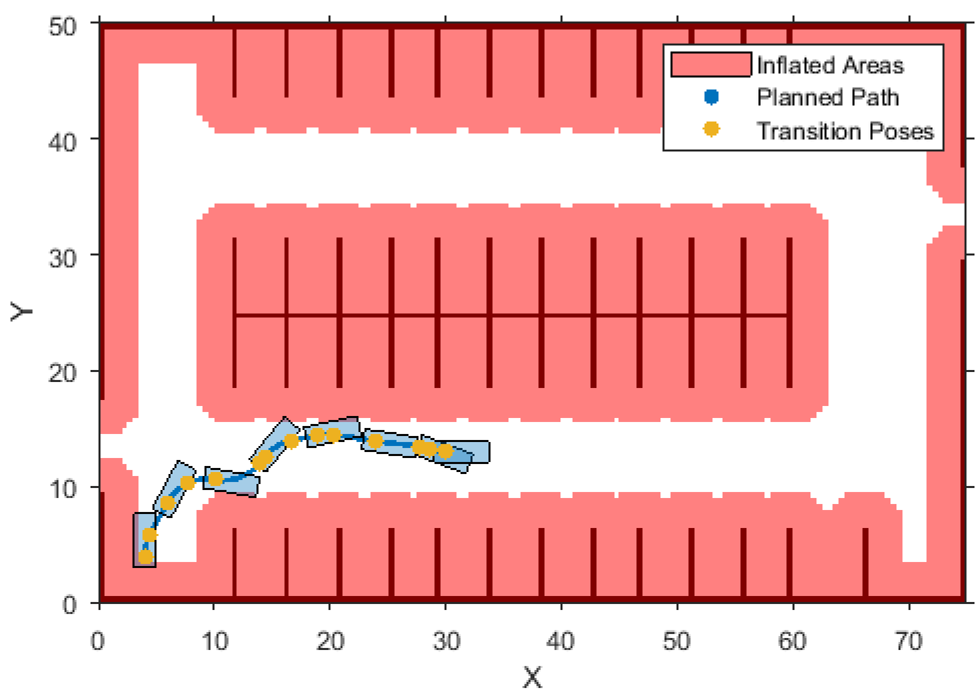
```
isPathValid = logical
   1
```

Interpolate the transition poses along the path.

```
transitionPoses = interpolate(refPath);
```

Plot the planned path and the transition poses on the costmap.

```
hold on
plot(refPath,'DisplayName','Planned Path')
scatter(transitionPoses(:,1),transitionPoses(:,2),[],'filled', ...
    'DisplayName','Transition Poses')
hold off
```

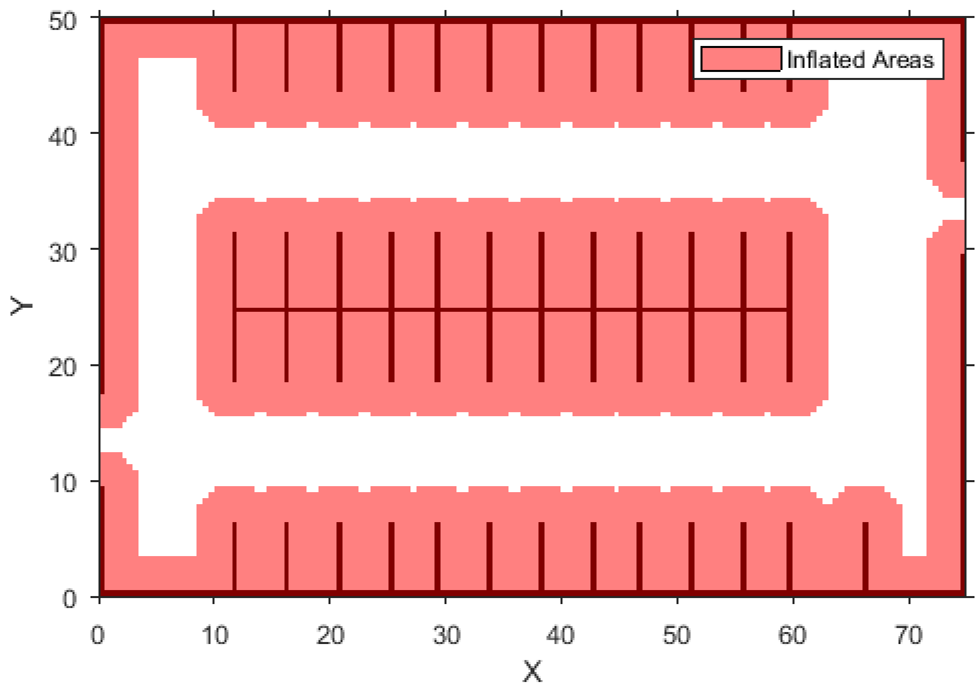


Plan Path and Interpolate Along Path

Plan a vehicle path through a parking lot by using the rapidly exploring random tree (RRT*) algorithm. Interpolate the poses of the vehicle at points along the path.

Load a costmap of a parking lot. Plot the costmap to see the parking lot and inflated areas for the vehicle to avoid.

```
data = load('parkingLotCostmap.mat');
costmap = data.parkingLotCostmap;
plot(costmap)
```



Define start and goal poses for the vehicle as `[`*x*, *y*, *Θ*`]` vectors. World units for the (*x*,*y*) locations are in meters. World units for the *Θ* orientation angles are in degrees.

```
startPose = [4, 4, 90]; % [meters, meters, degrees]
goalPose = [30, 13, 0];
```

Use a `pathPlannerRRT` object to plan a path from the start pose to the goal pose.

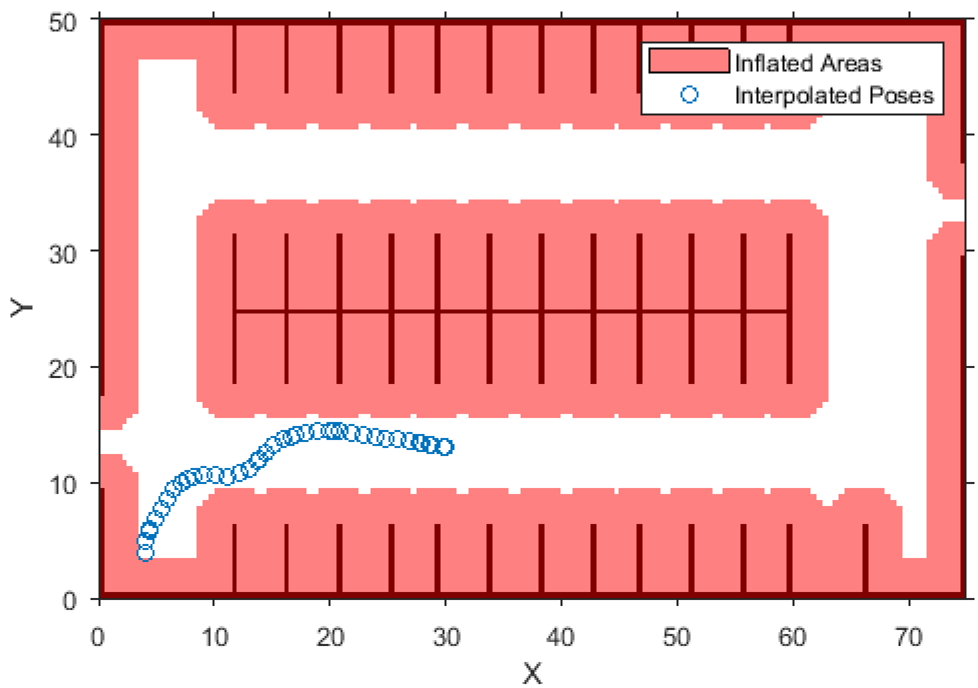
```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Interpolate the vehicle poses every 1 meter along the entire path.

```
lengths = 0 : 1 : refPath.Length;
poses = interpolate(refPath,lengths);
```

Plot the interpolated poses on the costmap.

```
plot(costmap)
hold on
scatter(poses(:,1),poses(:,2),'DisplayName','Interpolated Poses')
hold off
```



Compatibility Considerations

`connectingPoses` function and `driving.Path` object properties `KeyPoses` and `NumSegments` are not recommended

Not recommended starting in R2018b

The `connectingPoses` function and the `KeyPoses` and `NumSegments` properties of the `driving.Path` object are not recommended. Instead, use the `interpolate` function, which returns key poses, connecting poses, transition poses, and direction changes. The `KeyPoses` and `NumSegments` properties are no longer relevant. `KeyPoses`, `NumSegments`, and `connectingPoses` will be removed in a future release.

In R2018a, `connectingPoses` enabled you to obtain intermediate poses either along the entire path or along the path segments that are between key poses (as specified by `KeyPoses`). Using the `interpolate` function, you can now obtain intermediate poses at any specified point along the path. The `interpolate` function also provides transition poses at which changes in direction occur.

Update Code

Remove all instances of `KeyPoses` and `NumSegments` and replace all instances of `connectingPoses` with `interpolate`. The table shows typical usages of `connectingPoses` and how to update your code to use `interpolate` instead. Here, `path` is a `driving.Path` object returned by `pathPlannerRRT`.

Discouraged Usage	Recommended Replacement
`poses = connectingPoses(path);`	`poses = interpolate(path);`

`segID = 1;`
`posesSegment = connectingPoses(path,segID);`
	`interpolate` does not have a direct syntax for obtaining segment poses. However, you can sample poses of a segment using a specified step time. For example:
	
	`step = 0.1;`
	`samples = 0 : step : path.PathSegments(1).Length;`
	`segmentPoses = interpolate(path,samples);`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`checkPathValidity` | `interpolate` | `plan` | `plot` | `smoothPathSpline`

Objects

`driving.DubinsPathSegment` | `driving.ReedsSheppPathSegment` | `pathPlannerRRT` | `vehicleCostmap`

Topics

"Automated Parking Valet"

Introduced in R2018a

connectingPoses

Package: `driving`

(Not recommended) Obtain connecting poses along vehicle path

Note `connectingPoses` is not recommended. Use `interpolate` instead. For more information, see "Compatibility Considerations"

Syntax

```
poses = connectingPoses(path)
poses = connectingPoses(path,segID)
poses = connectingPoses( ___ ,'NumSamples',numSamples)
```

Description

`poses = connectingPoses(path)` returns the connecting poses that are between the key poses of a vehicle path.

`poses = connectingPoses(path,segID)` returns the connecting poses that are along the path segment specified by `segID`.

`poses = connectingPoses( ___ ,'NumSamples',numSamples)` specifies the number of connecting poses to compute between successive key poses, using either of the preceding syntaxes.

Input Arguments

`path` — Planned vehicle path
`driving.Path` object

Planned vehicle path from which to obtain connecting poses, specified as a `driving.Path` object.

`segID` — ID of path segment
positive integer

ID of the path segment from which to obtain connecting poses, specified as a positive integer. Each path segment has two successive key poses as its endpoints. `segID` must be less than the number of segments in the input `path`.

`numSamples` — Number of connecting poses to sample
`100` (default) | integer greater than 1

Number of connecting poses to sample from each segment, specified as an integer greater than 1.

Example: `'NumSamples',50`

Output Arguments

poses — Connecting poses
m-by-3 matrix of [*x*, *y*, *Θ*] poses

Connecting poses, returned as an *m*-by-3 matrix of [*x*, *y*, *Θ*] poses. Each row corresponds to a separate pose. *x* and *y* are specified in world coordinates and *Θ* is in degrees. `poses` includes all key poses.

Compatibility Considerations

`connectingPoses` function and `driving.Path` object properties `KeyPoses` and `NumSegments` are not recommended
Not recommended starting in R2018b

The `connectingPoses` function and the `KeyPoses` and `NumSegments` properties of the `driving.Path` object are not recommended. Instead, use the `interpolate` function, which returns key poses, connecting poses, transition poses, and direction changes. The `KeyPoses` and `NumSegments` properties are no longer relevant. `KeyPoses`, `NumSegments`, and `connectingPoses` will be removed in a future release.

In R2018a, `connectingPoses` enabled you to obtain intermediate poses either along the entire path or along the path segments that are between key poses (as specified by `KeyPoses`). Using the `interpolate` function, you can now obtain intermediate poses at any specified point along the path. The `interpolate` function also provides transition poses at which changes in direction occur.

Update Code

Remove all instances of `KeyPoses` and `NumSegments` and replace all instances of `connectingPoses` with `interpolate`. The table shows typical usages of `connectingPoses` and how to update your code to use `interpolate` instead. Here, `path` is a `driving.Path` object returned by `pathPlannerRRT`.

Discouraged Usage	Recommended Replacement
`poses = connectingPoses(path);`	`poses = interpolate(path);`

`segID = 1;`
`posesSegment = connectingPoses(path,segID);`
	`interpolate` does not have a direct syntax for obtaining segment poses. However, you can sample poses of a segment using a specified step time. For example:
	
	`step = 0.1;`
	`samples = 0 : step : path.PathSegments(1).Length;`
	`segmentPoses = interpolate(path,samples);`

See Also

Functions
`checkPathValidity` | `interpolate` | `plan`

Objects
`driving.Path` | `pathPlannerRRT`

Topics
"Automated Parking Valet"

Introduced in R2018a

plot

Package: `driving`

Plot planned vehicle path

Syntax

```
plot(refPath)
plot(refPath,Name,Value)
```

Description

`plot(refPath)` plots the planned vehicle path.

`plot(refPath,Name,Value)` specifies options using one or more name-value pair arguments. For example, `plot(path,'Vehicle','off')` plots the path without displaying the vehicle.

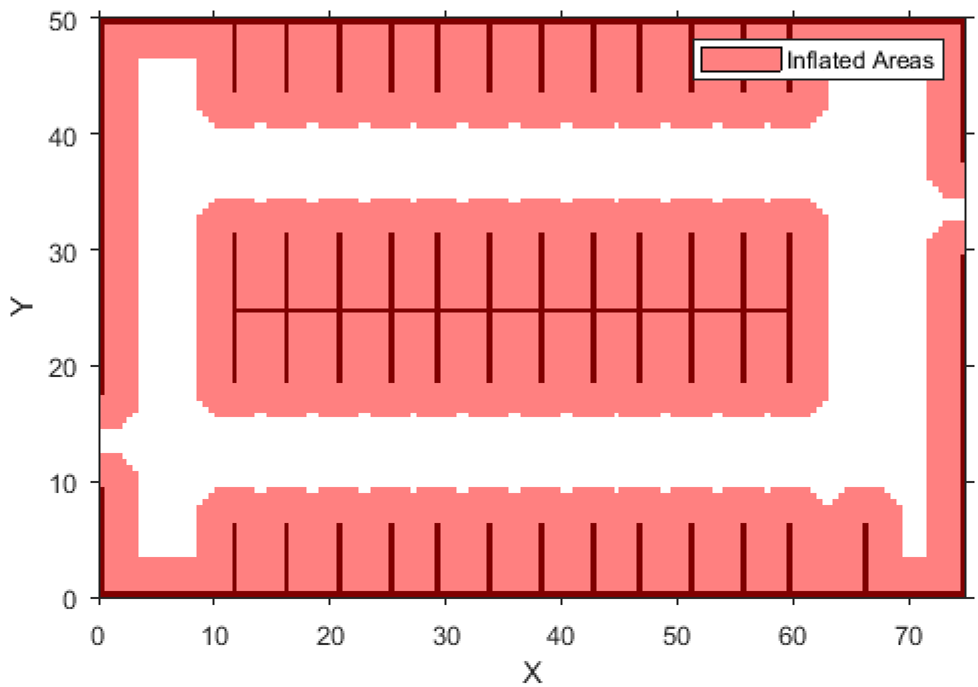
Examples

Plan Path and Check Its Validity

Plan a vehicle path through a parking lot by using the optimal rapidly exploring random tree (RRT*) algorithm. Check that the path is valid, and then plot the transition poses along the path.

Load a costmap of a parking lot. Plot the costmap to see the parking lot and inflated areas for the vehicle to avoid.

```
data = load('parkingLotCostmap.mat');
costmap = data.parkingLotCostmap;
plot(costmap)
```



Define start and goal poses for the vehicle as [*x*, *y*, *Θ*] vectors. World units for the (*x*,*y*) locations are in meters. World units for the *Θ* orientation angles are in degrees.

```
startPose = [4, 4, 90]; % [meters, meters, degrees]
goalPose = [30, 13, 0];
```

Use a `pathPlannerRRT` object to plan a path from the start pose to the goal pose.

```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Check that the path is valid.

```
isPathValid = checkPathValidity(refPath,costmap)
```

```
isPathValid = logical
   1
```

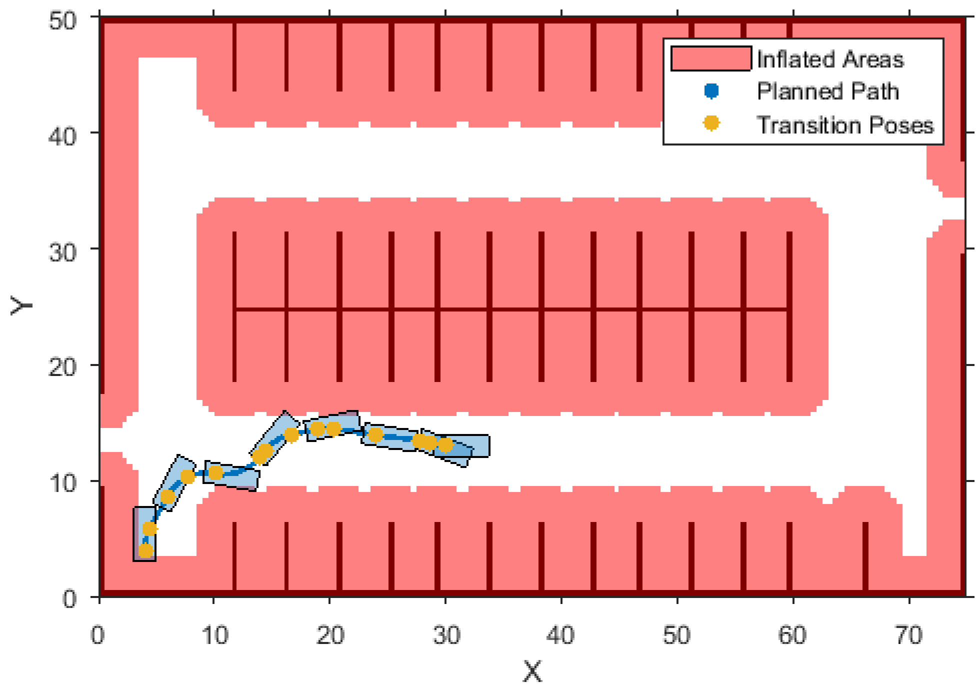
Interpolate the transition poses along the path.

```
transitionPoses = interpolate(refPath);
```

Plot the planned path and the transition poses on the costmap.

```
hold on
plot(refPath,'DisplayName','Planned Path')
scatter(transitionPoses(:,1),transitionPoses(:,2),[],'filled', ...
```

```
    'DisplayName','Transition Poses')
hold off
```



Input Arguments

refPath — Planned vehicle path
driving.Path object

Planned vehicle path, specified as a `driving.Path` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Inflation','off'`

Parent — Axes object
axes object

Axes object in which to draw the plot, specified as the comma-separated pair consisting of `'Parent'` and an `axes` object. If you do not specify `Parent`, a new figure is created.

Vehicle — Display vehicle
`'on'` (default) | `'off'`

Display vehicle, specified as the comma-separated pair consisting of `'Vehicle'` and `'on'` or `'off'`. Setting this argument to `'on'` displays the vehicle along the path.

`VehicleDimensions` — Dimensions of vehicle

`vehicleDimensions` object

Dimensions of the vehicle, specified as the comma-separated pair consisting of `'VehicleDimensions'` and a `vehicleDimensions` object.

`DisplayName` — Name of entry in legend

`''` (default) | character vector | string scalar

Name of the entry in the legend, specified as the comma-separated pair consisting of `'DisplayName'` and a character vector or string scalar.

`Color` — Path color

color name | short color name | RGB triplet

Path color, specified as the comma-separated pair consisting of `'Color'` and a color name, short color name, or RGB triplet.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the named color options and the equivalent RGB triplet values.

Color Name	Color Short Name	RGB Triplet	Appearance
`'red'`	`'r'`	`[1 0 0]`	
`'green'`	`'g'`	`[0 1 0]`	
`'blue'`	`'b'`	`[0 0 1]`	
`'cyan'`	`'c'`	`[0 1 1]`	
`'magenta'`	`'m'`	`[1 0 1]`	
`'yellow'`	`'y'`	`[1 1 0]`	
`'black'`	`'k'`	`[0 0 0]`	
`'white'`	`'w'`	`[1 1 1]`	

Example: `'Color',[1 0 1]`

Example: `'Color','m'`

Example: `'Color','magenta'`

`Tag` — Tag to identify path

`''` (default) | character vector | string scalar

Tag to identify path, specified as the comma-separated pair consisting of `'Tag'` and a character vector or string scalar.

See Also

Functions
checkPathValidity | interpolate | plan

Objects
driving.Path | pathPlannerRRT | vehicleDimensions

Topics
"Automated Parking Valet"

Introduced in R2018a

interpolate

Package: `driving`

Interpolate poses along planned vehicle path

Syntax

```
poses = interpolate(refPath)
poses = interpolate(refPath,lengths)
[poses,directions] = interpolate( ___ )
```

Description

`poses = interpolate(refPath)` interpolates along the length of a reference path, returning transition poses. For more information, see Transition Poses on page 4-254.

`poses = interpolate(refPath,lengths)` interpolates poses at specified points along the length of the path. In addition to including poses corresponding to specified lengths, `poses` also includes the transition poses.

`[poses,directions] = interpolate( ___ )` also returns the motion directions of the vehicle at each pose, using inputs from any of the preceding syntaxes.

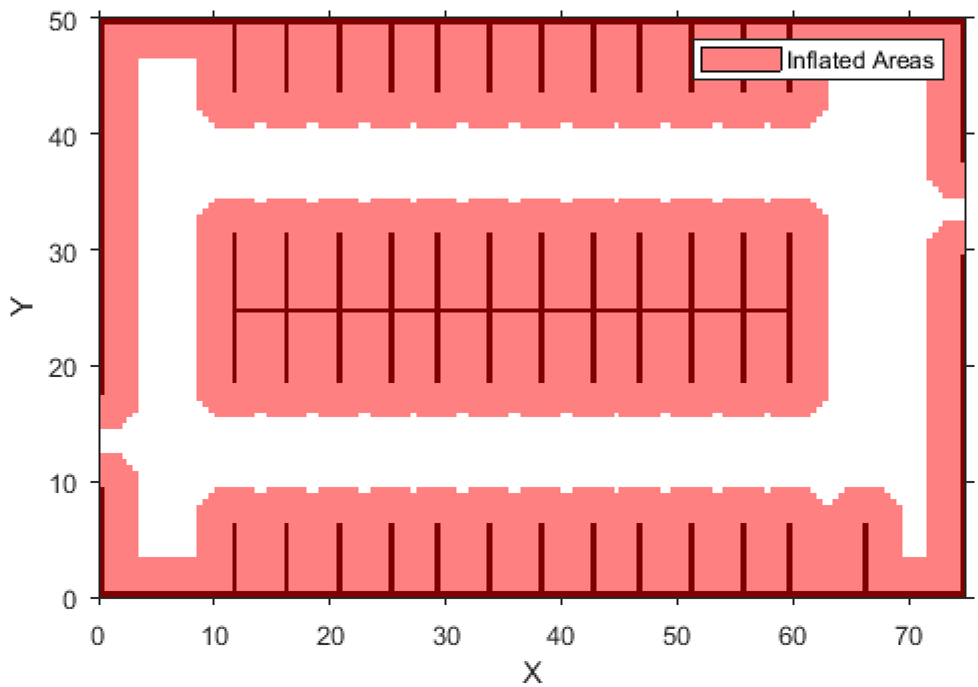
Examples

Plan Path and Check Its Validity

Plan a vehicle path through a parking lot by using the optimal rapidly exploring random tree (RRT*) algorithm. Check that the path is valid, and then plot the transition poses along the path.

Load a costmap of a parking lot. Plot the costmap to see the parking lot and inflated areas for the vehicle to avoid.

```
data = load('parkingLotCostmap.mat');
costmap = data.parkingLotCostmap;
plot(costmap)
```



Define start and goal poses for the vehicle as [*x*, *y*, *Θ*] vectors. World units for the (*x*,*y*) locations are in meters. World units for the *Θ* orientation angles are in degrees.

```
startPose = [4, 4, 90]; % [meters, meters, degrees]
goalPose = [30, 13, 0];
```

Use a `pathPlannerRRT` object to plan a path from the start pose to the goal pose.

```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Check that the path is valid.

```
isPathValid = checkPathValidity(refPath,costmap)
```

```
isPathValid = logical
   1
```

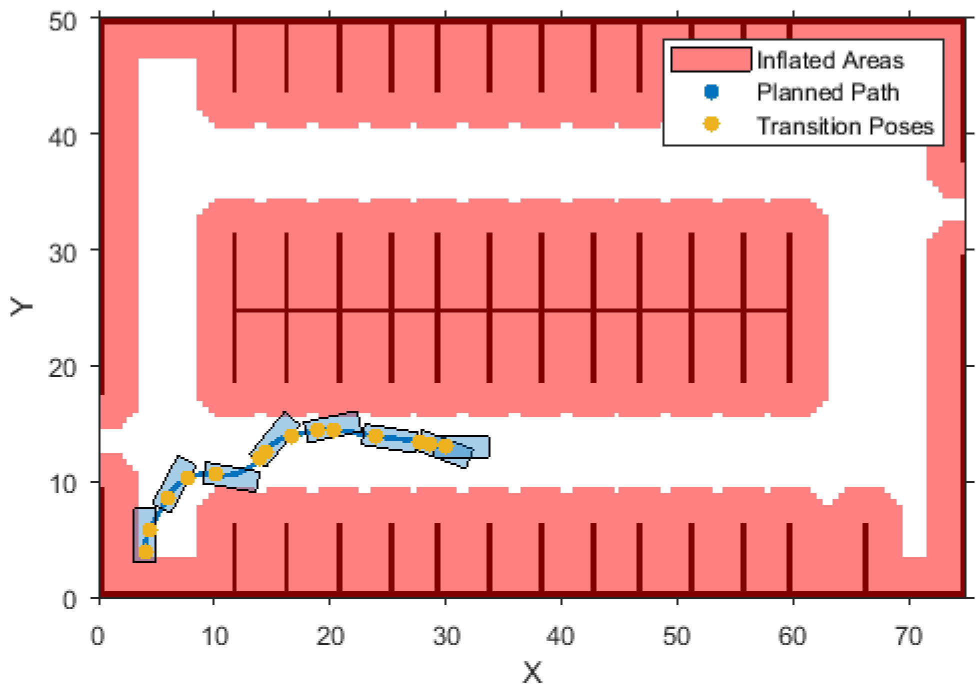
Interpolate the transition poses along the path.

```
transitionPoses = interpolate(refPath);
```

Plot the planned path and the transition poses on the costmap.

```
hold on
plot(refPath,'DisplayName','Planned Path')
scatter(transitionPoses(:,1),transitionPoses(:,2),[],'filled', ...
```

```
    'DisplayName','Transition Poses')
hold off
```

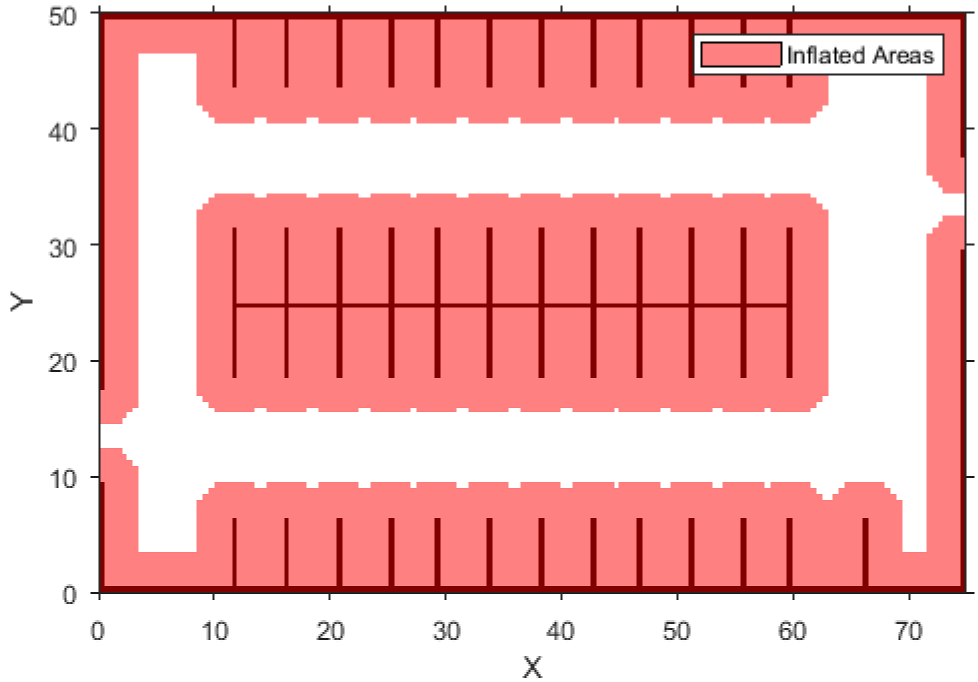


Plan Path and Interpolate Along Path

Plan a vehicle path through a parking lot by using the rapidly exploring random tree (RRT*) algorithm. Interpolate the poses of the vehicle at points along the path.

Load a costmap of a parking lot. Plot the costmap to see the parking lot and inflated areas for the vehicle to avoid.

```
data = load('parkingLotCostmap.mat');
costmap = data.parkingLotCostmap;
plot(costmap)
```



Define start and goal poses for the vehicle as [*x*, *y*, *Θ*] vectors. World units for the (*x*,*y*) locations are in meters. World units for the *Θ* orientation angles are in degrees.

```
startPose = [4, 4, 90]; % [meters, meters, degrees]
goalPose = [30, 13, 0];
```

Use a `pathPlannerRRT` object to plan a path from the start pose to the goal pose.

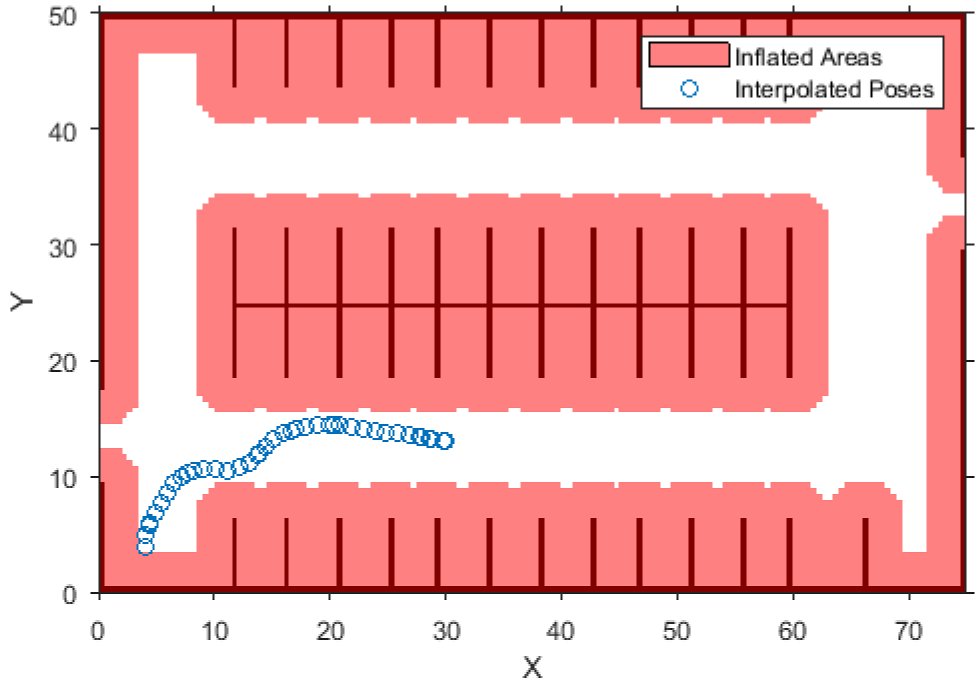
```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Interpolate the vehicle poses every 1 meter along the entire path.

```
lengths = 0 : 1 : refPath.Length;
poses = interpolate(refPath,lengths);
```

Plot the interpolated poses on the costmap.

```
plot(costmap)
hold on
scatter(poses(:,1),poses(:,2),'DisplayName','Interpolated Poses')
hold off
```



Input Arguments

`refPath` — Planned vehicle path
`driving.Path` object

Planned vehicle path, specified as a `driving.Path` object.

`lengths` — Points along length of path
real-valued vector

Points along the length of the path, specified as a real-valued vector. Values must be in the range from 0 to the length of the path, as determined by the `Length` property of `refPath`. The `interpolate` function interpolates poses at these specified points. `lengths` is in world units, such as meters.

Example: `poses = interpolate(refPath,0:0.1:refPath.Length)` interpolates poses every 0.1 meter along the entire length of the path.

Output Arguments

`poses` — Vehicle poses
m-by-3 matrix of [*x*, *y*, *Θ*] vectors

Vehicle poses along the path, returned as an *m*-by-3 matrix of [*x*, *y*, *Θ*] vectors. *m* is the number of returned poses.

x and *y* specify the location of the vehicle in world units, such as meters. *Θ* specifies the orientation angle of the vehicle in degrees.

`poses` always includes the transition poses, even if you interpolate only at specified points along the path. If you do not specify the `lengths` input argument, then `poses` includes only the transition poses.

`directions` — Motion directions
m-by-1 vector of 1s (forward motion) and –1s (reverse motion)

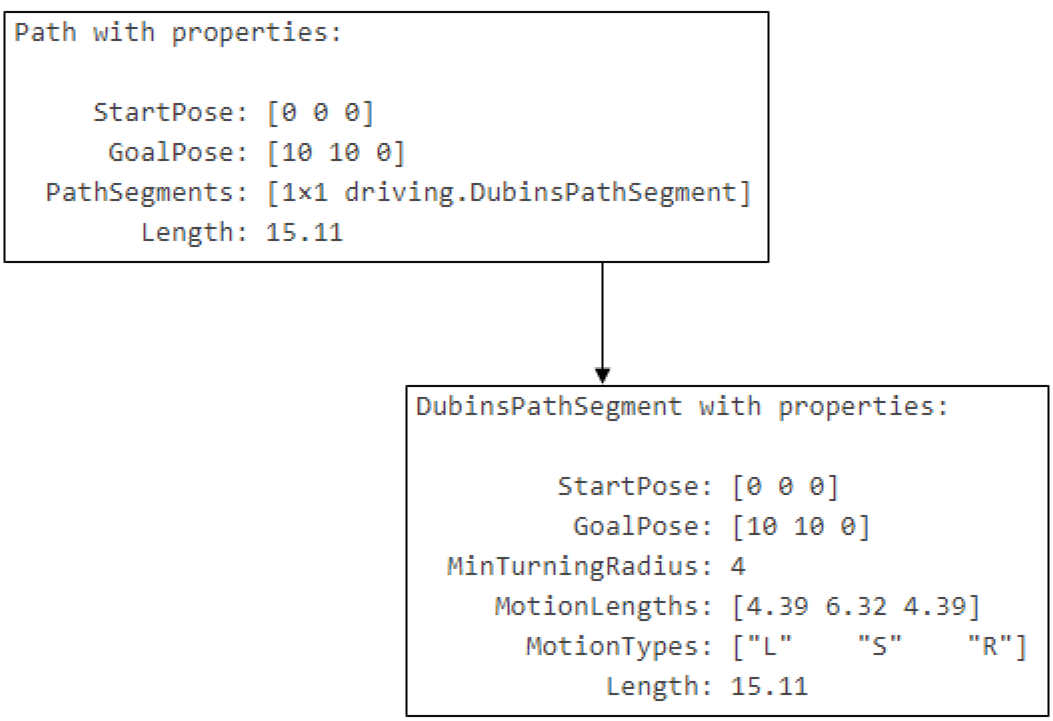
Motion directions of vehicle poses, returned as an *m*-by-1 vector of 1s (forward motion) and –1s (reverse motion). *m* is the number of returned poses. Each element of `directions` corresponds to a row of `poses`.

More About

Transition Poses

A path is composed of multiple segments that are combinations of motions (for example, left turn, straight, and right turn). Transition poses are vehicle poses corresponding to the end of one motion and the beginning of another motion. They represent points along the path corresponding to a change in the direction or orientation of the vehicle. The `interpolate` function always returns transition poses, even if you interpolate only at specified points along the path.

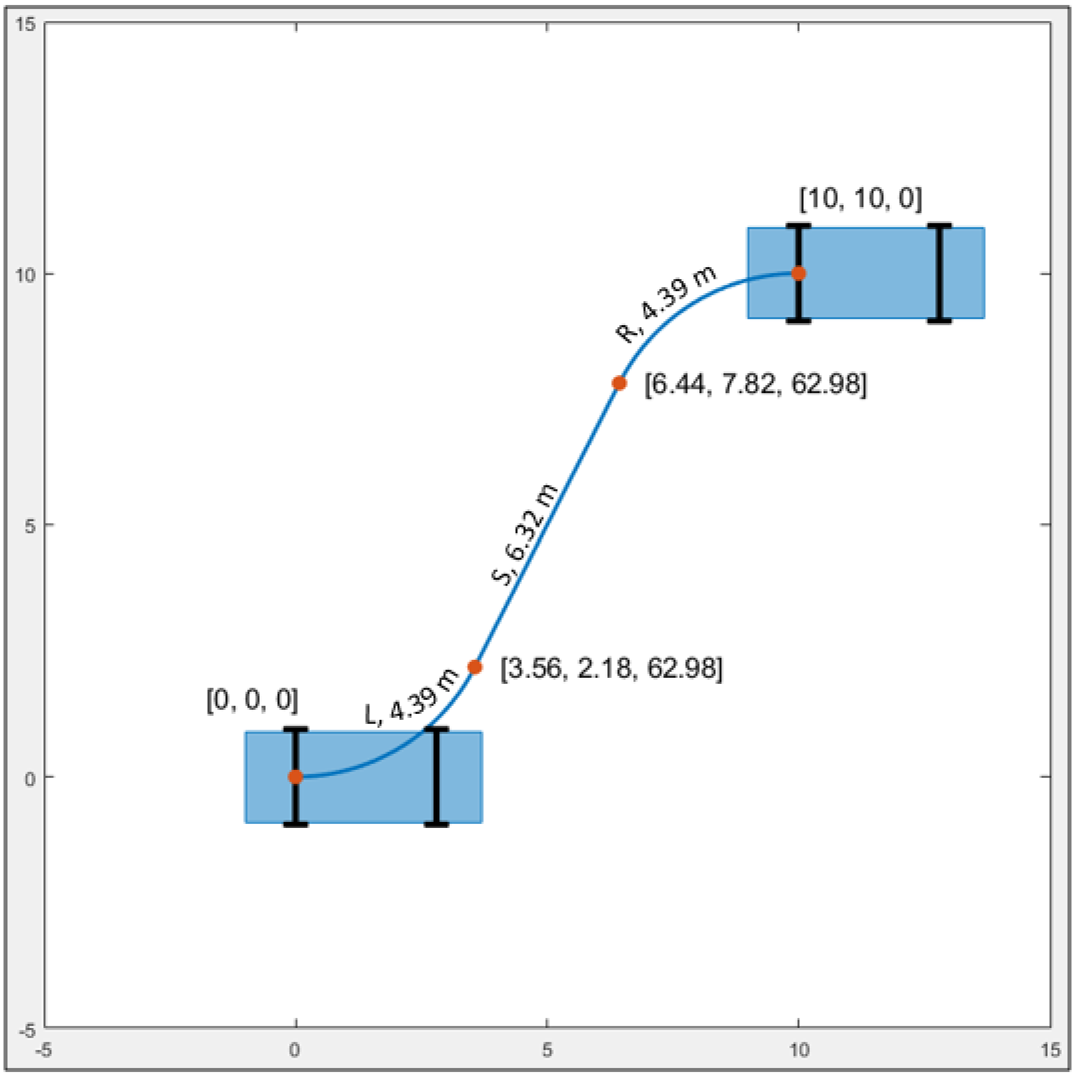
The path length between transition poses is given by the `MotionLengths` property of the path segments. For example, consider the following path, which is a `driving.Path` object composed of a single Dubins path segment. This segment consists of three motions, as described by the `MotionLengths` and `MotionTypes` properties of the segment.



The `interpolate` function interpolates the following transition poses in this order:

1 The initial pose of the vehicle, `StartPose`.
2 The pose after the vehicle turns left (`"L"`) for 4.39 meters at its maximum steering angle.
3 The pose after the vehicle goes straight (`"S"`) for 6.32 meters.
4 The pose after the vehicle turns right (`"R"`) for 4.39 meters at its maximum steering angle. This pose is also the goal pose, because it is the last pose of the entire path.

The plot shows these transition poses, which are [*x*, *y*, *Θ*] vectors. *x* and *y* specify the location of the vehicle in world units, such as meters. *Θ* specifies the orientation angle of the vehicle in degrees.



Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`checkPathValidity` | `smoothPathSpline`

Objects
`driving.Path` | `pathPlannerRRT`

Topics
"Automated Parking Valet"

Introduced in R2018b

driving.DubinsPathSegment

Dubins path segment

Description

A `driving.DubinsPathSegment` object represents a segment of a planned vehicle path that was connected using the Dubins connection method [1]. A Dubins path segment is composed of a sequence of three motions. Each motion is one of these types:

- Straight
- Left turn at the maximum steering angle of the vehicle
- Right turn at the maximum steering angle of the vehicle

A vehicle path composed of Dubins path segments allows motion in the forward direction only.

The `driving.DubinsPathSegment` objects that represent a path are stored in the `PathSegments` property of a `driving.Path` object. These paths are planned by a `pathPlannerRRT` object whose `ConnectionMethod` property is set to `'Dubins'`.

Properties

`StartPose` — Initial pose of vehicle
[*x*, *y*, *Θ*] vector

This property is read-only.

Initial pose of the vehicle at the start of the path segment, specified as an [*x*, *y*, *Θ*] vector. *x* and *y* are in world units, such as meters. *Θ* is in degrees.

`GoalPose` — Goal pose of vehicle
[*x*, *y*, *Θ*] vector

This property is read-only.

Goal pose of the vehicle at the end of the path segment, specified as an [*x*, *y*, *Θ*] vector. *x* and *y* are in world units, such as meters. *Θ* is in degrees.

`MinTurningRadius` — Minimum turning radius of vehicle
positive real scalar

This property is read-only.

Minimum turning radius of the vehicle, in world units, specified as a positive real scalar. This value corresponds to the radius of the turning circle at the maximum steering angle of the vehicle.

`MotionLengths` — Length of each motion
three-element real-valued vector

This property is read-only.

Length of each motion in the path segment, in world units, specified as a three-element real-valued vector. Each motion length corresponds to a motion type specified in `MotionTypes`.

`MotionTypes` — Type of each motion
three-element string array

This property is read-only.

Type of each motion in the path segment, specified as a three-element string array. Valid values are shown in this table.

Motion Type	Description
"S"	Straight
"L"	Left turn at the maximum steering angle of the vehicle
"R"	Right turn at the maximum steering angle of the vehicle

Each motion type corresponds to a motion length specified in `MotionLengths`.

Example: `["R" "S" "R"]`

`Length` — Length of path segment
positive real scalar

This property is read-only.

Length of the path segment, in world units, specified as a positive real scalar.

References

[1] Shkel, Andrei M., and Vladimir Lumelsky. "Classification of the Dubins Set." *Robotics and Autonomous Systems*. Vol. 34, Number 4, 2001, pp. 179–202.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only one-dimensional indexing is supported.

See Also

Objects
`driving.Path` | `driving.ReedsSheppPathSegment` | `pathPlannerRRT`

Topics
"Automated Parking Valet"

Introduced in R2018b

driving.ReedsSheppPathSegment

Reeds-Shepp path segment

Description

A `driving.ReedsSheppPathSegment` object represents a segment of a planned vehicle path that was connected using the Reeds-Shepp connection method [1]. A Reeds-Shepp path segment is composed of a sequence of three to five motions. Each motion is one of these types:

- Straight (forward or reverse)
- Left turn at the maximum steering angle of the vehicle (forward or reverse)
- Right turn at the maximum steering angle of the vehicle (forward or reverse)

The `driving.ReedsSheppPathSegment` objects that represent a path are stored in the `PathSegments` property of a `driving.Path` object. These paths are planned by a `pathPlannerRRT` object whose `ConnectionMethod` property is set to `'Dubins'`.

Properties

`StartPose` — Initial pose of vehicle
[*x*, *y*, *Θ*] vector

This property is read-only.

Initial pose of the vehicle at the start of the path segment, specified as an [*x*, *y*, *Θ*] vector. *x* and *y* are in world units, such as meters. *Θ* is in degrees.

`GoalPose` — Goal pose of vehicle
[*x*, *y*, *Θ*] vector

This property is read-only.

Goal pose of the vehicle at the end of the path segment, specified as an [*x*, *y*, *Θ*] vector. *x* and *y* are in world units, such as meters. *Θ* is in degrees.

`MinTurningRadius` — Minimum turning radius of vehicle
positive real scalar

This property is read-only.

Minimum turning radius of the vehicle, in world units, specified as a positive real scalar. This value corresponds to the radius of the turning circle at the maximum steering angle of the vehicle.

`MotionLengths` — Length of each motion
five-element real-valued vector

This property is read-only.

Length of each motion in the path segment, in world units, specified as a five-element real-valued vector. Each motion length corresponds to a motion type specified in `MotionTypes` and a motion direction specified in `MotionDirections`.

When a path segment requires fewer than five motions, the remaining `MotionLengths` elements are set to `0`. The remaining `MotionTypes` elements are set to `"N"` (no motion).

`MotionTypes` — Type of each motion
five-element string array

This property is read-only.

Type of each motion in the path segment, specified as a five-element string array. Valid values are shown in this table.

Motion Type	Description
`"S"`	Straight (forward or reverse)
`"L"`	Left turn at the maximum steering angle of the vehicle (forward or reverse)
`"R"`	Right turn at the maximum steering angle of the vehicle (forward or reverse)
`"N"`	No motion

`MotionTypes` contains a minimum of three motions, specified as a combination of `"S"`, `"L"`, and `"R"` elements. If a path segment has fewer than five motions, the remaining elements of `MotionTypes` are `"N"` (no motion).

Each motion type corresponds to a motion length specified in `MotionLengths` and a motion direction specified in `MotionDirections`.

Example: `["R" "S" "R" "L" "N"]`

`MotionDirections` — Direction of each motion
five-element vector of 1s (forward motion) and –1s (reverse motion)

This property is read-only.

Direction of each motion in the path segment, specified as a five-element vector of 1s (forward motion) and –1s (reverse motion). Each motion direction corresponds to a motion length specified in `MotionLengths` and a motion type specified in `MotionTypes`.

When no motion occurs, that is, when a `MotionTypes` value is `"N"`, then the corresponding `MotionDirections` element is `1`.

Example: `[-1 1 -1 1 1]`

`Length` — Length of path segment
positive real scalar

This property is read-only.

Length of the path segment, in world units, specified as a positive real scalar.

References

[1] Reeds, J. A., and L. A. Shepp. "Optimal Paths for a Car That Goes Both Forwards and Backwards." *Pacific Journal of Mathematics*. Vol. 145, Number 2, 1990, pp. 367–393.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only one-dimensional indexing is supported.

See Also

Objects
`driving.DubinsPathSegment` | `driving.Path` | `pathPlannerRRT`

Topics
"Automated Parking Valet"

Introduced in R2018b

drivingScenario

Create driving scenario

Description

The `drivingScenario` object represents a 3-D arena containing roads, vehicles, pedestrians, and other aspects of a driving scenario. Use this object to model realistic traffic scenarios and to generate synthetic detections for testing controllers or sensor fusion algorithms.

- To add roads, use the `road` function. To specify lanes in the roads, create a `lanespec` object. You can also import roads from a third-party road network by using the `roadNetwork` function.
- To add actors (cars, pedestrians, bicycles, and so on), use the `actor` function. To add actors with properties designed specifically for vehicles, use the `vehicle` function. All actors, including vehicles, are modeled as cuboids (box shapes).
- To simulate a scenario, call the `advance` function in a loop, which advances the simulation one time step at a time.

You can also create driving scenarios interactively by using the **Driving Scenario Designer** app. In addition, you can export `drivingScenario` objects from the app to produce scenario variations for use in either the app or in Simulink. For more details, see “Create Driving Scenario Variations Programmatically”.

Creation

Syntax

```
scenario = drivingScenario
scenario = drivingScenario(Name,Value)
```

Description

`scenario = drivingScenario` creates an empty driving scenario.

`scenario = drivingScenario(Name,Value)` sets the `SampleTime` and `StopTime` properties using name-value pairs. For example, `drivingScenario('SampleTime',0.1','StopTime',10)` samples the scenario every 0.1 seconds for 10 seconds. Enclose each property name in quotes.

Properties

`SampleTime` — Time interval between scenario simulation steps
`0.01` (default) | positive real scalar

Time interval between scenario simulation steps, specified as a positive real scalar. Units are in seconds.

Example: `1.5`

StopTime — End time of simulation
Inf (default) | positive real scalar

End time of simulation, specified as a positive real scalar. Units are in seconds. The default StopTime of Inf causes the simulation to end when the first actor reaches the end of its trajectory.

Example: 60.0

SimulationTime — Current time of simulation
positive real scalar

This property is read-only.

Current time of the simulation, specified as a positive real scalar. To reset the time to zero, call the restart function. Units are in seconds.

IsRunning — Simulation state
true | false

This property is read-only.

Simulation state, specified as true or false. If the simulation is running, IsRunning is true.

Actors — Actors and vehicles contained in scenario
heterogeneous array of Actor and Vehicle objects

This property is read-only.

Actors and vehicles contained in the scenario, specified as a heterogeneous array of Actor and Vehicle objects. To add actors and vehicles to a driving scenario, use the actor and vehicle functions.

GeoReference — Geographic coordinates of road network origin
three-element numeric row vector of form [lat, lon, alt]

This property is read-only.

Geographic coordinates of the road network origin, specified as a three-element numeric row vector of the form [lat, lon, alt], where:

- lat is the latitude of the coordinate in degrees.
- lon is the longitude of the coordinate in degrees.
- alt is the altitude of the coordinate in meters.

These values are with respect to the WGS84 reference ellipsoid, which is a standard ellipsoid used by GPS data.

The road network origin corresponds to the first set of geographic coordinates specified as input to the roadNetwork function. If you specify to import roads by region, then this point corresponds to the center point of the specified rectangular region.

In drivingScenario objects generated from the **Driving Scenario Designer** app, if you imported the road network from a map, then the road network origin corresponds to the first (or only) set of geographic coordinates specified.

By specifying these coordinates as the origin in the `latlon2local` function, you can the convert geographic coordinates of a driving route into the local coordinates of a driving scenario. Then, you can specify this converted route as a vehicle trajectory in the scenario.

Dependencies

To enable this property, import roads from the HERE HD Live Map[3] web service into a driving scenario by using the `roadNetwork` function or the **Driving Scenario Designer** app.

Object Functions

Scenarios

advance	Advance driving scenario simulation by one time step
plot	Create driving scenario plot
record	Run driving scenario and record actor states
restart	Restart driving scenario simulation from beginning
updatePlots	Update driving scenario plots
export	Export road network to OpenDRIVE

Actors

actor	Add actor to driving scenario
actorPoses	Positions, velocities, and orientations of actors in driving scenario
actorProfiles	Physical and radar characteristics of actors in driving scenario
vehicle	Add vehicle to driving scenario
chasePlot	Ego-centric projective perspective plot
trajectory	Create actor or vehicle trajectory in driving scenario
targetPoses	Target positions and orientations relative to ego vehicle
targetOutlines	Outlines of targets viewed by actor
driving.scenario.targetsToEgo	Convert target poses from scenario to ego coordinates
driving.scenario.targetsToScenario	Convert target poses from ego to scenario coordinates

Roads

road	Add road to driving scenario
roadNetwork	Add road network to driving scenario
roadBoundaries	Get road boundaries
driving.scenario.roadBoundariesToEgo	Convert road boundaries to ego vehicle coordinates

Lanes

currentLane	Get current lane of actor
lanespec	Create road lane specifications
laneMarking	Create road lane marking object
laneMarkingVertices	Lane marking vertices and faces in driving scenario
laneBoundaries	Get lane boundaries of actor lane
clothoidLaneBoundary	Clothoid-shaped lane boundary model
computeBoundaryModel	Compute lane boundary points from clothoid lane boundary model
laneType	Create road lane type object

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.

Examples

Create Driving Scenario with Multiple Actors and Roads

Create a driving scenario containing a curved road, two straight roads, and two actors: a car and a bicycle. Both actors move along the road for 60 seconds.

Create the driving scenario object.

```
scenario = drivingScenario('SampleTime',0.1','StopTime',60);
```

Create the curved road using road center points following the arc of a circle with an 800-meter radius. The arc starts at 0°, ends at 90°, and is sampled at 5° increments.

```
angs = [0:5:90]';
R = 800;
roadcenters = R*[cosd(angs) sind(angs) zeros(size(angs))];
roadwidth = 10;
road(scenario,roadcenters,roadwidth);
```

Add two straight roads with the default width, using road center points at each end.

```
roadcenters = [700 0 0; 100 0 0];
road(scenario,roadcenters)
```

```
ans = 
  Road with properties:

           Name: ""
         RoadID: 2
    RoadCenters: [2x3 double]
      RoadWidth: 6
      BankAngle: [2x1 double]
```

```
roadcenters = [400 400 0; 0 0 0];
road(scenario,roadcenters)
```

```
ans = 
  Road with properties:

           Name: ""
         RoadID: 3
    RoadCenters: [2x3 double]
      RoadWidth: 6
      BankAngle: [2x1 double]
```

Get the road boundaries.

```
rbdry = roadBoundaries(scenario);
```

Add a car and a bicycle to the scenario. Position the car at the beginning of the first straight road.

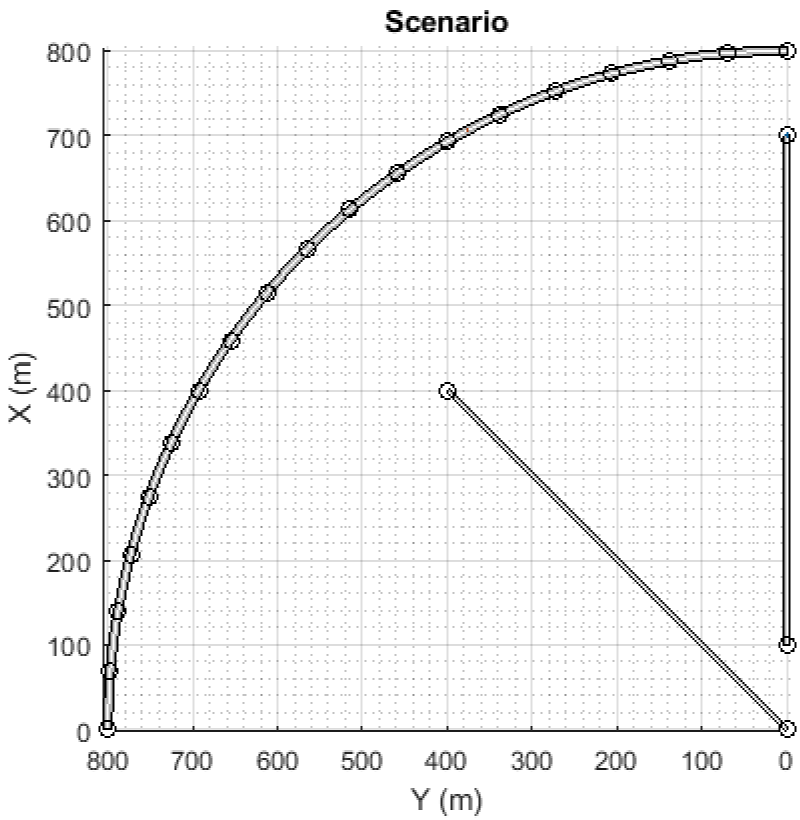
```
car = vehicle(scenario,'ClassID',1,'Position',[700 0 0], ...
    'Length',3,'Width',2,'Height',1.6);
```

Position the bicycle farther down the road.

```
bicycle = actor(scenario,'ClassID',3,'Position',[706 376 0]', ...
    'Length',2,'Width',0.45,'Height',1.5);
```

Plot the scenario.

```
plot(scenario,'Centerline','on','RoadCenters','on');
title('Scenario');
```



Display the actor poses and profiles.

```
poses = actorPoses(scenario)
```

```
poses=2×1 struct array with fields:
    ActorID
    Position
    Velocity
    Roll
    Pitch
    Yaw
    AngularVelocity
```

```
profiles = actorProfiles(scenario)
```

```
profiles=2×1 struct array with fields:
    ActorID
    ClassID
    Length
```

```
    Width
    Height
    OriginOffset
    MeshVertices
    MeshFaces
    RCSPattern
    RCSAzimuthAngles
    RCSElevationAngles
```

Show Target Outlines in Driving Scenario Simulation

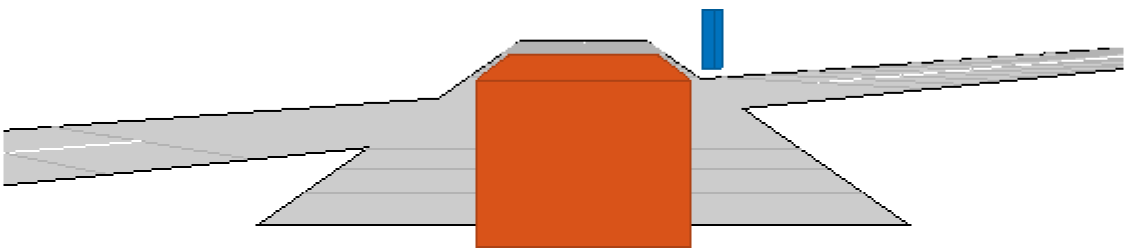
Create a driving scenario and show how target outlines change as the simulation advances.

Create a driving scenario consisting of two intersecting straight roads. The first road segment is 45 meters long. The second straight road is 32 meters long and intersects the first road. A car traveling at 12.0 meters per second along the first road approaches a running pedestrian crossing the intersection at 2.0 meters per second.

```
scenario = drivingScenario('SampleTime',0.1,'StopTime',1);
road(scenario,[-10 0 0; 45 -20 0]);
road(scenario,[-10 -10 0; 35 10 0]);
ped = actor(scenario,'ClassID',4,'Length',0.4,'Width',0.6,'Height',1.7);
car = vehicle(scenario,'ClassID',1);
pedspeed = 2.0;
carspeed = 12.0;
trajectory(ped,[15 -3 0; 15 3 0],pedspeed);
trajectory(car,[-10 -10 0; 35 10 0],carspeed);
```

Create an ego-centric chase plot for the vehicle.

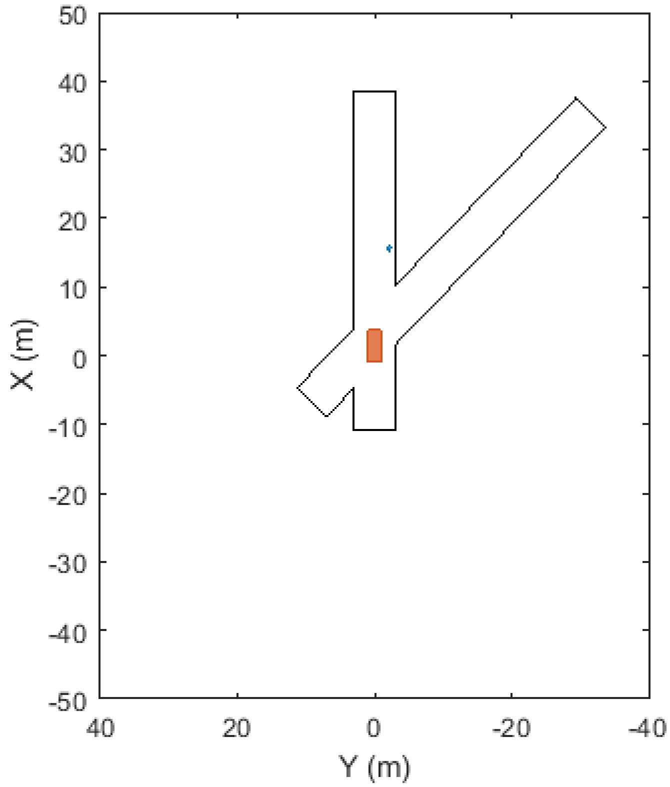
```
chasePlot(car,'Centerline','on')
```



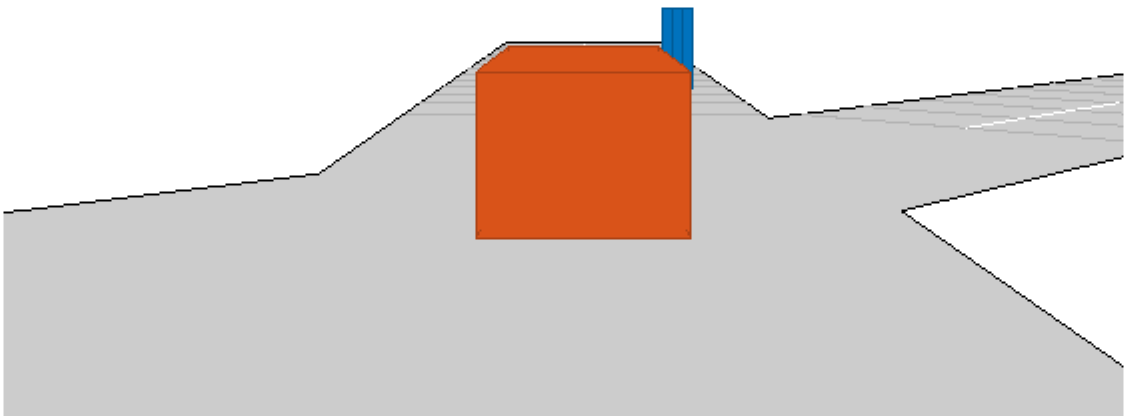
Create an empty bird's-eye plot and add an outline plotter and lane boundary plotter. Then, run the simulation. At each simulation step:

- Update the chase plot to display the road boundaries and target outlines.
- Update the bird's-eye plot to display the updated road boundaries and target outlines. The plot perspective is always with respect to the ego vehicle.

```
bepPlot = birdsEyePlot('XLim',[-50 50],'YLim',[-40 40]);
outlineplotter = outlinePlotter(bepPlot);
laneplotter = laneBoundaryPlotter(bepPlot);
legend('off')

while advance(scenario)
    rb = roadBoundaries(car);
    [position,yaw,length,width,originOffset,color] = targetOutlines(car);
    plotLaneBoundary(laneplotter,rb)
    plotOutline(outlineplotter,position,yaw,length,width, ...
        'OriginOffset',originOffset,'Color',color)
    pause(0.01)
end
```



50
40
30
20
10
0
-10
-20
-30
-40
-50
X (m)
40
20
0
-20
-40
Y (m)



Generate Object and Lane Boundary Detections

Create a driving scenario containing an ego vehicle and a target vehicle traveling along a three-lane road. Detect the lane boundaries by using a vision detection generator.

```
scenario = drivingScenario;
```

Create a three-lane road by using lane specifications.

```
roadCenters = [0 0 0; 60 0 0; 120 30 0];
lspc = lanespec(3);
road(scenario,roadCenters,'Lanes',lspc);
```

Specify that the ego vehicle follows the center lane at 30 m/s.

```
egovehicle = vehicle(scenario,'ClassID',1);
egopath = [1.5 0 0; 60 0 0; 111 25 0];
egospeed = 30;
trajectory(egovehicle,egopath,egospeed);
```

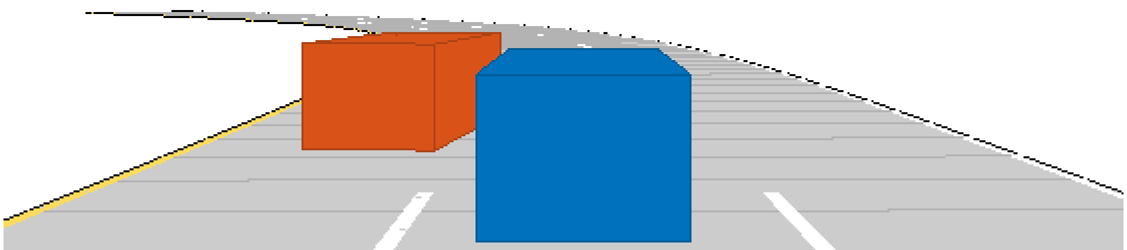
Specify that the target vehicle travels ahead of the ego vehicle at 40 m/s and changes lanes close to the ego vehicle.

```
targetcar = vehicle(scenario,'ClassID',1);
targetpath = [8 2; 60 -3.2; 120 33];
```

```
targetspeed = 40;
trajectory(targetcar,targetpath,targetspeed);
```

Display a chase plot for a 3-D view of the scenario from behind the ego vehicle.

```
chasePlot(egovehicle)
```



Create a vision detection generator that detects lanes and objects. The pitch of the sensor points one degree downward.

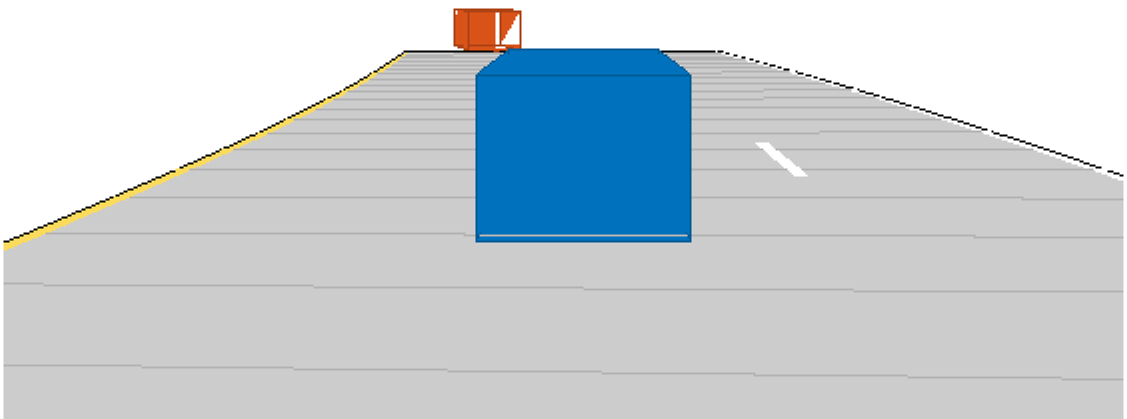
```
visionSensor = visionDetectionGenerator('Pitch',1.0);
visionSensor.DetectorOutput = 'Lanes and objects';
visionSensor.ActorProfiles = actorProfiles(scenario);
```

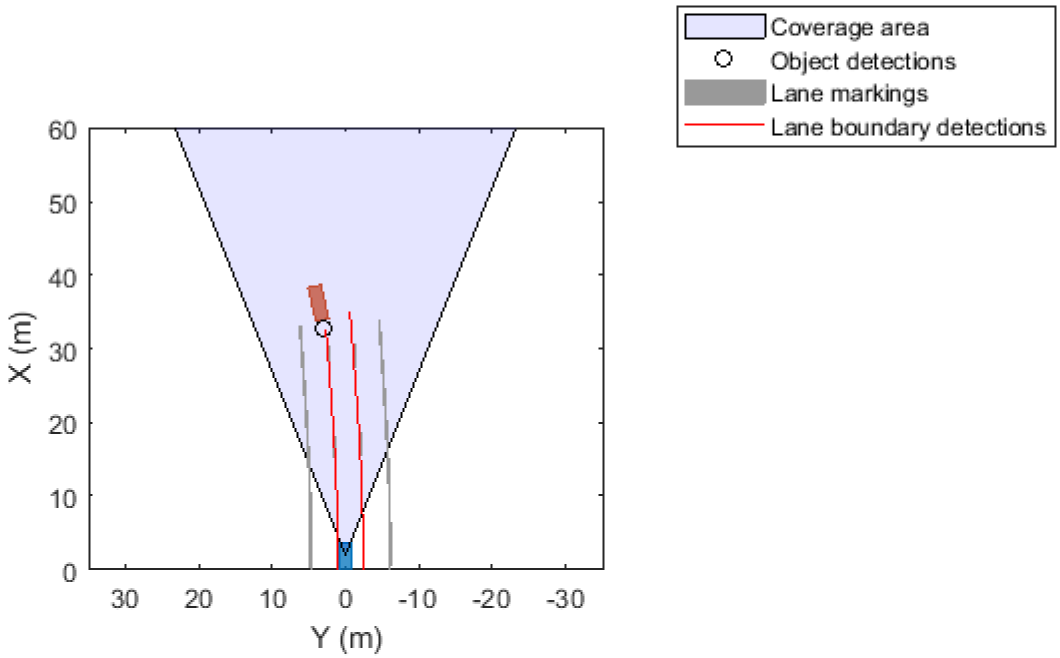
Run the simulation.

1. Create a bird's-eye plot and the associated plotters.
2. Display the sensor coverage area.
3. Display the lane markings.
4. Obtain ground truth poses of targets on the road.
5. Obtain ideal lane boundary points up to 60 m ahead.
6. Generate detections from the ideal target poses and lane boundaries.
7. Display the outline of the target.
8. Display object detections when the object detection is valid.

9 Display the lane boundary when the lane detection is valid.

```
bep = birdsEyePlot('XLim',[0 60],'YLim',[-35 35]);
caPlotter = coverageAreaPlotter(bep,'DisplayName','Coverage area', ...
    'FaceColor','blue');
detPlotter = detectionPlotter(bep,'DisplayName','Object detections');
lmPlotter = laneMarkingPlotter(bep,'DisplayName','Lane markings');
lbPlotter = laneBoundaryPlotter(bep,'DisplayName', ...
    'Lane boundary detections','Color','red');
olPlotter = outlinePlotter(bep);
plotCoverageArea(caPlotter,visionSensor.SensorLocation,...
    visionSensor.MaxRange,visionSensor.Yaw, ...
    visionSensor.FieldOfView(1));
while advance(scenario)
    [lmv,lmf] = laneMarkingVertices(egovehicle);
    plotLaneMarking(lmPlotter,lmv,lmf)
    tgtpose = targetPoses(egovehicle);
    lookaheadDistance = 0:0.5:60;
    lb = laneBoundaries(egovehicle,'XDistance',lookaheadDistance,'LocationType','inner');
    [obdets,nobdets,obValid,lb_dets,nlb_dets,lbValid] = ...
        visionSensor(tgtpose,lb,scenario.SimulationTime);
    [objposition,objyaw,objlength,objwidth,objoriginOffset,color] = targetOutlines(egovehicle);
    plotOutline(olPlotter,objposition,objyaw,objlength,objwidth, ...
        'OriginOffset',objoriginOffset,'Color',color)
    if obValid
        detPos = cellfun(@(d)d.Measurement(1:2),obdets,'UniformOutput',false);
        detPos = vertcat(zeros(0,2),cell2mat(detPos')');
        plotDetection(detPlotter,detPos)
    end
    if lbValid
        plotLaneBoundary(lbPlotter,vertcat(lb_dets.LaneBoundaries))
    end
end
```





Algorithms

Specify Actor Motion in Driving Scenarios

To specify the motion of actors in a driving scenario, you can either define trajectories for the actors or specify their motion manually.

Specify Motion Using Trajectory

The `trajectory` function determines actor pose properties based on a set of waypoints and the speeds at which the actor travels between those waypoints. Actor pose properties are position, velocity, roll, pitch, yaw, and angular velocity. With this approach, motion is defined by speed, not velocity, because the trajectory determines the direction of motion.

The actor moves along the trajectory each time the `advance` function is called. You can manually update actor pose properties at any time during a simulation. However, these properties are overwritten with updated values at the next call to `advance`.

Specify Motion Manually

When you specify actor motion manually, setting the velocity or angular velocity properties does not automatically move the actor in successive calls to the `advance` function. Therefore, you must use your own motion model to update the position, velocity, and other pose parameters at each simulation time step.

See Also

Apps
Driving Scenario Designer

Objects
`lidarPointCloudGenerator` | `multiObjectTracker` | `radarDetectionGenerator` | `visionDetectionGenerator`

Topics
"Create Driving Scenario Variations Programmatically"
"Create Driving Scenario Programmatically"
"Define Road Layouts Programmatically"
"Create Actor and Vehicle Trajectories Programmatically"
"Scenario Generation from Recorded Vehicle Data"
"Coordinate Systems in Automated Driving Toolbox"

Introduced in R2017a

advance

Advance driving scenario simulation by one time step

Syntax

```
isRunning = advance(scenario)
```

Description

`isRunning = advance(scenario)` advances a driving scenario simulation by one time step. To specify the step time, use the `SampleTime` property of the input `drivingScenario` object, `scenario`. The function returns the status, `isRunning`, of the simulation.

Examples

Advance Driving Scenario Simulation

Create a driving scenario. Use the default sample time of 0.01 second.

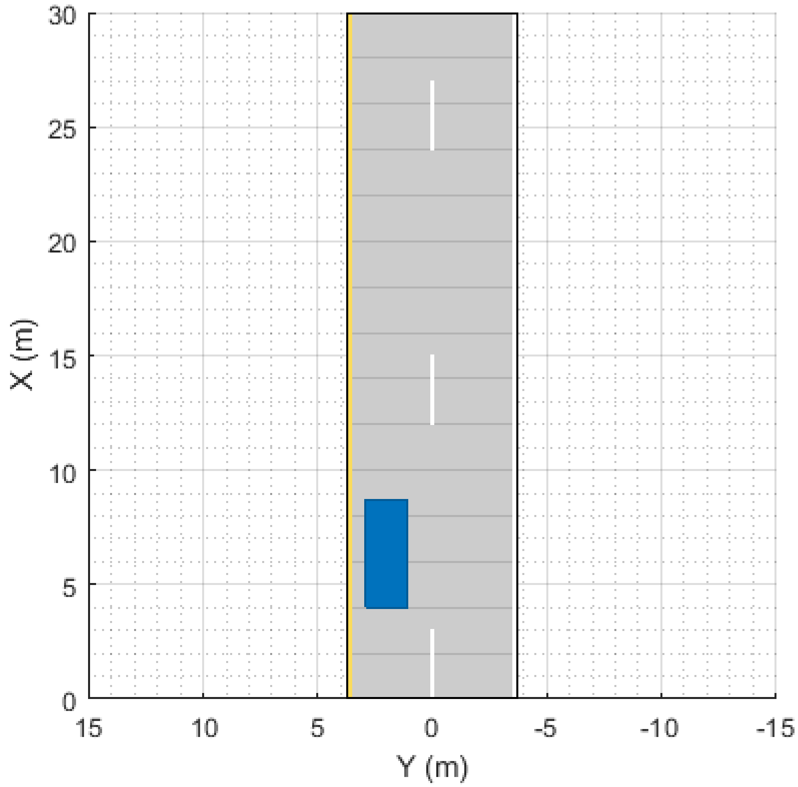
```
scenario = drivingScenario;
```

Add a straight, 30-meter road to the scenario. The road has two lanes.

```
roadCenters = [0 0; 30 0];
road(scenario,roadCenters,'Lanes',lanespec(2));
```

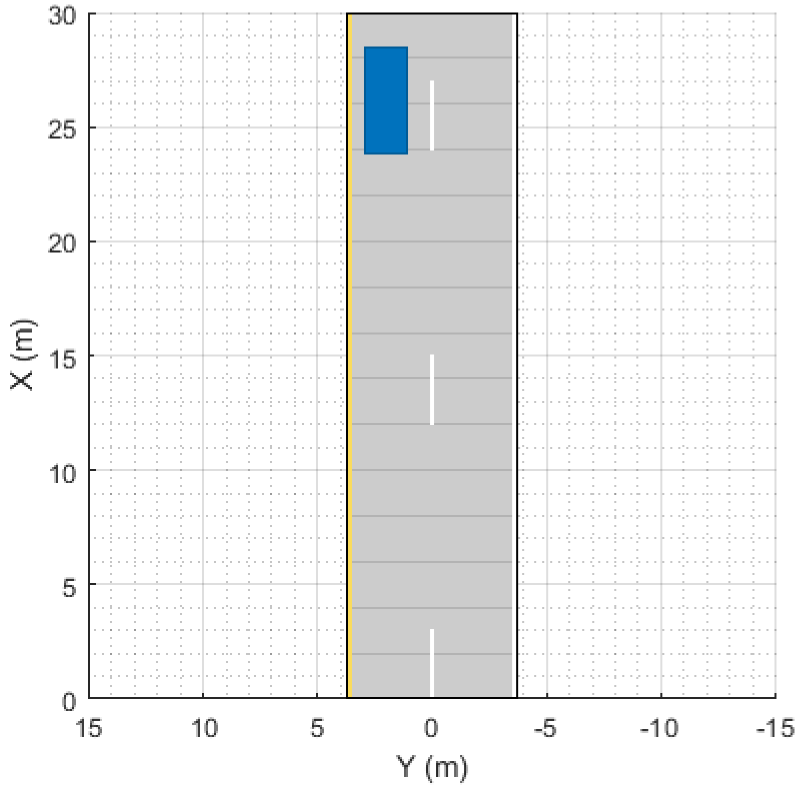
Add a vehicle that travels in the left lane at a constant speed of 30 meters per second. Plot the scenario before running the simulation.

```
v = vehicle(scenario,'ClassID',1);
waypoints = [5 2; 25 2];
speed = 30; % m/s
trajectory(v,waypoints,speed)

plot(scenario)
```



Call the `advance` function in a loop to advance the simulation one time step at a time. Pause every 0.01 second to observe the motion of the vehicle on the plot.

```
while advance(scenario)
    pause(0.01)
end
```



Show Target Outlines in Driving Scenario Simulation

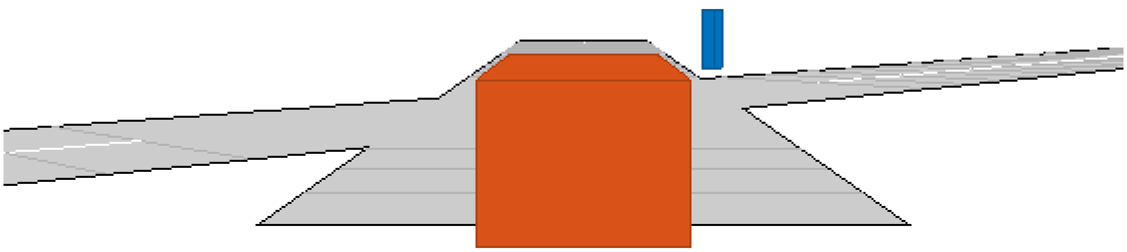
Create a driving scenario and show how target outlines change as the simulation advances.

Create a driving scenario consisting of two intersecting straight roads. The first road segment is 45 meters long. The second straight road is 32 meters long and intersects the first road. A car traveling at 12.0 meters per second along the first road approaches a running pedestrian crossing the intersection at 2.0 meters per second.

```
scenario = drivingScenario('SampleTime',0.1,'StopTime',1);
road(scenario,[-10 0 0; 45 -20 0]);
road(scenario,[-10 -10 0; 35 10 0]);
ped = actor(scenario,'ClassID',4,'Length',0.4,'Width',0.6,'Height',1.7);
car = vehicle(scenario,'ClassID',1);
pedspeed = 2.0;
carspeed = 12.0;
trajectory(ped,[15 -3 0; 15 3 0],pedspeed);
trajectory(car,[-10 -10 0; 35 10 0],carspeed);
```

Create an ego-centric chase plot for the vehicle.

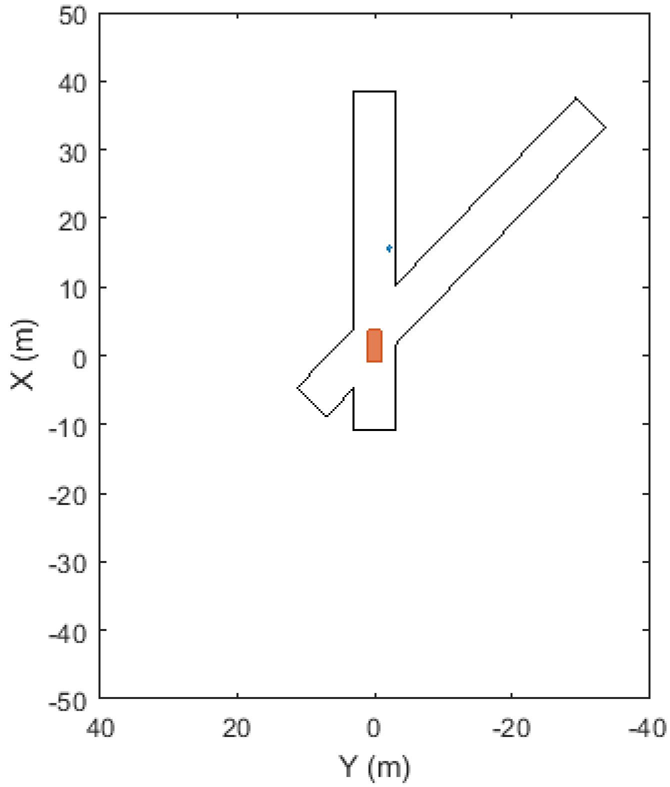
```
chasePlot(car,'Centerline','on')
```

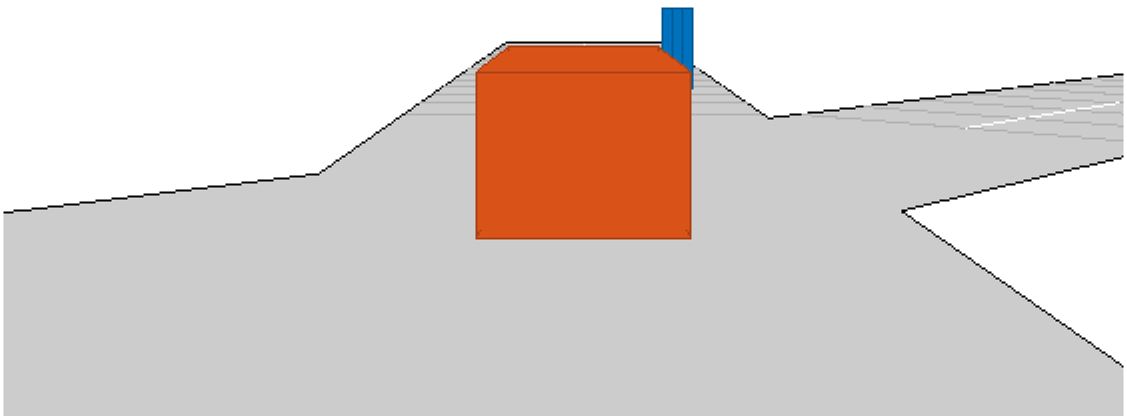


Create an empty bird's-eye plot and add an outline plotter and lane boundary plotter. Then, run the simulation. At each simulation step:

- Update the chase plot to display the road boundaries and target outlines.
- Update the bird's-eye plot to display the updated road boundaries and target outlines. The plot perspective is always with respect to the ego vehicle.

```
bepPlot = birdsEyePlot('XLim',[-50 50],'YLim',[-40 40]);
outlineplotter = outlinePlotter(bepPlot);
laneplotter = laneBoundaryPlotter(bepPlot);
legend('off')

while advance(scenario)
    rb = roadBoundaries(car);
    [position,yaw,length,width,originOffset,color] = targetOutlines(car);
    plotLaneBoundary(laneplotter,rb)
    plotOutline(outlineplotter,position,yaw,length,width, ...
        'OriginOffset',originOffset,'Color',color)
    pause(0.01)
end
```


50
40
30
20
10
0
-10
-20
-30
-40
-50
X (m)
40
20
0
-20
-40
Y (m)



Simulate Car Traveling on S-Curve

Simulate a driving scenario with one car traveling on an S-curve. Create and plot the lane boundaries.

Create the driving scenario with one road having an S-curve.

```
scenario = drivingScenario('StopTime',3);
roadcenters = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
```

Create the lanes and add them to the road.

```
lm = [laneMarking('Solid','Color','w'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Solid','Color','w')];
ls = lanespec(3,'Marking',lm);
road(scenario,roadcenters,'Lanes',ls);
```

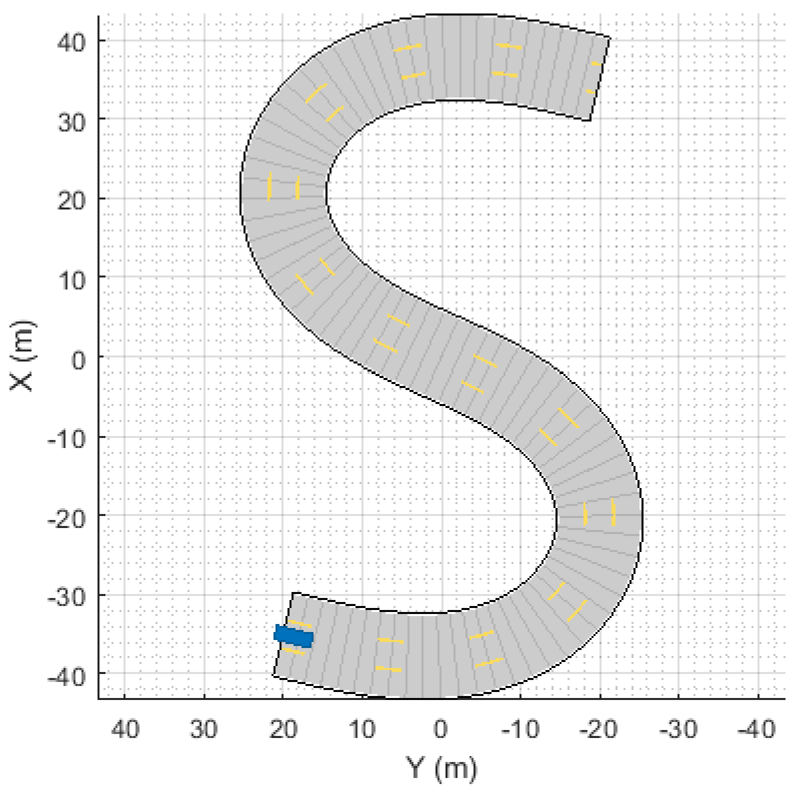
Add an ego vehicle and specify its trajectory from its waypoints. By default, the car travels at a speed of 30 meters per second.

```
car = vehicle(scenario, ...
    'ClassID',1, ...
    'Position',[-35 20 0]);
```

```
waypoints = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
trajectory(car,waypoints);
```

Plot the scenario and corresponding chase plot.

```
plot(scenario)
```

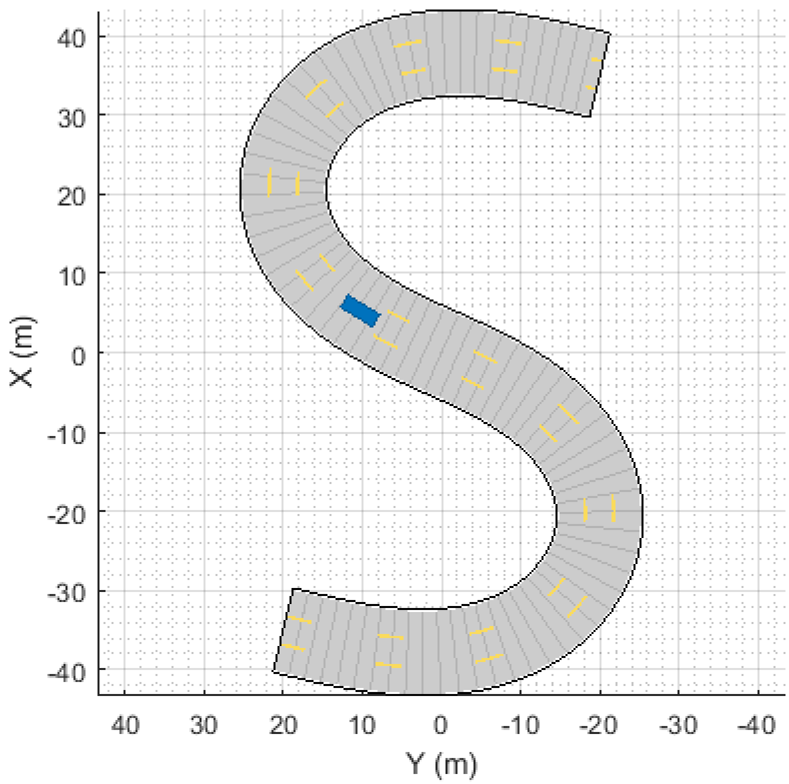


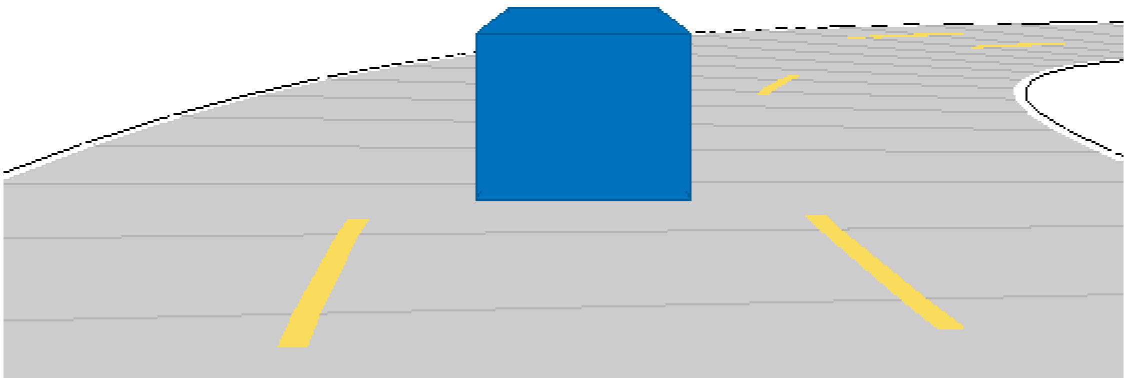

```
chasePlot(car)
```

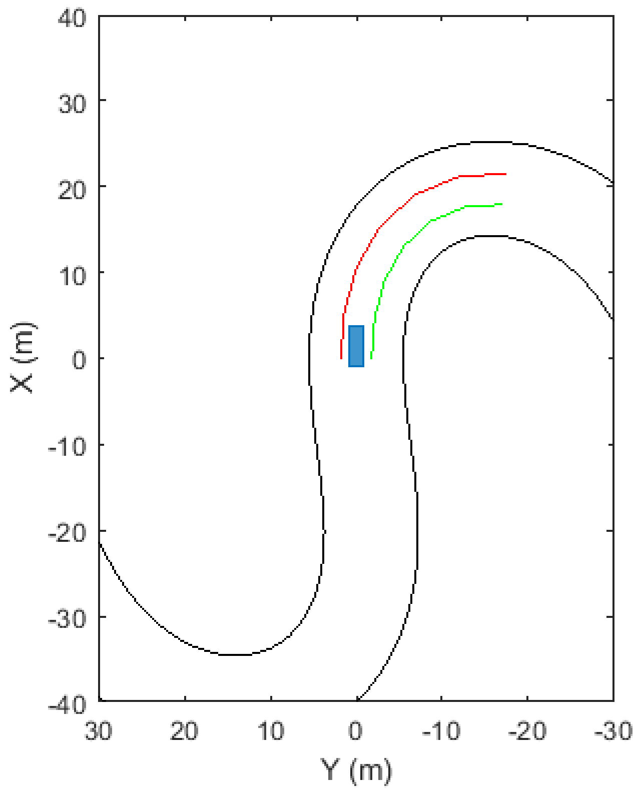
Run the simulation loop.

1. Initialize a bird's-eye plot and create an outline plotter, left-lane and right-lane boundary plotters, and a road boundary plotter.
2. Obtain the road boundaries and rectangular outlines.
3. Obtain the lane boundaries to the left and right of the vehicle.
4. Advance the simulation and update the plotters.

```
bep = birdsEyePlot('XLim',[-40 40],'YLim',[-30 30]);
olPlotter = outlinePlotter(bep);
lblPlotter = laneBoundaryPlotter(bep,'Color','r','LineStyle','-');
lbrPlotter = laneBoundaryPlotter(bep,'Color','g','LineStyle','-');
rbsEdgePlotter = laneBoundaryPlotter(bep);
legend('off');
while advance(scenario)
    rbs = roadBoundaries(car);
    [position,yaw,length,width,originOffset,color] = targetOutlines(car);
    lb = laneBoundaries(car,'XDistance',0:5:30,'LocationType','Center', ...
        'AllBoundaries',false);
    plotLaneBoundary(rbsEdgePlotter,rbs)
    plotLaneBoundary(lblPlotter,{lb(1).Coordinates})
    plotLaneBoundary(lbrPlotter,{lb(2).Coordinates})
    plotOutline(olPlotter,position,yaw,length,width, ...
        'OriginOffset',originOffset,'Color',color)
end
```


40
30
20
10
0
-10
-20
-30
-40
X (m)
40 30 20 10 0 -10 -20 -30 -40
Y (m)





Input Arguments

scenario — Driving scenario
drivingScenario object

Driving scenario, specified as a drivingScenario object.

Output Arguments

isRunning — Run state of simulation
1 | 0

Run state of the simulation, returned as logical 1 (true) or 0 (false).

- If isRunning is 1, the simulation is running.
- If isRunning is 0, the simulation has stopped running.

A simulation runs until at least one of these conditions are met:

- The simulation time exceeds the simulation stop time. To specify the stop time, use the StopTime property of scenario.
- Any actor or vehicle reaches the end of its assigned trajectory. The assigned trajectory is specified by the most recent call to the trajectory function.

The `advance` function updates actors and vehicles only if they have an assigned trajectory. To update actors and vehicles that have no assigned trajectories, you can set the `Position`, `Velocity`, `Roll`, `Pitch`, `Yaw`, or `AngularVelocity` properties at any time during simulation.

See Also

Objects
`drivingScenario`

Functions
`chasePlot` | `plot` | `record` | `restart` | `trajectory`

Topics
“Create Driving Scenario Programmatically”
“Create Actor and Vehicle Trajectories Programmatically”

Introduced in R2017a

export

Export road network to OpenDRIVE

Syntax

```
export(scenario,'OpenDRIVE',filename)
```

Description

`export(scenario,'OpenDRIVE',filename)` exports the roads, lanes, and junctions in a driving scenario to the OpenDRIVE 1.4H file format. There may be variations between the original scenario and the exported scenario. For details, see “Limitations” on page 4-294.

Examples

Export OpenStreetMap Road Network to OpenDRIVE File

Create a driving scenario.

```
inputScenario = drivingScenario;
```

Import a OpenStreetMap road network into the driving scenario. For more information about the osm file, see [1] on page 4-0 .

```
fileName = 'chicago.osm';
roadNetwork(inputScenario,'OpenStreetMap',fileName);
```

Export to OpenDRIVE file.

```
fileName = 'chicago.xodr';
export(inputScenario,'OpenDRIVE',fileName);
```

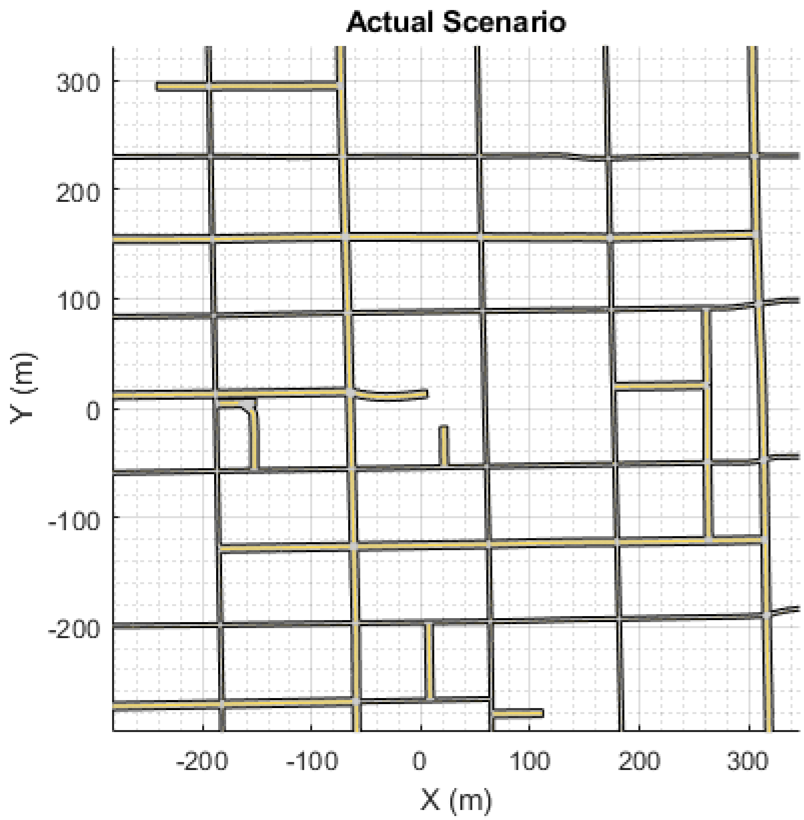
```
Warning: There may be minor variation between the actual driving scenario and the exported OpenDR
```

Read the exported OpenDRIVE file by using the `roadNetwork` function.

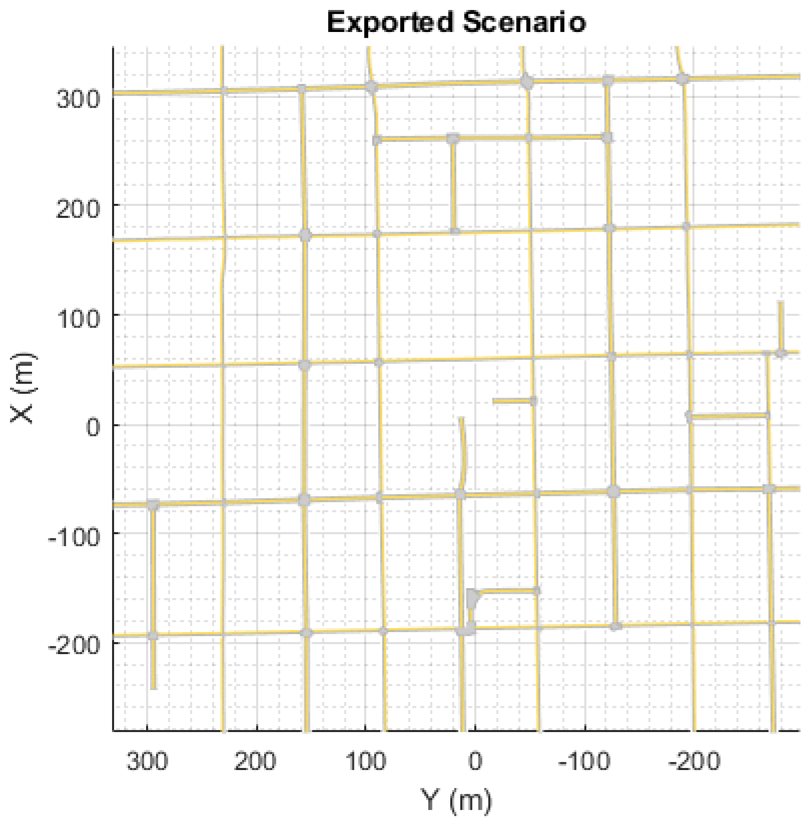
```
scenario = drivingScenario;
roadNetwork(scenario,'OpenDRIVE',fileName);
```

Plot the exported scenario. Notice that the display for the exported road network is flipped along the x and y dimensions and does not have the border lines.

```
figure
plot(inputScenario)
zoom(2);
title('Actual Scenario')
```




```
figure
plot(scenario)
zoom(2);
title('Exported Scenario')
```



Appendix

[1] The osm file is 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/.

Create S-Curve Road and Export to OpenDRIVE Format

Create the driving scenario with one road having an S-curve.

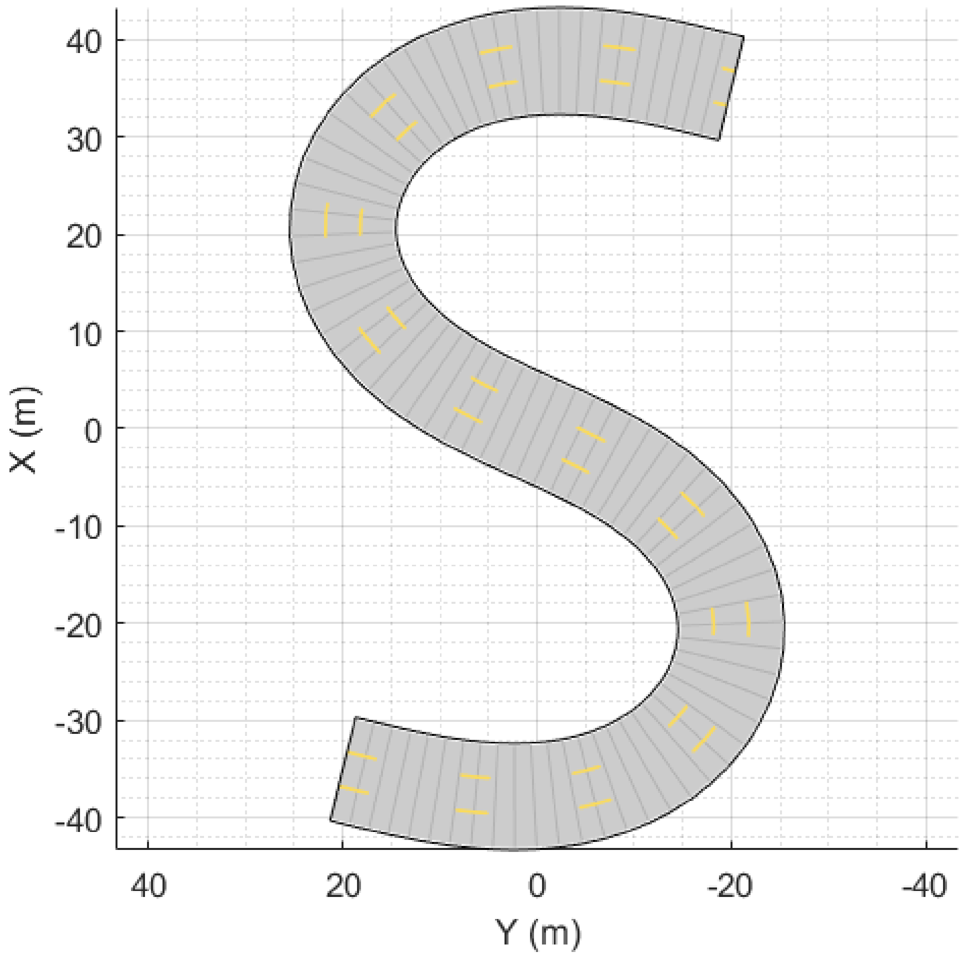
```
scenario = drivingScenario;
roadcenters = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
```

Create the lanes and add them to the road.

```
lm = [laneMarking('Solid','Color','w') ...
     laneMarking('Dashed','Color','y') ...
     laneMarking('Dashed','Color','y') ...
     laneMarking('Solid','Color','w')];
ls = lanespec(3,'Marking',lm);
road(scenario,roadcenters,'Lanes',ls);
```

Plot the scenario.

```
plot(scenario)
```



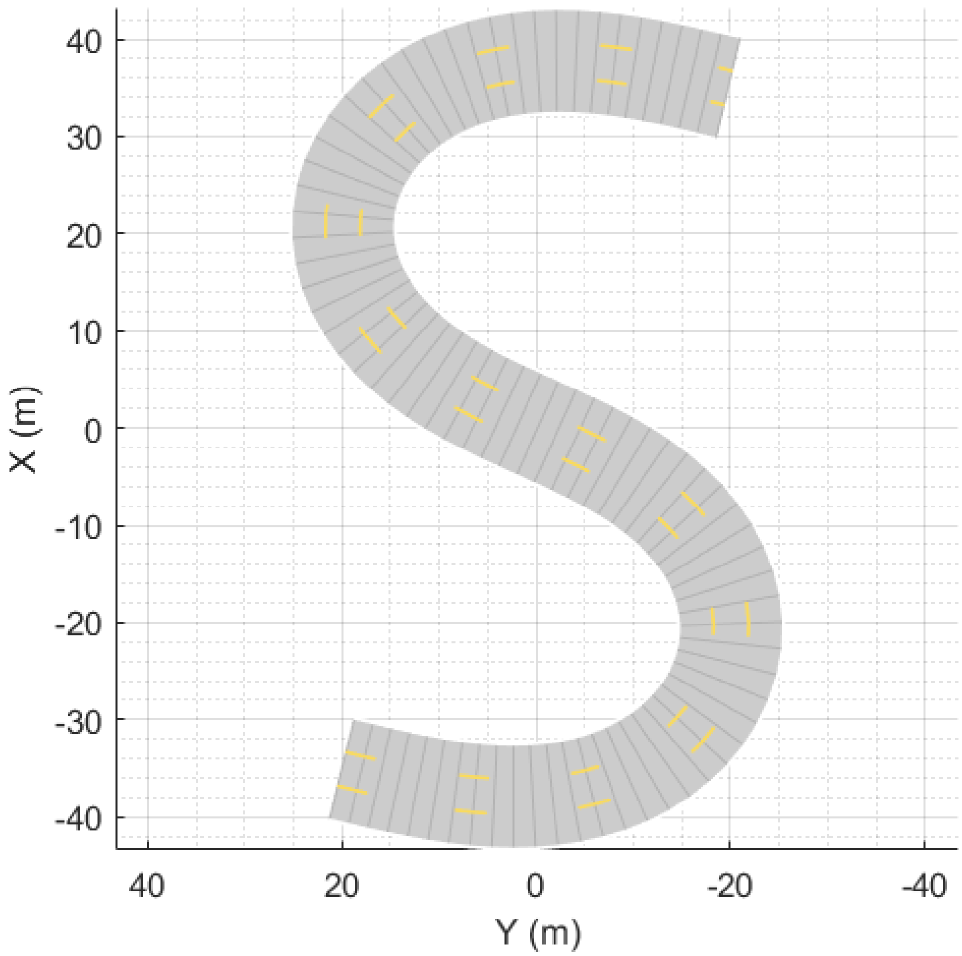
Export the road network in the scenario to OpenDRIVE file.

```
fileName = 'scurveroad.xodr';
export(scenario,'OpenDRIVE',fileName)
```

```
Warning: There may be minor variation between the actual driving scenario and the exported OpenDR
```

You can import the OpenDRIVE file to MATLAB workspace by using the `roadNetwork` function.

```
scenario = drivingScenario;
roadNetwork(scenario,'OpenDRIVE',fileName)
plot(scenario)
```



Input Arguments

scenario — Driving scenario
drivingScenario object

Driving scenario, specified as a drivingScenario object. The driving scenario must contain one or more road networks in order to export it to the OpenDRIVE file format.

If the driving scenario does not have lane specifications, then the export function assigns a default lane specification while exporting to the OpenDRIVE file format.

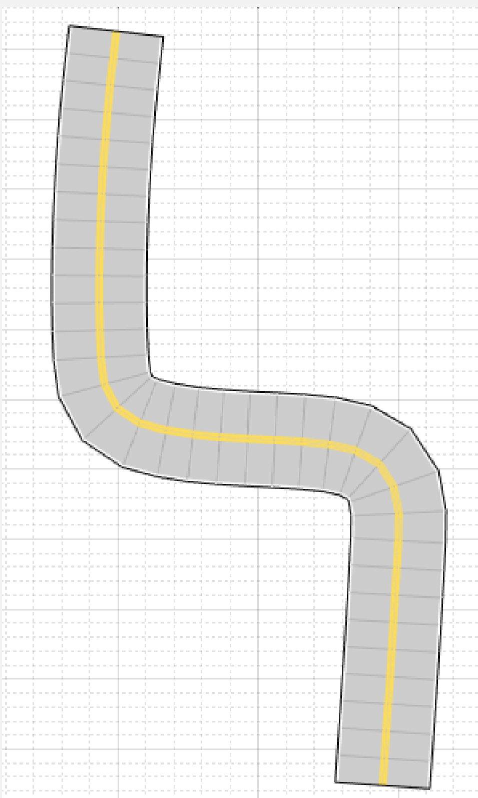
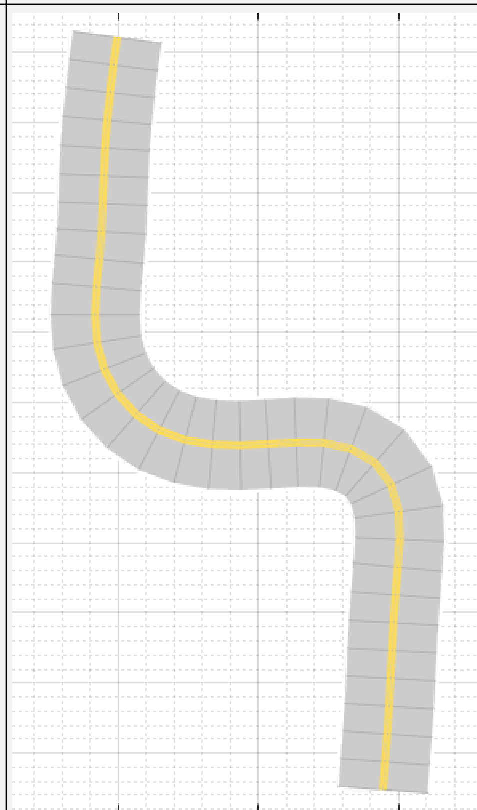
filename — Name of destination OpenDRIVE file
character vector | string scalar

Name of the destination OpenDRIVE file, specified as a character vector or string scalar. You can specify the file name with or without the file extension. If you choose to specify a file extension, the file extension must be either .xodr or .xml. The function uses .xodr as the default. If the specified file name, including the file extension already exists, then the function overwrites the data in the existing file with the road network specified in the scenario argument.

Data Types: `char` | `string`

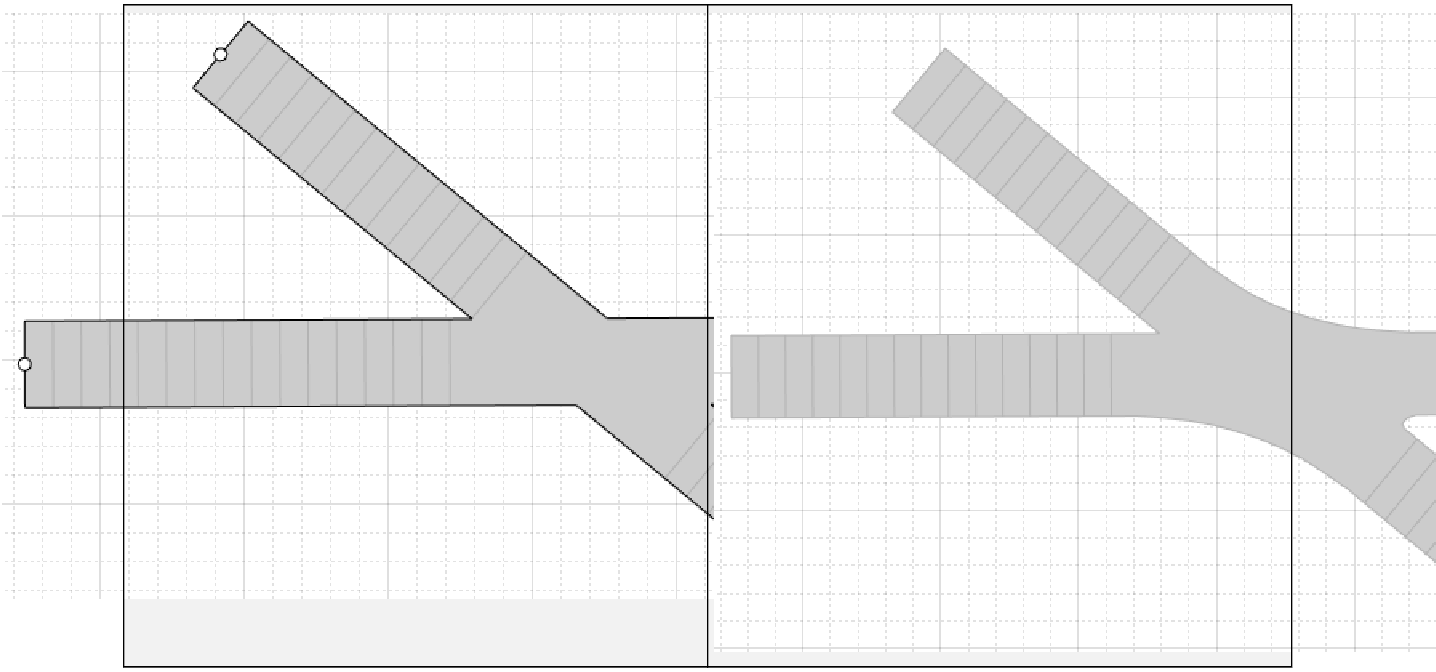
Limitations

- The `export` function does not export the actors and their properties from the original scenario to the OpenDRIVE format.
- The cubic polynomial and the parametric cubic polynomial geometry types in the scenario are exported as spiral geometry types. This causes some variations in the exported road geometry if the road is a curved road. For example, in the figure below, notice that the sharp corners in the input road became relatively smooth when exported to the OpenDRIVE format.

Input Road	Exported OpenDRIVE Road
	

- The junctions of the road network are processed without lane connection information and so, the junction shapes may not be accurate in the exported scenario.

Input Road	Exported OpenDRIVE Road



- The limitations in OpenDRIVE import applies to OpenDRIVE export, if you export a driving scenario object that contains an imported OpenDRIVE scenario. You can import an OpenDRIVE scenario to a `drivingScenario` object by using the `roadNetwork` function. For information on limitations in OpenDRIVE import, see `roadNetwork`.

See Also

Objects
`drivingScenario`

Topics
"Export Driving Scenario to OpenDRIVE File"

Introduced in R2020b

plot

Create driving scenario plot

Syntax

```
plot(scenario)
plot(scenario,Name,Value)
```

Description

`plot(scenario)` creates a 3-D plot with orthonormal perspective, as seen from immediately above the driving scenario, `scenario`.

`plot(scenario,Name,Value)` specifies options using one or more name-value pairs. For example, you can use these options to display road centers and actor waypoints on the plot.

Examples

Create and Display Road Boundaries

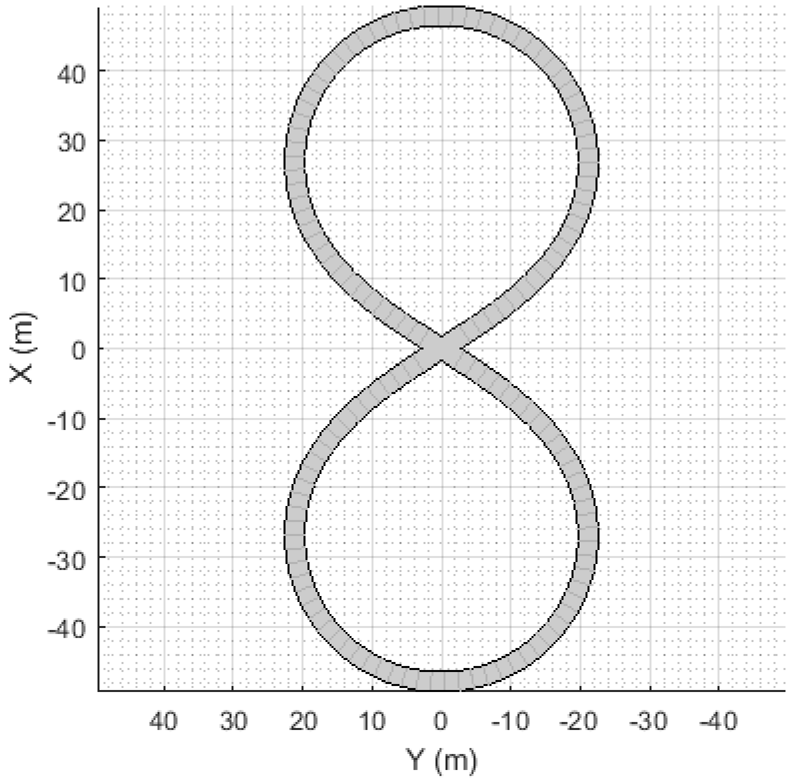
Create a driving scenario containing a figure-8 road specified in the world coordinates of the scenario. Convert the world coordinates of the scenario to the coordinate system of the ego vehicle.

Create an empty driving scenario.

```
scenario = drivingScenario;
```

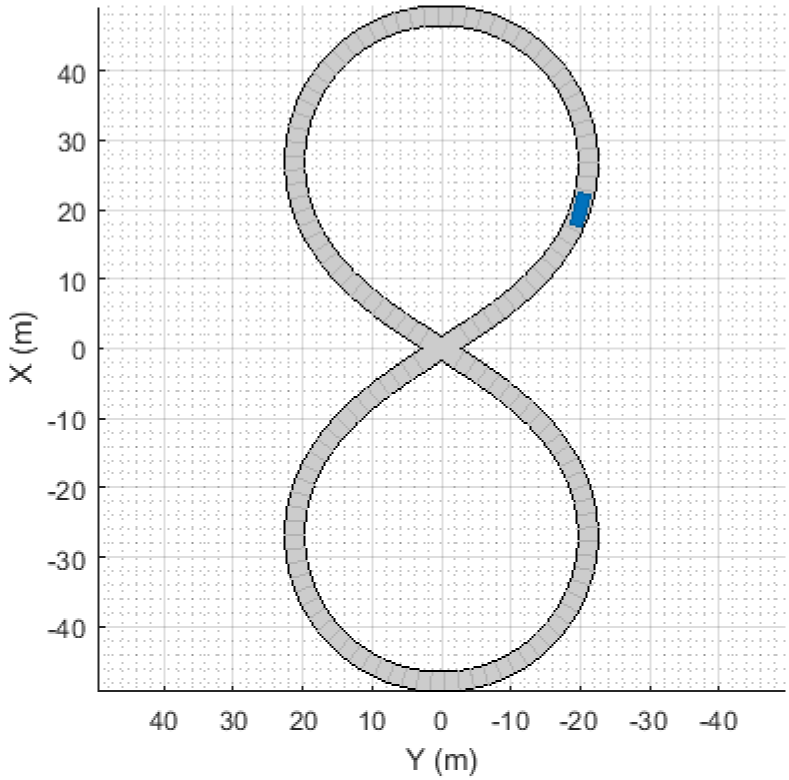
Add a figure-8 road to the scenario. Display the scenario.

```
roadCenters = [0  0  1
             20 -20  1
             20  20  1
            -20 -20  1
            -20  20  1
              0   0  1];

roadWidth = 3;
bankAngle = [0 15 15 -15 -15 0];
road(scenario,roadCenters,roadWidth,bankAngle);
plot(scenario)
```



Add an ego vehicle to the scenario. Position the vehicle at world coordinates (20, –20) and orient it at a –15 degree yaw angle.

```
ego = actor(scenario,'ClassID',1,'Position',[20 -20 0],'Yaw',-15);
```

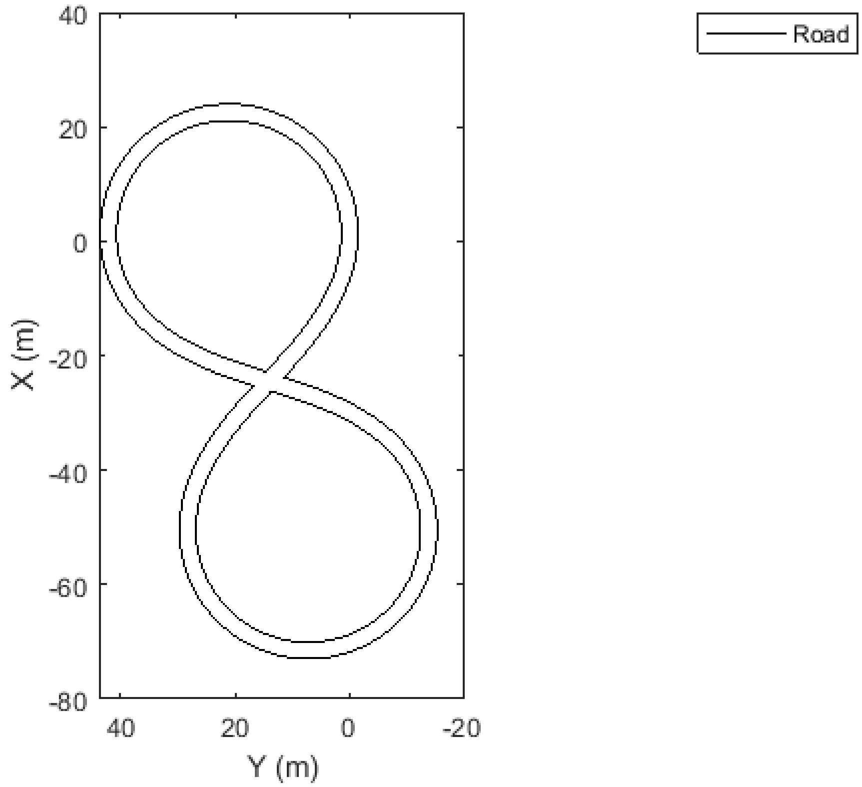


Obtain the road boundaries in ego vehicle coordinates by using the `roadBoundaries` function. Specify the ego vehicle as the input argument.

```
rbEgo1 = roadBoundaries(ego);
```

Display the result on a bird's-eye plot.

```
bep = birdsEyePlot;
lbp = laneBoundaryPlotter(bep,'DisplayName','Road');
plotLaneBoundary(lbp,rbEgo1)
```



Obtain the road boundaries in world coordinates by using the `roadBoundaries` function. Specify the scenario as the input argument.

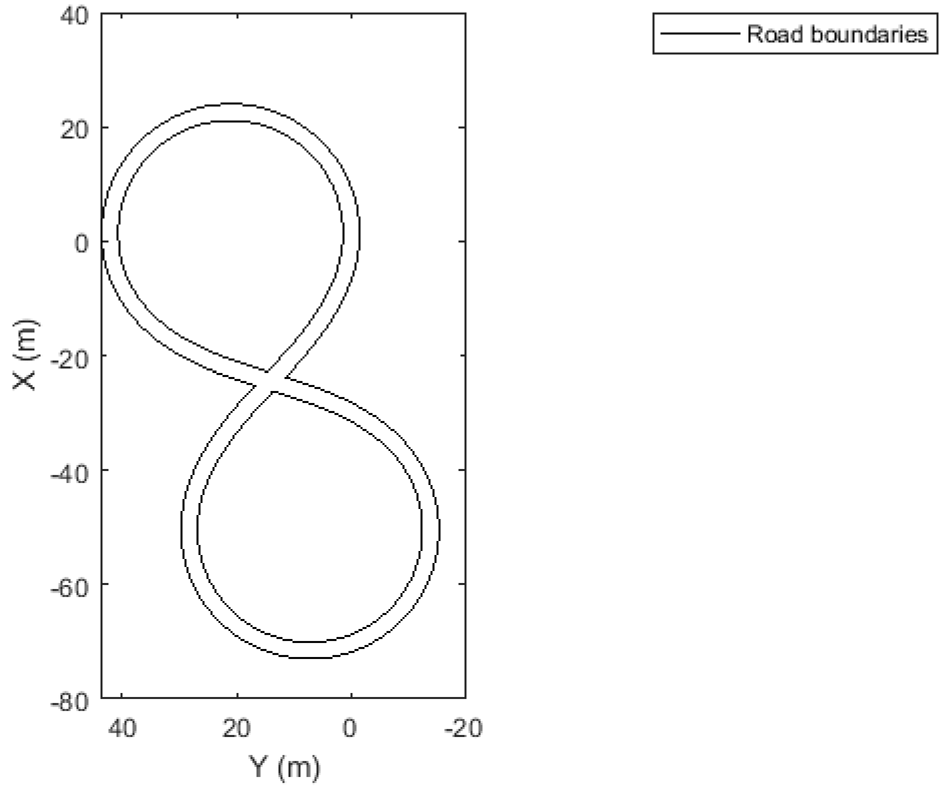
```
rbScenario = roadBoundaries(scenario);
```

Obtain the road boundaries in ego vehicle coordinates by using the `driving.scenario.roadBoundariesToEgo` function.

```
rbEgo2 = driving.scenario.roadBoundariesToEgo(rbScenario,ego);
```

Display the road boundaries on a bird's-eye plot.

```
bep = birdsEyePlot;
lbp = laneBoundaryPlotter(bep,'DisplayName','Road boundaries');
plotLaneBoundary(lbp,{rbEgo2})
```



Create Driving Scenario with Multiple Actors and Roads

Create a driving scenario containing a curved road, two straight roads, and two actors: a car and a bicycle. Both actors move along the road for 60 seconds.

Create the driving scenario object.

```
scenario = drivingScenario('SampleTime',0.1','StopTime',60);
```

Create the curved road using road center points following the arc of a circle with an 800-meter radius. The arc starts at 0°, ends at 90°, and is sampled at 5° increments.

```
angs = [0:5:90]';
R = 800;
roadcenters = R*[cosd(angs) sind(angs) zeros(size(angs))];
roadwidth = 10;
road(scenario,roadcenters,roadwidth);
```

Add two straight roads with the default width, using road center points at each end.

```
roadcenters = [700 0 0; 100 0 0];
road(scenario,roadcenters)
```

```
ans = 
  Road with properties:
```

```
          Name: ""
        RoadID: 2
   RoadCenters: [2x3 double]
     RoadWidth: 6
     BankAngle: [2x1 double]
```

```
roadcenters = [400 400 0; 0 0 0];
road(scenario,roadcenters)
```

```
ans =
  Road with properties:

          Name: ""
        RoadID: 3
   RoadCenters: [2x3 double]
     RoadWidth: 6
     BankAngle: [2x1 double]
```

Get the road boundaries.

```
rbdry = roadBoundaries(scenario);
```

Add a car and a bicycle to the scenario. Position the car at the beginning of the first straight road.

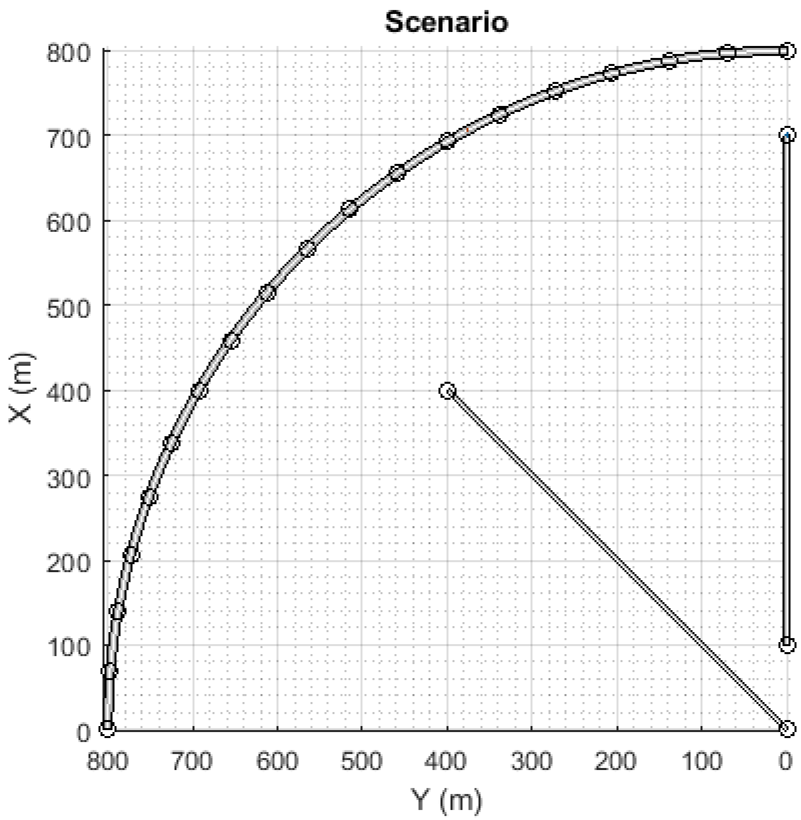
```
car = vehicle(scenario,'ClassID',1,'Position',[700 0 0], ...
    'Length',3,'Width',2,'Height',1.6);
```

Position the bicycle farther down the road.

```
bicycle = actor(scenario,'ClassID',3,'Position',[706 376 0]', ...
    'Length',2,'Width',0.45,'Height',1.5);
```

Plot the scenario.

```
plot(scenario,'Centerline','on','RoadCenters','on');
title('Scenario');
```



Display the actor poses and profiles.

```
poses = actorPoses(scenario)
```

```
poses=2×1 struct array with fields:
    ActorID
    Position
    Velocity
    Roll
    Pitch
    Yaw
    AngularVelocity
```

```
profiles = actorProfiles(scenario)
```

```
profiles=2×1 struct array with fields:
    ActorID
    ClassID
    Length
    Width
    Height
    OriginOffset
    MeshVertices
    MeshFaces
    RCSPattern
    RCSAzimuthAngles
```

RCSElevationAngles

Input Arguments

scenario — Driving scenario
drivingScenario object

Driving scenario, specified as a drivingScenario object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: plot(sc,'Centerline','on','RoadCenters','on') displays the center line and road centers of each road segment.

Parent — Axes in which to draw plot
Axes object

Axes in which to draw the plot, specified as the comma-separated pair consisting of 'Parent' and an Axes object. If you do not specify Parent, a new figure is created.

Centerline — Display center line of roads
'off' (default) | 'on'

Display the center line of roads, specified as the comma-separated pair consisting of 'Centerline' and 'off' or 'on'. The center line follows the middle of each road segment. Center lines are discontinuous through areas such as intersections or road splits.

RoadCenters — Display road centers
'off' (default) | 'on'

Display road centers, specified as the comma-separated pair consisting of 'RoadCenters' and 'off' or 'on'. The road centers define the roads shown in the plot.

Waypoints — Display actor waypoints
'off' (default) | 'on'

Display actor waypoints, specified as the comma-separated pair consisting of 'Waypoints' and 'off' or 'on'. Waypoints define the trajectory of the actor.

Meshes — Display actor meshes
'off' (default) | 'on'

Display actor meshes instead of cuboids, specified as the comma-separated pair consisting of 'Meshes' and 'off' or 'on'.

ActorIndicators — Actors around which to draw indicator
[] (default) | vector of ActorID integers

Actors around which to draw indicators, specified as the comma-separated pair consisting of 'ActorIndicators' and a vector of ActorID integers. The driving scenario plot draws circles around the actors that have the specified `ActorID` values. Each circle is the same color as the actor that it surrounds. The circles are not sensor coverage areas.

Use this name-value pair to highlight the ego vehicle in driving scenarios that contain several vehicles.

Tips

- To rotate any plot, in the figure window, select **View** > **Camera Toolbar**.

See Also

Objects
`drivingScenario`

Functions
`actor` | `chasePlot` | `road` | `trajectory` | `vehicle`

Topics
"Create Driving Scenario Programmatically"
"Create Actor and Vehicle Trajectories Programmatically"
"Define Road Layouts Programmatically"

Introduced in R2017a

record

Run driving scenario and record actor states

Syntax

```
rec = record(scenario)
```

Description

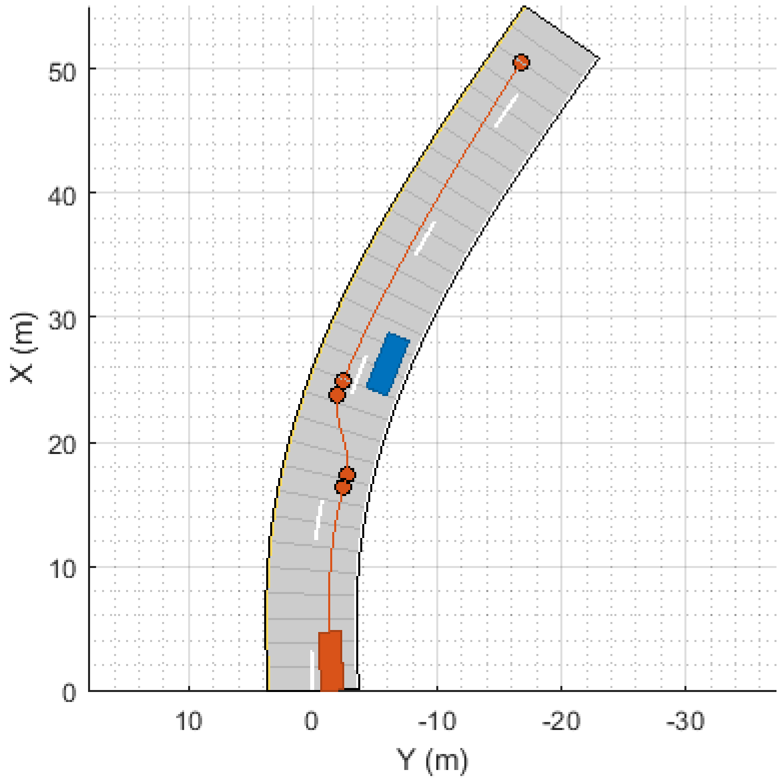
`rec = record(scenario)` returns a recording, `rec`, of the states of actors in a driving scenario simulation, `scenario`. To record a scenario, you must define the trajectory of at least one actor.

Examples

Record Actor Poses from Driving Scenario

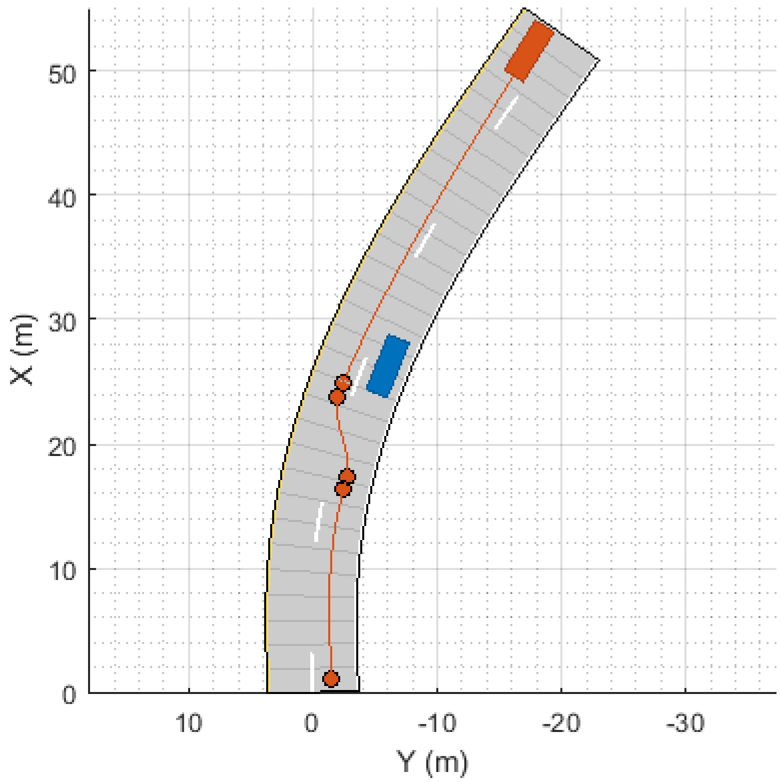
Create a driving scenario in which one car passes a stationary car on a two-lane road.

```
scenario = drivingScenario;
road(scenario,[0 0; 10 0; 53 -20],'lanes',lanespec(2));
plot(scenario,'Waypoints','on');
stationaryCar = 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; % m/s
trajectory(passingCar,waypoints,speed);
```



Record the driving scenario simulation.

```
rec = record(scenario);
```



Compare the recorded poses of the passing car at the start and end of the simulation.

```
rec(1).ActorPoses(2)
```

```
ans = struct with fields:
            ActorID: 2
           Position: [1 -1.5000 0]
           Velocity: [14.9816 0.7423 0]
               Roll: 0
              Pitch: 0
                Yaw: 2.8367
    AngularVelocity: [0 0 1.2537e-05]
```

```
rec(end).ActorPoses(2)
```

```
ans = struct with fields:
            ActorID: 2
           Position: [50.4717 -16.6823 0]
           Velocity: [12.7171 -7.9546 0]
               Roll: 0
              Pitch: 0
                Yaw: -32.0261
    AngularVelocity: [0 0 -0.0099]
```

Input Arguments

scenario — Driving scenario
drivingScenario object

Driving scenario, specified as a drivingScenario object.

Output Arguments

rec — Recording of actor states during simulation
M-by-1 vector of structures

Recording of actor states during simulation, returned as an *M*-by-1 vector of structures. *M* is the number of time steps in the simulation. Each structure corresponds to a simulation time step.

The rec structure has these fields:

Field	Description	Type
SimulationTime	Simulation time at each time step	Real scalar
ActorPoses	Actor poses in scenario coordinates	*N*-by-1 vector of ActorPoses structures, where *N* is the number of actors, including vehicles.

Each ActorPoses structure has these fields.

Field	Description
ActorID	Scenario-defined actor identifier, specified as a positive integer.
Position	Position of actor, specified as a real-valued vector of the form [x, y, z]. Units are in meters.
Velocity	Velocity (v) of actor in the x-, y-, and z-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
Roll	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
Pitch	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
Yaw	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
AngularVelocity	Angular velocity (ω) of actor in the x-, y-, and z-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

For full definitions of these structure fields, see the actor and vehicle functions.

Data Types: struct

See Also

Objects
`drivingScenario`

Functions
`actor` | `actorPoses` | `advance` | `restart` | `vehicle`

Topics
"Create Driving Scenario Programmatically"

Introduced in R2017a

restart

Restart driving scenario simulation from beginning

Syntax

`restart(scenario)`

Description

`restart(scenario)` restarts the simulation of the driving scenario, `scenario`, from the beginning. The function sets the `SimulationTime` property of the driving scenario to `0`.

Examples

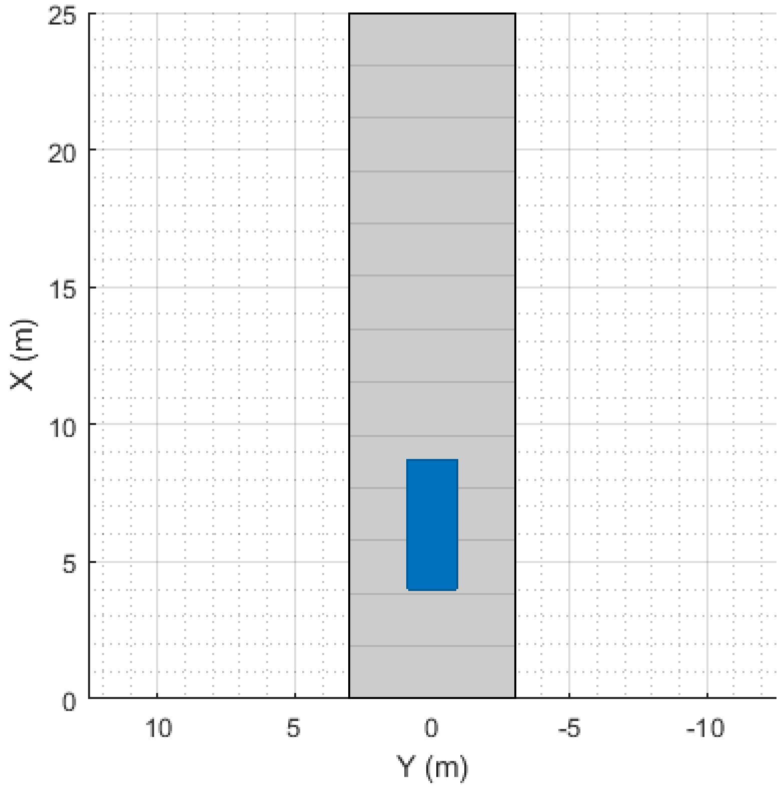
Restart Driving Scenario Simulation

Create a driving scenario in which a vehicle travels down a straight, 25-meter road at 20 meters per second. Plot the scenario.

```
scenario = drivingScenario('SampleTime',0.1);

roadcenters= [0 0 0; 25 0 0];
road(scenario,roadcenters)
```

```
ans = 
  Road with properties:

           Name: ""
         RoadID: 1
    RoadCenters: [2x3 double]
      RoadWidth: 6
      BankAngle: [2x1 double]
```

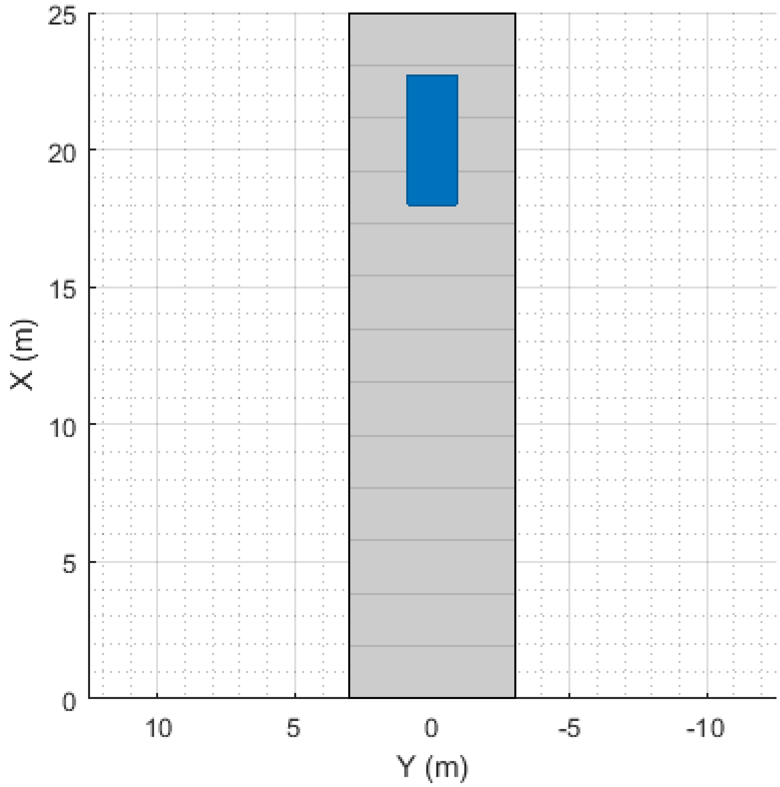
```
v = vehicle(scenario,'ClassID',1);

waypoints = [5 0 0; 20 0 0];
speed = 20; % m/s
trajectory(v,waypoints,speed)

plot(scenario)
```



Run the simulation and display the location of the vehicle at each time step.

```
while advance(scenario)
    fprintf('Vehicle location: %0.2f meters at t = %0.0f ms\n', ...
            v.Position(1), ...
            scenario.SimulationTime * 1000)
end
```

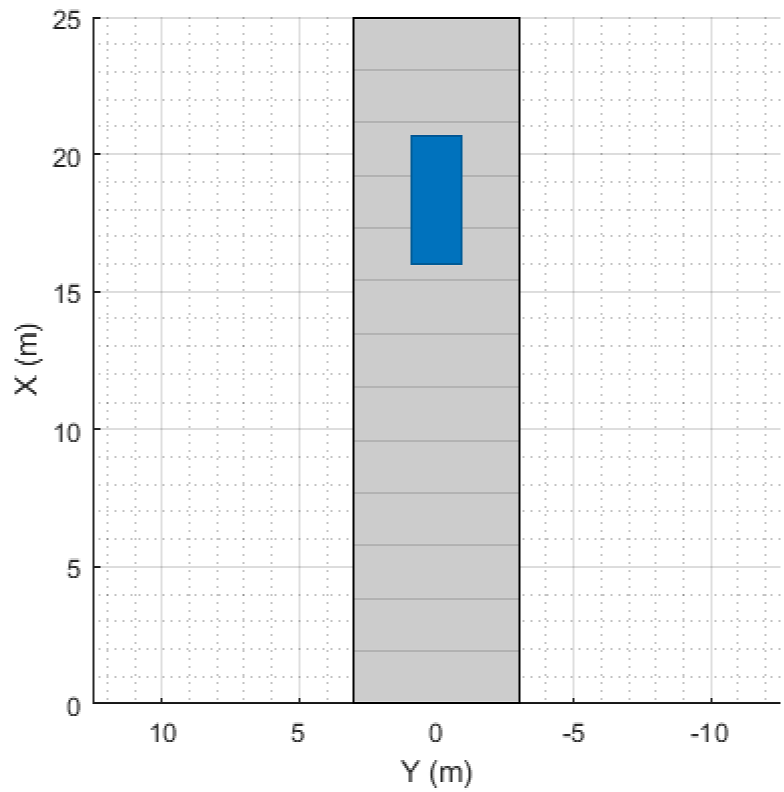
```
Vehicle location: 7.00 meters at t = 100 ms

Vehicle location: 9.00 meters at t = 200 ms

Vehicle location: 11.00 meters at t = 300 ms

Vehicle location: 13.00 meters at t = 400 ms

Vehicle location: 15.00 meters at t = 500 ms

Vehicle location: 17.00 meters at t = 600 ms

Vehicle location: 19.00 meters at t = 700 ms
```



Restart the simulation. Increase the sample time and rerun the simulation.

```
restart(scenario);
scenario.SampleTime = 0.2;

while advance(scenario)
    fprintf('Vehicle location: %0.2f meters at t = %0.0f ms\n', ...
            v.Position(1), ...
            scenario.SimulationTime * 1000)
end
```

```
Vehicle location: 9.00 meters at t = 200 ms

Vehicle location: 13.00 meters at t = 400 ms

Vehicle location: 17.00 meters at t = 600 ms
```



Input Arguments

`scenario` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

See Also

Objects
`drivingScenario`

Functions
`advance` | `record`

Topics
"Create Driving Scenario Programmatically"

Introduced in R2017a

updatePlots

Update driving scenario plots

Syntax

```
updatePlots(scenario)
```

Description

`updatePlots(scenario)` updates the display of all existing plots for the driving scenario, `scenario`. Driving scenario plots are automatically updated every time you call the `advance` function to advance the simulation. Use `updatePlots` after you update any actor properties and want to refresh the plot without having to call `advance`.

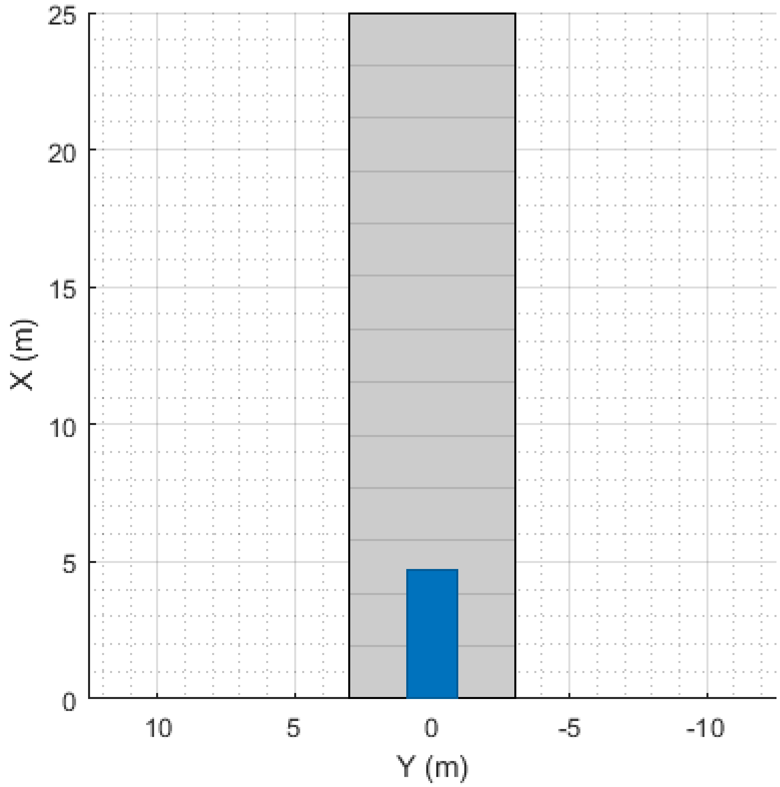
Examples

Update Driving Scenario Plots

Update driving scenario plots after changing aspects of the scenario.

Create a driving scenario containing a vehicle on a straight, 25-meter road segment. Plot the scenario.

```
scenario = drivingScenario;
roadcenters = [0 0 0; 25 0 0];
road(scenario,roadcenters);

v = vehicle(scenario,'ClassID',1);
v.Position = [1 0 0];

plot(scenario)
```



Use a chase plot to plot the scenario from the perspective of the vehicle.

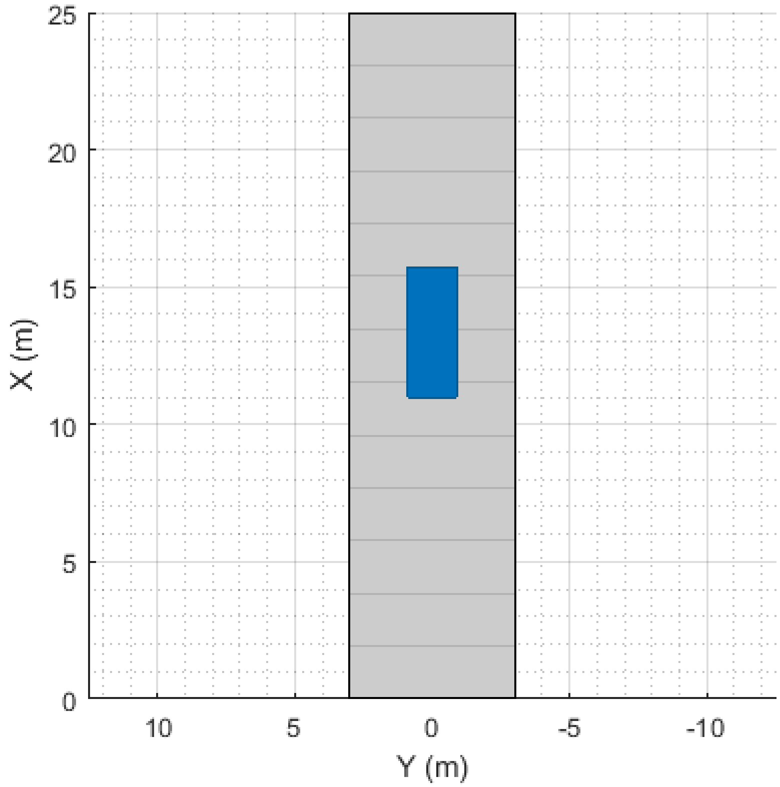
```
chasePlot(v)
```

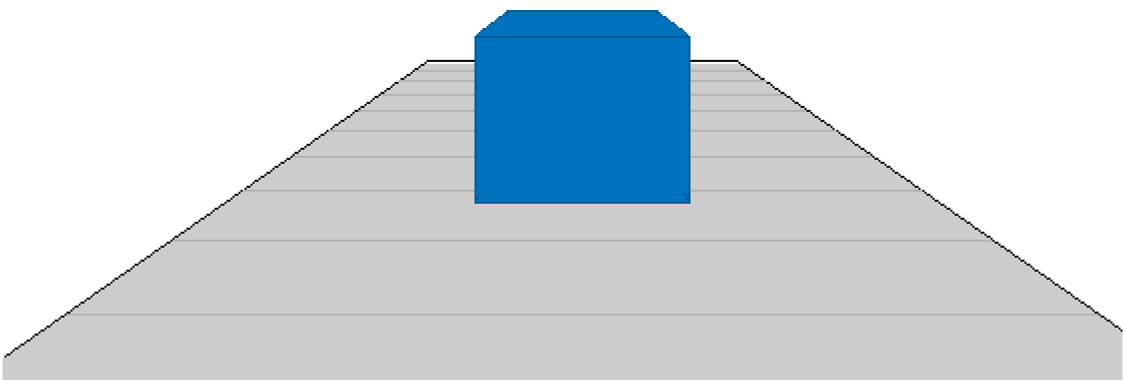
Set a new position for the vehicle.

```
v.Position = [12 0 0];
```

Update both plots to show the new position of the vehicle.

```
updatePlots(scenario)
```


25
20
15
10
5
0
X (m)
10
5
0
-5
-10
Y (m)



Input Arguments

scenario — Driving scenario
drivingScenario object

Driving scenario, specified as a drivingScenario object.

See Also

Objects
drivingScenario

Functions
advance | chasePlot | plot

Topics
"Create Driving Scenario Programmatically"

Introduced in R2017a

actor

Package:

Add actor to driving scenario

Syntax

```
ac = actor(scenario)
ac = actor(scenario,Name,Value)
```

Description

`ac = actor(scenario)` adds an `Actor` object, `ac`, to the driving scenario, `scenario`. The actor has default property values.

Actors are cuboids (box shapes) that represent objects in motion, such as cars, pedestrians, and bicycles. Actors can also represent stationary obstacles that can influence the motion of other actors, such as barriers. For more details about how actors are defined, see “Actor and Vehicle Positions and Dimensions” on page 4-331.

`ac = actor(scenario,Name,Value)` sets actor properties using one or more name-value pair arguments. For example, you can set the position, velocity, dimensions, and orientation of the actor. You can also set a time for the actor to spawn or despawn in the scenario.

Note You can configure the actors in a driving scenario to spawn and despawn, and then import the associated `drivingScenario` object into the **Driving Scenario Designer** app. The app considers the first actor created in the driving scenario to be the ego actor and does not allow the ego actor to either spawn or despawn in the scenario.

Examples

Create Driving Scenario with Multiple Actors and Roads

Create a driving scenario containing a curved road, two straight roads, and two actors: a car and a bicycle. Both actors move along the road for 60 seconds.

Create the driving scenario object.

```
scenario = drivingScenario('SampleTime',0.1','StopTime',60);
```

Create the curved road using road center points following the arc of a circle with an 800-meter radius. The arc starts at 0°, ends at 90°, and is sampled at 5° increments.

```
angs = [0:5:90]';
R = 800;
roadcenters = R*[cosd(angs) sind(angs) zeros(size(angs))];
roadwidth = 10;
road(scenario,roadcenters,roadwidth);
```

Add two straight roads with the default width, using road center points at each end.

```
roadcenters = [700 0 0; 100 0 0];
road(scenario,roadcenters)
```

```
ans = 
  Road with properties:

           Name: ""
         RoadID: 2
    RoadCenters: [2x3 double]
      RoadWidth: 6
      BankAngle: [2x1 double]
```

```
roadcenters = [400 400 0; 0 0 0];
road(scenario,roadcenters)
```

```
ans = 
  Road with properties:

           Name: ""
         RoadID: 3
    RoadCenters: [2x3 double]
      RoadWidth: 6
      BankAngle: [2x1 double]
```

Get the road boundaries.

```
rbdry = roadBoundaries(scenario);
```

Add a car and a bicycle to the scenario. Position the car at the beginning of the first straight road.

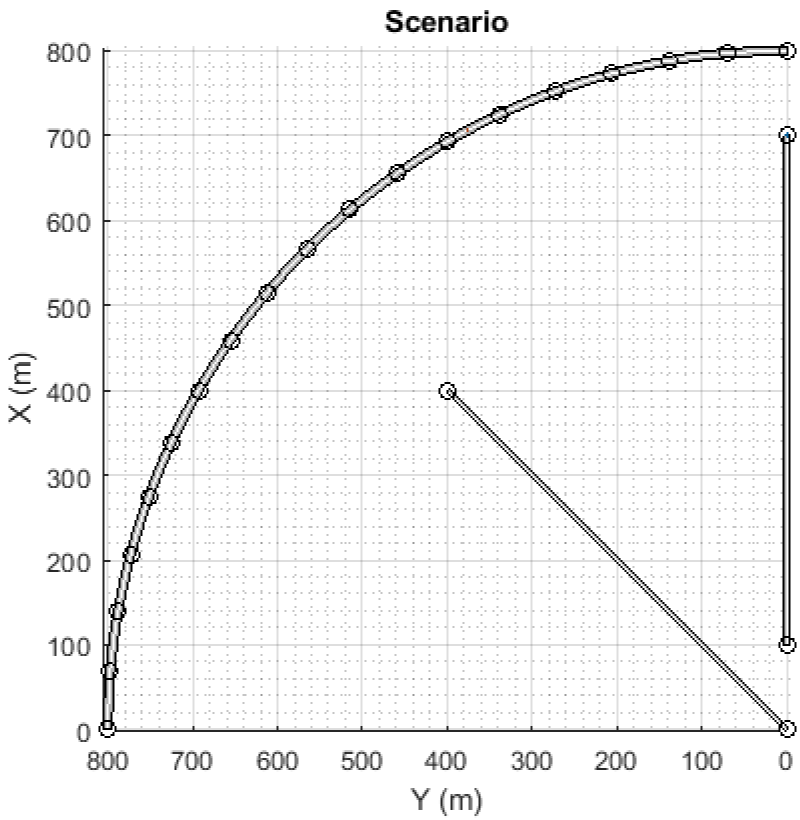
```
car = vehicle(scenario,'ClassID',1,'Position',[700 0 0], ...
    'Length',3,'Width',2,'Height',1.6);
```

Position the bicycle farther down the road.

```
bicycle = actor(scenario,'ClassID',3,'Position',[706 376 0]', ...
    'Length',2,'Width',0.45,'Height',1.5);
```

Plot the scenario.

```
plot(scenario,'Centerline','on','RoadCenters','on');
title('Scenario');
```



Display the actor poses and profiles.

```
poses = actorPoses(scenario)
```

```
poses=2×1 struct array with fields:
    ActorID
    Position
    Velocity
    Roll
    Pitch
    Yaw
    AngularVelocity
```

```
profiles = actorProfiles(scenario)
```

```
profiles=2×1 struct array with fields:
    ActorID
    ClassID
    Length
    Width
    Height
    OriginOffset
    MeshVertices
    MeshFaces
    RCSPattern
    RCSAzimuthAngles
```

```
    RCSElevationAngles
```

Spawn and Despawn Actors in Scenario During Simulation

Create a driving scenario. Set the stop time for the scenario to 3 seconds.

```
scenario = drivingScenario('StopTime',3);
```

Add a two-lane road to the scenario.

```
roadCenters = [0 1 0; 53 1 0];
laneSpecification = lanespec([1 1]);
road(scenario,roadCenters,'Lanes',laneSpecification);
```

Add another road that intersects the first road at a right angle to form a T-shape.

```
roadCenters = [20.3 33.4 0; 20 3 0];
laneSpecification = lanespec(2);
road(scenario,roadCenters,'Lanes',laneSpecification)
```

```
ans = 
  Road with properties:

           Name: ""
         RoadID: 2
    RoadCenters: [2x3 double]
      RoadWidth: 7.3500
      BankAngle: [2x1 double]
```

Add the ego vehicle to the scenario and define its waypoints. Set the ego vehicle speed to 20 m/s and generate the trajectories for the ego vehicle.

```
egoVehicle = vehicle(scenario,'ClassID',1, ...
                    'Position',[1.5 2.5 0]);
waypoints = [2 3 0; 13 3 0;
            21 3 0; 31 3 0;
            43 3 0; 47 3 0];
speed = 15;
trajectory(egoVehicle,waypoints,speed)
```

Add a non-ego actor to the scenario. Set the non-ego actor to spawn and despawn during the simulation by specifying an entry time and an exit time.

```
nonEgoactor1 = actor(scenario,'ClassID',1, ...
                'Position',[22 30 0],'EntryTime',0.8,'ExitTime',2);
```

Define the waypoints for the non-ego actor. Set the non-ego actor speed to 35 m/s and generate its trajectories.

```
waypoints = [22 30 0; 22 23 0;
            22 13 0; 22 7 0;
            18 -0.3 0; 12 -0.8 0; 3 -0.8 0];
speed = 35;
trajectory(nonEgoactor1,waypoints,speed)
```

Add another non-ego actor to the scenario. Set the second non-ego actor to spawn during the simulation by specifying an entry time. Since you do not specify an exit time, this actor will remain in the scenario until the scenario ends.

```
nonEgoactor2 = actor(scenario,'ClassID',1, ...
                'Position',[48 -1 0],'EntryTime',2);
```

Define the waypoints for the second non-ego actor. Set the actor speed to 60 m/s and generate its trajectories.

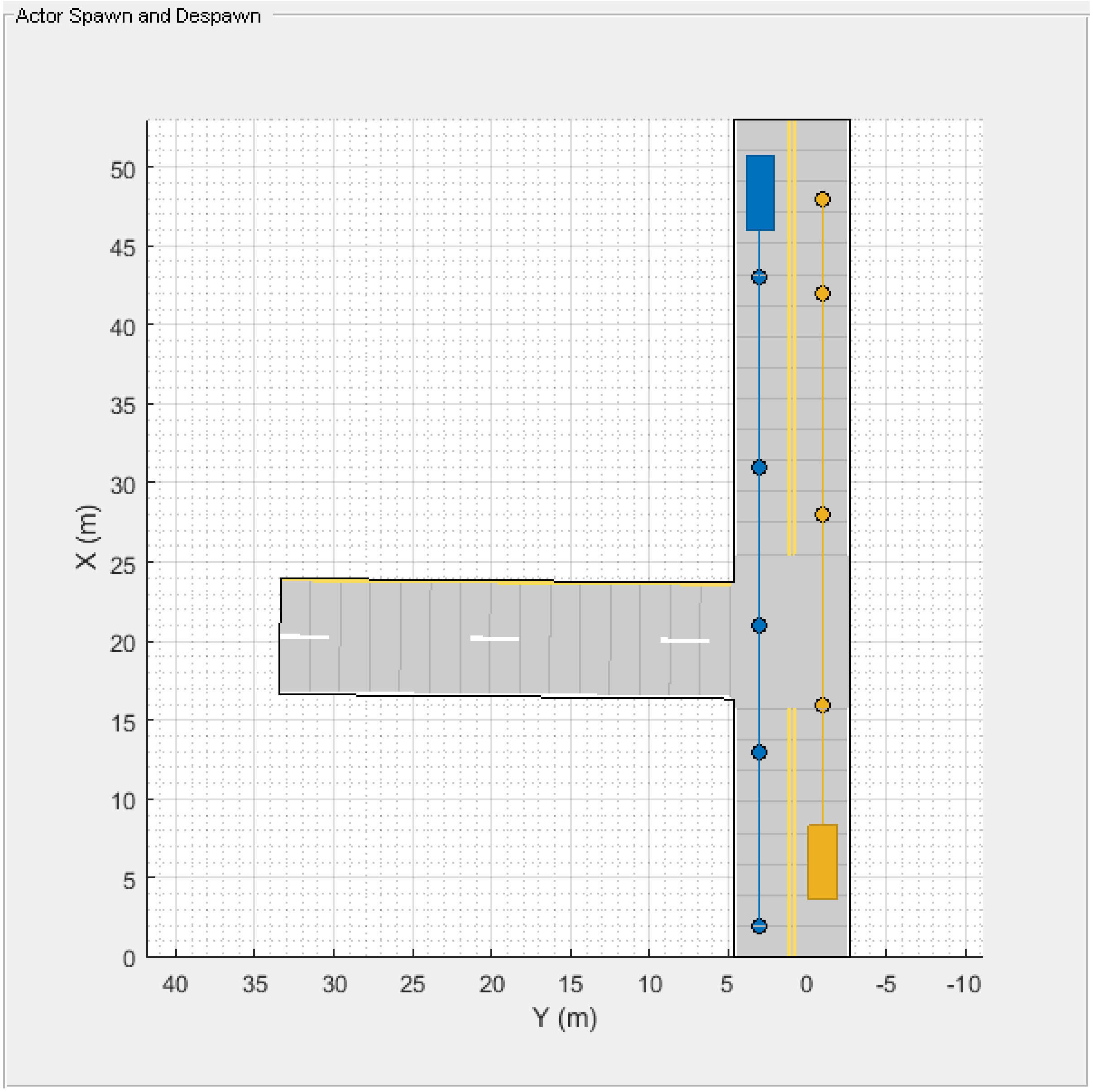
```
waypoints = [48 -1 0; 42 -1 0; 28 -1 0;
             16 -1 0; 6 -1 0];
speed = 60;
trajectory(nonEgoactor2,waypoints,speed)
```

Create a custom figure window to plot the scenario.

```
fig = figure;
set(fig,'Position',[0 0 600 600])
movegui(fig,'center')
hViewPnl = uipanel(fig,'Position',[0 0 1 1],'Title','Actor Spawn and Despawn');
hPlt = axes(hViewPnl);
```

Plot the scenario and run the simulation. Observe how the non-ego actors spawn and despawn in the scenario while simulation is running.

```
plot(scenario,'Waypoints','on','Parent',hPlt)
while advance(scenario)
    pause(0.1)
end
```



Input Arguments

`scenario` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'Height',1.7 sets the height of the actor to 1.7 meters upon creation.

ClassID — Classification identifier
0 (default) | nonnegative integer

Classification identifier of actor, specified as the comma-separated pair consisting of 'ClassID' and a nonnegative integer.

Specify ClassID values to group together actors that have similar dimensions, radar cross-section (RCS) patterns, or other properties. As a best practice, before adding actors to a drivingScenario object, determine the actor classification scheme you want to use. Then, when creating the actors, specify the ClassID name-value pair to set classification identifiers according to the actor classification scheme.

Suppose you want to create a scenario containing these actors:

- Two cars, one of which is the ego vehicle
- A truck
- A bicycle

The code shows a sample classification scheme for this scenario, where 1 refers to cars, 2 refers to trucks, and 3 refers to bicycles. The cars have default vehicle properties. The truck and bicycle have the dimensions of a typical truck and bicycle, respectively.

```
scenario = drivingScenario;
ego = vehicle(scenario,'ClassID',1);
car = vehicle(scenario,'ClassID',1);
truck = vehicle(scenario,'ClassID',2,'Length',8.2,'Width',2.5,'Height',3.5);
bicycle = actor(scenario,'ClassID',3,'Length',1.7,'Width',0.45,'Height',1.7);
```

The default ClassID of 0 is reserved for an object of an unknown or unassigned class. If you plan to import drivingScenario objects into the **Driving Scenario Designer** app, do not leave the ClassID property of actors set to 0. The app does not recognize a ClassID of 0 for actors and returns an error. Instead, set ClassID values of actors according to the actor classification scheme used in the app.

ClassID	Class Name
1	Car
2	Truck
3	Bicycle
4	Pedestrian
5	Barrier

Name — Name of actor
"" (default) | character vector | string scalar

Name of the actor, specified as the comma-separated pair consisting of 'Name' and a character vector or string scalar.

Example: 'Name','Actor1'

Example: "Name","Actor1"

Data Types: char | string

EntryTime — Entry time for actor to spawn
0 (default) | positive scalar

Entry time for an actor to spawn in the driving scenario, specified as the comma-separated pair consisting of 'EntryTime' and a positive scalar. Units are in seconds, measured from the start time of the scenario.

Specify this name-value pair argument to add or make an actor appear in the driving scenario at the specified time, while the simulation is running.

- If the actor has an associated exit time, then the entry time must be less than the specified exit time.
- If the actor does not have an associated exit time, then the entry time must be less than or equal to the stop time of the scenario. You can set the stop time for the scenario by specifying a value for the 'StopTime' property of the drivingScenario object.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

ExitTime — Exit time for actor to despawn
Inf (default) | positive scalar

Exit time for an actor to despawn from the driving scenario, specified as the comma-separated pair consisting of 'ExitTime' and a positive scalar. Units are in seconds, measured from the start time of the scenario.

Specify this name-value pair argument to remove or make an actor disappear from the scenario at a specified time while the simulation is running.

- If the actor has an associated entry time, then the exit time must be greater than the specified entry time.
- If the actor does not have an associated entry time, then the exit time must be less than or equal to the stop time of the scenario. You can set the stop time for the scenario by specifying a value for the 'StopTime' property of the drivingScenario object.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

PlotColor — Display color of actor
RGB triplet | hexadecimal color code | color name | short color name

Display color of actor, specified as the comma-separated pair consisting of 'PlotColor' and an RGB triplet, hexadecimal color code, color name, or short color name.

The actor appears in the specified color in all programmatic scenario visualizations, including the plot function, chasePlot function, and plotting functions of birdsEyePlot objects. If you import the scenario into the **Driving Scenario Designer** app, then the actor appears in this color in all app visualizations. If you import the scenario into Simulink, then the actor appears in this color in the **Bird's-Eye Scope**.

If you do not specify a color for the actor, the function assigns one based on the default color order of Axes objects. For more details, see the ColorOrder property for Axes objects.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Position — Position of actor center
`[0 0 0]` (default) | [*x y z*] real-valued vector

Position of the actor center, specified as the comma-separated pair consisting of `'Position'` and an [*x y z*] real-valued vector.

The center of the actor is [*L*/2 *W*/2 *b*], where:

- *L*/2 is the midpoint of actor length *L*.
- *W*/2 is the midpoint of actor width *W*.
- *b* is the bottom of the cuboid.

Units are in meters.

Example: [10;50;0]

Velocity — Velocity of actor center
[0 0 0] (default) | [v_x v_y v_z] real-valued vector

Velocity (v) of the actor center in the x-, y- and z-directions, specified as the comma-separated pair consisting of 'Velocity' and a [v_x v_y v_z] real-valued vector. The 'Position' name-value pair specifies the actor center. Units are in meters per second.

Example: [-4;7;10]

Yaw — Yaw angle of actor
0 (default) | real scalar

Yaw angle of the actor, specified as the comma-separated pair consisting of 'Yaw' and a real scalar. Yaw is the angle of rotation of the actor around the z-axis. Yaw is clockwise-positive when looking in the forward direction of the axis, which points up from the ground. Therefore, when viewing actors from the top down, such as on a bird's-eye plot, yaw is counterclockwise-positive. Angle values are wrapped to the range [–180, 180]. Units are in degrees.

Example: -0.4

Pitch — Pitch angle of actor
0 (default) | real scalar

Pitch angle of the actor, specified as the comma-separated pair consisting of 'Pitch' and a real scalar. Pitch is the angle of rotation of the actor around the y-axis and is clockwise-positive when looking in the forward direction of the axis. Angle values are wrapped to the range [–180, 180]. Units are in degrees.

Example: 5.8

Roll — Roll angle of actor
0 (default) | real scalar

Roll angle of the actor, specified as the comma-separated pair consisting of 'Roll' and a real scalar. Roll is the angle of rotation of the actor around the x-axis and is clockwise-positive when looking in the forward direction of the axis. Angle values are wrapped to the range [–180, 180]. Units are in degrees.

Example: -10

AngularVelocity — Angular velocity of actor
[0 0 0] (default) | [ω_x ω_y ω_z] real-valued vector

Angular velocity (ω) of the actor, in world coordinates, specified as the comma-separated pair consisting of 'AngularVelocity' and a [ω_x ω_y ω_z] real-valued vector. Units are in degrees per second.

Example: [20 40 20]

Length — Length of actor
4.7 (default) | positive real scalar

Length of the actor, specified as the comma-separated pair consisting of 'Length' and a positive real scalar. Units are in meters.

Example: `5.5`

`Width` — Width of actor

`1.8` (default) | positive real scalar

Width of the actor, specified as the comma-separated pair consisting of `'Width'` and a positive real scalar. Units are in meters.

Example: `3.0`

`Height` — Height of actor

`1.4` (default) | positive real scalar

Height of the actor, specified as the comma-separated pair consisting of `'Height'` and a positive real scalar. Units are meters.

Example: `2.1`

`Mesh` — Extended object mesh

`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`RCSPattern` — Radar cross-section pattern of actor

`[10 10; 10 10]` (default) | *Q*-by-*P* real-valued matrix

Radar cross-section (RCS) pattern of actor, specified as the comma-separated pair consisting of `'RCSPattern'` and a *Q*-by-*P* real-valued matrix. RCS is a function of the azimuth and elevation angles, where:

- *Q* is the number of elevation angles specified by the `'RCSElevationAngles'` name-value pair.
- *P* is the number of azimuth angles specified by the `'RCSAzimuthAngles'` name-value pair.

Units are in decibels per square meter (dBsm).

Example: `5.8`

`RCSAzimuthAngles` — Azimuth angles of actor's RCS pattern

`[-180 180]` (default) | *P*-element real-valued vector

Azimuth angles of the actor's RCS pattern, specified as the comma-separated pair consisting of `'RCSAzimuthAngles'` and a *P*-element real-valued vector. *P* is the number of azimuth angles. Values are in the range [–180°, 180°].

Each element of `RCSAzimuthAngles` defines the azimuth angle of the corresponding column of the `'RCSPattern'` name-value pair. Units are in degrees.

Example: `[-90:90]`

`RCSElevationAngles` — Elevation angles of actor's RCS pattern

`[-90 90]` (default) | *Q*-element real-valued vector

Elevation angles of the actor's RCS pattern, specified as the comma-separated pair consisting of `'RCSElevationAngles'` and a *Q*-element real-valued vector. *Q* is the number of elevation angles. Values are in the range [–90°, 90°].

Each element of RCSElevationAngles defines the elevation angle of the corresponding row of the RCSPattern property. Units are in degrees.

Example: [0:90]

Output Arguments

ac — Driving scenario actor
Actor object

Driving scenario actor, returned as an Actor object belonging to the driving scenario specified by scenario.

You can modify the Actor object by changing its property values. The property names correspond to the name-value pair arguments used to create the object. The only property that you cannot modify is ActorID, which is a positive integer indicating the unique, scenario-defined ID of the actor.

To specify or visualize actor motion, use these functions:

trajectory	Create actor or vehicle trajectory in driving scenario
chasePlot	Ego-centric projective perspective plot

To get information about actor characteristics, use these functions:

actorPoses	Positions, velocities, and orientations of actors in driving scenario
actorProfiles	Physical and radar characteristics of actors in driving scenario
targetOutlines	Outlines of targets viewed by actor
targetPoses	Target positions and orientations relative to ego vehicle
driving.scenario.targetsToEgo	Convert actor poses to ego vehicle coordinates
driving.scenario.targetsToScenario	Convert target actor poses from ego vehicle coordinates to world coordinates of scenario

To get information about the roads and lanes that the actor is on, use these functions:

roadBoundaries	Get road boundaries
driving.scenario.roadBoundariesToEgo	Convert road boundaries to ego vehicle coordinates
currentLane	Get current lane of actor
laneBoundaries	Get lane boundaries of actor lane
laneMarkingVertices	Lane marking vertices and faces in driving scenario
roadMesh	Mesh representation of an actor's nearest roads in driving scenario.

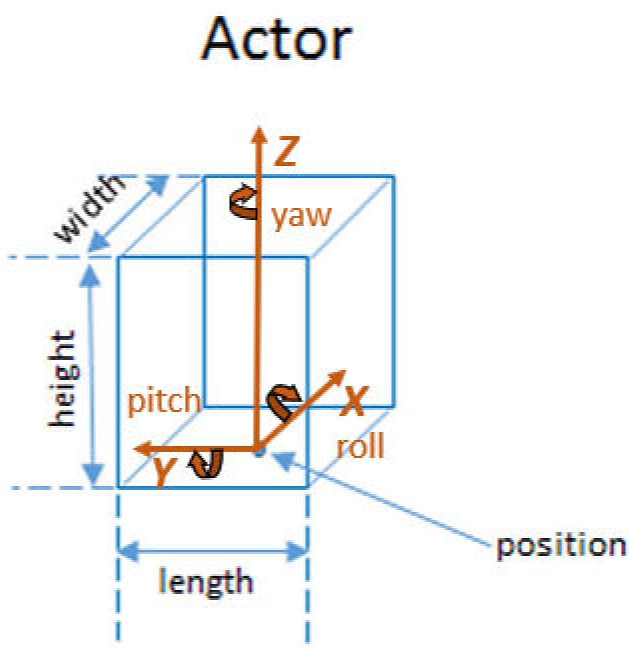
More About

Actor and Vehicle Positions and Dimensions

In driving scenarios, an actor is a cuboid (box-shaped) object with a specific length, width, and height. Actors also have a radar cross-section (RCS) pattern, specified in dBsm, which you can refine by setting angular azimuth and elevation coordinates. The position of an actor is defined as the center of its bottom face. This center point is used as the actor's rotational center, its point of contact with the ground, and its origin in its local coordinate system. In this coordinate system:

- The *X*-axis points forward from the actor.
- The *Y*-axis points left from the actor.
- 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.



A vehicle is an actor that moves on wheels. Vehicles have three extra properties that govern the placement of their front and rear axle.

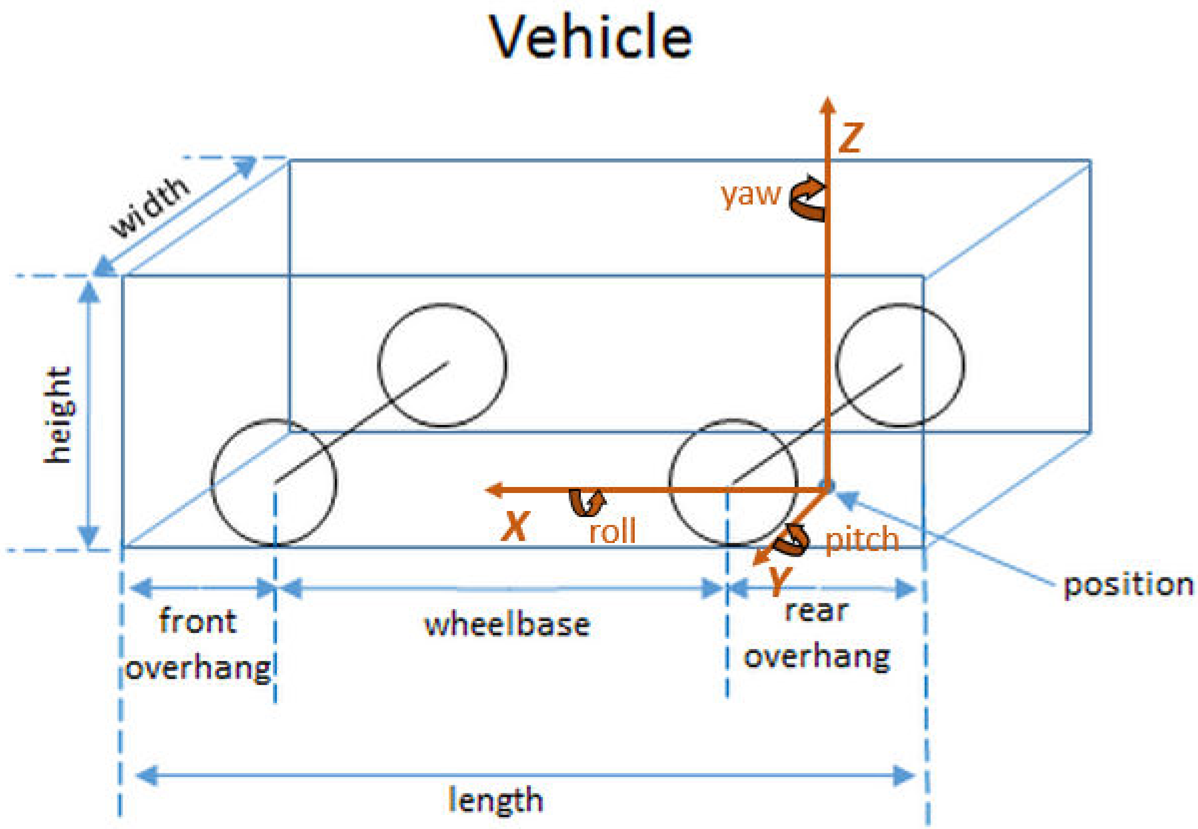
- Wheelbase — Distance between the front and rear axles
- Front overhang — Distance between the front of the vehicle and the front axle
- Rear overhang — Distance between the rear axle and the rear of the vehicle

Unlike other types of actors, the position of a vehicle is defined by the point on the ground that is below the center of its rear axle. This point corresponds to the natural center of rotation of the vehicle. As with nonvehicle actors, this point is the origin in the local coordinate system of the vehicle, where:

- The *X*-axis points forward from the vehicle.

- The *Y*-axis points left from 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.



This table shows a list of common actors and their dimensions. To specify these values in `Actor` and `Vehicle` objects, set the corresponding properties shown.

Actor Classification	Actor Object	Actor Properties						
		`Length`	`Width`	`Height`	`FrontOverhang`	`RearOverhang`	`Wheelbase`	`RCSPattern`
Pedestrian	`Actor`	0.24 m	0.45 m	1.7 m	N/A	N/A	N/A	-8 dBsm
Car	`Vehicle`	4.7 m	1.8 m	1.4 m	0.9 m	1.0 m	2.8 m	10 dBsm
Motorcycle	`Vehicle`	2.2 m	0.6 m	1.5 m	0.37 m	0.32 m	1.51 m	0 dBsm

See Also

`drivingScenario` | `vehicle`

Topics

"Create Driving Scenario Programmatically"

"Create Actor and Vehicle Trajectories Programmatically"

Introduced in R2017a

actorPoses

Positions, velocities, and orientations of actors in driving scenario

Syntax

```
poses = actorPoses(scenario)
```

Description

`poses = actorPoses(scenario)` returns the current poses (positions, velocities, and orientations) for all actors in the driving scenario, `scenario`. Actors include `Actor` and `Vehicle` objects, which you can create using the `actor` and `vehicle` functions, respectively. Actor poses are in scenario coordinates.

Examples

Create Driving Scenario with Multiple Actors and Roads

Create a driving scenario containing a curved road, two straight roads, and two actors: a car and a bicycle. Both actors move along the road for 60 seconds.

Create the driving scenario object.

```
scenario = drivingScenario('SampleTime',0.1','StopTime',60);
```

Create the curved road using road center points following the arc of a circle with an 800-meter radius. The arc starts at 0°, ends at 90°, and is sampled at 5° increments.

```
angs = [0:5:90]';
R = 800;
roadcenters = R*[cosd(angs) sind(angs) zeros(size(angs))];
roadwidth = 10;
road(scenario,roadcenters,roadwidth);
```

Add two straight roads with the default width, using road center points at each end.

```
roadcenters = [700 0 0; 100 0 0];
road(scenario,roadcenters)
```

```
ans = 
  Road with properties:

           Name: ""
         RoadID: 2
    RoadCenters: [2x3 double]
      RoadWidth: 6
      BankAngle: [2x1 double]
```

```
roadcenters = [400 400 0; 0 0 0];
road(scenario,roadcenters)
```

```
ans =
  Road with properties:

           Name: ""
         RoadID: 3
    RoadCenters: [2x3 double]
      RoadWidth: 6
      BankAngle: [2x1 double]
```

Get the road boundaries.

```
rbdry = roadBoundaries(scenario);
```

Add a car and a bicycle to the scenario. Position the car at the beginning of the first straight road.

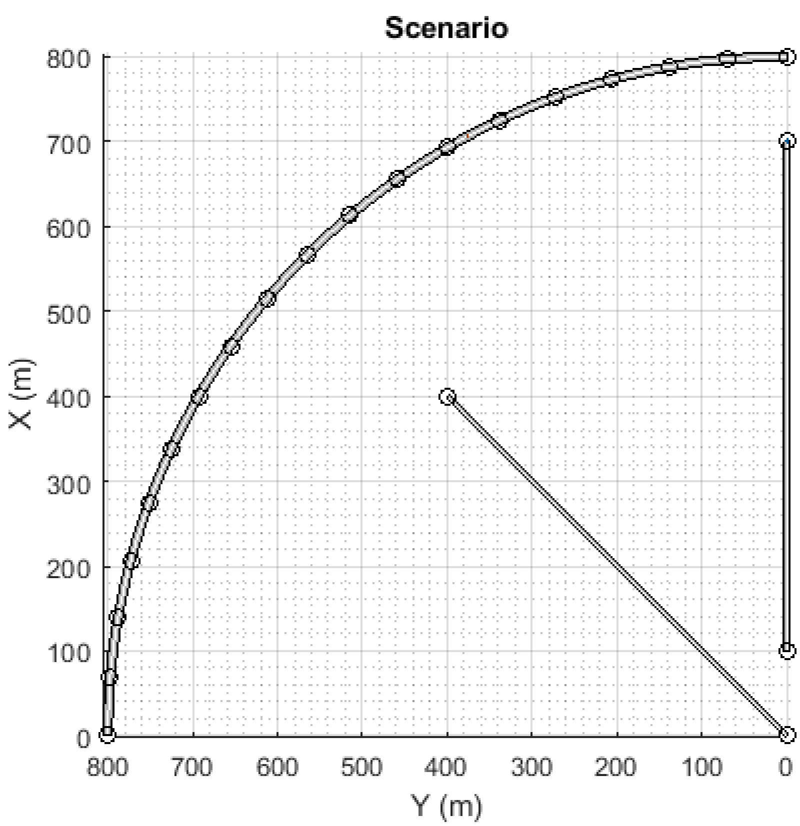
```
car = vehicle(scenario,'ClassID',1,'Position',[700 0 0], ...
    'Length',3,'Width',2,'Height',1.6);
```

Position the bicycle farther down the road.

```
bicycle = actor(scenario,'ClassID',3,'Position',[706 376 0]', ...
    'Length',2,'Width',0.45,'Height',1.5);
```

Plot the scenario.

```
plot(scenario,'Centerline','on','RoadCenters','on');
title('Scenario');
```



Display the actor poses and profiles.

```
poses = actorPoses(scenario)
```

```
poses=2×1 struct array with fields:
    ActorID
    Position
    Velocity
    Roll
    Pitch
    Yaw
    AngularVelocity
```

```
profiles = actorProfiles(scenario)
```

```
profiles=2×1 struct array with fields:
    ActorID
    ClassID
    Length
    Width
    Height
    OriginOffset
    MeshVertices
    MeshFaces
    RCSPattern
    RCSAzimuthAngles
    RCSElevationAngles
```

Input Arguments

`scenario` — Driving scenario

`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Output Arguments

`poses` — Actor poses

structures | array of structures

Actor poses, in scenario coordinates, returned as a structure or an array of structures. Poses are the positions, velocities, and orientations of actors.

Each structure in `poses` has these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.

Field	Description
Velocity	Velocity (v) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
Roll	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
Pitch	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
Yaw	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
AngularVelocity	Angular velocity (ω) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

For full definitions of these structure fields, see the `actor` and `vehicle` functions.

See Also

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

Functions
`actor` | `actorProfiles` | `targetOutlines` | `targetPoses` | `vehicle`

Topics
"Create Driving Scenario Programmatically"

Introduced in R2017a

actorProfiles

Physical and radar characteristics of actors in driving scenario

Syntax

```
profiles = actorProfiles(scenario)
```

Description

`profiles = actorProfiles(scenario)` returns the physical and radar characteristics, `profiles`, for all actors in a driving scenario, `scenario`. Actors include `Actor` and `Vehicle` objects, which you can create using the `actor` and `vehicle` functions, respectively.

You can use actor profiles as inputs to radar, vision, and lidar sensors, such as `radarDetectionGenerator`, `visionDetectionGenerator` and `lidarPointCloudGenerator` objects.

Examples

Create Driving Scenario with Multiple Actors and Roads

Create a driving scenario containing a curved road, two straight roads, and two actors: a car and a bicycle. Both actors move along the road for 60 seconds.

Create the driving scenario object.

```
scenario = drivingScenario('SampleTime',0.1','StopTime',60);
```

Create the curved road using road center points following the arc of a circle with an 800-meter radius. The arc starts at 0°, ends at 90°, and is sampled at 5° increments.

```
angs = [0:5:90]';
R = 800;
roadcenters = R*[cosd(angs) sind(angs) zeros(size(angs))];
roadwidth = 10;
road(scenario,roadcenters,roadwidth);
```

Add two straight roads with the default width, using road center points at each end.

```
roadcenters = [700 0 0; 100 0 0];
road(scenario,roadcenters)
```

```
ans = 
  Road with properties:

           Name: ""
         RoadID: 2
    RoadCenters: [2x3 double]
      RoadWidth: 6
      BankAngle: [2x1 double]
```

```
roadcenters = [400 400 0; 0 0 0];
road(scenario,roadcenters)
```

```
ans =
  Road with properties:

           Name: ""
         RoadID: 3
    RoadCenters: [2x3 double]
      RoadWidth: 6
      BankAngle: [2x1 double]
```

Get the road boundaries.

```
rbdry = roadBoundaries(scenario);
```

Add a car and a bicycle to the scenario. Position the car at the beginning of the first straight road.

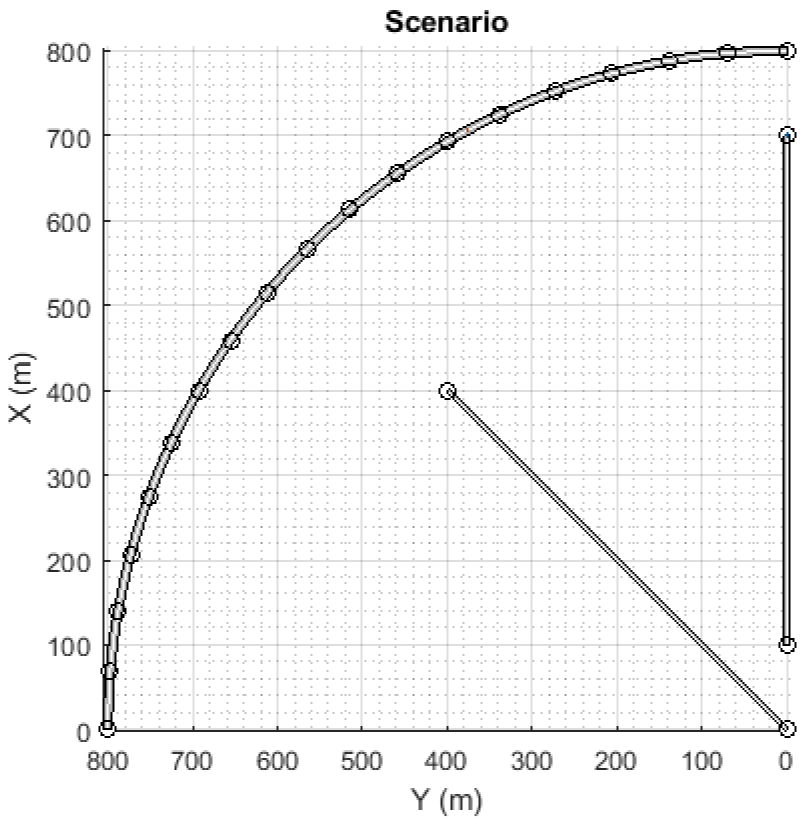
```
car = vehicle(scenario,'ClassID',1,'Position',[700 0 0], ...
    'Length',3,'Width',2,'Height',1.6);
```

Position the bicycle farther down the road.

```
bicycle = actor(scenario,'ClassID',3,'Position',[706 376 0]', ...
    'Length',2,'Width',0.45,'Height',1.5);
```

Plot the scenario.

```
plot(scenario,'Centerline','on','RoadCenters','on');
title('Scenario');
```



Display the actor poses and profiles.

```
poses = actorPoses(scenario)
```

```
poses=2×1 struct array with fields:
    ActorID
    Position
    Velocity
    Roll
    Pitch
    Yaw
    AngularVelocity
```

```
profiles = actorProfiles(scenario)
```

```
profiles=2×1 struct array with fields:
    ActorID
    ClassID
    Length
    Width
    Height
    OriginOffset
    MeshVertices
    MeshFaces
    RCSPattern
    RCSAzimuthAngles
```

```
RCSElevationAngles
```

Input Arguments

`scenario` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Output Arguments

`profiles` — Actor profiles
structure | array of structures

Actor profiles, returned as a structure or as an array of structures. Each structure contains the physical and radar characteristics of an actor.

The actor profile structures have these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`ClassID`	Classification identifier, specified as a nonnegative integer. `0` represents an object of an unknown or unassigned class.
`Length`	Length of actor, specified as a positive real-valued scalar. Units are in meters.
`Width`	Width of actor, specified as a positive real-valued scalar. Units are in meters.
`Height`	Height of actor, specified as a positive real-valued scalar. Units are in meters.
`OriginOffset`	Offset of actor's rotational center from its geometric center, specified as a real-valued vector of the form [*x*, *y*, *z*]. The rotational center, or origin, is located at the bottom center of the actor. For vehicles, the rotational center is the point on the ground beneath the center of the rear axle. Units are in meters.
`MeshVertices`	Mesh vertices of actor, specified as an *n*-by-*3* real-valued matrix of vertices. Each row in the matrix defines a point in 3-D space.
`MeshFaces`	Mesh faces of actor, specified as an *m*-by-*3* matrix of integers. Each row of `MeshFaces` represents a triangle defined by the vertex IDs, which are the row numbers of vertices.

Field	Description
RCSPattern	Radar cross-section (RCS) pattern of actor, specified as a numel(RCSElevationAngles)-by-numel(RCSAzimuthAngles) real-valued matrix. Units are in decibels per square meter.
RCSAzimuthAngles	Azimuth angles corresponding to rows of RCSPattern, specified as a vector of values in the range [–180, 180]. Units are in degrees.
RCSElevationAngles	Elevation angles corresponding to rows of RCSPattern, specified as a vector of values in the range [–90, 90]. Units are in degrees.

For full definitions of these structure fields, see the actor and vehicle functions.

See Also

Objects
drivingScenario | lidarPointCloudGenerator | radarDetectionGenerator | visionDetectionGenerator

Functions
actor | actorPoses | targetOutlines | targetPoses | vehicle

Introduced in R2017a

vehicle

Package:

Add vehicle to driving scenario

Syntax

```
vc = vehicle(scenario)
vc = vehicle(scenario,Name,Value)
```

Description

`vc = vehicle(scenario)` adds a `Vehicle` object, `vc`, to the driving scenario, `scenario`. The vehicle has default property values.

Vehicles are a specialized type of actor cuboid (box-shaped) object that has four wheels. For more details about how vehicles are defined, see “Actor and Vehicle Positions and Dimensions” on page 4-356.

`vc = vehicle(scenario,Name,Value)` sets vehicle properties using one or more name-value pairs. For example, you can set the position, velocity, dimensions, orientation, and wheelbase of the vehicle. You can also set a time for the vehicle to spawn or despawn in the scenario.

Note You can configure the vehicles in a driving scenario to spawn and despawn, and then import the associated `drivingScenario` object into the **Driving Scenario Designer** app. The app considers the first vehicle created in the driving scenario to be the ego vehicle and does not allow the ego vehicle to either spawn or despawn in the scenario.

Examples

Create Driving Scenario with Multiple Actors and Roads

Create a driving scenario containing a curved road, two straight roads, and two actors: a car and a bicycle. Both actors move along the road for 60 seconds.

Create the driving scenario object.

```
scenario = drivingScenario('SampleTime',0.1','StopTime',60);
```

Create the curved road using road center points following the arc of a circle with an 800-meter radius. The arc starts at 0°, ends at 90°, and is sampled at 5° increments.

```
angs = [0:5:90]';
R = 800;
roadcenters = R*[cosd(angs) sind(angs) zeros(size(angs))];
roadwidth = 10;
road(scenario,roadcenters,roadwidth);
```

Add two straight roads with the default width, using road center points at each end.

```
roadcenters = [700 0 0; 100 0 0];
road(scenario,roadcenters)
```

```
ans = 
  Road with properties:

           Name: ""
         RoadID: 2
    RoadCenters: [2x3 double]
      RoadWidth: 6
      BankAngle: [2x1 double]
```

```
roadcenters = [400 400 0; 0 0 0];
road(scenario,roadcenters)
```

```
ans = 
  Road with properties:

           Name: ""
         RoadID: 3
    RoadCenters: [2x3 double]
      RoadWidth: 6
      BankAngle: [2x1 double]
```

Get the road boundaries.

```
rbdry = roadBoundaries(scenario);
```

Add a car and a bicycle to the scenario. Position the car at the beginning of the first straight road.

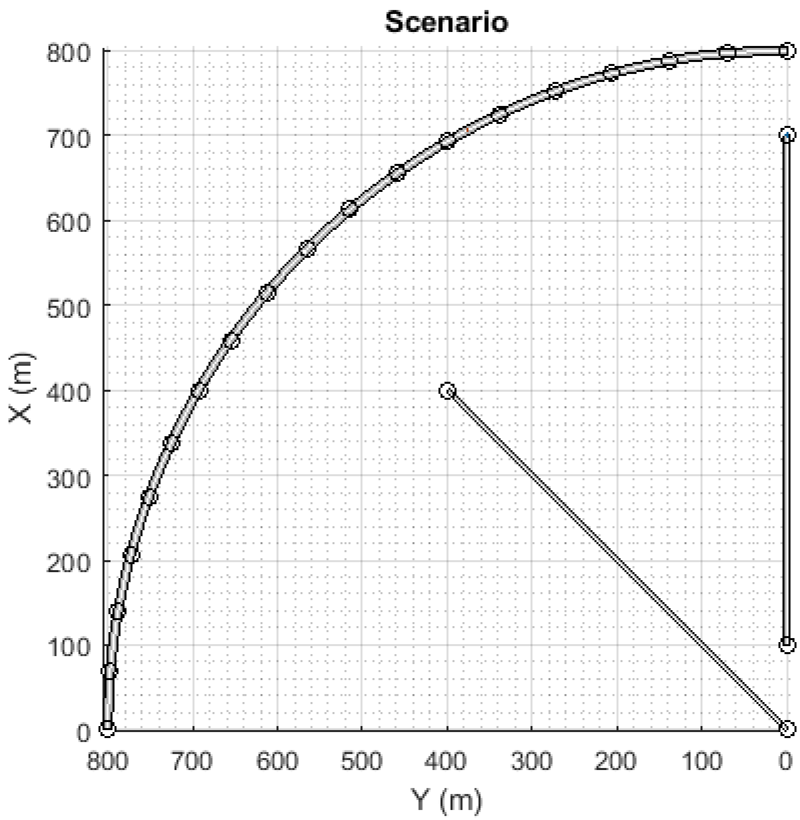
```
car = vehicle(scenario,'ClassID',1,'Position',[700 0 0], ...
    'Length',3,'Width',2,'Height',1.6);
```

Position the bicycle farther down the road.

```
bicycle = actor(scenario,'ClassID',3,'Position',[706 376 0]', ...
    'Length',2,'Width',0.45,'Height',1.5);
```

Plot the scenario.

```
plot(scenario,'Centerline','on','RoadCenters','on');
title('Scenario');
```



Display the actor poses and profiles.

```
poses = actorPoses(scenario)
```

```
poses=2×1 struct array with fields:
    ActorID
    Position
    Velocity
    Roll
    Pitch
    Yaw
    AngularVelocity
```

```
profiles = actorProfiles(scenario)
```

```
profiles=2×1 struct array with fields:
    ActorID
    ClassID
    Length
    Width
    Height
    OriginOffset
    MeshVertices
    MeshFaces
    RCSPattern
    RCSAzimuthAngles
```

```
    RCSElevationAngles
```

Spawn and Despawn Vehicles in Scenario During Simulation

Create a driving scenario. Set the stop time for the scenario to 3 seconds.

```
scenario = drivingScenario('StopTime',3);
```

Add a two-lane road to the scenario.

```
roadCenters = [0 1 0; 53 1 0];
laneSpecification = lanespec([1 1]);
road(scenario,roadCenters,'Lanes',laneSpecification);
```

Add another road that intersects the first road at a right angle to form a T-shape.

```
roadCenters = [20.3 33.4 0; 20 3 0];
laneSpecification = lanespec(2);
road(scenario,roadCenters,'Lanes',laneSpecification)
```

```
ans = 
  Road with properties:

           Name: ""
         RoadID: 2
    RoadCenters: [2x3 double]
      RoadWidth: 7.3500
      BankAngle: [2x1 double]
```

Add the ego vehicle to the scenario and define its waypoints. Set the ego vehicle speed to 20 m/s and generate the trajectories for the ego vehicle.

```
egoVehicle = vehicle(scenario,'ClassID',1, ...
                    'Position',[1.5 2.5 0]);
waypoints = [2 3 0; 13 3 0;
            21 3 0; 31 3 0;
            43 3 0; 47 3 0];
speed = 15;
trajectory(egoVehicle,waypoints,speed)
```

Add a non-ego vehicle to the scenario. Set the non-ego vehicle to spawn and despawn during the simulation by specifying an entry time and an exit time.

```
nonEgovehicle1 = vehicle(scenario,'ClassID',1, ...
                'Position',[22 30 0],'EntryTime',0.8,'ExitTime',2);
```

Define the waypoints for the non-ego vehicle. Set the non-ego vehicle speed to 35 m/s and generate its trajectories.

```
waypoints = [22 30 0; 22 23 0;
            22 13 0; 22 7 0;
            18 -0.3 0; 12 -0.8 0; 3 -0.8 0];
speed = 35;
trajectory(nonEgovehicle1,waypoints,speed)
```

Add another non-ego vehicle to the scenario. Set the second non-ego vehicle to spawn during the simulation by specifying an entry time. Since you do not specify an exit time, this vehicle will remain in the scenario until the scenario ends.

```
nonEgovehicle2 = vehicle(scenario,'ClassID',1, ...
                'Position',[48 -1 0],'EntryTime',2);
```

Define the waypoints for the second non-ego vehicle. Set the vehicle speed to 60 m/s and generate its trajectories.

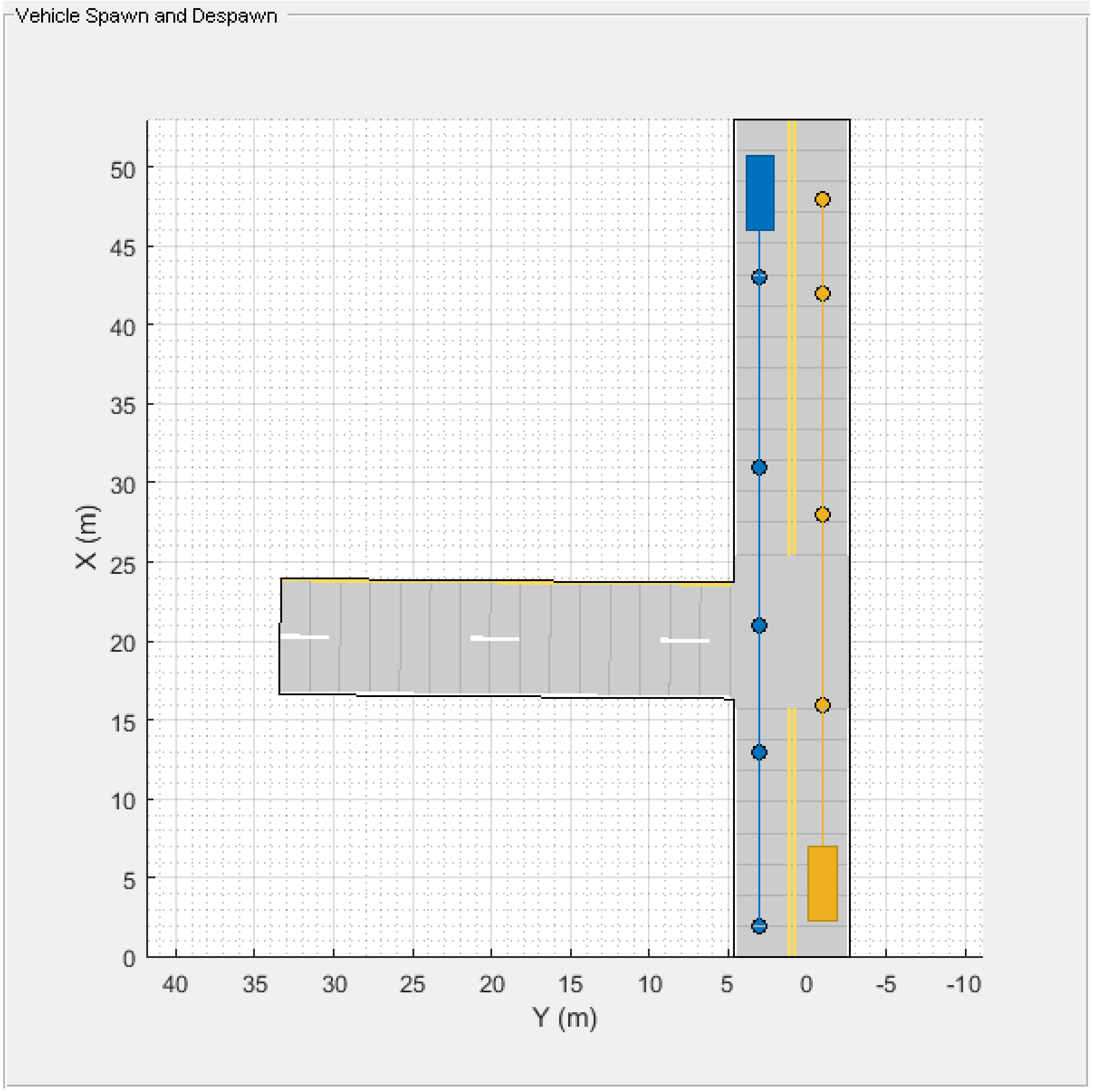
```
waypoints = [48 -1 0; 42 -1 0; 28 -1 0;
            16 -1 0; 6 -1 0];
speed = 60;
trajectory(nonEgovehicle2,waypoints,speed)
```

Create a custom figure window to plot the scenario.

```
fig = figure;
set(fig,'Position',[0 0 600 600])
movegui(fig,'center')
hViewPnl = uipanel(fig,'Position',[0 0 1 1],'Title','Vehicle Spawn and Despawn');
hPlt = axes(hViewPnl);
```

Plot the scenario and run the simulation. Observe how the non-ego vehicles spawn and despawn in the scenario while simulation is running.

```
plot(scenario,'Waypoints','on','Parent',hPlt)
while advance(scenario)
    pause(0.1)
end
```



Input Arguments

`scenario` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `vehicle('Length',2.2,'Width',0.6,'Height',1.5)` creates a vehicle that has the dimensions of a motorcycle. Units are in meters.

ClassID — Classification identifier
`0` (default) | nonnegative integer

Classification identifier of actor, specified as the comma-separated pair consisting of `'ClassID'` and a nonnegative integer.

Specify `ClassID` values to group together actors that have similar dimensions, radar cross-section (RCS) patterns, or other properties. As a best practice, before adding actors to a `drivingScenario` object, determine the actor classification scheme you want to use. Then, when creating the actors, specify the `ClassID` name-value pair to set classification identifiers according to the actor classification scheme.

Suppose you want to create a scenario containing these actors:

- Two cars, one of which is the ego vehicle
- A truck
- A bicycle

The code shows a sample classification scheme for this scenario, where `1` refers to cars, `2` refers to trucks, and `3` refers to bicycles. The cars have default vehicle properties. The truck and bicycle have the dimensions of a typical truck and bicycle, respectively.

```
scenario = drivingScenario;
ego = vehicle(scenario,'ClassID',1);
car = vehicle(scenario,'ClassID',1);
truck = vehicle(scenario,'ClassID',2,'Length',8.2,'Width',2.5,'Height',3.5);
bicycle = actor(scenario,'ClassID',3,'Length',1.7,'Width',0.45,'Height',1.7);
```

The default `ClassID` of `0` is reserved for an object of an unknown or unassigned class. If you plan to import `drivingScenario` objects into the **Driving Scenario Designer** app, do not leave the `ClassID` property of actors set to `0`. The app does not recognize a `ClassID` of `0` for actors and returns an error. Instead, set `ClassID` values of actors according to the actor classification scheme used in the app.

ClassID	Class Name
1	Car
2	Truck
3	Bicycle
4	Pedestrian
5	Barrier

Name — Name of vehicle
`""` (default) | character vector | string scalar

Name of the vehicle, specified as the comma-separated pair consisting of `'Name'` and a character vector or string scalar.

Example: `'Name','Vehicle1'`

Example: `"Name","Vehicle1"`

Data Types: `char` | `string`

`EntryTime` — Entry time for vehicle to spawn
`0` (default) | positive scalar

Entry time for a vehicle to spawn in the driving scenario, specified as the comma-separated pair consisting of `'EntryTime'` and a positive scalar. Units are in seconds, measured from the start time of the scenario.

Specify this name-value pair argument to add or make a vehicle appear in the driving scenario at the specified time, while the simulation is running.

- If the vehicle has an associated exit time, then the entry time must be less than the specified exit time.
- If the vehicle does not have an associated exit time, then the entry time must be less than or equal to the stop time of the scenario. You can set the stop time for the scenario by specifying a value for the `'StopTime'` property of the `drivingScenario` object.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ExitTime` — Exit time for vehicle to despawn
`Inf` (default) | positive scalar

Exit time for a vehicle to despawn from the driving scenario, specified as the comma-separated pair consisting of `'ExitTime'` and a positive scalar. Units are in seconds, measured from the start time of the scenario.

Specify this name-value pair argument to remove or make a vehicle disappear from the scenario at a specified time while the simulation is running.

- If the vehicle has an associated entry time, then the exit time must be greater than the specified entry time.
- If the vehicle does not have an associated entry time, then the exit time must be less than or equal to the stop time of the scenario. You can set the stop time for the scenario by specifying a value for the `'StopTime'` property of the `drivingScenario` object.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`PlotColor` — Display color of vehicle
RGB triplet | hexadecimal color code | color name | short color name

Display color of vehicle, specified as the comma-separated pair consisting of `'PlotColor'` and an RGB triplet, hexadecimal color code, color name, or short color name.

The vehicle appears in the specified color in all programmatic scenario visualizations, including the `plot` function, `chasePlot` function, and plotting functions of `birdsEyePlot` objects. If you import the scenario into the **Driving Scenario Designer** app, then the vehicle appears in this color in all app visualizations. If you import the scenario into Simulink, then the vehicle appears in this color in the **Bird's-Eye Scope**.

If you do not specify a color for the vehicle, the function assigns one based on the default color order of `Axes` objects. For more details, see the `ColorOrder` property for `Axes` objects.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Position — Position of vehicle center

[0 0 0] (default) | [*x y z*] real-valued vector

Position of the rotational center of the vehicle, specified as the comma-separated pair consisting of 'Position' and an [*x y z*] real-valued vector.

The rotational center of a vehicle is the midpoint of its rear axle. The vehicle extends rearward by a distance equal to the rear overhang. The vehicle extends forward by a distance equal to the sum of the wheelbase and forward overhang. Units are in meters.

Example: [10;50;0]

Velocity — Velocity of vehicle center

[0 0 0] (default) | [v_x v_y v_z] real-valued vector

Velocity (v) of the vehicle center in the x-, y- and z-directions, specified as the comma-separated pair consisting of `'Velocity'` and a [v_x v_y v_z] real-valued vector. The `'Position'` name-value pair specifies the vehicle center. Units are in meters per second.

Example: `[-4;7;10]`

`Yaw` — Yaw angle of vehicle

`0` (default) | real scalar

Yaw angle of the vehicle, specified as the comma-separated pair consisting of `'Yaw'` and a real scalar. Yaw is the angle of rotation of the vehicle around the z-axis. Yaw is clockwise-positive when looking in the forward direction of the axis, which points up from the ground. Therefore, when viewing vehicles from the top down, such as on a bird's-eye plot, yaw is counterclockwise-positive. Angle values are wrapped to the range [–180, 180]. Units are in degrees.

Example: `-0.4`

`Pitch` — Pitch angle of vehicle

`0` (default) | real scalar

Pitch angle of the vehicle, specified as the comma-separated pair consisting of `'Pitch'` and a real scalar. Pitch is the angle of rotation of the vehicle around the y-axis and is clockwise-positive when looking in the forward direction of the axis. Angle values are wrapped to the range [–180, 180]. Units are in degrees.

Example: `5.8`

`Roll` — Roll angle of vehicle

`0` (default) | real scalar

Roll angle of the vehicle, specified as the comma-separated pair consisting of `'Roll'` and a real scalar. Roll is the angle of rotation of the vehicle around the x-axis and is clockwise-positive when looking in the forward direction of the axis. Angle values are wrapped to the range [–180, 180]. Units are in degrees.

Example: `-10`

`AngularVelocity` — Angular velocity of vehicle

`[0 0 0]` (default) | [ω_x ω_y ω_z] real-valued vector

Angular velocity (ω) of the vehicle, in world coordinates, specified as the comma-separated pair consisting of `'AngularVelocity'` and a [ω_x ω_y ω_z] real-valued vector. Units are in degrees per second.

Example: `[20 40 20]`

`Length` — Length of vehicle

`4.7` (default) | positive real scalar

Length of the vehicle, specified as the comma-separated pair consisting of `'Length'` and a positive real scalar. Units are in meters.

In `Vehicle` objects, this equation defines the values of the `Length`, `FrontOverhang`, `Wheelbase`, and `RearOverhang` properties:

```
Length = FrontOverhang + Wheelbase + RearOverhang
```

- If you update the `Length`, `RearOverhang`, or `Wheelbase` property, to maintain the equation, the `Vehicle` object increases or decreases the `FrontOverhang` property and keeps the other properties constant.
- If you update the `FrontOverhang` property, to maintain this equation, the `Vehicle` object increases or decreases the `Wheelbase` property and keeps the other properties constant.

When setting both the `FrontOverhang` and `RearOverhang` properties, to prevent the `Vehicle` object from overriding the `FrontOverhang` value, set `RearOverhang` first, followed by `FrontOverhang`. The object calculates the new `Wheelbase` property value automatically.

Example: `5.5`

`Width` — Width of vehicle
`1.8` (default) | positive real scalar

Width of the vehicle, specified as the comma-separated pair consisting of `'Width'` and a positive real scalar. Units are in meters.

Example: `2.0`

`Height` — Height of vehicle
`1.4` (default) | positive real scalar

Height of the vehicle, specified as the comma-separated pair consisting of `'Height'` and a positive real scalar. Units are in meters.

Example: `2.1`

`Mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`RCSPattern` — Radar cross-section pattern of vehicle
`[10 10; 10 10]` (default) | *Q*-by-*P* real-valued matrix

Radar cross-section (RCS) pattern of the vehicle, specified as the comma-separated pair consisting of `'RCSPattern'` and a *Q*-by-*P* real-valued matrix. RCS is a function of the azimuth and elevation angles, where:

- *Q* is the number of elevation angles specified by the `'RCSElevationAngles'` name-value pair.
- *P* is the number of azimuth angles specified by the `'RCSAzimuthAngles'` name-value pair.

Units are in decibels per square meter (dBsm).

Example: `5.8`

`RCSAzimuthAngles` — Azimuth angles of vehicle's RCS pattern
`[-180 180]` (default) | *P*-element real-valued vector

Azimuth angles of the vehicle's RCS pattern, specified as the comma-separated pair consisting of `'RCSAzimuthAngles'` and a *P*-element real-valued vector. *P* is the number of azimuth angles. Values are in the range [–180°, 180°].

Each element of `RCSAzimuthAngles` defines the azimuth angle of the corresponding column of the `'RCSPattern'` name-value pair. Units are in degrees.

Example: [-90:90]

RCSElevationAngles — Elevation angles of vehicle's RCS pattern
[-90 90] (default) | *Q*-element real-valued vector

Elevation angles of the vehicle's RCS pattern, specified as the comma-separated pair consisting of 'RCSElevationAngles' and a *Q*-element real-valued vector. *Q* is the number of elevation angles. Values are in the range [–90°, 90°].

Each element of RCSElevationAngles defines the elevation angle of the corresponding row of the 'RCSPattern' name-value pair. Units are in degrees.

Example: [0:90]

FrontOverhang — Front overhang of vehicle
0.9 (default) | real scalar

Front overhang of the vehicle, specified as the comma-separated pair consisting of 'FrontOverhang' and a real scalar. The front overhang is the distance that the vehicle extends beyond the front axle. If the vehicle does not extend past the front axle, then the front overhang is negative. Units are in meters.

In Vehicle objects, this equation defines the values of the Length, FrontOverhang, Wheelbase, and RearOverhang properties:

```
Length = FrontOverhang + Wheelbase + RearOverhang
```

- If you update the Length, RearOverhang, or Wheelbase property, to maintain the equation, the Vehicle object increases or decreases the FrontOverhang property and keeps the other properties constant.
- If you update the FrontOverhang property, to maintain this equation, the Vehicle object increases or decreases the Wheelbase property and keeps the other properties constant.

When setting both the FrontOverhang and RearOverhang properties, to prevent the Vehicle object from overriding the FrontOverhang value, set RearOverhang first, followed by FrontOverhang. The object calculates the new Wheelbase property value automatically.

Example: 0.37

RearOverhang — Rear overhang of vehicle
1.0 (default) | real scalar

Rear overhang of the vehicle, specified as the comma-separated pair consisting of 'RearOverhang' and a real scalar. The rear overhang is the distance that the vehicle extends beyond the rear axle. If the vehicle does not extend past the rear axle, then the rear overhang is negative. Negative rear overhang is common in semitrailer trucks, where the cab of the truck does not overhang the rear wheel. Units are in meters.

In Vehicle objects, this equation defines the values of the Length, FrontOverhang, Wheelbase, and RearOverhang properties:

```
Length = FrontOverhang + Wheelbase + RearOverhang
```

- If you update the Length, RearOverhang, or Wheelbase property, to maintain the equation, the Vehicle object increases or decreases the FrontOverhang property and keeps the other properties constant.

- If you update the FrontOverhang property, to maintain this equation, the Vehicle object increases or decreases the Wheelbase property and keeps the other properties constant.

When setting both the FrontOverhang and RearOverhang properties, to prevent the Vehicle object from overriding the FrontOverhang value, set RearOverhang first, followed by FrontOverhang. The object calculates the new Wheelbase property value automatically.

Example: 0.32

Wheelbase — Distance between vehicle axles
2.8 (default) | positive real scalar

Distance between the front and rear axles of a vehicle, specified as the comma-separated pair consisting of 'Wheelbase' and a positive real scalar. Units are in meters.

In Vehicle objects, this equation defines the values of the Length, FrontOverhang, Wheelbase, and RearOverhang properties:

```
Length = FrontOverhang + Wheelbase + RearOverhang
```

- If you update the Length, RearOverhang, or Wheelbase property, to maintain the equation, the Vehicle object increases or decreases the FrontOverhang property and keeps the other properties constant.
- If you update the FrontOverhang property, to maintain this equation, the Vehicle object increases or decreases the Wheelbase property and keeps the other properties constant.

When setting both the FrontOverhang and RearOverhang properties, to prevent the Vehicle object from overriding the FrontOverhang value, set RearOverhang first, followed by FrontOverhang. The object calculates the new Wheelbase property value automatically.

Example: 1.51

Output Arguments

vc — Driving scenario vehicle
Vehicle object

Driving scenario vehicle, returned as a Vehicle object belonging to the driving scenario specified in scenario.

You can modify the Vehicle object by changing its property values. The property names correspond to the name-value pair arguments used to create the object.

The only property that you cannot modify is ActorID, which is a positive integer indicating the unique, scenario-defined ID of the vehicle.

To specify and visualize vehicle motion, use these functions:

trajectory	Create actor or vehicle trajectory in driving scenario
chasePlot	Ego-centric projective perspective plot

To get information about vehicle characteristics, use these functions:

`actorPoses`	Positions, velocities, and orientations of actors in driving scenario
`actorProfiles`	Physical and radar characteristics of actors in driving scenario
`targetOutlines`	Outlines of targets viewed by actor
`targetPoses`	Target positions and orientations relative to ego vehicle
`driving.scenario.targetsToEgo`	Convert target actor poses from world coordinates of scenario to ego vehicle coordinates
`driving.scenario.targetsToScenario`	Convert target actor poses from ego vehicle coordinates to world coordinates of scenario

To get information about the roads and lanes that the vehicle is on, use these functions:

`roadBoundaries`	Get road boundaries
`driving.scenario.roadBoundariesToEgo`	Convert road boundaries to ego vehicle coordinates
`currentLane`	Get current lane of actor
`laneBoundaries`	Get lane boundaries of actor lane
`laneMarkingVertices`	Lane marking vertices and faces in driving scenario
`roadMesh`	Mesh representation of an actor's nearest roads in driving scenario.

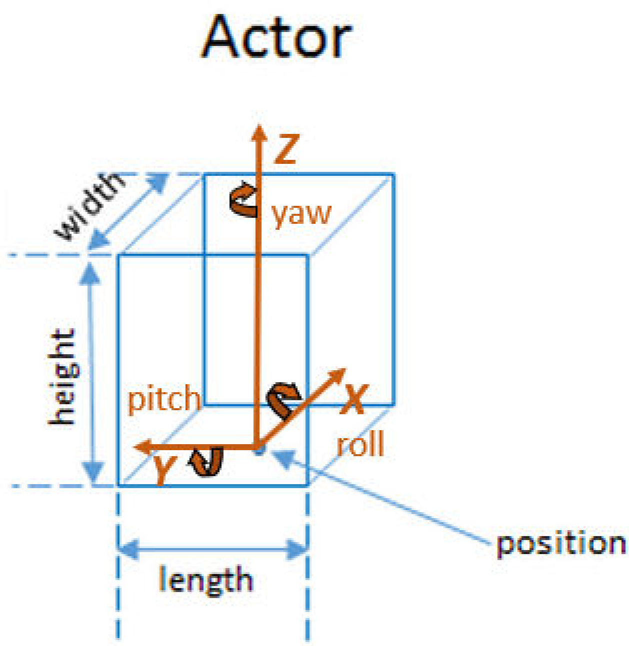
More About

Actor and Vehicle Positions and Dimensions

In driving scenarios, an actor is a cuboid (box-shaped) object with a specific length, width, and height. Actors also have a radar cross-section (RCS) pattern, specified in dBsm, which you can refine by setting angular azimuth and elevation coordinates. The position of an actor is defined as the center of its bottom face. This center point is used as the actor's rotational center, its point of contact with the ground, and its origin in its local coordinate system. In this coordinate system:

- The *X*-axis points forward from the actor.
- The *Y*-axis points left from the actor.
- 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.



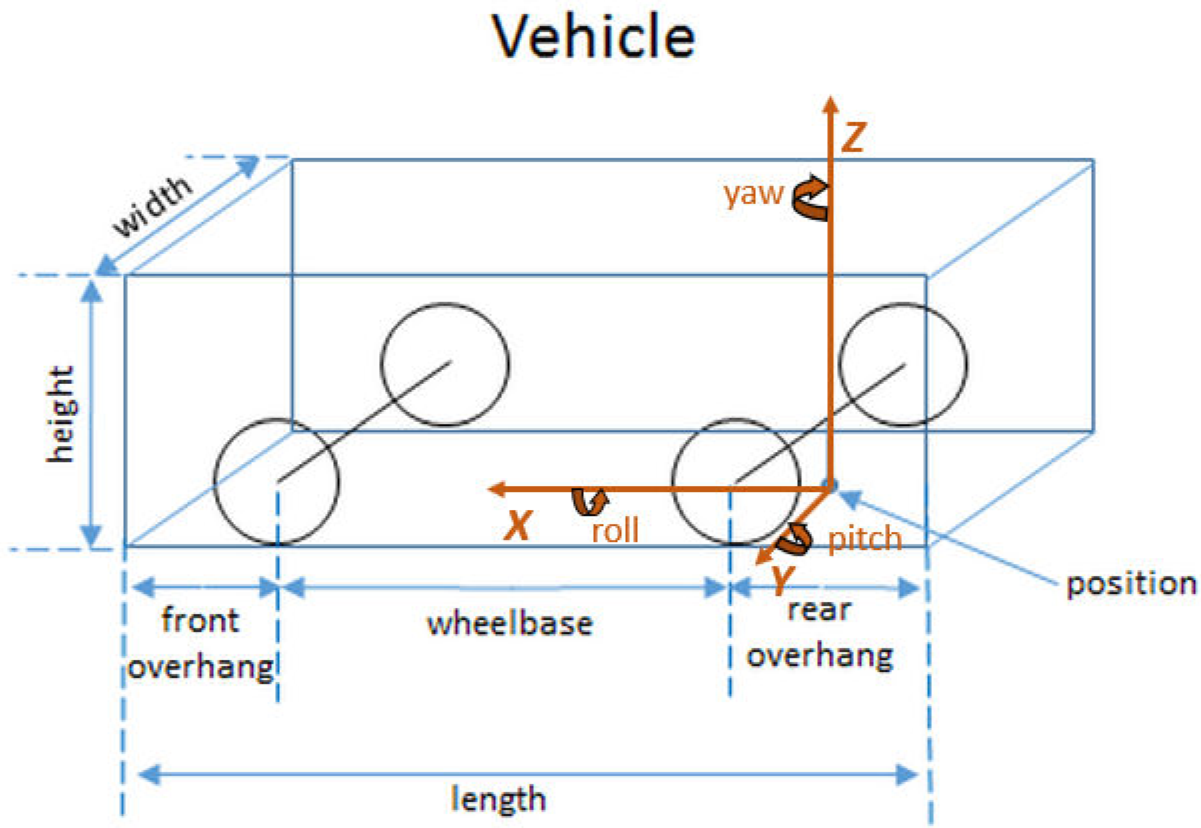
A vehicle is an actor that moves on wheels. Vehicles have three extra properties that govern the placement of their front and rear axle.

- Wheelbase — Distance between the front and rear axles
- Front overhang — Distance between the front of the vehicle and the front axle
- Rear overhang — Distance between the rear axle and the rear of the vehicle

Unlike other types of actors, the position of a vehicle is defined by the point on the ground that is below the center of its rear axle. This point corresponds to the natural center of rotation of the vehicle. As with nonvehicle actors, this point is the origin in the local coordinate system of the vehicle, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points left from 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.



This table shows a list of common actors and their dimensions. To specify these values in `Actor` and `Vehicle` objects, set the corresponding properties shown.

Actor Classification	Actor Object	Actor Properties						
		`Length`	`Width`	`Height`	`FrontOverhang`	`RearOverhang`	`Wheelbase`	`RCSPattern`
Pedestrian	`Actor`	0.24 m	0.45 m	1.7 m	N/A	N/A	N/A	-8 dBsm
Car	`Vehicle`	4.7 m	1.8 m	1.4 m	0.9 m	1.0 m	2.8 m	10 dBsm
Motorcycle	`Vehicle`	2.2 m	0.6 m	1.5 m	0.37 m	0.32 m	1.51 m	0 dBsm

See Also
`actor` | `drivingScenario`

Topics
"Create Driving Scenario Programmatically"
"Create Actor and Vehicle Trajectories Programmatically"

Introduced in R2017a

chasePlot

Package:

Ego-centric projective perspective plot

Syntax

```
chasePlot(ac)
chasePlot(ac,Name,Value)
```

Description

`chasePlot(ac)` plots a driving scenario from the perspective of actor `ac`. This plot is called a chase plot and has an ego-centric projective perspective, where the view is positioned immediately behind the actor.

`chasePlot(ac,Name,Value)` specifies options using one or more name-value pairs. For example, you can display road centers and actor waypoints on the plot.

Examples

Simulate Car Traveling on S-Curve

Simulate a driving scenario with one car traveling on an S-curve. Create and plot the lane boundaries.

Create the driving scenario with one road having an S-curve.

```
scenario = drivingScenario('StopTime',3);
roadcenters = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
```

Create the lanes and add them to the road.

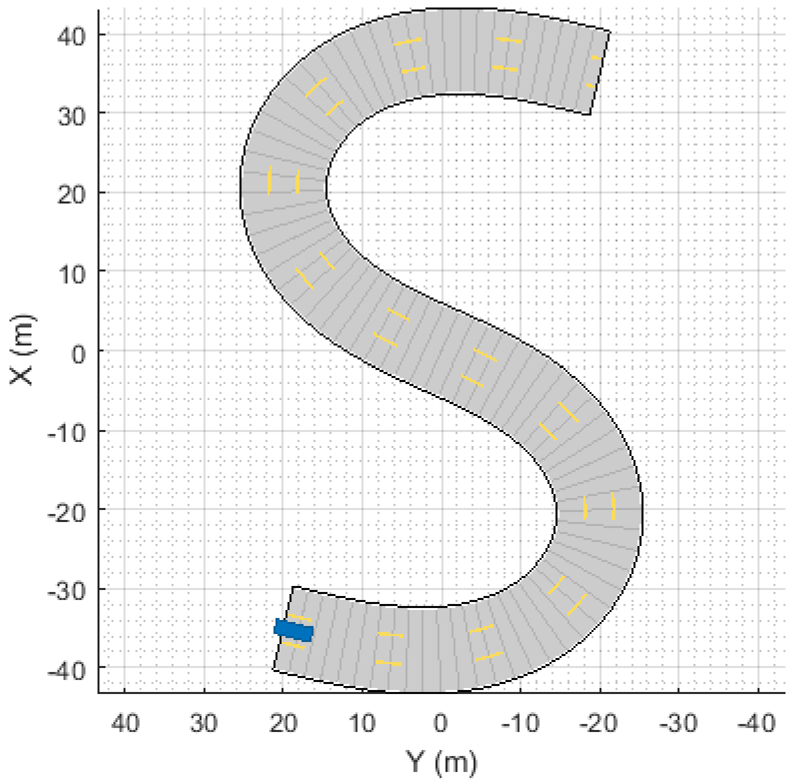
```
lm = [laneMarking('Solid','Color','w'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Solid','Color','w')];
ls = lanespec(3,'Marking',lm);
road(scenario,roadcenters,'Lanes',ls);
```

Add an ego vehicle and specify its trajectory from its waypoints. By default, the car travels at a speed of 30 meters per second.

```
car = vehicle(scenario, ...
    'ClassID',1, ...
    'Position',[-35 20 0]);
waypoints = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
trajectory(car,waypoints);
```

Plot the scenario and corresponding chase plot.

```
plot(scenario)
```

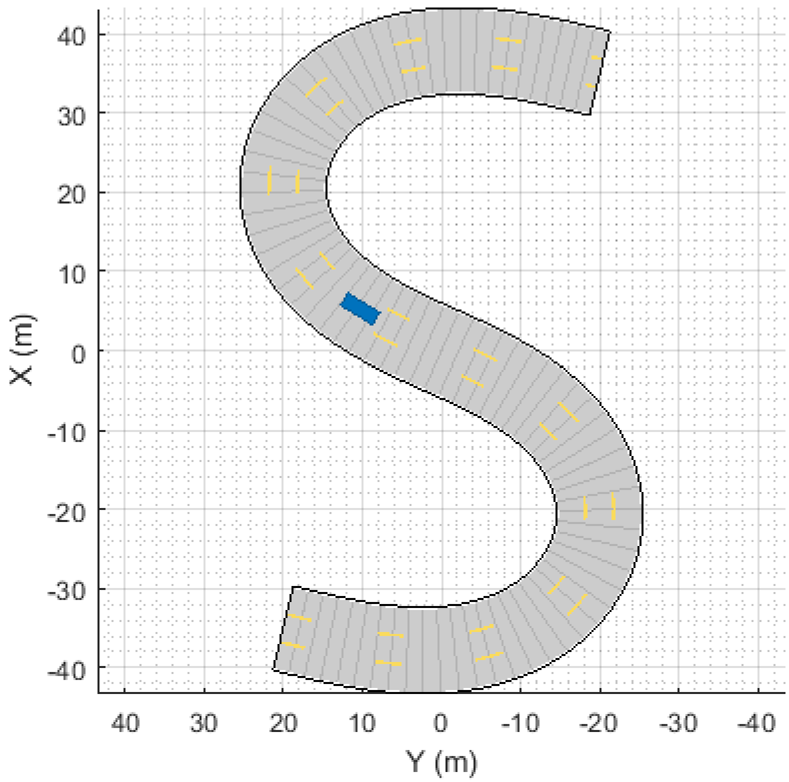



```
chasePlot(car)
```

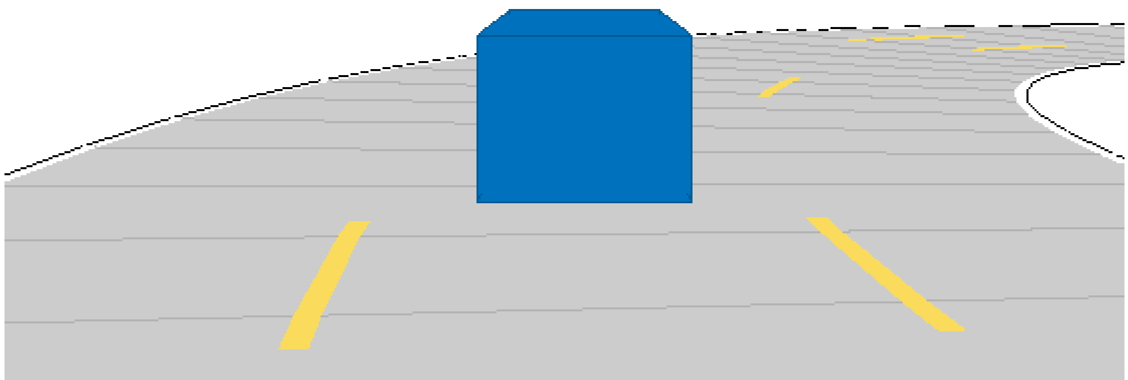
Run the simulation loop.

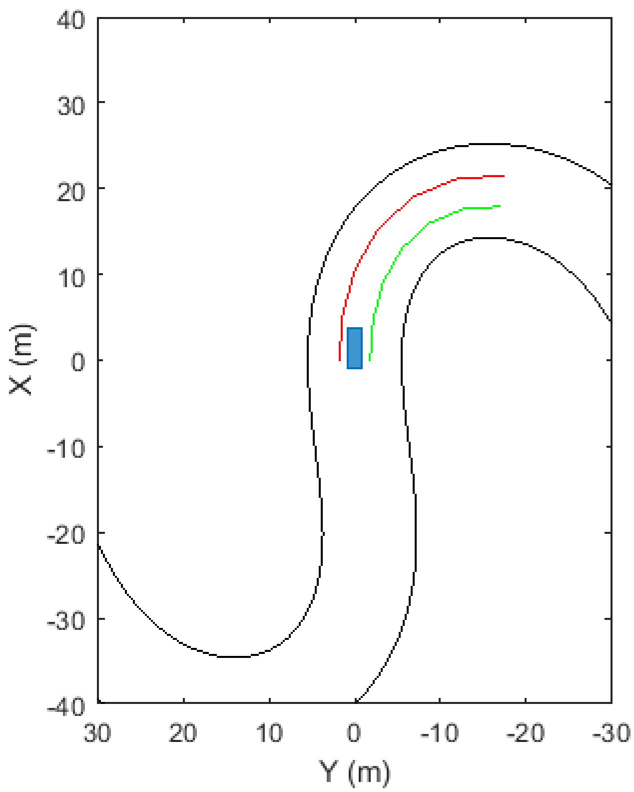
1. Initialize a bird's-eye plot and create an outline plotter, left-lane and right-lane boundary plotters, and a road boundary plotter.
2. Obtain the road boundaries and rectangular outlines.
3. Obtain the lane boundaries to the left and right of the vehicle.
4. Advance the simulation and update the plotters.

```
bep = birdsEyePlot('XLim',[-40 40],'YLim',[-30 30]);
olPlotter = outlinePlotter(bep);
lblPlotter = laneBoundaryPlotter(bep,'Color','r','LineStyle','-');
lbrPlotter = laneBoundaryPlotter(bep,'Color','g','LineStyle','-');
rbsEdgePlotter = laneBoundaryPlotter(bep);
legend('off');
while advance(scenario)
    rbs = roadBoundaries(car);
    [position,yaw,length,width,originOffset,color] = targetOutlines(car);
    lb = laneBoundaries(car,'XDistance',0:5:30,'LocationType','Center', ...
        'AllBoundaries',false);
    plotLaneBoundary(rbsEdgePlotter,rbs)
    plotLaneBoundary(lblPlotter,{lb(1).Coordinates})
    plotLaneBoundary(lbrPlotter,{lb(2).Coordinates})
    plotOutline(olPlotter,position,yaw,length,width, ...
        'OriginOffset',originOffset,'Color',color)
end
```



40
30
20
10
0
-10
-20
-30
-40
X (m)
40
30
20
10
0
-10
-20
-30
-40
Y (m)





Show Target Outlines in Driving Scenario Simulation

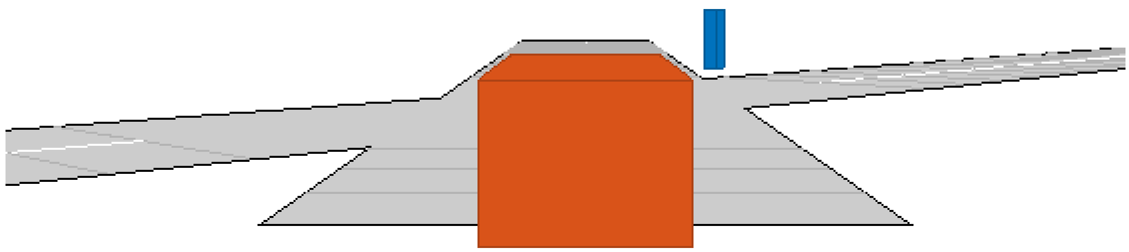
Create a driving scenario and show how target outlines change as the simulation advances.

Create a driving scenario consisting of two intersecting straight roads. The first road segment is 45 meters long. The second straight road is 32 meters long and intersects the first road. A car traveling at 12.0 meters per second along the first road approaches a running pedestrian crossing the intersection at 2.0 meters per second.

```
scenario = drivingScenario('SampleTime',0.1,'StopTime',1);
road(scenario,[-10 0 0; 45 -20 0]);
road(scenario,[-10 -10 0; 35 10 0]);
ped = actor(scenario,'ClassID',4,'Length',0.4,'Width',0.6,'Height',1.7);
car = vehicle(scenario,'ClassID',1);
pedspeed = 2.0;
carspeed = 12.0;
trajectory(ped,[15 -3 0; 15 3 0],pedspeed);
trajectory(car,[-10 -10 0; 35 10 0],carspeed);
```

Create an ego-centric chase plot for the vehicle.

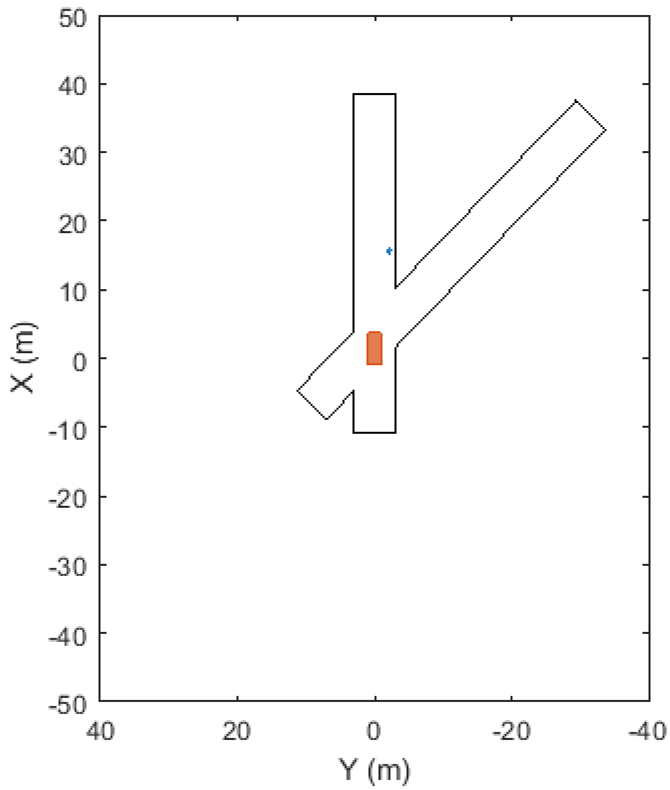
```
chasePlot(car,'Centerline','on')
```

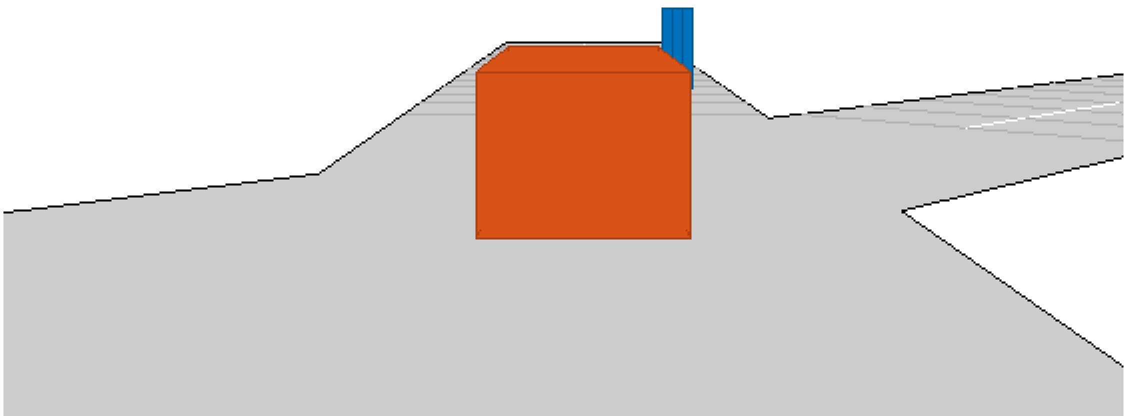


Create an empty bird's-eye plot and add an outline plotter and lane boundary plotter. Then, run the simulation. At each simulation step:

- Update the chase plot to display the road boundaries and target outlines.
- Update the bird's-eye plot to display the updated road boundaries and target outlines. The plot perspective is always with respect to the ego vehicle.

```
bepPlot = birdsEyePlot('XLim',[-50 50],'YLim',[-40 40]);
outlineplotter = outlinePlotter(bepPlot);
laneplotter = laneBoundaryPlotter(bepPlot);
legend('off')

while advance(scenario)
    rb = roadBoundaries(car);
    [position,yaw,length,width,originOffset,color] = targetOutlines(car);
    plotLaneBoundary(laneplotter,rb)
    plotOutline(outlineplotter,position,yaw,length,width, ...
        'OriginOffset',originOffset,'Color',color)
    pause(0.01)
end
```


50
40
30
20
10
0
-10
-20
-30
-40
-50
X (m)
40
20
0
-20
-40
Y (m)



Input Arguments

ac — Actor
`Actor` object | `Vehicle` object

Actor belonging to a `drivingScenario` object, specified as an `Actor` or `Vehicle` object. To create these objects, use the `actor` and `vehicle` functions, respectively.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `chasePlot(ac,'Centerline','on','RoadCenters,'on')` displays the center line and road centers of each road segment.

`Parent` — Axes in which to draw plot
`Axes` object

Axes in which to draw the plot, specified as the comma-separated pair consisting of `'Parent'` and an `Axes` object. If you do not specify `Parent`, a new figure is created.

`Centerline` — Display center line of roads
`'off'` (default) | `'on'`

Display the center line of roads, specified as the comma-separated pair consisting of 'Centerline' and 'off' or 'on'. The center line follows the middle of each road segment. Center lines are discontinuous through areas such as intersections or road splits.

RoadCenters — Display road centers
'off' (default) | 'on'

Display road centers, specified as the comma-separated pair consisting of 'RoadCenters' and 'off' or 'on'. The road centers define the roads shown in the plot.

Waypoints — Display actor waypoints
'off' (default) | 'on'

Display actor waypoints, specified as the comma-separated pair consisting of 'Waypoints' and 'off' or 'on'. Waypoints define the trajectory of the actor.

Meshes — Display actor meshes
'off' (default) | 'on'

Display actor meshes instead of cuboids, specified as the comma-separated pair consisting of 'Meshes' and 'off' or 'on'.

ViewHeight — Height of plot viewpoint
1.5 × actor height (default) | positive real scalar

Height of the plot viewpoint, specified as the comma-separated pair consisting of 'ViewHeight' and a positive real scalar. The height is with respect to the bottom of the actor. Units are in meters.

ViewLocation — Location of plot viewpoint
2.5 × actor length (default) | [*x*, *y*] real-valued vector

Location of the plot viewpoint, specified as the comma-separated pair consisting of 'ViewLocation' and an [*x*, *y*] real-valued vector. The location is with respect to the cuboid center in the coordinate system of the actor. The default location of the viewpoint is behind the cuboid center, [2.5*actor.Length 0]. Units are in meters.

ViewRoll — Roll angle orientation of plot viewpoint
0 (default) | real scalar

Roll angle orientation of the plot viewpoint, specified as the comma-separated pair consisting of 'ViewRoll' and a real scalar. Units are in degrees.

ViewPitch — Pitch angle orientation of plot viewpoint
0 (default) | real scalar

Pitch angle orientation of the plot viewpoint, specified as the comma-separated pair consisting of 'ViewPitch' and a real scalar. Units are in degrees.

ViewYaw — Yaw angle orientation of plot viewpoint
0 (default) | real scalar

Yaw angle orientation of the plot viewpoint, specified as the comma-separated pair consisting of 'ViewYaw' and a real scalar. Units are in degrees.

See Also

Objects
drivingScenario

Functions
actor | plot | road | trajectory | vehicle

Topics
"Create Driving Scenario Programmatically"

Introduced in R2017a

trajectory

Package:

Create actor or vehicle trajectory in driving scenario

Syntax

```
trajectory(ac,waypoints)
trajectory(ac,waypoints,speed)
trajectory(ac,waypoints,speed,waittime)
trajectory( ___ ,'Yaw',yaw)
```

Description

`trajectory(ac,waypoints)` creates a trajectory for an actor or vehicle, `ac`, from a set of waypoints.

`trajectory(ac,waypoints,speed)` also specifies the speed with which the actor or vehicle travels along the trajectory, in either forward or reverse motion.

`trajectory(ac,waypoints,speed,waittime)` specifies the wait time for an actor or vehicle in addition to the input arguments in the previous syntax. Use this syntax to generate stop-and-go driving scenarios by pausing an actor or vehicle actors at specific waypoints.

`trajectory( ___ ,'Yaw',yaw)` specifies the yaw orientation angle of the actor or vehicle at each waypoint, in addition to any of the input argument combinations from preceding syntaxes.

Examples

Simulate Car Traveling on S-Curve

Simulate a driving scenario with one car traveling on an S-curve. Create and plot the lane boundaries.

Create the driving scenario with one road having an S-curve.

```
scenario = drivingScenario('StopTime',3);
roadcenters = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
```

Create the lanes and add them to the road.

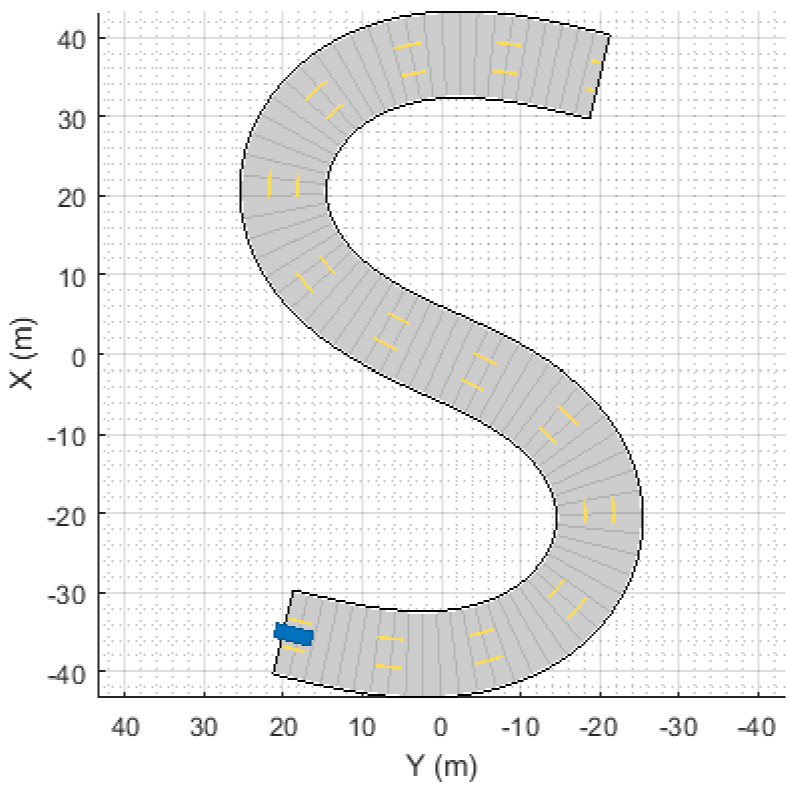
```
lm = [laneMarking('Solid','Color','w'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Solid','Color','w')];
ls = lanespec(3,'Marking',lm);
road(scenario,roadcenters,'Lanes',ls);
```

Add an ego vehicle and specify its trajectory from its waypoints. By default, the car travels at a speed of 30 meters per second.

```
car = vehicle(scenario, ...
    'ClassID',1, ...
    'Position',[-35 20 0]);
waypoints = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
trajectory(car,waypoints);
```

Plot the scenario and corresponding chase plot.

```
plot(scenario)
```

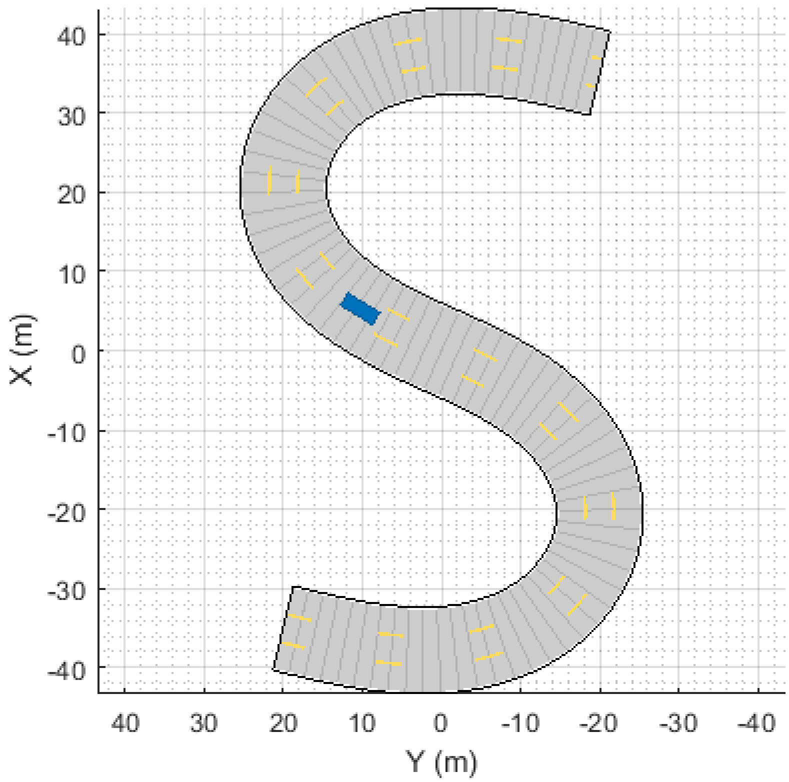


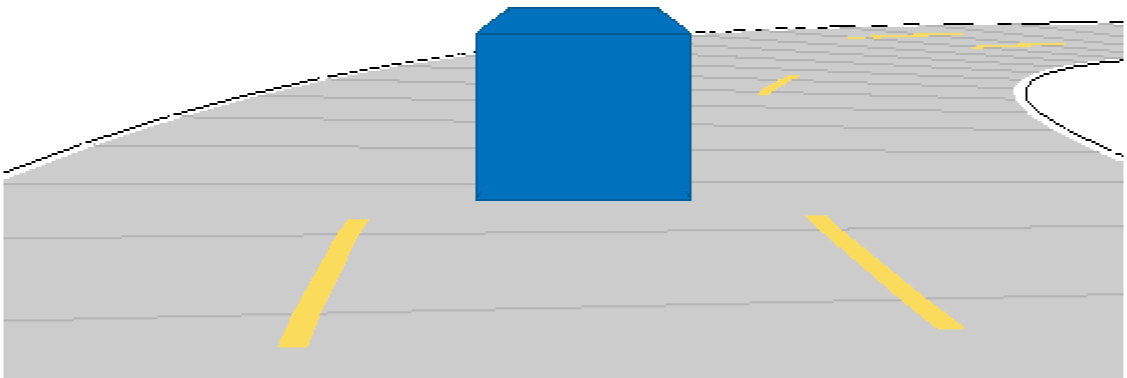

```
chasePlot(car)
```

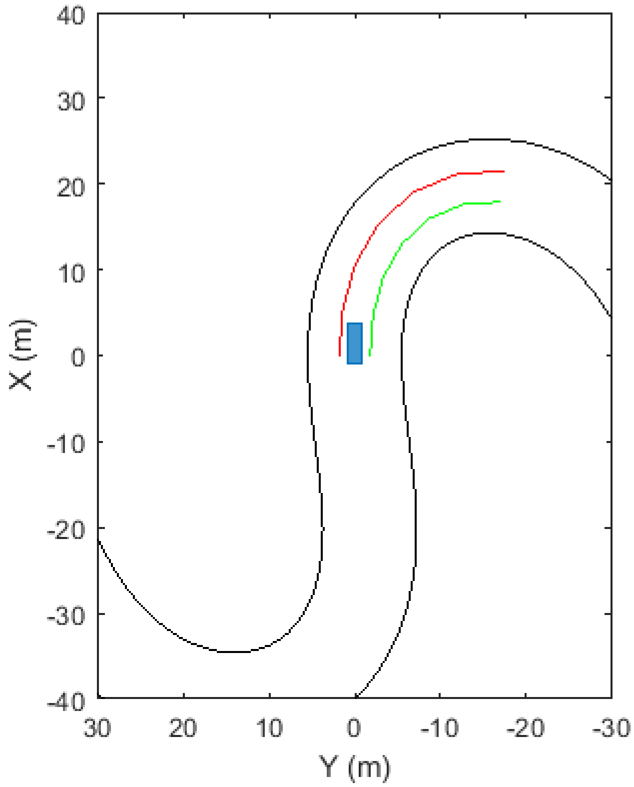
Run the simulation loop.

1. Initialize a bird's-eye plot and create an outline plotter, left-lane and right-lane boundary plotters, and a road boundary plotter.
2. Obtain the road boundaries and rectangular outlines.
3. Obtain the lane boundaries to the left and right of the vehicle.
4. Advance the simulation and update the plotters.

```
bep = birdsEyePlot('XLim',[-40 40],'YLim',[-30 30]);
olPlotter = outlinePlotter(bep);
lblPlotter = laneBoundaryPlotter(bep,'Color','r','LineStyle','-');
lbrPlotter = laneBoundaryPlotter(bep,'Color','g','LineStyle','-');
rbsEdgePlotter = laneBoundaryPlotter(bep);
legend('off');
while advance(scenario)
    rbs = roadBoundaries(car);
    [position,yaw,length,width,originOffset,color] = targetOutlines(car);
    lb = laneBoundaries(car,'XDistance',0:5:30,'LocationType','Center', ...
        'AllBoundaries',false);
    plotLaneBoundary(rbsEdgePlotter,rbs)
    plotLaneBoundary(lblPlotter,{lb(1).Coordinates})
    plotLaneBoundary(lbrPlotter,{lb(2).Coordinates})
    plotOutline(olPlotter,position,yaw,length,width, ...
        'OriginOffset',originOffset,'Color',color)
end
```


40
30
20
10
0
-10
-20
-30
-40
X (m)
40
30
20
10
0
-10
-20
-30
-40
Y (m)





Simulate Vehicle with Trajectory of Varying Speeds

Create a driving scenario and add a curved two-lane road to it.

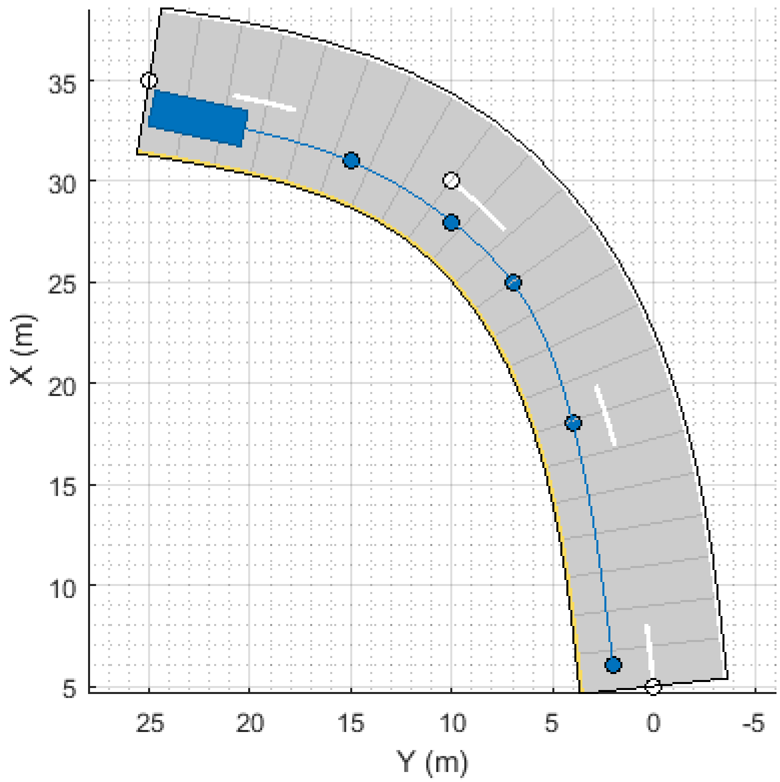
```
scenario = drivingScenario('SampleTime',0.05);
roadcenters = [5 0; 30 10; 35 25];
lspec = lanespec(2);
road(scenario,roadcenters,'Lanes',lspec);
```

Add a vehicle to the scenario. Set a trajectory in which the vehicle drives around the curve at varying speeds.

```
v = vehicle(scenario,'ClassID',1);
waypoints = [6 2; 18 4; 25 7; 28 10; 31 15; 33 22];
speeds = [30 10 5 5 10 30];
trajectory(v,waypoints,speeds)
```

Plot the scenario and run the simulation. Observe how the vehicle slows down as it drives along the curve.

```
plot(scenario,'Waypoints','on','RoadCenters','on')
while advance(scenario)
    pause(0.1)
end
```



Generate Stop-and-Go Driving Scenario

Create a driving scenario consisting of two, two-lane roads that intersect at a right angle.

```
scenario = drivingScenario('StopTime',2.75);
roadCenters = [50 1 0; 2 0.9 0];
laneSpecification = lanespec(2,'Width',4);
road(scenario,roadCenters,'Lanes',laneSpecification);
roadCenters = [27 24 0; 27 -21 0];
road(scenario,roadCenters,'Lanes',laneSpecification);
```

Add an ego vehicle to the scenario. Specify the waypoints and the speed values for the vehicle at each waypoint. Set a wait time for the vehicle at the second waypoint. Generate a trajectory in which the ego vehicle travels through the specified waypoints at the specified speed.

```
egoVehicle = vehicle(scenario,'ClassID',1,'Position',[5 -1 0]);
waypoints = [5 -1 0; 16 -1 0; 40 -1 0];
speed = [30; 0; 30];
waittime = [0; 0.3; 0];
trajectory(egoVehicle,waypoints,speed,waittime);
```

Add a car to the scenario. Specify the waypoints and the speed values for the car at each waypoint. Set a wait time for the car at the second waypoint. Generate a trajectory in which the car travels through the specified waypoints at the specified speed.

```
car = vehicle(scenario,'ClassID',1,'Position',[48 4 0],'PlotColor',[0.494 0.184 0.556], 'Name','(
waypoints = [47 3 0; 38 3 0; 10 3 0];
speed = [30; 0; 30];
waittime = [0; 0.3; 0];
trajectory(car,waypoints,speed,waittime);
```

Add an ambulance to the scenario. Generate a trajectory in which the ambulance travels through the specified waypoints at a constant speed.

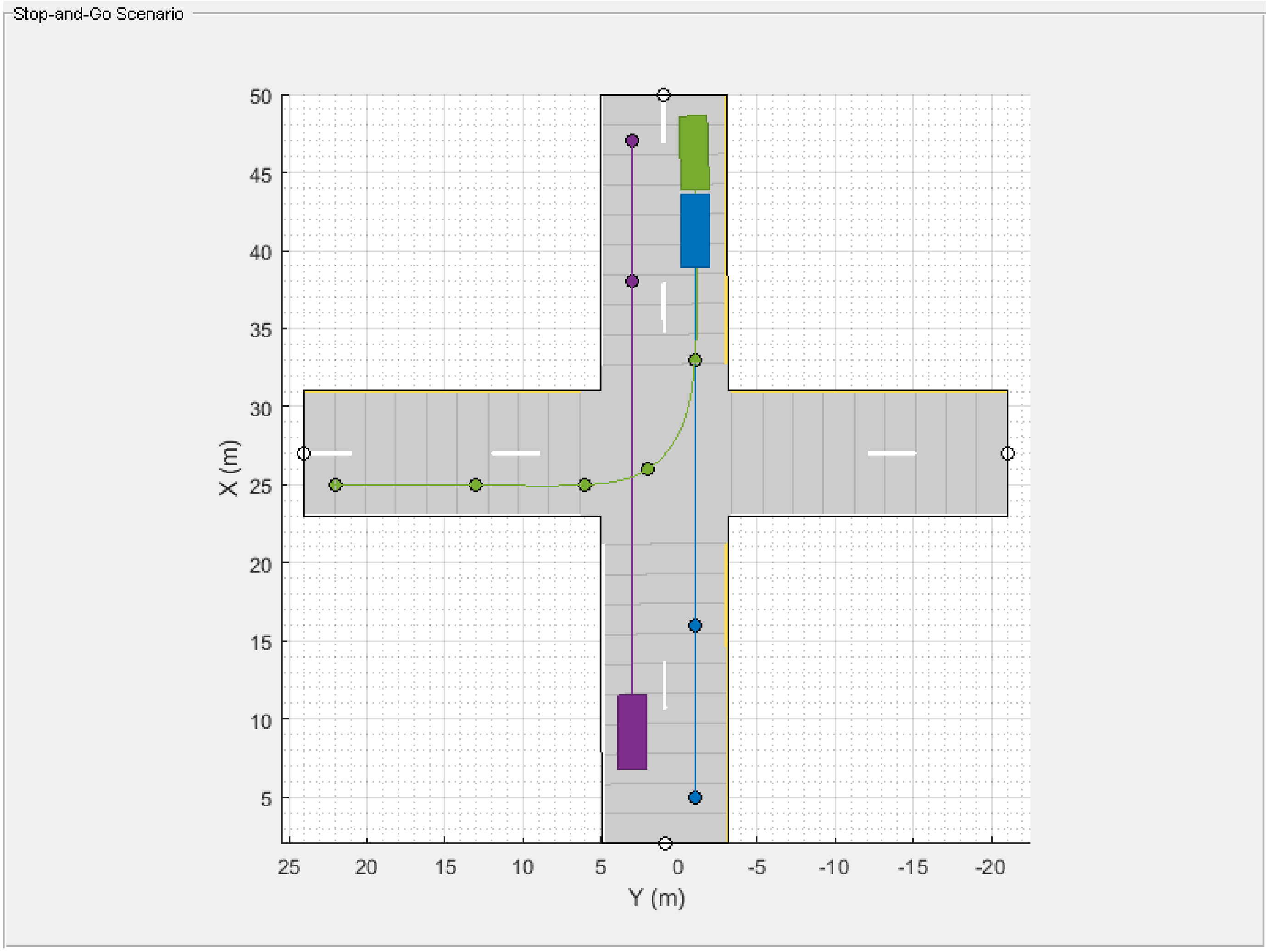
```
ambulance = vehicle(scenario,'ClassID',6,'Position',[25 22 0],'PlotColor',[0.466 0.674 0.188],'Na
waypoints = [25 22 0; 25 13 0; 25 6 0; 26 2 0; 33 -1 0; 45 -1 0];
speed = 25;
trajectory(ambulance,waypoints,speed);
```

Create a custom figure window to plot the scenario.

```
fig = figure;
set(fig,'Position',[0,0,800,600]);
movegui(fig,'center');
hViewPnl = uipanel(fig,'Position',[0 0 1 1],'Title','Stop-and-Go Scenario');
hPlt = axes(hViewPnl);
```

Plot the scenario and run the simulation. The ego vehicle and the car pause for their specified wait times to avoid collision with the ambulance.

```
plot(scenario,'Waypoints','on','RoadCenters','on','Parent',hPlt)
while advance(scenario)
    pause(0.1)
end
```



Simulate Vehicle Backing into Parking Spot

Simulate a driving scenario in which a car drives in reverse to back into a parking spot.

Create a driving scenario. Add road segments to define a parking lot. The first road segment defines the parking spaces. The second road segment defines the driving lane and overlays the first road segment.

```
scenario = drivingScenario;

roadCentersParking = [6 0; 24 0];
lmParking = [laneMarking('Unmarked') ...
            repmat(laneMarking('Solid'),1,5) ...
            laneMarking('Unmarked')];
lspecParking = lanespec(6,'Width',3,'Marking',lmParking);
road(scenario,roadCentersParking,'Lanes',lspecParking);

roadCentersDriving = [12 0; 18 0];
```

```
lmDriving = [laneMarking('Unmarked') laneMarking('Unmarked')];
lspecDriving = lanespec(1,'Width',18,'Marking',lmDriving);
road(scenario,roadCentersDriving,'Lanes',lspecDriving);
```

Add a vehicle to the driving scenario.

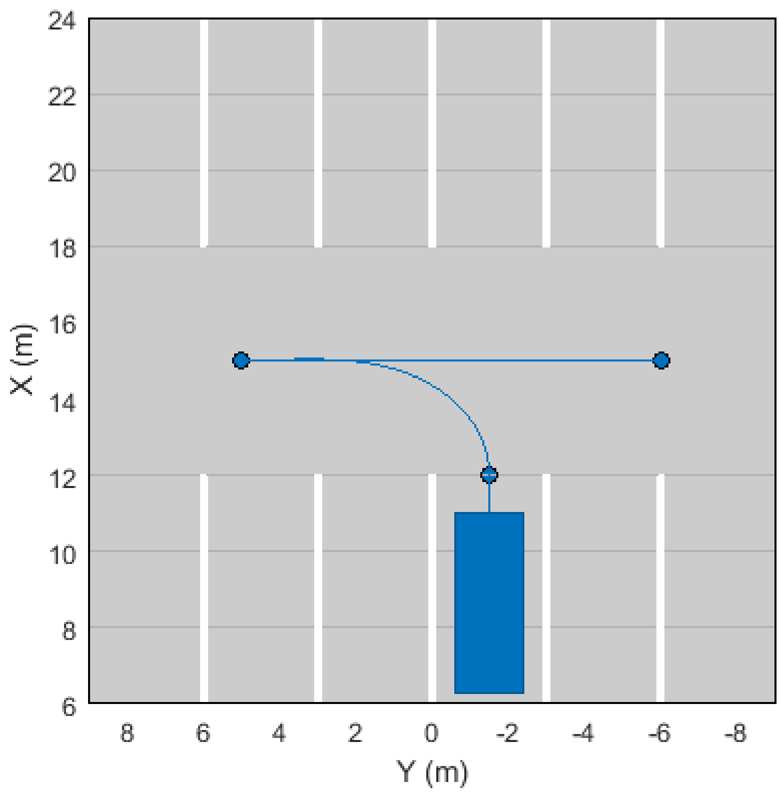
```
car = vehicle(scenario,'ClassID',1,'Position',[15 -6 0],'Yaw',90);
```

Define the trajectory of the vehicle. The vehicle drives forward, stops, and then drives in reverse until it backs into the parking spot. As the vehicle enters the parking spot, it has a yaw orientation angle that is 90 degrees counterclockwise from where it started.

```
waypoints = [15 -6; 15 5; 12 -1.5; 7.3 -1.5];
speed = [4.5; 0; -2; 0];
trajectory(car,waypoints,speed,'Yaw',[90 90 180 180]);
```

Plot the driving scenario and display the waypoints of the trajectory.

```
plot(scenario,'Waypoints','on')
while advance(scenario)
    pause(0.001)
end
```



Define Trajectory of Pedestrian

Define the trajectory of a pedestrian who takes a sharp right turn at an intersection.

Create a driving scenario. Add road segments that define an intersection.

```
scenario = drivingScenario;
roadCenters = [0 10; 0 -10];
road(scenario,roadCenters);
road(scenario,flip(roadCenters,2));
```

Add a pedestrian actor to the scenario.

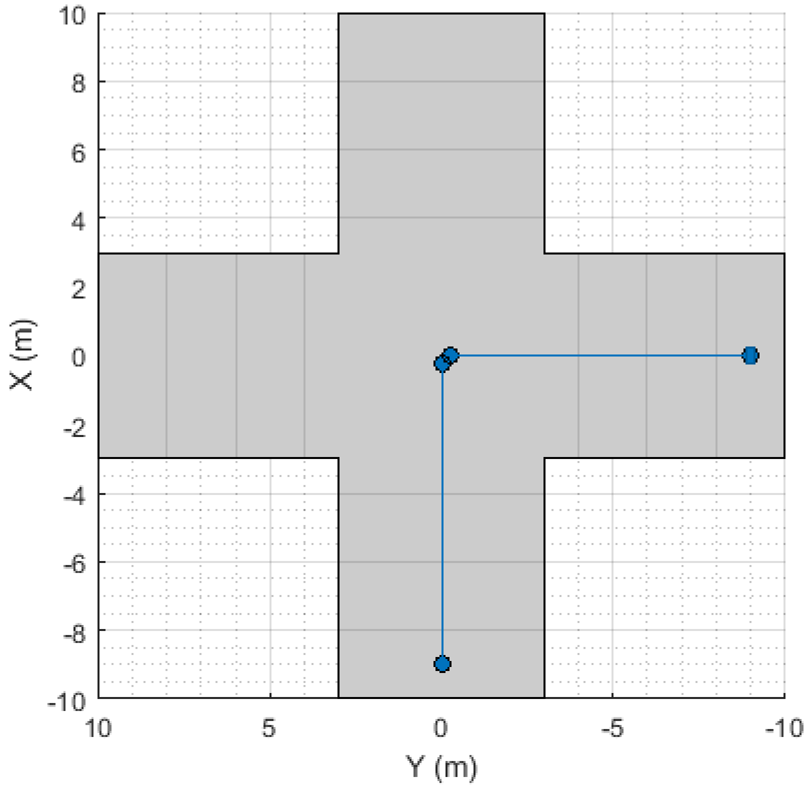
```
pedestrian = actor(scenario, ...
    'ClassID',4, ...
    'Length',0.24, ...
    'Width',0.45, ...
    'Height',1.7, ...
    'Position',[-9 0 0], ...
    'RCSPattern',[-8 -8; -8 -8], ...
    'Mesh', driving.scenario.pedestrianMesh, ...
    'Name','Pedestrian');
```

Define the trajectory of the pedestrian. The pedestrian approaches the intersection, pauses briefly, and then take a sharp right turn at the intersection. To define the sharp right turn, specify two waypoints at the intersection that are close together. For these waypoints, specify the yaw orientation angle of the second waypoint at a 90-degree angle from the first waypoint.

```
waypoints = [-9 0; -0.25 0; 0 -0.25; 0 -9];
speed = [1.5; 0; 0.5; 1.5];
yaw =  [0; 0; -90; -90];
waittime = [0; 0.2; 0; 0];
trajectory(pedestrian,waypoints,speed,waittime,'Yaw', yaw);
```

Plot the driving scenario and display the waypoints of the pedestrian.

```
plot(scenario,'Waypoints','on')
while advance(scenario)
    pause(0.001)
end
```



Input Arguments

ac — Actor
Actor object | Vehicle object

Actor belonging to a `drivingScenario` object, specified as an `Actor` or `Vehicle` object. To create these objects, use the `actor` and `vehicle` functions, respectively.

`waypoints` — Trajectory waypoints
real-valued *N*-by-2 matrix | real-valued *N*-by-3 matrix

Trajectory waypoints, specified as a real-valued *N*-by-2 or *N*-by-3 matrix, where *N* is the number of waypoints.

- If `waypoints` is an *N*-by-2 matrix, then each matrix row represents the (*x*, *y*) coordinates of a waypoint. The *z*-coordinate of each waypoint is zero.
- If `waypoints` is an *N*-by-3 matrix, then each matrix row represents the (*x*, *y*, *z*) coordinates of a waypoint.

Waypoints are in the world coordinate system. Units are in meters.

Example: `[1 0 0; 2 7 7; 3 8 8]`

Data Types: `single` | `double`

speed — Actor speed
30.0 | real-valued scalar | *N*-element real-valued vector

Actor speed at each waypoint in `waypoints`, specified as a real-valued scalar or *N*-element real-valued vector. *N* is the number of waypoints.

- When `speed` is a scalar, the speed is constant throughout the actor motion.
- When `speed` is a vector, the vector values specify the speed at each waypoint. For forward motion, specify positive speed values. For reverse motion, specify negative speed values. To change motion directions, separate the positive speeds and negative speeds by a waypoint with `0` speed.

Speeds are interpolated between waypoints. `speed` can be zero at any waypoint but cannot be zero at two consecutive waypoints. Units are in meters per second.

Example: `[10 8 9]` specifies speeds of 10 m/s, 8 m/s, and 9 m/s.

Example: `[10 0 -10]` specifies a speed of 10 m/s in forward motion, followed by a pause, followed by a speed of 10 m/s in reverse.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

waittime — Pause time for actor
0 (default) | *N*-element vector of nonnegative values

Pause time for the actor, specified as an *N*-element vector of nonnegative values. *N* is the number of waypoints. When you specify a pause time for the actor at a particular waypoint, you must set the corresponding speed value to `0`. You can set the `waitime` to `0` at any waypoint, but you cannot set `waittime` at two consecutive waypoints to non-zero values. Units are in seconds.

Data Types: `single` | `double`

yaw — Yaw orientation angle of actor
N-element real-valued vector

Yaw orientation angle of the actor at each waypoint, specified as an *N*-element real-valued vector, where *N* is the number of waypoints. Units are in degrees and angles are positive in the counterclockwise direction.

If you do not specify `yaw`, then the yaw at each waypoint is `NaN`, meaning that the yaw has no constraints.

Example: `[0 90]` specifies an actor at a 0-degree angle at the first waypoint and a 90-degree angle at the second waypoint.

Example: `[0 NaN]` specifies an actor at a 0-degree angle at the first waypoint. The actor has no constraints on its yaw at the second waypoint.

Data Types: `single` | `double`

Algorithms

The `trajectory` function creates a trajectory for an actor to follow in a scenario. A trajectory consists of the path followed by an object and its speed along the path. You specify the path using *N* two-dimensional or three-dimensional waypoints. Each of the *N* – 1 segments between waypoints defines a curve whose curvature varies linearly with distance along the segment. The function fits a piecewise clothoid curve to the (*x*, *y*) coordinates of the waypoints by matching the curvature on both sides of the waypoint. For a nonclosed curve, the curvature at the first and last waypoint is zero. If the first and last waypoints coincide, then the curvatures before and after the endpoints are matched. The *z*-coordinates of the trajectory are interpolated using a shape-preserving piecewise cubic curve.

You can specify speed as a scalar or a vector. When speed is a scalar, the actor follows the trajectory with constant speed. When speed is an *N*-element vector, speed is linearly interpolated between waypoints. Setting the speed to zero at two consecutive waypoints creates a stationary actor.

See Also

Objects
`drivingScenario`

Functions
`actor` | `road` | `vehicle`

Topics
"Scenario Generation from Recorded Vehicle Data"
"Create Actor and Vehicle Trajectories Programmatically"
"Create Driving Scenario Programmatically"

Introduced in R2018a

targetMeshes

Package:

Mesh vertices and faces relative to specific actor

Syntax

```
[vertices,faces] = targetMeshes(ac)
[vertices,faces,colors] = targetMeshes(ac)
```

Description

`[vertices,faces] = targetMeshes(ac)` returns the mesh vertices and faces of all actors in a driving scenario relative to the specified actor, `ac`. When displaying meshes on page 4-388 by using a `birdsEyePlot` object, you can use the output mesh information as inputs to the `plotMesh` function.

`[vertices,faces,colors] = targetMeshes(ac)` also returns the color of the mesh faces for each actor.

Examples

Display Actor Meshes in Driving Scenario

Display actors in a driving scenario by using their mesh representations instead of their cuboid representations.

Create a driving scenario, and add a 25-meter straight road to the scenario.

```
scenario = drivingScenario;
roadcenters = [0 0 0; 25 0 0];
road(scenario,roadcenters);
```

Add a pedestrian and a vehicle to the scenario. Specify the mesh dimensions of the actors using prebuilt meshes.

- Specify the pedestrian mesh as a `driving.scenario.pedestrianMesh` object.
- Specify the vehicle mesh as a `driving.scenario.carMesh` object.

```
p = actor(scenario,'ClassID',4, ...
            'Length',0.2,'Width',0.4, ...
            'Height',1.7,'Mesh',driving.scenario.pedestrianMesh);

v = vehicle(scenario,'ClassID',1, ...
            'Mesh',driving.scenario.carMesh);
```

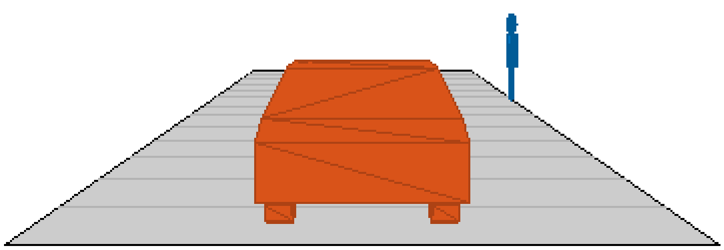
Add trajectories for the pedestrian and vehicle.

- Specify for the pedestrian to cross the road at 1 meter per second.
- Specify for the vehicle to follow the road at 10 meters per second.

```
waypointsP = [15 -3 0; 15 3 0];
speedP = 1;
trajectory(p,waypointsP,speedP);

wayPointsV = [v.RearOverhang 0 0; (25 - v.Length + v.RearOverhang) 0 0];
speedV = 10;
trajectory(v,wayPointsV,speedV)
```

Add an egocentric plot for the vehicle. Turn the display of meshes on.

```
chasePlot(v,'Meshes','on')
```

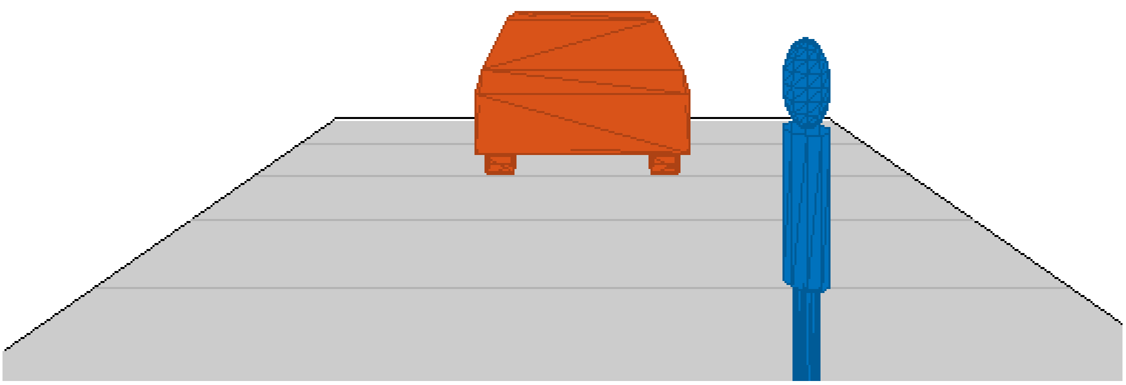


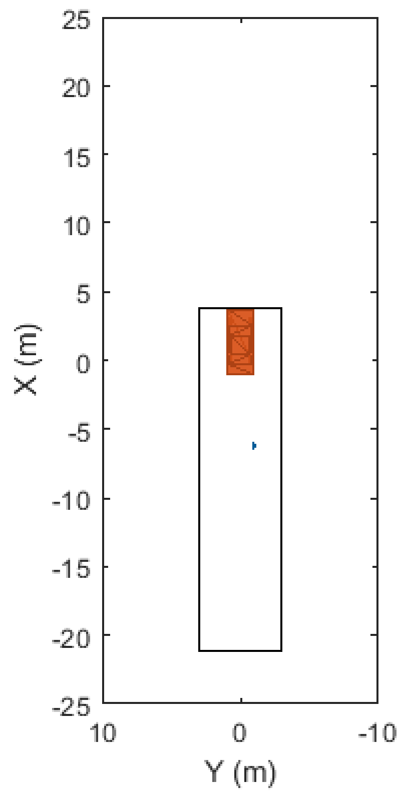
Create a bird's-eye plot in which to display the meshes. Also create a mesh plotter and lane boundary plotter. Then run the simulation loop.

1. Obtain the road boundaries of the road the vehicle is on.
2. Obtain the mesh vertices, faces, and colors of the actor meshes, with positions relative to the vehicle.
3. Plot the road boundaries and actor meshes on the bird's-eye plot.
4. Pause the scenario to allow time for the plots to update. The chase plot updates every time you advance the scenario.

```
bep = birdsEyePlot('XLim',[-25 25],'YLim',[-10 10]);
mPlotter = meshPlotter(bep);
lbPlotter = laneBoundaryPlotter(bep);
legend('off')
```

```
while advance(scenario)

   rb = roadBoundaries(v);

   [vertices,faces,colors] = targetMeshes(v);

   plotLaneBoundary(lbPlotter,rb)
   plotMesh(mPlotter,vertices,faces,'Color',colors)

   pause(0.01)
end
```





Input Arguments

ac — Actor
`Actor` object | `Vehicle` object

Actor belonging to a `drivingScenario` object, specified as an `Actor` or `Vehicle` object. To create these objects, use the `actor` and `vehicle` functions, respectively.

Output Arguments

`vertices` — Mesh vertices of each actor
N-element cell array

Mesh vertices of each actor, returned as an *N*-element cell array, where *N* is the number of actors.

Each element in `vertices` must be a *V*-by-3 real-valued matrix containing the vertices of an actor, where:

- *V* is the number of vertices.
- Each row defines the 3-D (*x*,*y*,*z*) position of a vertex. The vertex positions are relative to the position of the input actor `ac`. Units are in meters.

`faces` — Mesh faces of each actor
N-element cell array

Mesh faces of each actor, returned as an *N*-element cell array, where *N* is the number of actors.

Each element in `faces` must be an *F*-by-3 integer-valued matrix containing the faces of an actor, where:

- *F* is the number of faces.
- Each row defines a triangle of vertex IDs that make up the face. The vertex IDs correspond to row numbers within `vertices`.

Suppose the first face of the `i`th element of `faces` has these vertex IDs.

```
faces{i}(1,:)
```

```
ans =

     1     2     3
```

In the `i`th element of `vertices`, rows 1, 2, and 3 contain the (*x*, *y*, *z*) positions of the vertices that make up this face.

```
vertices{i}(1:3,:)
```

```
ans =

    3.7000    0.9000    0.8574
    3.7000   -0.9000    0.8574
    3.7000   -0.9000    0.3149
```

`colors` — Color of mesh faces for each actor
N-by-3 matrix of *N* RGB triplets

Color of the mesh faces for each actor, returned as an *N*-by-3 matrix of RGB triplets. *N* is the number of actors and is equal to the number of elements in `vertices` and `faces`.

The `i`th row of `colors` is the RGB color value of the faces in the `i`th element of `faces`. The function applies the same color to all mesh faces of an actor.

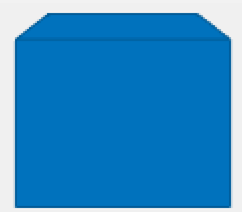
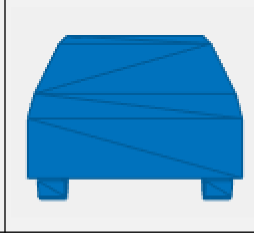
An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0, 1]. For example, `[0.4 0.6 0.7]`.

More About

Meshes

In driving scenarios, a mesh is a triangle-based 3-D representation of an object. Mesh representations of objects are more detailed than the default cuboid (box-shaped) representations of objects. Meshes are useful for generating synthetic point cloud data from a driving scenario.

This table shows the difference between a cuboid representation and a mesh representation of a vehicle in a driving scenario.

Cuboid	Mesh
	

See Also

Objects
birdsEyePlot | drivingScenario

Functions
actor | actorPoses | targetOutlines | targetPoses | vehicle

Introduced in R2020b

targetPoses

Package:

Target positions and orientations relative to ego vehicle

Syntax

```
poses = targetPoses(ac)
poses = targetPoses(ac,range)
```

Description

`poses = targetPoses(ac)` returns the poses of all targets in a driving scenario with respect to the ego vehicle actor, `ac`. See “Ego Vehicle and Targets” on page 4-395 for more details.

`poses = targetPoses(ac,range)` returns the poses of targets that are within a specified range around the ego vehicle actor.

Examples

Convert Target Poses Between Ego Vehicle and Scenario Coordinates

In a simple driving scenario, obtain the poses of target vehicles in the coordinate system of the ego vehicle. Then convert these poses back to the world coordinates of the driving scenario.

Create a driving scenario.

```
scenario = drivingScenario;
```

Create target actors.

```
actor(scenario,'ClassID',1, ...
    'Position',[10 20 30], ...
    'Velocity',[12 113 14], ...
    'Yaw',54, ...
    'Pitch',25, ...
    'Roll',22, ...
    'AngularVelocity',[24 42 27]);

actor(scenario,'ClassID',1, ...
    'Position',[17 22 12], ...
    'Velocity',[19 13 15], ...
    'Yaw',45, ...
    'Pitch',52, ...
    'Roll',2, ...
    'AngularVelocity',[42 24 29]);
```

Add an ego vehicle actor.

```
egoActor = actor(scenario,'ClassID',1, ...
    'Position',[1 2 3], ...
```

```
    'Velocity',[1.2 1.3 1.4], ...
    'Yaw',4, ...
    'Pitch',5, ...
    'Roll',2, ...
    'AngularVelocity',[4 2 7]);
```

Use the `actorPoses` function to return the poses of all actors in the scenario. Pose properties (position, velocity, and orientation) are in the world coordinates of the driving scenario. Save the target actors to a separate variable and inspect the pose of the first target actor.

```
allPoses = actorPoses(scenario);
targetPosesScenarioCoords = allPoses(1:2);
targetPosesScenarioCoords(1)
```

```
ans = struct with fields:
            ActorID: 1
           Position: [10 20 30]
           Velocity: [12 113 14]
               Roll: 22
              Pitch: 25
                Yaw: 54
    AngularVelocity: [24 42 27]
```

Use the `driving.scenario.targetsToEgo` function to convert the target poses to the ego-centric coordinates of the ego actor. Inspect the pose of the first actor.

```
targetPosesEgoCoords = driving.scenario.targetsToEgo(targetPosesScenarioCoords,egoActor);
targetPosesEgoCoords(1)
```

```
ans = struct with fields:
            ActorID: 1
           Position: [7.8415 18.2876 27.1675]
           Velocity: [18.6826 112.0403 9.2960]
               Roll: 16.4327
              Pitch: 23.2186
                Yaw: 47.8114
    AngularVelocity: [-3.3744 47.3021 18.2569]
```

Alternatively, use the `targetPoses` function to obtain all target actor poses in ego vehicle coordinates. Display the first target pose, which matches the previously calculated pose.

```
targetPosesEgoCoords = targetPoses(egoActor);
targetPosesEgoCoords(1)
```

```
ans = struct with fields:
            ActorID: 1
            ClassID: 1
           Position: [7.8415 18.2876 27.1675]
           Velocity: [18.6826 112.0403 9.2960]
               Roll: 16.4327
              Pitch: 23.2186
                Yaw: 47.8114
    AngularVelocity: [-3.3744 47.3021 18.2569]
```

Use the `driving.scenario.targetsToScenario` to convert the target poses back to the world coordinates of the scenario. Display the first target pose, which matches the original target pose.

```
targetPosesScenarioCoords = driving.scenario.targetsToScenario(targetPosesEgoCoords,egoActor);
targetPosesScenarioCoords(1)
```

```
ans = struct with fields:
            ActorID: 1
            ClassID: 1
           Position: [10.0000 20.0000 30.0000]
           Velocity: [12.0000 113.0000 14.0000]
               Roll: 22
              Pitch: 25.0000
                Yaw: 54
    AngularVelocity: [24.0000 42.0000 27.0000]
```

Obtain Target Poses Within Sensor Range

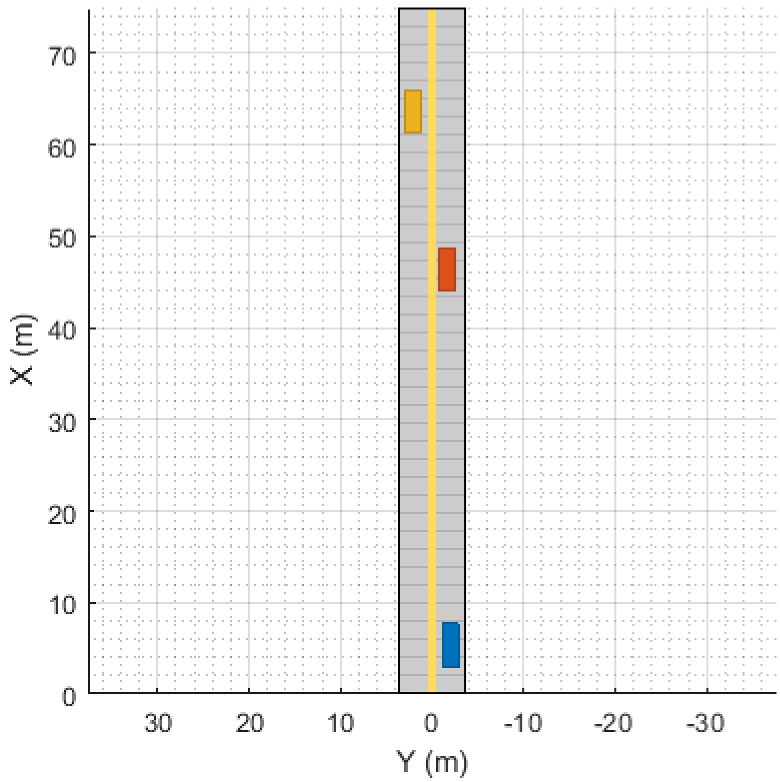
Obtain the poses of targets that are within the maximum range of a sensor mounted to the ego vehicle.

Create a driving scenario. The scenario contains a 75-meter straight road, an ego vehicle, and two target vehicles.

- The nearest target vehicle is 45 meters away and in the same lane as the ego vehicle.
- The farthest target vehicle is 65 meters away and in the opposite lane of the ego vehicle.

Plot the driving scenario.

```
scenario = drivingScenario;

roadCenters = [0 0 0; 75 0 0];
laneSpecification = lanespec([1 1]);
road(scenario,roadCenters,'Lanes',laneSpecification,'Name','Road');

egoVehicle = vehicle(scenario, ...
    'ClassID',1, ...
    'Position',[4 -2 0], ...
    'Name','Ego');

vehicle(scenario, ...
    'ClassID',1, ...
    'Position',[45 -1.7 0], ...
    'Name','Near Target');

vehicle(scenario, ...
    'ClassID',1, ...
    'Position',[65 2 0], ...
    'Yaw',-180, ...
    'Name','Far Target');

plot(scenario)
```

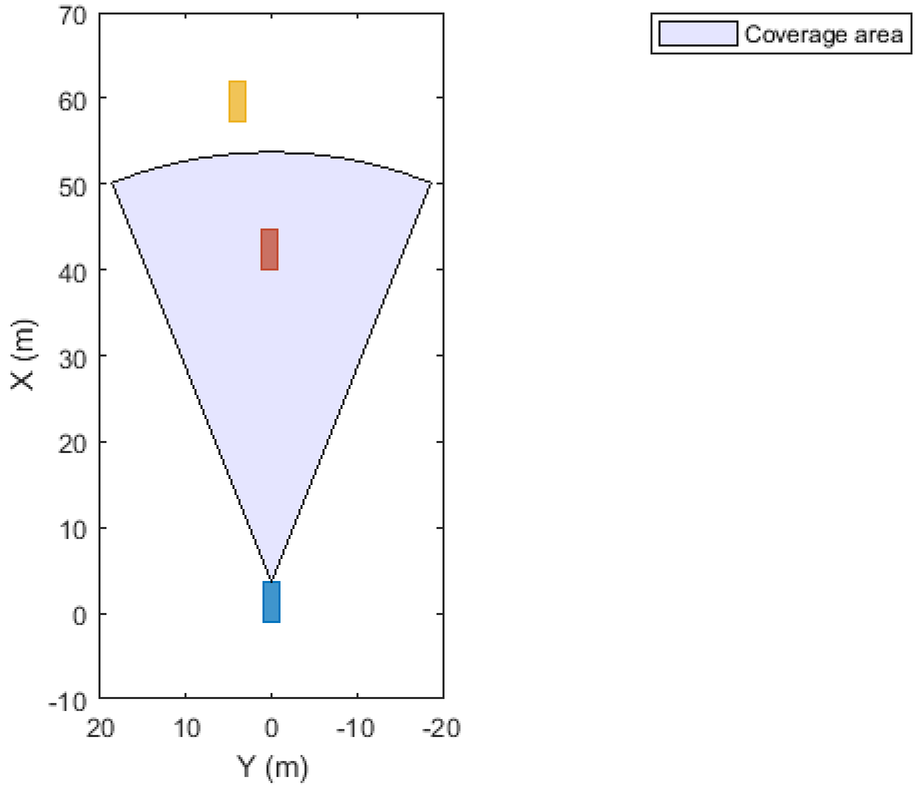


Create a vision sensor mounted to the front bumper of the ego vehicle. Configure the sensor to have a maximum detection range of 50 meters.

```
sensor = visionDetectionGenerator('SensorIndex',1, ...
    'SensorLocation',[3.7 0], ...
    'MaxRange',50);
```

Plot the outlines of the vehicles and the coverage area of the sensor. The nearest target vehicle is within range of the sensor but the farthest target vehicle is not.

```
bep = birdsEyePlot;

olPlotter = outlinePlotter(bep);
[position,yaw,length,width,originOffset,color] = targetOutlines(egoVehicle);
plotOutline(olPlotter,position,yaw,length,width, ...
    'OriginOffset',originOffset,'Color',color)

caPlotter = coverageAreaPlotter(bep,'DisplayName','Coverage area','FaceColor','blue');
mountPosition = sensor.SensorLocation;
range = sensor.MaxRange;
orientation = sensor.Yaw;
fieldOfView = sensor.FieldOfView(1);
plotCoverageArea(caPlotter,mountPosition,range,orientation,fieldOfView);
```



Obtain the poses of targets that are within the range of the sensor. The output structure contains the pose of only the nearest target, which is under 50 meters away from the ego vehicle.

```
poses = targetPoses(egoVehicle,range)
```

```
poses = struct with fields:
            ActorID: 2
            ClassID: 1
           Position: [41 0.3000 0]
           Velocity: [0 0 0]
               Roll: 0
              Pitch: 0
                Yaw: 0
    AngularVelocity: [0 0 0]
```

Input Arguments

ac — Actor

`Actor` object | `Vehicle` object

Actor belonging to a `drivingScenario` object, specified as an `Actor` or `Vehicle` object. To create these objects, use the `actor` and `vehicle` functions, respectively.

range — Circular range around ego vehicle

nonnegative real scalar

Circular range around the ego vehicle actor, specified as a nonnegative real scalar. The `targetPoses` function returns only the poses of targets that lie within this range. Units are in meters.

Output Arguments

poses — Target poses
structure | array of structures

Target poses, in ego vehicle coordinates, returned as a structure or as an array of structures. The pose of the ego vehicle actor, `ac`, is not included.

A target pose defines the position, velocity, and orientation of a target in ego vehicle coordinates. Target poses also include the rates of change in actor position and orientation.

Each pose structure has these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`ClassID`	Classification identifier, specified as a nonnegative integer. `0` represents an object of an unknown or unassigned class.
`Position`	Position of actor, specified as a real-valued vector of the form [*x* *y* *z*]. Units are in meters.
`Velocity`	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x v_y v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (*ω*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x ω_y ω_z]. Units are in degrees per second.

For full definitions of these structure fields, see the `actor` and `vehicle` functions.

More About

Ego Vehicle and Targets

In a driving scenario, you can specify one actor as the observer of all other actors, similar to how the driver of a car observes all other cars. The observer actor is called the ego actor or, more specifically, the ego vehicle. From the perspective of the ego vehicle, all other actors (such as vehicles and pedestrians) are the observed actors, called targets. Ego vehicle coordinates are centered and oriented with reference to the ego vehicle. The coordinates of the driving scenario are world coordinates.

See Also

Objects
`drivingScenario`

Functions
`actor` | `actorPoses` | `actorProfiles` | `vehicle`

Topics
"Create Driving Scenario Programmatically"

Introduced in R2017a

targetOutlines

Package:

Outlines of targets viewed by actor

Syntax

```
[position,yaw,length,width,originOffset,color] = targetOutlines(ac)
```

Description

`[position,yaw,length,width,originOffset,color] = targetOutlines(ac)` returns the oriented rectangular outlines of all non-ego target actors in a driving scenario. The outlines are as viewed from a designated ego vehicle actor, `ac`. See "Ego Vehicle and Targets" on page 4-401 for more details.

A target outline is the projection of the target actor cuboid into the (x,y) plane of the local coordinate system of the ego vehicle. The target outline components are the `position`, `yaw`, `length`, `width`, `originOffset`, and `color` output arguments.

You can use the returned outlines as input arguments to the outline plotter of a `birdsEyePlot`. First, call the `outlinePlotter` function to create the plotter object. Then, use the `plotOutline` function to plot the outlines of all the actors in a bird's-eye plot.

Examples

Show Target Outlines in Driving Scenario Simulation

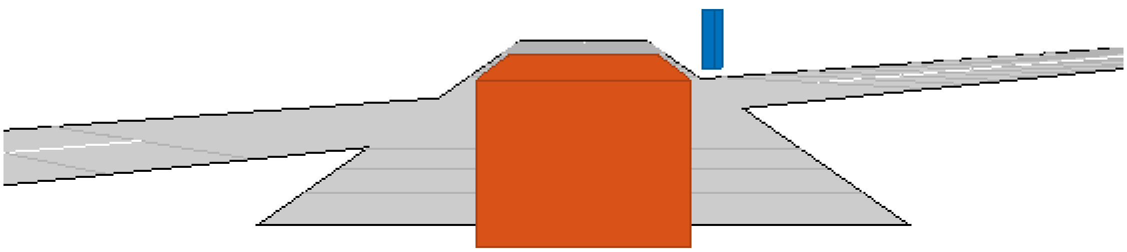
Create a driving scenario and show how target outlines change as the simulation advances.

Create a driving scenario consisting of two intersecting straight roads. The first road segment is 45 meters long. The second straight road is 32 meters long and intersects the first road. A car traveling at 12.0 meters per second along the first road approaches a running pedestrian crossing the intersection at 2.0 meters per second.

```
scenario = drivingScenario('SampleTime',0.1,'StopTime',1);
road(scenario,[-10 0 0; 45 -20 0]);
road(scenario,[-10 -10 0; 35 10 0]);
ped = actor(scenario,'ClassID',4,'Length',0.4,'Width',0.6,'Height',1.7);
car = vehicle(scenario,'ClassID',1);
pedspeed = 2.0;
carspeed = 12.0;
trajectory(ped,[15 -3 0; 15 3 0],pedspeed);
trajectory(car,[-10 -10 0; 35 10 0],carspeed);
```

Create an ego-centric chase plot for the vehicle.

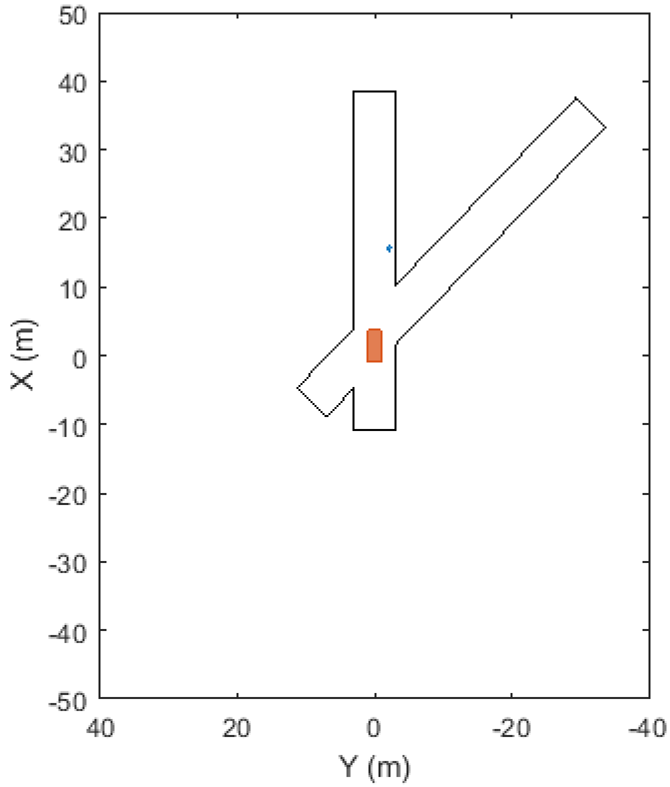
```
chasePlot(car,'Centerline','on')
```



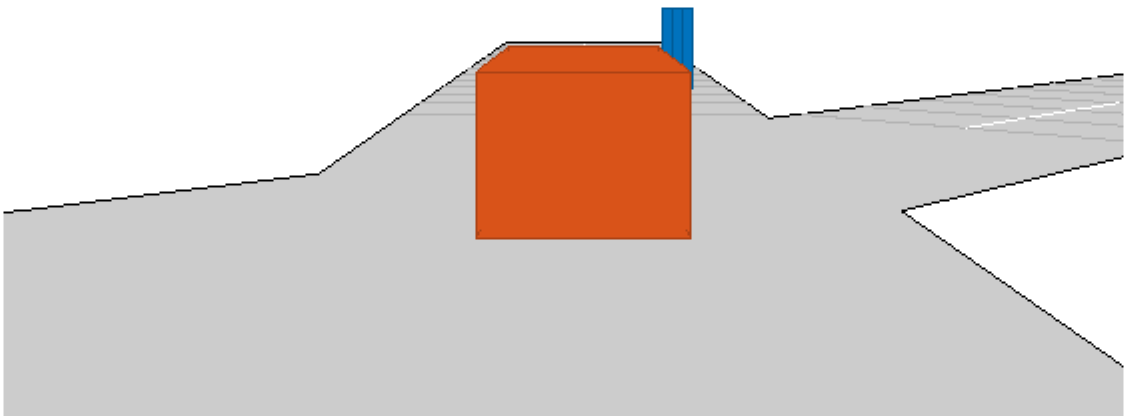
Create an empty bird's-eye plot and add an outline plotter and lane boundary plotter. Then, run the simulation. At each simulation step:

- Update the chase plot to display the road boundaries and target outlines.
- Update the bird's-eye plot to display the updated road boundaries and target outlines. The plot perspective is always with respect to the ego vehicle.

```
bepPlot = birdsEyePlot('XLim',[-50 50],'YLim',[-40 40]);
outlineplotter = outlinePlotter(bepPlot);
laneplotter = laneBoundaryPlotter(bepPlot);
legend('off')

while advance(scenario)
    rb = roadBoundaries(car);
    [position,yaw,length,width,originOffset,color] = targetOutlines(car);
    plotLaneBoundary(laneplotter,rb)
    plotOutline(outlineplotter,position,yaw,length,width, ...
        'OriginOffset',originOffset,'Color',color)
    pause(0.01)
end
```



50
40
30
20
10
0
-10
-20
-30
-40
-50
X (m)
40
20
0
-20
-40
Y (m)



Input Arguments

ac — Actor
`Actor` object | `Vehicle` object

Actor belonging to a `drivingScenario` object, specified as an `Actor` or `Vehicle` object. To create these objects, use the `actor` and `vehicle` functions, respectively.

Output Arguments

`position` — Rotational centers of targets
real-valued *N*-by-2 matrix

Rotational centers of targets, returned as a real-valued *N*-by-2 matrix. *N* is the number of targets. Each row contains the *x*- and *y*-coordinates of the rotational center of a target. Units are in meters.

`yaw` — Yaw angles of targets
real-valued *N*-element vector

Yaw angles of targets about the rotational center, returned as a real-valued *N*-element vector. *N* is the number of targets. Yaw angles are measured in the counterclockwise direction, as seen from above. Units are in degrees.

`length` — Lengths of rectangular outlines of targets
positive, real-valued *N*-element vector

Lengths of rectangular outlines of targets, returned as a positive, real-valued *N*-element vector. *N* is the number of targets. Units are in meters.

`width` — Widths of rectangular outlines of targets
positive, real-valued *N*-element vector

Widths of rectangular outline of targets, returned as a positive, real-valued *N*-element vector. *N* is the number of targets. Units are in meters.

`originOffset` — Offsets of rotational centers from geometric centers
real-valued *N*-by-2 matrix

Offset of the rotational centers of targets from their geometric centers, returned as a real-valued *N*-by-2 matrix. *N* is the number of targets. Each row contains the *x*- and *y*-coordinates defining this offset. In vehicle targets, the rotational center, or origin, is located on the ground, directly beneath the center of the rear axle. Units are in meters.

`color` — RGB representation of target colors
nonnegative, real-valued *N*-by-3 matrix

RGB representation of target colors, returned as a nonnegative, real-valued *N*-by-3 matrix. *N* is the number of target actors.

More About

Ego Vehicle and Targets

In a driving scenario, you can specify one actor as the observer of all other actors, similar to how the driver of a car observes all other cars. The observer actor is called the ego actor or, more specifically, the ego vehicle. From the perspective of the ego vehicle, all other actors (such as vehicles and pedestrians) are the observed actors, called targets. Ego vehicle coordinates are centered and oriented with reference to the ego vehicle. The coordinates of the driving scenario are world coordinates.

See Also

Objects
`birdsEyePlot` | `drivingScenario`

Functions
`actor` | `actorPoses` | `outlinePlotter` | `plotOutline` | `targetPoses` | `vehicle`

Topics
"Create Driving Scenario Programmatically"

Introduced in R2017a

driving.scenario.targetsToEgo

Convert target poses from scenario to ego coordinates

Syntax

```
targetPosesEgoCoords = driving.scenario.targetsToEgo(
targetPosesScenarioCoords,egoPose)
targetPosesEgoCoords = driving.scenario.targetsToEgo(
targetPosesScenarioCoords,egoActor)
```

Description

`targetPosesEgoCoords = driving.scenario.targetsToEgo(targetPosesScenarioCoords,egoPose)` converts the poses of target actors from the world coordinates of a driving scenario to the coordinate system relative to the pose of an ego actor. A pose is the position, velocity, and orientation of an actor. For more details on the coordinate systems of actors, see "Ego Vehicle and Targets" on page 4-406.

`targetPosesEgoCoords = driving.scenario.targetsToEgo(targetPosesScenarioCoords,egoActor)` converts target poses by using the pose of the specified ego actor.

Examples

Convert Target Poses Between Ego Vehicle and Scenario Coordinates

In a simple driving scenario, obtain the poses of target vehicles in the coordinate system of the ego vehicle. Then convert these poses back to the world coordinates of the driving scenario.

Create a driving scenario.

```
scenario = drivingScenario;
```

Create target actors.

```
actor(scenario,'ClassID',1, ...
    'Position',[10 20 30], ...
    'Velocity',[12 113 14], ...
    'Yaw',54, ...
    'Pitch',25, ...
    'Roll',22, ...
    'AngularVelocity',[24 42 27]);

actor(scenario,'ClassID',1, ...
    'Position',[17 22 12], ...
    'Velocity',[19 13 15], ...
    'Yaw',45, ...
    'Pitch',52, ...
    'Roll',2, ...
    'AngularVelocity',[42 24 29]);
```

Add an ego vehicle actor.

```
egoActor = actor(scenario,'ClassID',1, ...
    'Position',[1 2 3], ...
    'Velocity',[1.2 1.3 1.4], ...
    'Yaw',4, ...
    'Pitch',5, ...
    'Roll',2, ...
    'AngularVelocity',[4 2 7]);
```

Use the `actorPoses` function to return the poses of all actors in the scenario. Pose properties (position, velocity, and orientation) are in the world coordinates of the driving scenario. Save the target actors to a separate variable and inspect the pose of the first target actor.

```
allPoses = actorPoses(scenario);
targetPosesScenarioCoords = allPoses(1:2);
targetPosesScenarioCoords(1)
```

```
ans = struct with fields:
            ActorID: 1
           Position: [10 20 30]
           Velocity: [12 113 14]
               Roll: 22
              Pitch: 25
                Yaw: 54
    AngularVelocity: [24 42 27]
```

Use the `driving.scenario.targetsToEgo` function to convert the target poses to the ego-centric coordinates of the ego actor. Inspect the pose of the first actor.

```
targetPosesEgoCoords = driving.scenario.targetsToEgo(targetPosesScenarioCoords,egoActor);
targetPosesEgoCoords(1)
```

```
ans = struct with fields:
            ActorID: 1
           Position: [7.8415 18.2876 27.1675]
           Velocity: [18.6826 112.0403 9.2960]
               Roll: 16.4327
              Pitch: 23.2186
                Yaw: 47.8114
    AngularVelocity: [-3.3744 47.3021 18.2569]
```

Alternatively, use the `targetPoses` function to obtain all target actor poses in ego vehicle coordinates. Display the first target pose, which matches the previously calculated pose.

```
targetPosesEgoCoords = targetPoses(egoActor);
targetPosesEgoCoords(1)
```

```
ans = struct with fields:
            ActorID: 1
            ClassID: 1
           Position: [7.8415 18.2876 27.1675]
           Velocity: [18.6826 112.0403 9.2960]
               Roll: 16.4327
              Pitch: 23.2186
                Yaw: 47.8114
```

```
    AngularVelocity: [-3.3744 47.3021 18.2569]
```

Use the `driving.scenario.targetsToScenario` to convert the target poses back to the world coordinates of the scenario. Display the first target pose, which matches the original target pose.

```
targetPosesScenarioCoords = driving.scenario.targetsToScenario(targetPosesEgoCoords,egoActor);
targetPosesScenarioCoords(1)
```

```
ans = struct with fields:
            ActorID: 1
            ClassID: 1
           Position: [10.0000 20.0000 30.0000]
           Velocity: [12.0000 113.0000 14.0000]
               Roll: 22
              Pitch: 25.0000
                Yaw: 54
    AngularVelocity: [24.0000 42.0000 27.0000]
```

Input Arguments

`targetPosesScenarioCoords` — Target poses in world coordinates of scenario
structure | array of structures

Target poses in the world coordinates of a driving scenario, specified as a structure or an array of structures.

Each target pose structure must contain at least these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
`Velocity`	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (*ω*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

For full definitions of these structure fields, see the `actor` and `vehicle` functions.

egoPose — Ego actor pose
structure

Ego actor pose in the world coordinates of a driving scenario, specified as a structure.

The ego actor pose structure must contain at least these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
`Velocity`	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (*ω*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

For full definitions of these structure fields, see the `actor` and `vehicle` functions.

`egoActor` — Ego actor
`Actor` object | `Vehicle` object

Ego actor in the world coordinates of a driving scenario, specified as an `Actor` or `Vehicle` object. The function converts the coordinates of target actors relative to the pose of `egoActor`. The pose information is stored in the `Position`, `Velocity`, `Roll`, `Pitch`, `Yaw`, and `AngularVelocity` properties of the ego actor.

For the full definitions of pose properties, see the `actor` and `vehicle` functions.

Output Arguments

`targetPosesEgoCoords` — Target poses in ego actor coordinates
structure | array of structures

Target poses in ego vehicle coordinates, returned as a structure or an array of structures.

At a minimum, each returned target pose structure has these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.

Field	Description
`Position`	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
`Velocity`	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (*ω*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

The returned `targetPosesEgoCoords` structures include the same fields as those in the input `targetPosesScenarioCoords` structures. For example, if the input structures include a `ClassID` field, then the returned structures also include `ClassID`.

For full definitions of these structure fields, see the `actor` and `vehicle` functions.

More About

Ego Vehicle and Targets

In a driving scenario, you can specify one actor as the observer of all other actors, similar to how the driver of a car observes all other cars. The observer actor is called the ego actor or, more specifically, the ego vehicle. From the perspective of the ego vehicle, all other actors (such as vehicles and pedestrians) are the observed actors, called targets. Ego vehicle coordinates are centered and oriented with reference to the ego vehicle. The coordinates of the driving scenario are world coordinates.

See Also

Objects
`drivingScenario`

Functions
`actor` | `actorPoses` | `driving.scenario.roadBoundariesToEgo` | `driving.scenario.targetsToScenario` | `road` | `roadBoundaries` | `targetPoses` | `vehicle`

Introduced in R2017a

driving.scenario.targetsToScenario

Convert target poses from ego to scenario coordinates

Syntax

```
targetPosesScenarioCoords = driving.scenario.targetsToScenario(
targetPosesEgoCoords,egoPose)
targetPosesScenarioCoords = driving.scenario.targetsToScenario(
targetPosesEgoCoords,egoActor)
```

Description

`targetPosesScenarioCoords = driving.scenario.targetsToScenario( targetPosesEgoCoords,egoPose)` converts the poses of target actors from coordinates relative to the pose of an ego actor to the world coordinates of a driving scenario. A pose is the position, velocity, and orientation of an actor. For more details on the coordinate systems of actors, see “Ego Vehicle and Targets” on page 4-411.

`targetPosesScenarioCoords = driving.scenario.targetsToScenario( targetPosesEgoCoords,egoActor)` converts target poses by using the pose of the specified ego actor.

Examples

Convert Target Poses Between Ego Vehicle and Scenario Coordinates

In a simple driving scenario, obtain the poses of target vehicles in the coordinate system of the ego vehicle. Then convert these poses back to the world coordinates of the driving scenario.

Create a driving scenario.

```
scenario = drivingScenario;
```

Create target actors.

```
actor(scenario,'ClassID',1, ...
    'Position',[10 20 30], ...
    'Velocity',[12 113 14], ...
    'Yaw',54, ...
    'Pitch',25, ...
    'Roll',22, ...
    'AngularVelocity',[24 42 27]);

actor(scenario,'ClassID',1, ...
    'Position',[17 22 12], ...
    'Velocity',[19 13 15], ...
    'Yaw',45, ...
    'Pitch',52, ...
    'Roll',2, ...
    'AngularVelocity',[42 24 29]);
```

Add an ego vehicle actor.

```
egoActor = actor(scenario,'ClassID',1, ...
    'Position',[1 2 3], ...
    'Velocity',[1.2 1.3 1.4], ...
    'Yaw',4, ...
    'Pitch',5, ...
    'Roll',2, ...
    'AngularVelocity',[4 2 7]);
```

Use the `actorPoses` function to return the poses of all actors in the scenario. Pose properties (position, velocity, and orientation) are in the world coordinates of the driving scenario. Save the target actors to a separate variable and inspect the pose of the first target actor.

```
allPoses = actorPoses(scenario);
targetPosesScenarioCoords = allPoses(1:2);
targetPosesScenarioCoords(1)
```

```
ans = struct with fields:
            ActorID: 1
           Position: [10 20 30]
           Velocity: [12 113 14]
               Roll: 22
              Pitch: 25
                Yaw: 54
    AngularVelocity: [24 42 27]
```

Use the `driving.scenario.targetsToEgo` function to convert the target poses to the ego-centric coordinates of the ego actor. Inspect the pose of the first actor.

```
targetPosesEgoCoords = driving.scenario.targetsToEgo(targetPosesScenarioCoords,egoActor);
targetPosesEgoCoords(1)
```

```
ans = struct with fields:
            ActorID: 1
           Position: [7.8415 18.2876 27.1675]
           Velocity: [18.6826 112.0403 9.2960]
               Roll: 16.4327
              Pitch: 23.2186
                Yaw: 47.8114
    AngularVelocity: [-3.3744 47.3021 18.2569]
```

Alternatively, use the `targetPoses` function to obtain all target actor poses in ego vehicle coordinates. Display the first target pose, which matches the previously calculated pose.

```
targetPosesEgoCoords = targetPoses(egoActor);
targetPosesEgoCoords(1)
```

```
ans = struct with fields:
            ActorID: 1
            ClassID: 1
           Position: [7.8415 18.2876 27.1675]
           Velocity: [18.6826 112.0403 9.2960]
               Roll: 16.4327
              Pitch: 23.2186
                Yaw: 47.8114
```

```
    AngularVelocity: [-3.3744 47.3021 18.2569]
```

Use the `driving.scenario.targetsToScenario` to convert the target poses back to the world coordinates of the scenario. Display the first target pose, which matches the original target pose.

```
targetPosesScenarioCoords = driving.scenario.targetsToScenario(targetPosesEgoCoords,egoActor);
targetPosesScenarioCoords(1)
```

```
ans = struct with fields:
            ActorID: 1
            ClassID: 1
           Position: [10.0000 20.0000 30.0000]
           Velocity: [12.0000 113.0000 14.0000]
               Roll: 22
              Pitch: 25.0000
                Yaw: 54
    AngularVelocity: [24.0000 42.0000 27.0000]
```

Input Arguments

`targetPosesEgoCoords` — Target poses in ego actor coordinates
structure | array of structures

Target poses in ego actor coordinates, specified as a structure or an array of structures.

Each target pose structure must contain at least these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier, specified as a positive integer.
`Position`	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
`Velocity`	Velocity (v) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (ω) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

For full definitions of these structure fields, see the `actor` and `vehicle` functions.

egoPose — Ego actor pose
structure

Ego actor pose in the world coordinates of a driving scenario, specified as a structure.

The ego actor pose structure must contain at least these fields.

Field	Description
ActorID	Scenario-defined actor identifier, specified as a positive integer.
Position	Position of actor, specified as a real-valued vector of the form [x, y, z]. Units are in meters.
Velocity	Velocity (v) of actor in the x-, y-, and z-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
Roll	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
Pitch	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
Yaw	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
AngularVelocity	Angular velocity (ω) of actor in the x-, y-, and z-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

For full definitions of these structure fields, see the `actor` and `vehicle` functions.

`egoActor` — Ego actor
`Actor` object | `Vehicle` object

Ego actor in the world coordinates of a driving scenario, specified as an `Actor` or `Vehicle` object. The function converts the coordinates of target actors relative to the pose of `egoActor`. The pose information is stored in the `Position`, `Velocity`, `Roll`, `Pitch`, `Yaw`, and `AngularVelocity` properties of the ego actor.

For the full definitions of pose properties, see the `actor` and `vehicle` functions.

Output Arguments

`targetPosesScenarioCoords` — Target poses in world coordinates of scenario
structure | array of structures

Target actor poses in the world coordinates of a driving scenario, returned as a structure or an array of structures.

At a minimum, each returned target pose structure has these fields.

Field	Description
ActorID	Scenario-defined actor identifier, specified as a positive integer.
Position	Position of actor, specified as a real-valued vector of the form [x, y, z]. Units are in meters.

Field	Description
`Velocity`	Velocity (v) of actor in the x-, y-, and z-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
`AngularVelocity`	Angular velocity (ω) of actor in the x-, y-, and z-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

The returned `targetPosesScenarioCoords` structures include the same fields as those in the input `targetPosesEgoCoords` structures. For example, if the input structures include a `ClassID` field, then the returned structures also include `ClassID`.

For full definitions of these structure fields, see the `actor` and `vehicle` functions.

More About

Ego Vehicle and Targets

In a driving scenario, you can specify one actor as the observer of all other actors, similar to how the driver of a car observes all other cars. The observer actor is called the ego actor or, more specifically, the ego vehicle. From the perspective of the ego vehicle, all other actors (such as vehicles and pedestrians) are the observed actors, called targets. Ego vehicle coordinates are centered and oriented with reference to the ego vehicle. The coordinates of the driving scenario are world coordinates.

See Also

Objects
`drivingScenario`

Functions
`actor` | `actorPoses` | `driving.scenario.roadBoundariesToEgo` | `driving.scenario.targetsToEgo` | `road` | `roadBoundaries` | `targetPoses` | `vehicle`

Introduced in R2020a

path

(To be removed) Create actor or vehicle path in driving scenario

Note `path` will be removed in a future release. Use `trajectory` instead.

Syntax

```
path(ac,waypoints)
path(ac,waypoints,speed)
```

Description

`path(ac,waypoints)` creates a path for an actor or vehicle, `ac`, using a set of waypoints. The actor follows the path at 30 m/s.

`path(ac,waypoints,speed)` also specifies the actor speed.

Input Arguments

ac — Actor

`Actor` object | `Vehicle` object

Actor belonging to a `drivingScenario` object, specified as an `Actor` or `Vehicle` object. To create these objects, use the `actor` and `vehicle` functions, respectively.

waypoints — Path waypoints

real-valued *N*-by-2 matrix | real-valued *N*-by-3 matrix

Path waypoints, specified as a real-valued *N*-by-2 or *N*-by-3 matrix, where *N* is the number of waypoints.

- If you specify the waypoints as an *N*-by-2 matrix, then each matrix row represents the (*x*,*y*) coordinates of a waypoint. The *z*-coordinate of each waypoint is zero.
- If you specify the waypoints as an *N*-by-3 matrix, then each matrix row represents the (*x*,*y*,*z*) coordinates of a waypoint.

All coordinates belong to the scenario coordinate system. Units are in meters.

Example: `[1 0 0; 2 7 7]`

speed — Actor speed

`30.0` | positive real scalar | *N*-element vector of nonnegative values

Actor speed, specified as a positive real scalar or *N*-element vector of nonnegative values. *N* is the number of waypoints.

- When `speed` is a scalar, the speed is constant throughout the actor motion.
- When `speed` is a vector, the vector values specify the speed at each waypoint.

Speeds are interpolated between waypoints. `speed` can be zero at any waypoint but cannot be zero at two consecutive waypoints. Units are in meters per second.

Example: `[10,8,10,11]`

Algorithms

The `path` function creates a path for an actor to follow in a scenario. You specify the path using *N* two-dimensional or three-dimensional waypoints. Each of the *N* – 1 segments between waypoints defines a curve whose curvature varies linearly with distance along the segment. The function fits a piecewise clothoid curve to the (*x*,*y*) coordinates of the waypoints by matching the curvature on both sides of the waypoint. For a nonclosed curve, the curvature at the first and last waypoint is zero. If the first and last waypoints coincide, then the curvatures before and after the endpoints are matched. The *z*-coordinates of the path are interpolated using a shape-preserving piecewise cubic curve.

You can specify speed as a scalar or a vector. When speed is a scalar, the actor follows the path with constant speed. When speed is an *N*-element vector, speed is linearly interpolated between waypoints. Setting the speed to zero at two consecutive waypoints creates a stationary actor.

Compatibility Considerations

`path` is not recommended
Not recommended starting in R2018a

`path` will be removed in a future release. Use `trajectory` instead.

Update Code

Replace all instances of `path` with `trajectory`. If you used `path` without specifying a speed, you must now specify one. The `trajectory` function does not include a syntax that assumes a default speed.

Discouraged Usage	Recommended Replacement

`scenario = drivingScenario;`
`road(scenario,[-10 0 0; 45 -20 0]);`
`car = vehicle(scenario);`
`waypoints = [-10 -10 0; 35 10 0];`
 	
`path(car,waypoints) % default speed = 30 m/s`
	`scenario = drivingScenario;`
	`road(scenario,[-10 0 0; 45 -20 0]);`
	`car = vehicle(scenario);`
	`waypoints = [-10 -10 0; 35 10 0];`
	
	`speed = 30;`
	`trajectory(car,waypoints,speed)`

See Also
`trajectory`

Introduced in R2017a

road

Add road to driving scenario

Syntax

```
road(scenario,roadcenters)
road(scenario,roadcenters,roadwidth)
road(scenario,roadcenters,roadwidth,bankingangle)

road(scenario,roadcenters,'Lanes',lspec)
road(scenario,roadcenters,bankingangle,'Lanes',lspec)

road( ___ ,'Name',name)
rd = road( ___ )
```

Description

`road(scenario,roadcenters)` adds a road to a driving scenario, `scenario`. You specify the road shape and the orientation of a road in the 2-D plane by using a set of road centers, `roadcenters`, at discrete points. When you specify the number of lanes on a road, the lanes are numbered with respect to the road centers. For more information, see “Draw Direction of Road and Numbering of Lanes” on page 4-427.

`road(scenario,roadcenters,roadwidth)` adds a road with the specified width, `roadwidth`.

`road(scenario,roadcenters,roadwidth,bankingangle)` adds a road with the specified width and banking angle, `bankingangle`.

`road(scenario,roadcenters,'Lanes',lspec)` adds a road with the specified lanes, `lspec`.

`road(scenario,roadcenters,bankingangle,'Lanes',lspec)` adds a road with the specified banking angle and lanes.

`road( ___ ,'Name',name)` specifies the name of the road, using any of the input argument combinations from previous syntaxes.

`rd = road( ___ )` returns a `Road` object that stores the properties of the created road.

Examples

Create Driving Scenario with Multiple Actors and Roads

Create a driving scenario containing a curved road, two straight roads, and two actors: a car and a bicycle. Both actors move along the road for 60 seconds.

Create the driving scenario object.

```
scenario = drivingScenario('SampleTime',0.1','StopTime',60);
```

Create the curved road using road center points following the arc of a circle with an 800-meter radius. The arc starts at 0°, ends at 90°, and is sampled at 5° increments.

```
angs = [0:5:90]';
R = 800;
roadcenters = R*[cosd(angs) sind(angs) zeros(size(angs))];
roadwidth = 10;
road(scenario,roadcenters,roadwidth);
```

Add two straight roads with the default width, using road center points at each end.

```
roadcenters = [700 0 0; 100 0 0];
road(scenario,roadcenters)
```

```
ans = 
  Road with properties:

           Name: ""
         RoadID: 2
    RoadCenters: [2x3 double]
      RoadWidth: 6
      BankAngle: [2x1 double]
```

```
roadcenters = [400 400 0; 0 0 0];
road(scenario,roadcenters)
```

```
ans = 
  Road with properties:

           Name: ""
         RoadID: 3
    RoadCenters: [2x3 double]
      RoadWidth: 6
      BankAngle: [2x1 double]
```

Get the road boundaries.

```
rbdry = roadBoundaries(scenario);
```

Add a car and a bicycle to the scenario. Position the car at the beginning of the first straight road.

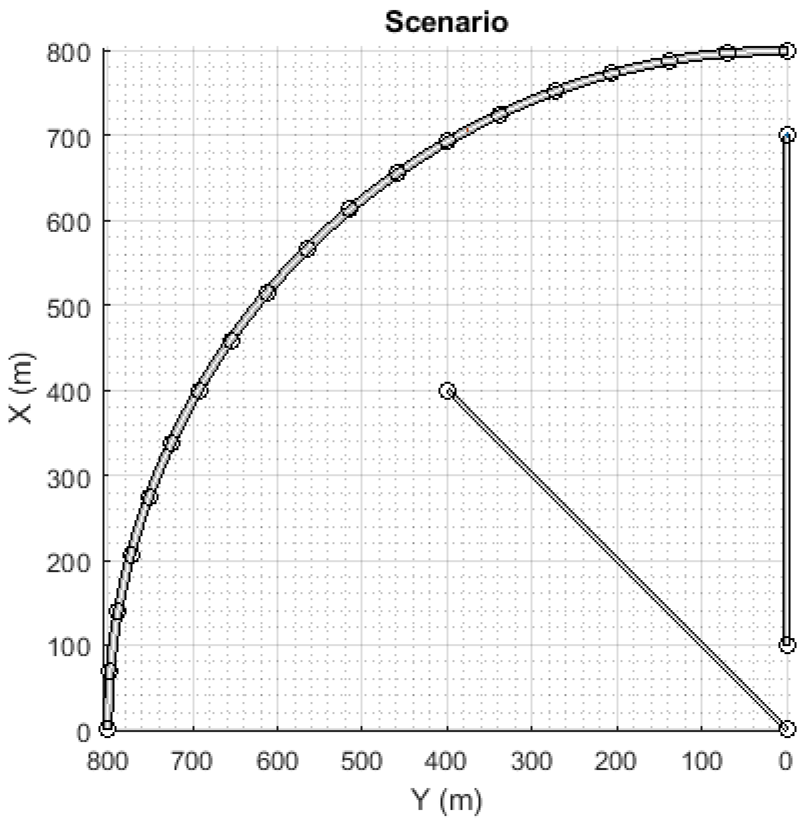
```
car = vehicle(scenario,'ClassID',1,'Position',[700 0 0], ...
    'Length',3,'Width',2,'Height',1.6);
```

Position the bicycle farther down the road.

```
bicycle = actor(scenario,'ClassID',3,'Position',[706 376 0]', ...
    'Length',2,'Width',0.45,'Height',1.5);
```

Plot the scenario.

```
plot(scenario,'Centerline','on','RoadCenters','on');
title('Scenario');
```



Display the actor poses and profiles.

```
poses = actorPoses(scenario)

poses=2×1 struct array with fields:
    ActorID
    Position
    Velocity
    Roll
    Pitch
    Yaw
    AngularVelocity


profiles = actorProfiles(scenario)

profiles=2×1 struct array with fields:
    ActorID
    ClassID
    Length
    Width
    Height
    OriginOffset
    MeshVertices
    MeshFaces
    RCSPattern
    RCSAzimuthAngles
```

```
RCSElevationAngles
```

Create and Display Road Boundaries

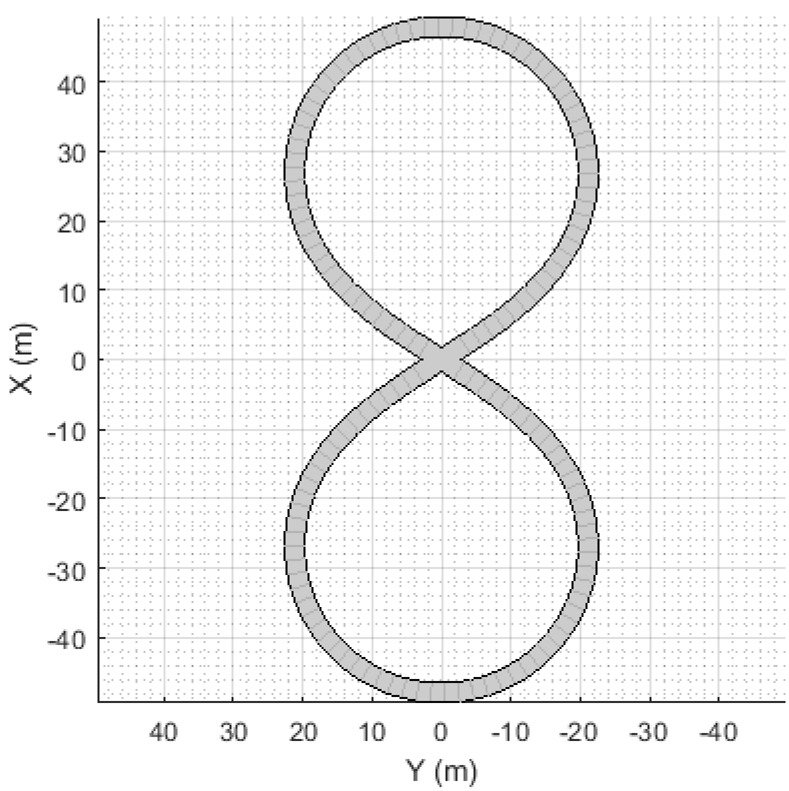
Create a driving scenario containing a figure-8 road specified in the world coordinates of the scenario. Convert the world coordinates of the scenario to the coordinate system of the ego vehicle.

Create an empty driving scenario.

```
scenario = drivingScenario;
```

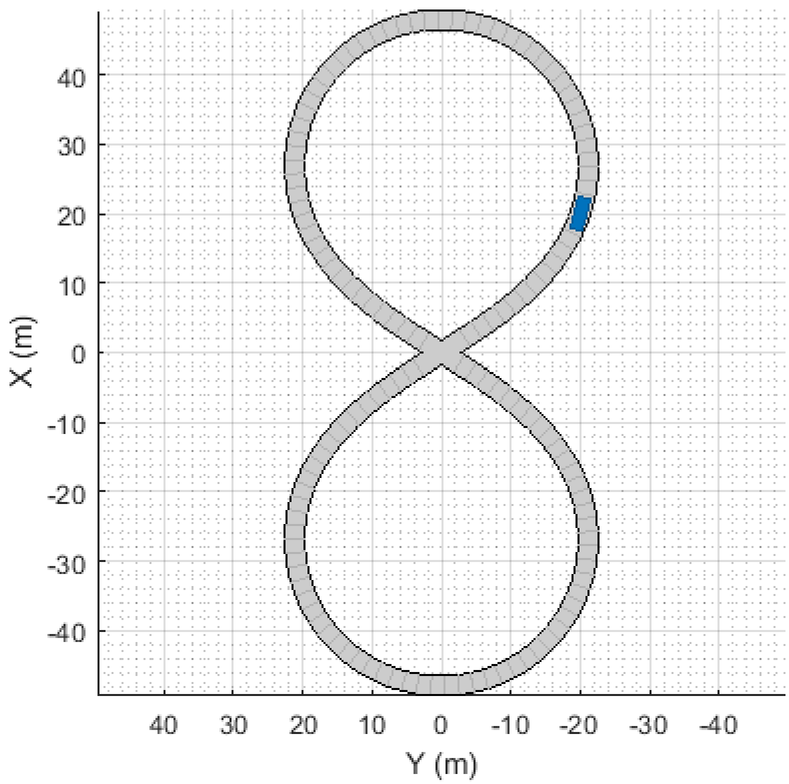
Add a figure-8 road to the scenario. Display the scenario.

```
roadCenters = [0  0  1
             20 -20  1
             20  20  1
            -20 -20  1
            -20  20  1
              0   0  1];

roadWidth = 3;
bankAngle = [0 15 15 -15 -15 0];
road(scenario,roadCenters,roadWidth,bankAngle);
plot(scenario)
```



Add an ego vehicle to the scenario. Position the vehicle at world coordinates (20, –20) and orient it at a –15 degree yaw angle.

```
ego = actor(scenario,'ClassID',1,'Position',[20 -20 0],'Yaw',-15);
```

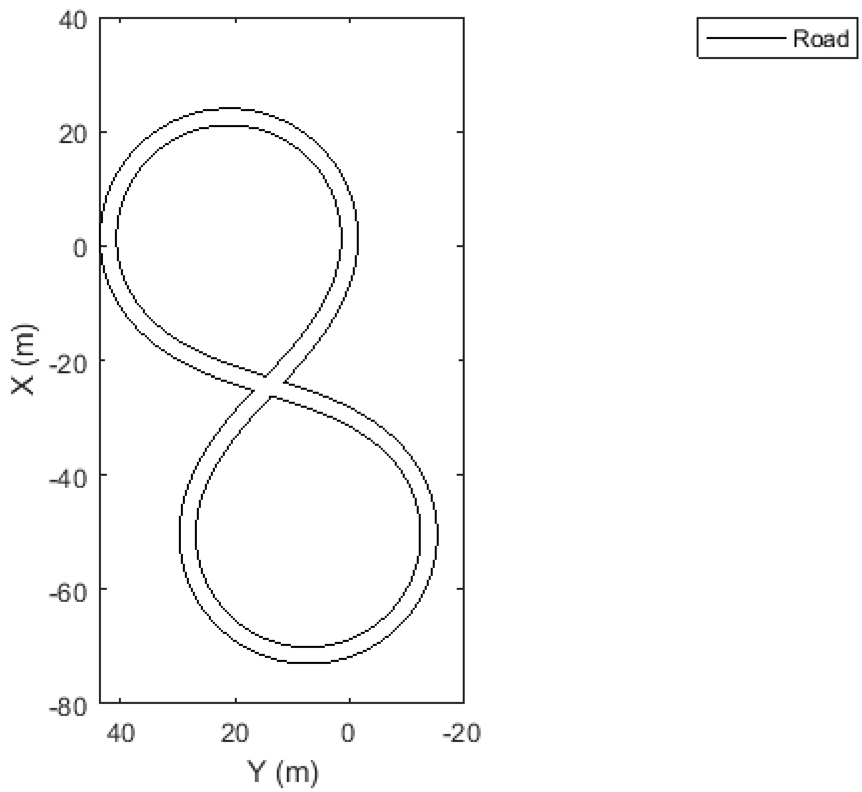


Obtain the road boundaries in ego vehicle coordinates by using the `roadBoundaries` function. Specify the ego vehicle as the input argument.

```
rbEgo1 = roadBoundaries(ego);
```

Display the result on a bird's-eye plot.

```
bep = birdsEyePlot;
lbp = laneBoundaryPlotter(bep,'DisplayName','Road');
plotLaneBoundary(lbp,rbEgo1)
```



Obtain the road boundaries in world coordinates by using the `roadBoundaries` function. Specify the scenario as the input argument.

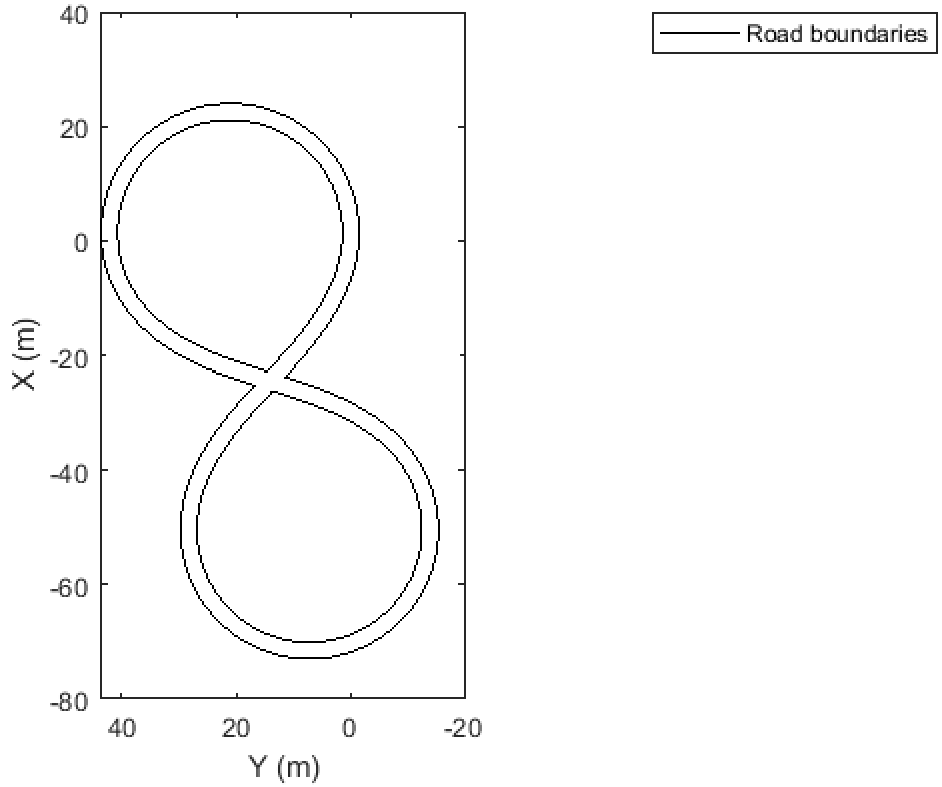
```
rbScenario = roadBoundaries(scenario);
```

Obtain the road boundaries in ego vehicle coordinates by using the `driving.scenario.roadBoundariesToEgo` function.

```
rbEgo2 = driving.scenario.roadBoundariesToEgo(rbScenario,ego);
```

Display the road boundaries on a bird's-eye plot.

```
bep = birdsEyePlot;
lbp = laneBoundaryPlotter(bep,'DisplayName','Road boundaries');
plotLaneBoundary(lbp,{rbEgo2})
```



Display Lane Markings in Car and Pedestrian Scenario

Create a driving scenario containing a car and pedestrian on a straight road. Then, create and display the lane markings of the road on a bird's-eye plot.

Create an empty driving scenario.

```
scenario = drivingScenario;
```

Create a straight, 25-meter road segment with two travel lanes in one direction.

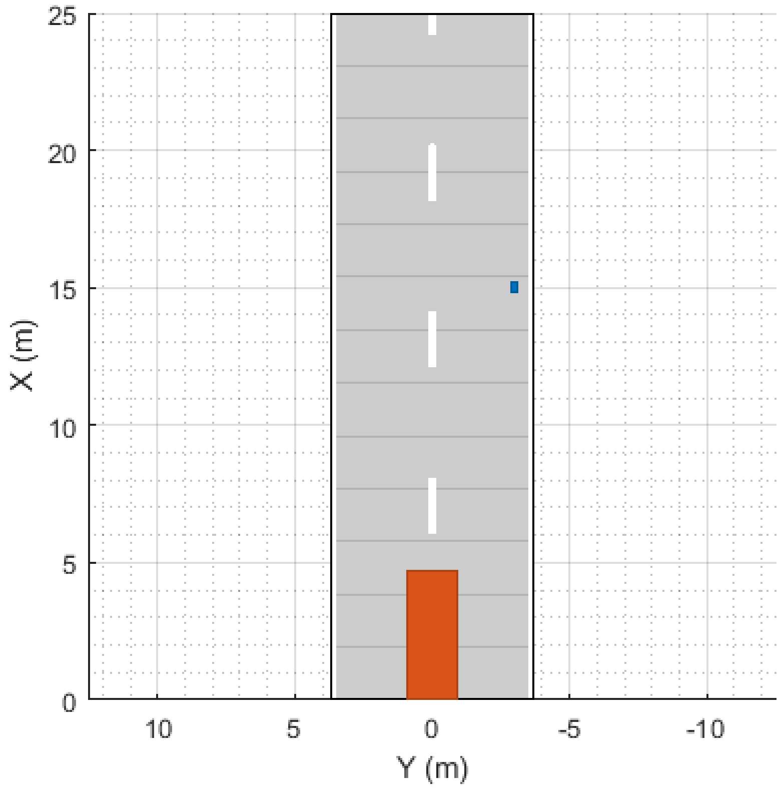
```
lm = [laneMarking('Solid')
      laneMarking('Dashed','Length',2,'Space',4)
      laneMarking('Solid')];
l = lanespec(2,'Marking',lm);
road(scenario,[0 0 0; 25 0 0],'Lanes',l);
```

Add to the driving scenario a pedestrian crossing the road at 1 meter per second and a car following the road at 10 meters per second.

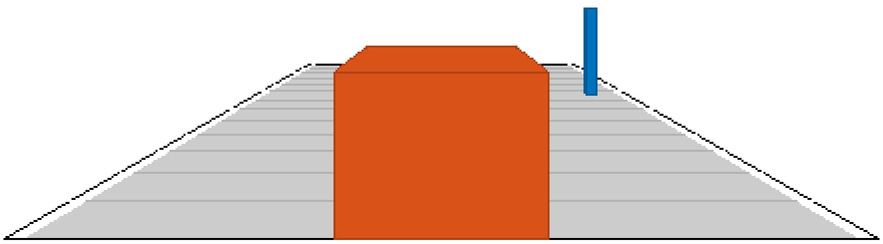
```
ped = actor(scenario,'ClassID',4,'Length',0.2,'Width',0.4,'Height',1.7);
car = vehicle(scenario,'ClassID',1);
trajectory(ped,[15 -3 0; 15 3 0],1);
trajectory(car,[car.RearOverhang 0 0; 25-car.Length+car.RearOverhang 0 0],10);
```

Display the scenario and corresponding chase plot.

```
plot(scenario)
```



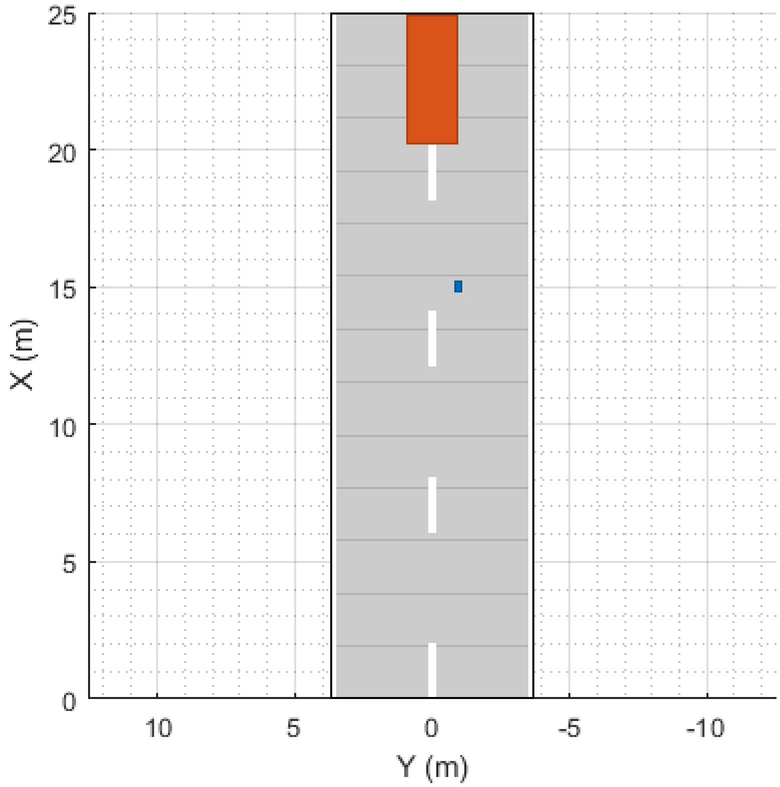

```
chasePlot(car)
```

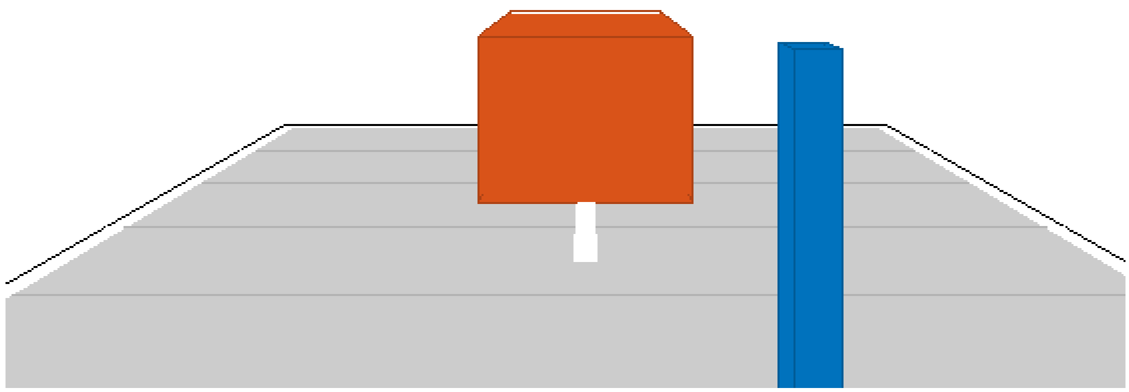


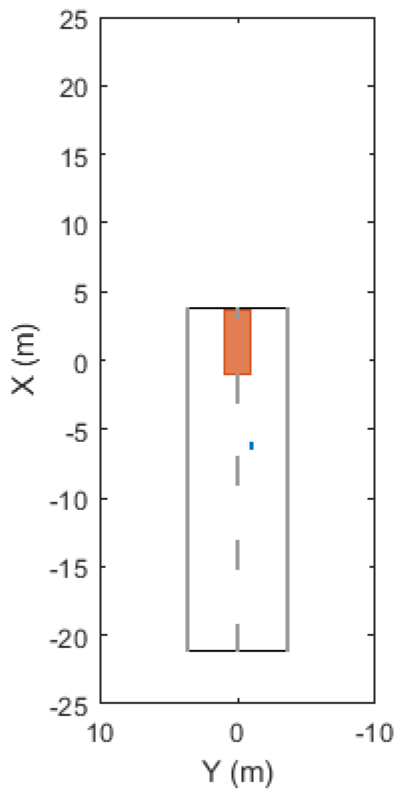
Run the simulation.

1 Create a bird's-eye plot.
2 Create an outline plotter, lane boundary plotter, and lane marking plotter for the bird's-eye plot.
3 Obtain the road boundaries and target outlines.
4 Obtain the lane marking vertices and faces.
5 Display the lane boundaries and lane markers.
6 Run the simulation loop.

```
bep = birdsEyePlot('XLim',[-25 25],'YLim',[-10 10]);
olPlotter = outlinePlotter(bep);
lbPlotter = laneBoundaryPlotter(bep);
lmPlotter = laneMarkingPlotter(bep,'DisplayName','Lanes');
legend('off');
while advance(scenario)
    rb = roadBoundaries(car);
    [position,yaw,length,width,originOffset,color] = targetOutlines(car);
    [lmv,lmf] = laneMarkingVertices(car);
    plotLaneBoundary(lbPlotter,rb);
    plotLaneMarking(lmPlotter,lmv,lmf);
    plotOutline(olPlotter,position,yaw,length,width, ...
        'OriginOffset',originOffset,'Color',color);
end
```


25
20
15
10
5
0
X (m)
10
5
0
-5
-10
Y (m)





Input Arguments

`scenario` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

`roadcenters` — Road centers used to define road
real-valued *N*-by-2 matrix | real-valued *N*-by-3 matrix

Road centers used to define a road, specified as a real-valued *N*-by-2 or *N*-by-3 matrix. Road centers determine the center line of the road at discrete points.

- If `roadcenters` is an *N*-by-2 matrix, then each matrix row represents the (*x*, *y*) coordinates of a road center. The *z*-coordinate of each road center is zero.
- If `roadcenters` is an *N*-by-3 matrix, then each matrix row represents the (*x*, *y*, *z*) coordinates of a road center.

If the first row of the matrix is the same as the last row, the road is a loop. Units are in meters.

Data Types: `double`

`roadwidth` — Width of road
`6.0` (default) | positive real scalar | `[]`

Width of road, specified as a positive real scalar. The width is constant along the entire road. Units are in meters.

To specify the `bankingangle` input but not `roadwidth`, specify `roadwidth` as an empty argument, `[]`.

If you specify `roadwidth`, then you cannot specify the `lspec` input.

Data Types: `double`

`bankingangle` — Banking angle of road
`0` (default) | real-valued *N*-by-1 vector

Banking angle of road, specified as a real-valued *N*-by-1 vector. *N* is the number of road centers. The banking angle is the roll angle of the road along the direction of the road. Units are in degrees.

`lspec` — Lane specification
`lanespec` object

Lane specification, specified as a `lanespec` object. Use `lanespec` to specify the number of lanes, the width of each lane, and the type of lane markings. To specify the lane markings within `lanespec`, use the `laneMarking` function.

If you specify `lspec`, then you cannot specify the `roadwidth` input.

Example: `'Lane',lanespec(3)` specifies a three-lane road with default lane widths and lane markings.

`name` — Name of road
`""` (default) | character vector | string scalar

Name of the road, specified as a character vector or string scalar.

Example: `'Name','Road1'`

Example: `"Name","Road1"`

Data Types: `char` | `string`

Output Arguments

`rd` — Output road
`Road` object

Output road, returned as a `Road` object that has the properties described in this table. With the exception of `RoadID`, which is a scenario-generated property, the property names correspond to the input arguments used to create the road. All properties are read-only.

Property	Value
`Name`	Name of road, specified as a string scalar The `name` input argument sets this property. Even if you specify `name` as a character vector, the `Name` property value is a string scalar.

Property	Value
RoadID	Identifier of road, specified as a positive integer The `scenario` input argument generates a unique ID for each road in the driving scenario.
RoadCenters	Road centers used to define a road, specified as a real-valued *N*-by-2 or *N*-by-3 matrix, where *N* is the number of road centers The `roadcenters` input argument sets this property.
RoadWidth	Width of road, specified as a positive real scalar The `roadwidth` input argument sets this property.
BankAngle	Banking angle of road, specified as an *N*-by-1 real-valued vector, where *N* is the number of road centers in the road The `bankingangle` input argument sets this property.

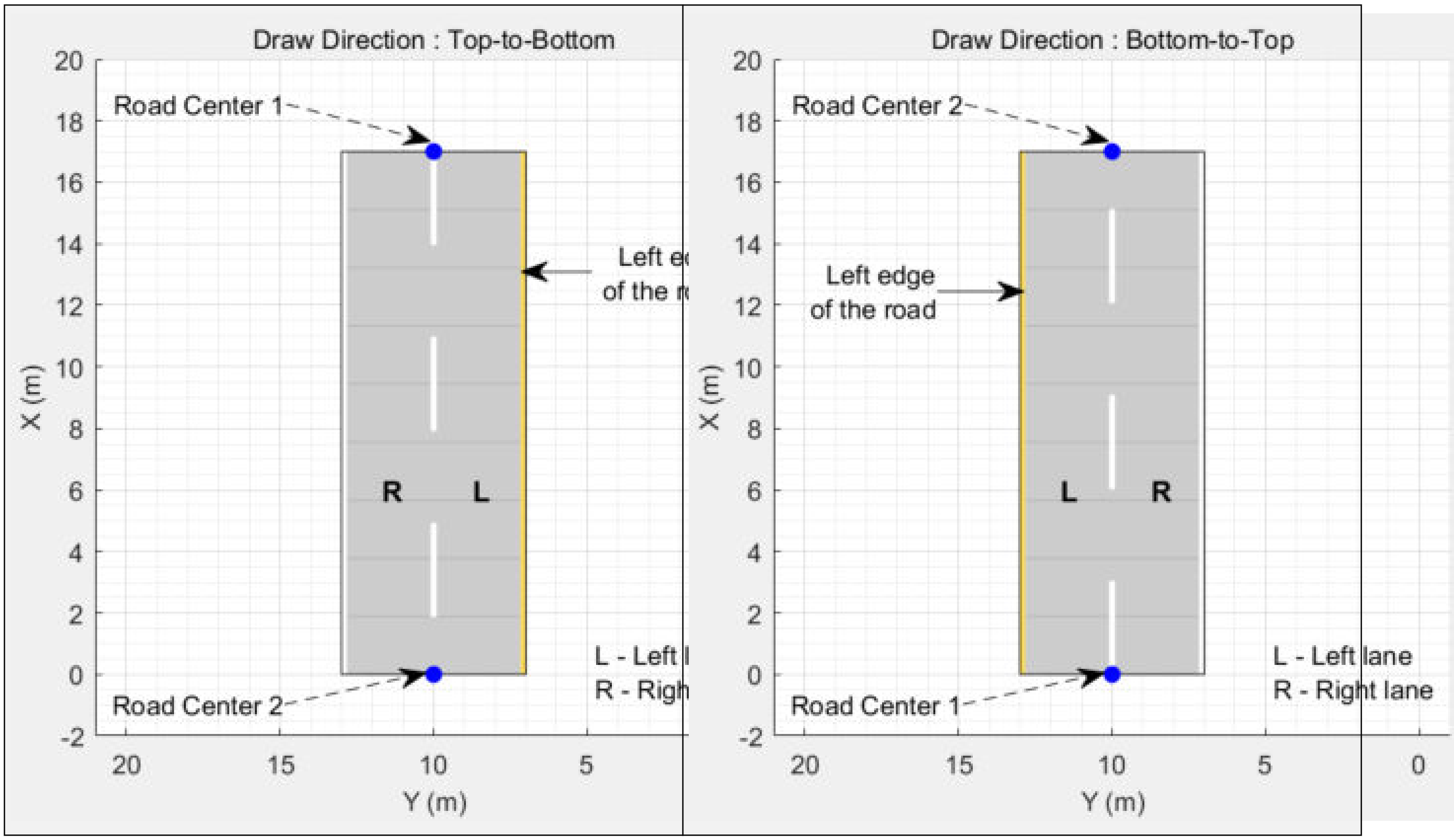
More About

Draw Direction of Road and Numbering of Lanes

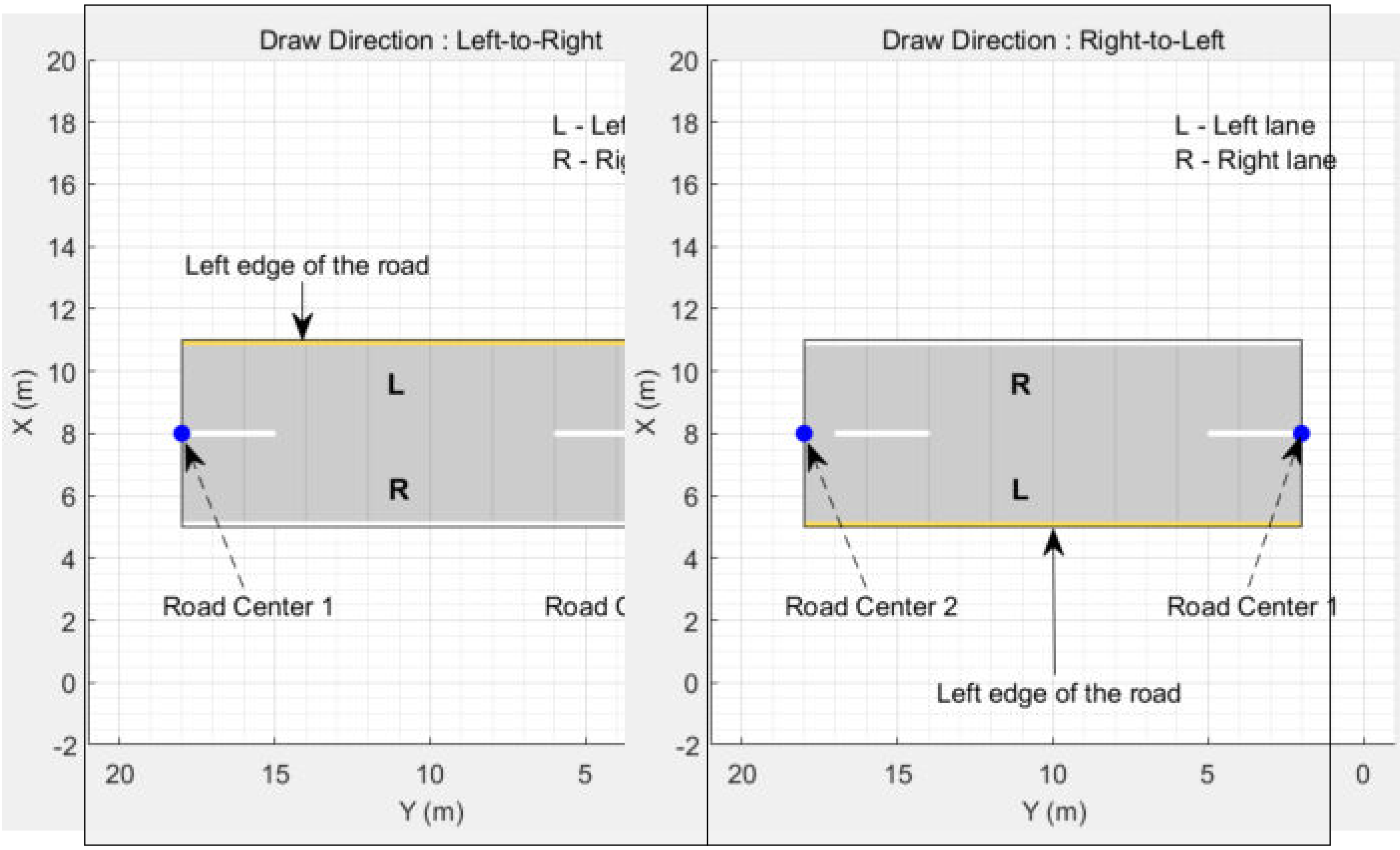
To create a road by using the `road` function, specify the road centers as a matrix input. The function creates a directed line that traverses the road centers, starting from the coordinates in the first row of the matrix and ending at the coordinates in the last row of the matrix. The coordinates in the first two rows of the matrix specify the draw direction of the road. These coordinates correspond to the first two consecutive road centers. The draw direction is the direction in which the roads render in the scenario plot.

To create a road by using the **Driving Scenario Designer** app, you can either specify the **Road Centers** parameter or interactively draw on the **Scenario Canvas**. For a detailed example, see “Create a Driving Scenario” on page 1-14. In this case, the draw direction is the direction in which roads render in the **Scenario Canvas**.

- For a road with a top-to-bottom draw direction, the difference between the *x*-coordinates of the first two consecutive road centers is positive.
- For a road with a bottom-to-top draw direction, the difference between the *x*-coordinates of the first two consecutive road centers is negative.



- For a road with a left-to-right draw direction, the difference between the *y*-coordinates of the first two consecutive road centers is positive.
- For a road with a right-to-left draw direction, the difference between the *y*-coordinates of the first two consecutive road centers is negative.

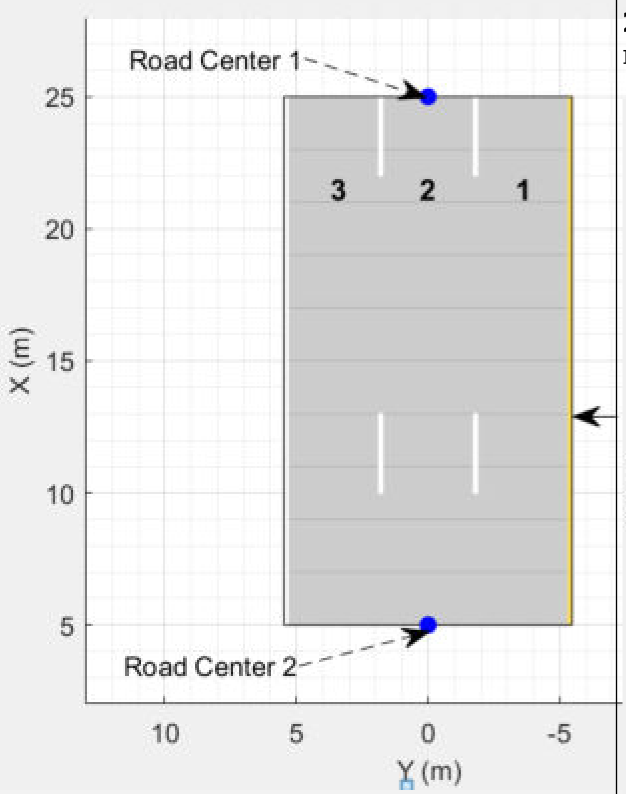
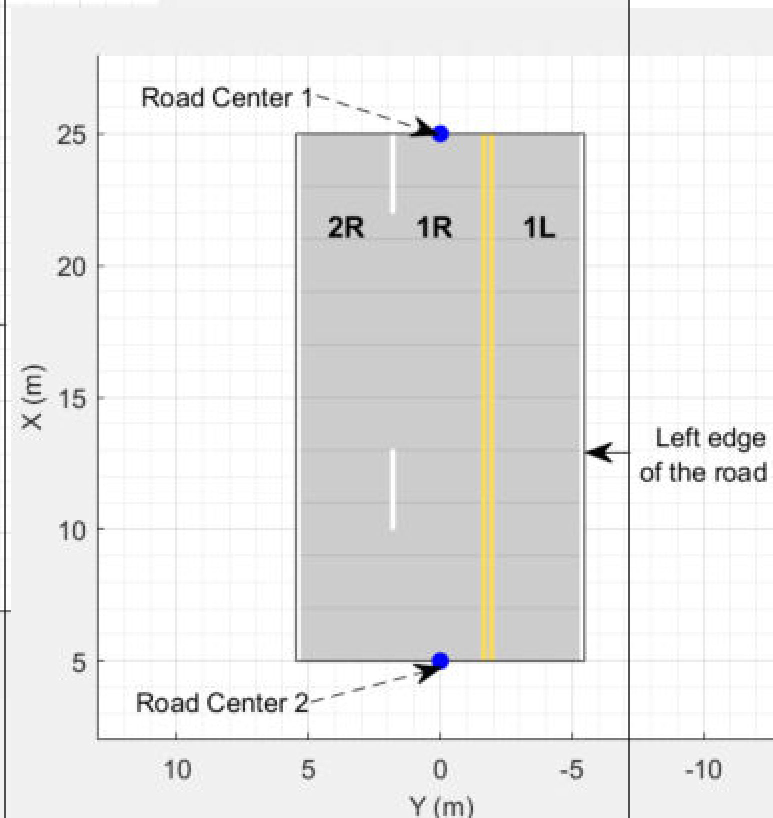


Numbering Lanes

Lanes must be numbered from left to right, with the left edge of the road defined relative to the draw direction of the road. For a one-way road, by default, the left edge of the road is a solid yellow marking which indicates the end of the road in transverse direction (direction perpendicular to draw direction). For a two-way road, by default, both edges are marked with solid white lines.

For example, these diagrams show how the lanes are numbered in a one-way and two-way road with a draw direction from top-to-bottom.

Numbering Lanes in a One-Way Road	Numbering Lanes in a Two-Way Road

Specify the number of lanes as a positive integer for a one-way road. If you set the integer value as `3`, then the road has three lanes that travel in the same direction. The lanes are numbered starting from the left edge of the road. **1**, **2**, **3** denote the first, second, and third lanes of the road, respectively.  	Specify the number of lanes as a two-element vector of positive integer for a two-way road. If you set the vector as `[1 2]`, then the road has three lanes: two lanes traveling in one direction and one lane traveling in the opposite direction. Because of the draw direction, the road has one left lane and two right lanes. The lanes are numbered starting from the left edge of the road. **1L** denote the only left lane of the road. **1R** and **2R** denote the first and second right lanes of the road, respectively. 

The lane specifications apply by the order in which the lanes are numbered.

Algorithms

The `road` function creates a road for an actor to follow in a driving scenario. You specify the road using *N* two-dimensional or three-dimensional waypoints. Each of the *N* – 1 segments between waypoints defines a curve whose curvature varies linearly with distance along the segment. The function fits a piecewise clothoid curve to the (*x*, *y*) coordinates of the waypoints by matching the curvature on both sides of the waypoint. For a nonclosed curve, the curvature at the first and last waypoint is zero. If the first and last waypoints coincide, then the curvatures before and after the endpoints are matched. The *z*-coordinates of the road are interpolated using a shape-preserving piecewise cubic curve.

See Also

Objects
drivingScenario | lanespec

Functions
laneMarking | roadBoundaries | roadNetwork

Topics
"Define Road Layouts Programmatically"
"Create Driving Scenario Programmatically"

Introduced in R2017a

roadNetwork

Add road network to driving scenario

Syntax

```
roadNetwork(scenario,'OpenDRIVE',filename)
roadNetwork(scenario,'OpenDRIVE','ShowLaneTypes',showLaneTypes)

roadNetwork(scenario,'HEREHDLiveMap',lat,lon)
roadNetwork(scenario,'HEREHDLiveMap',minLat,minLon,maxLat,maxLon)

roadNetwork(scenario,'OpenStreetMap',filename)
```

Description

OpenDRIVE

`roadNetwork(scenario,'OpenDRIVE',filename)` imports roads from an OpenDRIVE road network file into a driving scenario. This function supports OpenDRIVE format specification version 1.4H [1].

`roadNetwork(scenario,'OpenDRIVE','ShowLaneTypes',showLaneTypes)` uses the name-value pair `'ShowLaneTypes'` to also import lane type information from the file and display it in the driving scenario.

HERE HD Live Map

`roadNetwork(scenario,'HEREHDLiveMap',lat,lon)` imports roads from a HERE HD Live Map[4] (HERE HDLM) road network into a driving scenario. The function imports the roads that are nearest to the latitude and longitude coordinates specified in `lat` and `lon`, respectively.

`roadNetwork(scenario,'HEREHDLiveMap',minLat,minLon,maxLat,maxLon)` imports HERE HDLM roads that are at least partially within the geographic bounding box specified by `minLat`, `minLon`, `maxLat`, and `maxLon`.

OpenStreetMap

`roadNetwork(scenario,'OpenStreetMap',filename)` imports roads from an OpenStreetMap road network file into a driving scenario.

Examples

Import OpenDRIVE Road Network into Driving Scenario

Create an empty driving scenario.

```
scenario = drivingScenario;
```

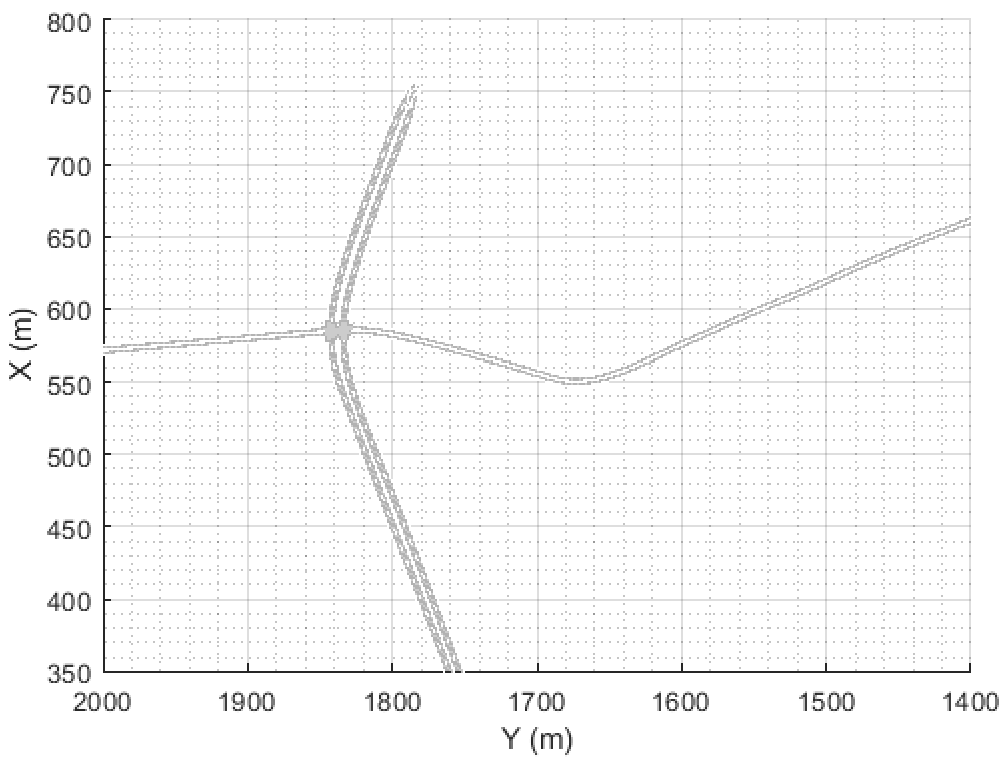
4. You need to enter into a separate agreement with HERE in order to gain access to the HDLM services and to get the required credentials (access_key_id and access_key_secret) for using the HERE Service.

Import an OpenDRIVE road network into the scenario.

```
filePath = 'intersection.xodr';
roadNetwork(scenario,'OpenDRIVE',filePath);
```

Plot the scenario and zoom in on the road network by setting the axes limits.

```
plot(scenario)
xlim([350 800])
ylim([1400 2000])
zlim([0.00 10.00])
```



Import OpenDRIVE Road with Multiple Lane Types into Driving Scenario

Create an empty driving scenario.

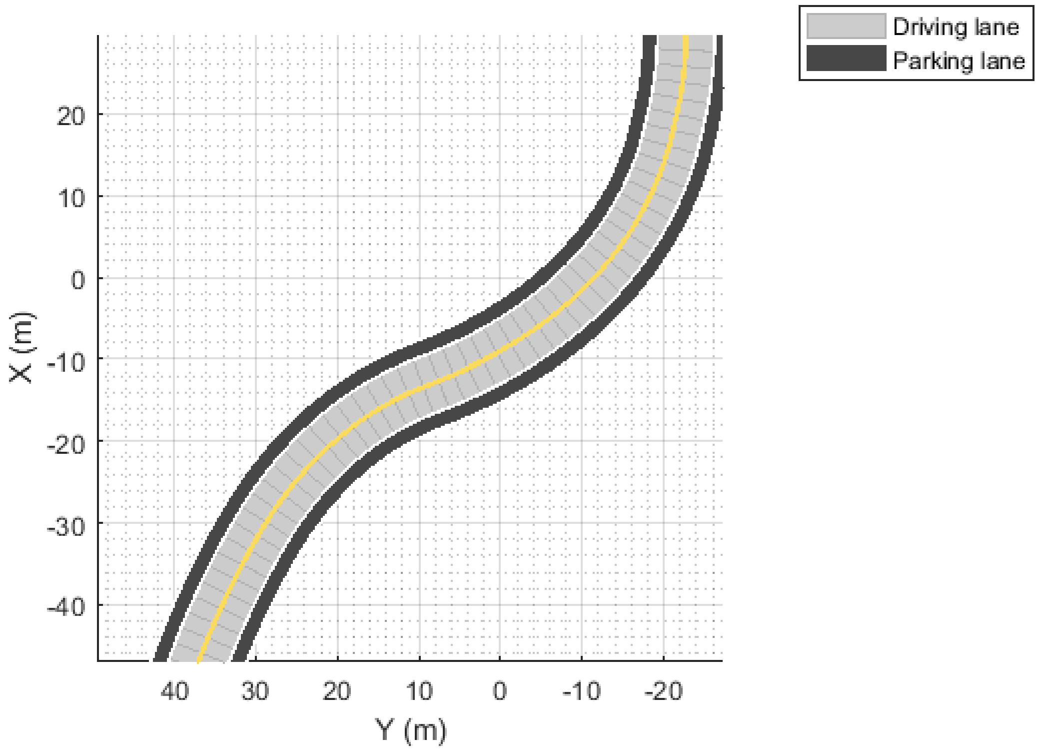
```
scenario = drivingScenario;
```

Import an OpenDRIVE road composed of driving and parking lanes into the scenario. By default, the function interprets the lane type information and imports the lanes into driving scenario without altering the lane type.

```
filePath = 'parking.xodr';
roadNetwork(scenario,'OpenDRIVE',filePath);
```

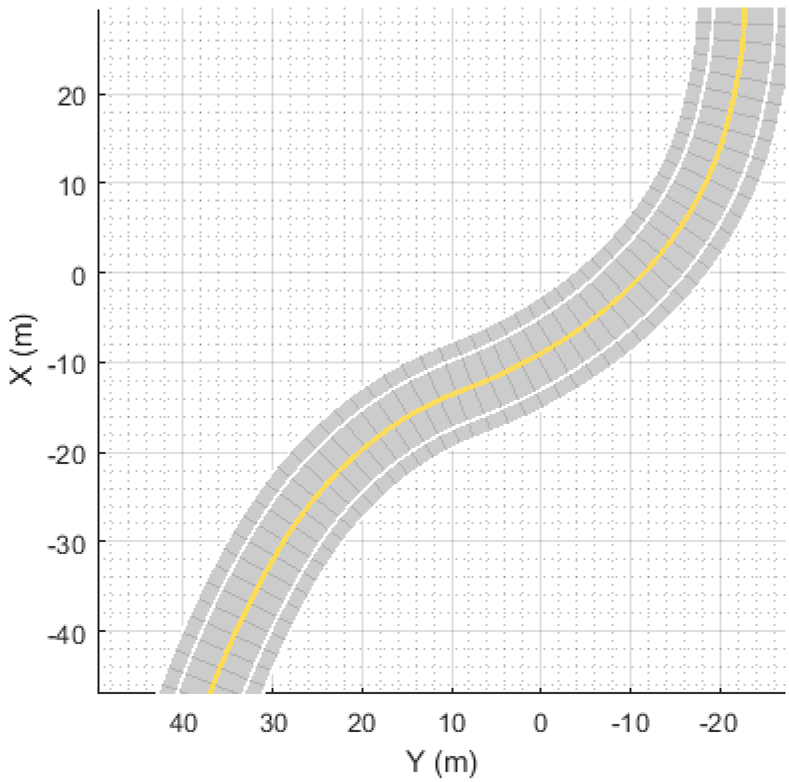
Plot the scenario.

```
plot(scenario)
zoom(2)
legend('Driving lane','Parking lane')
```



Import the OpenDRIVE road into the scenario. Set the `'ShowLaneTypes'` value to `false` to suppress multiple lane types. The function ignores the lane type information and imports all the lanes as driving lanes.

```
scenario = drivingScenario;
roadNetwork(scenario,'OpenDRIVE',filePath,'ShowLaneTypes',false);
plot(scenario)
zoom(2)
```



Import HERE HDLM Roads Using Specified Coordinates

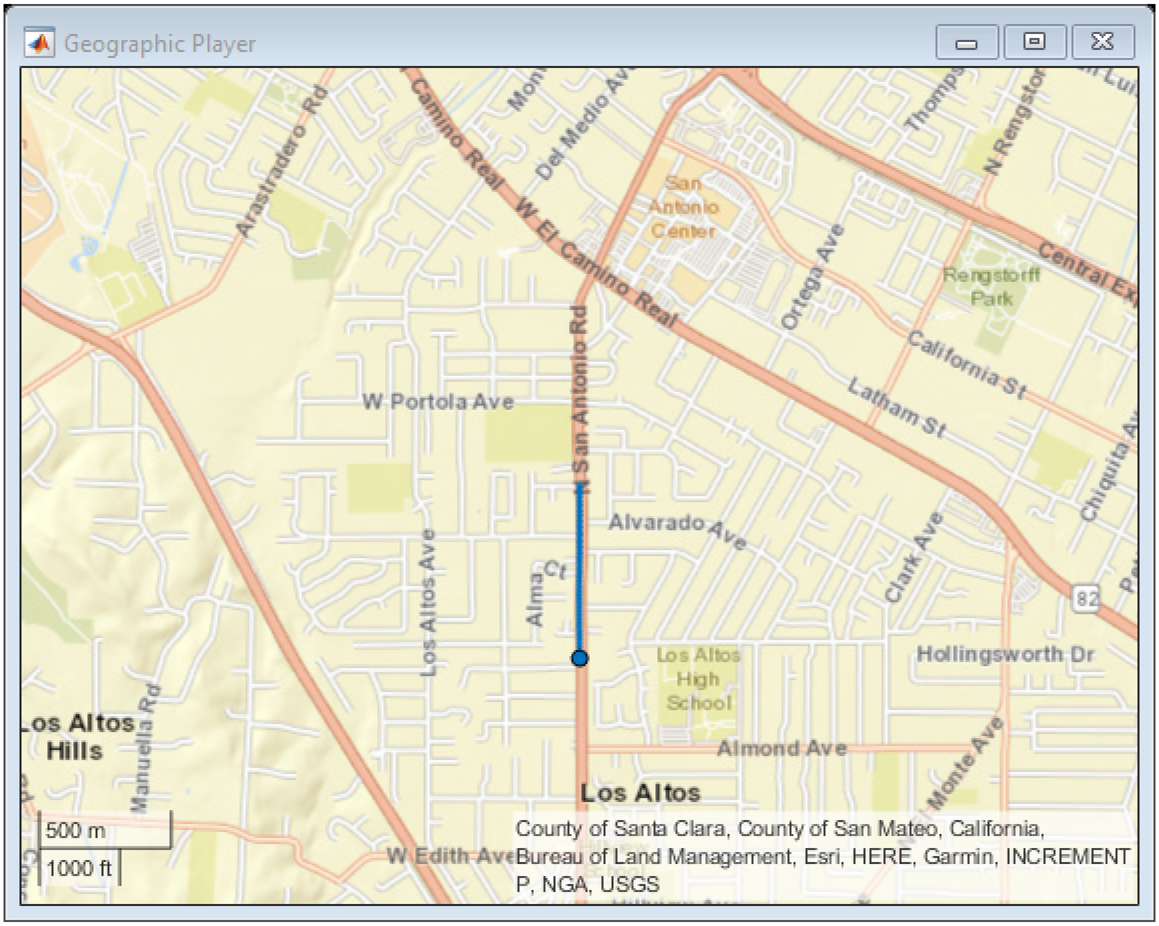
Import HERE HDLM road network data that is nearest to the coordinates of a specified driving route into a driving scenario. Plot a vehicle following this route in the driving scenario.

Load a sequence of geographic coordinates that correspond to a driving route.

```
data = load('geoSequence.mat');
lat = data.latitude;
lon = data.longitude;
```

Display the route by streaming the coordinates on a geographic player. Set the zoom level to 14 and configure the player to display all points in its history. To speed up the streaming, plot only every tenth coordinate in the route.

```
zoomLevel = 14;
player = geoplayer(lat(1),lon(1),zoomLevel,'HistoryDepth',Inf);
timestep = 10;

for i = 1:timestep:length(lat)
    plotPosition(player,lat(i),lon(i));
end
```



Create a driving scenario. Import the HERE HDLM road data that is nearest to the driving route into the scenario.

```
scenario = drivingScenario;
roadNetwork(scenario,'HEREHDLiveMap',lat,lon);
```

Use the `latlon2local` function to convert the driving route from geographic coordinates to local east-north-up (ENU) Cartesian coordinates used in the driving scenario. For the origin of the ENU coordinate system, use the geographic road network origin stored in the `GeoReference` property of the scenario. The origin is the first coordinate specified in the driving route. Because the driving route contains only latitudinal and longitudinal data, set the altitude to `0`.

```
alt = 0;
origin = scenario.GeoReference;
[xEast,yNorth,zUp] = latlon2local(lat,lon,alt,origin);
```

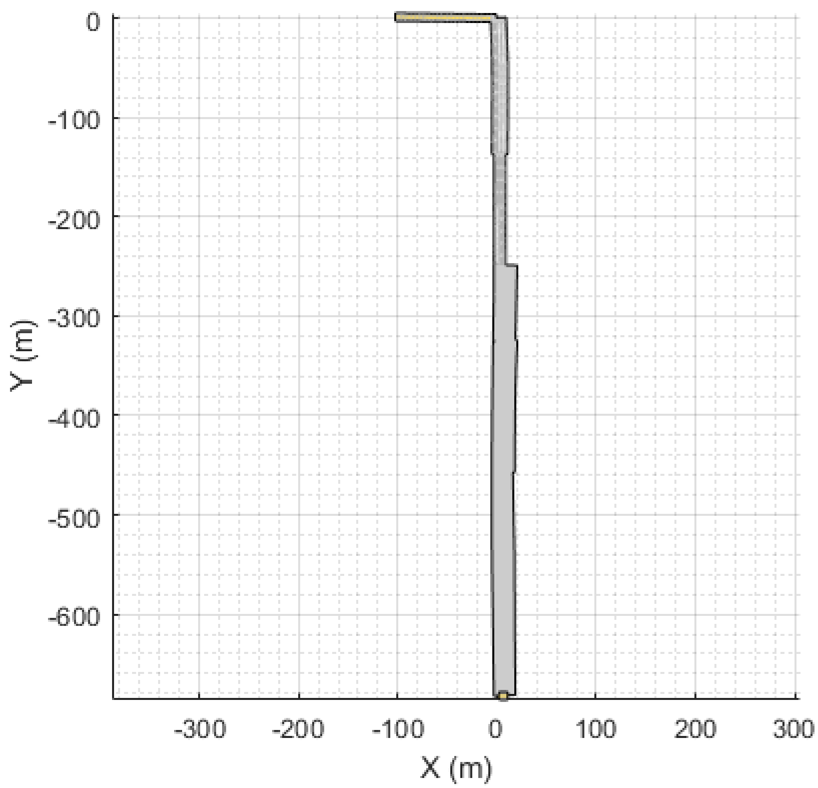
Add a vehicle to the driving scenario. Specify the converted driving route as the trajectory of the vehicle. Set a vehicle speed of 30 meters per second.

```
v = vehicle(scenario,'ClassID',1);
speed = 30;
trajectory(v,[xEast,yNorth,zUp],speed);
```

Plot the scenario and pause every 0.01 seconds to slow down the simulation. To maintain the same alignment with geographic coordinate displays, the *X*-axis is on the bottom and the *Y*-axis is on the left. In driving scenarios 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.

After a few seconds, the vehicle appears to drive underneath the road. This issue occurs because the converted trajectory contains no altitude data but the imported road network does. To avoid this issue, if you are specifying a driving route recorded from a GPS, include the altitude data.

```
plot(scenario)
while advance(scenario)
    pause(0.01)
end
```



Import HERE HDLM Roads Using Specified Region

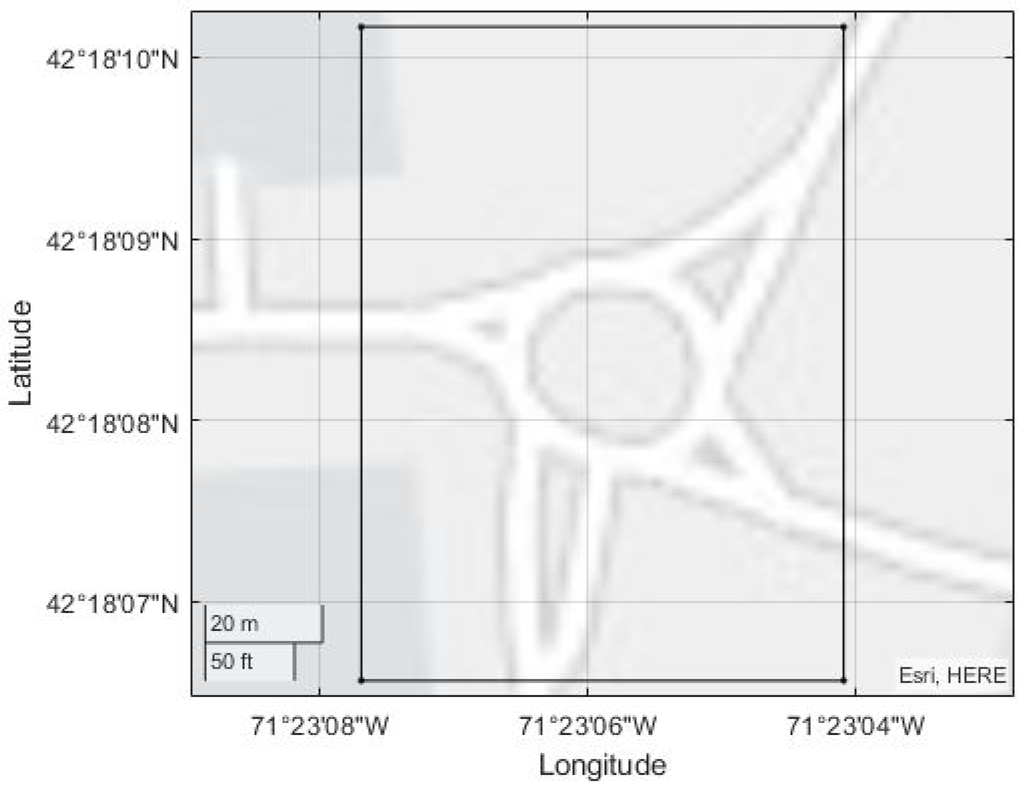
Import HERE HDLM road network data into driving scenario. Select this data from a region that is centered around a specified geographic coordinate.

Define a latitude and longitude coordinates corresponding to a roundabout.

```
latCenter = 42.302324;
lonCenter = -71.384970;
```

Specify the minimum and maximum latitudinal and longitudinal coordinates for a rectangular region around the roundabout. Display a bounding box corresponding to this region on a geographic plot.

```
offset = 5e-4;
minLat = latCenter - offset;
minLon = lonCenter - offset;
maxLat = latCenter + offset;
maxLon = lonCenter + offset;

gx = geoaxes;
LineSpec = '.-k';
geoplot(gx, ...
        [minLat maxLat],[minLon minLon],LineSpec, ...
        [maxLat maxLat],[minLon maxLon],LineSpec, ...
        [maxLat minLat],[maxLon maxLon],LineSpec, ...
        [minLat minLat],[maxLon minLon],LineSpec)
```

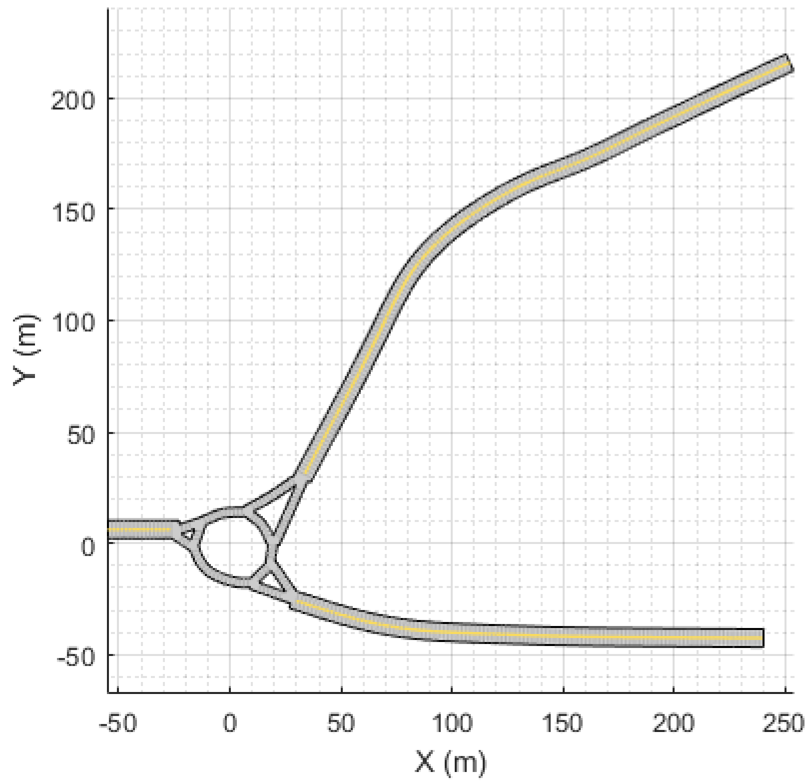


Create a driving scenario and import roads from the region by using the minimum and maximum coordinates. The `roadNetwork` function imports roads that are at least partially within this region.

```
scenario = drivingScenario;
roadNetwork(scenario,'HEREHDLiveMap',minLat,minLon,maxLat,maxLon);
```

Plot the scenario. To maintain the same alignment with geographic coordinate displays, the *X*-axis is on the bottom and the *Y*-axis is on the left. In driving scenarios 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.

```
plot(scenario)
```



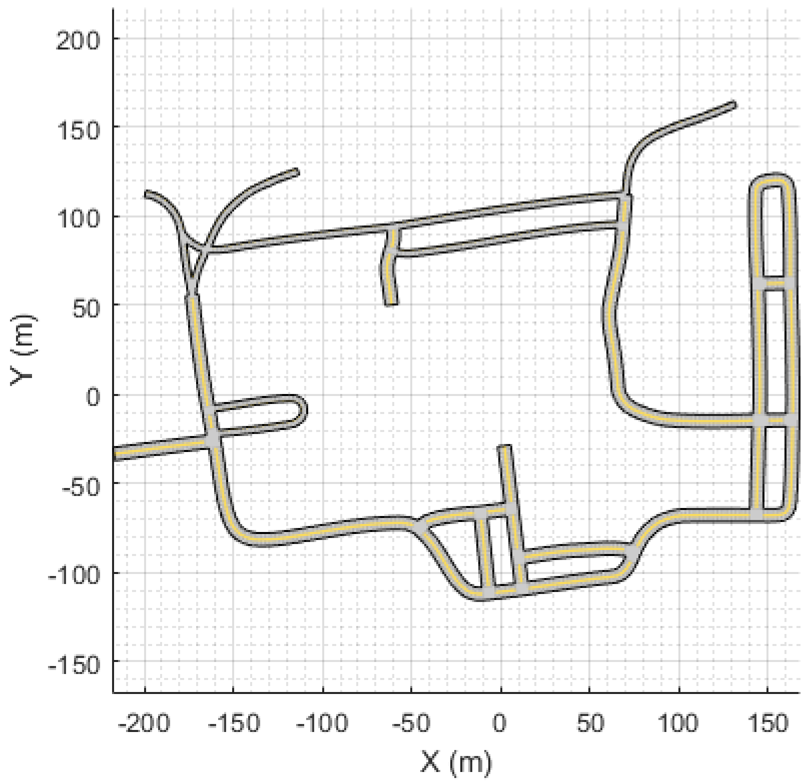
Import OpenStreetMap Road Network and Plot Route

Import roads from the OpenStreetMap® web service into a driving scenario. Then, plot a vehicle following a route in the imported road network.

Import a road network of the MathWorks® Apple Hill campus into an empty driving scenario. 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/.

Plot the imported road network. To maintain the same alignment with geographic coordinate displays, the *X*-axis is on the bottom and the *Y*-axis is on the left. In driving scenarios 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.

```
scenario = drivingScenario;
roadNetwork(scenario,'OpenStreetMap','applehill.osm');
plot(scenario)
```



Load the latitude and longitude coordinates for a driving route in this road network.

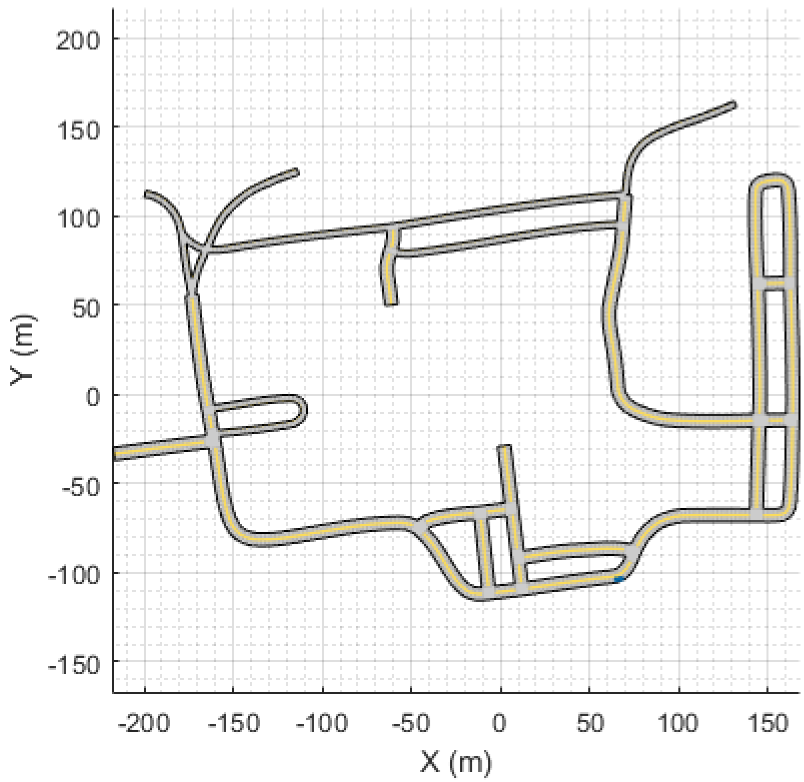
```
data = load('geoRouteAH.mat');
lat = data.latitude;
lon = data.longitude;
```

Use the `latlon2local` function to convert the driving route from geographic coordinates to local east-north-up (ENU) Cartesian coordinates used in the driving scenario. For the origin of the ENU coordinate system, use the geographic road network origin stored in the `GeoReference` property of the scenario. The origin is the first coordinate specified in the driving route. Because the driving route contains only latitudinal and longitudinal data, set the altitude to 0.

```
alt = 0;
origin = scenario.GeoReference;
[xEast,yNorth,zUp] = latlon2local(lat,lon,alt,origin);
```

Add a vehicle to the driving scenario. Specify the converted driving route as the trajectory of the vehicle. Set a vehicle speed of 30 meters per second. Plot the vehicle trajectory and pause every 0.01 seconds to slow down the simulation.

```
v = vehicle(scenario,'ClassID',1);
speed = 30;
trajectory(v,[xEast,yNorth,zUp],speed);

while advance(scenario)
    pause(0.01)
end
```



Input Arguments

scenario — Driving scenario
drivingScenario object

Driving scenario, specified as a `drivingScenario` object. `scenario` must contain no previously created or imported roads.

filename — Name of road network file
character vector | string scalar

Name of the road network file, specified as a character vector or string scalar.

`filename` must specify a file in the current folder, a file that is on the MATLAB search path, or a full or relative path to a file.

`filename` must end with a file extension that is valid for the source of the road network.

Road Network Source	Valid File Extensions	Sample Syntax
OpenDRIVE	.xodr .xml	`roadNetwork(scenario, ...` `'OpenDRIVE','C:\Desktop\roads.xodr')`

Road Network Source	Valid File Extensions	Sample Syntax

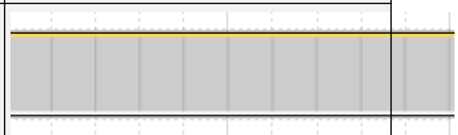
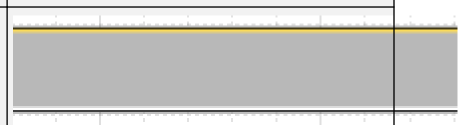
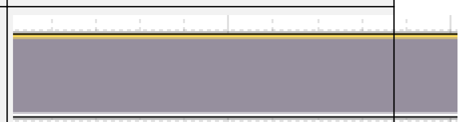
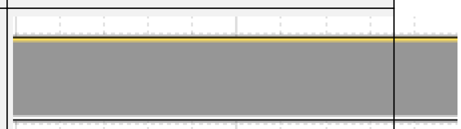
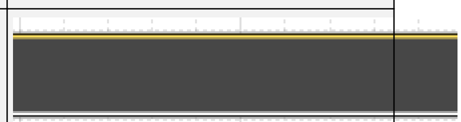
OpenStreetMap	`.osm`
	
	`.xml`
		`roadNetwork(scenario, ...`
		`    'OpenStreetMap','C:\Desktop\map.osm`

showLaneTypes — Import lane type information
`true` or 1 (default) | `false` or 0

Import lane type information from the OpenDRIVE road network file and display it in the driving scenario, specified as a comma-separated pair consisting of `'ShowLaneTypes'` and one of these values:

- `true` or 1 — Import lane type information and render lane types.
- `false` or 0 — Ignore lane type information and import all lanes as driving lanes in the driving scenario.

The table summarized the supported lane types and their default appearance after importing them into the driving scenario.

Supported Lane Types	Description	Default Appearance
Driving lanes	Lanes for driving	
Border lanes	Lanes at the road borders	
Restricted lanes	Lanes reserved for high-occupancy vehicles	
Shoulder lanes	Lanes reserved for emergency stopping	
Parking lanes	Lanes alongside driving lanes, intended for parking vehicles	

Any other unsupported lane types are rendered as border lanes.

Example: `'ShowLaneTypes',false`

lat — Latitude coordinates
vector of elements in range [–90, 90]

Latitude coordinates, specified as a vector of elements in the range [–90, 90]. `lat` must be the same size as `lon`. Units are in degrees.

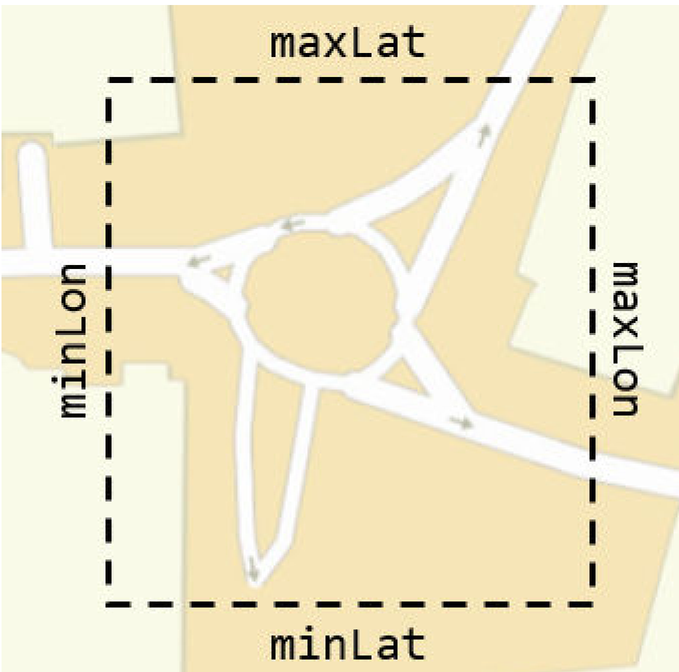
`lon` — Longitude coordinates
vector of elements in range [–180, 180]

Longitude coordinates, specified as a vector of elements in the range [–180, 180]. `lon` must be the same size as `lat`. Units are in degrees.

`minLat` — Minimum latitude coordinate of bounding box
scalar in range [–90, 90]

Minimum latitude coordinate of the bounding box, specified as a scalar in the range [–90, 90]. `minLat` must be less than `maxLat`. Units are in degrees.

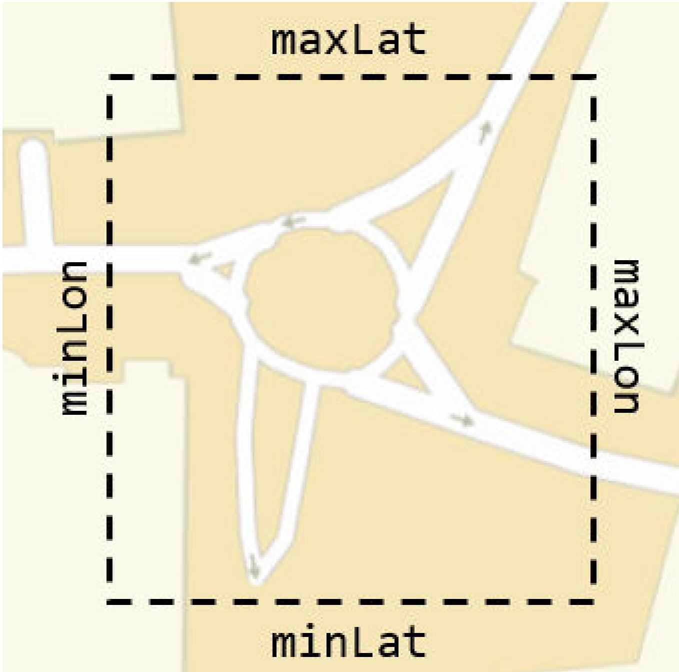
The `roadNetwork` function imports any roads that are at least partially within the bounding box specified by inputs `minLat`, `minLon`, `maxLat`, and `maxLon`. This diagram displays the relationship between these coordinates.



`minLon` — Minimum longitude coordinate of bounding box
scalar in range [–180, 180]

Minimum longitude coordinate of the bounding box, specified as a scalar in the range [–180, 180]. `minLon` must be less than `maxLon`. Units are in degrees.

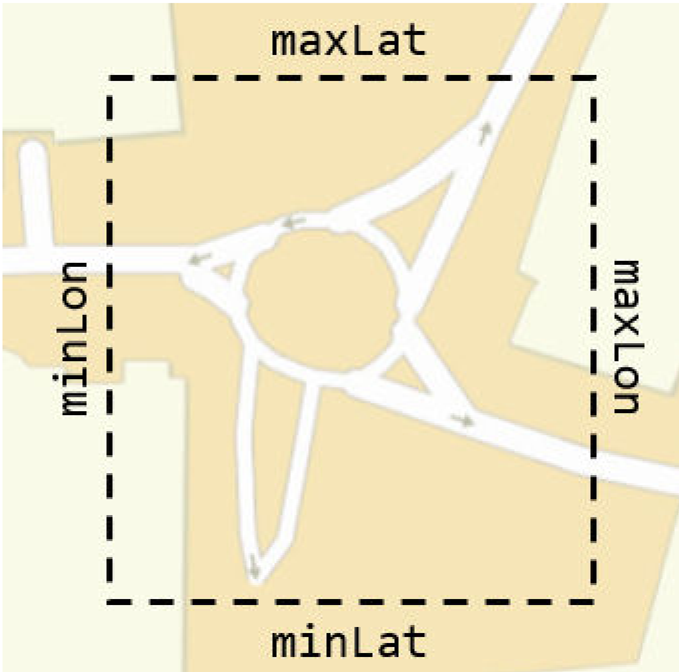
The `roadNetwork` function imports any roads that are at least partially within the bounding box specified by inputs `minLat`, `minLon`, `maxLat`, and `maxLon`. This diagram displays the relationship between these coordinates.



`maxLat` — Maximum latitude coordinate of bounding box
scalar in range [–90, 90]

Maximum latitude coordinate of the bounding box, specified as a scalar in the range [–90, 90]. `maxLat` must be greater than `minLat`. Units are in degrees.

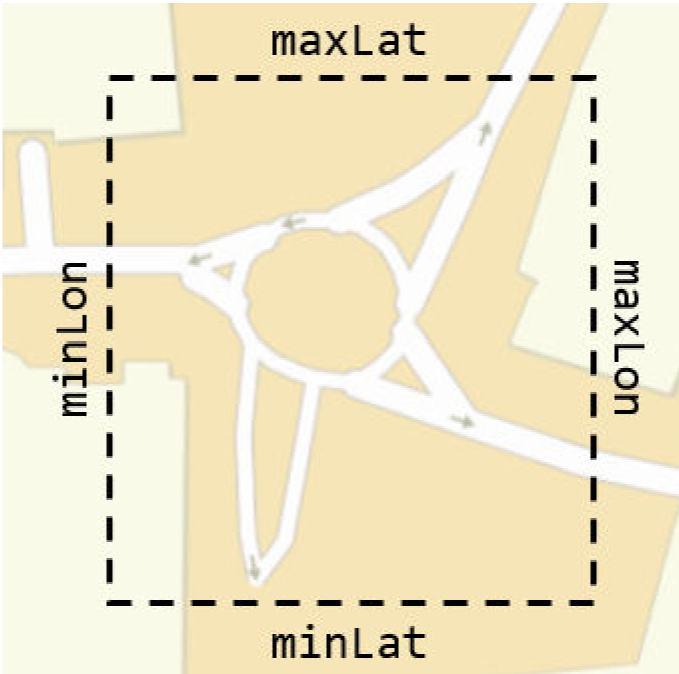
The `roadNetwork` function imports any roads that are at least partially within the bounding box specified by inputs `minLat`, `minLon`, `maxLat`, and `maxLon`. This diagram displays the relationship between these coordinates.



maxLon — Maximum longitude coordinate of bounding box
scalar in range [–180, 180]

Maximum longitude coordinate of the bounding box, specified as a scalar in the range [–180, 180]. `maxLon` must be greater than `minLon`. Units are in degrees.

The `roadNetwork` function imports any roads that are at least partially within the bounding box specified by inputs `minLat`, `minLon`, `maxLat`, and `maxLon`. This diagram displays the relationship between these coordinates.



Limitations

OpenDRIVE Import Limitations

- 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. 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 function sets the lane width to 4 meters throughout.
- Roads with lane type information specified as `driving`, `border`, `restricted`, `shoulder`, and `parking` are supported. Lanes with any other lane type information are imported as border lanes.
- Roads with multiple lane marking styles are not supported. The function applies the first found marking style to all lanes in the road. For example, if a road has `Dashed` and `Solid` lane markings, the function applies `Dashed` lane markings throughout.
- Lane marking styles `Bott Dots`, `Curbs`, and `Grass` are not supported. Lanes with these marking styles are imported as unmarked.

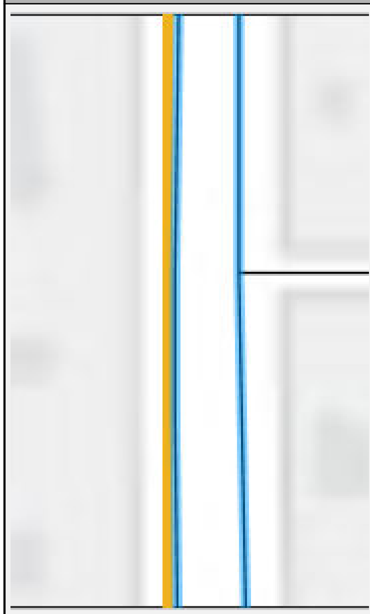
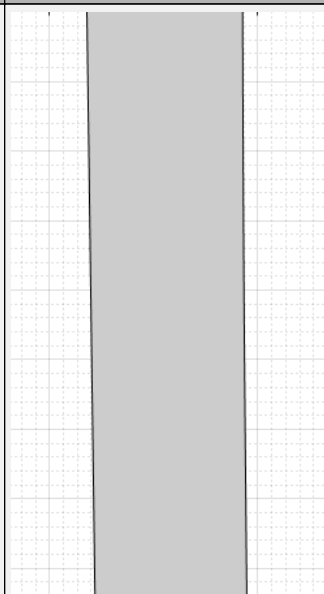
HERE HD Live Map Import Limitations

When importing HERE HDLM data, these road and lane features are not supported:

- Lanes with varying widths — In the generated road network, each lane is set to have the maximum width found along its entire length. Consider a HERE HDLM lane with a width that varies from 2 to 4 meters along its length. In the generated road network, the lane width is 4 meters along its entire length.

- Roads with varying numbers of lanes along their lengths — In the generated road network, each road is set to have the maximum number of lanes along its entire length. Consider a HERE HDLM road with 3 lanes on one half and 2 lanes on the other half. In the generated road network, the road has 3 lanes along its entire length.
- Multiple lane marking styles along a lane — In the generated road network, each lane is set to have the marking style of the lane segment with the maximum width along the road. Consider a HERE HDLM lane with 2 lane segments. The first lane segment is 2 meters wide and has solid markings. The second lane segment is 4 meters wide and has dashed markings. In the generated road network, the lane has a fixed width of 4 meters throughout and dashed markings along its entire length.

These modifications to the road networks can sometimes cause roads to overlap in the driving scenario. Consider the HERE HDLM roads for the divided highway highlighted in blue in the table. Due to the unsupported features, in the imported driving scenario, the lane widths of the roads increase. This limitation causes the roads to overlap and appear as one road. Sensors that detect lanes are unable to detect the covered lanes.

HERE HDLM Road Network	Imported Driving Scenario
	

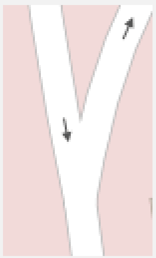
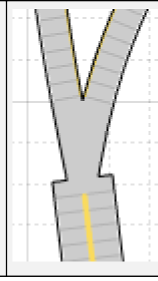
If you receive a warning that the geometry of a road is unable to be computed, then the curvature of the road is too sharp for it to render properly and it is not imported.

OpenStreetMap Import Limitations

When importing OpenStreetMap data, road and lane features have these limitations:

- Lane-level information is not imported from OpenStreetMap roads. Lane specifications are based only on the direction of travel specified in the OpenStreetMap road network, where:
  - One-way roads are imported as single-lane roads with default lane specifications. These lanes are programmatically equivalent to `lanespec(1)`.
  - Two-way roads are imported as two-lane roads with bidirectional travel and default lane specifications. These lanes are programmatically equivalent to `lanespec([1 1])`.

The table shows these differences in the OpenStreetMap road network and the road network in the imported driving scenario.

OpenStreetMap Road Network	Imported Driving Scenario
	

- When importing OpenStreetMap road networks that specify elevation data, if elevation data is not specified for all roads being imported, then the generated road network might contain inaccuracies and some roads might overlap.
- The basemap used in the app can have slight differences from the map used in the OpenStreetMap service. Some imported road issues might also be due to missing or inaccurate map data in the OpenStreetMap service. To check whether the data is missing or inaccurate due to the map service, consider viewing the map data on an external map viewer.
- If you receive a warning that the geometry of a road is unable to be computed, then the curvature of the road is too sharp for it to render properly and it is not imported.

Tips

- If the roads that you import do not look as expected, consider importing them by using the Driving Scenario Designer app. The app can make the process of troubleshooting and correcting roads easier than trying to troubleshoot and correct them by using the `roadNetwork` function.

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

Objects
`drivingScenario`

Functions
`actor` | `trajectory` | `vehicle`

Topics
"Import OpenDRIVE Roads into Driving Scenario"
"Import HERE HD Live Map Roads into Driving Scenario"
"Import OpenStreetMap Data into Driving Scenario"
"Scenario Generation from Recorded Vehicle Data"

External Websites
ASAM OpenDRIVE
HERE Technologies
openstreetmap.org

Introduced in R2018b

roadBoundaries

Package:

Get road boundaries

Syntax

```
rbdry = roadBoundaries(scenario)
rbdry = roadBoundaries(ac)
```

Description

`rbdry = roadBoundaries(scenario)` returns the road boundaries, `rbdry`, of a driving scenario, `scenario`.

`rbdry = roadBoundaries(ac)` returns the road boundaries that the actor, `ac`, follows in a driving scenario.

Examples

Create Driving Scenario with Multiple Actors and Roads

Create a driving scenario containing a curved road, two straight roads, and two actors: a car and a bicycle. Both actors move along the road for 60 seconds.

Create the driving scenario object.

```
scenario = drivingScenario('SampleTime',0.1','StopTime',60);
```

Create the curved road using road center points following the arc of a circle with an 800-meter radius. The arc starts at 0°, ends at 90°, and is sampled at 5° increments.

```
angs = [0:5:90]';
R = 800;
roadcenters = R*[cosd(angs) sind(angs) zeros(size(angs))];
roadwidth = 10;
road(scenario,roadcenters,roadwidth);
```

Add two straight roads with the default width, using road center points at each end.

```
roadcenters = [700 0 0; 100 0 0];
road(scenario,roadcenters)
```

```
ans =
  Road with properties:

           Name: ""
         RoadID: 2
    RoadCenters: [2x3 double]
      RoadWidth: 6
```

```
      BankAngle: [2x1 double]
```

```
roadcenters = [400 400 0; 0 0 0];
road(scenario,roadcenters)
```

```
ans = 
  Road with properties:

           Name: ""
         RoadID: 3
    RoadCenters: [2x3 double]
      RoadWidth: 6
      BankAngle: [2x1 double]
```

Get the road boundaries.

```
rbdry = roadBoundaries(scenario);
```

Add a car and a bicycle to the scenario. Position the car at the beginning of the first straight road.

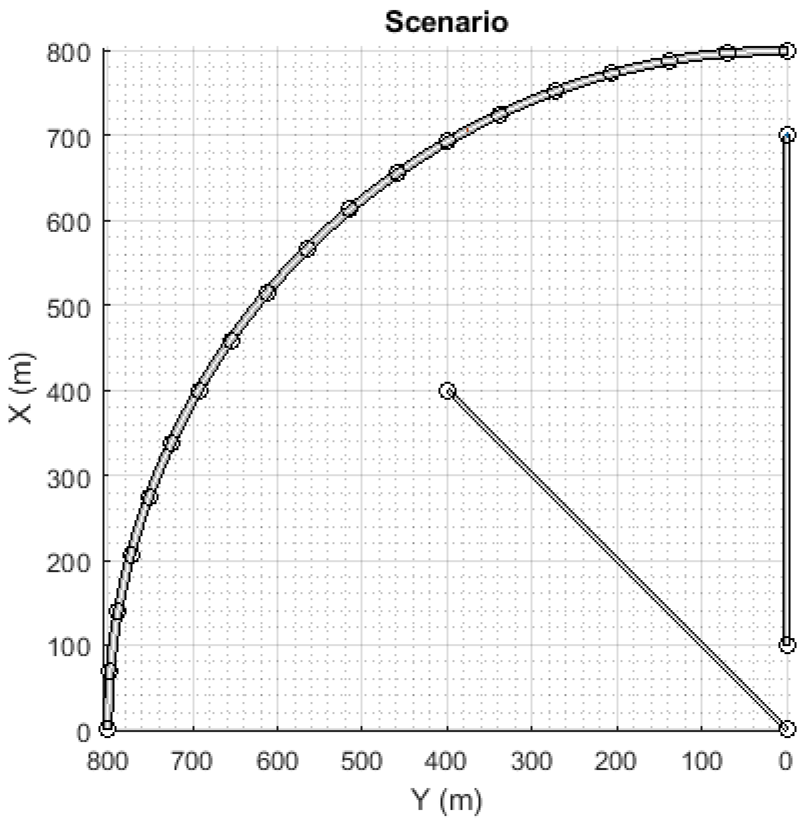
```
car = vehicle(scenario,'ClassID',1,'Position',[700 0 0], ...
    'Length',3,'Width',2,'Height',1.6);
```

Position the bicycle farther down the road.

```
bicycle = actor(scenario,'ClassID',3,'Position',[706 376 0]', ...
    'Length',2,'Width',0.45,'Height',1.5);
```

Plot the scenario.

```
plot(scenario,'Centerline','on','RoadCenters','on');
title('Scenario');
```



Display the actor poses and profiles.

```
poses = actorPoses(scenario)
```

```
poses=2×1 struct array with fields:
    ActorID
    Position
    Velocity
    Roll
    Pitch
    Yaw
    AngularVelocity
```

```
profiles = actorProfiles(scenario)
```

```
profiles=2×1 struct array with fields:
    ActorID
    ClassID
    Length
    Width
    Height
    OriginOffset
    MeshVertices
    MeshFaces
    RCSPattern
    RCSAzimuthAngles
```

```
RCSElevationAngles
```

Create and Display Road Boundaries

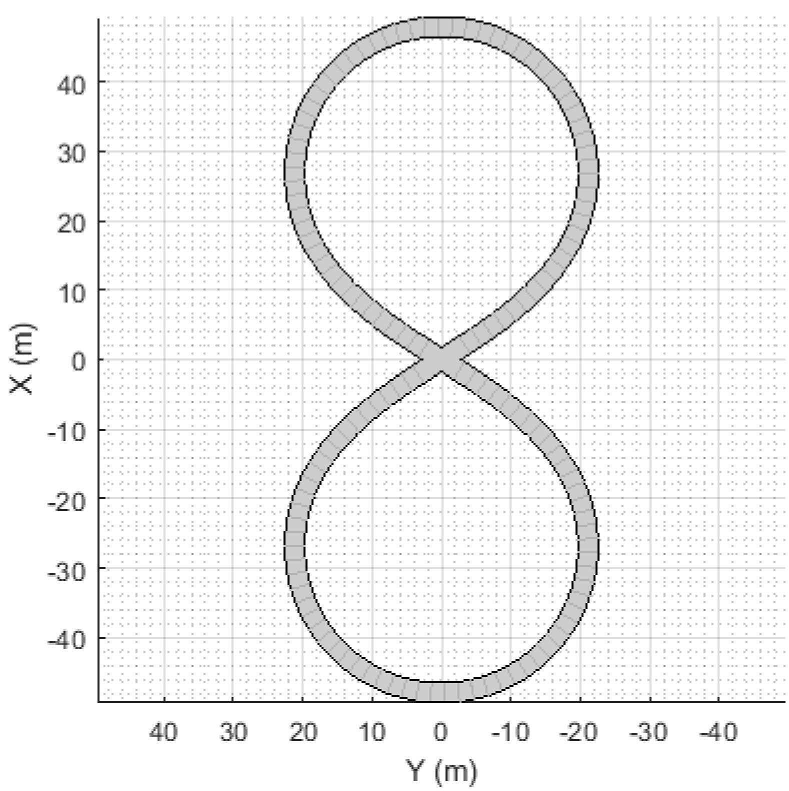
Create a driving scenario containing a figure-8 road specified in the world coordinates of the scenario. Convert the world coordinates of the scenario to the coordinate system of the ego vehicle.

Create an empty driving scenario.

```
scenario = drivingScenario;
```

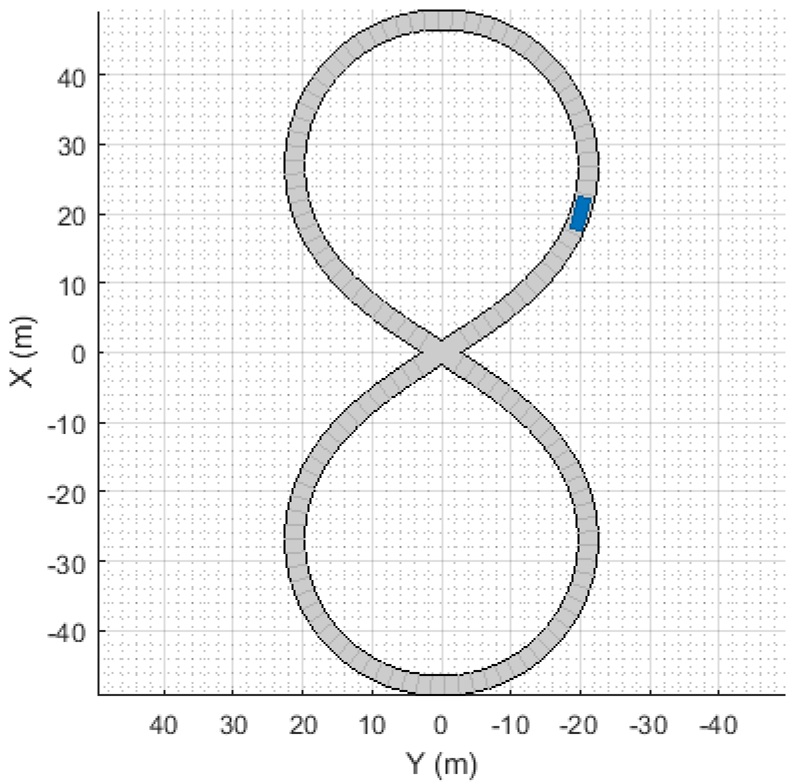
Add a figure-8 road to the scenario. Display the scenario.

```
roadCenters = [0  0  1
             20 -20  1
             20  20  1
            -20 -20  1
            -20  20  1
              0   0  1];

roadWidth = 3;
bankAngle = [0 15 15 -15 -15 0];
road(scenario,roadCenters,roadWidth,bankAngle);
plot(scenario)
```



Add an ego vehicle to the scenario. Position the vehicle at world coordinates (20, –20) and orient it at a –15 degree yaw angle.

```
ego = actor(scenario,'ClassID',1,'Position',[20 -20 0],'Yaw',-15);
```

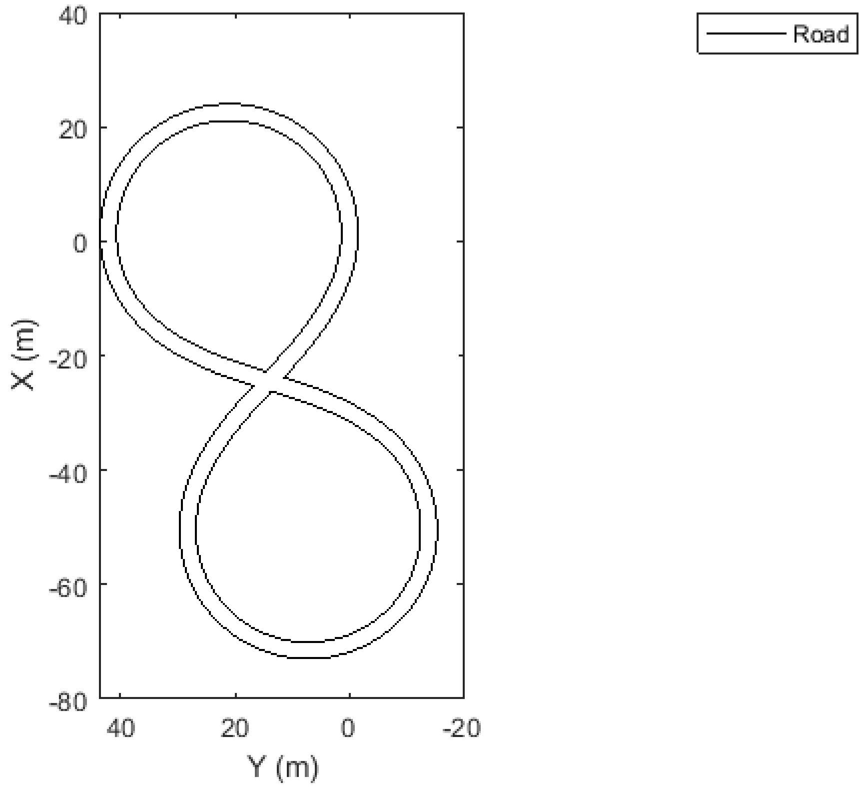


Obtain the road boundaries in ego vehicle coordinates by using the `roadBoundaries` function. Specify the ego vehicle as the input argument.

```
rbEgo1 = roadBoundaries(ego);
```

Display the result on a bird's-eye plot.

```
bep = birdsEyePlot;
lbp = laneBoundaryPlotter(bep,'DisplayName','Road');
plotLaneBoundary(lbp,rbEgo1)
```



Obtain the road boundaries in world coordinates by using the `roadBoundaries` function. Specify the scenario as the input argument.

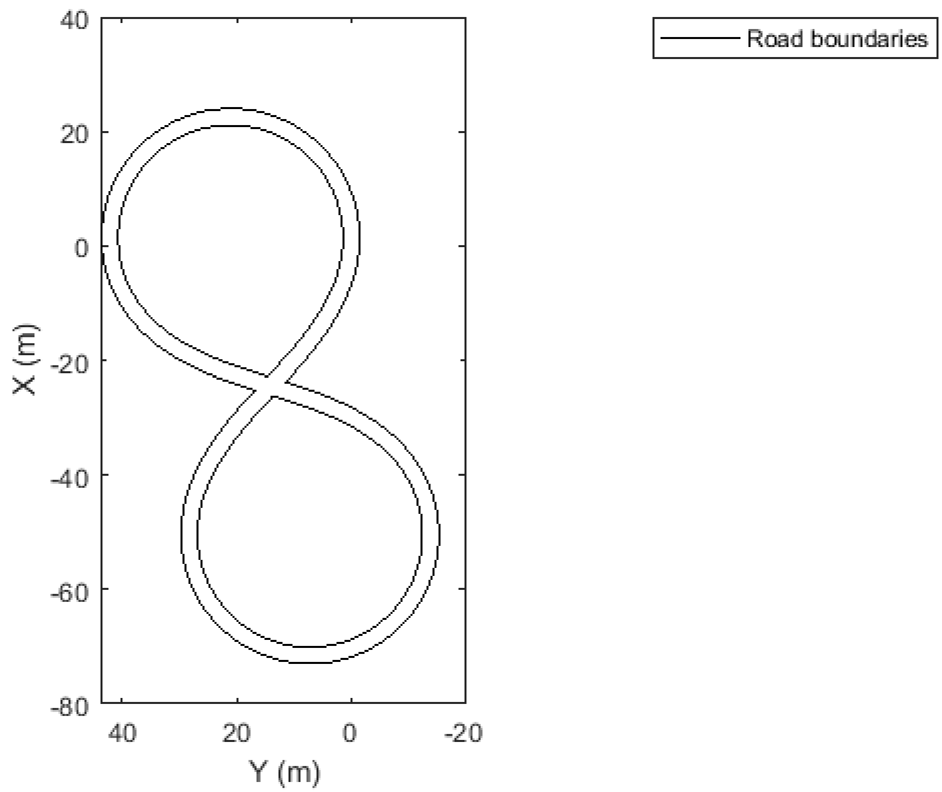
```
rbScenario = roadBoundaries(scenario);
```

Obtain the road boundaries in ego vehicle coordinates by using the `driving.scenario.roadBoundariesToEgo` function.

```
rbEgo2 = driving.scenario.roadBoundariesToEgo(rbScenario,ego);
```

Display the road boundaries on a bird's-eye plot.

```
bep = birdsEyePlot;
lbp = laneBoundaryPlotter(bep,'DisplayName','Road boundaries');
plotLaneBoundary(lbp,{rbEgo2})
```



Input Arguments

`scenario` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

`ac` — Actor
`Actor` object | `Vehicle` object

Actor belonging to a `drivingScenario` object, specified as an `Actor` or `Vehicle` object. To create these objects, use the `actor` and `vehicle` functions, respectively.

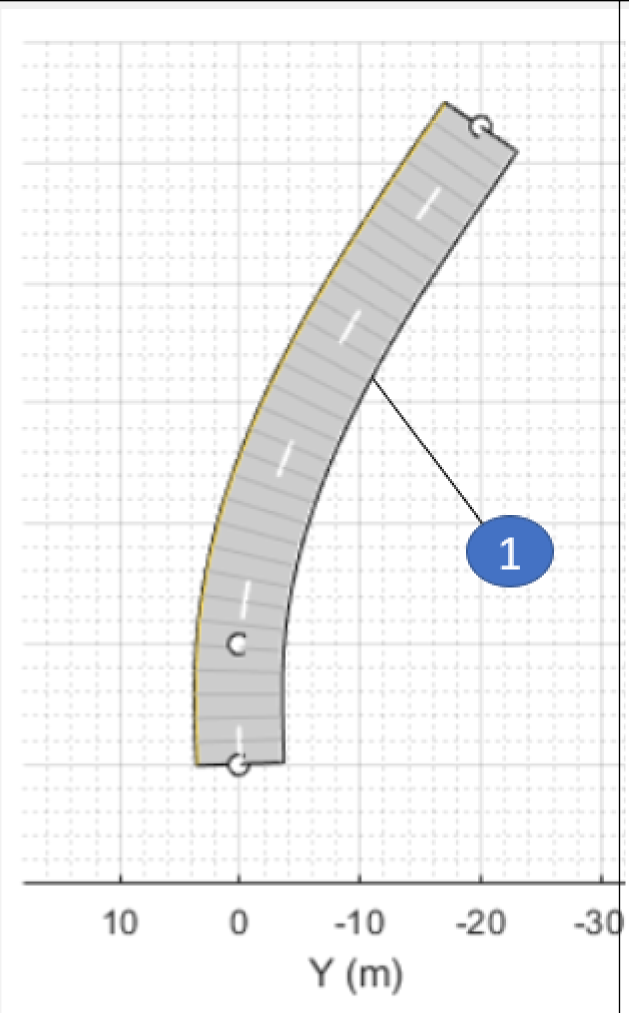
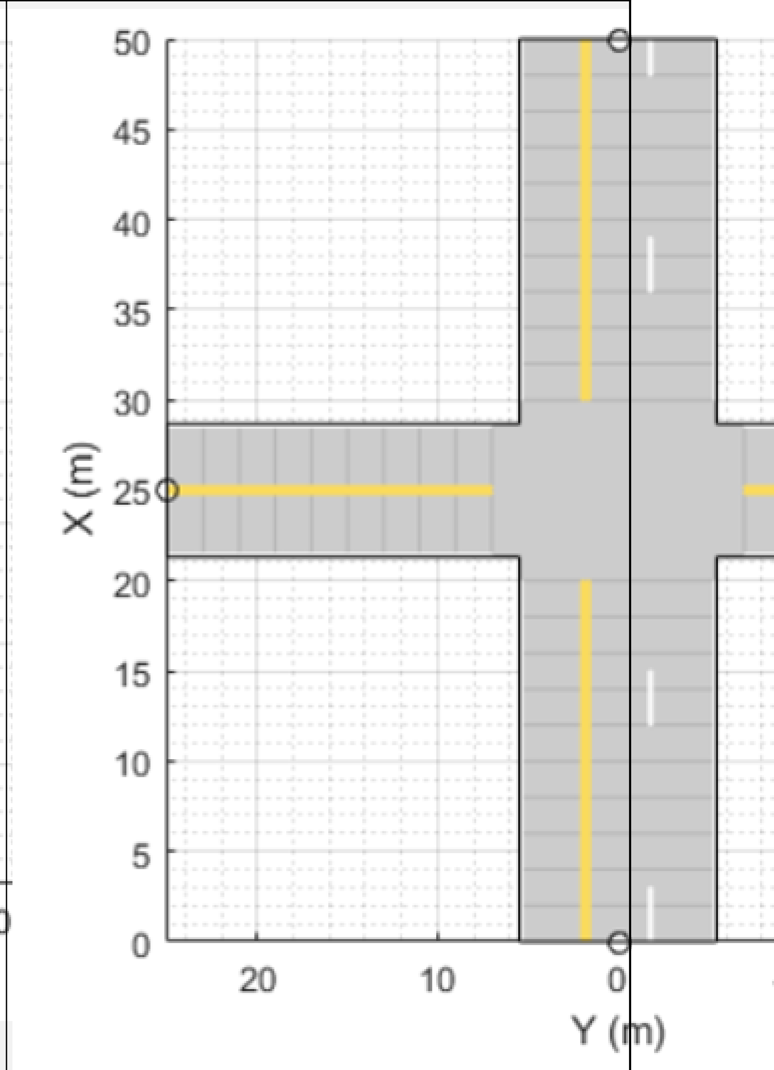
Output Arguments

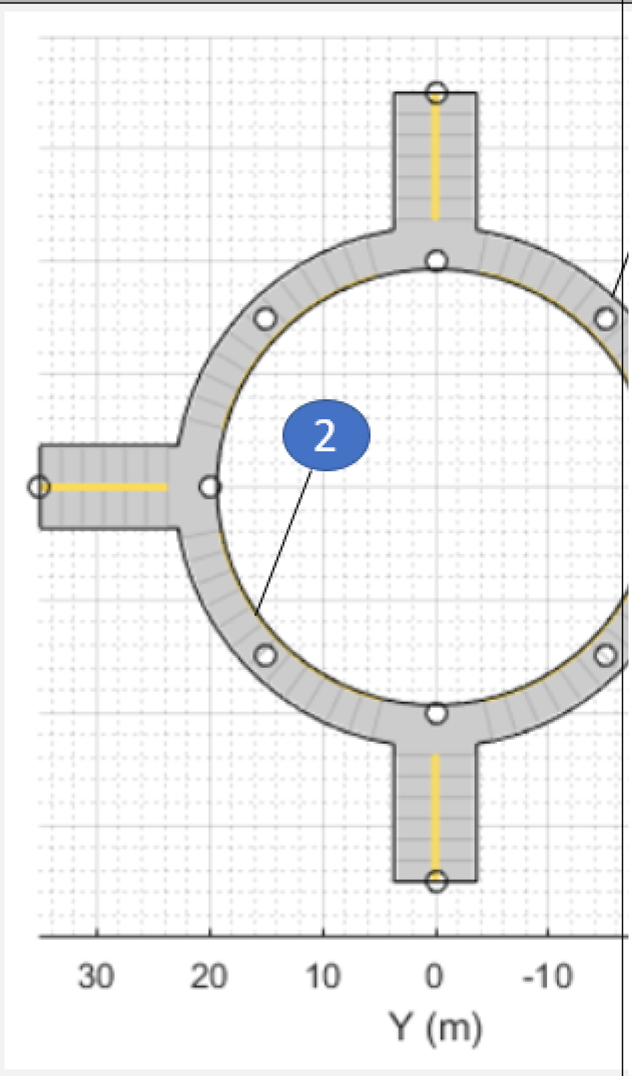
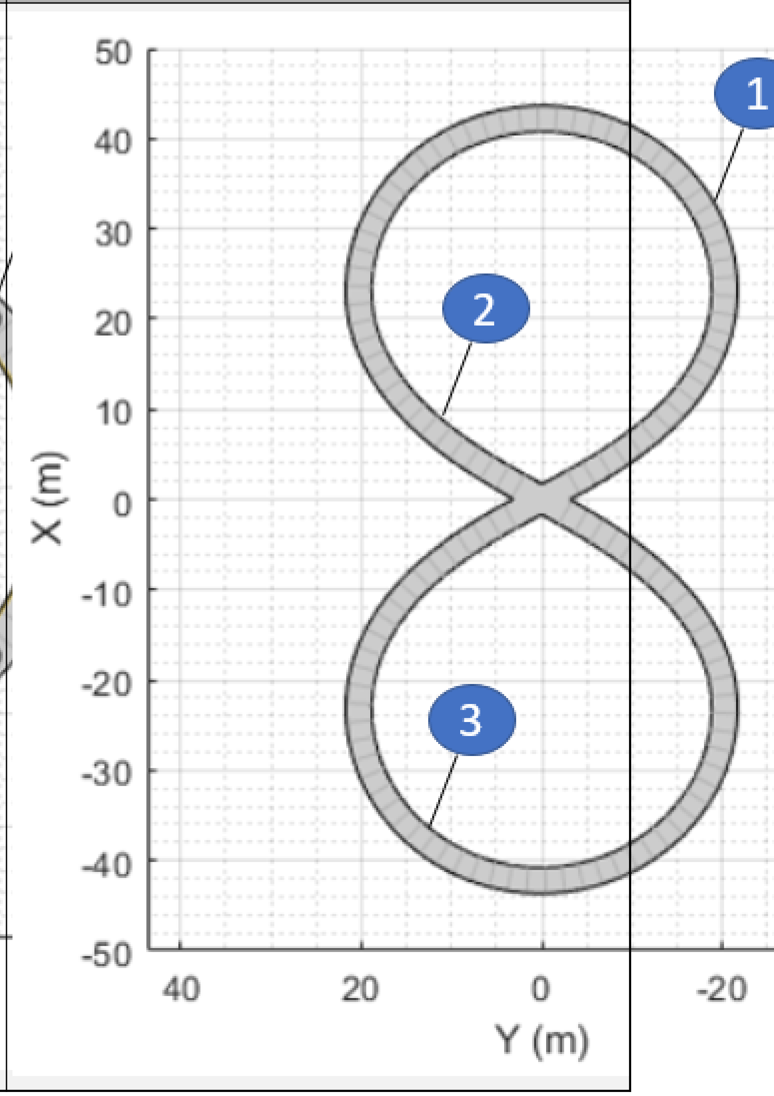
`rbdry` — Road boundaries
cell array

Road boundaries, returned as a cell array. Each cell in the cell array contains a real-valued *N*-by-3 matrix representing a road boundary in the scenario, where *N* is the number of road boundaries. Each row of the matrix corresponds to the (*x*, *y*, *z*) coordinates of a road boundary vertex.

When the input argument is a driving scenario, the road coordinates are with respect to the world coordinates of the driving scenario. When the input argument is an actor, the road coordinates are with respect to the actor coordinate system.

The figures show the number of road boundaries that `rbdry` contains for various road types.

Single Road — One Road Boundary	**Intersection — One Road Boundary**
	

<table>
<tr><th>Roundabout — Two Road Boundaries</th><th>Figure-8 — Three Road Boundaries</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

See Also

Objects
`drivingScenario`

Functions
`actor` | `road` | `vehicle`

Topics
"Create Driving Scenario Programmatically"

Introduced in R2017a

driving.scenario.roadBoundariesToEgo

Convert road boundaries to ego vehicle coordinates

Syntax

```
egoRoadBoundaries = driving.scenario.roadBoundariesToEgo(
scenarioRoadBoundaries,ego)
egoRoadBoundaries = driving.scenario.roadBoundariesToEgo(
scenarioRoadBoundaries,egoPose)
```

Description

`egoRoadBoundaries = driving.scenario.roadBoundariesToEgo(scenarioRoadBoundaries,ego)` converts road boundaries from the world coordinates of a driving scenario to the coordinate system of the ego vehicle, `ego`.

`egoRoadBoundaries = driving.scenario.roadBoundariesToEgo(scenarioRoadBoundaries,egoPose)` converts road boundaries from world coordinates to vehicle coordinates using the pose of the ego vehicle, `egoPose`.

Examples

Create and Display Road Boundaries

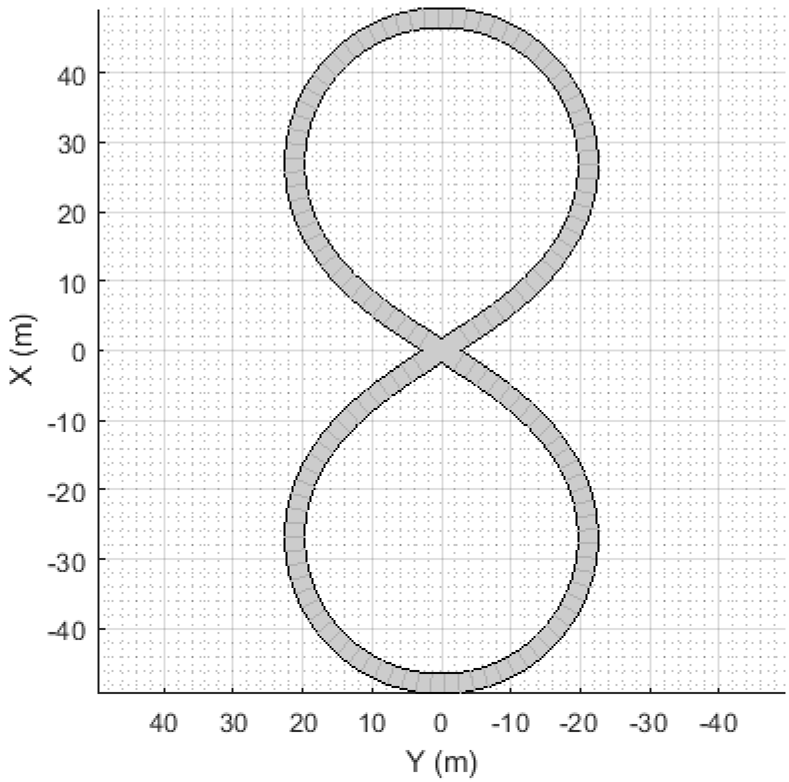
Create a driving scenario containing a figure-8 road specified in the world coordinates of the scenario. Convert the world coordinates of the scenario to the coordinate system of the ego vehicle.

Create an empty driving scenario.

```
scenario = drivingScenario;
```

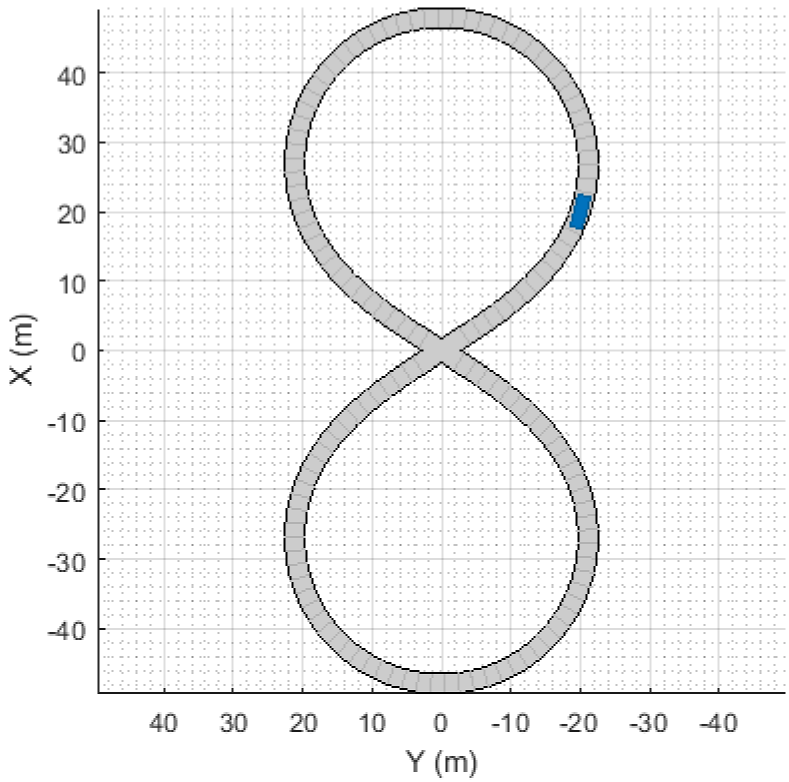
Add a figure-8 road to the scenario. Display the scenario.

```
roadCenters = [0  0  1
             20 -20  1
             20  20  1
            -20 -20  1
            -20  20  1
              0   0  1];

roadWidth = 3;
bankAngle = [0 15 15 -15 -15 0];
road(scenario,roadCenters,roadWidth,bankAngle);
plot(scenario)
```



Add an ego vehicle to the scenario. Position the vehicle at world coordinates (20, –20) and orient it at a –15 degree yaw angle.

```
ego = actor(scenario,'ClassID',1,'Position',[20 -20 0],'Yaw',-15);
```

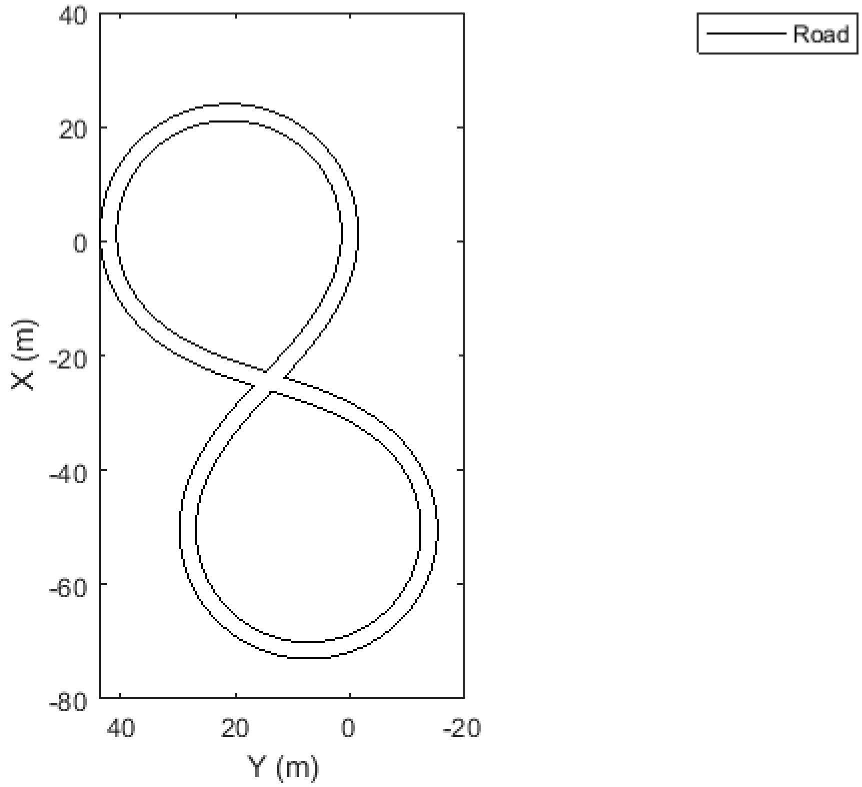


Obtain the road boundaries in ego vehicle coordinates by using the `roadBoundaries` function. Specify the ego vehicle as the input argument.

```
rbEgo1 = roadBoundaries(ego);
```

Display the result on a bird's-eye plot.

```
bep = birdsEyePlot;
lbp = laneBoundaryPlotter(bep,'DisplayName','Road');
plotLaneBoundary(lbp,rbEgo1)
```



Obtain the road boundaries in world coordinates by using the `roadBoundaries` function. Specify the scenario as the input argument.

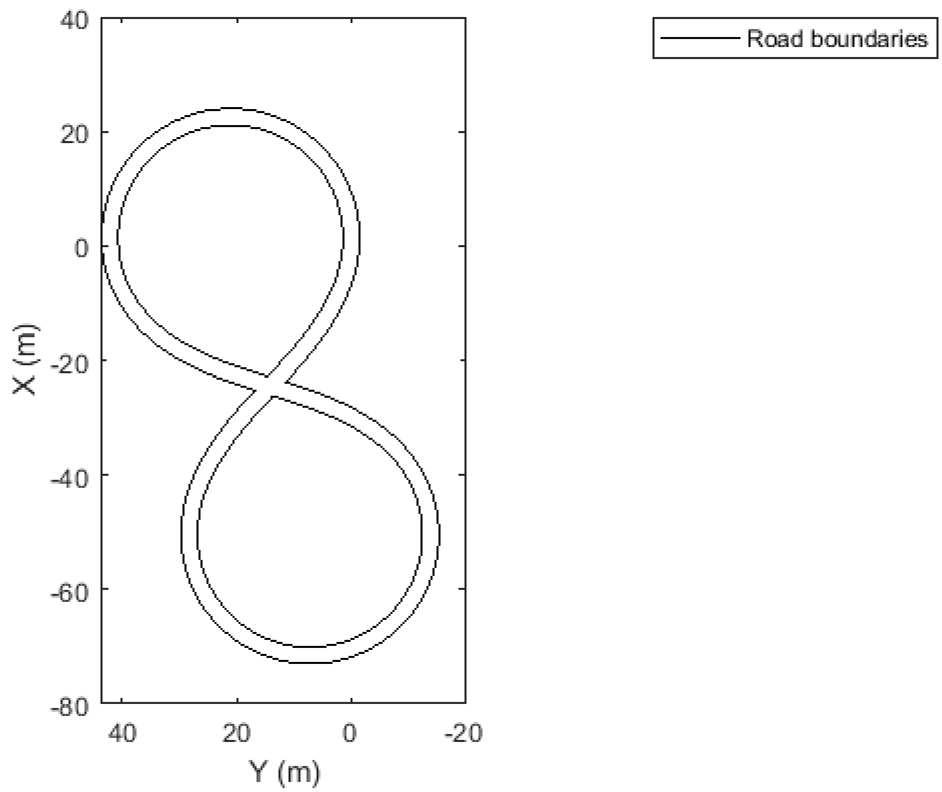
```
rbScenario = roadBoundaries(scenario);
```

Obtain the road boundaries in ego vehicle coordinates by using the `driving.scenario.roadBoundariesToEgo` function.

```
rbEgo2 = driving.scenario.roadBoundariesToEgo(rbScenario,ego);
```

Display the road boundaries on a bird's-eye plot.

```
bep = birdsEyePlot;
lbp = laneBoundaryPlotter(bep,'DisplayName','Road boundaries');
plotLaneBoundary(lbp,{rbEgo2})
```



Input Arguments

scenarioRoadBoundaries — Road boundaries of scenario in world coordinates
1-by-*N* cell array

Road boundaries of the scenario in world coordinates, specified as a 1-by-*N* cell array. *N* is the number of road boundaries within the scenario. Each cell corresponds to a road and contains the (*x*, *y*, *z*) coordinates of the road boundaries in a real-valued *P*-by-3 matrix. *P* is the number of boundaries and varies from cell to cell. Units are in meters.

ego — Ego vehicle
`Actor` object | `Vehicle` object

Ego vehicle, specified as an `Actor` or `Vehicle` object. To create these objects, use the `actor` and `vehicle` functions, respectively.

egoPose — Ego actor pose
structure

Ego actor pose in the world coordinates of a driving scenario, specified as a structure.

The ego actor pose structure must contain at least these fields.

Field	Description
ActorID	Scenario-defined actor identifier, specified as a positive integer.
Position	Position of actor, specified as a real-valued vector of the form [*x*, *y*, *z*]. Units are in meters.
Velocity	Velocity (*v*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [v_x, v_y, v_z]. Units are in meters per second.
Roll	Roll angle of actor, specified as a real-valued scalar. Units are in degrees.
Pitch	Pitch angle of actor, specified as a real-valued scalar. Units are in degrees.
Yaw	Yaw angle of actor, specified as a real-valued scalar. Units are in degrees.
AngularVelocity	Angular velocity (*ω*) of actor in the *x*-, *y*-, and *z*-direction, specified as a real-valued vector of the form [ω_x, ω_y, ω_z]. Units are in degrees per second.

For full definitions of these structure fields, see the `actor` and `vehicle` functions.

Output Arguments

`egoRoadBoundaries` — Road boundaries in ego vehicle coordinates
real-valued *Q*-by-3 matrix

Road boundaries in ego vehicle coordinates, returned as a real-valued *Q*-by-3 matrix. *Q* is the number of road boundary point coordinates of the form (*x*, *y*, *z*).

All road boundaries are contained in the same matrix, with a row of `NaN` values separating points in different road boundaries. For example, if the input has three road boundaries of length P_1, P_2, and P_3, then $Q = P_1 + P_2 + P_3 + 2$. Units are in meters.

See Also

Objects
`drivingScenario`

Functions
`actor` | `actorPoses` | `driving.scenario.targetsToEgo` | `driving.scenario.targetsToScenario` | `road` | `roadBoundaries` | `targetPoses` | `vehicle`

Introduced in R2017a

currentLane

Package:

Get current lane of actor

Syntax

```
cl = currentLane(ac)
[cl,numlanes] = currentLane(ac)
```

Description

`cl = currentLane(ac)` returns the current lane, `cl`, of an actor, `ac`.

`[cl,numlanes] = currentLane(ac)` also returns the number of road lanes, `numlanes`.

Examples

Find Current Lanes of Two Cars

Obtain the current lane boundaries of cars during a driving scenario simulation.

Create a driving scenario containing a straight, three-lane road.

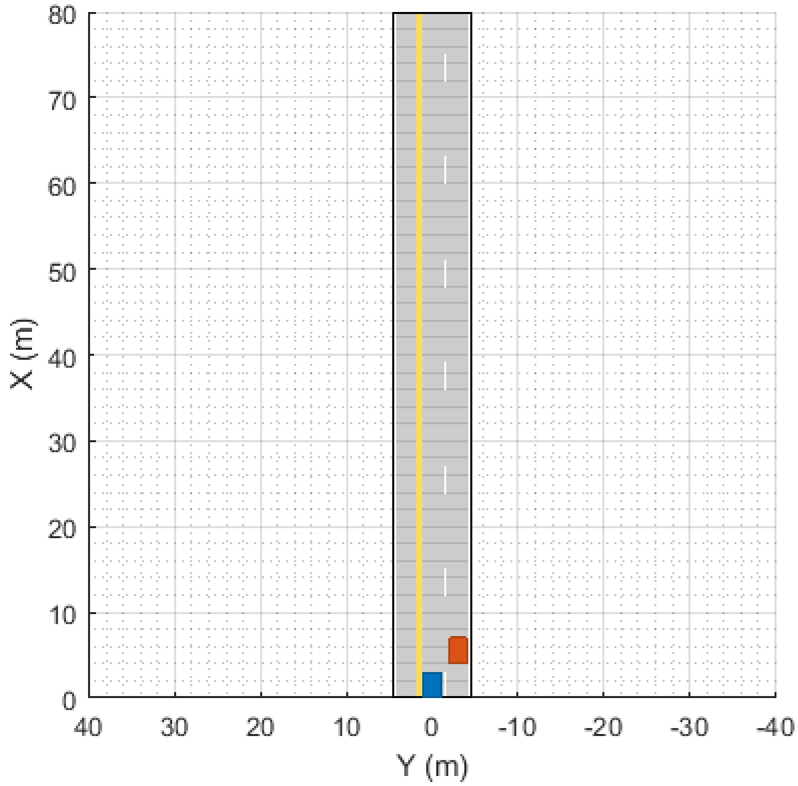
```
scenario = drivingScenario;
roadCenters = [0 0; 80 0];
road(scenario,roadCenters,'Lanes',lanespec([1 2],'Width',3));
```

Add an ego vehicle moving at 20 meters per second and a target vehicle moving at 10 meters per second.

```
ego = vehicle(scenario,'ClassID',1,'Position',[5 0 0], ...
    'Length',3,'Width',2,'Height',1.6);
trajectory(ego,[1 0 0; 20 0 0; 30 0 0;50 0 0],20);

target = vehicle(scenario,'ClassID',1,'Position',[5 0 0], ...
    'Length',3,'Width',2,'Height',1.6);
trajectory(target,[5 -3 0; 20 -3 0; 30 -3 0;50 -3 0],10);
```

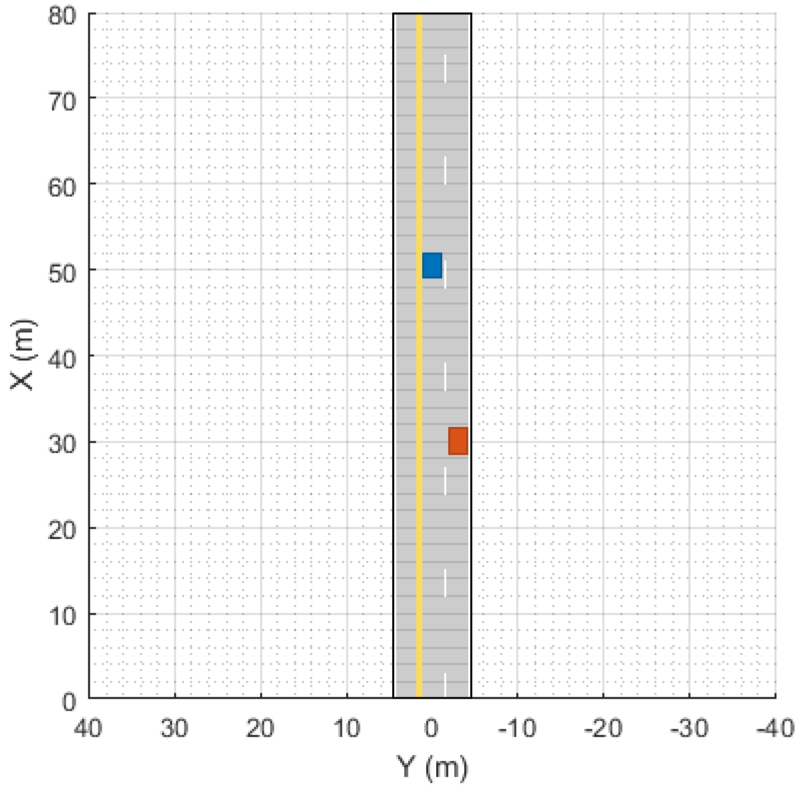
Plot the scenario.

```
plot(scenario)
```



Run the simulation loop.

```
while advance(scenario)
    [cl1,numlanes] = currentLane(ego);
    [cl2,numlanes] = currentLane(target);
end
```



Display the current lane of each vehicle.

```
disp(cl1)
disp(cl2)
```

```
2

3
```

Input Arguments

ac — Actor
`Actor` object | `Vehicle` object

Actor belonging to a `drivingScenario` object, specified as an `Actor` or `Vehicle` object. To create these objects, use the `actor` and `vehicle` functions, respectively.

Output Arguments

`cl` — Current lane of actor
positive integer | `[]`

Current lane of the actor, returned as a positive integer. Lanes are numbered from left to right, relative to the actor, starting from 1. When the actor is not on a road or is on a road without any lanes specified, `cl` is returned as empty, `[]`.

`numlanes` — Number of lanes on road
positive integer | `[]`

Number of lanes on the road that the actor is traveling on, returned as a positive integer. When the actor is not on a road or is on a road without any lanes specified, `numlanes` is returned as empty, `[]`.

See Also

Objects
`drivingScenario` | `lanespec`

Functions
`actor` | `laneBoundaries` | `vehicle`

Introduced in R2018a

lanespec

Create road lane specifications

Description

The `lanespec` object defines the lane specifications of a road that was added to a `drivingScenario` object using the `road` function. For more details, see "Lane Specifications" on page 4-481.

Creation

Syntax

```
lnspec = lanespec(numlanes)
lnspec = lanespec(numlanes,Name,Value)
```

Description

`lnspec = lanespec(numlanes)` creates lane specifications for a road having `numlanes` lanes. `numLanes` sets the `NumLanes` property of the `lanespec` object. The order for numbering the lanes on a road depend on the orientation of the road. For more details, see "Draw Direction of Road and Numbering of Lanes" on page 4-478.

`lnspec = lanespec(numlanes,Name,Value)` sets properties on page 4-469 using one or more name-value pairs. For example, `lanespec(3,'Width',[2.25 3.5 2.25])` specifies a three-lane road with widths from left to right of 2.25 meters, 3.5 meters, and 2.25 meters. For more information on the geometrical properties of a lane, see "Lane Specifications" on page 4-481.

Properties

`NumLanes` — Number of lanes in road
positive integer | two-element vector of positive integers

This property is read-only.

Number of lanes in the road, specified as a positive integer or two-element vector of positive integers, $[N_L, N_R]$. When `NumLanes` is a positive integer, all lanes flow in the same direction. When `NumLanes` is a vector:

- N_L is the number of left lanes, all flowing in one direction.
- N_R is the number of right lanes, all flowing in the opposite direction.

The total number of lanes in the road is the sum of these vector values: $N = N_L + N_R$.

You can set this property when you create the object. After you create the object, this property is read-only.

Example: `[2 2]` specifies two left lanes and two right lanes.

Width — Lane widths
3.6 (default) | positive real scalar | 1-by-*N* vector of positive real scalars

Lane widths, specified as a positive real scalar or 1-by-*N* vector of positive real scalars, where *N* is the number of lanes in the road. *N* must be equal to numlanes and the corresponding value set in the NumLanes property.

When Width is a scalar, the same value is applied to all lanes. When Width is a vector, the vector elements apply to lanes from left to right. Units are in meters.

Example: [3.5 3.7 3.7 3.5]

Data Types: double

Marking — Lane markings
LaneMarking object (default) | SolidMarking object | DashedMarking object | CompoundMarking object | 1-by-*M* array of lane marking objects

Lane markings of road, specified as one of these values:

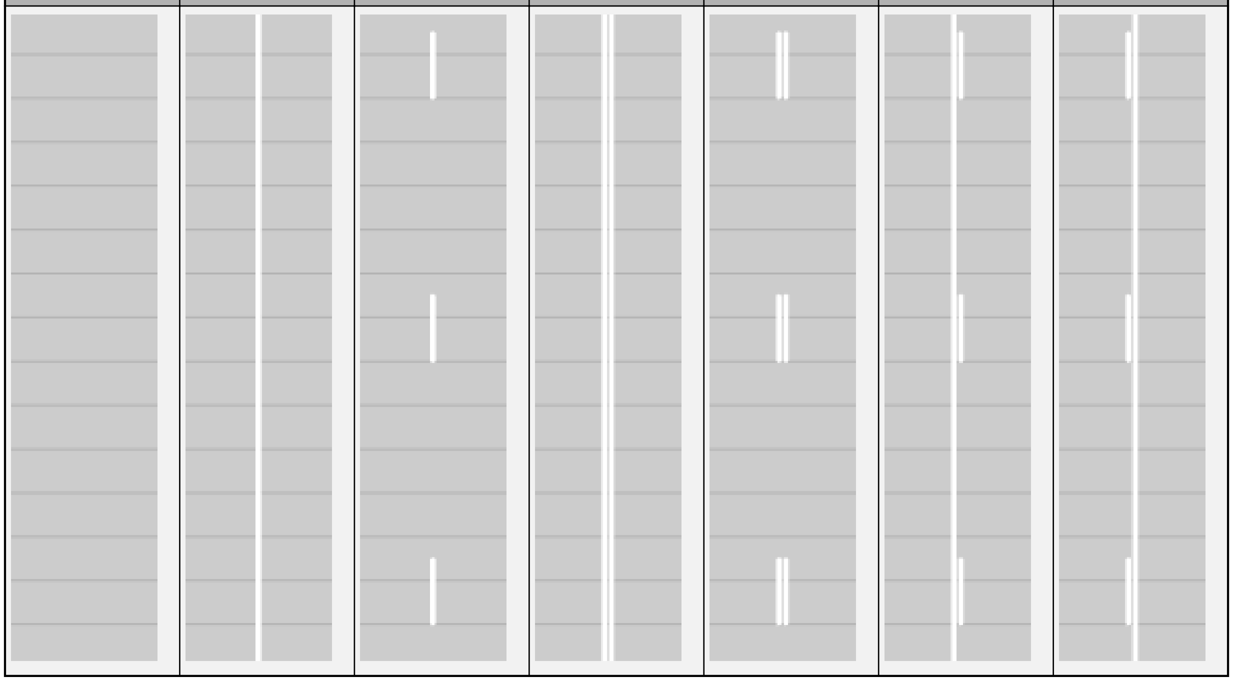
- LaneMarking object. This is the default.
- SolidMarking object
- DashedMarking object
- CompoundMarking object
- 1-by-*M* array of lane marking objects.

M is the number of lane markings. For a road with *N* lanes, *M* = *N* + 1.

To create lane marking objects, use the laneMarking function and specify the type of lane marking.

'Unmarked'	'Solid'	'Dashed'	'DoubleSolid'	'DoubleDashed'	'SolidDashed'	'DashedSolid'
No lane marking	Solid line	Dashed line	Two solid lines	Two dashed lines	Solid line on left, dashed line on right	Dashed line on left, solid line on right

'Unmarked'	'Solid'	'Dashed'	'DoubleSolid'	'DoubleDashed'	'SolidDashed'	'DashedSolid'



By default, for a one-way road, the rightmost and center lane markings are white and the leftmost lane marking is yellow. For two-way roads, the color of the dividing lane marking is yellow.

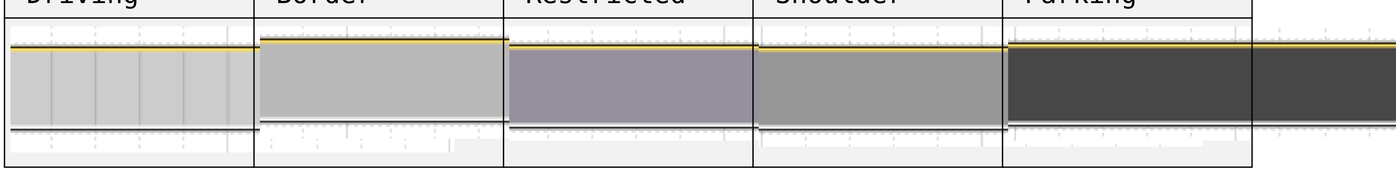
Example: `[laneMarking('Solid') laneMarking('DoubleDashed') laneMarking('Solid')]` specifies lane markings for a two-lane road. The leftmost and rightmost lane markings are solid lines, and the dividing lane marking is a double-dashed line.

Type — Lane types
`DrivingLaneType` object (default) | `RestrictedLaneType` object | `ShoulderLaneType` object | `ParkingLaneType` object | 1-by-*M* array of lane type objects

Lane types of road, specified as a homogeneous lane type object or a 1-by-*M* array of lane type objects. *M* is the number of lane types.

To create lane type objects, use the `laneType` function and specify the type of lane.

'Driving'	'Border'	'Restricted'	'Shoulder'	'Parking'



Example: `[laneType('Shoulder') laneType('Driving')]` specifies the lane types for a two-lane road. The leftmost lane is the shoulder lane and the rightmost lane is the driving lane.

Examples

Create Straight Four-Lane Road

Create a driving scenario and the road centers for a straight, 80-meter road.

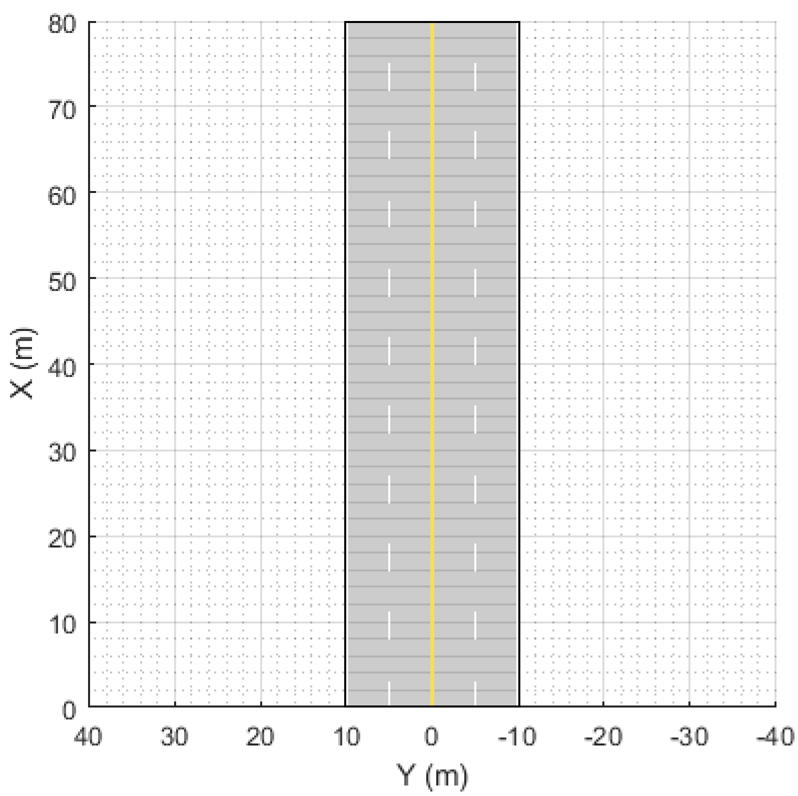
```
scenario = drivingScenario;
roadCenters = [0 0; 80 0];
```

Create a `lanespec` object for a four-lane road. Use the `laneMarking` function to specify its five lane markings. The center line is double-solid and double yellow. The outermost lines are solid and white. The inner lines are dashed and white.

```
solidW = laneMarking('Solid','Width',0.3);
dashW = laneMarking('Dashed','Space',5);
doubleY = laneMarking('DoubleSolid','Color','yellow');
lspec = lanespec([2 2],'Width',[5 5 5 5], ...
    'Marking',[solidW dashW doubleY dashW solidW]);
```

Add the road to the driving scenario. Display the road.

```
road(scenario,roadCenters,'Lanes',lspec);
plot(scenario)
```



Simulate Car Traveling on S-Curve

Simulate a driving scenario with one car traveling on an S-curve. Create and plot the lane boundaries.

Create the driving scenario with one road having an S-curve.

```
scenario = drivingScenario('StopTime',3);
roadcenters = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
```

Create the lanes and add them to the road.

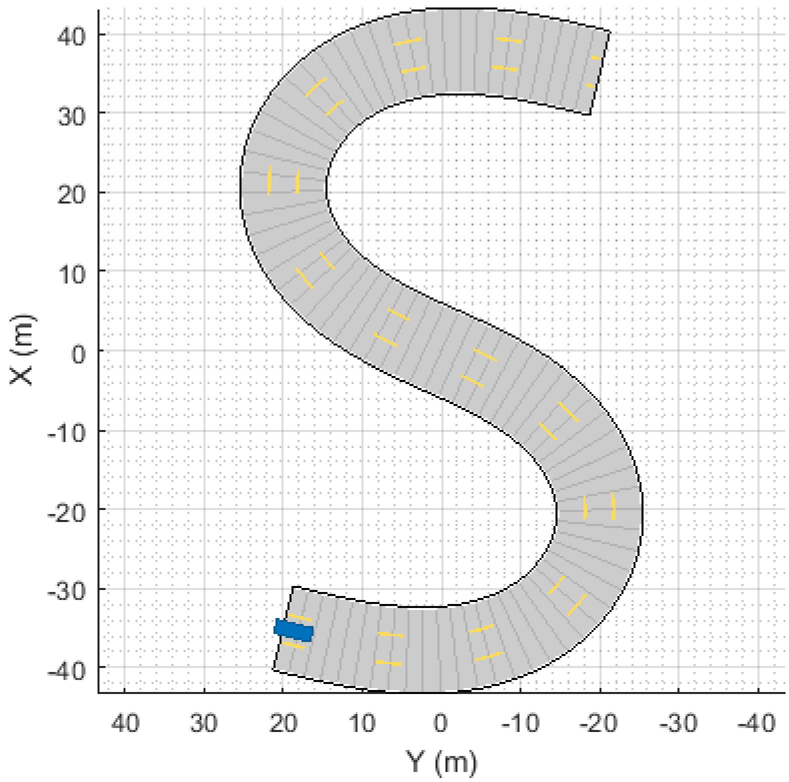
```
lm = [laneMarking('Solid','Color','w'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Solid','Color','w')];
ls = lanespec(3,'Marking',lm);
road(scenario,roadcenters,'Lanes',ls);
```

Add an ego vehicle and specify its trajectory from its waypoints. By default, the car travels at a speed of 30 meters per second.

```
car = vehicle(scenario, ...
    'ClassID',1, ...
    'Position',[-35 20 0]);
waypoints = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
trajectory(car,waypoints);
```

Plot the scenario and corresponding chase plot.

```
plot(scenario)
```

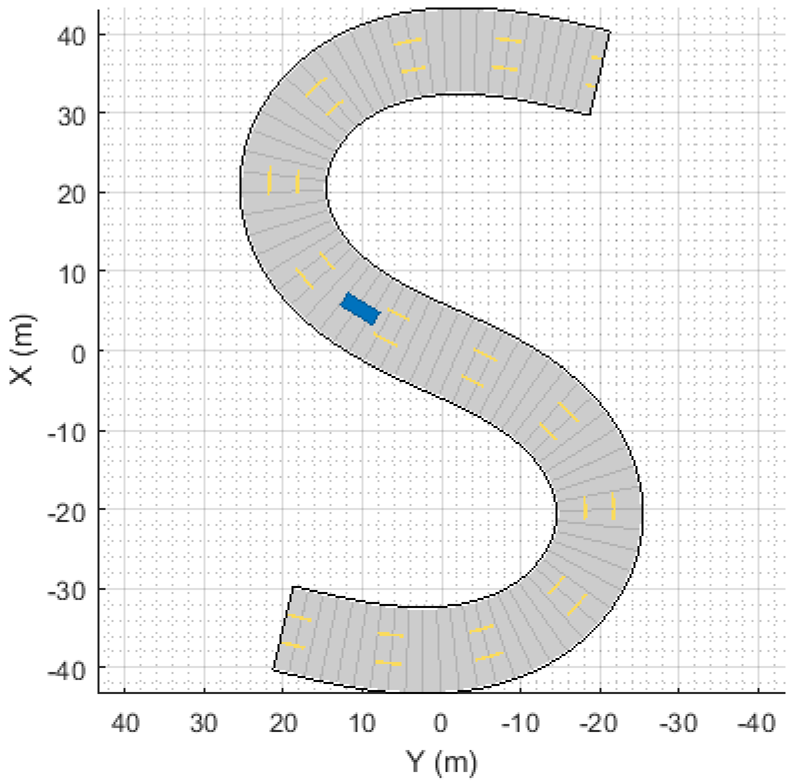


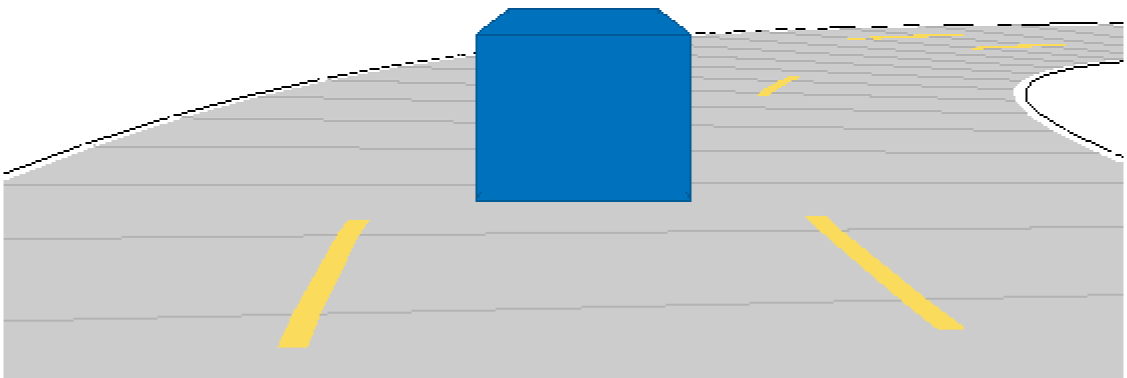

```
chasePlot(car)
```

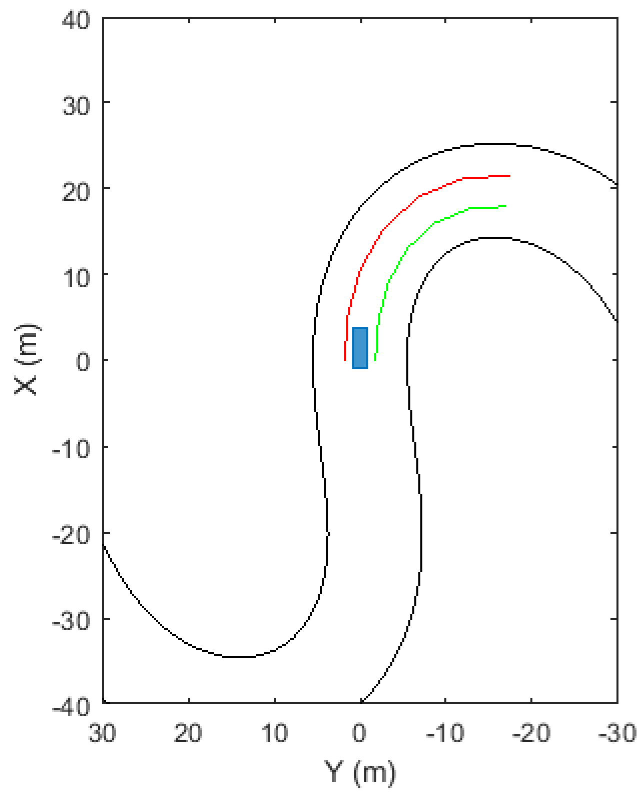
Run the simulation loop.

1. Initialize a bird's-eye plot and create an outline plotter, left-lane and right-lane boundary plotters, and a road boundary plotter.
2. Obtain the road boundaries and rectangular outlines.
3. Obtain the lane boundaries to the left and right of the vehicle.
4. Advance the simulation and update the plotters.

```
bep = birdsEyePlot('XLim',[-40 40],'YLim',[-30 30]);
olPlotter = outlinePlotter(bep);
lblPlotter = laneBoundaryPlotter(bep,'Color','r','LineStyle','-');
lbrPlotter = laneBoundaryPlotter(bep,'Color','g','LineStyle','-');
rbsEdgePlotter = laneBoundaryPlotter(bep);
legend('off');
while advance(scenario)
    rbs = roadBoundaries(car);
    [position,yaw,length,width,originOffset,color] = targetOutlines(car);
    lb = laneBoundaries(car,'XDistance',0:5:30,'LocationType','Center', ...
        'AllBoundaries',false);
    plotLaneBoundary(rbsEdgePlotter,rbs)
    plotLaneBoundary(lblPlotter,{lb(1).Coordinates})
    plotLaneBoundary(lbrPlotter,{lb(2).Coordinates})
    plotOutline(olPlotter,position,yaw,length,width, ...
        'OriginOffset',originOffset,'Color',color)
end
```


X (m)
Y (m)
40
30
20
10
0
-10
-20
-30
-40
40
30
20
10
0
-10
-20
-30
-40





Limitations

- Lane markings in intersections are not supported.
- The number of lanes for a road is fixed. You cannot change lane specifications for a road during a simulation.
- A road can have only one lane specification.

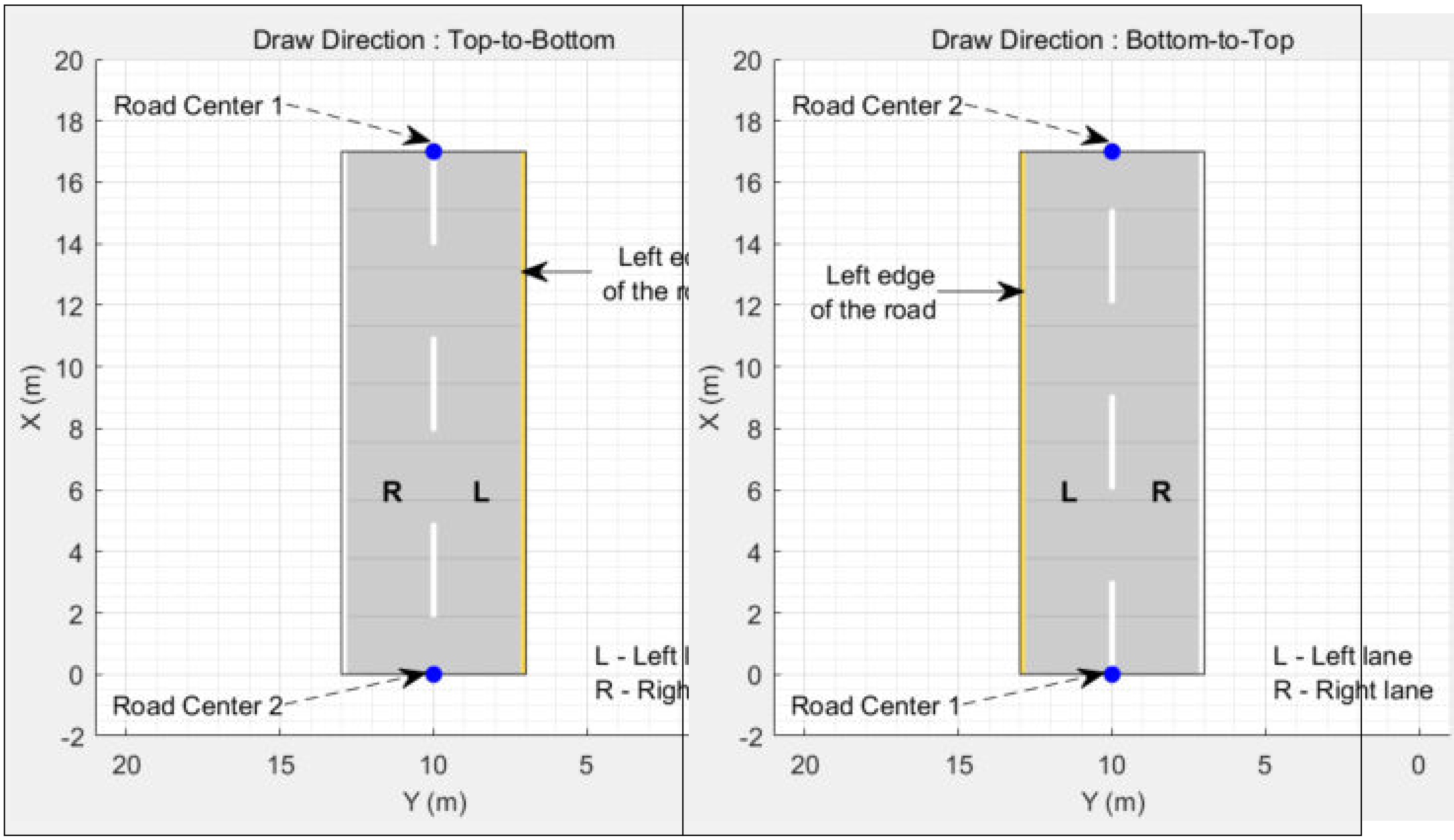
More About

Draw Direction of Road and Numbering of Lanes

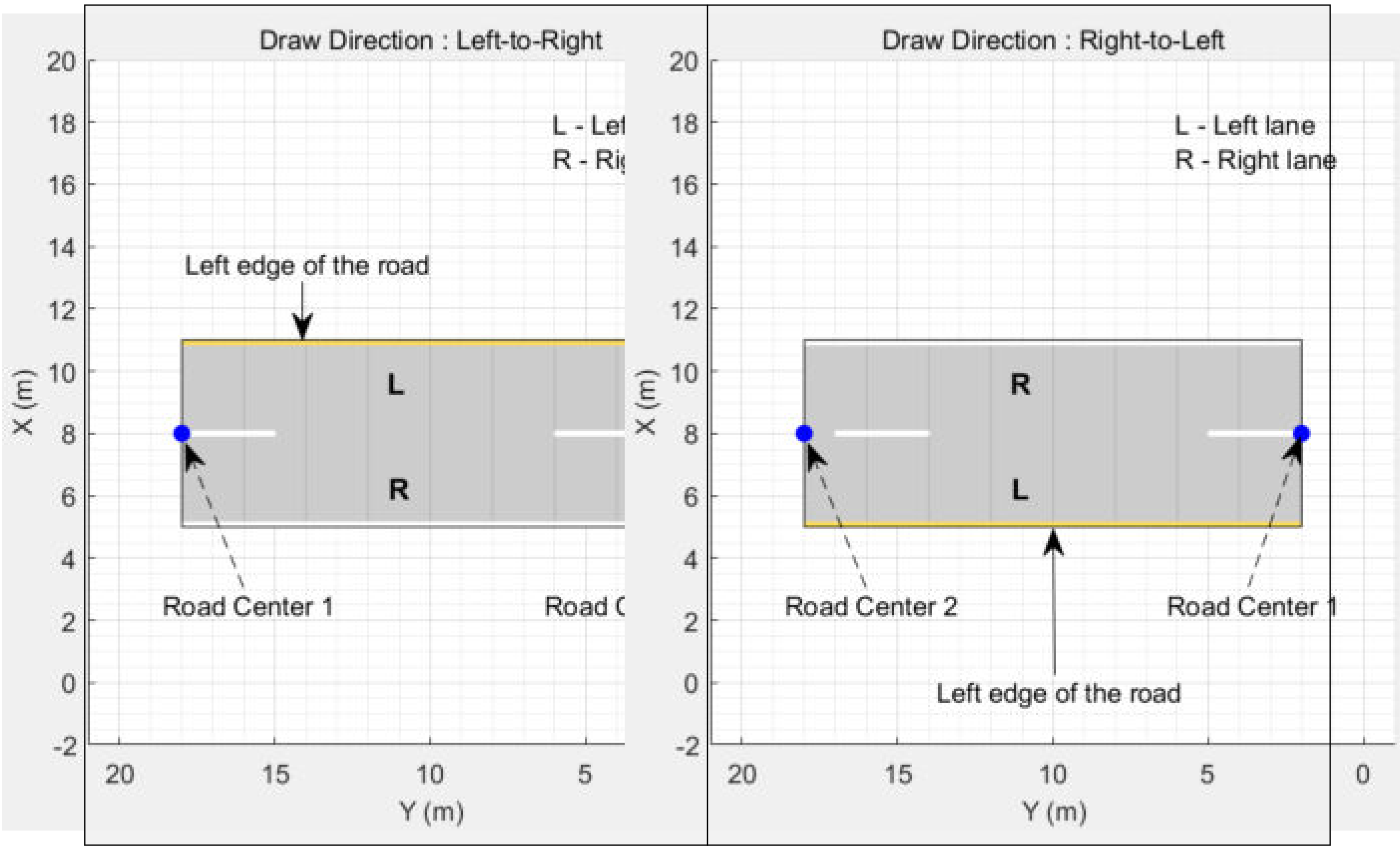
To create a road by using the `road` function, specify the road centers as a matrix input. The function creates a directed line that traverses the road centers, starting from the coordinates in the first row of the matrix and ending at the coordinates in the last row of the matrix. The coordinates in the first two rows of the matrix specify the draw direction of the road. These coordinates correspond to the first two consecutive road centers. The draw direction is the direction in which the roads render in the scenario plot.

To create a road by using the **Driving Scenario Designer** app, you can either specify the **Road Centers** parameter or interactively draw on the **Scenario Canvas**. For a detailed example, see “Create a Driving Scenario” on page 1-14. In this case, the draw direction is the direction in which roads render in the **Scenario Canvas**.

- For a road with a top-to-bottom draw direction, the difference between the *x*-coordinates of the first two consecutive road centers is positive.
- For a road with a bottom-to-top draw direction, the difference between the *x*-coordinates of the first two consecutive road centers is negative.



- For a road with a left-to-right draw direction, the difference between the *y*-coordinates of the first two consecutive road centers is positive.
- For a road with a right-to-left draw direction, the difference between the *y*-coordinates of the first two consecutive road centers is negative.

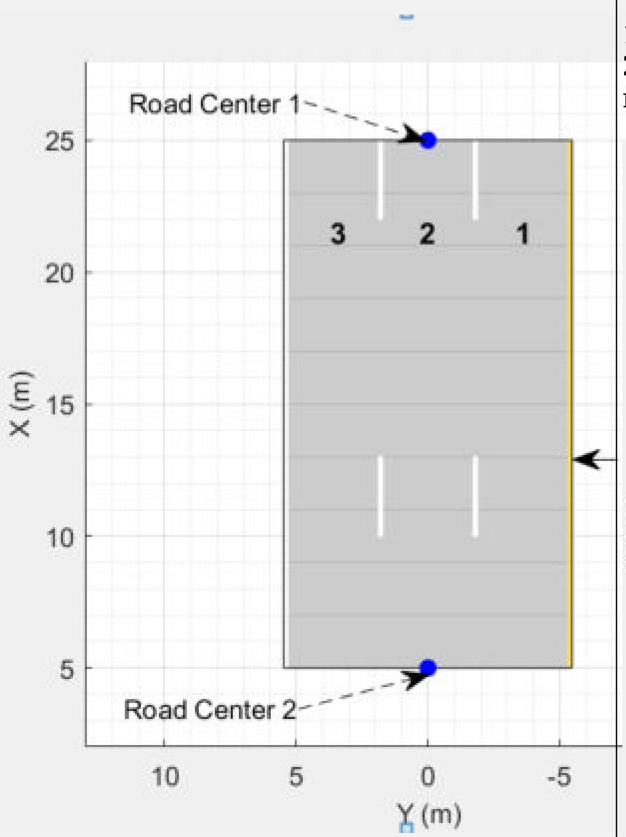
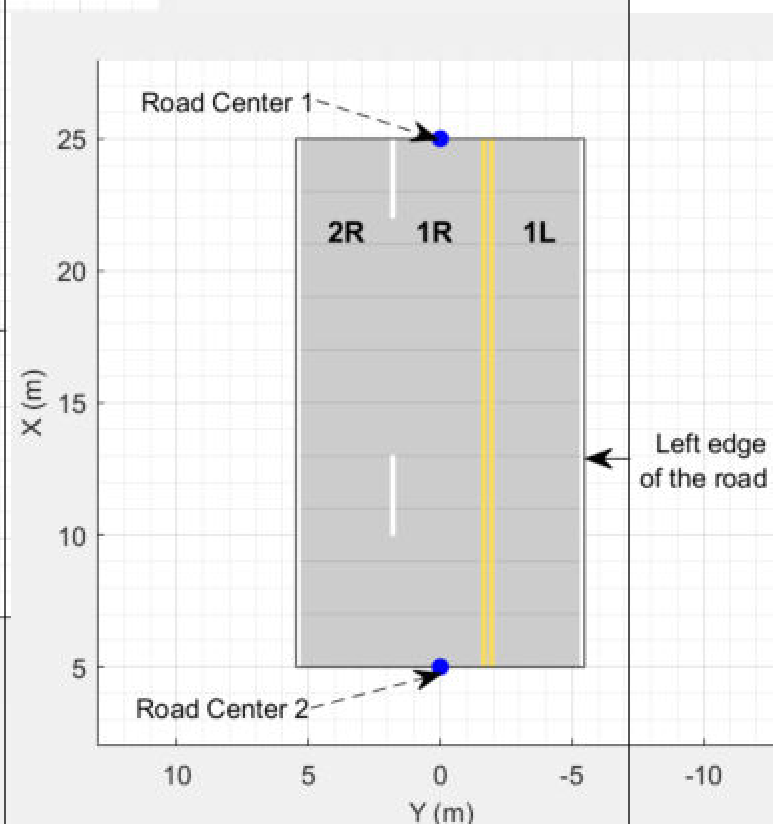


Numbering Lanes

Lanes must be numbered from left to right, with the left edge of the road defined relative to the draw direction of the road. For a one-way road, by default, the left edge of the road is a solid yellow marking which indicates the end of the road in transverse direction (direction perpendicular to draw direction). For a two-way road, by default, both edges are marked with solid white lines.

For example, these diagrams show how the lanes are numbered in a one-way and two-way road with a draw direction from top-to-bottom.

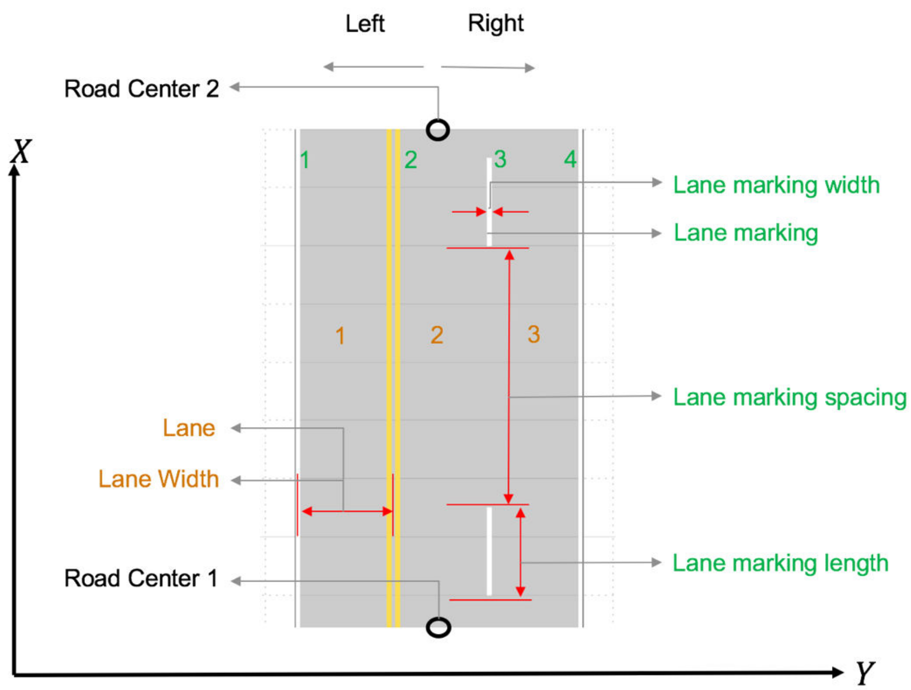
Numbering Lanes in a One-Way Road	Numbering Lanes in a Two-Way Road

Specify the number of lanes as a positive integer for a one-way road. If you set the integer value as 3, then the road has three lanes that travel in the same direction. The lanes are numbered starting from the left edge of the road. **1**, **2**, **3** denote the first, second, and third lanes of the road, respectively.  	Specify the number of lanes as a two-element vector of positive integer for a two-way road. If you set the vector as [1 2], then the road has three lanes: two lanes traveling in one direction and one lane traveling in the opposite direction. Because of the draw direction, the road has one left lane and two right lanes. The lanes are numbered starting from the left edge of the road. **1L** denote the only left lane of the road. **1R** and **2R** denote the first and second right lanes of the road, respectively. 

The lane specifications apply by the order in which the lanes are numbered.

Lane Specifications

The diagram shows the components and geometric properties of roads, lanes, and lane markings.



The lane specification object, `lanespec`, defines the road lanes.

- The `NumLanes` property specifies the number of lanes. You must specify the number of lanes when you create this object.
- The `Width` property specifies the width of each lane.
- The `Marking` property contains the specifications of each lane marking in the road. `Marking` is an array of lane marking objects, with one object per lane. To create these objects, use the `laneMarking` function. Lane marking specifications include:
  - `Type` — Type of lane marking (solid, dashed, and so on)
  - `Width` — Lane marking width
  - `Color` — Lane marking color
  - `Strength` — Saturation value for lane marking color
  - `Length` — For dashed lanes, the length of each dashed line
  - `Space` — For dashed lanes, the spacing between dashes
  - `SegmentRange` — For composite lane marking, the normalized length of each marker segment.
- The `Type` property contains the lane type specifications of each lane in the road. `Type` can be a homogeneous lane type object or a heterogeneous lane type array.
  - Homogeneous lane type object contain lane type specifications of all the lanes in the road.
  - Heterogeneous lane type array contain an array of lane type objects, with one object per lane.

  To create these objects, use the `laneType` function. Lane type specifications include:

  - `Type` — Type of lane (driving, border, and so on)

- `Color` — Lane color
- `Strength` — Strength of the lane color

See Also

`drivingScenario` | `laneBoundaryPlotter` | `laneMarking` | `laneMarkingPlotter` | `laneMarkingVertices` | `laneType` | `plotLaneBoundary` | `plotLaneMarking` | `road`

Introduced in R2018a

laneMarking

Create road lane marking object

Syntax

```
lm = laneMarking(type)
lm = laneMarking(type,Name,Value)
```

```
cm = laneMarking(lmArray)
cm = laneMarking(lmArray,'SegmentRange',range)
```

Description

Single Marking Type Along Lane

`lm = laneMarking(type)` creates a default lane marking object of the specified type (solid lane, dashed lane, and so on). This object defines the characteristics of a lane boundary marking on a road. When creating roads in a driving scenario, you can use lane marking objects as inputs to the `lanespec` object.

`lm = laneMarking(type,Name,Value)` set the properties of the lane marking object using one or more name-value pairs. For example, `laneMarking('Solid','Color','yellow')` creates a solid yellow lane marking.

Multiple Marking Types Along Lane

`cm = laneMarking(lmArray)` creates a composite lane marking object from an array of lane marking objects, `lmArray`. Use this syntax to generate lane markings that contain multiple marker types.

For example, create a lane boundary marking that has both solid and dashed marking types by defining `lmArray`.

```
lmArray = [laneMarking('Solid') laneMarking('Dashed')]
cm = laneMarking(lmArray)
```

The order in which the marking types occur depend on the draw direction of the road. For more information, see Draw Direction of Road and Numbering of Lanes on page 4-498 and Composite Lane Marking on page 4-501.

`cm = laneMarking(lmArray,'SegmentRange',range)` specifies the range for each marker type in the composite lane marking by using the name-value pair argument `'SegmentRange'`, `range`.

Examples

Create Straight Four-Lane Road

Create a driving scenario and the road centers for a straight, 80-meter road.

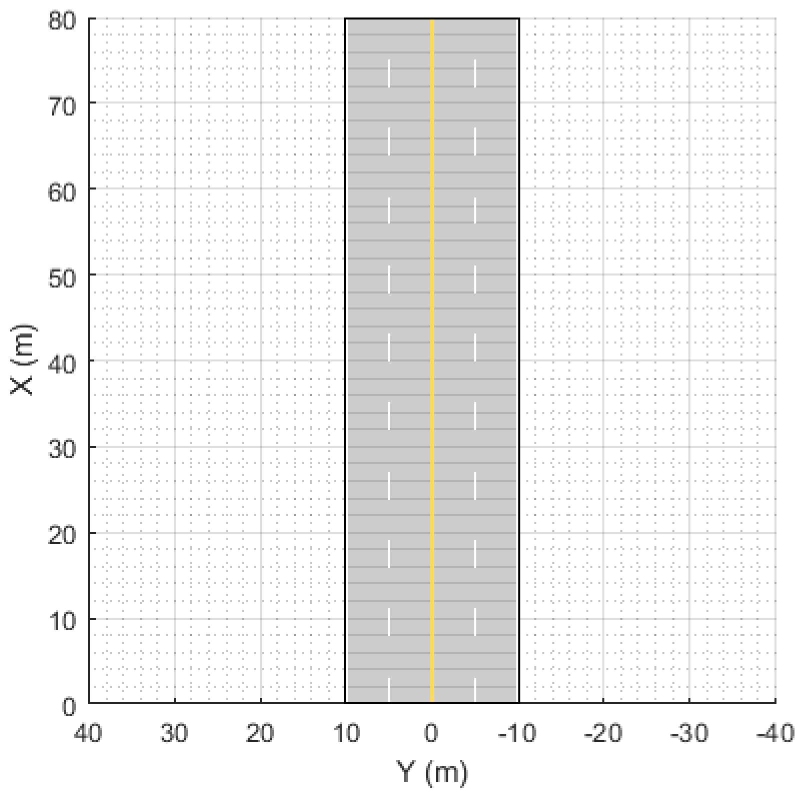
```
scenario = drivingScenario;
roadCenters = [0 0; 80 0];
```

Create a `lanespec` object for a four-lane road. Use the `laneMarking` function to specify its five lane markings. The center line is double-solid and double yellow. The outermost lines are solid and white. The inner lines are dashed and white.

```
solidW = laneMarking('Solid','Width',0.3);
dashW = laneMarking('Dashed','Space',5);
doubleY = laneMarking('DoubleSolid','Color','yellow');
lspec = lanespec([2 2],'Width',[5 5 5 5], ...
    'Marking',[solidW dashW doubleY dashW solidW]);
```

Add the road to the driving scenario. Display the road.

```
road(scenario,roadCenters,'Lanes',lspec);
plot(scenario)
```



Simulate Car Traveling on S-Curve

Simulate a driving scenario with one car traveling on an S-curve. Create and plot the lane boundaries.

Create the driving scenario with one road having an S-curve.

```
scenario = drivingScenario('StopTime',3);
roadcenters = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
```

Create the lanes and add them to the road.

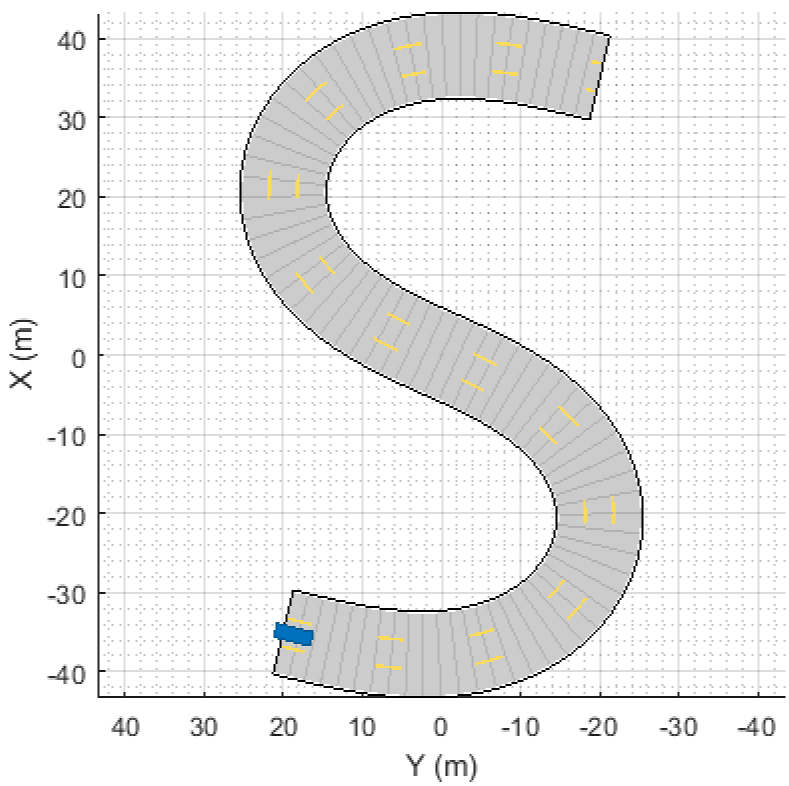
```
lm = [laneMarking('Solid','Color','w'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Solid','Color','w')];
ls = lanespec(3,'Marking',lm);
road(scenario,roadcenters,'Lanes',ls);
```

Add an ego vehicle and specify its trajectory from its waypoints. By default, the car travels at a speed of 30 meters per second.

```
car = vehicle(scenario, ...
    'ClassID',1, ...
    'Position',[-35 20 0]);
waypoints = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
trajectory(car,waypoints);
```

Plot the scenario and corresponding chase plot.

```
plot(scenario)
```

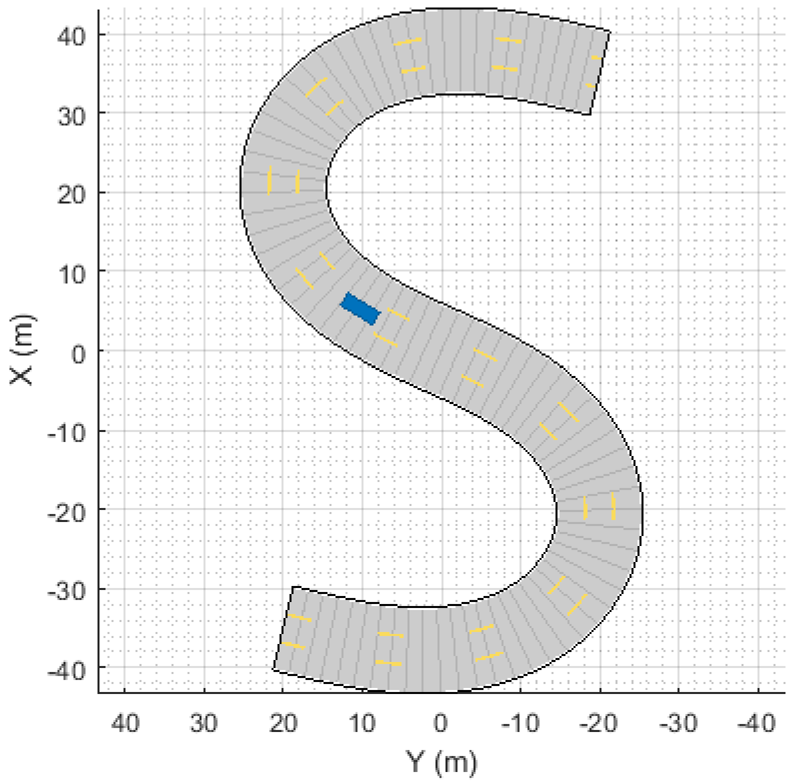


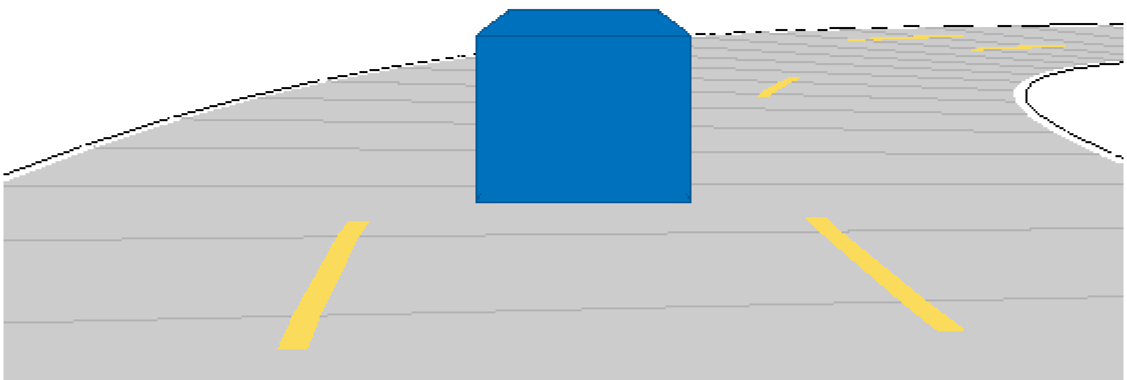

```
chasePlot(car)
```

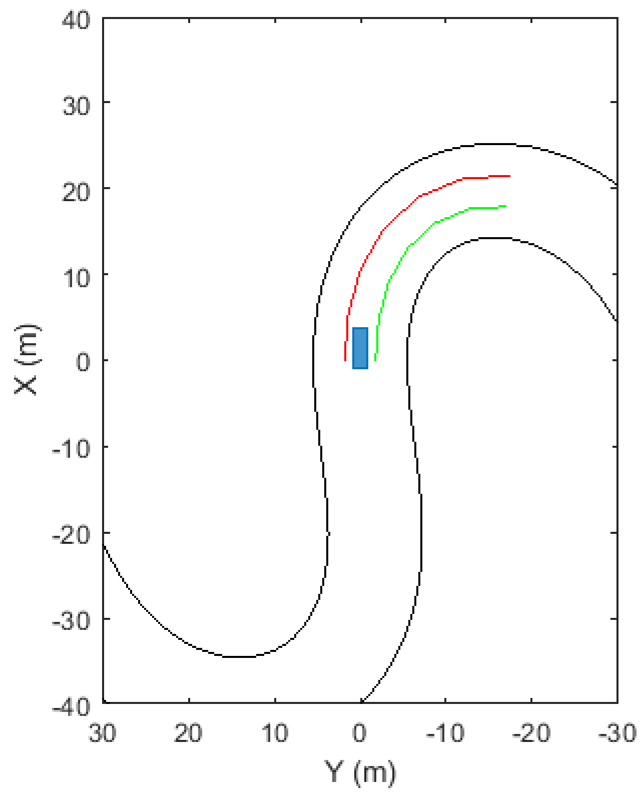
Run the simulation loop.

1 Initialize a bird's-eye plot and create an outline plotter, left-lane and right-lane boundary plotters, and a road boundary plotter.
2 Obtain the road boundaries and rectangular outlines.
3 Obtain the lane boundaries to the left and right of the vehicle.
4 Advance the simulation and update the plotters.

```
bep = birdsEyePlot('XLim',[-40 40],'YLim',[-30 30]);
olPlotter = outlinePlotter(bep);
lblPlotter = laneBoundaryPlotter(bep,'Color','r','LineStyle','-');
lbrPlotter = laneBoundaryPlotter(bep,'Color','g','LineStyle','-');
rbsEdgePlotter = laneBoundaryPlotter(bep);
legend('off');
while advance(scenario)
    rbs = roadBoundaries(car);
    [position,yaw,length,width,originOffset,color] = targetOutlines(car);
    lb = laneBoundaries(car,'XDistance',0:5:30,'LocationType','Center', ...
        'AllBoundaries',false);
    plotLaneBoundary(rbsEdgePlotter,rbs)
    plotLaneBoundary(lblPlotter,{lb(1).Coordinates})
    plotLaneBoundary(lbrPlotter,{lb(2).Coordinates})
    plotOutline(olPlotter,position,yaw,length,width, ...
        'OriginOffset',originOffset,'Color',color)
end
```


40
30
20
10
0
-10
-20
-30
-40
X (m)
40
30
20
10
0
-10
-20
-30
-40
Y (m)





Driving Scenario for Changing Lanes and Passing Vehicles

This example shows how to create a driving scenario for maneuvers such as changing lanes and passing other vehicles. You create roads with passing zones and add vehicles to the scenario. Then, define the trajectories for these vehicles to simulate vehicle lane change in passing zones.

Create Road with Passing Zones by Using Composite Lane Marking

Create a driving scenario. Specify the road centers and the number of lanes to add a two-way, two-lane straight road of 54 meters with draw direction from top-to-bottom.

```
scenario = drivingScenario('StopTime',10);
roadCenters = [50 0; -4 0];
numLanes = [1 1];
```

Typically, the number of lane markings is equal to number of lanes plus one. A two-way, two-lane road has 3 lane markings and the outermost lane markings at both the edges are solid white lines.

Create a solid marking object of marking width 0.25 meters, to constitute the outermost lane markings for the two-way road.

```
outerLM = laneMarking('Solid','Width',0.25);
```

Create a lane marking array of `SolidMarking` and `DashedMarking` objects that contain the properties for solid and dashed double yellow lines.

```
lmArray = [laneMarking('DoubleSolid','Color','Yellow','Width',0.25)
           laneMarking('DashedSolid','Color','Yellow','Length',1,'Space',1.5,'Width',0.25)
           laneMarking('DoubleSolid','Color','Yellow','Width',0.25)
           laneMarking('SolidDashed','Color','Yellow','Length',1,'Space',1.5,'Width',0.25)];
```

Create a composite lane marking object for the center lane marking by using the lane marking array. Specify the normalized length for each marking object.

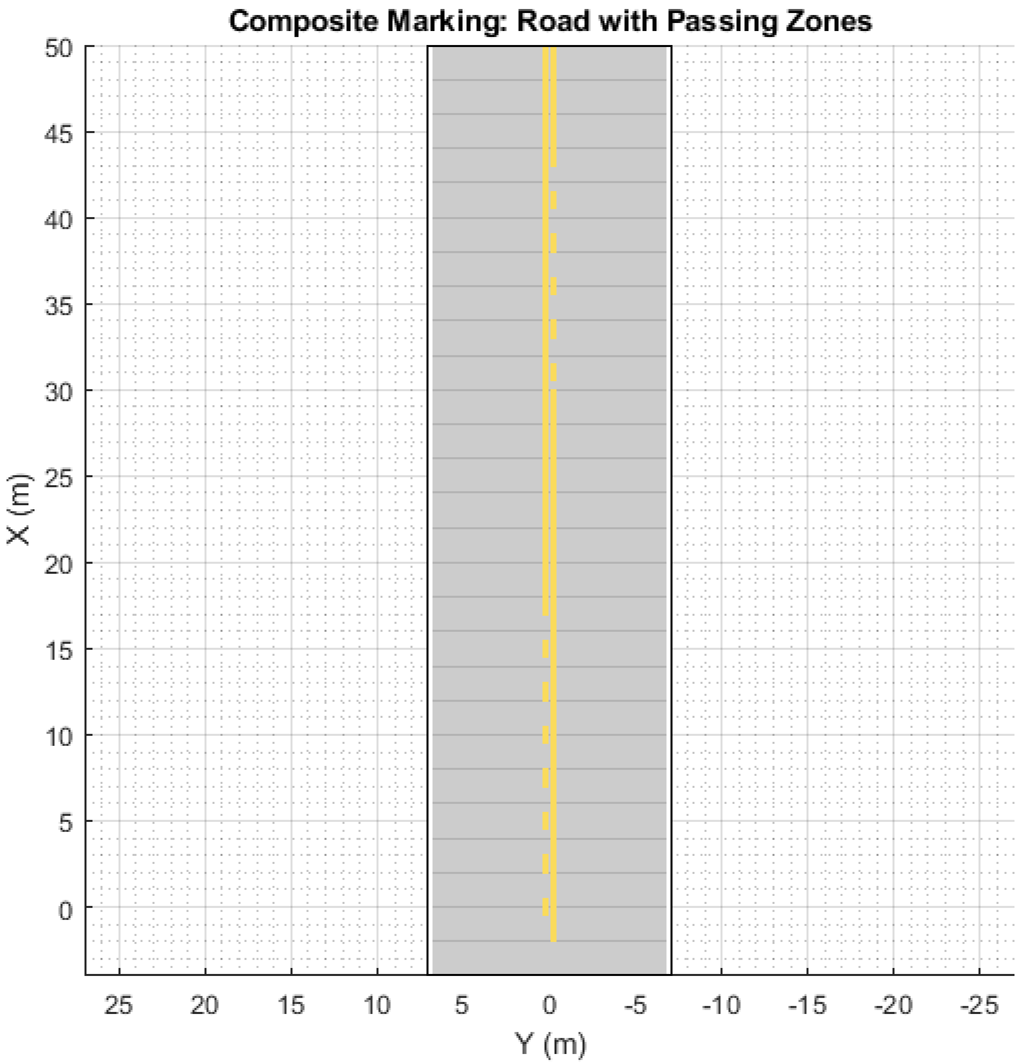
```
centerLM = laneMarking(lmArray,'SegmentRange',[0.1 0.25 0.2 0.35]);
```

Create a vector of the outermost and the center lane marking objects. Pass the vector as input to the `lanespec` function in order to define the lane specifications of the road.

```
marking = [outerLM centerLM outerLM];
ls = lanespec(numLanes,'Width',7,'Marking',marking);
```

Add the road to the driving scenario. Plot the driving scenario. Since the draw direction of the road is from top-to-bottom, the marking types in the composite lane marking also occur in top-to-bottom order.

```
road(scenario,roadCenters,'Lanes',ls);
figMark = figure;
set(figMark,'Position',[0 0 600 600]);
hPlot = axes(figMark);
plot(scenario,'Parent',hPlot);
title('Composite Marking: Road with Passing Zones')
```



Simulate Vehicle Lane Change in Passing Zones

Add a slow moving vehicle (SMV) to the scenario. Specify the waypoints and speed value to set the trajectory for the SMV.

```
slowVehicle = vehicle(scenario,'ClassID',1,'Position',[37 -3 0]);
waypoints = [37 -3;12 -3];
speed = 2;
trajectory(slowVehicle,waypoints,speed);
```

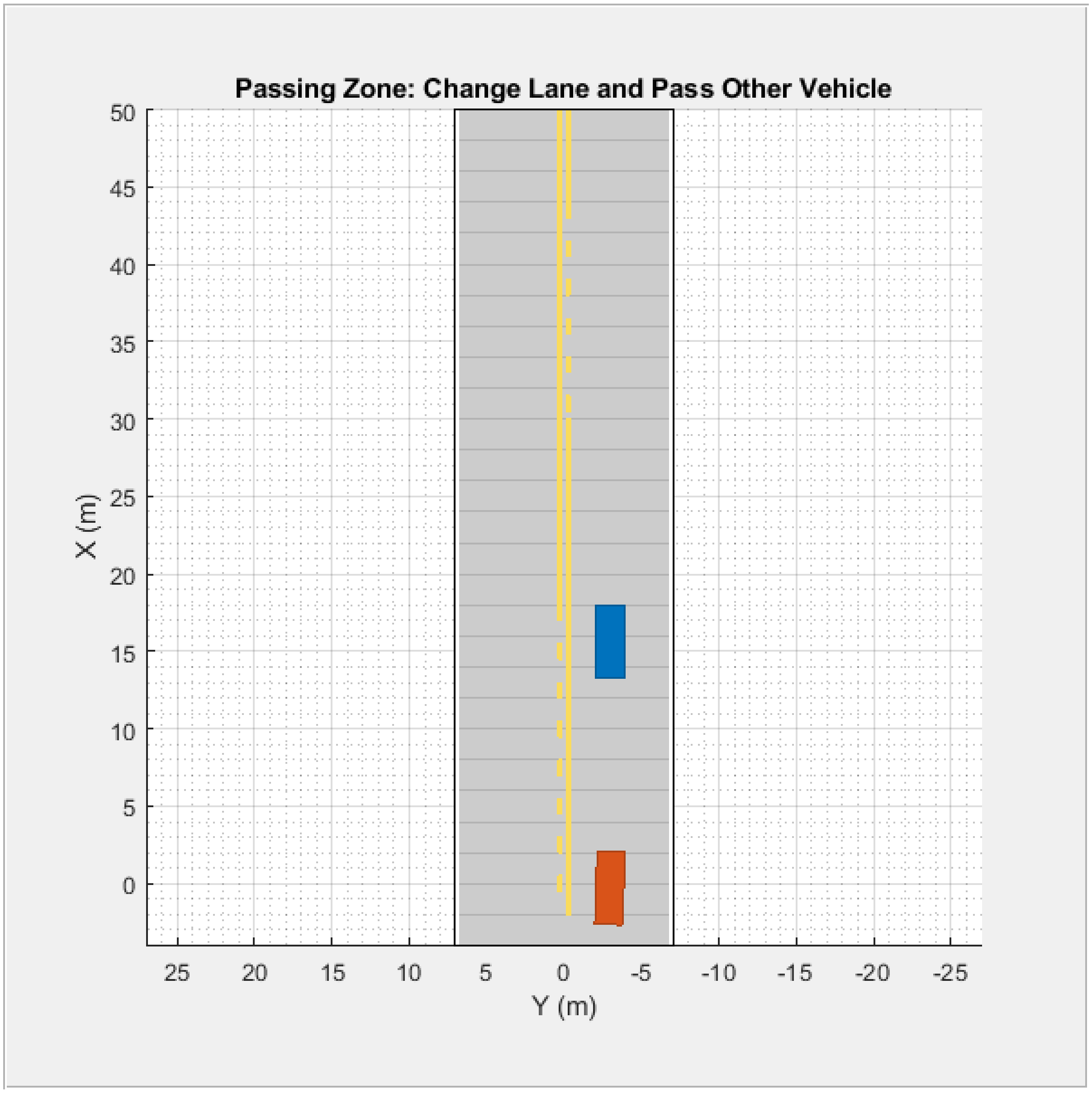
Add another vehicle to the scenario. Set the trajectory for the vehicle in such a way that it passes the SMV in front of it by changing lanes at the passing zones.

```
passingVehicle = vehicle(scenario,'ClassID',1,'Position',[49 -3 0]);
waypoints = [49 -3; 45 -3; 40 -3; 35 0;
```

```
             30 3; 26 3; 22 3; 18 3;
             8 0; 5 -2; 2 -3; 1 -3];
speed = 6;
trajectory(passingVehicle,waypoints,speed);
```

Create a custom figure window and plot the scenario.

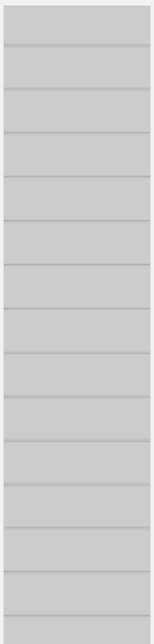
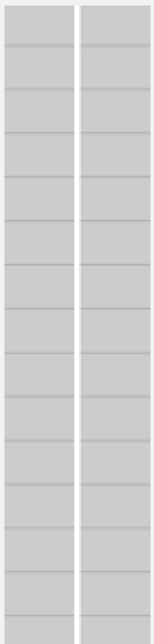
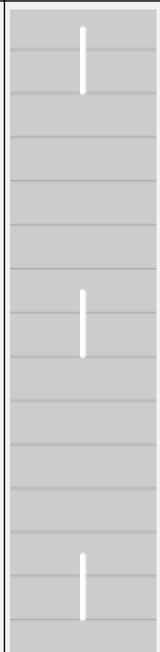
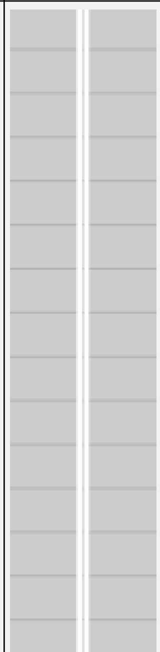
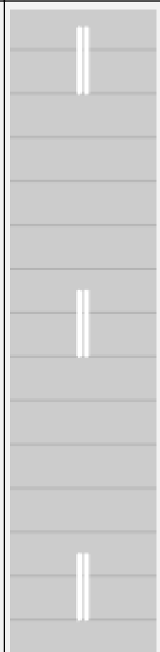
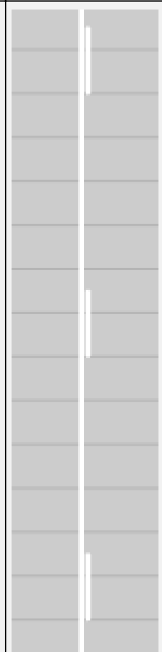
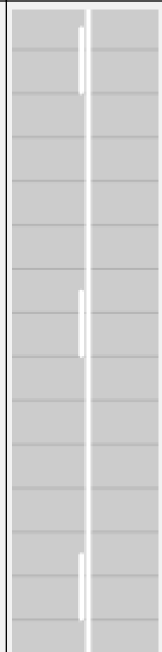
```
close all;
figScene = figure;
set(figScene,'Position',[0 0 600 600]);
hPanel = uipanel(figScene);
hPlot = axes(hPanel);
plot(scenario,'Parent',hPlot);
title('Passing Zone: Change Lane and Pass Other Vehicle')

% Run the simulation
while advance(scenario)
    pause(0.01)
end
```



Input Arguments

type — Type of lane marking
'Unmarked' | 'Solid' | 'Dashed' | 'DoubleSolid' | 'DoubleDashed' | 'SolidDashed' | 'DashedSolid'

Type of lane marking, specified as one of these values.

'Unmarked'	'Solid'	'Dashed'	'DoubleSolid'	'DoubleDashed'	'SolidDashed'	'DashedSolid'
No lane marking	Solid line	Dashed line	Two solid lines	Two dashed lines	Solid line on left, dashed line on right	Dashed line on left, solid line on right
						

The type of lane marking is stored in `Type`, a read-only property of the returned lane marking object.

`lmArray` — 1-D array of lane marking objects
`LaneMarking` object | `SolidMarking` object | `DashedMarking` object

1-D array of lane marking objects, specified as

- `LaneMarking` object for `'Unmarked'` type of lane marking.
- `SolidMarking` object for `'Solid'` and `'DoubleSolid'` types of lane marking.
- `DashedMarking` object for `'Dashed'`, `'DoubleDashed'`, `'SolidDashed'`, and `'DashedSolid'` types of lane marking.

Example: `lmArray = [laneMarking('Solid') laneMarking('Dashed')]`

range — Range for each marking type
vector

Range for each marking type, specified as a vector with normalized values in the interval [0, 1]. The length of the vector must be same as the number of marking types specified in the input array `lmArray`.

The default range value for each marking type in the lane is the inverse of the number of marking types specified in `lmArray`.

For example, if the input lane marking array contains three lane marking objects, such as lmArray = [laneMarking('Solid') laneMarking('Dashed') laneMarking('Solid')], then the default range value for each marking type is 1/3, that is range = [0.3330 0.3330 0.3330].

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: laneMarking('Dashed','Width',0.25,'Length',5.0) creates a lane with dashes that are 0.25 meters wide and spaced 5 meters apart.

Width — Lane marking widths

0.15 (default) | positive real scalar

Lane marking widths, specified as the comma-separated pair consisting of 'Width' and a positive real scalar. For a double lane marker, the same width is used for both lines. Units are in meters.

Example: 0.20

Color — Color of lane marking

[1 1 1] (white) (default) | RGB triplet | hexadecimal color code | color name | short color name

Color of lane marking, specified as the comma-separated pair consisting of 'Color' and an RGB triplet, a hexadecimal color code, a color name, or a short color name. For a double lane marker, the same color is used for both lines.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[0.98 0.86 0.36]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Example: [0.8 0.8 0.8]

Strength — Saturation strength of lane marking color
1 (default) | real scalar in the range [0, 1]

Saturation strength of lane marking color, specified as the comma-separated pair consisting of 'Strength' and a real scalar in the range [0, 1]. A value of 0 corresponds to a marking whose color is fully unsaturated. The marking is gray. A value of 1 corresponds to a marking whose color is fully saturated. For a double lane marking, the same strength is used for both lines.

Example: 0.20

Length — Length of dash in dashed lines
3.0 (default) | positive real scalar

Length of dash in dashed lines, specified as the comma-separated pair consisting of 'Length' and a positive real scalar. For a double lane marking, the same length is used for both lines. The dash is the visible part of a dashed line. Units are in meters.

Example: 2.0

Space — Length of space between dashes in dashed lines
9.0 (default) | positive real scalar

Length of space between the end of one dash and the beginning of the next dash, specified as the comma-separated pair consisting of 'Space' and a positive real scalar. For a double lane marking, the same length is used for both lines. Units are in meters.

Example: 2.0

Output Arguments

lm — Lane marking
LaneMarking object | SolidMarking object | DashedMarking object

Lane marking, returned as a LaneMarking object, SolidMarking object, or DashedMarking object. The type of returned object depends on the type of input lane marking specified for the type input.

Input Type	Output Lane Marking	Lane Marking Properties
'Unmarked'	LaneMarking object	• Type
'Solid' 'DoubleSolid'	SolidMarking object	• Color • Width • Strength • Type
'Dashed' 'DashedSolid' 'SolidDashed'	DashedMarking object	• Length • Space • Color • Width • Strength • Type

Input Type	Output Lane Marking	Lane Marking Properties
'DoubleDashed'		

You can set these properties when you create the lane marking object by using the corresponding name-value pairs of the `laneMarking` function. To update these properties after creation, use dot notation. For example:

```
lm = laneMarking('Solid');
lm.Width = 0.2;
```

You can set all properties after creation except `Type`, which is read-only. For details on the geometric properties of the `SolidMarking` and `DashedMarking` objects, see Lane Specifications on page 4-502.

cm — Composite lane marking
`CompositeMarking` object

Composite lane marking, returned as a `CompositeMarking` object with properties `Markings` and `SegmentRange`.

Composite Lane Marking Properties	Description
`Markings`	This property is an array of lane marking objects that defines the multiple marking types comprising the composite lane marking. The part of the lane marking defined by each lane marking object is a marker segment. This property is read-only.
`SegmentRange`	This property specifies the normalized range for each marker segment in the composite lane marking. This property is read-only.

Use this object to specify multiple marking types along a lane. For more details on how to specify composite lane markings and the order of marker segments along the lane, see “Composite Lane Marking” on page 4-501.

More About

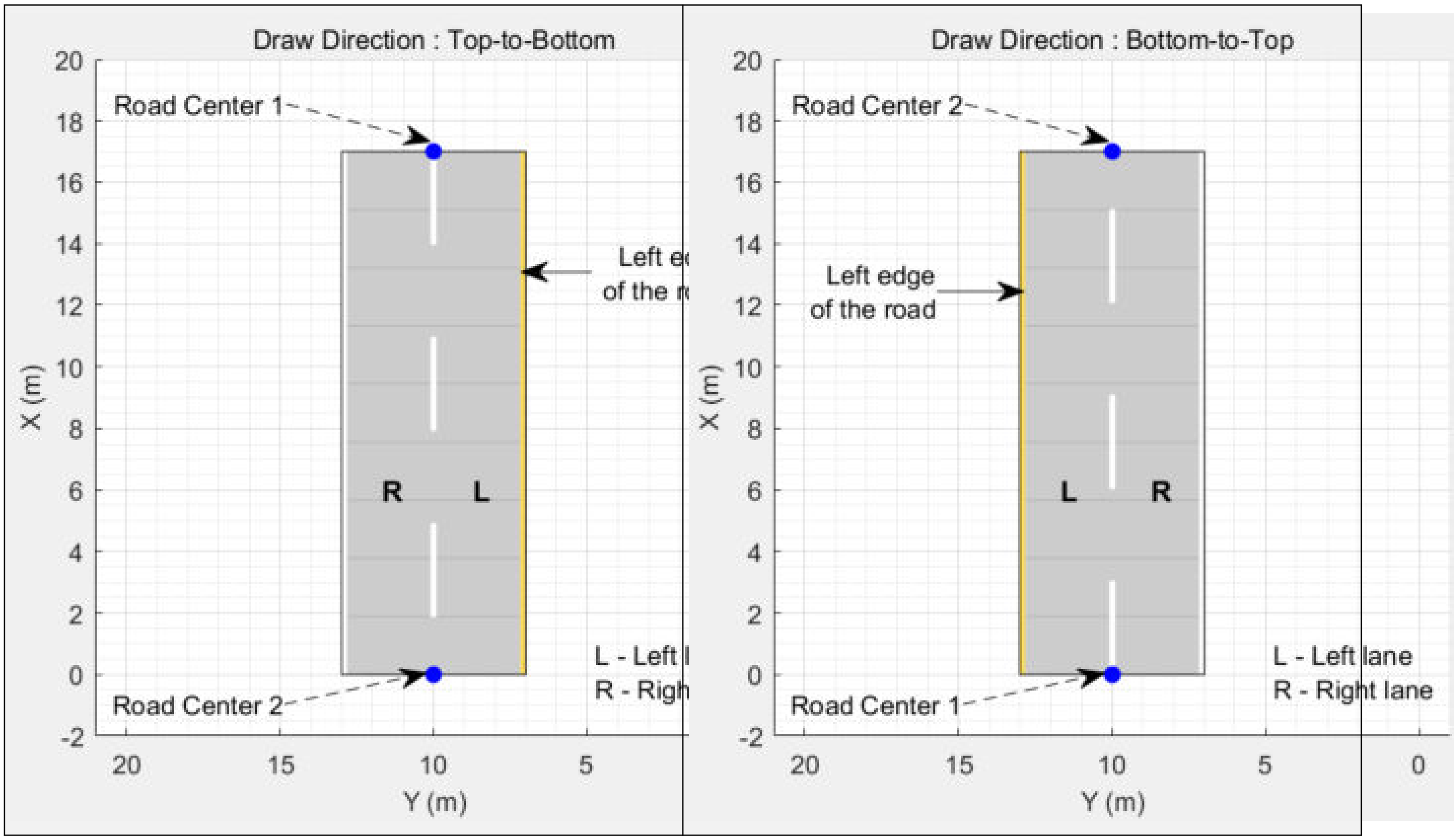
Draw Direction of Road and Numbering of Lanes

To create a road by using the `road` function, specify the road centers as a matrix input. The function creates a directed line that traverses the road centers, starting from the coordinates in the first row of the matrix and ending at the coordinates in the last row of the matrix. The coordinates in the first two rows of the matrix specify the draw direction of the road. These coordinates correspond to the first two consecutive road centers. The draw direction is the direction in which the roads render in the scenario plot.

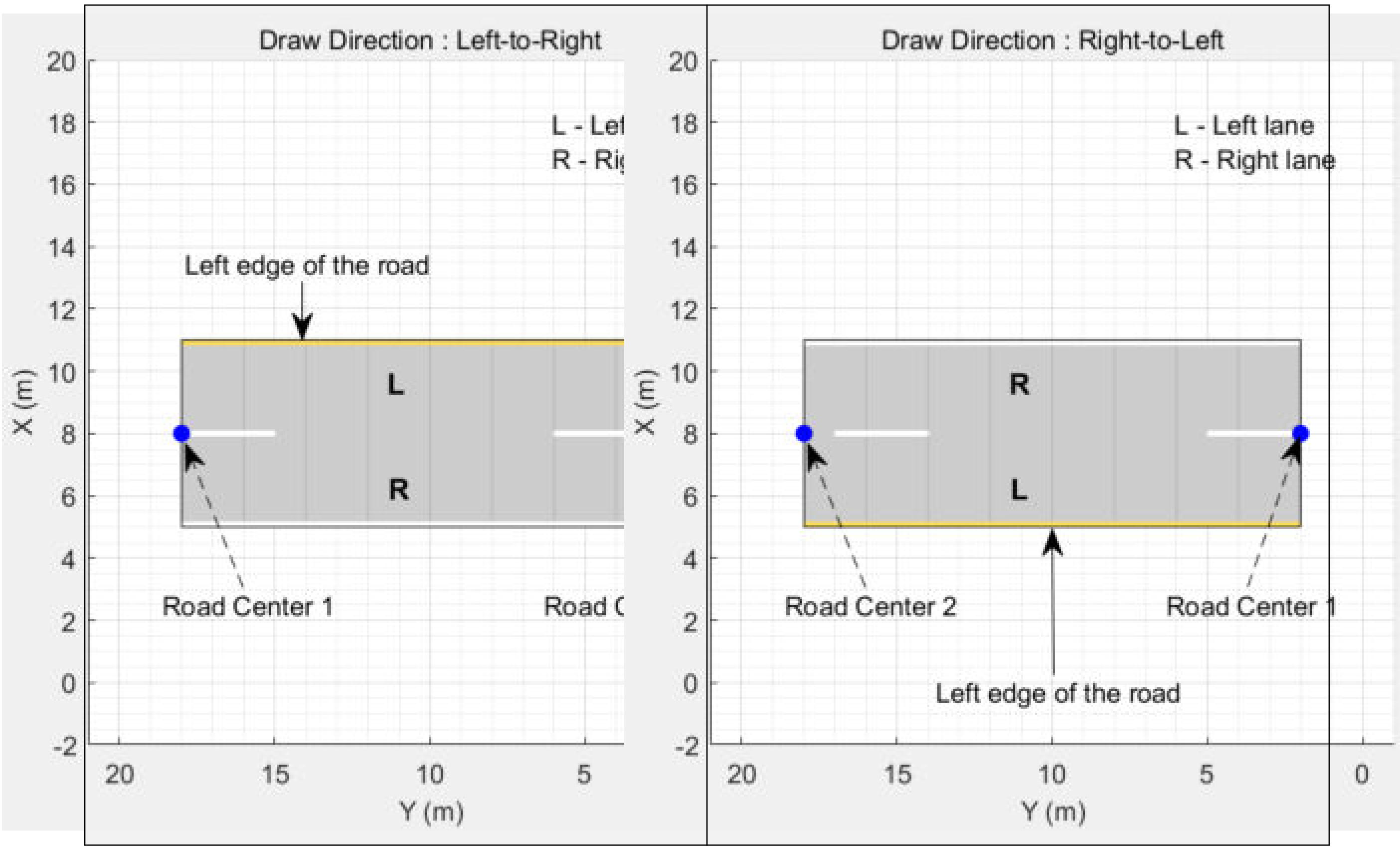
To create a road by using the **Driving Scenario Designer** app, you can either specify the **Road Centers** parameter or interactively draw on the **Scenario Canvas**. For a detailed example, see “Create a Driving Scenario” on page 1-14. In this case, the draw direction is the direction in which roads render in the **Scenario Canvas**.

- For a road with a top-to-bottom draw direction, the difference between the *x*-coordinates of the first two consecutive road centers is positive.

- For a road with a bottom-to-top draw direction, the difference between the *x*-coordinates of the first two consecutive road centers is negative.



- For a road with a left-to-right draw direction, the difference between the *y*-coordinates of the first two consecutive road centers is positive.
- For a road with a right-to-left draw direction, the difference between the *y*-coordinates of the first two consecutive road centers is negative.

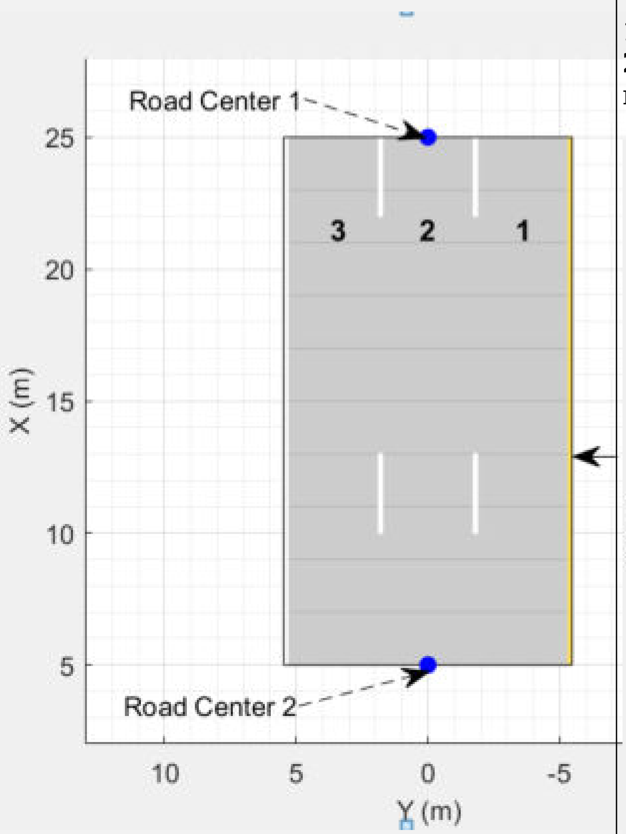


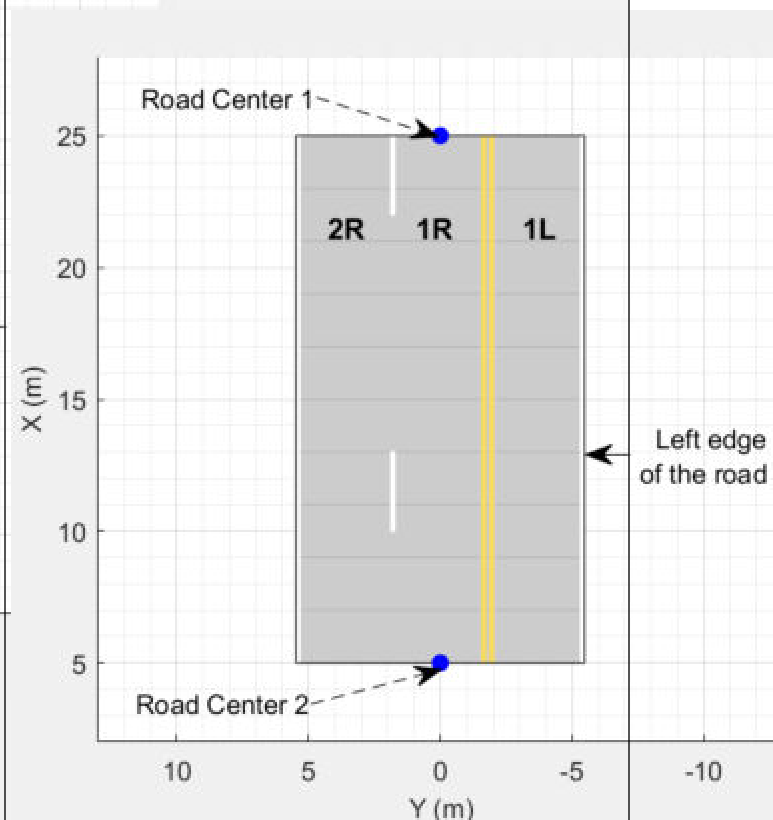
Numbering Lanes

Lanes must be numbered from left to right, with the left edge of the road defined relative to the draw direction of the road. For a one-way road, by default, the left edge of the road is a solid yellow marking which indicates the end of the road in transverse direction (direction perpendicular to draw direction). For a two-way road, by default, both edges are marked with solid white lines.

For example, these diagrams show how the lanes are numbered in a one-way and two-way road with a draw direction from top-to-bottom.

Numbering Lanes in a One-Way Road	Numbering Lanes in a Two-Way Road

<table>
<tr>
<td>
Specify the number of lanes as a positive integer for a one-way road. If you set the integer value as 3, then the road has three lanes that travel in the same direction. The lanes are numbered starting from the left edge of the road.
1, 2, 3 denote the first, second, and third lanes of the road, respectively.


</td>
<td>
Specify the number of lanes as a two-element vector of positive integer for a two-way road. If you set the vector as [1 2], then the road has three lanes: two lanes traveling in one direction and one lane traveling in the opposite direction. Because of the draw direction, the road has one left lane and two right lanes. The lanes are numbered starting from the left edge of the road.
1L denote the only left lane of the road. 1R and 2R denote the first and second right lanes of the road, respectively.


</td>
</tr>
</table>

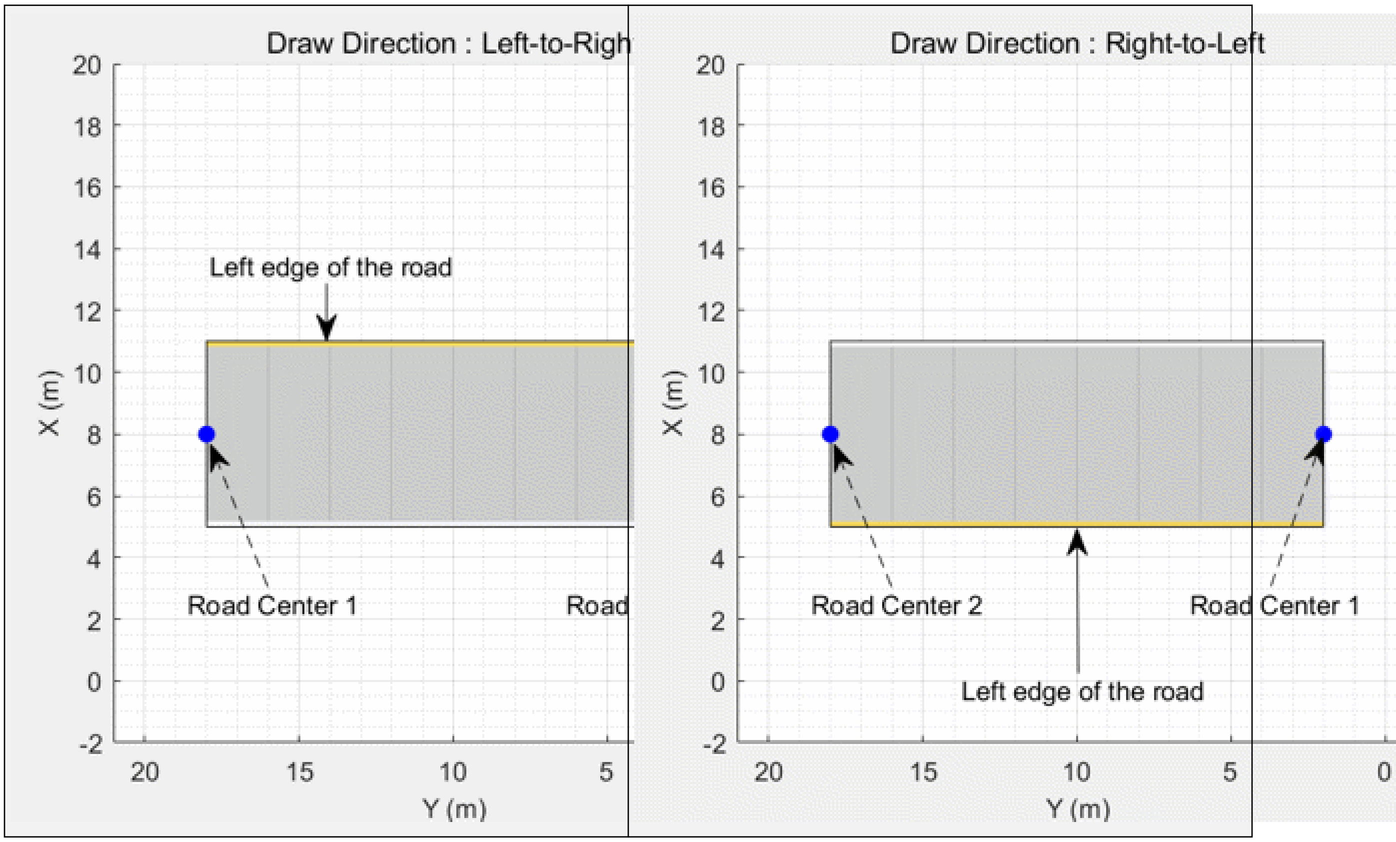
The lane specifications apply by the order in which the lanes are numbered.

Composite Lane Marking

A composite lane marking comprises two or more marker segments that define multiple marking types along a lane. The geometric properties for a composite lane marking include the geometric properties of each marking type and the normalized lengths of the marker segments.

The order in which the specified marker segments occur in a composite lane marking depends on the draw direction of the road. Each marker segment is a directed segment with a start point and moves towards the last road center. The first marker segment starts from the first road center and moves towards the last road center for a specified length. The second marker segment starts from the end point of the first marker segment and moves towards the last road center for a specified length. The same process applies for each marker segment that you specify for the composite lane marking. You can set the normalized length for each of these marker segments by specifying the `range` input argument.

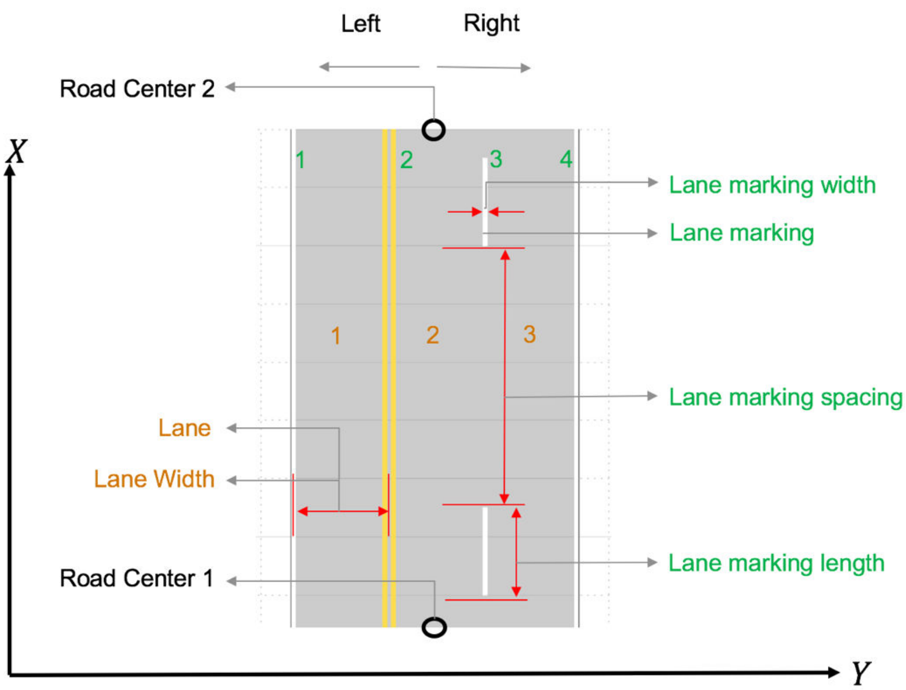
For example, consider a one-way road with two lanes. The second lane marking from the left edge of the road is a composite lane marking with marking types `Solid` and `Dashed`. The normalized range for each marking type is 0.5. The first marker segment is a solid marking and the second marker segment is a dashed marking. These diagrams show the order in which the marker segments apply for left-to-right and right-to-left draw directions of the road.



For information on the geometric properties of lane markings, see “Lane Specifications” on page 4-502.

Lane Specifications

The diagram shows the components and geometric properties of roads, lanes, and lane markings.



The lane specification object, `lanespec`, defines the road lanes.

- The `NumLanes` property specifies the number of lanes. You must specify the number of lanes when you create this object.
- The `Width` property specifies the width of each lane.
- The `Marking` property contains the specifications of each lane marking in the road. `Marking` is an array of lane marking objects, with one object per lane. To create these objects, use the `laneMarking` function. Lane marking specifications include:

  - `Type` — Type of lane marking (solid, dashed, and so on)
  - `Width` — Lane marking width
  - `Color` — Lane marking color
  - `Strength` — Saturation value for lane marking color
  - `Length` — For dashed lanes, the length of each dashed line
  - `Space` — For dashed lanes, the spacing between dashes
  - `SegmentRange` — For composite lane marking, the normalized length of each marker segment.
- The `Type` property contains the lane type specifications of each lane in the road. `Type` can be a homogeneous lane type object or a heterogeneous lane type array.

  - Homogeneous lane type object contain lane type specifications of all the lanes in the road.
  - Heterogeneous lane type array contain an array of lane type objects, with one object per lane.

  To create these objects, use the `laneType` function. Lane type specifications include:

  - `Type` — Type of lane (driving, border, and so on)

- `Color` — Lane color
- `Strength` — Strength of the lane color

See Also

Objects
`drivingScenario` | `lanespec`

Functions
`laneBoundaryPlotter` | `laneMarkingPlotter` | `laneMarkingVertices` | `plotLaneBoundary` | `plotLaneMarking` | `road`

Introduced in R2018a

laneType

Create road lane type object

Syntax

```
lt = laneType(type)
lt = laneType(type,Name,Value)
```

Description

`lt = laneType(type)` returns a road lane type object with properties `Type`, `Color`, and `Strength` to define different lane types for a road.

You can use this object to create driving scenarios with roads that have driving lanes, border lanes, restricted lanes, shoulder lanes, and parking lanes. You can also load this scenario into the **Driving Scenario Designer** app.

For details on the steps involved in using `laneType` function with the `drivingScenario` object and the **Driving Scenario Designer** app, see “More About” on page 4-512.

`lt = laneType(type,Name,Value)` sets the properties of the output lane type object by using one or more name-value pairs.

Examples

Add Roads That Have Different Lane Types to Driving Scenario

This example shows how to define lane types and simulate a driving scenario for a four-lane road that has different lane types.

Create a driving lane object with default property values.

```
drivingLane = laneType('Driving')
```

```
drivingLane = 
  DrivingLaneType with properties:

        Type: Driving
       Color: [0.8000 0.8000 0.8000]
    Strength: 1
```

Create a parking lane type object. Specify the color and the strength property values.

```
parkingLane = laneType('Parking','Color',[1 0 0],'Strength',0.1)
```

```
parkingLane = 
  ParkingLaneType with properties:

        Type: Parking
       Color: [1 0 0]
```

```
    Strength: 0.1000
```

Create a three-element, heterogeneous lane type array by concatenating the driving and the parking lane type objects. The lane type array contains lane types for a four-lane road.

```
lt = [parkingLane drivingLane drivingLane parkingLane];
```

Create lane specification for a four-lane road. Add the lane type array to the lane specification.

```
ls = lanespec([2 2],'Type',lt);
```

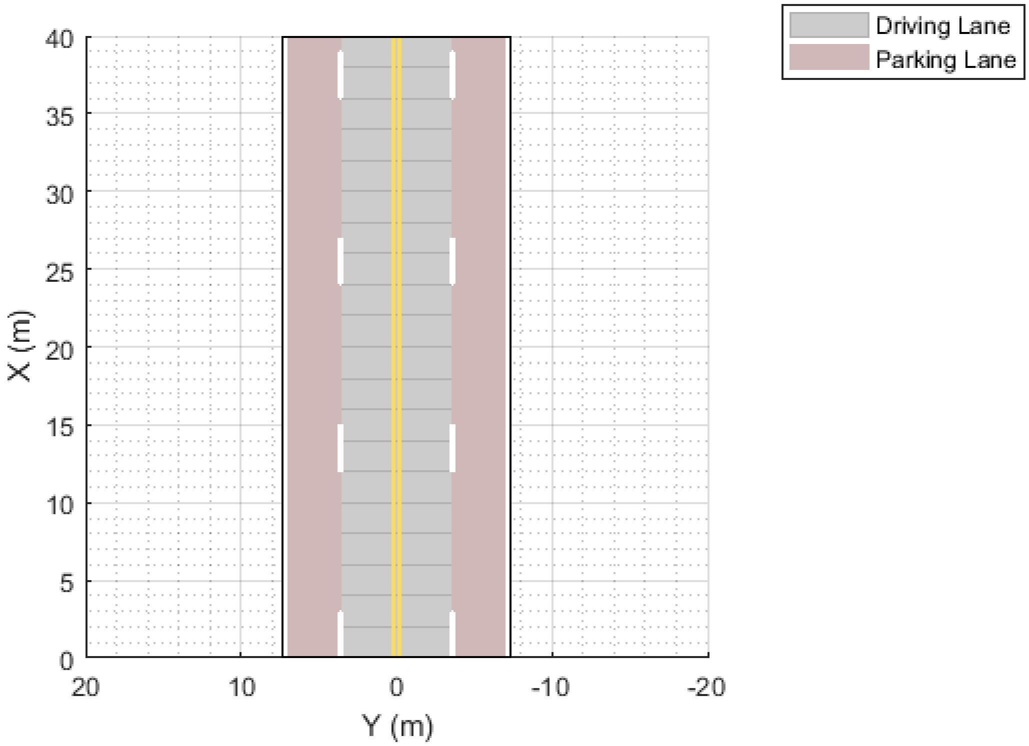
Create a driving scenario object. Add the four-lane road with lane specifications `ls` to the driving scenario.

```
scenario = drivingScenario;
roadCenters = [0 0 0;40 0 0];
road(scenario,roadCenters,'Lanes',ls)
```

```
ans = 
  Road with properties:

           Name: ""
         RoadID: 1
    RoadCenters: [2x3 double]
      RoadWidth: 14.5500
      BankAngle: [2x1 double]
```

Plot the scenario. The scenario contains the four-lane road that has two parking lanes and two driving lanes.

```
plot(scenario)
legend('Driving Lane','Parking Lane')
```



Simulate Vehicles Travelling on Road That Has Multiple Lane Types

Create a heterogeneous lane type object array to define driving, shoulder, and border lane types for a four-lane road.

```
lt = [laneType('Shoulder') laneType('Driving') laneType('Driving') laneType('Border','Color',[0.5
```

Display the lane type object array.

```
lt
```

```
lt=1×4 object
  1x4 heterogeneous LaneType (ShoulderLaneType, DrivingLaneType, BorderLaneType) array with prope

    Type
    Color
    Strength
```

Inspect the property values.

```
c = [{lt.Type}' {lt.Color}' {lt.Strength}'];
cell2table(c,'VariableNames',{'Type','Color','Strength'})
```

```
ans=4×3 table
      Type              Color              Strength
```

```
    ________    ____________________    ________

    Shoulder    0.59    0.59    0.59        1
    Driving      0.8     0.8     0.8        1
    Driving      0.8     0.8     0.8        1
    Border       0.5       0       1      0.1
```

Pass the lane type object array as input to the `lanespec` function, and then create a lane specification object for the four-lane road.

```
lspec = lanespec([2 2],'Type',lt);
```

Define the road centers.

```
roadCenters = [0 0 0; 40 0 0];
```

To add roads, create a driving scenario object.

```
scenario = drivingScenario('StopTime',8);
```

Add roads with the specified road centers and lane types to the driving scenario.

```
road(scenario,roadCenters,'Lanes',lspec);
```

Add two vehicles to the scenario. Position the vehicles on the driving lane.

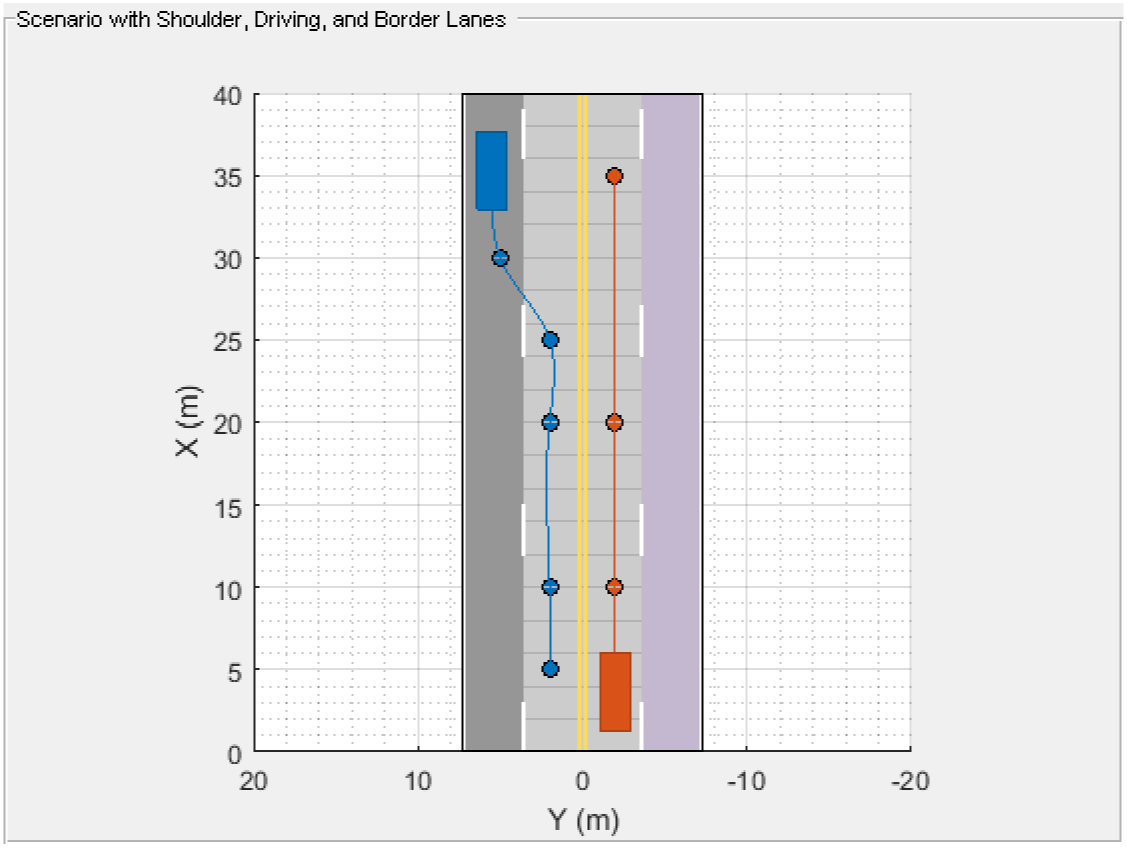
```
vehicle1 = vehicle(scenario,'ClassID',1,'Position',[5 2 0]);
vehicle2 = vehicle(scenario,'ClassID',1,'Position',[35 -2 0]);
```

Define the vehicle trajectories by using waypoints. Set the vehicle trajectory speeds.

```
waypoints1 = [5 2;10 2;20 2;25 2;30 5;34 5.5];
trajectory(vehicle1,waypoints1,10)
waypoints2 = [35 -2;20 -2;10 -2;5 -2];
trajectory(vehicle2,waypoints2,5)
```

Plot the scenario. To advance the simulation one time step at a time, call the `advance` function in a loop. Pause every 0.01 second to observe the motion of the vehicles on the plot. The first vehicle travels along the trajectory in the driving lane. It drifts to the shoulder lane for emergency stopping.

```
% Create a custom figure window and define an axes object
fig = figure;
movegui(fig,'center');
hView = uipanel(fig,'Position',[0 0 1 1],'Title','Scenario with Shoulder, Driving, and Border Lan
hPlt = axes(hView);

% Plot the generated driving scenario along with the waypoints.
plot(scenario,'Waypoints','on','Parent',hPlt);
while advance(scenario)
    pause(0.01)
end
```



Input Arguments

type — Lane type
'Driving' | 'Border' | 'Restricted' | 'Shoulder' | 'Parking'

Lane type, specified as 'Driving', 'Border', 'Restricted', 'Shoulder', or 'Parking'.

Lane Type	Description
'Driving'	Lanes for driving
'Border'	Lanes at the road borders
'Restricted'	Lanes reserved for high occupancy vehicles
'Shoulder'	Lanes reserved for emergency stopping
'Parking'	Lanes alongside driving lanes, intended for parking vehicles

Note The lane type input sets the Type property of the output lane type object.

Data Types: char | string

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

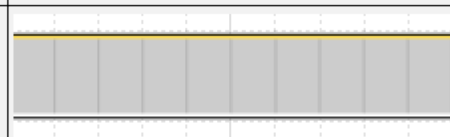
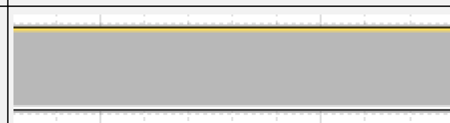
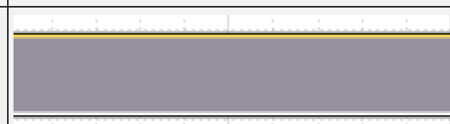
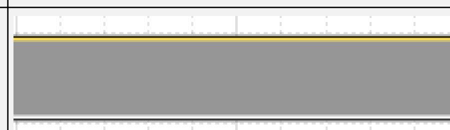
Example: `laneType('Driving','Color','r')`

`Color` — Lane color

RGB triplet | color name

Lane color, specified as the comma-separated pair consisting of `'Color'` and an RGB triplet or color name.

Specify the RGB triplet as a three-element row vector containing the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists the RGB triplet values that specify the default colors for different lane types.

Lane Type	RGB Triplet (Default values)	Appearance
'Driving'	[0.8 0.8 0.8]	
'Border'	[0.72 0.72 0.72]	
'Restricted'	[0.59 0.56 0.62]	
'Shoulder'	[0.59 0.59 0.59]	
'Parking'	[0.28 0.28 0.28]	

Alternatively, you can specify some common colors by name. This table lists the named color options and the equivalent RGB triplet values.

Color Name	RGB Triplet	Appearance
'red'	[1 0 0]	
'green'	[0 1 0]	
'blue'	[0 0 1]	
'cyan'	[0 1 1]	
'magenta'	[1 0 1]	

Color Name	RGB Triplet	Appearance
'yellow'	[0.98 0.86 0.36]	
'black'	[0 0 0]	
'white'	[1 1 1]	

Note Use the lane color name-value pair to set the `Color` property of the output lane type object.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`Strength` — Strength of lane color
1 (default) | real scalar in the range [0, 1]

Strength of lane color, specified as a comma-separated pair consisting of `'Strength'` and a real scalar in the range [0, 1]. A value of `0` desaturates the color and the lane color appears gray. A value of `1` fully saturates the color and the lane color is the pure color. You can vary the strength value to modify the level of saturation.

Note Use the strength of lane color name-value pair to set the `Strength` property of the lane type object.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`lt` — Lane type
`DrivingLaneType` object | `BorderLaneType` object | `RestrictedLaneType` object | `ShoulderLaneType` object | `ParkingLaneType` object

Lane type, returned as a

- `DrivingLaneType` object
- `BorderLaneType` object
- `RestrictedLaneType` object
- `ShoulderLaneType` object
- `ParkingLaneType` object

The returned object `lt` depends on the value of the input `type`.

type	lt
'Driving'	`DrivingLaneType` object
'Border'	`BorderLaneType` object
'Restricted'	`RestrictedLaneType` object
'Shoulder'	`ShoulderLaneType` object
'Parking'	`ParkingLaneType` object

You can create a heterogeneous `LaneType` array by concatenating these different lane type objects.

More About

Create Driving Scenario With Roads That Have Multiple Lane Types

You can add roads that have multiple lane types to the driving scenario by following these steps

1. Create an empty `drivingScenario` object.
2. Create a lane type object that defines different lane types on the road by using `laneType`.
3. Use lane type object as input to the `lanespec` object and define lane specifications for the road.
4. Use `lanespec` object as input to the `road` function and add roads that have the specified lane types to the driving scenario.

You can use the `plot` function to visualize the driving scenario.

You can also import a driving scenario containing roads that have different lane types into the **Driving Scenario Designer** app. To import a `drivingScenario` object named `scenario` into the app, use the syntax `drivingScenarioDesigner(scenario)`. In the scenarios, you can:

- Add or edit the road centers.
- Add actors and define actor trajectories.
- Mount sensors on the ego vehicle and simulate detection of actors and lane boundaries.

Note Editing the lane parameters resets all the lanes in the imported road to lane type `'Driving'` with the default property values.

See Also

Functions
`road` | `roadNetwork`

Objects
`drivingScenario` | `lanespec`

Introduced in R2019b

laneMarkingVertices

Package:

Lane marking vertices and faces in driving scenario

Syntax

```
[lmv,lmf] = laneMarkingVertices(scenario)
[lmv,lmf] = laneMarkingVertices(ac)
```

Description

`[lmv,lmf] = laneMarkingVertices(scenario)` returns the lane marking vertices, `lmv`, and lane marking faces, `lmf`, contained in driving scenario `scenario`. The `lmf` and `lmv` outputs are in the world coordinates of `scenario`. Use lane marking vertices and faces to display lane markings using the `laneMarkingPlotter` function with a bird's-eye plot.

`[lmv,lmf] = laneMarkingVertices(ac)` returns lane marking vertices and faces in the coordinates of driving scenario actor `ac`.

Examples

Display Lane Markings in Car and Pedestrian Scenario

Create a driving scenario containing a car and pedestrian on a straight road. Then, create and display the lane markings of the road on a bird's-eye plot.

Create an empty driving scenario.

```
scenario = drivingScenario;
```

Create a straight, 25-meter road segment with two travel lanes in one direction.

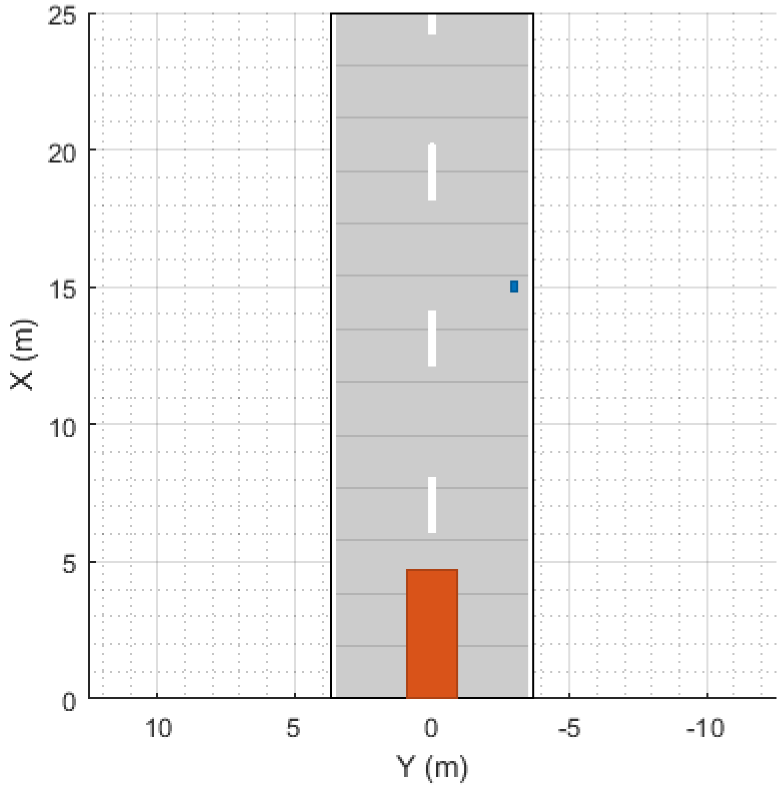
```
lm = [laneMarking('Solid')
      laneMarking('Dashed','Length',2,'Space',4)
      laneMarking('Solid')];
l = lanespec(2,'Marking',lm);
road(scenario,[0 0 0; 25 0 0],'Lanes',l);
```

Add to the driving scenario a pedestrian crossing the road at 1 meter per second and a car following the road at 10 meters per second.

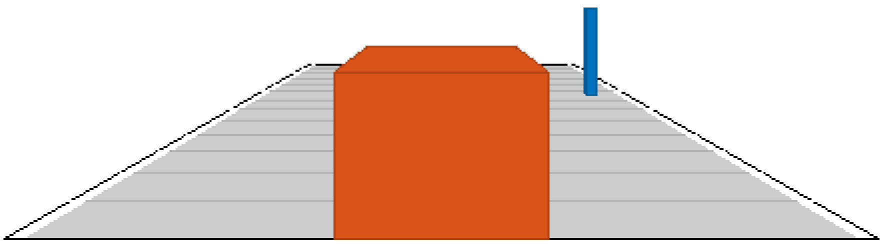
```
ped = actor(scenario,'ClassID',4,'Length',0.2,'Width',0.4,'Height',1.7);
car = vehicle(scenario,'ClassID',1);
trajectory(ped,[15 -3 0; 15 3 0],1);
trajectory(car,[car.RearOverhang 0 0; 25-car.Length+car.RearOverhang 0 0],10);
```

Display the scenario and corresponding chase plot.

```
plot(scenario)
```



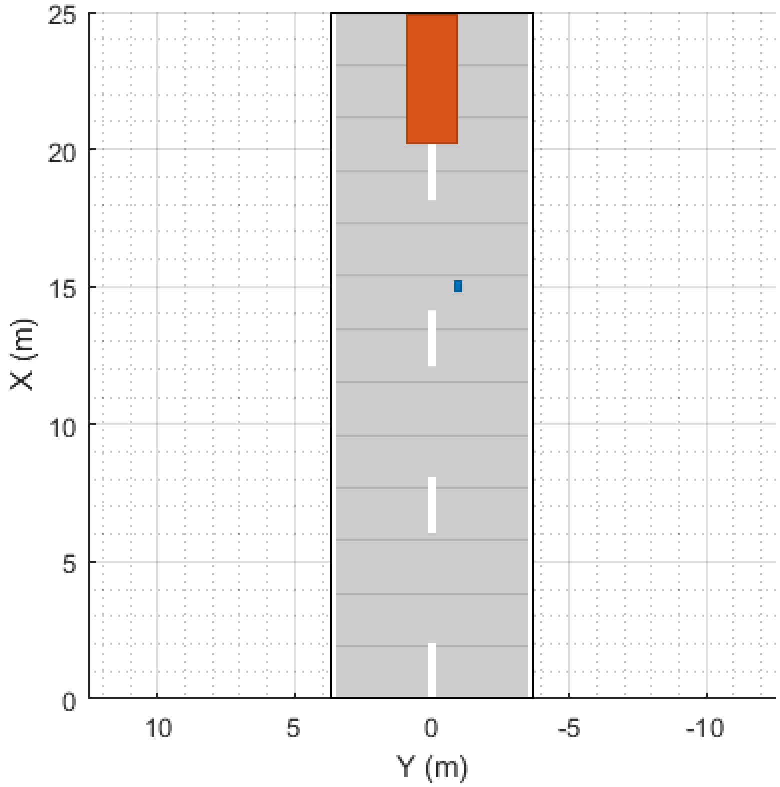

```
chasePlot(car)
```

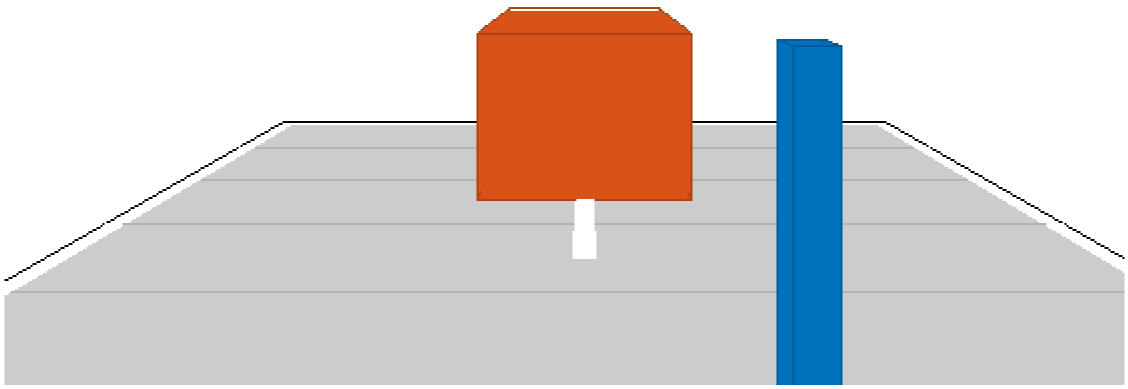


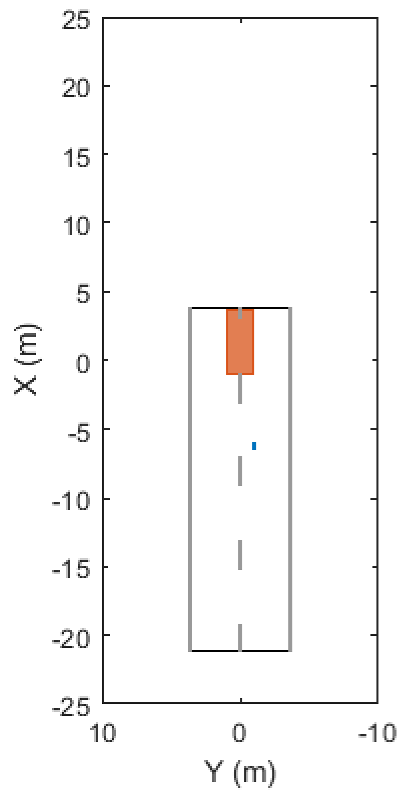
Run the simulation.

1 Create a bird's-eye plot.
2 Create an outline plotter, lane boundary plotter, and lane marking plotter for the bird's-eye plot.
3 Obtain the road boundaries and target outlines.
4 Obtain the lane marking vertices and faces.
5 Display the lane boundaries and lane markers.
6 Run the simulation loop.

```
bep = birdsEyePlot('XLim',[-25 25],'YLim',[-10 10]);
olPlotter = outlinePlotter(bep);
lbPlotter = laneBoundaryPlotter(bep);
lmPlotter = laneMarkingPlotter(bep,'DisplayName','Lanes');
legend('off');
while advance(scenario)
    rb = roadBoundaries(car);
    [position,yaw,length,width,originOffset,color] = targetOutlines(car);
    [lmv,lmf] = laneMarkingVertices(car);
    plotLaneBoundary(lbPlotter,rb);
    plotLaneMarking(lmPlotter,lmv,lmf);
    plotOutline(olPlotter,position,yaw,length,width, ...
        'OriginOffset',originOffset,'Color',color);
end
```


25
20
15
10
5
0
X (m)
10
5
0
-5
-10
Y (m)





Input Arguments

`scenario` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

`ac` — Actor
`Actor` object | `Vehicle` object

Actor belonging to a `drivingScenario` object, specified as an `Actor` or `Vehicle` object. To create these objects, use the `actor` and `vehicle` functions, respectively.

Output Arguments

`lmv` — Lane marking vertices
real-valued matrix

Lane marking vertices, returned as a real-valued matrix. Each row of the matrix represents the (*x*, *y*, *z*) coordinates of a vertex.

`lmf` — Lane marking faces
real-valued matrix

Lane marking faces, returned as a real-valued matrix. Each row of the matrix contains the vertex connections that define a face for one lane marking. For more details, see “Faces”.

Algorithms

This function uses the `patch` function to define lane marking vertices and faces.

See Also

Objects
`drivingScenario`

Functions
`actor` | `laneMarking` | `laneMarkingPlotter` | `patch` | `plotLaneMarking` | `road` | `vehicle`

Introduced in R2018a

laneBoundaries

Package:

Get lane boundaries of actor lane

Syntax

```
lbdry = laneBoundaries(ac)
lbdry = laneBoundaries(ac,Name,Value)
```

Description

`lbdry = laneBoundaries(ac)` returns the lane boundaries, `lbdry`, of the lane in which the ego vehicle actor, `ac`, is traveling. The lane boundaries are in the coordinate system of the ego vehicle.

`lbdry = laneBoundaries(ac,Name,Value)` specifies options using one or more name-value pairs. For example, `laneBoundaries(ac,'AllLaneBoundaries',true)` returns all lane boundaries of the road on which the ego vehicle actor is traveling.

Examples

Simulate Car Traveling on S-Curve

Simulate a driving scenario with one car traveling on an S-curve. Create and plot the lane boundaries.

Create the driving scenario with one road having an S-curve.

```
scenario = drivingScenario('StopTime',3);
roadcenters = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
```

Create the lanes and add them to the road.

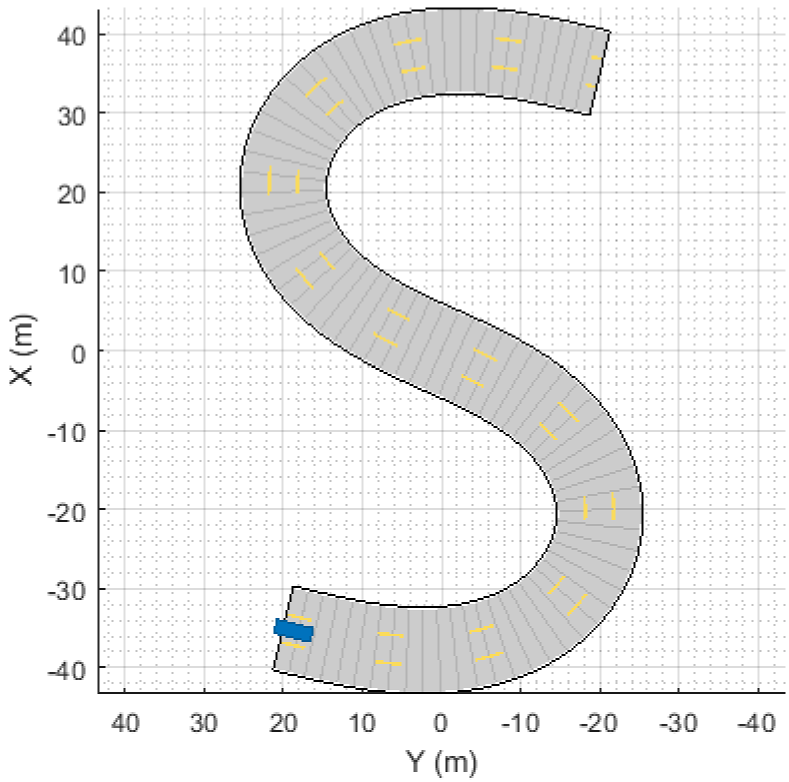
```
lm = [laneMarking('Solid','Color','w'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Solid','Color','w')];
ls = lanespec(3,'Marking',lm);
road(scenario,roadcenters,'Lanes',ls);
```

Add an ego vehicle and specify its trajectory from its waypoints. By default, the car travels at a speed of 30 meters per second.

```
car = vehicle(scenario, ...
    'ClassID',1, ...
    'Position',[-35 20 0]);
waypoints = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
trajectory(car,waypoints);
```

Plot the scenario and corresponding chase plot.

```
plot(scenario)
```

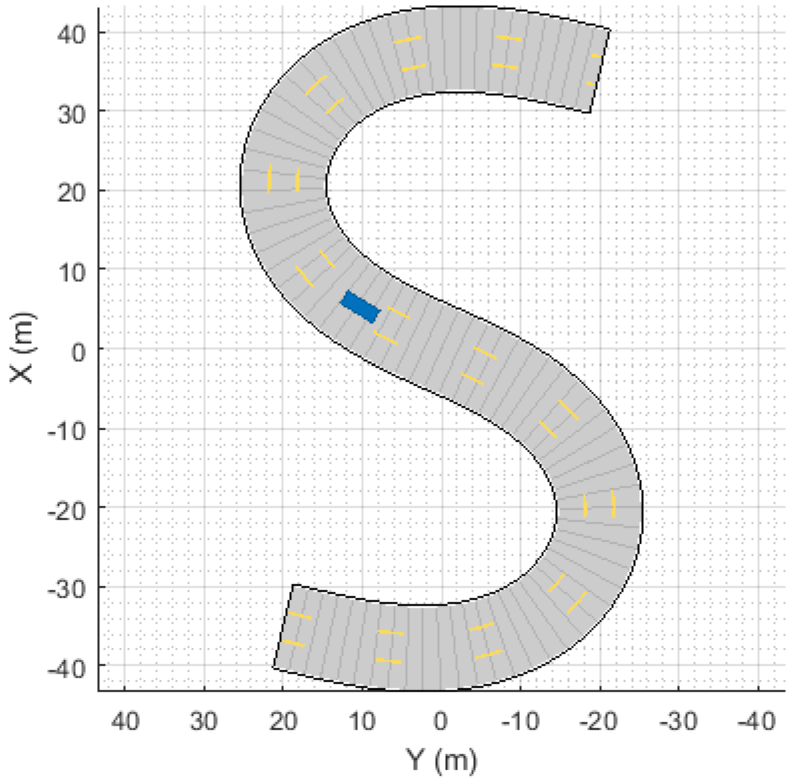


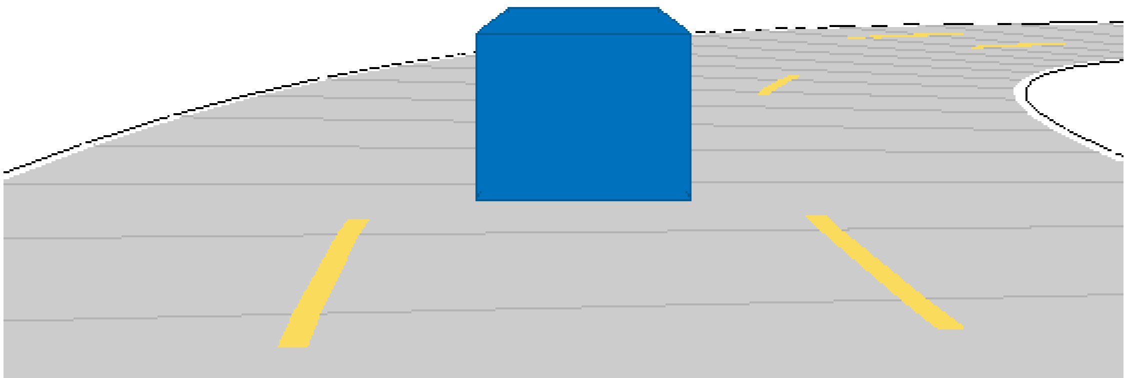

```
chasePlot(car)
```

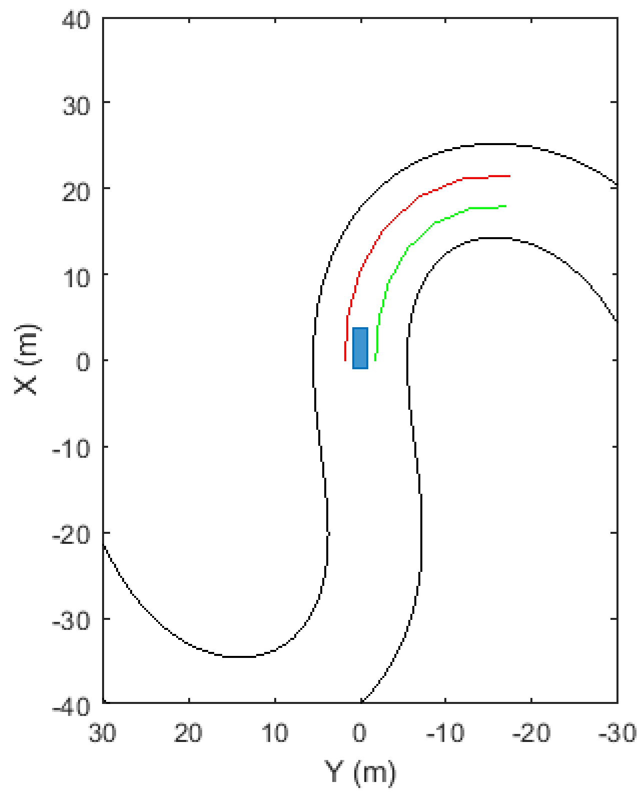
Run the simulation loop.

1. Initialize a bird's-eye plot and create an outline plotter, left-lane and right-lane boundary plotters, and a road boundary plotter.
2. Obtain the road boundaries and rectangular outlines.
3. Obtain the lane boundaries to the left and right of the vehicle.
4. Advance the simulation and update the plotters.

```
bep = birdsEyePlot('XLim',[-40 40],'YLim',[-30 30]);
olPlotter = outlinePlotter(bep);
lblPlotter = laneBoundaryPlotter(bep,'Color','r','LineStyle','-');
lbrPlotter = laneBoundaryPlotter(bep,'Color','g','LineStyle','-');
rbsEdgePlotter = laneBoundaryPlotter(bep);
legend('off');
while advance(scenario)
    rbs = roadBoundaries(car);
    [position,yaw,length,width,originOffset,color] = targetOutlines(car);
    lb = laneBoundaries(car,'XDistance',0:5:30,'LocationType','Center', ...
        'AllBoundaries',false);
    plotLaneBoundary(rbsEdgePlotter,rbs)
    plotLaneBoundary(lblPlotter,{lb(1).Coordinates})
    plotLaneBoundary(lbrPlotter,{lb(2).Coordinates})
    plotOutline(olPlotter,position,yaw,length,width, ...
        'OriginOffset',originOffset,'Color',color)
end
```


40
30
20
10
0
-10
-20
-30
-40
X (m)
40
30
20
10
0
-10
-20
-30
-40
Y (m)





Input Arguments

ac — Actor
`Actor` object | `Vehicle` object

Actor belonging to a `drivingScenario` object, specified as an `Actor` or `Vehicle` object. To create these objects, use the `actor` and `vehicle` functions, respectively.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'LocationType','center'` specifies that lane boundaries are centered on the lane markings.

`XDistance` — Distances from ego vehicle at which to compute lane boundaries
`0` (default) | *N*-element real-valued vector

Distances from the ego vehicle at which to compute the lane boundaries, specified as the comma-separated pair consisting of `'XDistance'` and an *N*-element real-valued vector. *N* is the number of distance values. When detecting lanes from rear-facing cameras, specify negative distances. When detecting lanes from front-facing cameras, specify positive distances. Units are in meters.

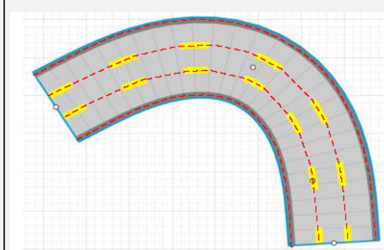
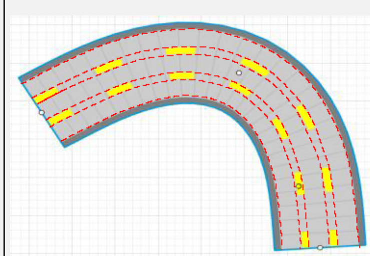
By default, the function computes the lane boundaries at a distance of `0` from the ego vehicle, which are the boundaries to the left and right of the ego-vehicle origin.

Example: `1:0.1:10` computes a lane boundary every 0.1 meters over the range from 1 to 10 meters ahead of the ego vehicle.

Example: `linspace(-150,150,101)` computes 101 lane boundaries over the range from 150 meters behind the ego vehicle to 150 meters ahead of the ego vehicle. These distances are linearly spaced 3 meters apart.

`LocationType` — Lane boundary location
`'Center'` (default) | `'Inner'`

Lane boundary location on the lane markings, specified as the comma-separated pair consisting of `'LocationType'` and one of the options in this table.

Lane Boundary Location	Description	Example
`'Center'`	Lane boundaries are centered on the lane markings.	A three-lane road has four lane boundaries: one per lane marking. 
`'Inner'`	Lane boundaries are placed at the inner edges of the lane markings.	A three-lane road has six lane boundaries: two per lane. 

`AllBoundaries` — Return all lane boundaries on road
`false` (default) | `true`

Return all lane boundaries on which the ego vehicle is traveling, specified as the comma-separated pair consisting of `'Value'` and `false` or `true`.

Lane boundaries are returned from left to right relative to the ego vehicle. When `'AllBoundaries'` is `false`, only the lane boundaries to the left and right of the ego vehicle are returned.

Output Arguments

`lbdry` — Lane boundaries
array of lane boundary structures

Lane boundaries, returned as an array of lane boundary structures. This table shows the fields for each structure.

Field	Description
`Coordinates`	Lane boundary coordinates, specified as a real-valued *N*-by-3 matrix, where *N* is the number of lane boundary coordinates. Lane boundary coordinates define the position of points on the boundary at specified longitudinal distances away from the ego vehicle, along the center of the road. • In MATLAB, specify these distances by using the `'XDistance'` name-value pair argument of the `laneBoundaries` function. • In Simulink, specify these distances by using the **Distances from ego vehicle for computing boundaries (m)** parameter of the Scenario Reader block or the **Distance from parent for computing lane boundaries** parameter of the Simulation 3D Vision Detection Generator block. This matrix also includes the boundary coordinates at zero distance from the ego vehicle. These coordinates are to the left and right of the ego-vehicle origin, which is located under the center of the rear axle. Units are in meters.
`Curvature`	Lane boundary curvature at each row of the `Coordinates` matrix, specified as a real-valued *N*-by-1 vector. *N* is the number of lane boundary coordinates. Units are in radians per meter.
`CurvatureDerivative`	Derivative of lane boundary curvature at each row of the `Coordinates` matrix, specified as a real-valued *N*-by-1 vector. *N* is the number of lane boundary coordinates. Units are in radians per square meter.
`HeadingAngle`	Initial lane boundary heading angle, specified as a real scalar. The heading angle of the lane boundary is relative to the ego vehicle heading. Units are in degrees.
`LateralOffset`	Distance of the lane boundary from the ego vehicle position, specified as a real scalar. An offset to a lane boundary to the left of the ego vehicle is positive. An offset to the right of the ego vehicle is negative. Units are in meters.

BoundaryType	Type of lane boundary marking, specified as one of these values: • 'Unmarked' — No physical lane marker exists • 'Solid' — Single unbroken line • 'Dashed' — Single line of dashed lane markers • 'DoubleSolid' — Two unbroken lines • 'DoubleDashed' — Two dashed lines • 'SolidDashed' — Solid line on the left and a dashed line on the right • 'DashedSolid' — Dashed line on the left and a solid line on the right
Strength	Saturation strength of the lane boundary marking, specified as a real scalar from 0 to 1. A value of 0 corresponds to a marking whose color is fully unsaturated. The marking is gray. A value of 1 corresponds to a marking whose color is fully saturated.
Width	Lane boundary width, specified as a positive real scalar. In a double-line lane marker, the same width is used for both lines and for the space between lines. Units are in meters.
Length	Length of dash in dashed lines, specified as a positive real scalar. In a double-line lane marker, the same length is used for both lines.
Space	Length of space between dashes in dashed lines, specified as a positive real scalar. In a dashed double-line lane marker, the same space is used for both lines.

See Also

Objects
drivingScenario | lanespec

Functions
laneBoundaryPlotter | laneMarking | laneMarkingPlotter | plotLaneBoundary | plotLaneMarking | road

Introduced in R2018a

clothoidLaneBoundary

Clothoid-shaped lane boundary model

Description

A `clothoidLaneBoundary` object contains information about a clothoid-shaped lane boundary model. A clothoid is a type of curve whose rate of change of curvature varies linearly with distance.

Creation

Syntax

```
bdry = clothoidLaneBoundary
bdry = clothoidLaneBoundary(Name,Value)
```

Description

`bdry = clothoidLaneBoundary` creates a clothoid lane boundary model, `bdry` with default property values.

`bdry = clothoidLaneBoundary(Name,Value)` sets properties using one or more name-value pairs. For example, `clothoidLaneBoundary('BoundaryType','Solid')` creates a clothoid lane boundary model with solid lane boundaries. Enclose each property name in quotes.

Properties

`Curvature` — Lane boundary curvature
`0` (default) | real scalar

Lane boundary curvature, specified as a real scalar. This property represents the rate of change of lane boundary direction with respect to distance. Units are in degrees per meter.

Example: `-1.0`

Data Types: `single` | `double`

`CurvatureDerivative` — Derivative of lane boundary curvature
`0` (default) | real scalar

Derivative of lane boundary curvature, specified as a real scalar. This property represents the rate of change of lane curvature with respect to distance. Units are in degrees per meter squared.

Example: `-0.01`

Data Types: `single` | `double`

`CurveLength` — Length of lane boundary along road
`0` (default) | nonnegative real scalar

Length of the lane boundary along the road, specified as a nonnegative real scalar. Units are in meters.

Example: 25

Data Types: single | double

HeadingAngle — Initial lane boundary heading
0 (default) | real scalar

Initial lane boundary heading, specified as a real scalar. The heading angle of the lane boundary is relative to the heading of the ego vehicle. Units are in degrees.

Example: 10

Data Types: single | double

LateralOffset — Distance of lane boundary
0 (default) | real scalar

Distance of the lane boundary from the ego vehicle position, specified as a real scalar. A lane boundary offset to the left of the ego vehicle is positive. An offset to the right of the ego vehicle is negative. Units are in meters.

Example: -1.2

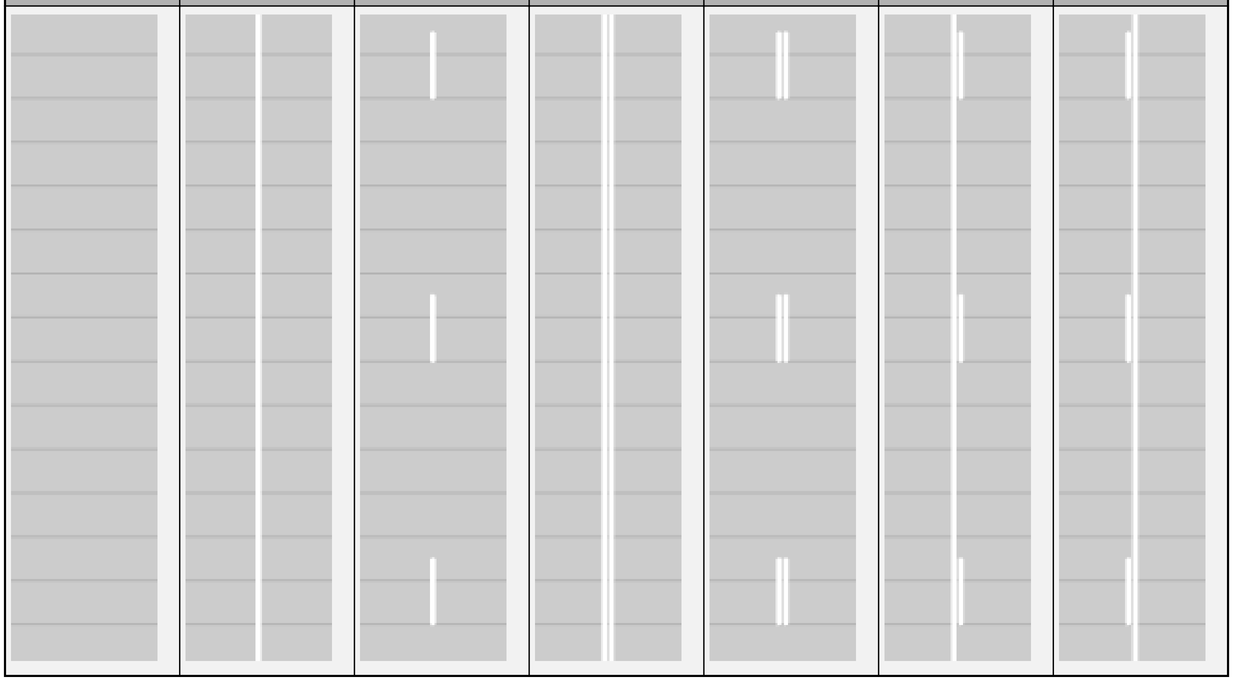
Data Types: single | double

BoundaryType — Type of lane boundary marking
'Unmarked' (default) | 'Solid' | 'Dashed' | 'DoubleSolid' | 'DoubleDashed' | 'SolidDashed' | 'DashedSolid'

Type of lane boundary marking, specified as one of these values.

'Unmarked'	'Solid'	'Dashed'	'DoubleSolid'	'DoubleDashed'	'SolidDashed'	'DashedSolid'
No lane marking	Solid line	Dashed line	Two solid lines	Two dashed lines	Solid line on left, dashed line on right	Dashed line on left, solid line on right

'Unmarked'	'Solid'	'Dashed'	'DoubleSolid'	'DoubleDashed'	'SolidDashed'	'DashedSolid'



`Strength` — Visibility of lane boundary marking

`1` (default) | real scalar in the range [0, 1]

Visibility of lane marking, specified as a real scalar in the range [0, 1]. A value of `0` corresponds to a marking that is not visible. A value of `1` corresponds to a marking that is completely visible. For a double lane marking, the same strength is used for both lines.

Example: `0.9`

Data Types: `single` | `double`

`XExtent` — Extent of lane boundary marking along *X*-axis

`[0 Inf]` (default) | real-valued vector of the form [X_{min} X_{max}]

Extent of the lane boundary marking along the *X*-axis, specified as a real-valued vector of the form [X_{min} X_{max}]. Units are in meters. The *X*-axis runs vertically and is positive in the forward direction of the ego vehicle.

Example: `[0 100]`

Data Types: `single` | `double`

`Width` — Width of lane boundary marking

`0` (default) | nonnegative real scalar

Width of lane boundary marking, specified as a nonnegative real scalar. For a double lane marking, this value applies to the width of each lane marking and to the distance between those markings. Units are in meters.

Example: `0.15`

Data Types: `single` | `double`

Object Functions

computeBoundaryModel Compute lane boundary points from clothoid lane boundary model

Examples

Create Clothoid Lane Boundaries

Create clothoid curves to represent left and right lane boundaries. Then, plot the curves.

Create the left boundary.

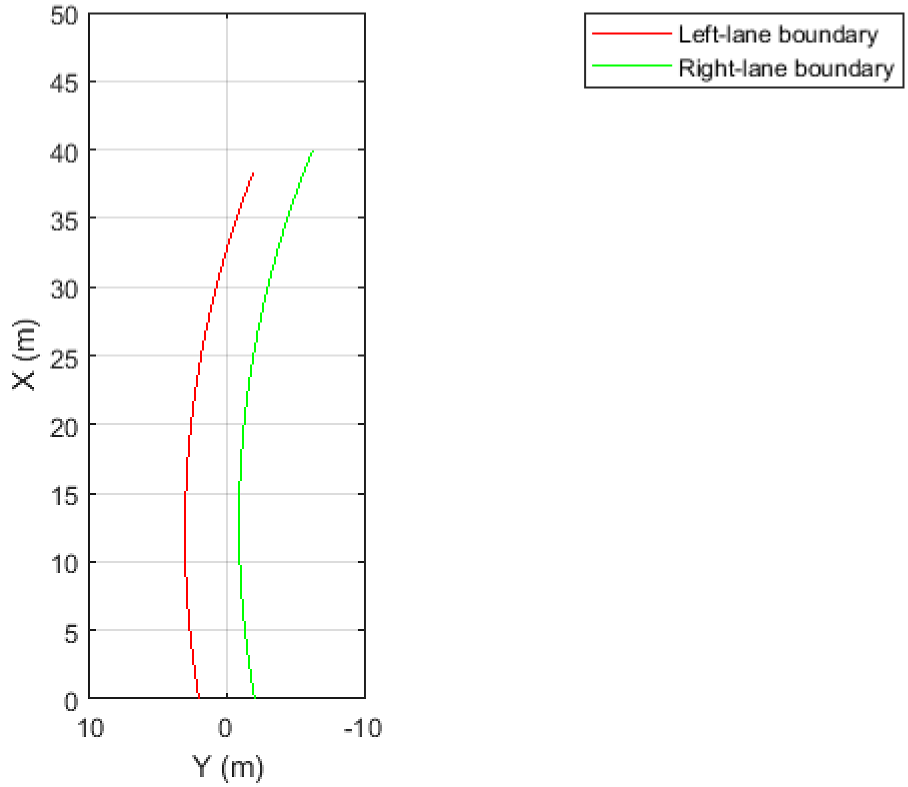
```
lb = clothoidLaneBoundary('BoundaryType','Solid', ...
'Strength',1,'Width',0.2,'CurveLength',40, ...
'Curvature',-0.8,'LateralOffset',2,'HeadingAngle',10);
```

Create the right boundary with almost identical properties.

```
rb = lb;
rb.LateralOffset = -2;
```

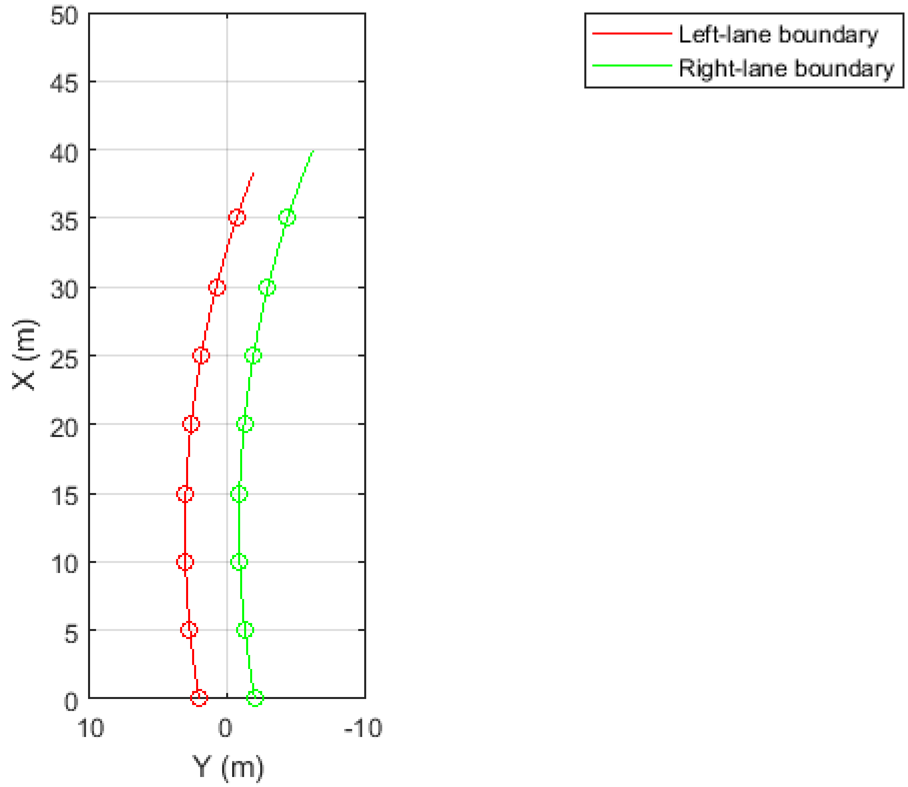
Create a bird's-eye plot. Then, create the lane boundary plotters and plot the boundaries.

```
bep = birdsEyePlot('XLimits',[0 50],'YLimits',[-10 10]);
lbPlotter = laneBoundaryPlotter(bep,'DisplayName','Left-lane boundary','Color','r');
rbPlotter = laneBoundaryPlotter(bep,'DisplayName','Right-lane boundary','Color','g');
plotLaneBoundary(lbPlotter,lb)
plotLaneBoundary(rbPlotter,rb);
grid
hold on
```



Plot the coordinates of selected points along the boundaries.

```
x = 0:5:50;
yl = computeBoundaryModel(lb,x);
yr = computeBoundaryModel(rb,x);
plot(x,yl,'ro')
plot(x,yr,'go')
hold off
```



See Also

Objects
lanespec

Functions
laneBoundaries | laneBoundaryPlotter | laneMarking | plotLaneBoundary

Introduced in R2018a

computeBoundaryModel

Compute lane boundary points from clothoid lane boundary model

Syntax

```
yworld = computeBoundaryModel(boundary,xworld)
```

Description

`yworld = computeBoundaryModel(boundary,xworld)` returns the *y*-coordinates of lane boundary points, `yworld`, derived from a lane boundary, `boundary`, at points specified by the *x*-coordinates, `xworld`. All points are in world coordinates.

Examples

Create Clothoid Lane Boundaries

Create clothoid curves to represent left and right lane boundaries. Then, plot the curves.

Create the left boundary.

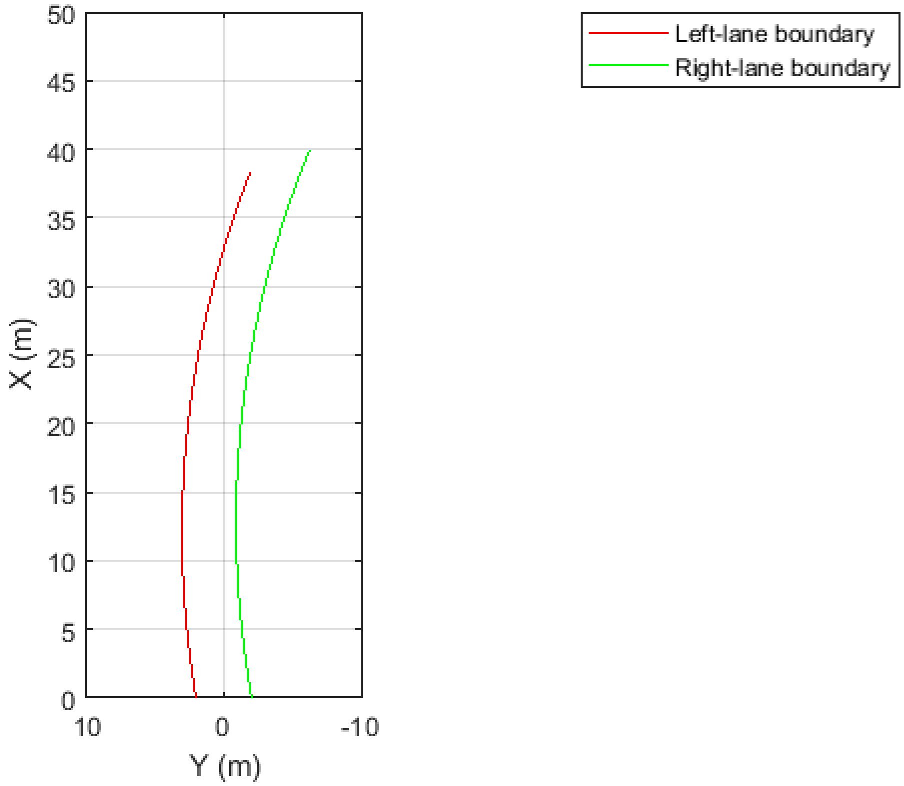
```
lb = clothoidLaneBoundary('BoundaryType','Solid', ...
'Strength',1,'Width',0.2,'CurveLength',40, ...
'Curvature',-0.8,'LateralOffset',2,'HeadingAngle',10);
```

Create the right boundary with almost identical properties.

```
rb = lb;
rb.LateralOffset = -2;
```

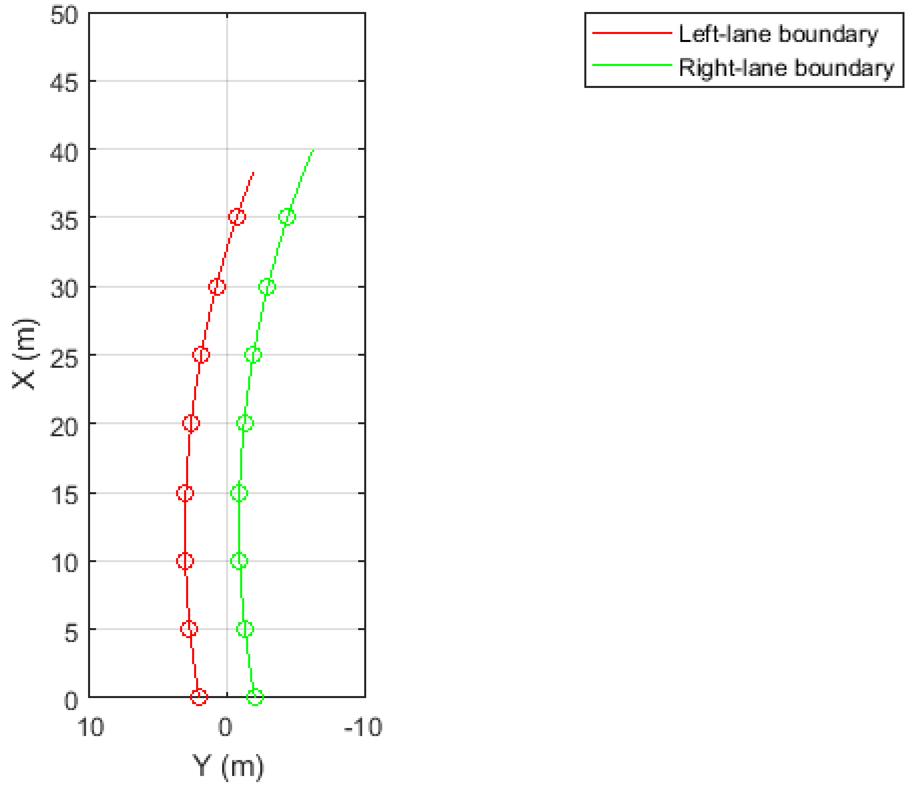
Create a bird's-eye plot. Then, create the lane boundary plotters and plot the boundaries.

```
bep = birdsEyePlot('XLimits',[0 50],'YLimits',[-10 10]);
lbPlotter = laneBoundaryPlotter(bep,'DisplayName','Left-lane boundary','Color','r');
rbPlotter = laneBoundaryPlotter(bep,'DisplayName','Right-lane boundary','Color','g');
plotLaneBoundary(lbPlotter,lb)
plotLaneBoundary(rbPlotter,rb);
grid
hold on
```



Plot the coordinates of selected points along the boundaries.

```
x = 0:5:50;
yl = computeBoundaryModel(lb,x);
yr = computeBoundaryModel(rb,x);
plot(x,yl,'ro')
plot(x,yr,'go')
hold off
```



Input Arguments

boundary — Lane boundary model
`clothoidLaneBoundary` object

Lane boundary model, specified as a `clothoidLaneBoundary` object.

xworld — *x*-world coordinates
real-valued vector of length *N*

x-world coordinates, specified as a real-valued vector of length *N*, where *N* is the number of coordinates.

Example: `2:2.5:100`

Data Types: `single` | `double`

Output Arguments

yworld — *y*-world coordinates
real-valued vector of length *N*

y-world coordinates, returned as a real-valued vector of length *N*, where *N* is the number of coordinates. The length and data type of `yWorld` are the same as for `xWorld`.

Data Types: `single` | `double`

See Also

`clothoidLaneBoundary` | `laneBoundaries`

Introduced in R2018a

driving.scenario.bicycleMesh

Mesh representation of bicycle in driving scenario

Syntax

`mesh = driving.scenario.bicycleMesh`

Description

`mesh = driving.scenario.bicycleMesh` creates a mesh representation of a bicycle as an `extendedObjectMesh` object, `mesh`.

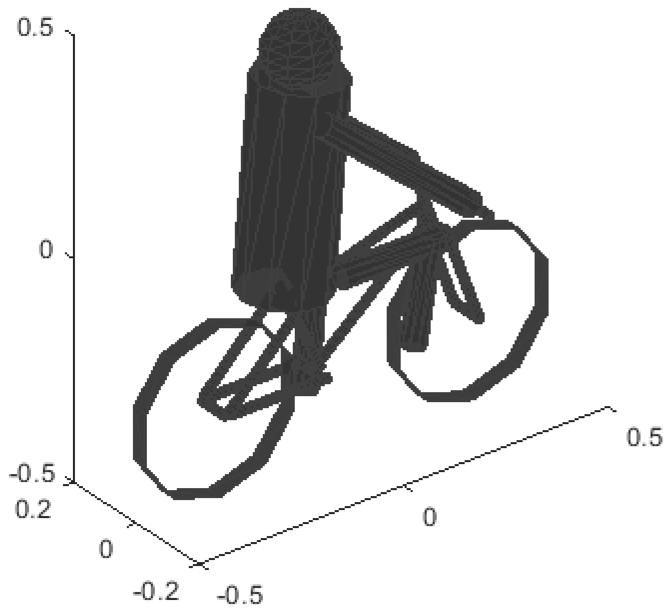
Examples

Generate Lidar Point Cloud by Using Bicycle Mesh

Add a prebuilt bicycle mesh to a driving scenario. Then, use a `lidarPointCloudGenerator` System object to generate a point cloud of the bicycle mesh.

Create and show the prebuilt bicycle mesh.

```
mesh = driving.scenario.bicycleMesh;
egoMesh = driving.scenario.carMesh;
figure
show(mesh)
```




```
ans = 
  Axes with properties:

             XLim: [-0.5000 0.5000]
             YLim: [-0.2000 0.2000]
           XScale: 'linear'
           YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.1100 0.7750 0.8150]
            Units: 'normalized'

  Show all properties
```

Create a driving scenario.

```
s = drivingScenario;
```

Add a straight road to the driving scenario. The road has one lane traveling in each direction.

```
road(s,[0 0 0; 30 0 0],'Lanes',lanespec([1 1]));
```

Add a car as an ego vehicle and a bicycle as a non-ego actor.

```
egoVehicle = vehicle(s,'ClassID',1,'Mesh',egoMesh);
bicycle = vehicle(s,'Position',[15 2 0],'Yaw',180,'ClassID',3,'Mesh',mesh);
trajectory(egoVehicle,[1 -2 0; 21.3 -2 0],20);
```

Plot the driving scenario. Set name-value pair `'Meshes','on'` to show the meshes of the actors in the plot.

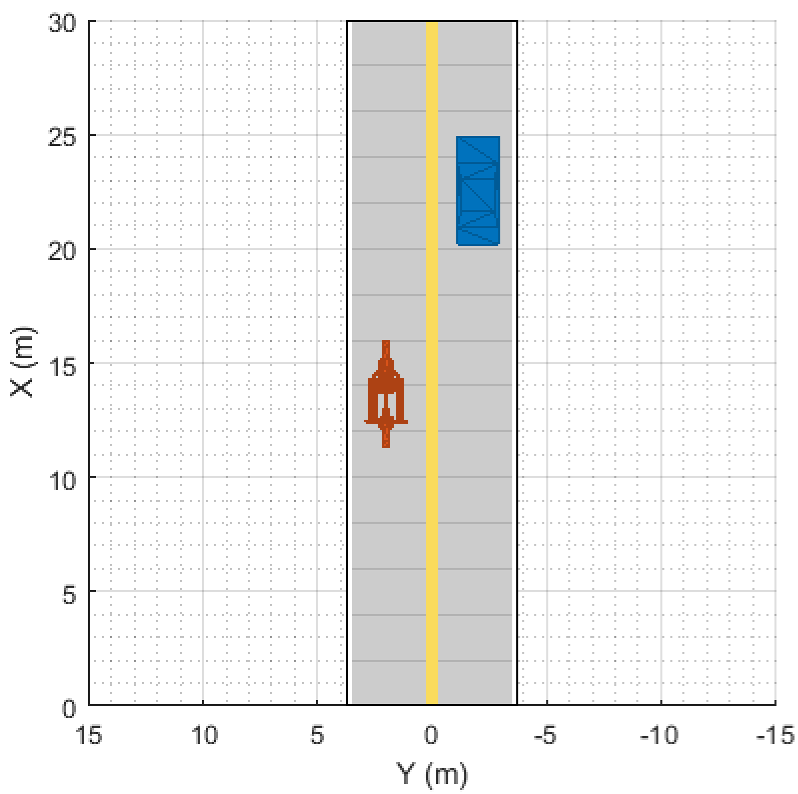
```
figure;
plot(s,'Meshes','on');
```

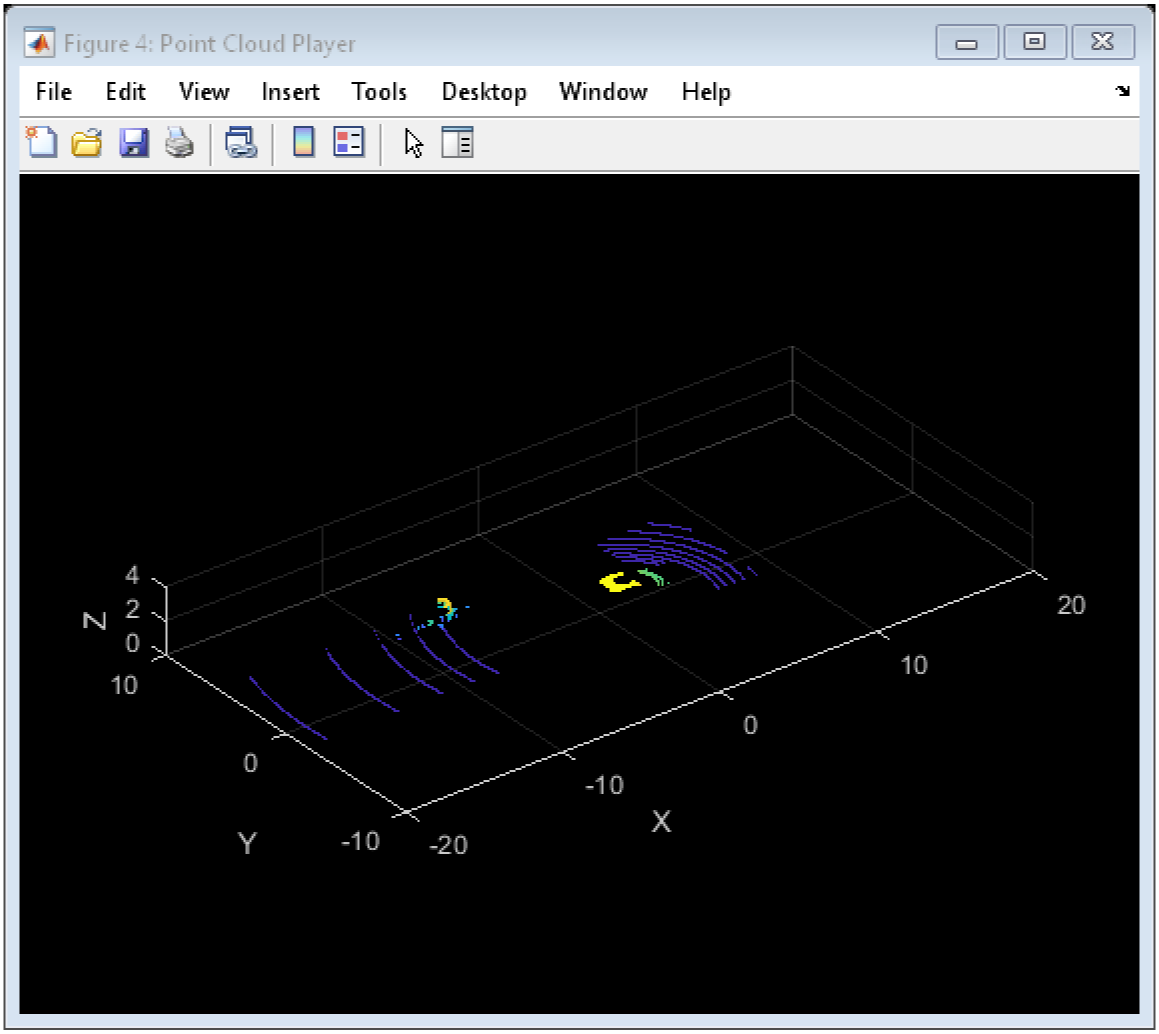
Create a `lidarPointCloudGenerator` System object. Set the actor profiles of the System object to those in the driving scenario.

```
lidar = lidarPointCloudGenerator;
lidar.ActorProfiles = actorProfiles(s);
```

Generate a lidar point cloud of the driving scenario.

```
player = pcplayer([-20 20],[-10 10],[0 4]);
while advance(s)
 tgts = targetPoses(egoVehicle);
 rdmesh = roadMesh(egoVehicle);
 [ptCloud,isValidTime] = lidar(tgts,rdmesh,s.SimulationTime);
    if isValidTime
    view(player,ptCloud);
    end
end
```





Output Arguments

mesh — Mesh representation of bicycle
`extendedObjectMesh` object

Mesh representation of bicycle, returned as an `extendedObjectMesh` object. The origin of the mesh is located at its geometric center.

You can develop your own meshes by using this prebuilt bicycle mesh as a starting point. At the MATLAB command line, enter:

```
edit driving.scenario.bicycleMesh
```

See Also

Objects
`extendedObjectMesh`

Functions
`applyTransform` | `driving.scenario.carMesh` | `driving.scenario.pedestrianMesh` | `driving.scenario.truckMesh` | `join` | `rotate` | `scale` | `scaleToFit` | `show` | `translate`

Introduced in R2020a

driving.scenario.carMesh

Mesh representation of car in driving scenario

Syntax

```
mesh = driving.scenario.carMesh
```

Description

`mesh = driving.scenario.carMesh` creates a mesh representation of a car as an `extendedObjectMesh` object, `mesh`.

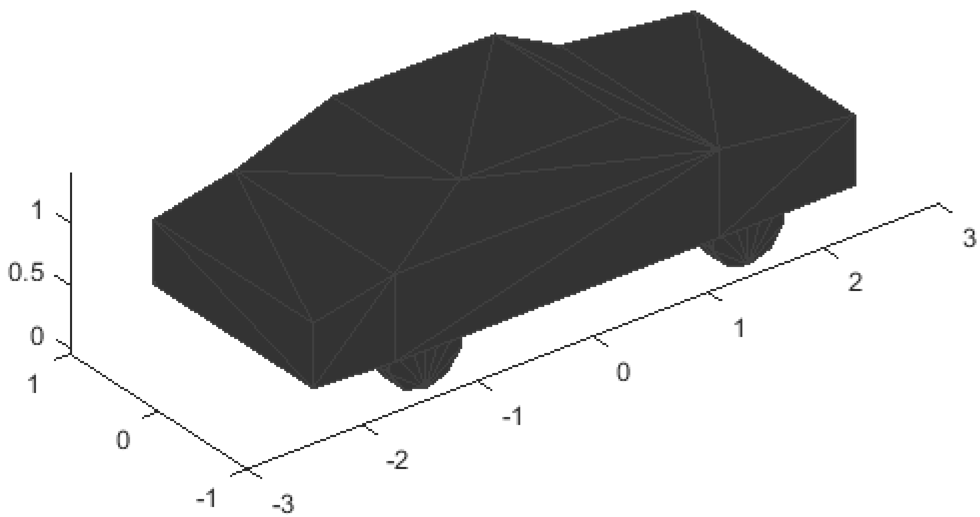
Examples

Generate Lidar Point Cloud by Using Car Mesh

Add the prebuilt car mesh to a driving scenario. Then, use `lidarPointCloudGenerator` System object to generate a point cloud of the car mesh.

Create and show the prebuilt car mesh.

```
mesh = driving.scenario.carMesh;
show(mesh);
```



Create a driving scenario.

```
s = drivingScenario;
```

Add a straight road to the driving scenario. The road has one lane in each direction.

```
road(s,[0 0 0; 25 0 0],'Lanes',lanespec([1 1]));
```

Add a car as an ego vehicle and as a non-ego actor.

```
egoVehicle = vehicle(s,'ClassID',1,'Mesh',mesh);
trajectory(egoVehicle,[1 -2 0; 21.3 -2 0],20);
car = vehicle(s,'Position',[15 2 0],'Yaw',180,'ClassID',1,'Mesh',mesh);
```

Plot the driving scenario. Set name-value pair `'Meshes','on'` to show the meshes of the actors in the plot.

```
plot(s,'Meshes','on');
```

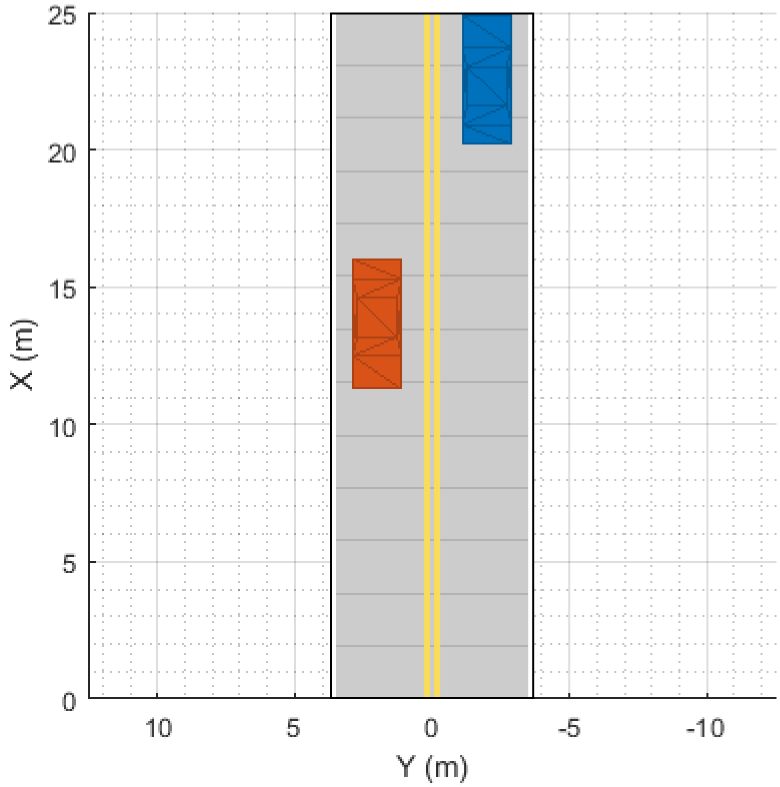
Create a `lidarPointCloudGenerator` System object. Set the actor profiles of the System object to those in the driving scenario.

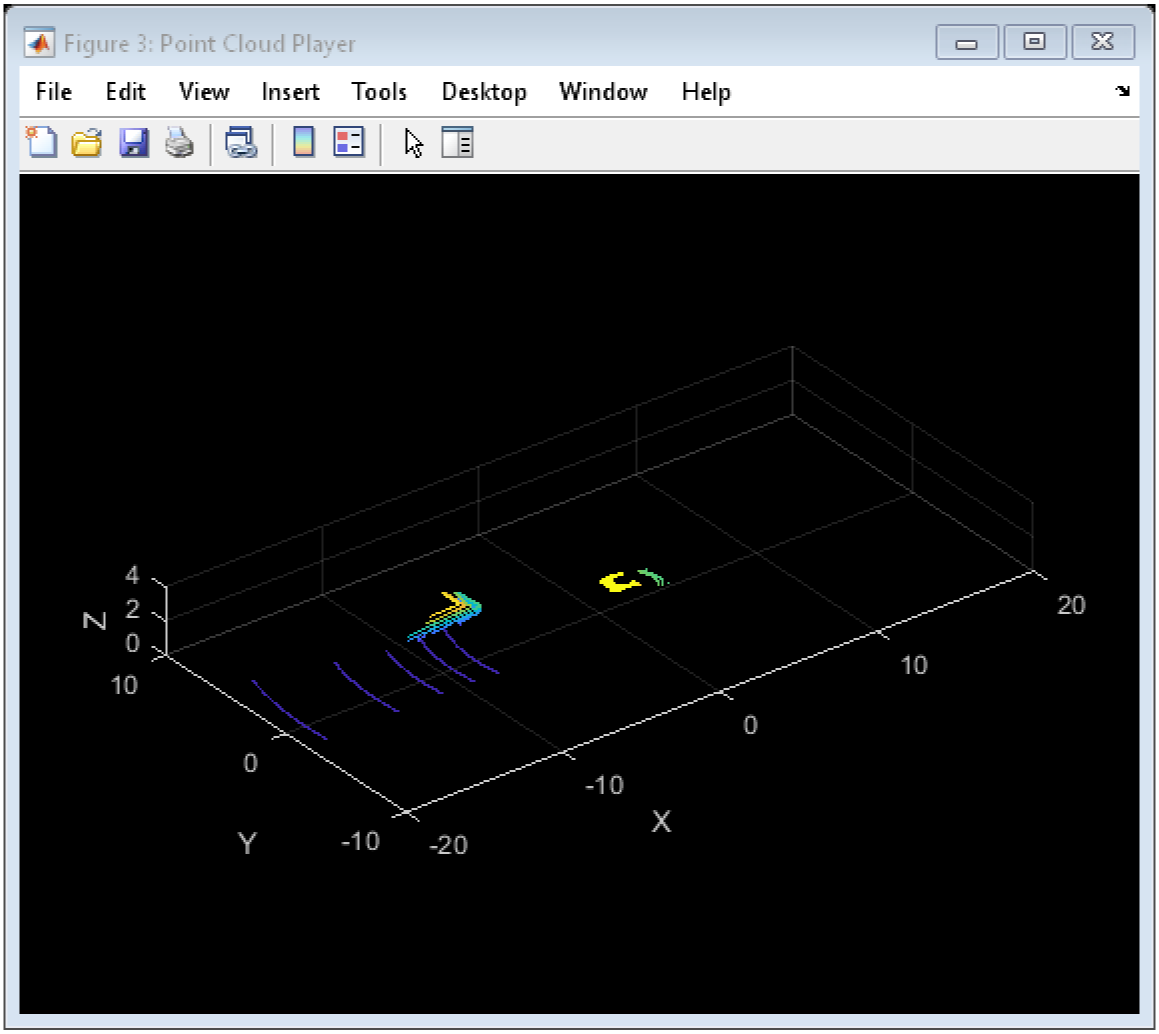
```
lidar = lidarPointCloudGenerator;
lidar.ActorProfiles = actorProfiles(s);
```

Generate a lidar point cloud of the driving scenario.

```
player = pcplayer([-20 20],[-10 10],[0 4]);
 while advance(s)
```

```
tgts = targetPoses(egoVehicle);
rdmesh = roadMesh(egoVehicle);
[ptCloud,isValidTime] = lidar(tgts,rdmesh,s.SimulationTime);
   if isValidTime
   view(player,ptCloud);
   end
end
```





Output Arguments

mesh — Mesh representation of car
`extendedObjectMesh` object

Mesh representation of car, returned as an `extendedObjectMesh` object. The origin of the mesh is located at its geometric center.

You can develop your own meshes by using this prebuilt car mesh as a starting point. At the MATLAB command line, enter:

```
edit driving.scenario.carMesh
```

See Also

Objects
`extendedObjectMesh`

Functions
`applyTransform` | `driving.scenario.bicycleMesh` | `driving.scenario.pedestrianMesh` | `driving.scenario.truckMesh` | `join` | `rotate` | `scale` | `scaleToFit` | `show` | `translate`

Introduced in R2020a

driving.scenario.pedestrianMesh

Mesh representation of pedestrian in driving scenario

Syntax

`mesh = driving.scenario.pedestrianMesh`

Description

`mesh = driving.scenario.pedestrianMesh` creates a mesh representation of a pedestrian as an `extendedObjectMesh` object, `mesh`.

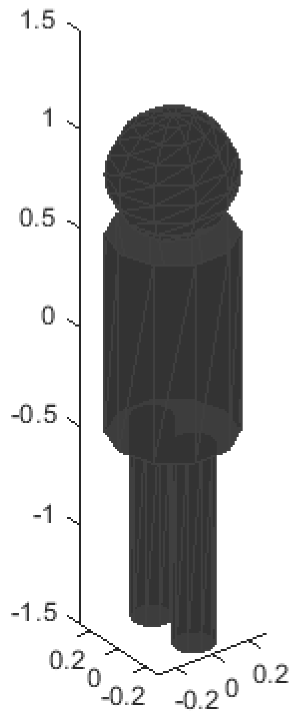
Examples

Generate Lidar Point Cloud by Using Pedestrian Mesh

Add the prebuilt pedestrian mesh to a driving scenario. Then, use `lidarPointCloudGenerator` System object to generate a point cloud of the pedestrian mesh.

Create and show the prebuilt pedestrian mesh.

```
mesh = driving.scenario.pedestrianMesh;
egoMesh = driving.scenario.carMesh;
show(mesh);
```



Create a driving scenario.

```
s = drivingScenario;
```

Add a straight road to the driving scenario. The road has one lane in each direction.

```
road(s,[0 0 0; 30 0 0],'Lanes',lanespec([1 1]));
```

Add a car as an ego vehicle and a pedestrian as a non-ego actor.

```
egoVehicle = vehicle(s,'ClassID',1,'Mesh',egoMesh);
trajectory(egoVehicle,[1 -2 0; 21.3 -2 0],20);
pedestrian = actor(s,'Length',0.24,'Width',0.45,'Height',1.7,'Position',[15 2 0],'ClassID',4,'Mes
```

Plot the driving scenario. Set name-value pair `'Meshes','on'` to show the meshes of the actors in the plot.

```
plot(s,'Meshes','on');
```

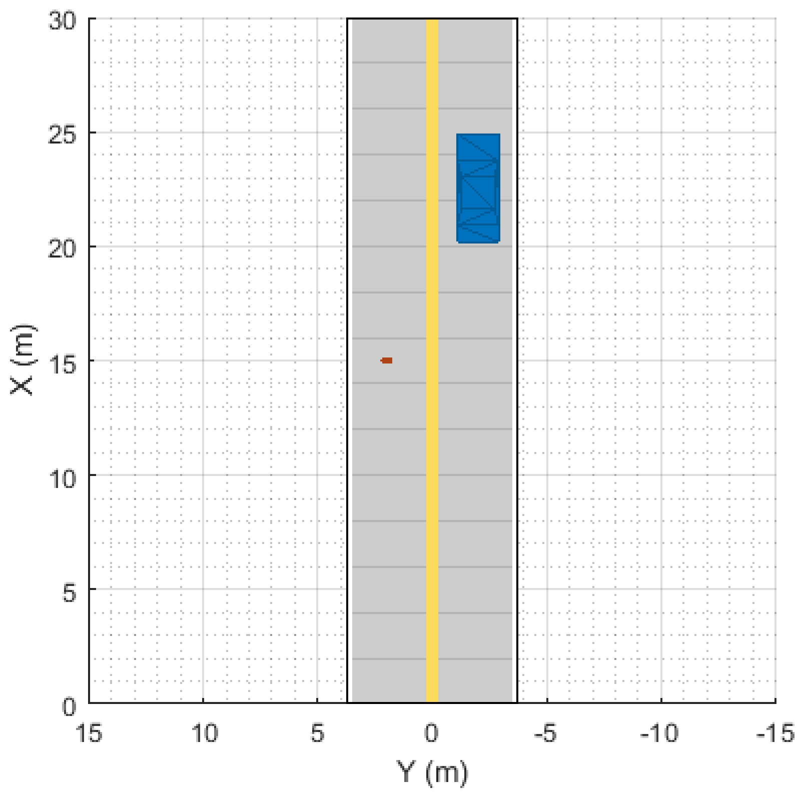
Create a `lidarPointCloudGenerator` System object. Set the actor profiles of the System object to those in the driving scenario.

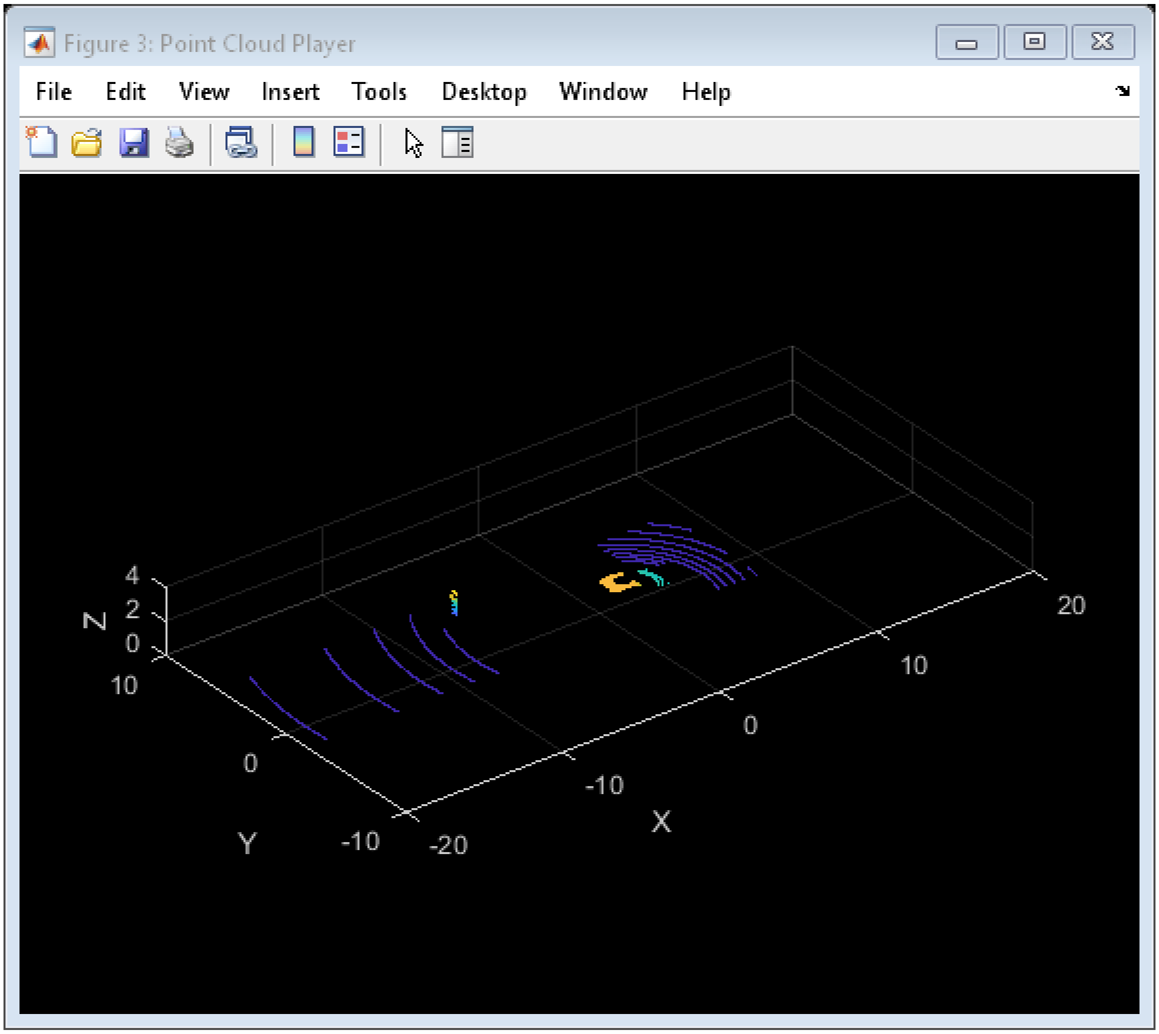
```
lidar = lidarPointCloudGenerator;
lidar.ActorProfiles = actorProfiles(s);
```

Generate a lidar point cloud of the driving scenario.

```
player = pcplayer([-20 20],[-10 10],[0 4]);
while advance(s)
```

```
    tgts = targetPoses(egoVehicle);
    rdmesh = roadMesh(egoVehicle);
    [ptCloud,isValidTime] = lidar(tgts,rdmesh,s.SimulationTime);
       if isValidTime
       view(player,ptCloud);
       end
end
```





Output Arguments

mesh — Mesh representation of pedestrian
extendedObjectMesh object

Mesh representation of pedestrian, returned as an extendedObjectMesh object. The origin of the mesh is located at its geometric center.

You can develop your own meshes by using this prebuilt pedestrian mesh as a starting point. At the MATLAB command line, enter:

```
edit driving.scenario.pedestrianMesh
```

See Also

Objects
extendedObjectMesh

Functions
`applyTransform` | `driving.scenario.bicycleMesh` | `driving.scenario.carMesh` | `driving.scenario.truckMesh` | `join` | `rotate` | `scale` | `scaleToFit` | `show` | `translate`

Introduced in R2020a

driving.scenario.truckMesh

Mesh representation of truck in driving scenario

Syntax

`mesh = driving.scenario.truckMesh`

Description

`mesh = driving.scenario.truckMesh` creates a mesh representation of a truck as an `extendedObjectMesh` object, `mesh`.

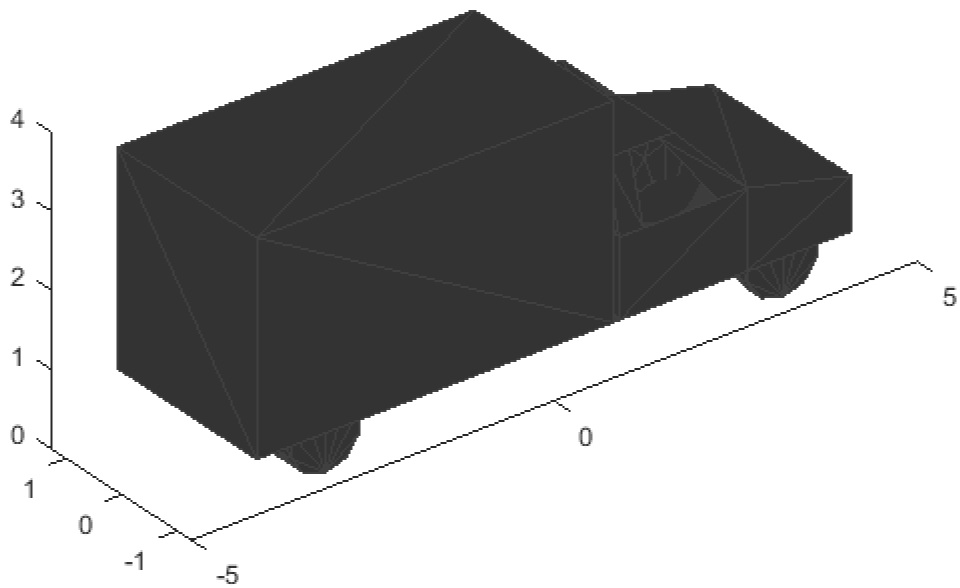
Examples

Generate Lidar Point Cloud by Using Truck Mesh

Add the prebuilt truck mesh to a driving scenario to generate a point cloud. Then, use `lidarPointCloudGenerator` System object to generate a point cloud of the truck mesh.

Create and show the prebuilt truck mesh.

```
mesh = driving.scenario.truckMesh;
egoMesh = driving.scenario.carMesh;
show(mesh);
```




```
pause(1);
```

Create a driving scenario.

```
s = drivingScenario;
```

Add a straight road to the driving scenario. The road has one lane in each direction.

```
road(s,[0 0 0; 30 0 0],'Lanes',lanespec([1 1]));
```

Add a car as the ego vehicle and a truck as a non-ego actor.

```
egoVehicle = vehicle(s,'ClassID',1,'Mesh',egoMesh);
trajectory(egoVehicle,[1 -2 0; 21.3 -2 0],20);
truck = vehicle(s,'Position',[15 2 0],'Yaw',180,'ClassID',2,'Mesh',mesh);
```

Plot the driving scenario. Set name-value pair `'Meshes','on'` to show the meshes of the actors in the plot.

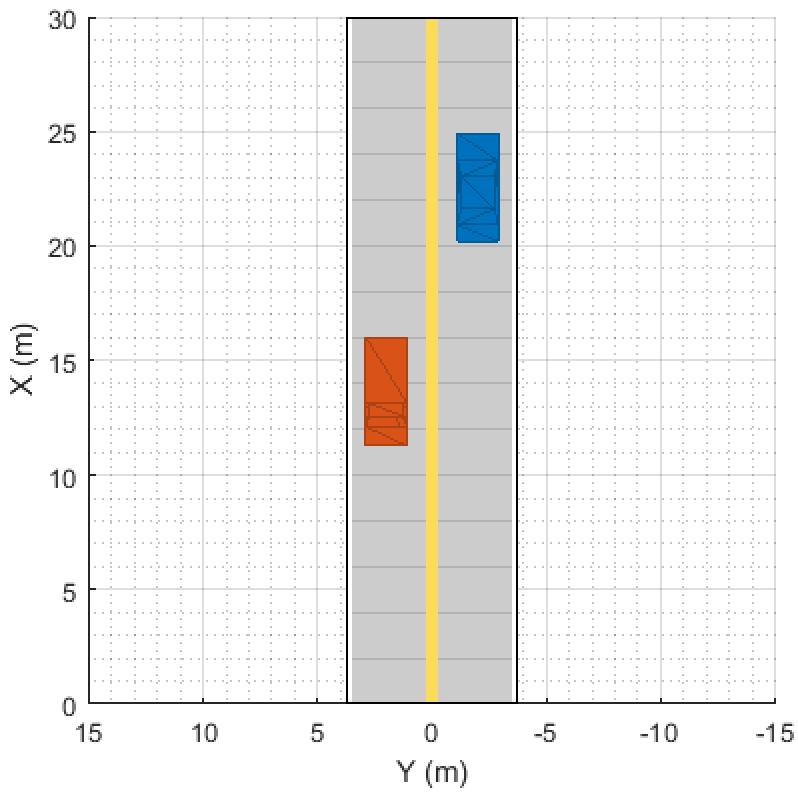
```
plot(s,'Meshes','on');
```

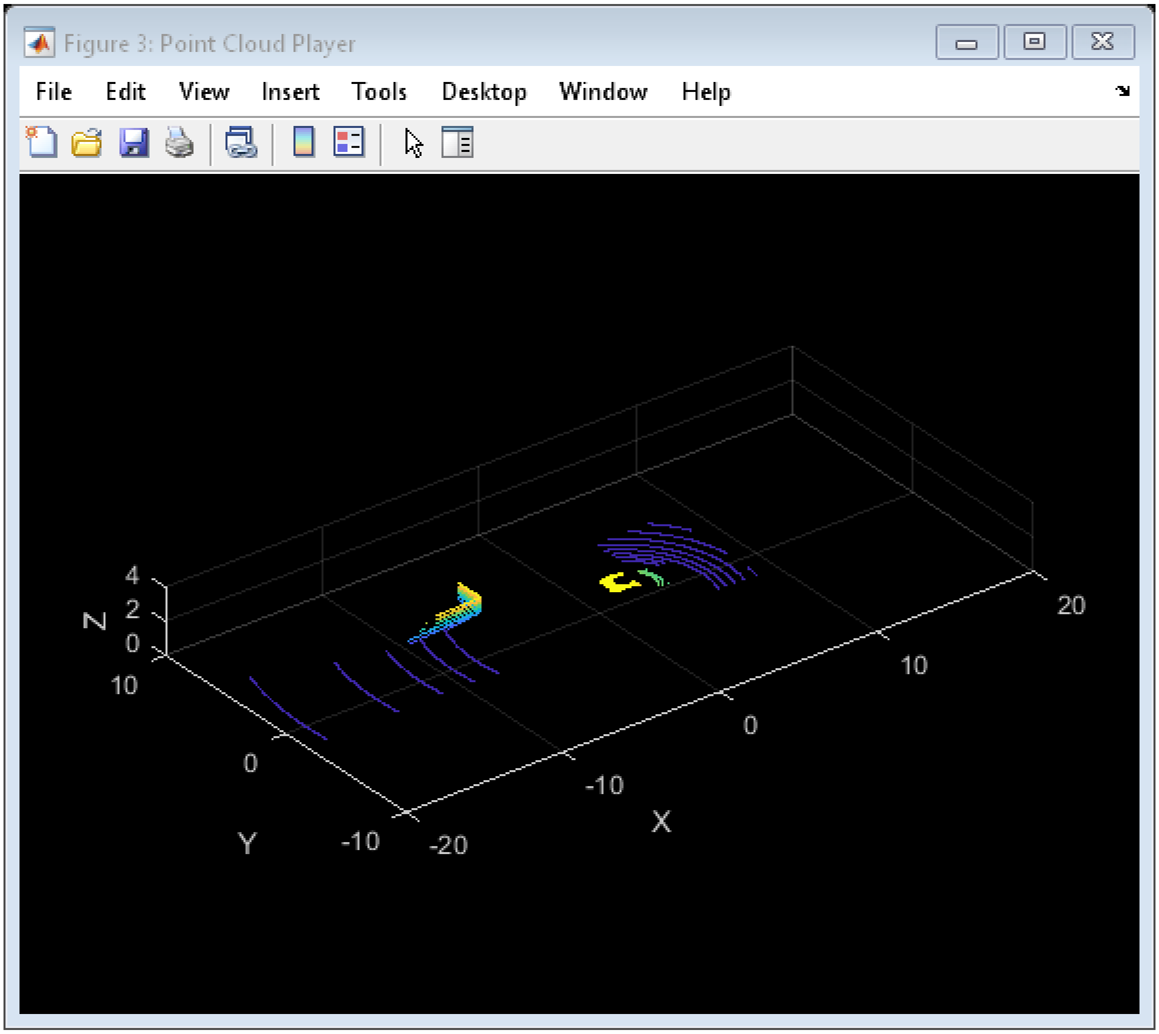
Create a `lidarPointCloudGenerator` System object. Set the actor profiles of the System object to those in the driving scenario.

```
lidar = lidarPointCloudGenerator;
lidar.ActorProfiles = actorProfiles(s);
```

Generate a lidar point cloud of the driving scenario.

```
player = pcplayer([-20 20],[-10 10],[0 4]);
while advance(s)
 tgts = targetPoses(egoVehicle);
 rdmesh = roadMesh(egoVehicle);
 [ptCloud,isValidTime] = lidar(tgts,rdmesh,s.SimulationTime);
    if isValidTime
    view(player,ptCloud);
    end
end
```





Output Arguments

mesh — Mesh representation of truck
`extendedObjectMesh` object

Mesh representation of truck, returned as an `extendedObjectMesh` object. The origin of the mesh is located at its geometric center.

You can develop your own meshes by using this prebuilt truck mesh as a starting point. At the MATLAB command line, enter:

```
edit driving.scenario.truckMesh
```

See Also

Objects
`extendedObjectMesh`

Functions
`applyTransform` | `driving.scenario.bicycleMesh` | `driving.scenario.carMesh` | `driving.scenario.pedestrianMesh` | `join` | `rotate` | `scale` | `scaleToFit` | `show` | `translate`

Introduced in R2020a

groundTruthMultisignal

Ground truth label data for multiple signals

Description

The groundTruthMultisignal object contains information about the ground truth data source, label definitions, and marked label annotations for multiple signals. The source of the signals can be a video, image sequence, lidar point cloud, or any other custom format containing multiple signals. You can export or import a groundTruthMultisignal object from the **Ground Truth Labeler** app.

To create training data for deep learning applications from arrays of groundTruthMultisignal objects, use the gatherLabelData function.

Creation

To export a groundTruthMultisignal object from the **Ground Truth Labeler** app, on the app toolstrip, select **Export Labels** > **To Workspace**. The app exports the object to the MATLAB workspace. To create a groundTruthMultisignal object programmatically, use the groundTruthMultisignal function (described here).

Syntax

gTruth = groundTruthMultisignal(dataSources,labelDefs,roiData,sceneData)

Description

gTruth = groundTruthMultisignal(dataSources,labelDefs,roiData,sceneData) returns an object containing ground truth labels that can be imported into the **Ground Truth Labeler** app.

- dataSources specifies the sources of the ground truth data and sets the DataSource property.
- labelDefs specifies the label, sublabel, and attribute definitions of the ground truth data and sets the LabelDefinitions property.
- roiData specifies the identifying information, position, and timestamps for the marked region of interest (ROI) labels and sets the ROILabelData property.
- sceneData specifies the identifying information and timestamps for marked scene labels and sets the SceneLabelData property.

Properties

DataSource — Sources of ground truth data
vector of MultiSignalSource objects

Sources of ground truth data, specified as a vector of MultiSignalSource objects. These objects contain information that describe the sources from which ground truth data was labeled. This table describes the type of MultiSignalSource objects that you can specify in this vector.

MultiSignalSource Object Type	Data Source	Class Reference
VideoSource	Video file	vision.labeler.loading.VideoSource
ImageSequenceSource	Image sequence folder	vision.labeler.loading.ImageSequenceSource
VelodyneLidarSource	Velodyne® packet capture (PCAP) file	vision.labeler.loading.VelodyneLidarSource
RosbagSource	Rosbag file	vision.labeler.loading.RosbagSource
PointCloudSequenceSource	Point cloud sequence folder	vision.labeler.loading.PointCloudSequenceSource
CustomImageSource	Custom image format	vision.labeler.loading.CustomImageSource

To specify additional data sources, create a new type of `MultiSignalSource` object by using the `vision.labeler.loading.MultiSignalSource` class.

LabelDefinitions — Label definitions
table

Label definitions, specified as a table. To create this table, use one of these options.

- In the **Ground Truth Labeler** app, create label definitions, and then export them as part of a `groundTruthMultisignal` object.
- Use a `labelDefinitionCreatorMultisignal` object to generate a label definitions table. If you save this table to a MAT-file, you can then load the label definitions into a **Ground Truth Labeler** app session by selecting **Open** > **Label Definitions** from the app toolstrip.
- Create the label definitions table at the MATLAB command line.

This table describes the required and optional columns of the table specified in the `LabelDefinitions` property.

Column	Description	Required or Optional
Name	Strings or character vectors specifying the name of each label definition.	Required

<table>
<tr><th>Column</th><th>Description</th><th>Required or Optional</th></tr>
<tr><td>SignalType</td><td> SignalType enumerations that specify the signal type supported for each label definition. Valid values are Image for image signals such as videos or image sequences, PointCloud for lidar signals, or Time for scene label definitions.
If a label definition supports multiple signal types, then the label definition has a separate row for each signal type. For example, consider a label definition named car. In the Ground Truth Labeler app, you draw this label as a rectangle in image signals and a cuboid in lidar point cloud signals. In the LabelDefinitions table, car appears twice and has these Name, SignalType, and LabelType values.
<table>
<tr><th>Name</th><th>SignalType</th><th>LabelType</th></tr>
<tr><td>'car'</td><td>Image</td><td>Rectangle</td></tr>
<tr><td>'car'</td><td>PointCloud</td><td>Cuboid</td></tr>
</table></td><td>Required</td></tr>
<tr><td>LabelType</td><td> labelType enumerations that specify the type of each label definition.
For ROI label definitions with a SignalType of Image, valid LabelType enumerations are:

labelType.Rectangle
labelType.Line
labelType.PixelLabel
labelType.ProjectedCuboid

For ROI label definitions with a SignalType of PointCloud, the only valid LabelType enumeration is labelType.Cuboid.
For scene label definitions, the only valid LabelType enumeration is labelType.Scene.
You can also add ground truth data that is not an ROI or scene label to a groundTruthMultisignal object. In the label definitions table, specify a label definition whose labelType is Custom. </td><td>Required</td></tr>
</table>

Column	Description	Required or Optional
`Group`	Strings or character vectors specifying the group to which each label definition belongs.	Optional If you create label definitions at the MATLAB command line, you do not need to include a `Group` column. If you export label definitions from the **Ground Truth Labeler** app or create them using a `labelDefinitionCreatorMultisignal` object, the label definitions table includes this column, even if you did not specify groups. The app assigns each label definition a `Group` value of `'None'`.

Column	Description	Required or Optional
`Description`	Strings or character vectors that describe each label definition.	Optional If you create label definitions at the MATLAB command line, you do not need to include a `Description` column. If you export label definitions from the **Ground Truth Labeler** app or create them using a `labelDefinitionCreatorMultisignal` object, the label definitions table includes this column, even if you did not specify descriptions. The `Description` for these label definitions is an empty character vector.

Column	Description	Required or Optional
`LabelColor`	1-by-3 row vectors of RGB triplets that specify the colors of the label definitions. Values are in the range [0, 1]. The color yellow (RGB triplet [1 1 0]) is reserved for the color of selected labels in the **Ground Truth Labeler** app.	Optional When you define labels in the **Ground Truth Labeler** app, you must specify a color. Therefore, an exported label definitions table always includes this column. When you create label definitions using the `labelDefinitionCreatorMultisignal` object without specifying colors, the returned label definition table includes this column, but all column values are empty.

Column	Description	Required or Optional
PixelLabelID	Scalars, column vectors, M-by-3 matrices of integer-valued label IDs. PixelLabelID specifies the pixel label values used to represent a label definition. Pixel label ID values must be between 0 and 255.	Optional When you define pixel labels in the **Ground Truth Labeler** app or the labelDefinitionCreatorMultisignal object, the generated label definitions table includes this column. When creating a label definitions table at the MATLAB command line, if you set LabelType to labelType.PixelLabel for any label, then this column is required.
Hierarchy	Structures containing sublabel and attribute data for each label definition. For an example of the Hierarchy format, see “Export and Explore Ground Truth Labels for Multiple Signals”.	Optional When you define sublabels or attributes in the **Ground Truth Labeler** app or the labelDefinitionCreatorMultisignal object, the generated label definitions table includes this column.

ROILabelData — ROI label data
ROILabelData object

ROI label data across all signals, specified as an ROILabelData object.

For Rectangle, Cuboid, ProjectedCuboid, and Line label types, ground truth data that is not a floating-point array has a data type of single.

SceneLabelData — Scene label data
SceneLabelData object

Scene label data across all signals, specified as a SceneLabelData object.

Object Functions

selectLabelsByLabelName	Select multisignal ground truth by label name
selectLabelsByLabelType	Select multisignal ground truth by label type
selectLabelsByGroupName	Select multisignal ground truth by label group name
selectLabelsBySignalName	Select multisignal ground truth by signal name
selectLabelsBySignalType	Select multisignal ground truth labels by signal type
gatherLabelData	Gather synchronized label data from ground truth
writeFrames	Write signal frames for ground truth data to disk
changeFilePaths	Change file paths in multisignal ground truth data

Examples

Create Ground Truth from Multiple Signals

Create ground truth data for a video signal and a lidar point cloud sequence signal that captures the same driving scene. Specify the signal sources, label definitions, and ROI and scene label data.

Create the video data source from an MP4 file.

```
sourceName = '01_city_c2s_fcw_10s.mp4';
sourceParams = [];
vidSource = vision.labeler.loading.VideoSource;
vidSource.loadSource(sourceName,sourceParams);
```

Create the point cloud sequence source from a folder of point cloud data (PCD) files.

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

lidarSourceData = load(fullfile(pcSeqFolder,'timestamps.mat'));

sourceName = pcSeqFolder;
sourceParams = struct;
sourceParams.Timestamps = timestamps;

pcseqSource = vision.labeler.loading.PointCloudSequenceSource;
pcseqSource.loadSource(sourceName,sourceParams);
```

Combine the signal sources into an array.

```
dataSource = [vidSource pcseqSource]
```

```
dataSource = 

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

    SourceName
    SourceParams
    SignalName
    SignalType
    Timestamp
    NumSignals
```

Create a table of label definitions for the ground truth data by using a `labelDefinitionCreatorMultisignal` object.

- The `Car` label definition appears twice. Even though `Car` is defined as a rectangle, you can draw rectangles only for image signals, such as videos. The `labelDefinitionCreatorMultisignal` object creates an additional row for lidar point cloud signals. In these signal types, you can draw `Car` labels as cuboids only.
- The label definitions have no descriptions and no assigned colors, so the `Description` and `LabelColor` columns are empty.
- The label definitions have no assigned groups, so for all label definitions, the corresponding cell in the `Group` column is set to `'None'`.
- `Road` is a pixel label definition, so the table includes a `PixelLabelID` column.
- No label definitions have sublabels or attributes, so the table does not include a `Hierarchy` column for storing such information.

```
ldc = labelDefinitionCreatorMultisignal;
addLabel(ldc,'Car','Rectangle');
addLabel(ldc,'Truck','ProjectedCuboid');
addLabel(ldc,'Lane','Line');
addLabel(ldc,'Road','PixelLabel');
addLabel(ldc,'Sunny','Scene');
labelDefs = create(ldc)
```

```
labelDefs = 

  6x7 table

      Name       SignalType        LabelType          Group       Description     LabelColor     PixelL
    _________    __________    _______________    _________    ___________    __________    ______

    {'Car'  }    Image         Rectangle          {'None'}       {' '}        {0x0 char}    {0x0 c
    {'Car'  }    PointCloud    Cuboid             {'None'}       {' '}        {0x0 char}    {0x0 c
    {'Truck'}    Image         ProjectedCuboid    {'None'}       {' '}        {0x0 char}    {0x0 c
    {'Lane' }    Image         Line               {'None'}       {' '}        {0x0 char}    {0x0 c
    {'Road' }    Image         PixelLabel         {'None'}       {' '}        {0x0 char}    {[
    {'Sunny'}    Time          Scene              {'None'}       {' '}        {0x0 char}    {0x0 c
```

Create ROI label data for the first frame of the video.

```
numVideoFrames = numel(vidSource.Timestamp{1});
carData = cell(numVideoFrames,1);
laneData = cell(numVideoFrames,1);
truckData = cell(numVideoFrames,1);
carData{1} = [304 212 37 33];
laneData{1} = [70 458; 311 261];
truckData{1} = [309,215,33,24,330,211,33,24];
videoData = timetable(vidSource.Timestamp{1},carData,laneData, ...
                      'VariableNames',{'Car','Lane'});
```

Create ROI label data for the first point cloud in the sequence.

```
numPCFrames = numel(pcseqSource.Timestamp{1});
carData = cell(numPCFrames, 1);
carData{1} = [27.35 18.32 -0.11 4.25 4.75 3.45 0 0 0];
lidarData = timetable(pcseqSource.Timestamp{1},carData,'VariableNames',{'Car'});
```

Combine the ROI label data for both sources.

```
signalNames = [dataSource.SignalName];
roiData = vision.labeler.labeldata.ROILabelData(signalNames,{videoData,lidarData})
```

```
roiData = 

  ROILabelData with properties:

    video_01_city_c2s_fcw_10s: [204x2 timetable]
                lidarSequence: [34x1 timetable]
```

Create scene label data for the first 10 seconds of the driving scene.

```
sunnyData = seconds([0 10]);
labelNames = ["Sunny"];
sceneData = vision.labeler.labeldata.SceneLabelData(labelNames,{sunnyData})
```

```
sceneData = 

  SceneLabelData with properties:

    Sunny: [0 sec    10 sec]
```

Create a ground truth object from the signal sources, label definitions, and ROI and scene label data. You can import this object into the **Ground Truth Labeler** app for manual labeling or to run a labeling automation algorithm on it. You can also extract training data from this object for deep learning models by using the `gatherLabelData` function.

```
gTruth = groundTruthMultisignal(dataSource,labelDefs,roiData,sceneData)
```

```
gTruth = 

  groundTruthMultisignal with properties:

          DataSource: [1x2 vision.labeler.loading.MultiSignalSource]
    LabelDefinitions: [6x7 table]
```

```
    ROILabelData: [1x1 vision.labeler.labeldata.ROILabelData]
  SceneLabelData: [1x1 vision.labeler.labeldata.SceneLabelData]
```

Tips

- `groundTruthMultisignal` objects with video-based data sources rely on the video reading capabilities of your operating system. A `groundTruthMultisignal` object created using video data sources remains consistent only for the same platform that was used to create it. To create a platform-specific `groundTruthMultisignal` object, convert the videos into sequences of images.
- To create a `groundTruthMultisignal` object containing ROI label data but no scene label data, specify the `SceneLabelData` property as an empty array. To create this array, at the MATLAB command prompt, enter this code.

  ```
  sceneData = vision.labeler.labeldata.SceneLabelData.empty
  ```

See Also

Apps
Ground Truth Labeler

Objects
`attributeType` | `labelDefinitionCreatorMultisignal` | `labelType`

Topics
"Get Started with the Ground Truth Labeler"
"Share and Store Labeled Ground Truth Data" (Computer Vision Toolbox)
"How Labeler Apps Store Exported Pixel Labels" (Computer Vision Toolbox)

Introduced in R2020a

selectLabelsByLabelName

Select multisignal ground truth by label name

Syntax

```
gtLabel = selectLabelsByLabelName(gTruth,labelNames)
```

Description

`gtLabel = selectLabelsByLabelName(gTruth,labelNames)` selects labels specified by `labelNames` from a `groundTruthMultisignal` object, `gTruth`. The function returns a corresponding `groundTruthMultisignal` object, `gtLabel`, that contains only the selected labels. If `gTruth` is a vector of `groundTruthMultisignal` objects, then the function returns a vector of corresponding `groundTruthMultisignal` objects that contain only the selected labels.

Examples

Select Ground Truth Labels by Label Name

Select ground truth labels from a `groundTruthMultisignal` object by specifying a label name.

Load a `groundTruthMultisignal` object containing ROI and scene label data for a video and corresponding lidar point cloud sequence. The helper function used to load this object is attached to the example as a supporting file.

```
gTruth = helperLoadGTruthVideoLidar;
```

Inspect the label definitions. The object contains ROI labels for different vehicle types and a scene label. Because image and lidar point cloud signals represent ROIs differently, the car label has two rows. On image signals, such as videos, you draw the label as a rectangle. On point cloud signals, you draw the label as a cuboid.

```
gTruth.LabelDefinitions
```

```
ans =

  5x7 table

     Name       SignalType    LabelType       Group         Description      LabelColor       Hiera
    _________    __________    _________    ____________    ___________    ____________    ______

    {'car'  }    Image         Rectangle    {'Vehicles'}    {0x0 char}     {1x3 double}    {1x1 s
    {'car'  }    PointCloud    Cuboid       {'Vehicles'}    {0x0 char}     {1x3 double}    {1x1 s
    {'truck'}    Image         Rectangle    {'Vehicles'}    {0x0 char}     {1x3 double}    {0x0 d
    {'truck'}    PointCloud    Cuboid       {'Vehicles'}    {0x0 char}     {1x3 double}    {0x0 d
    {'sunny'}    Time          Scene        {'None'    }    {0x0 char}     {1x3 double}    {0x0 d
```

Create a new `groundTruthMultisignal` object that contains labels for only the `"sunny"` label definition.

```
labelNames = "sunny";
gtLabel = selectLabelsByLabelName(gTruth,labelNames);
```

For the original and new objects, inspect the ROI label data. Because you did not select any ROI label names, in the new object, the signals do not contain any ROI label data at any timestamp.

```
gTruth.ROILabelData
gtLabel.ROILabelData
```

```
ans = 

  ROILabelData with properties:

    video_01_city_c2s_fcw_10s: [204x2 timetable]
                lidarSequence: [34x2 timetable]


ans = 

  ROILabelData with properties:

    video_01_city_c2s_fcw_10s: [204x0 timetable]
                lidarSequence: [34x0 timetable]
```

For the original and new objects, inspect the scene label data. Because you selected the `"sunny"` label, the original object and new object contain identical scene label data.

```
gTruth.SceneLabelData
gtLabel.SceneLabelData
```

```
ans = 

  SceneLabelData with properties:

    sunny: [0 sec    10.15 sec]


ans = 

  SceneLabelData with properties:

    sunny: [0 sec    10.15 sec]
```

Input Arguments

`gTruth` — Multisignal ground truth data
`groundTruthMultisignal` object | vector of `groundTruthMultisignal` objects

Multisignal ground truth data, specified as a `groundTruthMultisignal` object or vector of `groundTruthMultisignal` objects.

`labelNames` — Label names
character vector | string scalar | cell array of character vectors | string vector

Label names, specified as a character vector, string scalar, cell array of character vectors, or string vector.

To view all label names in a `groundTruthMultisignal` object, `gTruth`, enter this command at the MATLAB command prompt.

```
unique(gTruth.LabelDefinitions.Name)
```

Example: `'car'`

Example: `"car"`

Example: `{'car','lane'}`

Example: `["car" "lane"]`

Output Arguments

`gtLabel` — Ground truth with only selected labels
`groundTruthMultisignal` object | vector of `groundTruthMultisignal` objects

Ground truth with only the selected labels, returned as a `groundTruthMultisignal` object or vector of `groundTruthMultisignal` objects.

Each `groundTruthMultisignal` object in `gtLabel` corresponds to a `groundTruthMultisignal` object in the `gTruth` input. The returned objects contain only the labels that are of the label names specified by the `labelNames` input.

Limitations

- Selecting pixel labels by label name is not supported. However, you can select all labels of type pixel. Use the `selectLabelsByLabelType` function, specifying the label type as a `labelType.PixelLabel` enumeration.
- Selecting sublabels by label name is not supported.

See Also

Objects
`groundTruthMultisignal`

Functions
`selectLabelsByGroupName` | `selectLabelsByLabelType` | `selectLabelsBySignalName` | `selectLabelsBySignalType`

Introduced in R2020a

selectLabelsByLabelType

Select multisignal ground truth by label type

Syntax

```
gtLabel = selectLabelsByLabelType(gTruth,labelTypes)
```

Description

`gtLabel = selectLabelsByLabelType(gTruth,labelTypes)` selects labels of the types specified by `labelTypes` from a `groundTruthMultisignal` object, `gTruth`. The function returns a corresponding `groundTruthMultisignal` object, `gtLabel`, that contains only the selected labels. If `gTruth` is a vector of `groundTruthMultisignal` objects, then the function returns a vector of corresponding `groundTruthMultisignal` objects that contain only the selected labels.

Examples

Select Ground Truth Labels by Label Type

Select ground truth labels from a `groundTruthMultisignal` object by specifying a label type.

Load a `groundTruthMultisignal` object containing ROI and scene label data for a video and corresponding lidar point cloud sequence. The helper function used to load this object is attached to the example as a supporting file.

```
gTruth = helperLoadGTruthVideoLidar;
```

Inspect the label definitions. The object contains definitions for rectangle, cuboid, and scene label types.

```
gTruth.LabelDefinitions
```

```
ans =

  5x7 table

      Name       SignalType    LabelType        Group        Description      LabelColor       Hiera
    _________    __________    _________    ____________    ___________    ____________    ______

    {'car'  }    Image         Rectangle    {'Vehicles'}    {0x0 char}     {1x3 double}    {1x1 s
    {'car'  }    PointCloud    Cuboid       {'Vehicles'}    {0x0 char}     {1x3 double}    {1x1 s
    {'truck'}    Image         Rectangle    {'Vehicles'}    {0x0 char}     {1x3 double}    {0x0 d
    {'truck'}    PointCloud    Cuboid       {'Vehicles'}    {0x0 char}     {1x3 double}    {0x0 d
    {'sunny'}    Time          Scene        {'None'    }    {0x0 char}     {1x3 double}    {0x0 d
```

Inspect the ROI labels. The object contains labels for the lidar point cloud sequence and the video.

```
gTruth.ROILabelData
```

```
ans = 

  ROILabelData with properties:

    video_01_city_c2s_fcw_10s: [204x2 timetable]
                lidarSequence: [34x2 timetable]
```

Create a new `groundTruthMultisignal` object that contains only labels that are of type cuboid.

```
labelTypes = labelType.Cuboid;
gtLabel = selectLabelsByLabelType(gTruth,labelTypes);
```

For the original and new objects, inspect the first five rows of label data for the lidar point cloud sequence. Because the lidar point cloud signal in the original object contains only cuboid labels, the new object contains the same label data for the lidar sequence as the original object.

```
lidarLabels = gTruth.ROILabelData.lidarSequence;
lidarLabelsSelection = gtLabel.ROILabelData.lidarSequence;

numrows = 5;
head(lidarLabels,numrows)
head(lidarLabelsSelection,numrows)
```

```
ans =

  5x2 timetable

       Time            car             truck
    ___________    ____________    ____________

    0 sec          {1x1 struct}    {1x0 struct}
    0.29926 sec    {1x1 struct}    {1x0 struct}
    0.59997 sec    {1x1 struct}    {1x0 struct}
    0.8485 sec     {1x1 struct}    {1x0 struct}
    1.1484 sec     {1x1 struct}    {1x0 struct}


ans =

  5x2 timetable

       Time            car             truck
    ___________    ____________    ____________

    0 sec          {1x1 struct}    {1x0 struct}
    0.29926 sec    {1x1 struct}    {1x0 struct}
    0.59997 sec    {1x1 struct}    {1x0 struct}
    0.8485 sec     {1x1 struct}    {1x0 struct}
    1.1484 sec     {1x1 struct}    {1x0 struct}
```

For the original and new objects, inspect the first five rows of label data for the video. Because video signals do not support the `Cuboid` label type, the new object contains no label data for the video.

```
videoLabels = gTruth.ROILabelData.video_01_city_c2s_fcw_10s;
videoLabelsSelection = gtLabel.ROILabelData.video_01_city_c2s_fcw_10s;
```

```
head(videoLabels,numrows)
head(videoLabelsSelection,numrows)
```

```
ans =

  5x2 timetable

      Time          car             truck
    ________    ____________    ____________

    0 sec       {1x3 struct}    {1x0 struct}
    0.05 sec    {1x3 struct}    {1x0 struct}
    0.1 sec     {1x3 struct}    {1x0 struct}
    0.15 sec    {1x3 struct}    {1x0 struct}
    0.2 sec     {1x3 struct}    {1x0 struct}


ans =

  5x0 empty timetable
```

Input Arguments

gTruth — Multisignal ground truth data
groundTruthMultisignal object | vector of groundTruthMultisignal objects

Multisignal ground truth data, specified as a groundTruthMultisignal object or vector of groundTruthMultisignal objects.

labelTypes — Label types
labelType enumeration | vector of labelType enumerations

Label types, specified as a labelType enumeration or vector of labelType enumerations.

To view all label types in a groundTruthMultisignal object, gTruth, enter this command at the MATLAB command prompt.

```
unique(gTruth.LabelDefinitions.LabelType)
```

Example: labelType.Cuboid

Example: [labelType.Cuboid labelType.Scene]

Output Arguments

gtLabel — Ground truth with only selected labels
groundTruthMultisignal object | vector of groundTruthMultisignal objects

Ground truth with only the selected labels, returned as a groundTruthMultisignal object or vector of groundTruthMultisignal objects.

Each `groundTruthMultisignal` object in `gtLabel` corresponds to a `groundTruthMultisignal` object in the `gTruth` input. The returned objects contain only the labels that are of the label types specified by the `labelTypes` input.

Limitations

- Selecting sublabels by label type is not supported.

See Also

Objects
`groundTruthMultisignal`

Functions
`selectLabelsByGroupName` | `selectLabelsByLabelName` | `selectLabelsBySignalName` | `selectLabelsBySignalType`

Introduced in R2020a

selectLabelsByGroupName

Select multisignal ground truth by label group name

Syntax

`gtLabel = selectLabelsByGroupName(gTruth,labelGroups)`

Description

`gtLabel = selectLabelsByGroupName(gTruth,labelGroups)` selects labels belonging to the groups specified by `labelGroups` from a `groundTruthMultisignal` object, `gTruth`. The function returns a corresponding `groundTruthMultisignal` object, `gtLabel`, that contains only the selected labels. If `gTruth` is a vector of `groundTruthMultisignal` objects, then the function returns a vector of corresponding `groundTruthMultisignal` objects that contain only the selected labels.

Examples

Select Ground Truth Labels by Group Name

Select ground truth labels from a `groundTruthMultisignal` object by specifying a group name.

Load a `groundTruthMultisignal` object containing ROI and scene label data for a video and corresponding lidar point cloud sequence. The helper function used to load this object is attached to the example as a supporting file.

```
gTruth = helperLoadGTruthVideoLidar;
```

Inspect the label definitions. The object contains label definitions in a `"Vehicles"` group. Ungrouped labels are in the group named `"None"`.

```
gTruth.LabelDefinitions
```

```
ans =

  5x7 table

      Name       SignalType    LabelType       Group        Description     LabelColor      Hiera
    _________    __________    _________    ____________    ___________    ____________    ______

    {'car'  }    Image         Rectangle    {'Vehicles'}    {0x0 char}     {1x3 double}    {1x1 s
    {'car'  }    PointCloud    Cuboid       {'Vehicles'}    {0x0 char}     {1x3 double}    {1x1 s
    {'truck'}    Image         Rectangle    {'Vehicles'}    {0x0 char}     {1x3 double}    {0x0 d
    {'truck'}    PointCloud    Cuboid       {'Vehicles'}    {0x0 char}     {1x3 double}    {0x0 d
    {'sunny'}    Time          Scene        {'None'    }    {0x0 char}     {1x3 double}    {0x0 d
```

Create a new `groundTruthMultisignal` object that contains labels for only the `"Vehicles"` group.

```
groupNames = "Vehicles";
gtLabel = selectLabelsByGroupName(gTruth,groupNames);
```

For the original and new objects, inspect the ROI label data. Because `"Vehicles"` is the only group used for the ROI label data, the original and new object contain identical ROI label data.

```
gTruth.ROILabelData
gtLabel.ROILabelData
```

```
ans = 

  ROILabelData with properties:

    video_01_city_c2s_fcw_10s: [204x2 timetable]
                lidarSequence: [34x2 timetable]


ans = 

  ROILabelData with properties:

    video_01_city_c2s_fcw_10s: [204x2 timetable]
                lidarSequence: [34x2 timetable]
```

For the original and new objects, inspect the scene label data. The `"None"` group, which is used only in scene labels, was not selected. Therefore, the new object contains no scene label data.

```
gTruth.SceneLabelData
gtLabel.SceneLabelData
```

```
ans = 

  SceneLabelData with properties:

    sunny: [0 sec    10.15 sec]


ans = 

  SceneLabelData with properties:

    sunny: [0x0 duration]
```

Input Arguments

`gTruth` — Multisignal ground truth data
`groundTruthMultisignal` object | vector of `groundTruthMultisignal` objects

Multisignal ground truth data, specified as a `groundTruthMultisignal` object or vector of `groundTruthMultisignal` objects.

`labelGroups` — Label group names
character vector | string scalar | cell array of character vectors | string vector

Label group names, specified as a character vector, string scalar, cell array of character vectors, or string vector.

To view all label group names in a `groundTruthMultisignal` object, `gTruth`, enter this command at the MATLAB command prompt.

```
unique(gTruth.LabelDefinitions.Group)
```

Example: `'Vehicles'`

Example: `"Vehicles"`

Example: `{'Vehicles','Signs'}`

Example: `["Vehicles" "Signs"]`

Output Arguments

`gtLabel` — Ground truth with only selected labels
`groundTruthMultisignal` object | vector of `groundTruthMultisignal` objects

Ground truth with only the selected labels, returned as a `groundTruthMultisignal` object or vector of `groundTruthMultisignal` objects.

Each `groundTruthMultisignal` object in `gtLabel` corresponds to a `groundTruthMultisignal` object in the `gTruth` input. The returned objects contain only the labels belonging to the groups specified by the `labelGroups` input.

See Also

Objects
`groundTruthMultisignal`

Functions
`selectLabelsByLabelName` | `selectLabelsByLabelType` | `selectLabelsBySignalName` | `selectLabelsBySignalType`

Introduced in R2020a

selectLabelsBySignalName

Select multisignal ground truth by signal name

Syntax

```
gtLabel = selectLabelsBySignalName(gTruth,signalNames)
```

Description

`gtLabel = selectLabelsBySignalName(gTruth,signalNames)` selects labels for the signals specified by `signalNames` from a `groundTruthMultisignal` object, `gTruth`. The function returns a corresponding `groundTruthMultisignal` object, `gtLabel`, that contains only the selected labels. If `gTruth` is a vector of `groundTruthMultisignal` objects, then the function returns a vector of corresponding `groundTruthMultisignal` objects that contain only the selected labels.

Examples

Select Ground Truth Labels by Signal Name

Select ground truth labels from a `groundTruthMultisignal` object by specifying a signal name.

Load a `groundTruthMultisignal` object containing ROI and scene label data for a video and corresponding lidar point cloud sequence. The helper function used to load this object is attached to the example as a supporting file.

```
gTruth = helperLoadGTruthVideoLidar;
```

Inspect the ROI labels. The object contains labels for the lidar point cloud sequence and the video.

```
gTruth.ROILabelData
```

```
ans = 

  ROILabelData with properties:

    video_01_city_c2s_fcw_10s: [204x2 timetable]
                lidarSequence: [34x2 timetable]
```

Create a new `groundTruthMultisignal` object that contains labels for only the `lidarSequence` signal.

```
signalNames = "lidarSequence";
gtLabel = selectLabelsBySignalName(gTruth,signalNames);
```

For the original and new objects, inspect the first five rows of label data for the lidar point cloud sequence. The new object contains the same label data for the lidar sequence as the original object.

```
lidarLabels = gTruth.ROILabelData.lidarSequence;
lidarLabelsSelection = gtLabel.ROILabelData.lidarSequence;
```

```
numrows = 5;
head(lidarLabels,numrows)
head(lidarLabelsSelection,numrows)
```

```
ans =

  5x2 timetable

       Time            car             truck
    ___________    ____________    ____________

    0 sec          {1x1 struct}    {1x0 struct}
    0.29926 sec    {1x1 struct}    {1x0 struct}
    0.59997 sec    {1x1 struct}    {1x0 struct}
    0.8485 sec     {1x1 struct}    {1x0 struct}
    1.1484 sec     {1x1 struct}    {1x0 struct}


ans =

  5x2 timetable

       Time            car             truck
    ___________    ____________    ____________

    0 sec          {1x1 struct}    {1x0 struct}
    0.29926 sec    {1x1 struct}    {1x0 struct}
    0.59997 sec    {1x1 struct}    {1x0 struct}
    0.8485 sec     {1x1 struct}    {1x0 struct}
    1.1484 sec     {1x1 struct}    {1x0 struct}
```

For the original and new objects, inspect the first five rows of label data for the video. The new object contains no label data for the video.

```
videoLabels = gTruth.ROILabelData.video_01_city_c2s_fcw_10s;
videoLabelsSelection = gtLabel.ROILabelData.video_01_city_c2s_fcw_10s;

head(videoLabels,numrows)
head(videoLabelsSelection,numrows)
```

```
ans =

  5x2 timetable

      Time          car             truck
    ________    ____________    ____________

    0 sec       {1x3 struct}    {1x0 struct}
    0.05 sec    {1x3 struct}    {1x0 struct}
    0.1 sec     {1x3 struct}    {1x0 struct}
    0.15 sec    {1x3 struct}    {1x0 struct}
    0.2 sec     {1x3 struct}    {1x0 struct}
```

```
ans =

  5x0 empty timetable
```

Input Arguments

gTruth — Multisignal ground truth data
groundTruthMultisignal object | vector of groundTruthMultisignal objects

Multisignal ground truth data, specified as a groundTruthMultisignal object or vector of groundTruthMultisignal objects.

signalNames — Signal names
character vector | string scalar | cell array of character vectors | string vector

Signal names, specified as a character vector, string scalar, cell array of character vectors, or string vector.

To view all signal names in a groundTruthMultisignal object, gTruth, enter this command at the MATLAB command prompt.

```
gTruth.DataSource.SignalName
```

Example: 'lidarSequence'

Example: "lidarSequence"

Example: {'lidarSequence','imageSequence'}

Example: ["lidarSequence" "imageSequence"]

Output Arguments

gtLabel — Ground truth with only selected labels
groundTruthMultisignal object | vector of groundTruthMultisignal objects

Ground truth with only the selected labels, returned as a groundTruthMultisignal object or vector of groundTruthMultisignal objects.

Each groundTruthMultisignal object in gtLabel corresponds to a groundTruthMultisignal object in the gTruth input. The returned objects contain only the labels with signal names specified by the signalNames input.

See Also

Objects
groundTruthMultisignal

Functions
selectLabelsByGroupName | selectLabelsByLabelName | selectLabelsByLabelType | selectLabelsBySignalType

Introduced in R2020a

selectLabelsBySignalType

Select multisignal ground truth labels by signal type

Syntax

`gtLabel = selectLabelsBySignalType(gTruth,signalTypes)`

Description

`gtLabel = selectLabelsBySignalType(gTruth,signalTypes)` selects labels of the signal types specified by `signalTypes` from a `groundTruthMultisignal` object, `gTruth`. The function returns a corresponding `groundTruthMultisignal` object, `gtLabel`, that contains only the selected labels. If `gTruth` is a vector of `groundTruthMultisignal` objects, then the function returns a vector of corresponding `groundTruthMultisignal` objects that contain only the selected labels.

Examples

Select Ground Truth Labels by Signal Type

Select ground truth labels from a `groundTruthMultisignal` object by specifying a signal type.

Load a `groundTruthMultisignal` object containing ROI and scene label data for a video and corresponding lidar point cloud sequence. The helper function used to load this object is attached to the example as a supporting file.

```
gTruth = helperLoadGTruthVideoLidar;
```

Inspect the label definitions. The object contains definitions for image, point cloud, and time signals.

```
gTruth.LabelDefinitions
```

```
ans =

  5x7 table

      Name       SignalType    LabelType       Group         Description      LabelColor      Hiera
    _________    __________    _________    ____________    ___________    ____________    _____

    {'car'  }    Image         Rectangle    {'Vehicles'}    {0x0 char}     {1x3 double}    {1x1 s
    {'car'  }    PointCloud    Cuboid       {'Vehicles'}    {0x0 char}     {1x3 double}    {1x1 s
    {'truck'}    Image         Rectangle    {'Vehicles'}    {0x0 char}     {1x3 double}    {0x0 d
    {'truck'}    PointCloud    Cuboid       {'Vehicles'}    {0x0 char}     {1x3 double}    {0x0 d
    {'sunny'}    Time          Scene        {'None'    }    {0x0 char}     {1x3 double}    {0x0 d
```

Inspect the ROI labels. The object contains labels for the lidar point cloud sequence and the video.

```
gTruth.ROILabelData
```

```
ans = 

  ROILabelData with properties:

    video_01_city_c2s_fcw_10s: [204x2 timetable]
                lidarSequence: [34x2 timetable]
```

Create a new `groundTruthMultisignal` object that contains labels for only point cloud signals.

```
signalTypes = vision.labeler.loading.SignalType.PointCloud;
gtLabel = selectLabelsBySignalType(gTruth,signalTypes);
```

For the original and new objects, inspect the first five rows of label data for the lidar point cloud sequence. Because lidar signals are of type `PointCloud`, the new object contains the same label data for the lidar sequence as the original object.

```
lidarLabels = gTruth.ROILabelData.lidarSequence;
lidarLabelsSelection = gtLabel.ROILabelData.lidarSequence;

numrows = 5;
head(lidarLabels,numrows)
head(lidarLabelsSelection,numrows)
```

```
ans =

  5x2 timetable

       Time            car            truck
    ___________    ____________    ____________

    0 sec          {1x1 struct}    {1x0 struct}
    0.29926 sec    {1x1 struct}    {1x0 struct}
    0.59997 sec    {1x1 struct}    {1x0 struct}
    0.8485 sec     {1x1 struct}    {1x0 struct}
    1.1484 sec     {1x1 struct}    {1x0 struct}


ans =

  5x2 timetable

       Time            car            truck
    ___________    ____________    ____________

    0 sec          {1x1 struct}    {1x0 struct}
    0.29926 sec    {1x1 struct}    {1x0 struct}
    0.59997 sec    {1x1 struct}    {1x0 struct}
    0.8485 sec     {1x1 struct}    {1x0 struct}
    1.1484 sec     {1x1 struct}    {1x0 struct}
```

For the original and new objects, inspect the first five rows of label data for the video. Because video signals are of type `Image`, the new object contains no label data for the video.

```
videoLabels = gTruth.ROILabelData.video_01_city_c2s_fcw_10s;
videoLabelsSelection = gtLabel.ROILabelData.video_01_city_c2s_fcw_10s;
```

```
head(videoLabels,numrows)
head(videoLabelsSelection,numrows)
```

```
ans =

  5x2 timetable

      Time          car             truck
    ________    ____________    ____________

    0 sec       {1x3 struct}    {1x0 struct}
    0.05 sec    {1x3 struct}    {1x0 struct}
    0.1 sec     {1x3 struct}    {1x0 struct}
    0.15 sec    {1x3 struct}    {1x0 struct}
    0.2 sec     {1x3 struct}    {1x0 struct}


ans =

  5x0 empty timetable
```

Input Arguments

gTruth — Multisignal ground truth data
groundTruthMultisignal object | vector of groundTruthMultisignal objects

Multisignal ground truth data, specified as a groundTruthMultisignal object or vector of groundTruthMultisignal objects.

signalTypes — Signal types
vision.labeler.loading.SignalType enumeration | vector of vision.labeler.loading.SignalType enumerations

Signal types, specified as a vision.labeler.loading.SignalType enumeration or vector of vision.labeler.loading.SignalType enumerations.

To view all signal types in a groundTruthMultisignal object, gTruth, enter this command at the MATLAB command prompt.

```
unique(gTruth.LabelDefinitions.SignalType)
```

Example: vision.labeler.loading.SignalType.Image

Example: [vision.labeler.loading.SignalType.Image vision.labeler.loading.SignalType.PointCloud]

Output Arguments

gtLabel — Ground truth with only selected labels
groundTruthMultisignal object | vector of groundTruthMultisignal objects

Ground truth with only the selected labels, returned as a groundTruthMultisignal object or vector of groundTruthMultisignal objects.

Each groundTruthMultisignal object in gtLabel corresponds to a groundTruthMultisignal object in the gTruth input. The returned objects contain only the labels that are of the signal types specified by the signalTypes input.

Limitations

- Selecting sublabels by signal type is not supported.

See Also

Objects
groundTruthMultisignal

Functions
selectLabelsByGroupName | selectLabelsByLabelName | selectLabelsByLabelType | selectLabelsBySignalName

Introduced in R2020a

gatherLabelData

Gather synchronized label data from ground truth

Syntax

```
labelData = gatherLabelData(gTruth,signalNames,labelTypes)
[labelData,timestamps] = gatherLabelData( ___ )
[ ___ ] = gatherLabelData( ___ ,'SampleFactor',sampleFactor)
```

Description

`labelData = gatherLabelData(gTruth,signalNames,labelTypes)` returns synchronized label data gathered from multisignal ground truth data, `gTruth`. The function returns label data for the signals specified by `signalNames` and the label types specified by `labelTypes`.

`[labelData,timestamps] = gatherLabelData( ___ )` additionally returns the signal timestamps associated with the gathered label data, using the arguments from the previous syntax.

Use `timestamps` with the `writeFrames` function to write the associated signal frames from the `groundTruthMultisignal` objects to disk. Use these frames and the associated labels as training data for machine learning or deep learning models.

`[ ___ ] = gatherLabelData( ___ ,'SampleFactor',sampleFactor)` specifies the sample factor used to subsample label data.

Examples

Gather Label Data and Write Associated Signal Frames

Gather label data for a video signal and a lidar point cloud sequence signal from a `groundTruthMultisignal` object. Write the signal frames associated with that label data to disk and visualize the frames.

Add the point cloud sequence folder path to the MATLAB® search path. The video is already on the MATLAB search path.

```
pcSeqDir = fullfile(toolboxdir('driving'),'drivingdata', ...
    'lidarSequence');
addpath(pcSeqDir);
```

Load a `groundTruthMultisignal` object that contains label data for the video and the lidar point cloud sequence.

```
data = load('MultisignalGTruth.mat');
gTruth = data.gTruth;
```

Specify the signals from which to gather label data.

```
signalNames = ["video_01_city_c2s_fcw_10s" "lidarSequence"];
```

The video contains rectangle labels, whereas the lidar point cloud sequence contains cuboid labels. Gather the rectangle labels from the video and the cuboid labels from the lidar point cloud sequence.

```
labelTypes = [labelType.Rectangle labelType.Cuboid];
[labelData,timestamps] = gatherLabelData(gTruth,signalNames,labelTypes);
```

Display the first eight rows of label data from the two signals. Both signals contain data for the `Car` label. In the video, the `Car` label is drawn as a rectangle bounding box. In the lidar point cloud sequence, the `Car` label is drawn as a cuboid bounding box.

```
videoLabelSample = head(labelData{1})
lidarLabelSample = head(labelData{2})
```

```
videoLabelSample =

  table

        Car
    ____________

    {1x4 double}


lidarLabelSample =

  table

        Car
    ____________

    {1x9 double}
```

Write signal frames associated with the gathered label data to temporary folder locations, with one folder per signal. Use the timestamps returned by the `gatherLabelData` function to indicate which signal frames to write.

```
outputFolder = fullfile(tempdir,["videoFrames" "lidarFrames"]);
fileNames = writeFrames(gTruth,signalNames,outputFolder,timestamps);
```

```
Writing 2 frames from the following signals:

* video_01_city_c2s_fcw_10s
* lidarSequence
```

Load the written video signal frames by using an `imageDatastore` object. Load the associated rectangle label data by using a `boxLabelDatastore` (Computer Vision Toolbox) object.

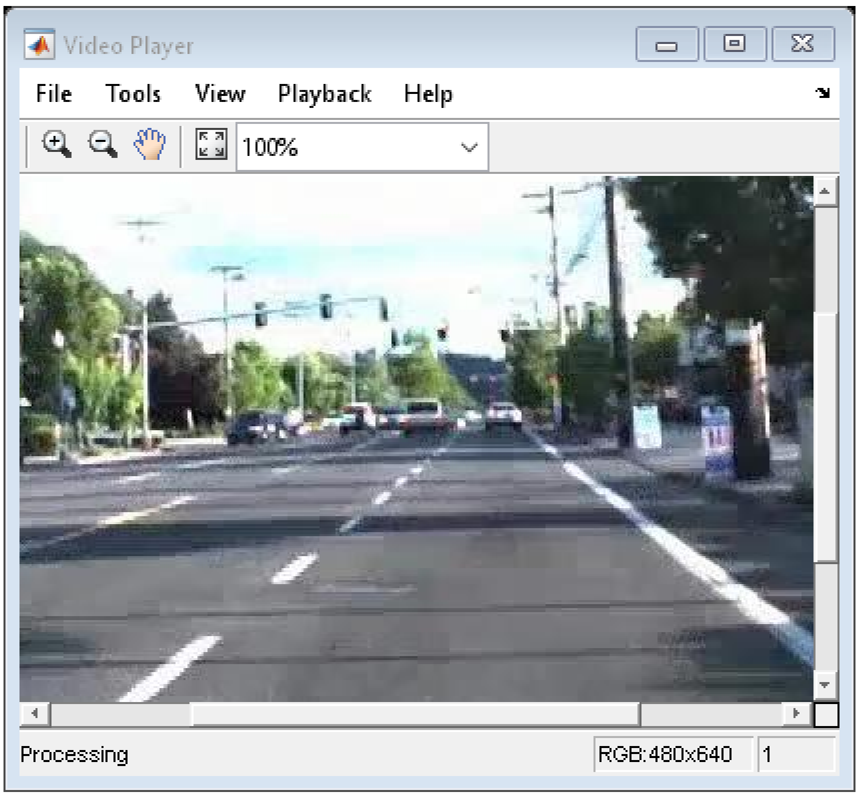
```
imds = imageDatastore(fileNames{1});
blds = boxLabelDatastore(labelData{1});
```

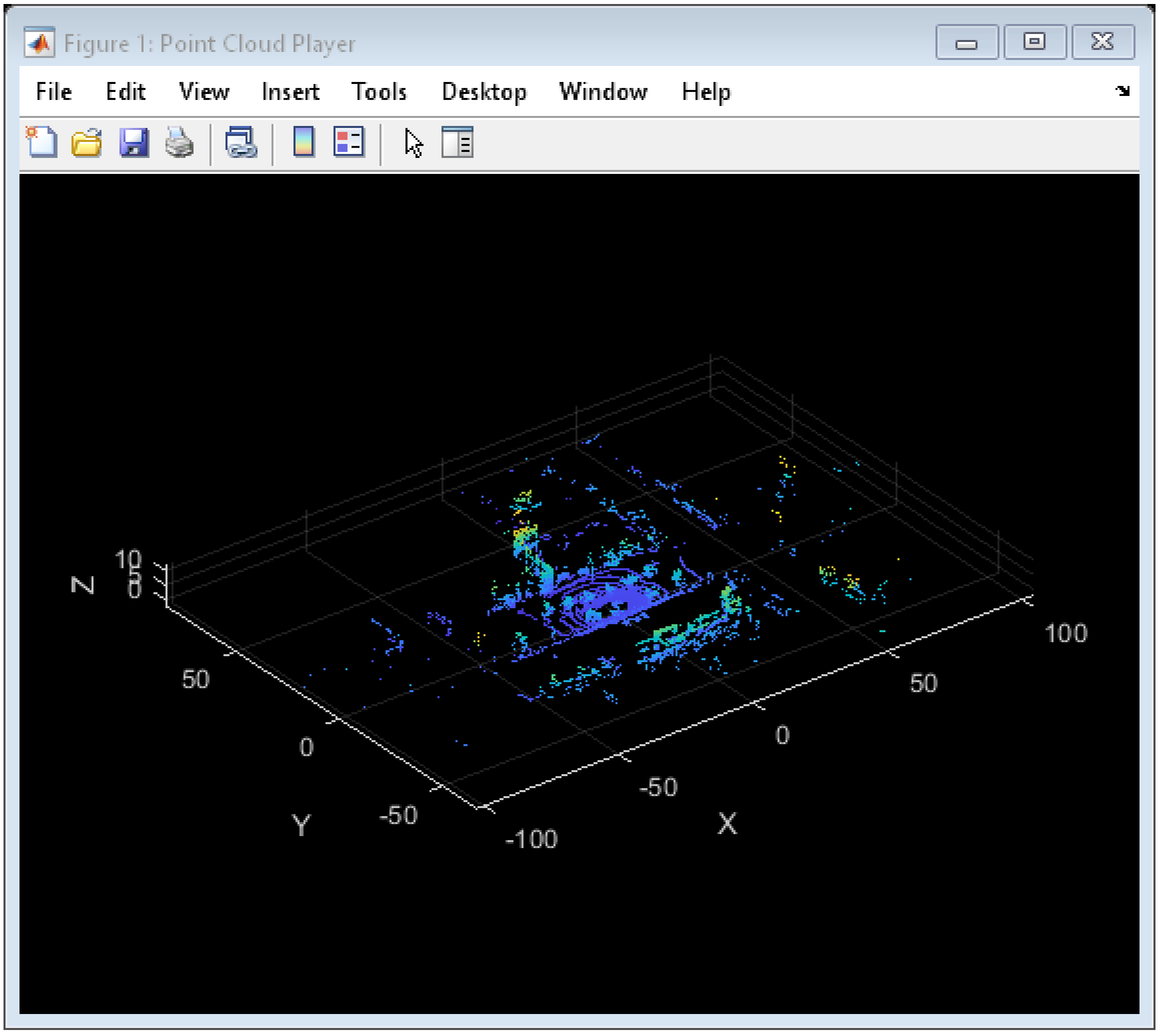
Load the written lidar signal frames by using a `fileDatastore` object. Load the associated cuboid label data by using a `boxLabelDatastore` object.

```
fds = fileDatastore(fileNames{2},'ReadFcn',@pcread);
clds = boxLabelDatastore(labelData{2});
```

Visualize the written video frames by using a `vision.VideoPlayer` (Computer Vision Toolbox) object. Visualize the written lidar frames by using a `pcplayer` (Computer Vision Toolbox) object.

```
videoPlayer = vision.VideoPlayer;

ptCloud = preview(fds);
ptCloudPlayer = pcplayer(ptCloud.XLimits,ptCloud.YLimits,ptCloud.ZLimits);

while hasdata(imds)
    % Read video and lidar frames.
    I = read(imds);
    ptCloud = read(fds);

    % Visualize video and lidar frames.
    videoPlayer(I);
    view(ptCloudPlayer,ptCloud);
end
```





Remove the path to the point cloud sequence folder.

```
rmpath(pcSeqDir);
```

Input Arguments

gTruth — Multisignal ground truth data
groundTruthMultisignal object | vector of groundTruthMultisignal objects

Multisignal ground truth data, specified as a `groundTruthMultisignal` object or vector of `groundTruthMultisignal` objects.

Each `groundTruthMultisignal` object in `gTruth` must include all the signals specified in the `signalNames` input.

In addition, each object must include at least one marked label per gathered label definition. Suppose `gTruth` is a `groundTruthMultisignal` object containing label data for a single video signal named `video_front_camera`. The object contains marked rectangle region of interest (ROI) labels for the `car` label definition but not for the `truck` label definition. If you use this syntax to gather labels of type `Rectangle` from this object, then the `gatherLabelData` function returns an error.

```
labelData = gatherLabelData(gTruth,"video_front_camera",labelType.Rectangle);
```

signalNames — Names of signals

character vector | string scalar | cell array of character vectors | string array

Names of the signals from which to gather label data, specified as a character vector, string scalar, cell array of character vectors, or string vector. The signal names must be valid signal names stored in the input multisignal ground truth data, `gTruth`.

To obtain the signal names from a `groundTruthMultisignal` object, use this syntax, where `gTruth` is the variable name of the object:

```
gTruth.DataSource.SignalName
```

Example: `'video_01_city_c2s_fcw_10s'`

Example: `"video_01_city_c2s_fcw_10s"`

Example: `{'video_01_city_c2s_fcw_10s','lidarSequence'}`

Example: `["video_01_city_c2s_fcw_10s" "lidarSequence"]`

labelTypes — Label types

`labelType` enumeration scalar | `labelType` enumeration vector | cell array of `labelType` enumeration scalars and vectors

Label types from which to gather label data, specified as a `labelType` enumeration scalar, `labelType` enumeration vector, or a cell array of `labelType` enumeration scalars and vectors. The `gatherLabelData` function gathers label data for each signal specified by input `signalNames` and each `groundTruthMultisignal` object specified by input `gTruth`. The number of elements in `labelTypes` must match the number of signals in `signalNames`.

Gather Label Data for Single Label Type per Signal

To gather label data for a single label type per signal, specify `labelTypes` as a `labelType` enumeration scalar or vector. Across all `groundTruthMultisignal` objects in `gTruth`, the `gatherLabelData` function gathers `labelTypes(n)` label data from `signalName(n)`, where `n` is the index of the label type and the corresponding signal name whose label data is to be gathered. Each returned table in the output `labelData` cell array contains data for only one label type per signal.

In this code sample, the `gatherLabelData` function gathers labels of type `Rectangle` from a video signal named `video_front_camera`. The function also gathers labels of type `Cuboid` from a lidar point cloud sequence signal stored in a folder named `lidarData`. The `gTruth` input contains the `groundTruthMultisignal` objects from which this data is to be gathered.

```
labelData = gatherLabelData(gTruth, ...
                            ["video_front_camera","lidarData"], ...
                            [labelType.Rectangle,labelType.Cuboid];
```

To gather label data for a single label type from separate signals, you must repeat the label type for each signal. In this code sample, the `gatherLabelData` function gathers labels of type `Rectangle` from the `video_left_camera` and `video_right_camera` video signals.

```
labelData = gatherLabelData(gTruth, ...
                            ["video_left_camera","video_right_camera"], ...
                            [labelType.Rectangle,labelType.Rectangle];
```

Gather Label Data for Multiple Label Types per Signal

To gather label data for multiple label types per signal, specify `labelTypes` as a cell array of `labelType` enumeration scalars and vectors. Across all `groundTruthMultisignal` objects in `gTruth`, the `gatherLabelData` function gathers `labelTypes{n}` label data from `signalName(n)`, where `n` is the index of the label types and the corresponding signal name whose label data is to be gathered. The function groups the data for these label types into one table per signal per `groundTruthMultisignal` object.

In this code sample, the `gatherLabelData` function gathers labels of type `Rectangle` and `Line` from the `video_front_camera` video signal. The function also gathers labels of type `Cuboid` from a lidar point cloud sequence signal stored in a folder named `lidarData`. The `gTruth` input contains the `groundTruthMultisignal` objects from which this data is to be gathered.

```
labelData = gatherLabelData(gTruth, ...
                            ["video_front_camera", ...
                             "lidarData"], ...
                            {[labelType.Rectangle labelType.Line], ...
                             labelType.Cuboid};
```

Valid Enumeration Types

You can specify one or more of these enumeration types.

- `labelType.Rectangle` — Rectangle ROI labels
- `labelType.Cuboid` — Cuboid ROI labels (point clouds)
- `labelType.ProjectedCuboid` — Projected cuboid ROI labels (images and video data)
- `labelType.Line` — Line ROI labels
- `labelType.PixelLabel` — Pixel ROI labels
- `labelType.Scene` — Scene labels

To gather label data for scenes, you must specify `labelTypes` as the `labelType.Scene` enumeration scalar. You cannot specify any other label types with `labelType.Scene`.

`sampleFactor` — Sample factor

`1` (default) | positive integer

Sample factor used to subsample label data, specified as a positive integer. A sample factor of `K` includes every `K`th signal frame. Increase the sample factor to drop redundant frames from signals with high sample rates, such as videos.

Example: `'SampleFactor',5`

Output Arguments

`labelData` — Label data

cell array of tables

Label data, returned as an `M`-by-`N` cell array of tables, where:

- `M` is the number of `groundTruthMultisignal` objects in `gTruth`.
- When `labelTypes` contains ROI `labelType` enumerations, `N` is the number of signals in `signalNames` and the number of elements in `labelTypes`. In this case, `labelData{m,n}`

contains a table of label data for the nth signal of signalNames that is in the mth groundTruthMultisignal object of gTruth. The table contains label data for only the label types in the nth position of labelTypes.

- When labelTypes contains only the labelType.Scene enumeration, N is equal to 1. In this case, labelData{m} contains a table of scene label data across all signals in the mth groundTruthMultisignal object of gTruth.

For a given label data table, tbl, the table is of size T-by-L, where:

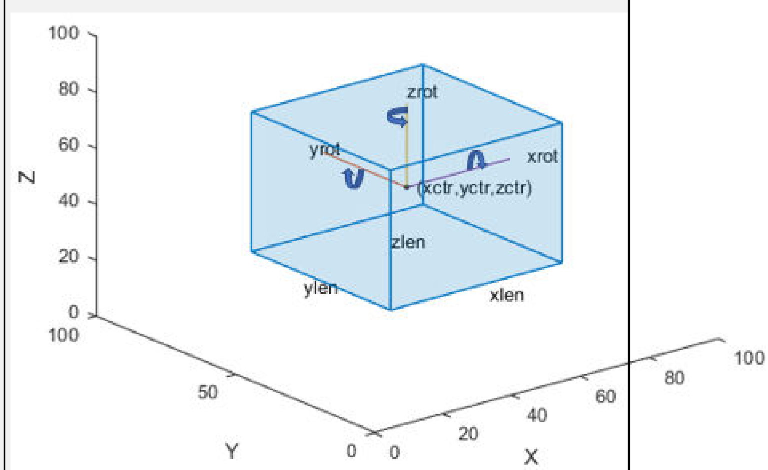
- T is the number of timestamps in the signal for which label data exists.
- L is the number of label definitions that are of the label types gathered for that signal.
- tbl(t,l) contains the label data gathered for the lth label at the tth timestamp.

If one of the signals has no label data at a timestamp, then the corresponding label data table does not include a row for that timestamp.

For each cell in the table, the format of the returned label data depends on the type of label.

Label Type	Storage Format for Labels at Each Timestamp
labelType.Rectangle	M-by-4 numeric matrix of the form [x, y, w, h], where: • M is the number of labels in the frame. • x and y specify the upper-left corner of the rectangle. • w specifies the width of the rectangle, which is its length along the *x*-axis. • h specifies the height of the rectangle, which is its length along the *y*-axis.

Label Type	Storage Format for Labels at Each Timestamp

`labelType.Cuboid`
 	
`labelType.ProjectedCuboid`
	M-by-9 numeric matrix with rows of the form `[xctr, yctr, zctr, xlen, ylen, zlen, xrot, yrot, zrot]`, where:
	• `M` is the number of labels in the frame.
	• `xctr`, `yctr`, and `zctr` specify the center of the cuboid.
	• `xlen`, `ylen`, and `zlen` specify the length of the cuboid along the *x*-axis, *y*-axis, and *z*-axis, respectively, before rotation has been applied.
	• `xrot`, `yrot`, and `zrot` specify the rotation angles for the cuboid along the *x*-axis, *y*-axis, and *z*-axis, respectively. These angles are clockwise-positive when looking in the forward direction of their corresponding axes.
	
	The figure shows how these values determine the position of a cuboid.
	

`labelType.Line`	`M`-by-1 vector of cell arrays, where `M` is the number of labels in the frame. Each cell array contains an `N`-by-2 numeric matrix of the form `[x1 y1; x2 y2; ... ; xN yN]` for `N` points in the polyline.
`labelType.PixelLabel`	Label data for all pixel label definitions is stored in a single `PixelLabelData` column as a categorical label matrix. The label matrix must be stored on disk as a `uint8` image.
`labelType.Scene`	Logical `1` (`true`) if the scene label is applied, otherwise logical `0` (`false`)

Label Data Format

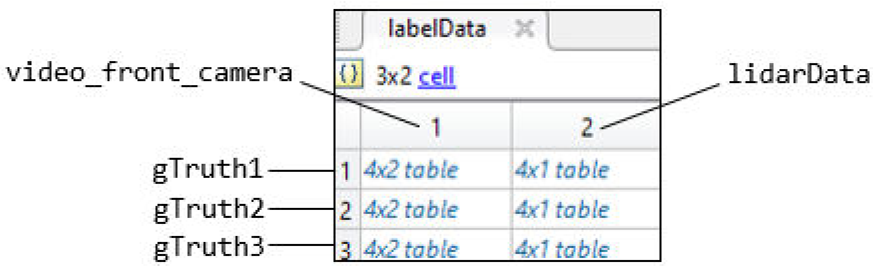
Consider a cell array of label data gathered by using the gatherLabelData function. The function gathers labels from three groundTruthMultisignal objects with variable names gTruth1, gTruth2, and gTruth3.

- For a video signal named video_front_camera, the function gathers labels of type Rectangle and Line.
- For a lidar point cloud sequence signal stored in a folder named lidarData, the function gathers labels of type Cuboid.

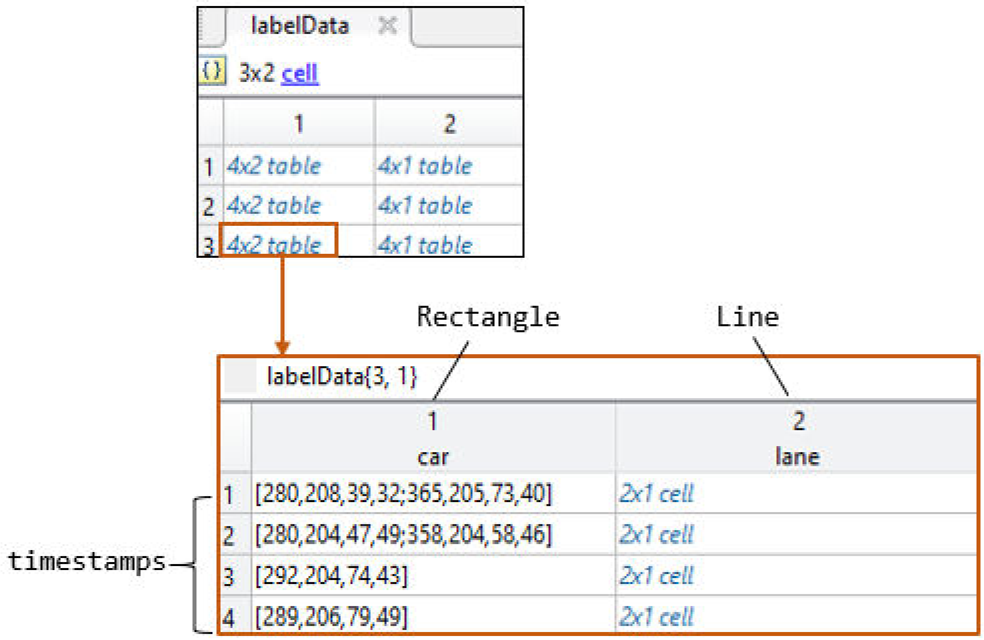
This code shows the call to the gatherLabelData function.

```
labelData = gatherLabelData([gTruth1 gTruth2 gTruth3], ...
                            ["video_front_camera", ...
                             "lidarData"], ...
                            {[labelType.Rectangle labelType.Line], ...
                             labelType.Cuboid});
```

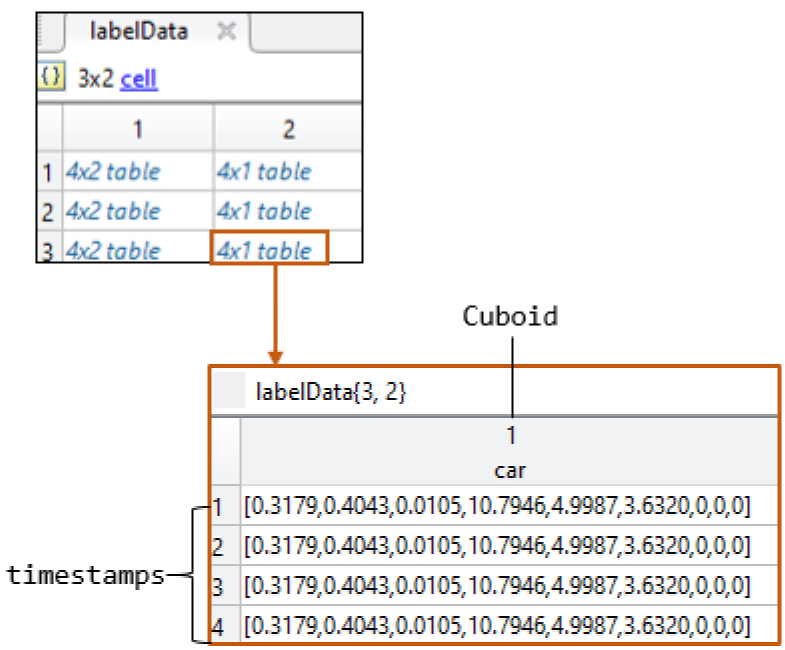
The labelData output is a 3-by-2 cell array of tables. Each row of the cell array contains label data for one of the groundTruthMultisignal objects. The first column contains the label data for the video signal, video_front_camera. The second column contains the label data for the point cloud sequence signal, lidarData. This figure shows the labelData cell array.



This figure shows the label data table for the video signal in the third groundTruthMultisignal object. The gatherLabelData function gathered data for a Rectangle label named car and a Line label named lane. The table contains label data at four timestamps in the signal.



This figure shows the label data table for the lidar signal in the third `groundTruthMultisignal` object. The `gatherLabelData` function gathered data for a `Cuboid` label, also named `car`. The `car` label appears in both signal types because it is marked as a `Rectangle` label for video signals and a `Cuboid` label for lidar signals. The table contains label data at four timestamps in the signal.



`timestamps` — Signal timestamps
cell array of `duration` vectors

Signal timestamps, returned as an `M`-by-`N` cell array of `duration` vectors, where:

- `M` is the number of `groundTruthMultisignal` objects in `gTruth`.
- `N` is the number of signals in `signalNames`.
- `labelData{m,n}` contains the timestamps for the `n`th signal of `signalNames` that is in the `m`th `groundTruthMultisignal` object of `gTruth`.

If you gather label data from multiple signals, the signal timestamps are synchronized to the timestamps of the first signal specified by `signalNames`.

Limitations

- The `gatherLabelData` function does not gather label data for sublabels or attributes. If a label contains sublabels or attributes, in the `labelData` output, the function returns the position of the parent label only.

See Also
`boxLabelDatastore` | `groundTruthMultisignal` | `writeFrames`

Introduced in R2020a

writeFrames

Write signal frames for ground truth data to disk

Syntax

```
fileNames = writeFrames(gTruth,signalNames,location)
fileNames = writeFrames(gTruth,signalNames,location,timestamps)
fileNames = writeFrames( ___ ,Name,Value)
```

Description

`fileNames = writeFrames(gTruth,signalNames,location)` writes the frames of ground truth signal sources to the specified folder locations. The function returns the names of the files containing the written frames. `fileNames` contains one file name per signal specified by `signalNames` per `groundTruthMultisignal` object specified by `gTruth`.

Use these written frames and the associated ground truth labels obtained from the `gatherLabelData` function as training data for machine learning or deep learning models.

`fileNames = writeFrames(gTruth,signalNames,location,timestamps)` specifies the timestamps of the signal frames to write. To obtain signal timestamps, use the `gatherLabelData` function.

`fileNames = writeFrames( ___ ,Name,Value)` specifies options using one or more name-value pair arguments, in addition to any of the input argument combinations from previous syntaxes. For example, you can specify the prefix and file type extension of the file names for the written frames.

Examples

Gather Label Data and Write Associated Signal Frames

Gather label data for a video signal and a lidar point cloud sequence signal from a `groundTruthMultisignal` object. Write the signal frames associated with that label data to disk and visualize the frames.

Add the point cloud sequence folder path to the MATLAB® search path. The video is already on the MATLAB search path.

```
pcSeqDir = fullfile(toolboxdir('driving'),'drivingdata', ...
    'lidarSequence');
addpath(pcSeqDir);
```

Load a `groundTruthMultisignal` object that contains label data for the video and the lidar point cloud sequence.

```
data = load('MultisignalGTruth.mat');
gTruth = data.gTruth;
```

Specify the signals from which to gather label data.

```
signalNames = ["video_01_city_c2s_fcw_10s" "lidarSequence"];
```

The video contains rectangle labels, whereas the lidar point cloud sequence contains cuboid labels. Gather the rectangle labels from the video and the cuboid labels from the lidar point cloud sequence.

```
labelTypes = [labelType.Rectangle labelType.Cuboid];
[labelData,timestamps] = gatherLabelData(gTruth,signalNames,labelTypes);
```

Display the first eight rows of label data from the two signals. Both signals contain data for the `Car` label. In the video, the `Car` label is drawn as a rectangle bounding box. In the lidar point cloud sequence, the `Car` label is drawn as a cuboid bounding box.

```
videoLabelSample = head(labelData{1})
lidarLabelSample = head(labelData{2})
```

```
videoLabelSample = 

  table

        Car     
    ____________

    {1x4 double}


lidarLabelSample = 

  table

        Car     
    ____________

    {1x9 double}
```

Write signal frames associated with the gathered label data to temporary folder locations, with one folder per signal. Use the timestamps returned by the `gatherLabelData` function to indicate which signal frames to write.

```
outputFolder = fullfile(tempdir,["videoFrames" "lidarFrames"]);
fileNames = writeFrames(gTruth,signalNames,outputFolder,timestamps);
```

```
Writing 2 frames from the following signals:

* video_01_city_c2s_fcw_10s
* lidarSequence
```

Load the written video signal frames by using an `imageDatastore` object. Load the associated rectangle label data by using a `boxLabelDatastore` (Computer Vision Toolbox) object.

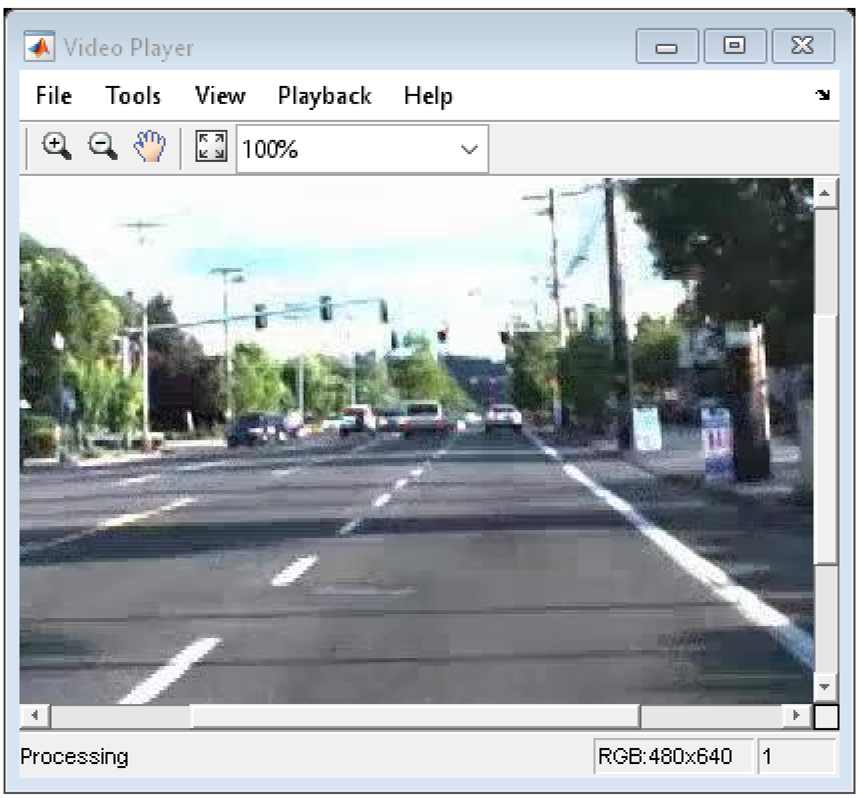
```
imds = imageDatastore(fileNames{1});
blds = boxLabelDatastore(labelData{1});
```

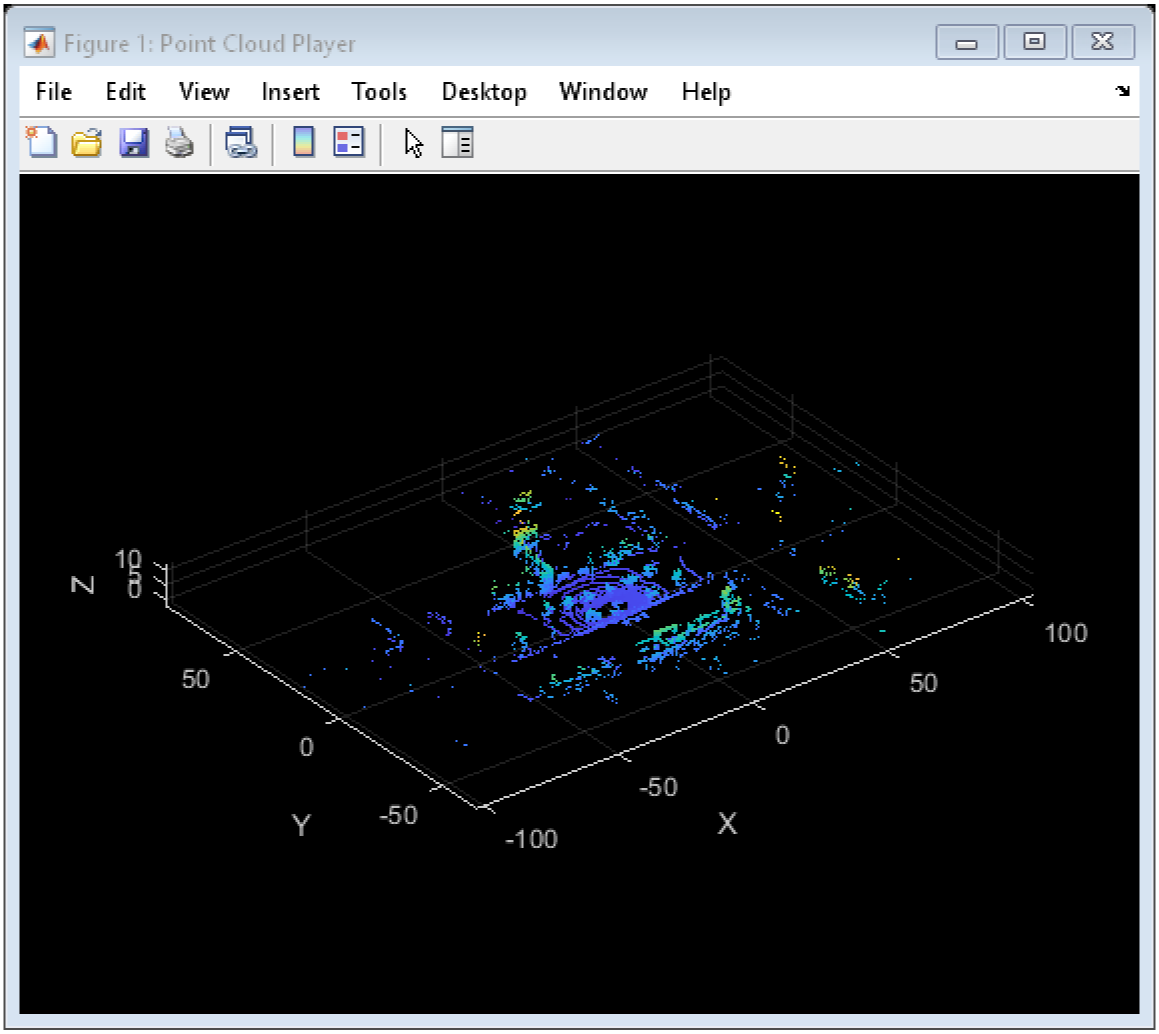
Load the written lidar signal frames by using a `fileDatastore` object. Load the associated cuboid label data by using a `boxLabelDatastore` object.

```
fds = fileDatastore(fileNames{2},'ReadFcn',@pcread);
clds = boxLabelDatastore(labelData{2});
```

Visualize the written video frames by using a `vision.VideoPlayer` (Computer Vision Toolbox) object. Visualize the written lidar frames by using a `pcplayer` (Computer Vision Toolbox) object.

```
videoPlayer = vision.VideoPlayer;

ptCloud = preview(fds);
ptCloudPlayer = pcplayer(ptCloud.XLimits,ptCloud.YLimits,ptCloud.ZLimits);

while hasdata(imds)
    % Read video and lidar frames.
    I = read(imds);
    ptCloud = read(fds);

    % Visualize video and lidar frames.
    videoPlayer(I);
    view(ptCloudPlayer,ptCloud);
end
```





Remove the path to the point cloud sequence folder.

```
rmpath(pcSeqDir);
```

Input Arguments

gTruth — Multisignal ground truth data
groundTruthMultisignal object | vector of groundTruthMultisignal objects

Multisignal ground truth data, specified as a groundTruthMultisignal object or vector of groundTruthMultisignal objects.

signalNames — Names of signals
character vector | cell array of character vectors | string scalar | string vector

Names of the signals for which to write frames, specified as a character vector, string scalar, cell array of character vectors, or string vector. The signal names must be valid signal names stored in the input multisignal ground truth data, gTruth.

To obtain the signal names from a `groundTruthMultisignal` object, use this syntax, where `gTruth` is the variable name of the object:

```
gTruth.DataSource.SignalName
```

Example: `'video_01_city_c2s_fcw_10s'`

Example: `"video_01_city_c2s_fcw_10s"`

Example: `{'video_01_city_c2s_fcw_10s','lidarSequence'}`

Example: `["video_01_city_c2s_fcw_10s" "lidarSequence"]`

`location` — Folder locations
matrix of strings | cell array of character vectors

Folder locations to which to write frames, specified as an `M`-by-`N` matrix of strings or an `M`-by-`N` cell array of character vectors, where:

- `M` is the number of `groundTruthMultisignal` objects in `gTruth`.
- `N` is the number of signals in `signalNames`.
- `location(m,n)` (for matrix inputs) or `location{m,n}` (for cell array inputs) contains the frame-writing folder location for the `n`th signal of `signalNames` that is in the `m`th `groundTruthMultisignal` object of `gTruth`.

You can specify folder locations as relative paths or full file paths. If any specified folder locations do not exist, the `writeFrames` function creates the folders. All folder locations must be unique. If files already exist in a specified folder location, and the existing files are writeable, then the `writeFrames` function overwrites them.

`timestamps` — Timestamps of frames to write
`duration` vector | cell array of `duration` vectors

Timestamps of the frames to write, specified as a `duration` vector or an `M`-by-`N` cell array of `duration` vectors, where:

- `M` is the number of `groundTruthMultisignal` objects in `gTruth`.
- `N` is the number of signals in `signalNames`.
- `timestamps{m,n}` contains the timestamps for the `n`th signal of `signalNames` that is in the `m`th `groundTruthMultisignal` object of `gTruth`.

If you are writing frames for only one signal and one `groundTruthMultisignal` object, specify `timestamps` as a single `duration` vector.

By default, the `writeFrames` function writes all signal frames. When a signal does not have a frame at the specified timestamps, the function writes the frame with the nearest preceding timestamp.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'NamePrefix',["video" "lidar"],'FileType',["png" "ply"]` writes video frames with file names of the format `video_001.png`, `video_002.png`, and so on, and writes lidar frames with file names of the format `lidar_001.ply`, `lidar_002.ply`, and so on.

NamePrefix — File name prefix for each signal
character vector | string scalar | cell array of character vectors | string vector

File name prefix for each signal in signalNames, specified as the comma-separated pair consisting of 'NamePrefix' and a character vector, string scalar, cell array of character vectors, or string vector.

Each element of 'NamePrefix' specifies the file type for the signal in the corresponding position of signalNames. By default, 'NamePrefix' is the name of each signal in signalNames.

FileType — File type for each signal
"jpg" for Image signals, "pcd" for PointCloud signals (default) | character vector | string scalar | cell array of character vectors | string vector

File type for each signal in signalNames, specified as the comma-separated pair consisting of 'FileType' and a character vector, string scalar, cell array of character vectors, or string vector.

Each element of 'FileType' specifies the file type for the signal in the corresponding position of signalNames. Use this name-value pair argument to specify the file extensions in the names of the written files.

The supported file types for a signal depend on whether that signal is of type Image or PointCloud.

Signal Type	Supported File Types
Image	All file types supported by the imwrite function
PointCloud	"pcd" or "ply" Point cloud data (PCD) and polygon (PLY) files are written using binary encoding. For more details on these file formats, see the pcwrite function.

To view the signal types for signals stored in a groundTruthMultisignal object, gTruth, use this code:

```
gTruth.DataSource.SignalType
```

Example: 'FileType','png'

Example: 'FileType',"png"

Example: 'FileType',{'png','ply'}

Example: 'FileType',["png" "ply"]

Verbose — Display writing progress information
trueor 1 (default) | false or 0

Display writing progress information at the MATLAB command line, specified as the comma-separated pair consisting of 'Verbose' and logical 1 (true) or 0 (false).

Output Arguments

fileNames — File names of written frames
cell array of string column vectors

File names of the written frames, returned as an `M`-by-`N` cell array of string vectors, where:

- `M` is the number of `groundTruthMultisignal` objects in `gTruth`.
- `N` is the number of signals in `signalNames`.
- `fileNames{m,n}` contains the file names for the frames of the `n`th signal of `signalNames` that is in the `m`th `groundTruthMultisignal` object of `gTruth`.

The file names for each signal are returned in a string column vector, where each row contains the file name for a written frame. If you specified the input `timestamps`, then each file name represents a written frame at the timestamp in the corresponding position of `timestamps`.

Each output file is named *`NamePrefix_UID.FileType`*, where:

- *`NamePrefix`* is the file name prefix. To set the file name prefix, use the `'NamePrefix'` name-value pair argument.
- *`UID`* is the unique integer index for each written frame. The `writeFrames` function generates these indices.
- *`FileType`* is the file type extension. To set the file type extension, use the `'FileType'` name-value pair argument.

See Also

`gatherLabelData` | `groundTruthMultisignal` | `imformats` | `imwrite` | `pcwrite`

Introduced in R2020a

changeFilePaths

Change file paths in multisignal ground truth data

Syntax

```
unresolvedPaths = changeFilePaths(gTruth,alternativePaths)
```

Description

`unresolvedPaths = changeFilePaths(gTruth,alternativePaths)` changes the file paths stored in a `groundTruthMultisignal` object, `gTruth`, based on pairs of current paths and alternative paths, `alternativePaths`. If `gTruth` is a vector of `groundTruthMultisignal` objects, the function changes the file paths across all objects. The function returns the unresolved paths in `unresolvedPaths`. An unresolved path is any current path in `alternativePaths` not found in `gTruth` or any alternative path in `alternativePaths` not found at the specified path location. In both cases, `unresolvedPaths` returns only the current paths.

Use this function to update the file paths of ground truth data that changes folder locations. You can change file paths for the ground truth data sources and pixel label data.

Examples

Change File Paths in Multisignal Ground Truth Data

Change the file paths to the data sources and pixel label data in a `groundTruthMultisignal` object.

Load a `groundTruthMultisignal` object containing ground truth data into the workspace. The data source and pixel label data of the object contain file paths corresponding to an image sequence showing a building. MATLAB® displays a warning that the path to the data source cannot be found.

```
load('gTruthMultiOldPaths.mat')
```

```
Warning: The data source for the following source names could not be loaded. Update the data sou
  'C:\Sources\building'
```

Display the current path to the data source.

```
gTruth.DataSource
```

```
ans = 
  ImageSequenceSource with properties:

           Name: "Image Sequence"
    Description: "An image sequence reader"
     SourceName: "C:\Sources\building"
   SourceParams: [1×1 struct]
     SignalName: "building"
     SignalType: Image
      Timestamp: {[5×1 duration]}
```

```
      NumSignals: 1
```

Specify the current path to the data source and an alternative path and store these paths in a cell array. Use the `changeFilePaths` function to update the data source path based on the paths in the cell array. Because the function does not find the pixel label data at the specified new path, it returns the current unresolved paths.

```
currentPathDataSource = "C:\Sources\building";
newPathDataSource = fullfile(matlabroot,"toolbox\vision\visiondata\building");
alternativePaths = {[currentPathDataSource newPathDataSource]};
unresolvedPaths = changeFilePaths(gTruth,alternativePaths)
```

```
unresolvedPaths = 5×1 string
    "C:\Pixels\Label_1.png"
    "C:\Pixels\Label_2.png"
    "C:\Pixels\Label_3.png"
    "C:\Pixels\Label_4.png"
    "C:\Pixels\Label_5.png"
```

Verify that the paths in the `groundTruthMultisignal` object match the unresolved paths returned by the `changeFilePaths` function. The unresolved paths are stored in the `ROILabelData` property of the `groundTruthMultisignal` object, in the `PixelLabelData` column of the table for the `building` image sequence signal.

```
gTruth.ROILabelData.building.PixelLabelData
```

```
ans = 5×1 cell
    {'C:\Pixels\Label_1.png'}
    {'C:\Pixels\Label_2.png'}
    {'C:\Pixels\Label_3.png'}
    {'C:\Pixels\Label_4.png'}
    {'C:\Pixels\Label_5.png'}
```

Specify the current path and an alternative path for the pixel label files and change the file paths. The function updates the paths for all pixel labels. Because the function resolves all paths, it returns an empty array of unresolved paths.

```
currentPathPixels = "C:\Pixels";
newPathPixels = fullfile(matlabroot,"toolbox\vision\visiondata\buildingPixellabels");
alternativePaths = {[currentPathPixels newPathPixels]};
unresolvedPaths = changeFilePaths(gTruth,alternativePaths)
```

```
unresolvedPaths = 

  0×0 empty string array
```

To view the new data source path, use the `gTruth.DataSource` command. To view the new pixel label data paths, use the `gTruth.ROILabelData.building.PixelLabelData` command.

Input Arguments

gTruth — Multisignal ground truth data
`groundTruthMultisignal` object | vector of `groundTruthMultisignal` objects

Multisignal ground truth data, specified as a `groundTruthMultisignal` object or vector of `groundTruthMultisignal` objects.

`alternativePaths` — Alternative file paths
1-by-2 string vector | cell array of 1-by-2 string vectors

Alternative file paths, specified as a 1-by-2 string vector or cell array of 1-by-2 string vectors of the form [$p_{current}$ p_{new}].

- $p_{current}$ is a current file path in `gTruth`. This file path can be from the data source or pixel label data of `gTruth`. Specify $p_{current}$ using backslashes as the path separators.
- p_{new} is the new path to which you want to change $p_{current}$. Specify p_{new} using either forward slashes or backslashes as the path separators.

You can specify alternatives paths to these files.

- Signal data sources — The `DataSource` property of `gTruth` contains one `MultiSignalSource` object per signal. The `changeFilePaths` function updates the signal paths stored in these objects.
- Pixel label data — The `ROILabelData` property of `gTruth` contains an `ROILabelData` object, which contains a table of ROI label data for each signal. For signals with pixel label data, which is stored in the `PixelLabelData` column of the table for that signal, the function updates the paths to the pixel label data.

If `gTruth` is a vector of `groundTruthMultisignal` objects, the function changes the file paths across all objects.

Example: `["C:\Pixels\PixelLabelData_1" "C:\Pixels\PixelLabelData_2]` changes the path to the pixel label data folder. The function updates the path in all pixel label files stored in that folder.

Example: `{["B:\Sources\video1.mp4" "C:\Sources\video1.mp4"]; ["B:\Sources\video2.mp4" "C:\Sources\video2.mp4"]}` changes the drive letter in the paths to the data sources.

Output Arguments

`unresolvedPaths` — Unresolved file paths
string array

Unresolved file paths, returned as a string array. If the `changeFilePaths` function cannot find either the current path or new path in the string vectors specified by the `alternativePaths` input, then it returns the unresolved current paths in `unresolvedPaths`.

If the function finds and resolves all file paths, then it returns `unresolvedPaths` as an empty string array.

See Also

`groundTruthMultisignal`

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

Introduced in R2020a

geoplayer

Visualize streaming geographic map data

Description

A `geoplayer` object is a geographic player that displays the streaming coordinates of a driving route on a map.

- To display the driving route of a vehicle, use the `plotRoute` function.
- To display the position of a vehicle as it drives along a route, use the `plotPosition` function. You can plot the position of multiple vehicles on different routes simultaneously by specifying a unique track ID for each route. For more information, see the `'TrackID'` name-value pair argument on `plotPosition`.
- To change the underlying map, or basemap, of the `geoplayer` object, update the `Basemap` property of the object. For more information, see “Custom Basemaps” on page 4-628.

Creation

Syntax

```
player = geoplayer(latCenter,lonCenter)
player = geoplayer(latCenter,lonCenter,zoomLevel)
player = geoplayer(___,Name,Value)
```

Description

`player = geoplayer(latCenter,lonCenter)` creates a geographic player, centered at latitude coordinate `latCenter` and longitude coordinate `lonCenter`.

`player = geoplayer(latCenter,lonCenter,zoomLevel)` creates a geographic player with a map magnification specified by `zoomLevel`.

`player = geoplayer(___,Name,Value)` sets properties on page 4-610 using one or more name-value pairs, in addition to specifying input arguments from previous syntaxes. For example, `geoplayer(45,0,'HistoryDepth',5)` creates a geographic player centered at the latitude-longitude coordinate (45, 0), and sets the `HistoryDepth` property such that the player displays the five previous geographic coordinates.

Input Arguments

`latCenter` — Latitude coordinate
real scalar in the range (–90, 90)

Latitude coordinate at which the geographic player is centered, specified as a real scalar in the range (–90, 90).

Data Types: `single` | `double`

lonCenter — Longitude coordinate
real scalar in the range [–180, 180]

Longitude coordinate at which the geographic player is centered, specified as a real scalar in the range [–180, 180].

Data Types: single | double

zoomLevel — Magnification
15 | integer in the range [0, 25]

Magnification of the geographic player, specified as an integer in the range [0, 25]. This magnification occurs on a logarithmic scale with base 2. Increasing zoomLevel by one doubles the map scale.

Properties

HistoryDepth — Number of previous geographic coordinates to display
0 (default) | nonnegative integer | Inf

Number of previous geographic coordinates to display, specified as a nonnegative integer or Inf. A value of 0 displays only the current geographic coordinates. A value of Inf displays all geographic coordinates previously plotted using the plotPosition function.

You can set this property only when you create the object. After you create the object, this property is read-only.

HistoryStyle — Style of displayed geographic coordinates
'point' (default) | 'line'

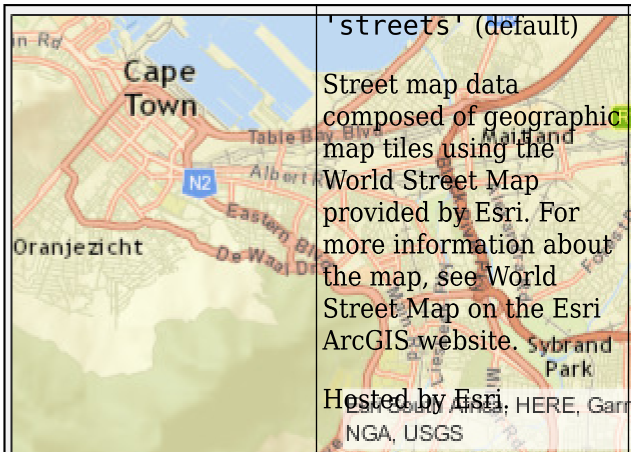
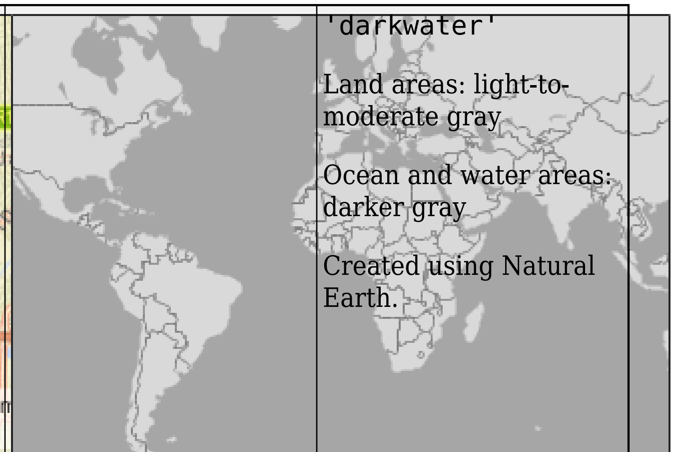
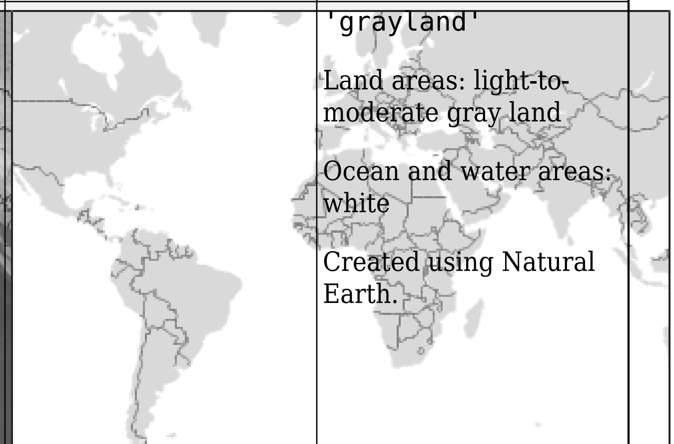
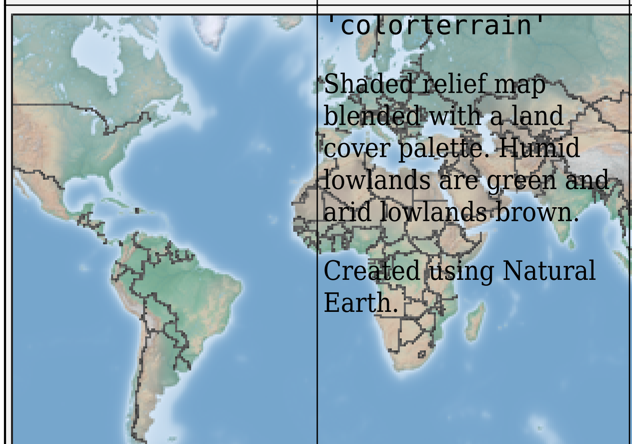
Style of displayed geographic coordinates, specified as one of these values:

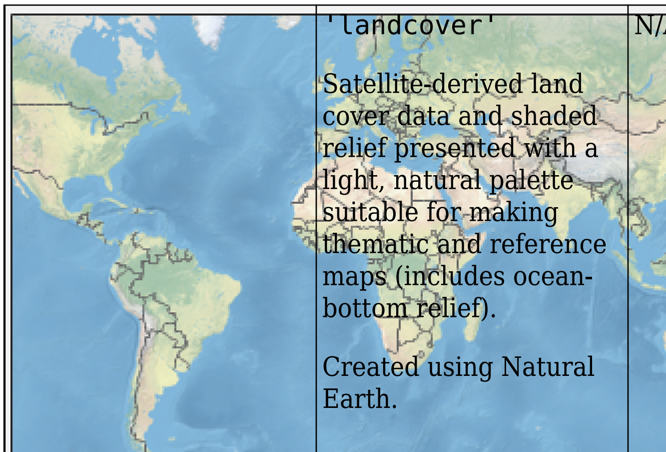
- 'point' — Display the coordinates as discrete, unconnected points.
- 'line' — Display the coordinates as a single connected line.

You can set this property when you create the object. After you create the object, this property is read-only.

Basemap — Map on which to plot data
'streets' (default) | 'darkwater' | 'grayterrain' | 'grayland' | 'colorterrain' | ...

Map on which to plot data, specified as one of the basemap names in this table, 'none', or a custom basemap defined using the addCustomBasemap function. For more information on adding custom basemaps, see “Custom Basemaps” on page 4-628. For examples on how to add custom basemaps, see “Display Data on OpenStreetMap Basemap” on page 4-615 and “Display Map Data on HERE Basemap” on page 4-620.

	'streets' (default) Street map data composed of geographic map tiles using the World Street Map provided by Esri. For more information about the map, see World Street Map on the Esri ArcGIS website. Hosted by Esri.		'darkwater' Land areas: light-to-moderate gray Ocean and water areas: darker gray Created using Natural Earth.
	'grayterrain' Worldwide terrain depicted monochromatically in shades of gray, combining shaded relief that emphasizes both high mountains and the micro terrain found in lowlands. Created using Natural Earth.		'grayland' Land areas: light-to-moderate gray land Ocean and water areas: white Created using Natural Earth.
	'colorterrain' Shaded relief map blended with a land cover palette. Humid lowlands are green and arid lowlands brown. Created using Natural Earth.		'bluegreen' Land areas: light green Ocean and water areas: light blue Created using Natural Earth.

	'landcover' Satellite-derived land cover data and shaded relief presented with a light, natural palette suitable for making thematic and reference maps (includes ocean-bottom relief). Created using Natural Earth.	N/A	'none' Geographic axes plots your data with latitude-longitude grid, ticks, and labels but does not include a map.

By default, access to basemaps requires an Internet connection. The exception is the `'darkwater'` basemap, which is installed with MATLAB.

If you do not have consistent access to the Internet, you can download the basemaps hosted by MathWorks® onto your local system. For more information about downloading basemaps, see “Access Basemaps for Geographic Axes and Charts”. You cannot download basemaps hosted by Esri.

Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Example: `player = geoplayer(latCenter,lonCenter,'Basemap','darkwater')`

Example: `player.Basemap = 'darkwater'`

Data Types: `char` | `string`

`CenterOnID` — Recenter display based on specified track ID
`[]`(center on first track) (default) | positive integer

Recenter display based on the specified track ID, specified as a positive integer. The `geoplayer` object recenters the map when the new position, specified by `latCenter` and `lonCenter`, moves outside of the current viewable map area. You can also use this property to recenter the map on a previously drawn track that is outside of the viewable area. Define the track ID by using the `'TrackID'` name-value pair argument when you call the `plotPosition` object function.

`Parent` — Parent axes of geographic player
`Figure` graphics object | `Panel` graphics object

Parent axes of the geographic player, specified as a `Figure` graphics object or `Panel` graphics object. If you do not specify `Parent`, then `geoplayer` creates the geographic player in a new figure.

You can set this property when you create the object. After you create the object, this property is read-only.

`Axes` — Axes used by geographic player
`GeographicAxes` object

Axes used by geographic player, specified as a `GeographicAxes` object. Use this axes to customize the map that the geographic player displays. For an example, see “Customize Geographic Axes” on page 4-622. For details on the properties that you can customize, see GeographicAxes Properties.

Object Functions

plotPosition	Display current position in geoplayer figure
plotRoute	Display continuous route in geoplayer figure
reset	Remove all existing plots from geoplayer figure
show	Make geoplayer figure visible
hide	Make geoplayer figure invisible
isOpen	Return true if geoplayer figure is visible

Examples

Animate Sequence of Latitude and Longitude Coordinates

Load a sequence of latitude and longitude coordinates.

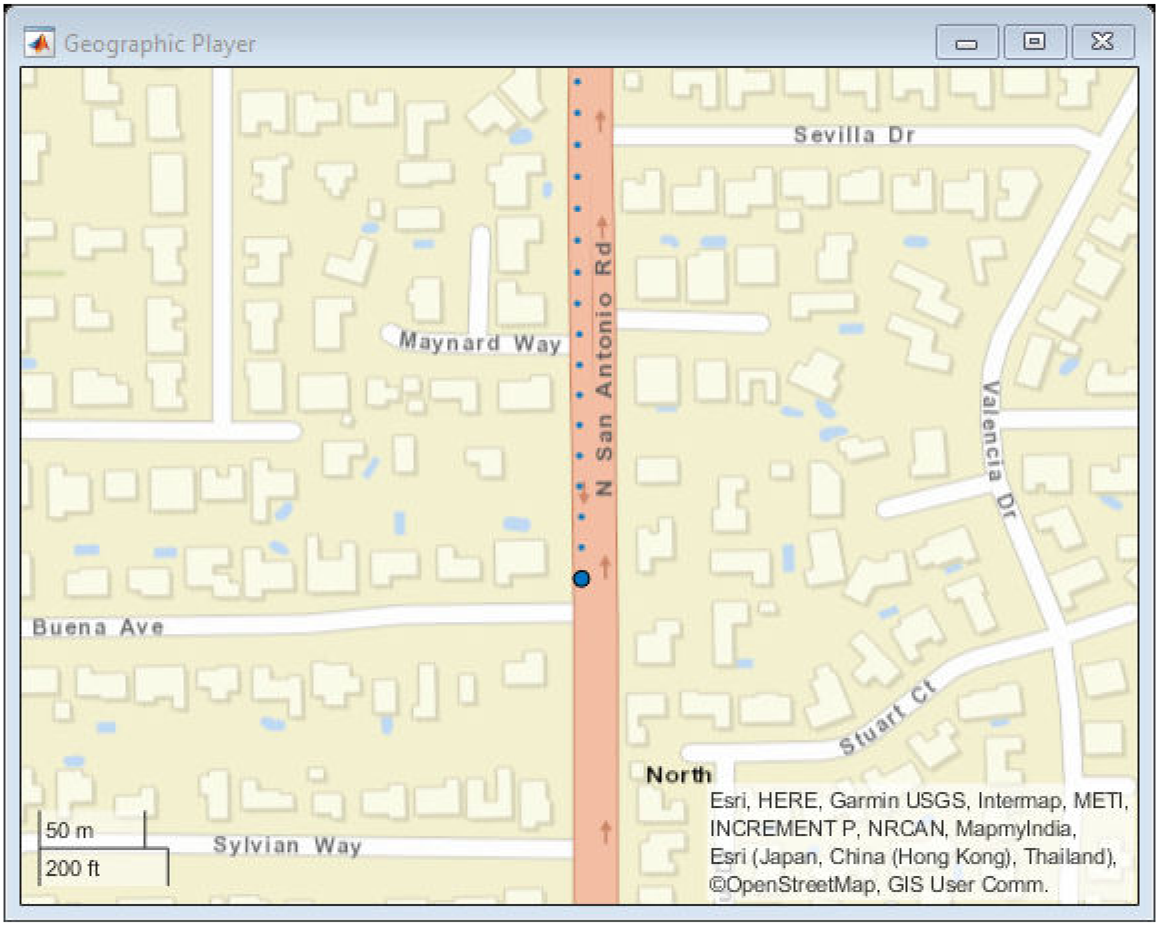
```
data = load('geoSequence.mat');
```

Create a geographic player and configure it to display all points in its history.

```
zoomLevel = 17;
player = geoplayer(data.latitude(1),data.longitude(1),zoomLevel,'HistoryDepth',Inf);
```

Display the sequence of coordinates.

```
for i = 1:length(data.latitude)
    plotPosition(player,data.latitude(i),data.longitude(i));
    pause(0.01)
end
```



View Position of Vehicle Along Route

Load a sequence of latitude and longitude coordinates.

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

Create a geographic player and set the zoom level to 12. Compared to the default zoom level, this zoom level zooms the map out by a factor of 8.

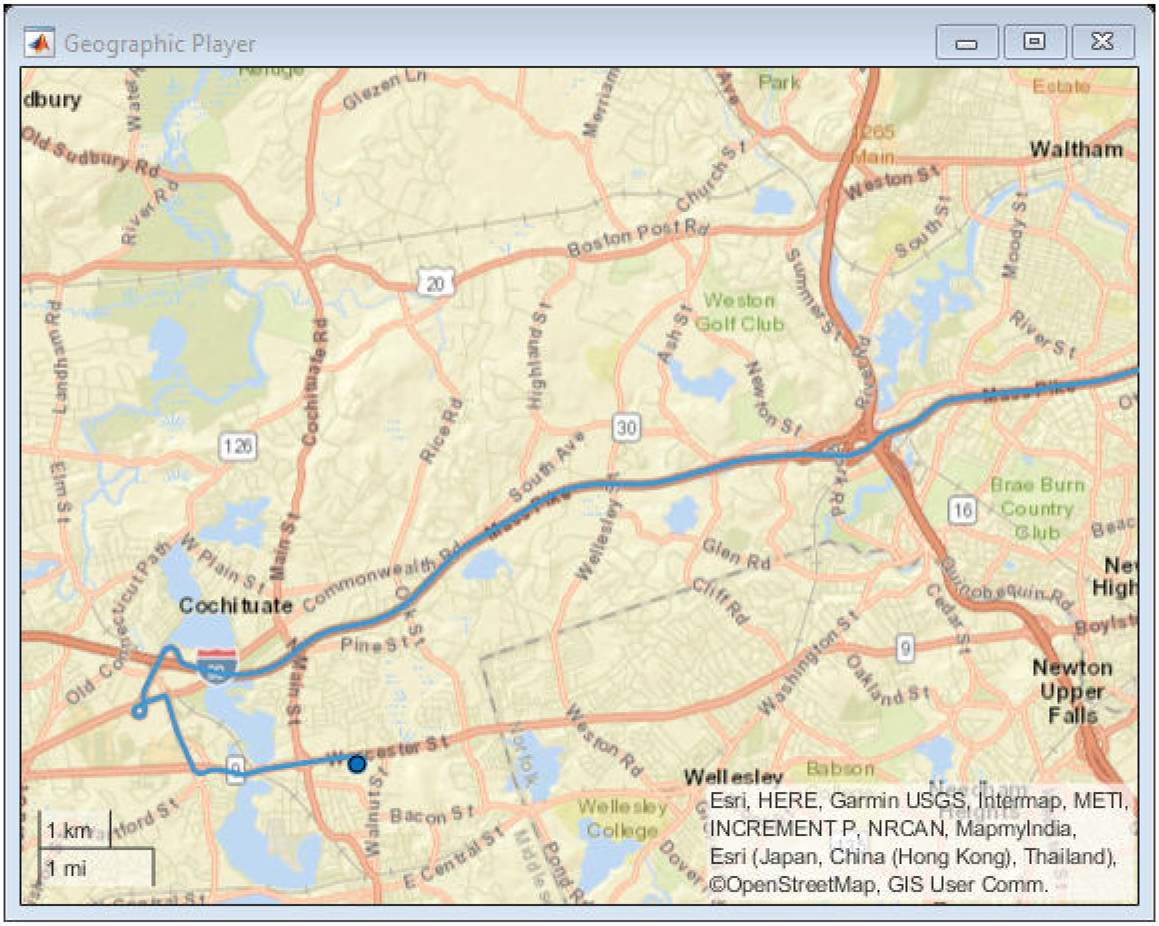
```
player = geoplayer(data.latitude(1),data.longitude(1),12);
```

Display the full route.

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

Display the coordinates in a sequence. The circle marker indicates the current position.

```
for i = 1:length(data.latitude)
    plotPosition(player,data.latitude(i),data.longitude(i));
    pause(0.05)
end
```



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.

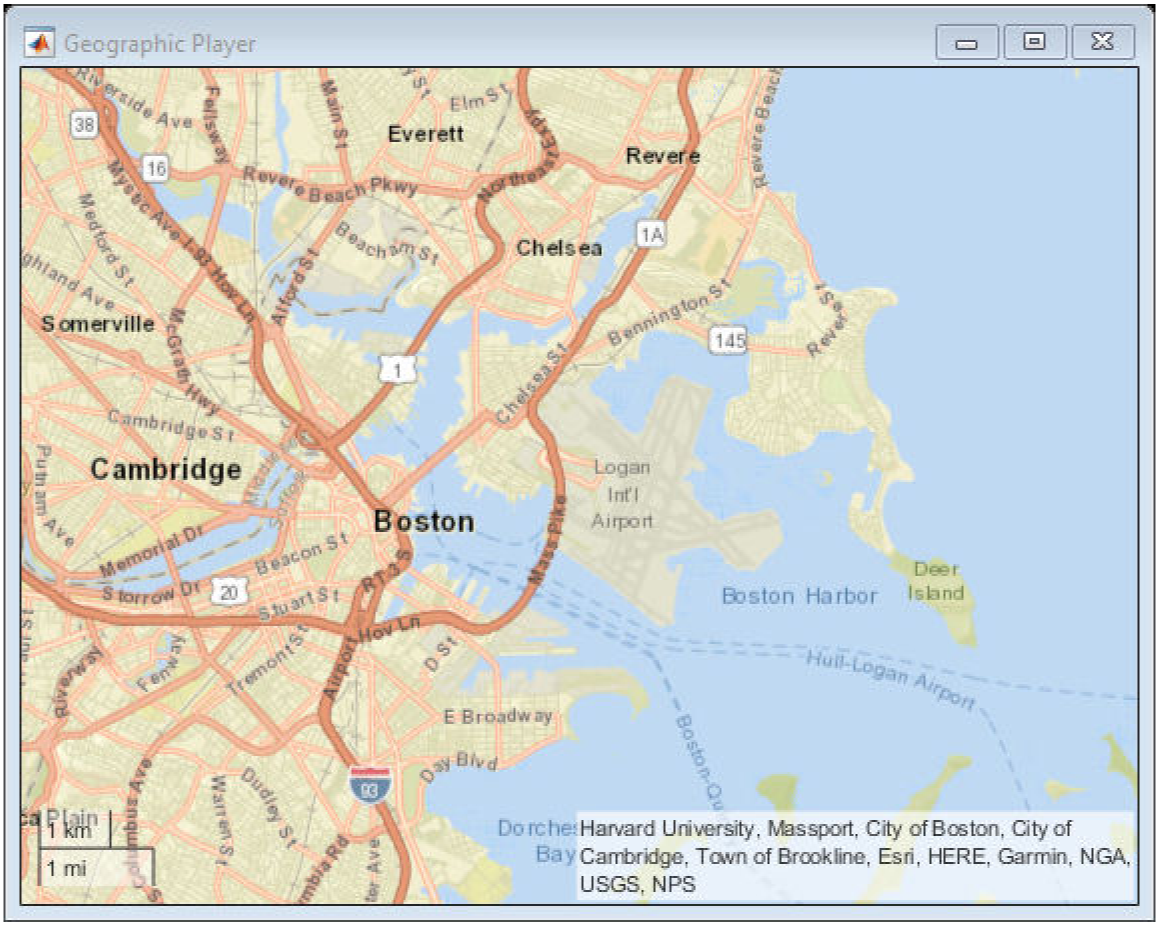
```
name = 'openstreetmap';
url = 'https://a.tile.openstreetmap.org/${z}/${x}/${y}.png';
copyright = char(uint8(169));
attribution = copyright + "OpenStreetMap contributors";
addCustomBasemap(name,url,'Attribution',attribution)
```

Load a sequence of latitude and longitude coordinates.

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

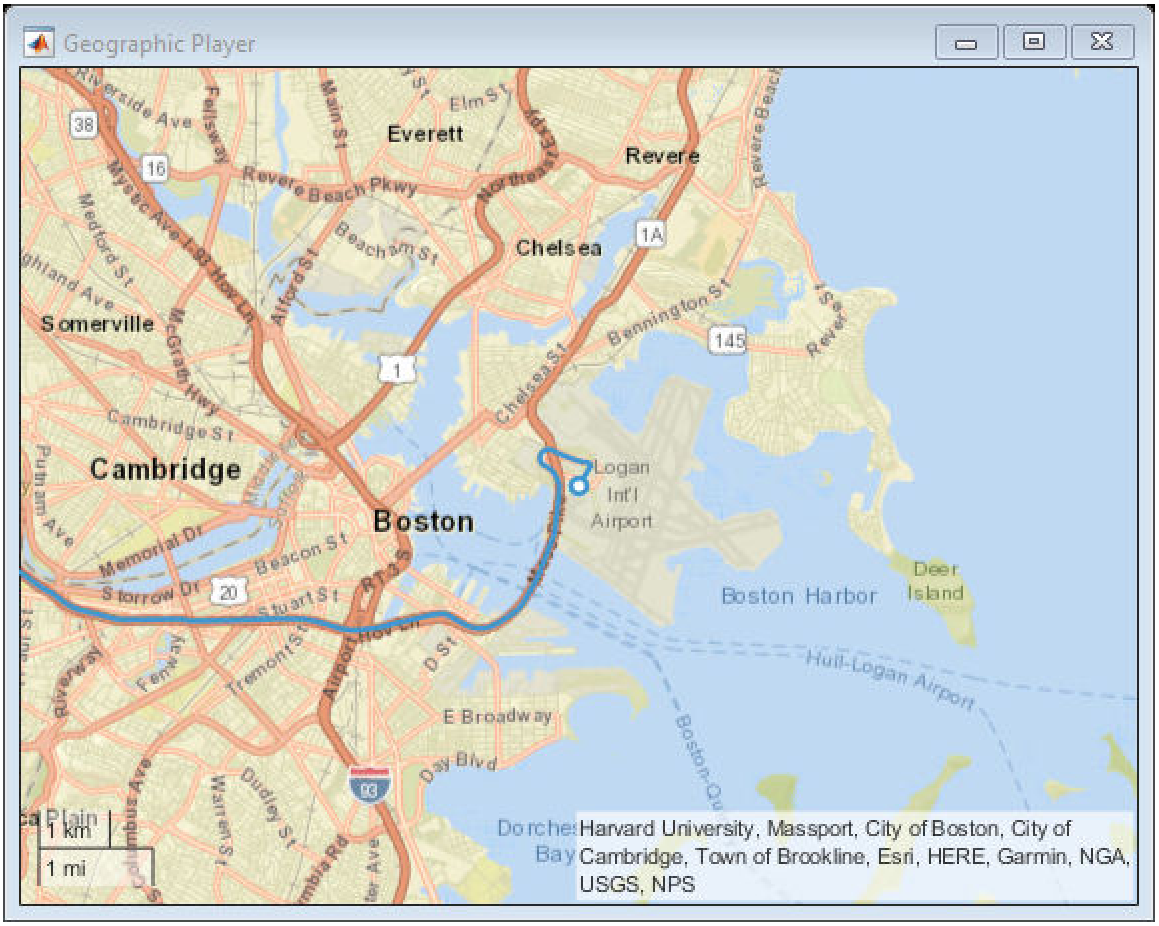
Create a geographic player. Center the geographic player on the first position of the driving route and set the zoom level to 12.

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



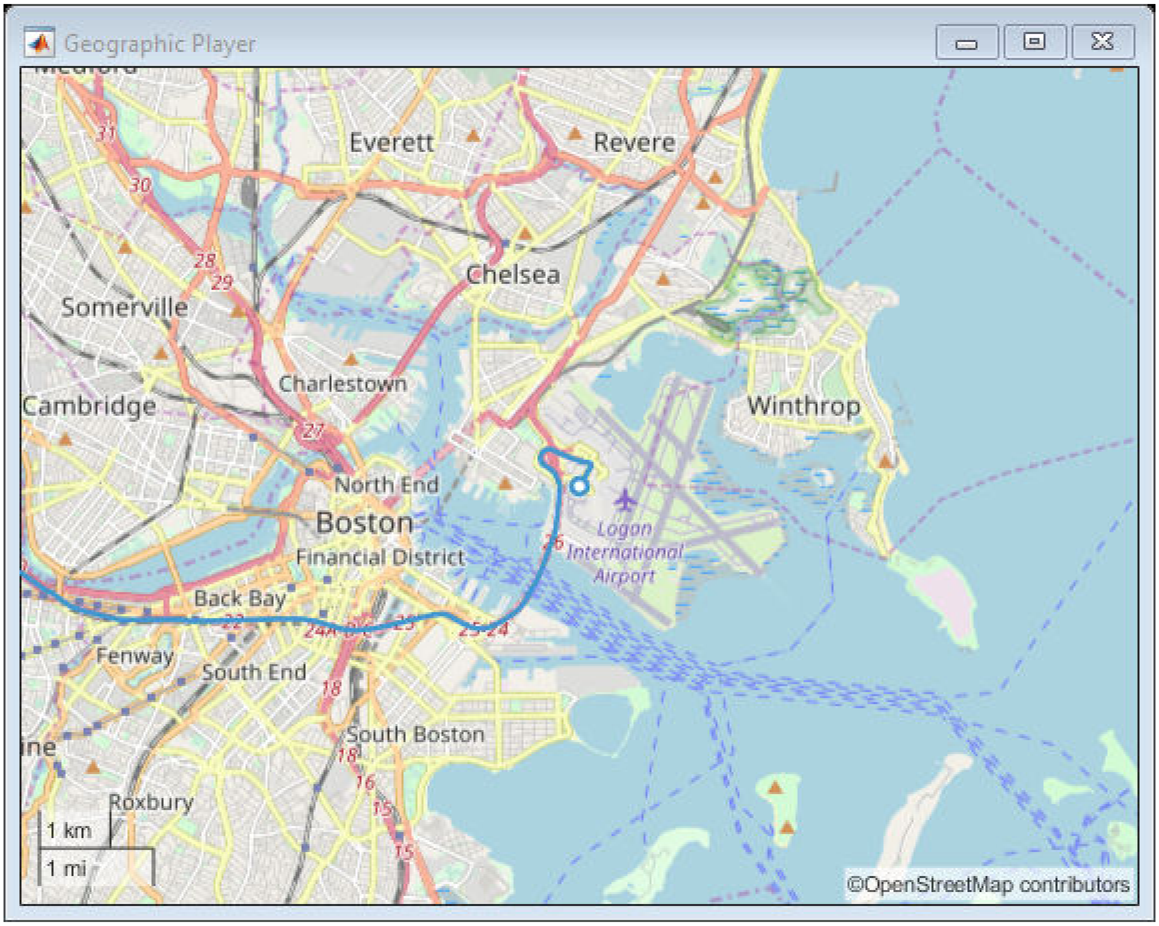
Display the full route.

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



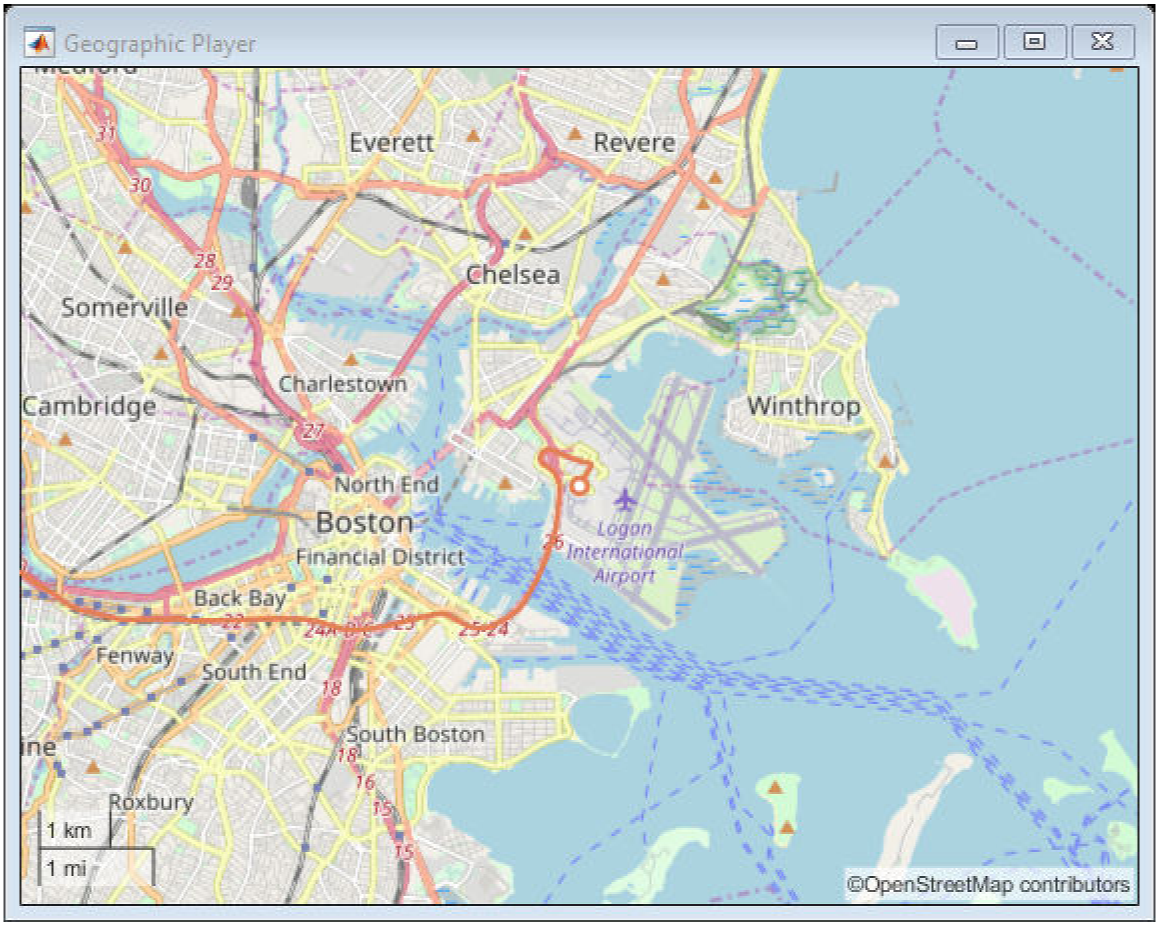
By default, the geographic player uses the World Street Map basemap (`'streets'`) provided by Esri®. Update the geographic player to use the added OpenStreetMap basemap instead.

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



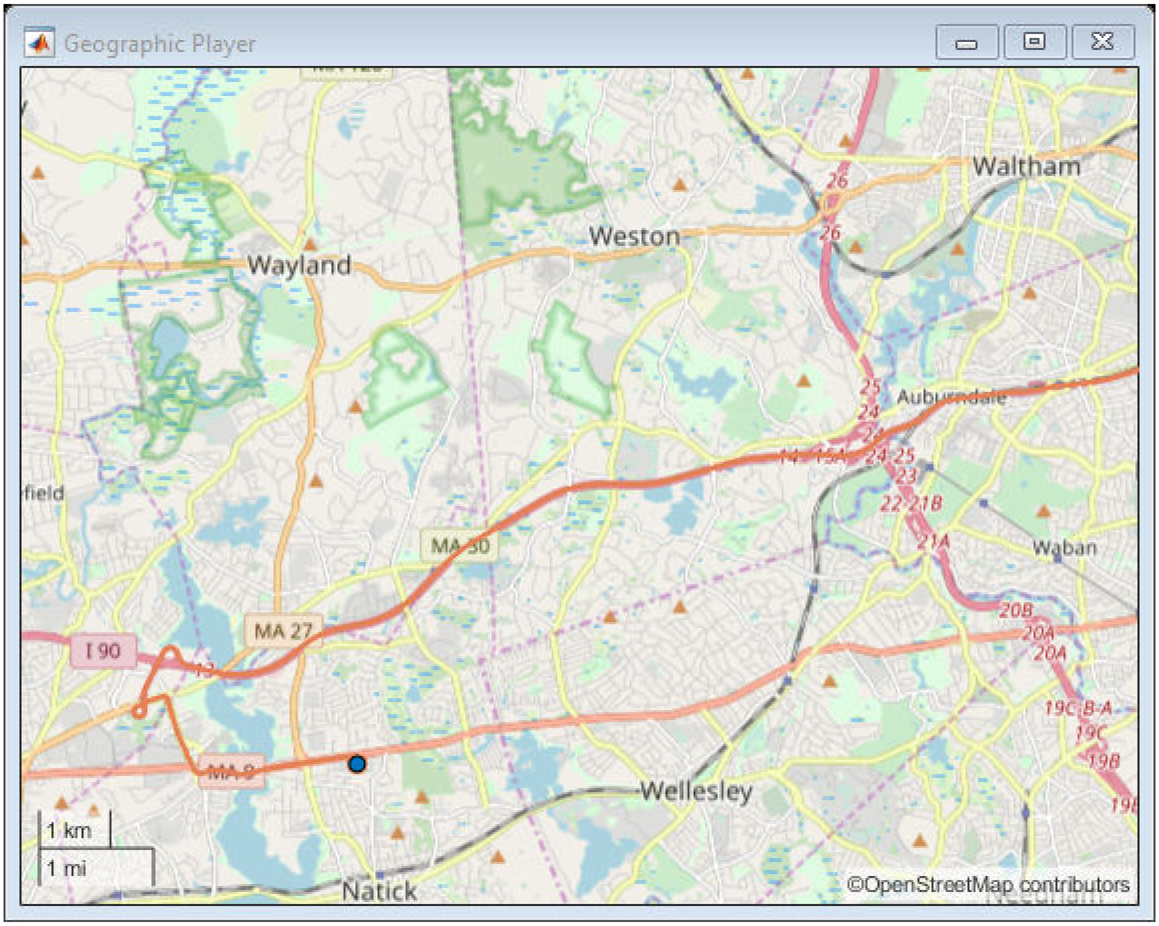
Display the route again.

```
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
```



Display Map Data on HERE Basemap

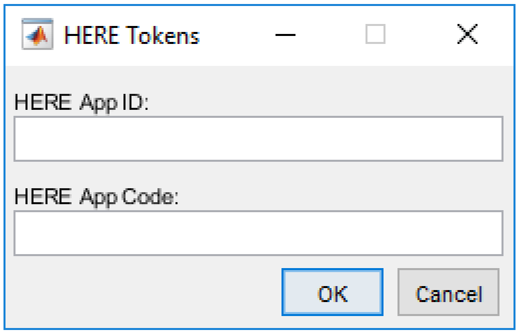
Display a driving route on a basemap provided by HERE Technologies. To use this example, you must have a valid license from HERE Technologies.

Specify the basemap name and map URL.

```
name = 'herestreets';
url  = ['https://2.base.maps.cit.api.here.com/maptile/2.1/maptile/', ...
        'newest/normal.day/${z}/${x}/${y}/256/png?app_id=%s&app_code=%s'];
```

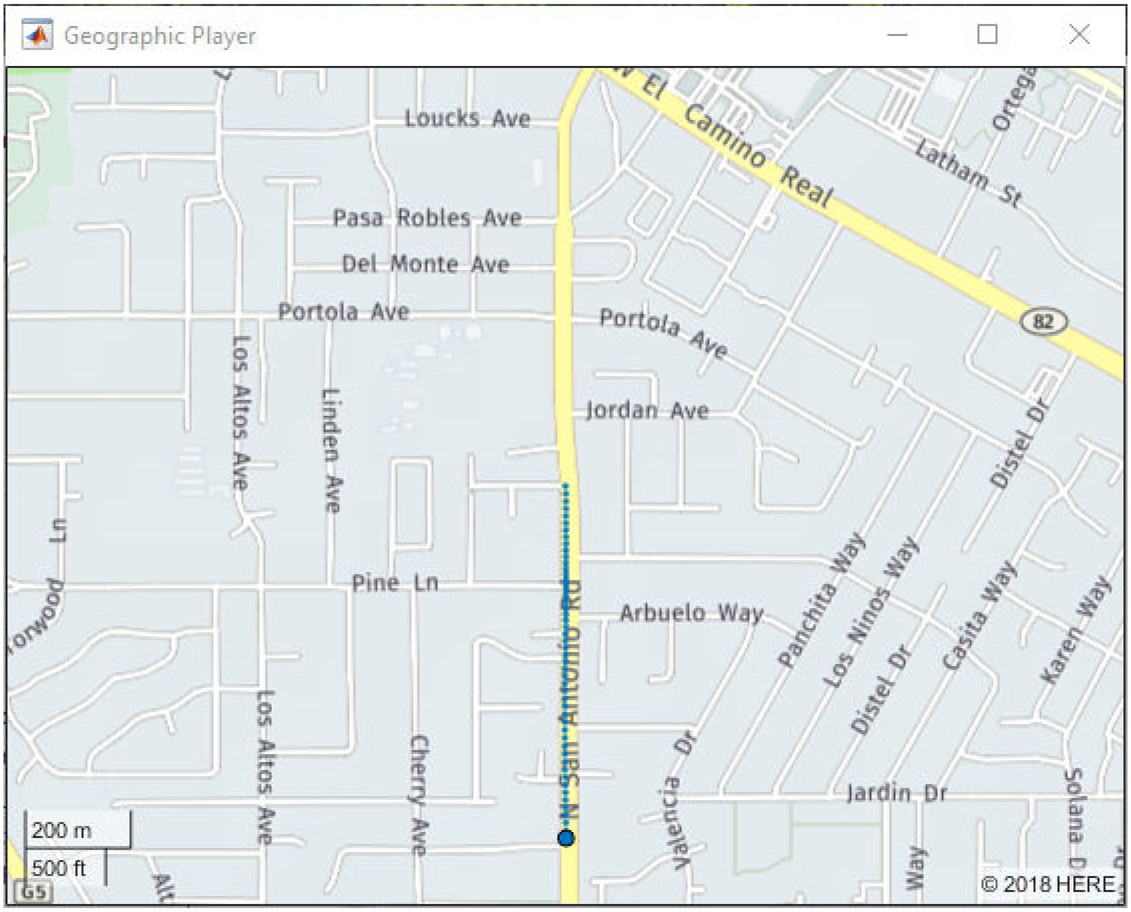
Maps from HERE Technologies require a valid license. Create a dialog box. In the dialog box, enter the App ID and App Code corresponding to your HERE license.

```
prompt = {'HERE App ID:','HERE App Code:'};
title = 'HERE Tokens';
dims = [1 40]; % Text edit field height and width
hereTokens = inputdlg(prompt,title,dims);
```



If the license is valid, specify the HERE credentials and a custom attribution, load coordinate data, and display the coordinates on the HERE basemap using a `geoplayer` object. If the license is not valid, display an error message.

```
if ~isempty(hereTokens)

    % Add HERE basemap with custom attribution.
    url = sprintf(url,hereTokens{1},hereTokens{2});
    copyrightSymbol = char(169); % Alt code
    attribution = [copyrightSymbol,' ',datestr(now,'yyyy'),' HERE'];
    addCustomBasemap(name,url,'Attribution',attribution);

    % Load sample lat,lon coordinates.
    data = load('geoSequence.mat');

    % Create geoplayer with HERE basemap.
    player = geoplayer(data.latitude(1),data.longitude(1), ...
        'Basemap','herestreets','HistoryDepth',Inf);

    % Display the coordinates in a sequence.
    for i = 1:length(data.latitude)
        plotPosition(player,data.latitude(i),data.longitude(i));
    end

else
    error('You must enter valid credentials to access maps from HERE Technologies');
end
```



Customize Geographic Axes

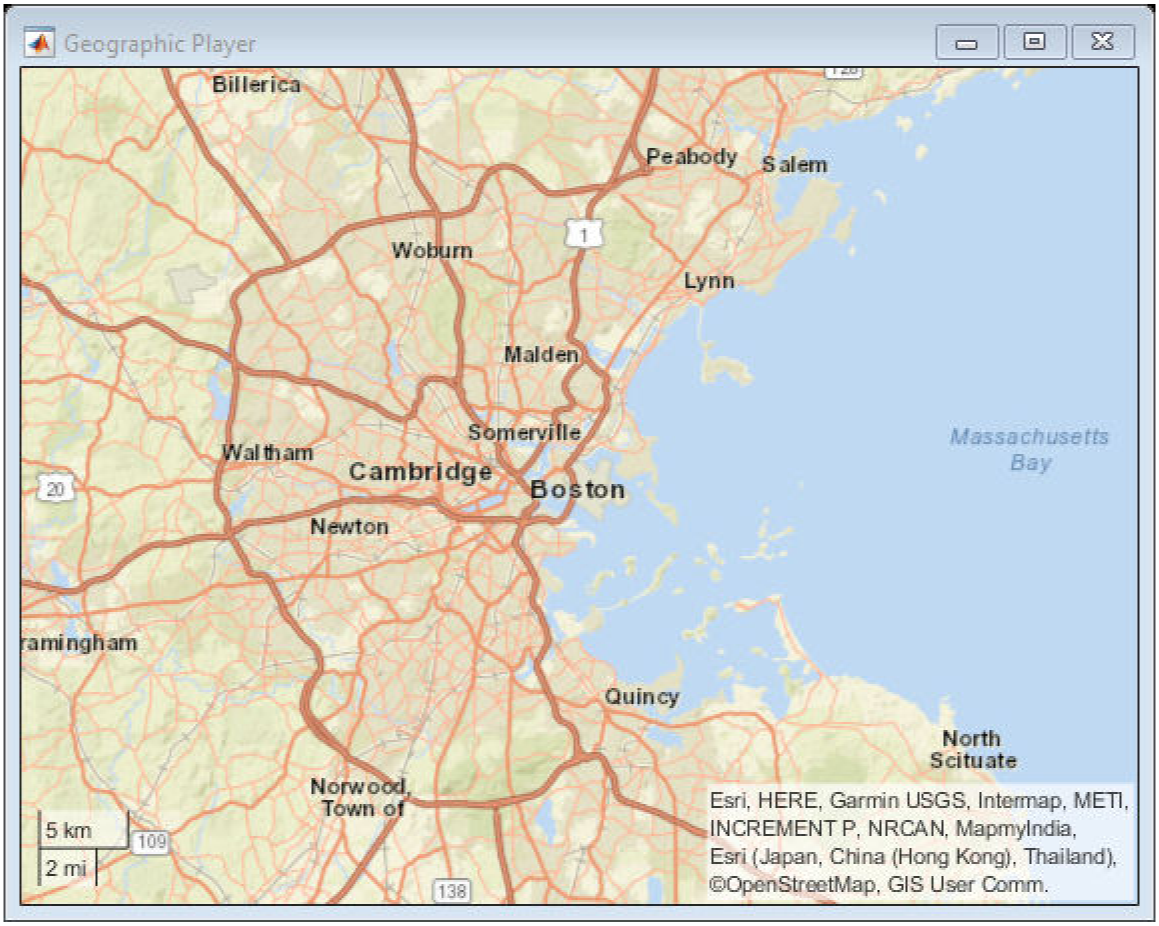
Customize the geographic axes of a `geoplayer` object by adding a custom line between route endpoints.

Load a driving route and vehicle positions along that route.

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

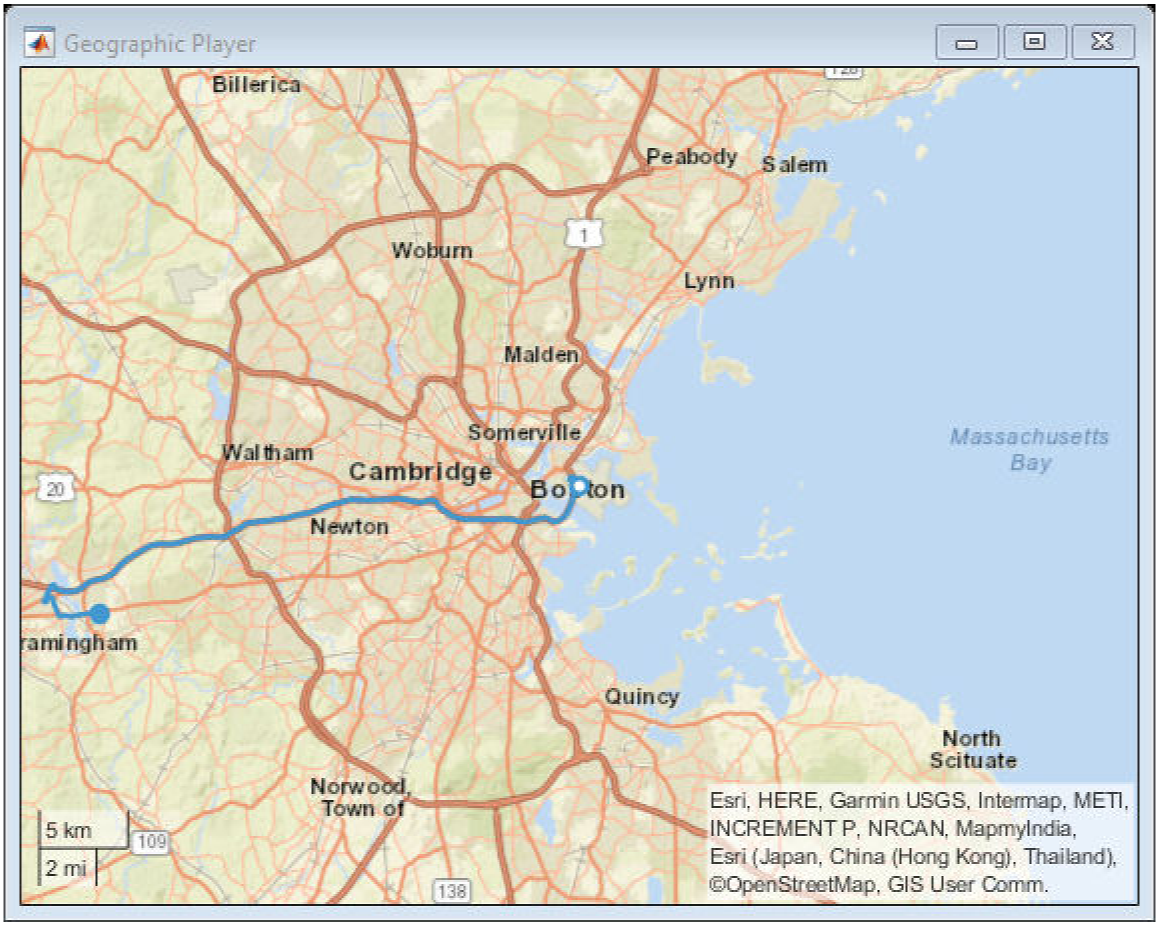
Create a geographic player that is centered on the first position of the vehicle.

```
zoomLevel = 10;
player = geoplayer(data.latitude(1),data.longitude(1),zoomLevel);
```



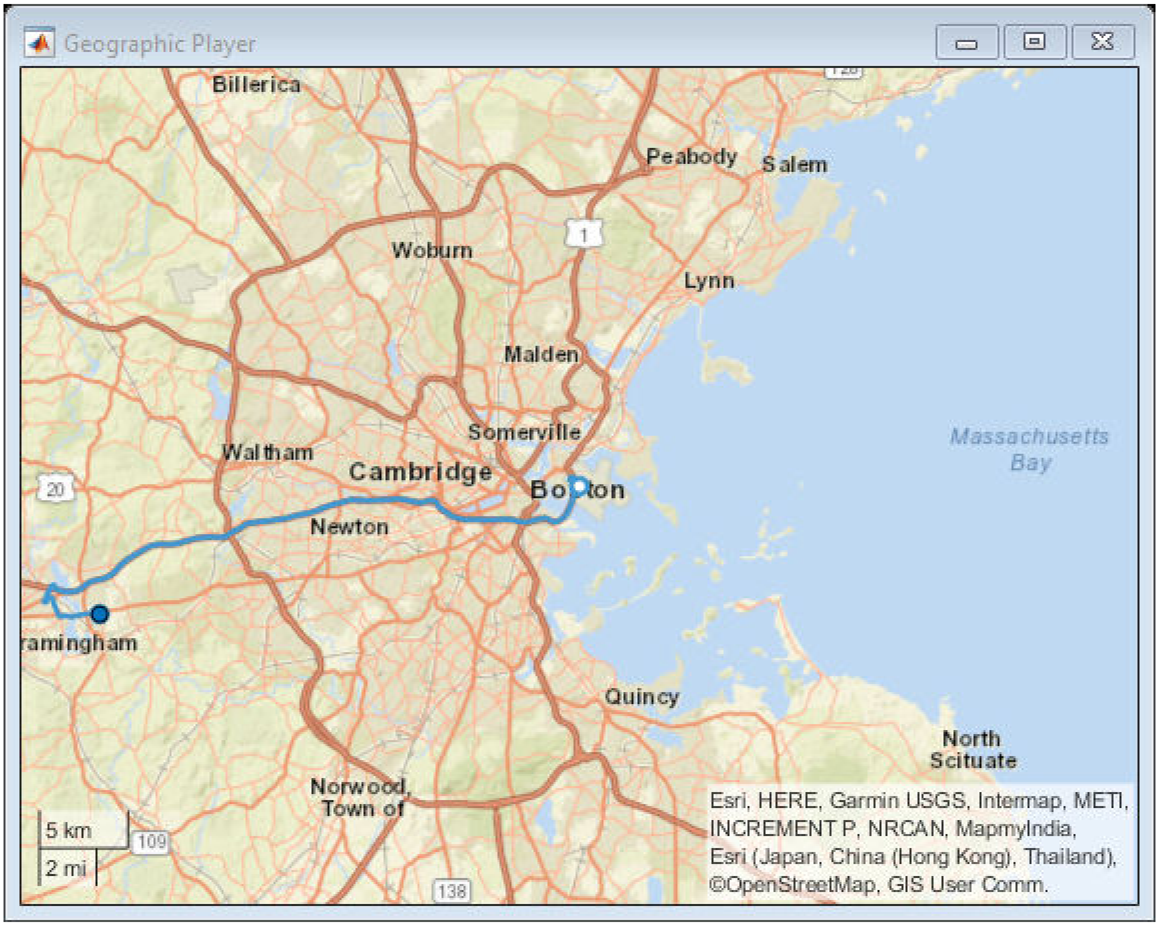
Display the full route.

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



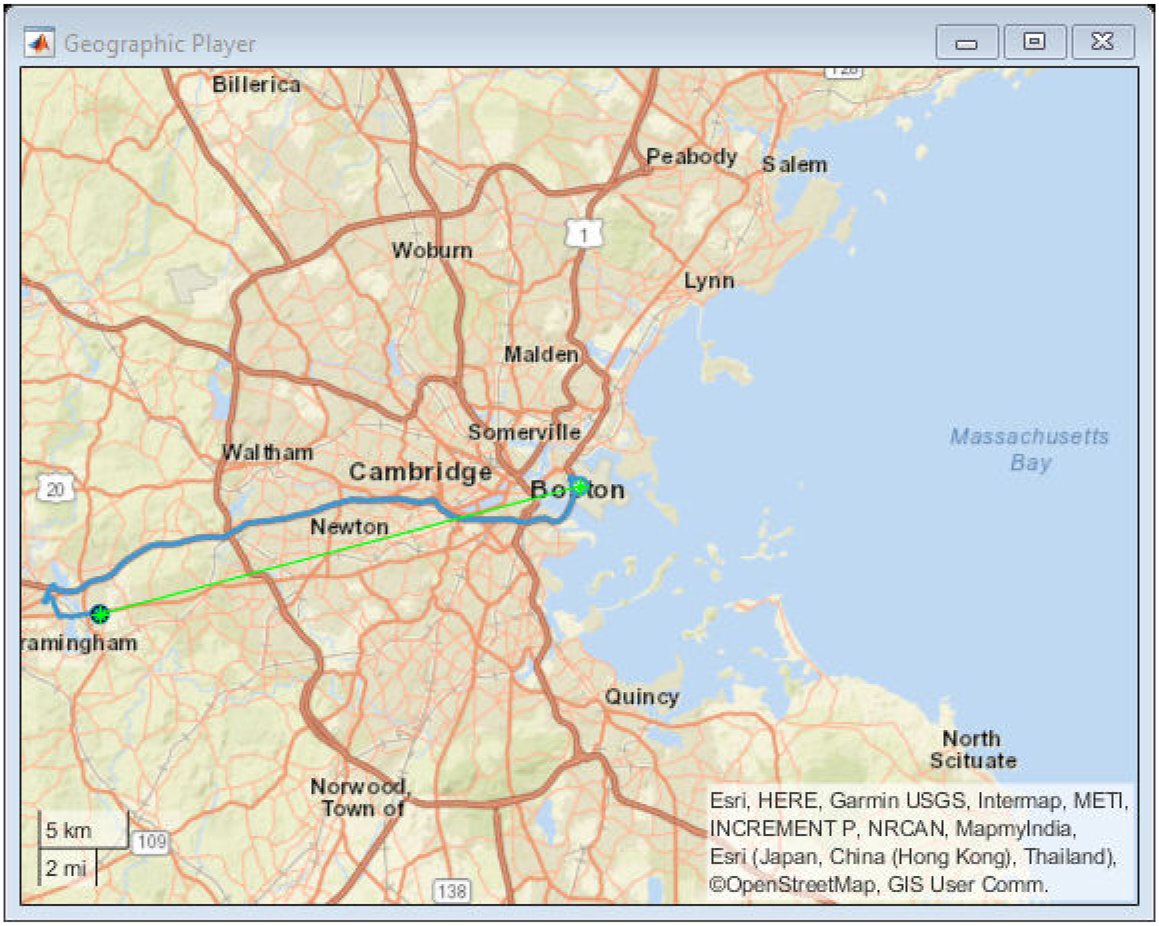
Display positions of the vehicle along the route.

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



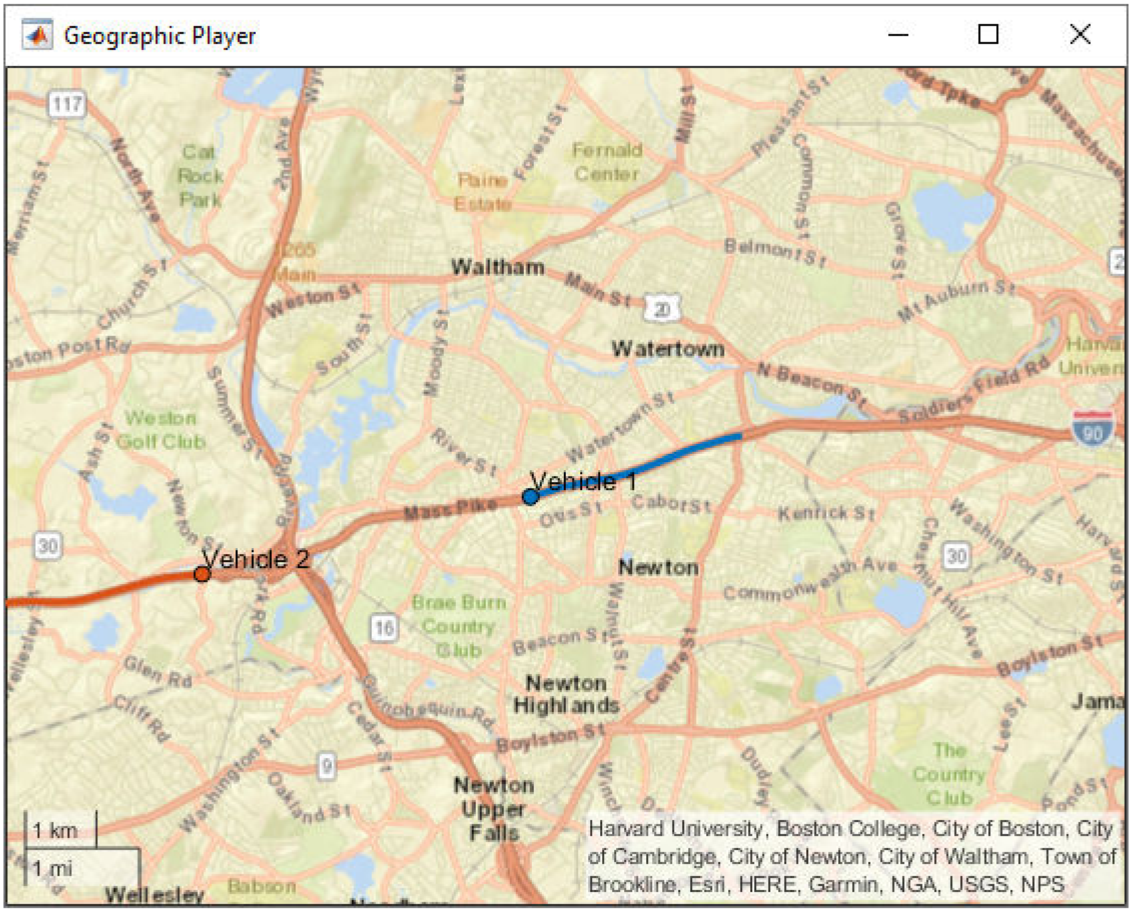
Customize the geographic axes by adding a line between the two endpoints of the route.

```
geoplot(player.Axes,[data.latitude(1) data.latitude(end)], ...
    [data.longitude(1) data.longitude(end)],'g-*')
```



Plot the Tracks of Two Vehicles

Plot multiple routes simultaneously in a geographic player. First, assign each route a unique identifier. Then, when plotting points on the routes using the `plotPosition` object function, specify the route identifier using the `'TrackID'` name-value pair argument. In this example, the routes are labeled Vehicle 1 and Vehicle 2. This screen capture shows the point where the two routes are about to cross paths.



Load data for a route.

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

Extract data for the first vehicle.

```
lat1 = data.latitude;
lon1 = data.longitude;
```

Create a synthetic route for the second vehicle that drives the same route in the opposite direction.

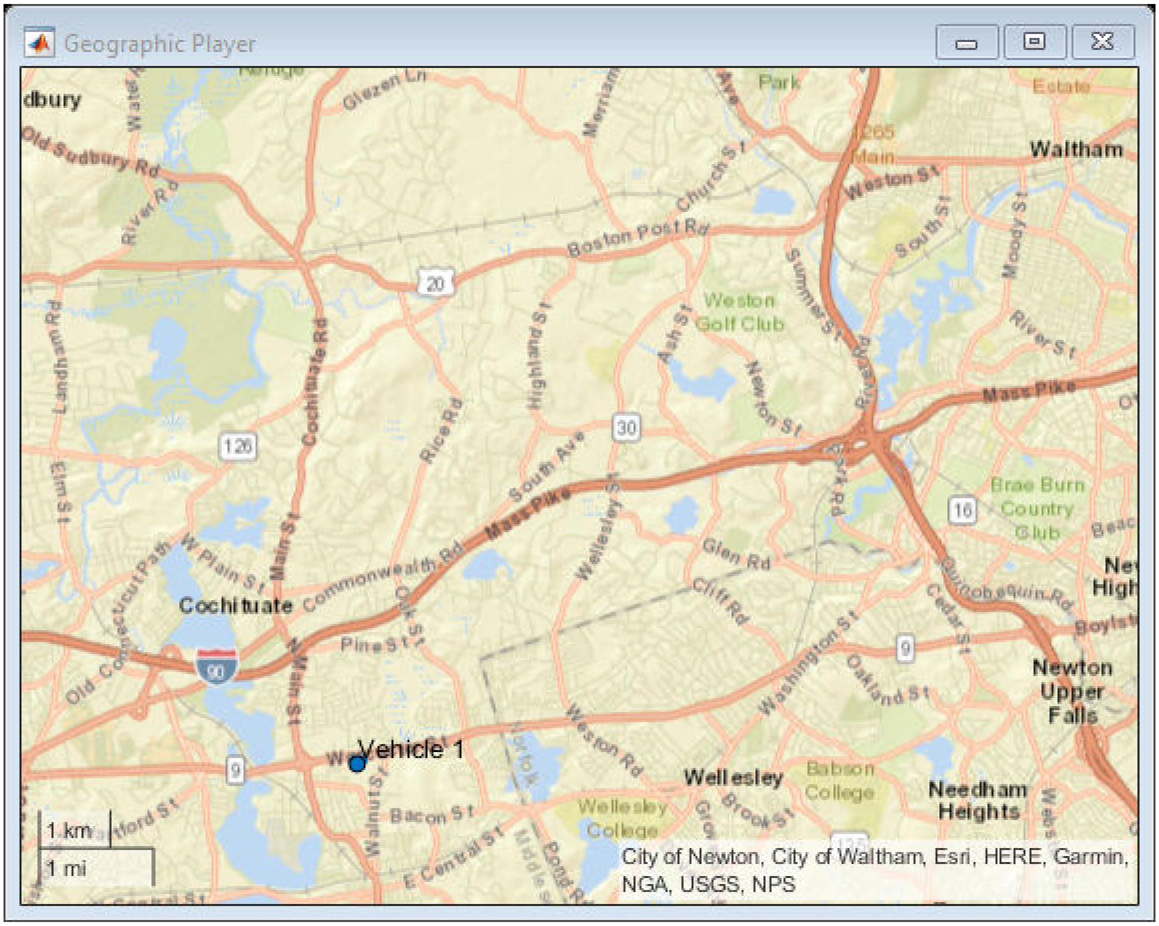
```
lat2 = flipud(lat1);
lon2 = flipud(lon1);
```

Create a `geoplayer` object. Initialize the player to display the last 10 positions as a line trailing the current position.

```
zoomLevel = 12;
player = geoplayer(lat1(1),lon1(1),zoomLevel,...
    'HistoryDepth',10,'HistoryStyle','line');
```

Plot the positions of both vehicles as they move over the route. Specify an ID for each track by using the `'TrackID'` name-value pair argument. By default, the `geoplayer` object centers the display of the vehicle on the first track. You can center the display on other tracks by using the `CenterOnID` property of the `geoplayer` object.

```
loopCounter = length(lat1);
for i = 1:loopCounter
    plotPosition(player,lat1(i),lon1(i),'TrackID',1,'Label','Vehicle 1');
    plotPosition(player,lat2(i),lon2(i),'TrackID',2,'Label','Vehicle 2');
end
```



Limitations

- Geographic map tiles are not available for all locations.

More About

Custom Basemaps

The `geoplayer` object can use custom basemaps from providers such as HERE Technologies and OpenStreetMap.

To make a custom basemap available for use with the `geoplayer` object, use the `addCustomBasemap` function. After you add a custom basemap, it remains available for use in future MATLAB sessions, until you remove the basemap by using the `removeCustomBasemap` function.

To display streaming coordinates on a custom basemap, specify the name of the basemap in the `Basemap` property of the `geoplayer` object.

Note For some custom basemaps, access to the map servers requires a valid license from the map provider.

Tips

- When the `geoplayer` object plots a position that is outside the current view of the map, the object automatically scrolls the map.

See Also

Functions
`addCustomBasemap` | `geoaxes` | `geobasemap` | `geobubble` | `geolimits` | `geoplot` | `latlon2local` | `local2latlon` | `removeCustomBasemap`

Properties
GeographicAxes Properties

Introduced in R2018a

plotPosition

Display current position in geoplayer figure

Syntax

```
plotPosition(player,lat,lon)
plotPosition(player,lat,lon,Name,Value)
```

Description

plotPosition(player,lat,lon) plots the point specified by latitude and longitude coordinates, (lat,lon), in the geoplayer figure, specified by player. To plot multiple routes simultaneously, specify a unique identifier for each route using the TrackID parameter.

plotPosition(player,lat,lon,Name,Value) uses Name,Value pair arguments to modify aspects of the plotted points.

For example, plotPosition(player,45,0,'Color','w','Marker','*') plots a point in the geoplayer figure as a white star.

Examples

View Position of Vehicle Along Route

Load a sequence of latitude and longitude coordinates.

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

Create a geographic player and set the zoom level to 12. Compared to the default zoom level, this zoom level zooms the map out by a factor of 8.

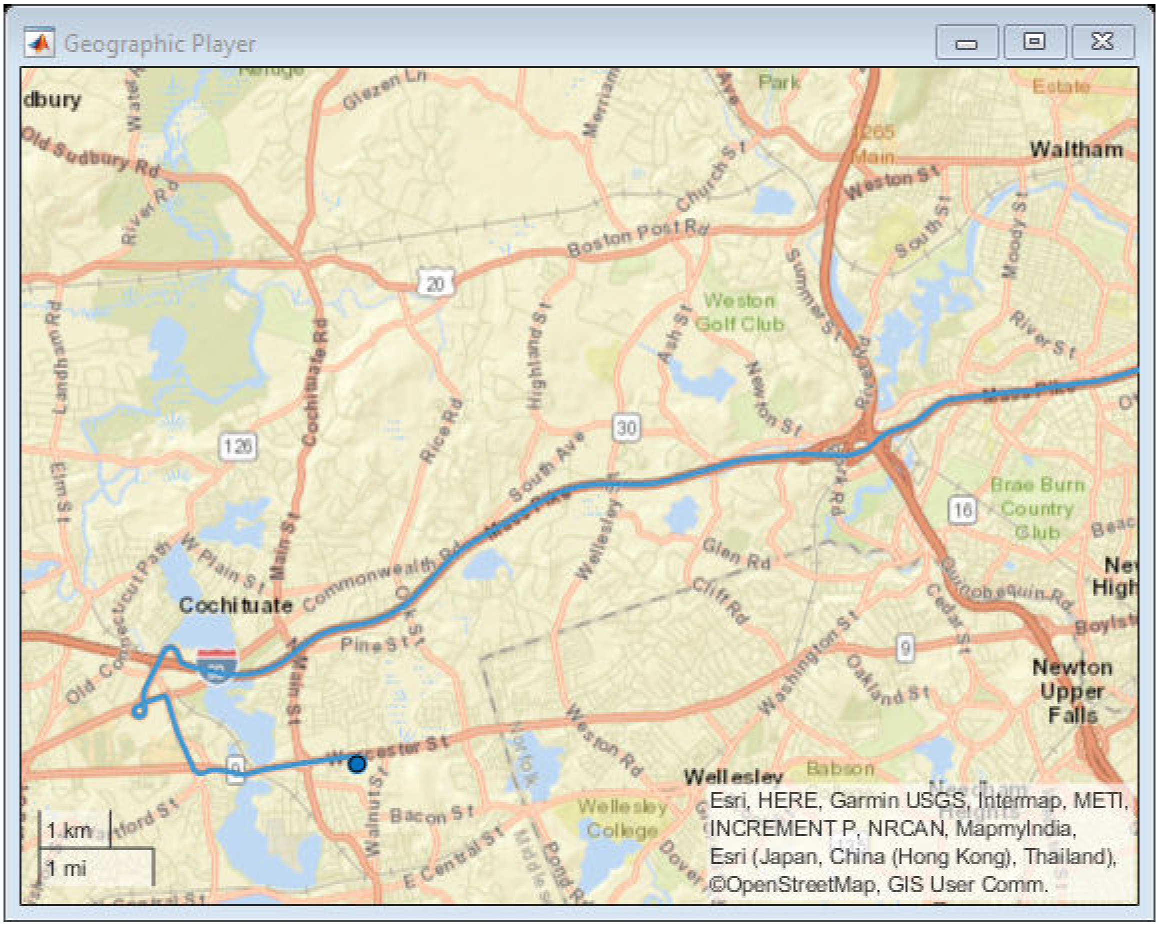
```
player = geoplayer(data.latitude(1),data.longitude(1),12);
```

Display the full route.

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

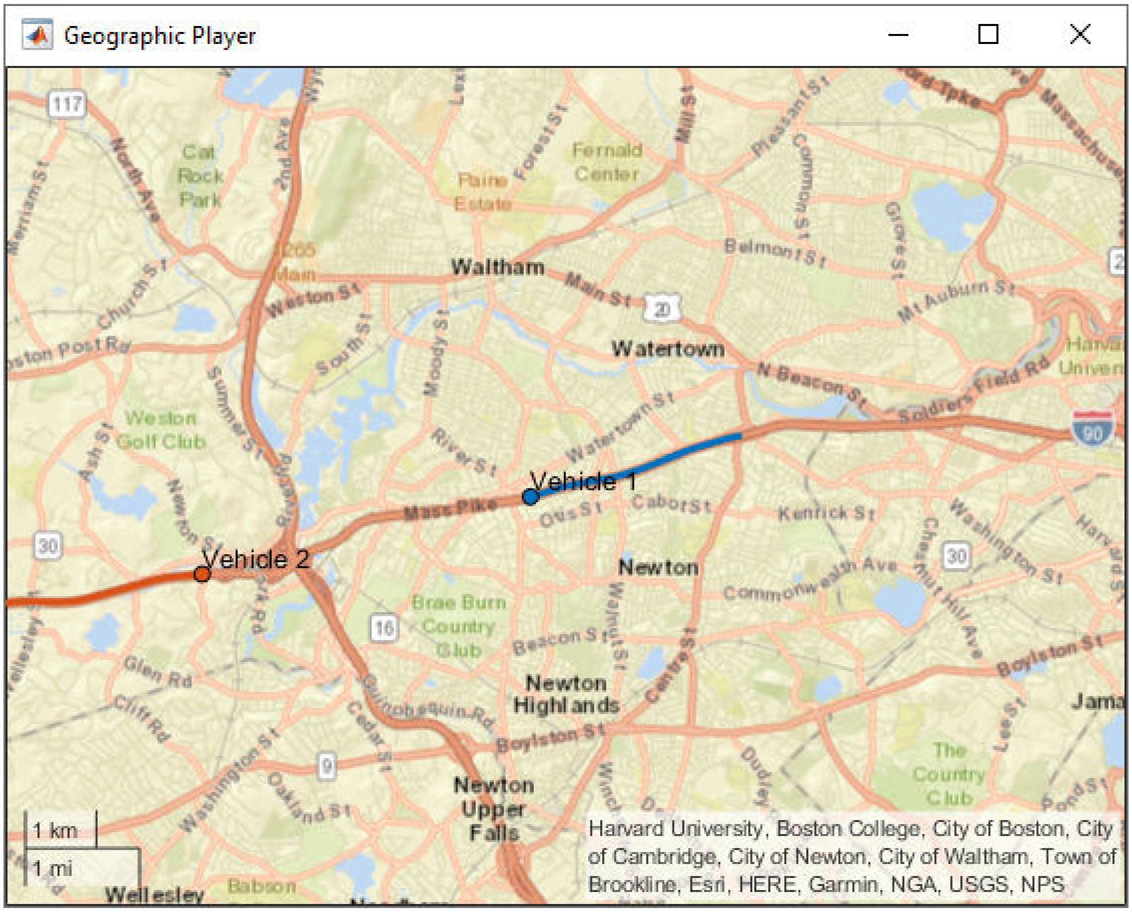
Display the coordinates in a sequence. The circle marker indicates the current position.

```
for i = 1:length(data.latitude)
    plotPosition(player,data.latitude(i),data.longitude(i));
    pause(0.05)
end
```



Plot the Tracks of Two Vehicles

Plot multiple routes simultaneously in a geographic player. First, assign each route a unique identifier. Then, when plotting points on the routes using the `plotPosition` object function, specify the route identifier using the `'TrackID'` name-value pair argument. In this example, the routes are labeled Vehicle 1 and Vehicle 2. This screen capture shows the point where the two routes are about to cross paths.



Load data for a route.

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

Extract data for the first vehicle.

```
lat1 = data.latitude;
lon1 = data.longitude;
```

Create a synthetic route for the second vehicle that drives the same route in the opposite direction.

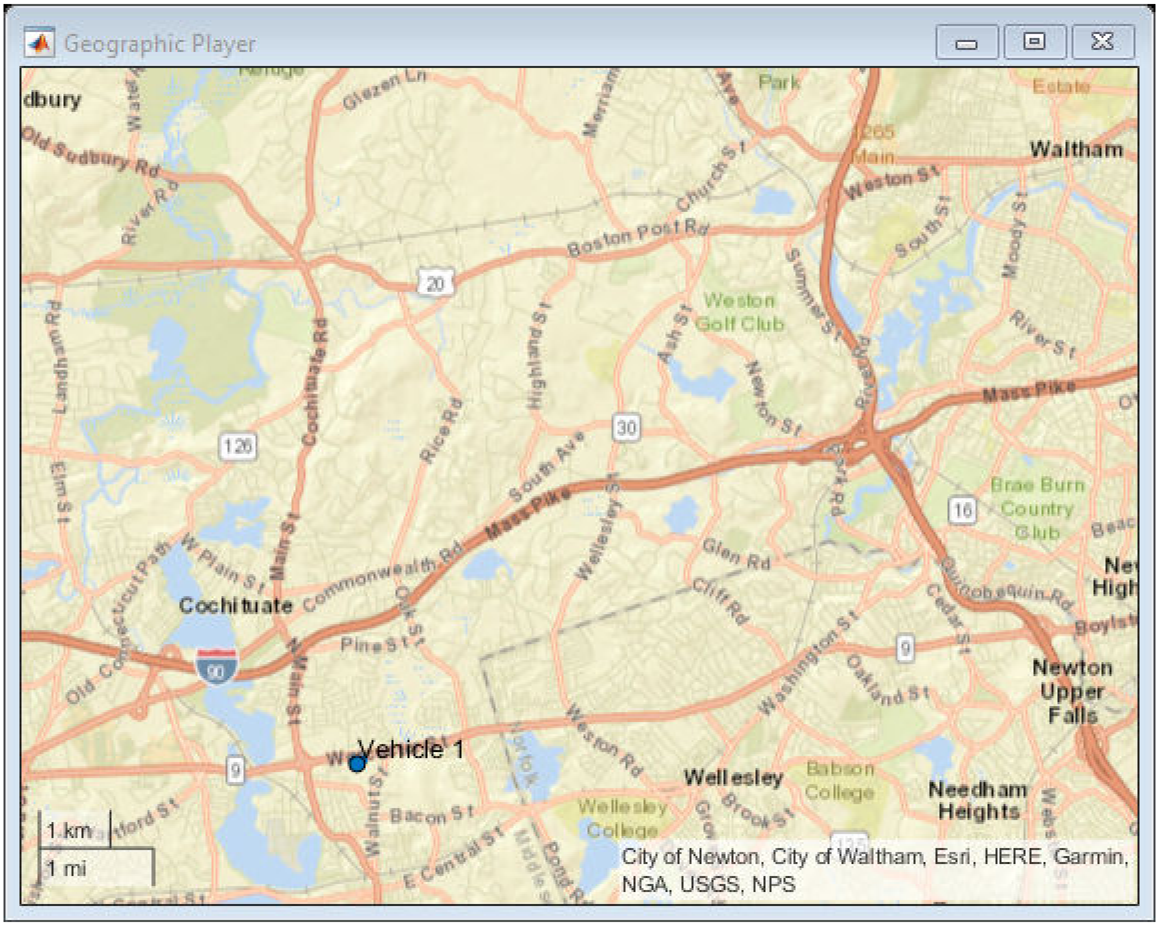
```
lat2 = flipud(lat1);
lon2 = flipud(lon1);
```

Create a `geoplayer` object. Initialize the player to display the last 10 positions as a line trailing the current position.

```
zoomLevel = 12;
player = geoplayer(lat1(1),lon1(1),zoomLevel,...
    'HistoryDepth',10,'HistoryStyle','line');
```

Plot the positions of both vehicles as they move over the route. Specify an ID for each track by using the `'TrackID'` name-value pair argument. By default, the `geoplayer` object centers the display of the vehicle on the first track. You can center the display on other tracks by using the `CenterOnID` property of the `geoplayer` object.

```
loopCounter = length(lat1);
for i = 1:loopCounter
    plotPosition(player,lat1(i),lon1(i),'TrackID',1,'Label','Vehicle 1');
    plotPosition(player,lat2(i),lon2(i),'TrackID',2,'Label','Vehicle 2');
end
```



Input Arguments

player — Streaming geographic player
`geoplayer` object

Streaming geographic player, specified as a `geoplayer` object.[5]

lat — Latitude coordinate
real scalar in the range [–90, 90]

Latitude coordinate of the point to display in the geographic player, specified as a real scalar in the range [–90, 90].

Data Types: `single` | `double`

5. Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

`lon` — Longitude coordinate

real scalar in the range [–180, 180]

Longitude coordinate of the point to display in the geographic player, specified as a real scalar in the range [–180, 180].

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Color','k'`

`Label` — Text description

`''` (default) | character vector | string scalar

Text description of the point, specified as the comma-separated pair consisting of `'Label'` and a character vector or string scalar.

Example: `'Label','07:45:00AM'`

`Color` — Marker color

color name | short color name | RGB triplet

Marker color, specified as the comma-separated pair consisting of `'Color'` and a color name, short color name, or RGB triplet. By default, the marker color is selected automatically.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the named color options and the equivalent RGB triplet values.

Color Name	Color Short Name	RGB Triplet	Appearance
`'red'`	`'r'`	`[1 0 0]`	
`'green'`	`'g'`	`[0 1 0]`	
`'blue'`	`'b'`	`[0 0 1]`	
`'cyan'`	`'c'`	`[0 1 1]`	
`'magenta'`	`'m'`	`[1 0 1]`	
`'yellow'`	`'y'`	`[1 1 0]`	
`'black'`	`'k'`	`[0 0 0]`	
`'white'`	`'w'`	`[1 1 1]`	

Example: `'Color',[1 0 1]`

Example: `'Color','m'`

Example: `'Color','magenta'`

`Marker` — Marker symbol

`'o'` (default) | `'+'` | `'*'` | `'.'` | `'x'` | ...

Marker symbol, specified as the comma-separated pair consisting of `'Marker'` and one of the markers in this table.

Marker	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'_'`	Horizontal line
`'|'`	Vertical line
`'s'`	Square
`'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'p'`	Pentagram
`'h'`	Hexagram

`MarkerSize` — Diameter of marker
6 (default) | positive real scalar

Approximate diameter of marker in points, specified as the comma-separated pair consisting of '`MarkerSize`' and a positive real scalar. 1 point = 1/72 inch. A marker size larger than 6 can reduce the rendering performance.

`TrackID` — Unique identifier for plotted track
1 (default) | positive integer

Unique identifier for plotted track, specified as a positive integer. Use this value to identify individual tracks when you plot multiple tracks. When you specify this value, all other name-value pair arguments for this function apply to only the track specified by this unique identifier.

Tips

- When a vehicle's track goes outside of viewable area, the map automatically re-centers based on the value of the `geoplayer` `CenterOnID` property.

See Also
`geoplayer` | `latlon2local` | `local2latlon` | `plotRoute` | `reset`

Introduced in R2018a

plotRoute

Display continuous route in `geoplayer` figure

Syntax

```
plotRoute(player,lat,lon)
plotRoute(player,lat,lon,Name,Value)
```

Description

`plotRoute(player,lat,lon)` displays a route, as defined by a series of latitude-longitude coordinates, in a `geoplayer` figure. The route appears as a continuous line on a map. To plot multiple routes in a `geoplayer`, call `plotRoute` for each route.

`plotRoute(player,lat,lon,Name,Value)` uses `Name,Value` pair arguments to modify the visual style of the route.

For example, `plotRoute(player,[45 46],[0 0],'Color','k')` plots a route in a `geoplayer` figure as a black line.

Examples

View Position of Vehicle Along Route

Load a sequence of latitude and longitude coordinates.

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

Create a geographic player and set the zoom level to 12. Compared to the default zoom level, this zoom level zooms the map out by a factor of 8.

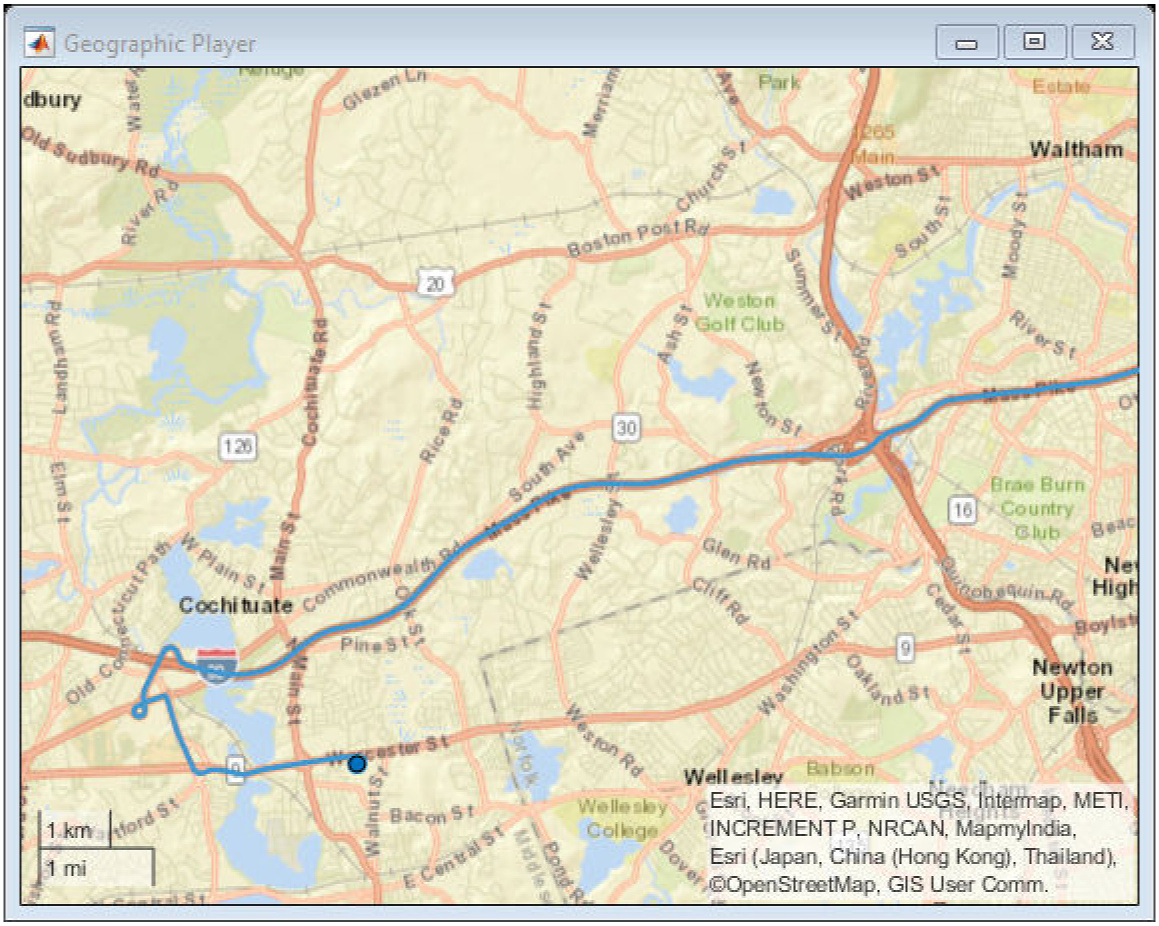
```
player = geoplayer(data.latitude(1),data.longitude(1),12);
```

Display the full route.

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

Display the coordinates in a sequence. The circle marker indicates the current position.

```
for i = 1:length(data.latitude)
    plotPosition(player,data.latitude(i),data.longitude(i));
    pause(0.05)
end
```



Plot Multiple Routes

Plot multiple routes in a geographic player by calling `plotRoute` multiple times.

Load data for a route.

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

Extract data for the first vehicle.

```
lat1 = data.latitude;
lon1 = data.longitude;
```

Create a synthetic route for the second vehicle. Add a small offset for better visibility.

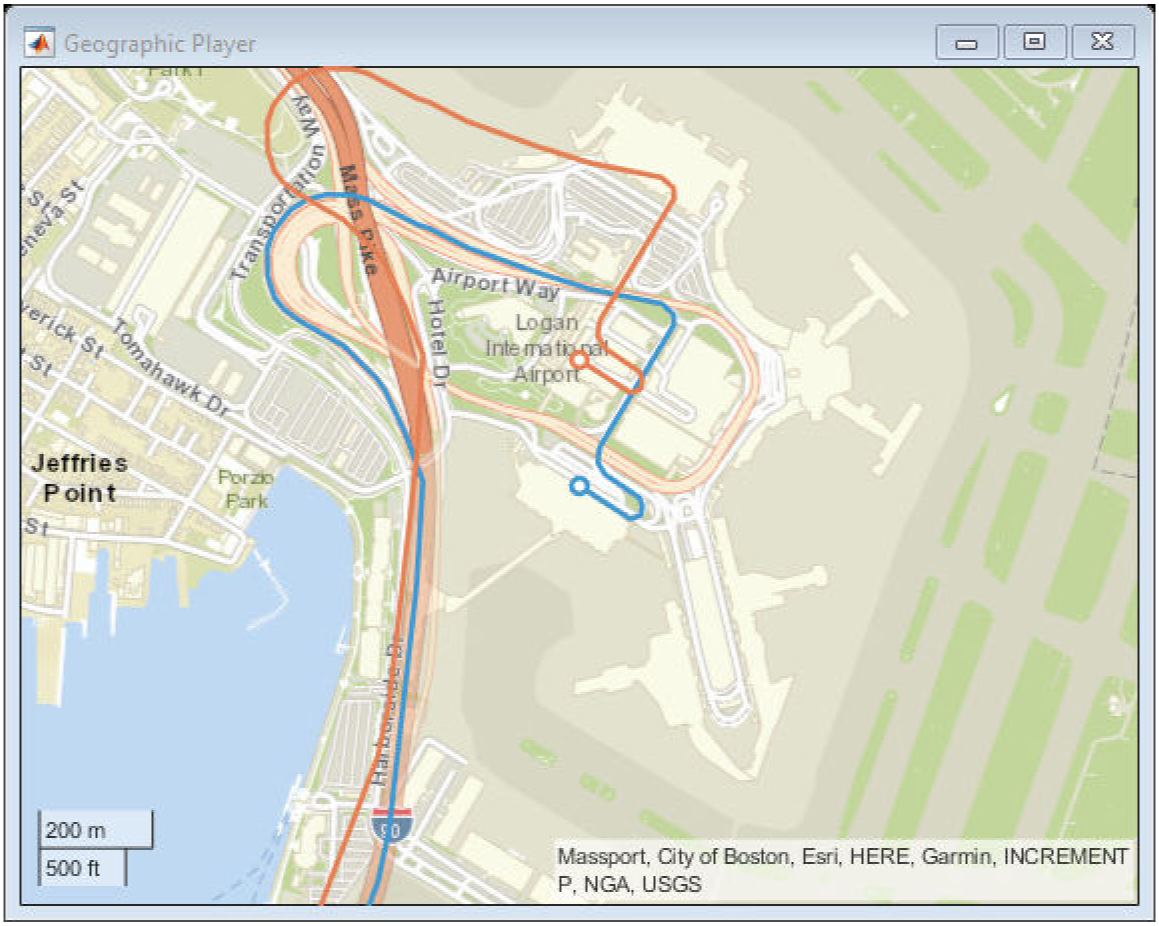
```
lat2 = lat1 + 0.002; % add a small offset in degrees
lon2 = lon1;
```

Create a `geoplayer` object, specifying the starting coordinates for one of the routes.

```
player = geoplayer(lat1(1),lon1(1));
```

Plot the routes in the geographic player by calling `plotRoute` for each route.

```
plotRoute(player,lat1,lon1);
plotRoute(player,lat2,lon2);
```



Input Arguments

`player` — Streaming geographic player
`geoplayer` object

Streaming geographic player, specified as a `geoplayer` object.[6]

`lat` — Latitude coordinates
real-valued vector

Latitude coordinates of points along the route, specified as a real-valued vector with elements in the range [–90, 90].

Data Types: `single` | `double`

`lon` — Longitude coordinates
real-valued vector

6. Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Longitude coordinates of points along the route, specified as a real-valued vector with elements in the range [–180, 180].

Data Types: single | double

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Color','g'

Color — Line color

color name | short color name | RGB triplet

Line color, specified as the comma-separated pair consisting of 'Color' and a color name, short color name, or RGB triplet. By default, the line color is selected automatically.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7]. Alternatively, you can specify some common colors by name. This table lists the named color options and the equivalent RGB triplet values.

Color Name	Color Short Name	RGB Triplet	Appearance
'red'	'r'	[1 0 0]	
'green'	'g'	[0 1 0]	
'blue'	'b'	[0 0 1]	
'cyan'	'c'	[0 1 1]	
'magenta'	'm'	[1 0 1]	
'yellow'	'y'	[1 1 0]	
'black'	'k'	[0 0 0]	
'white'	'w'	[1 1 1]	

Example: 'Color',[1 0 1]

Example: 'Color','m'

Example: 'Color','magenta'

LineWidth — Line width

2 (default) | positive number

Line width in points, specified as the comma-separated pair consisting of 'LineWidth' and a positive number. 1 point = 1/72 inch.

ShowEndpoints — Display origin and destination

'on' (default) | 'off'

Display the origin and destination points, specified as the comma-separated pair consisting of 'ShowEndpoints' and 'on' or 'off'. Specify 'on' to display the origin and destination points. The origin marker is white and the destination marker is filled with color.

See Also

`geoplayer` | `latlon2local` | `local2latlon` | `plotPosition` | `reset`

Introduced in R2018a

reset

Remove all existing plots from `geoplayer` figure

Syntax

`reset(player)`

Description

`reset(player)` removes all previously plotted points and routes from the `geoplayer` figure.

Examples

Reset Geographic Player

Load a sequence of latitude and longitude coordinates.

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

Create a geographic player with a zoom level of 12. Configure the geographic player to display all points in its history.

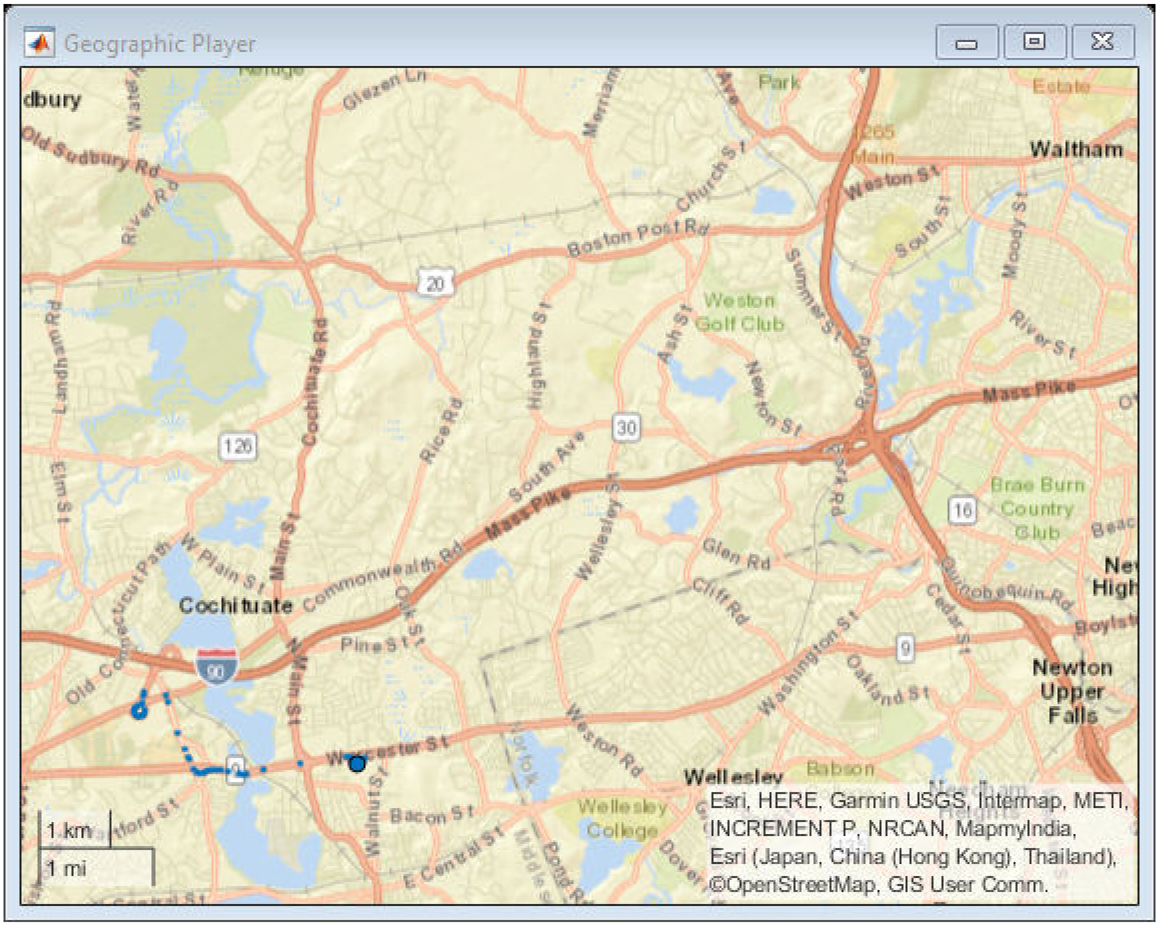
```
player = geoplayer(data.latitude(1),data.longitude(1),12,'HistoryDepth',Inf);
```

Display the full route.

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

Display the coordinates in a sequence. The circle marker indicates the current position. At the 200th point, reset the geographic player. Observe that the route and all previously plotted points are removed.

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

    if i == 200
        reset(player)
    end

    pause(.05)
end
```



Input Arguments

`player` — Streaming geographic player
`geoplayer` object

Streaming geographic player, specified as a `geoplayer` object.[7]

See Also

`geoplayer` | `plotPosition` | `plotRoute`

Introduced in R2018a

7. Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

show

Make geoplayer figure visible

Syntax

```
show(player)
```

Description

show(player) makes the geoplayer figure visible again after closing or hiding it.

Examples

Hide and Show Geographic Player

Load a sequence of latitude and longitude coordinates.

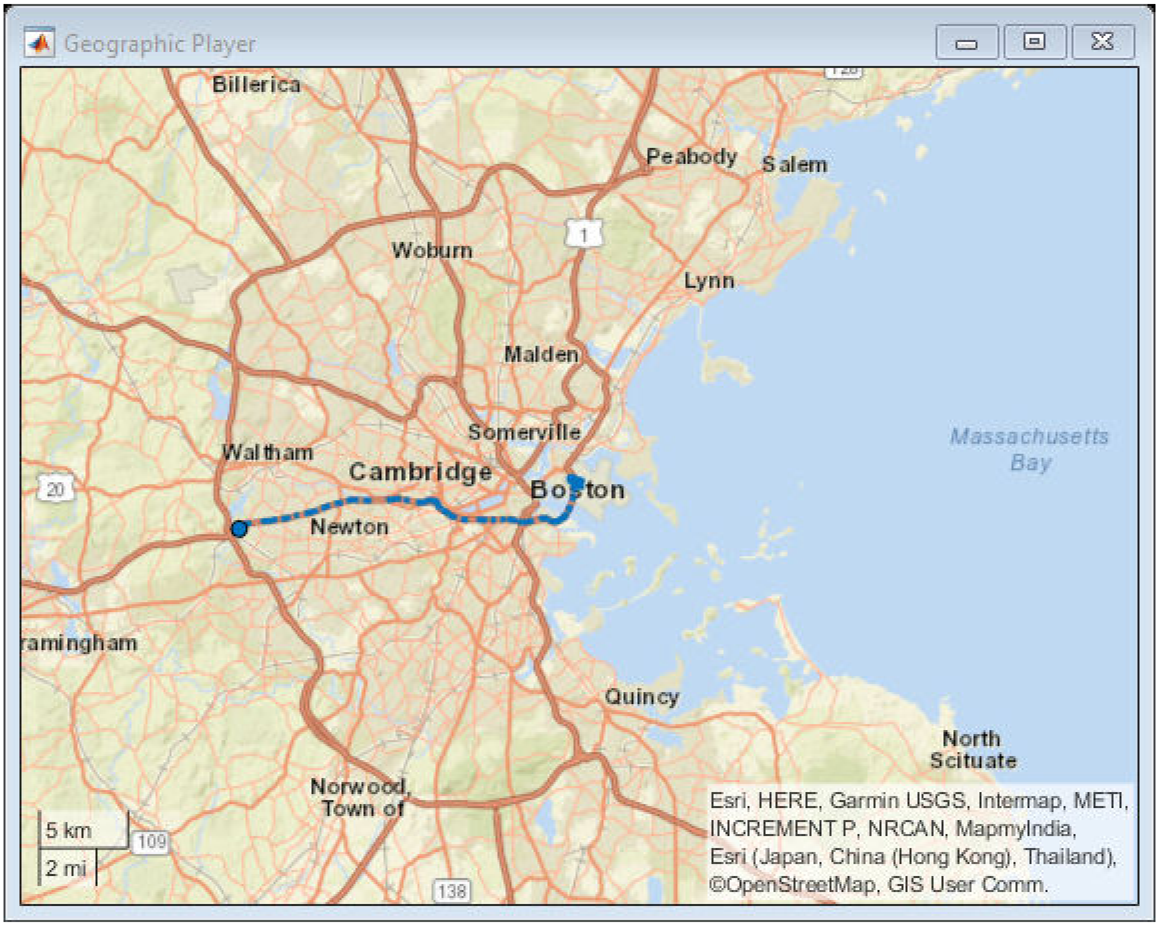
```
data = load('geoRoute.mat');
```

Create a geographic player with a zoom level of 10. Configure the player to show its complete history of plotted points.

```
player = geoplayer(data.latitude(1),data.longitude(1),10,'HistoryDepth',Inf);
```

Display the first half of the geographic coordinates in a sequence. The circle marker indicates the current position.

```
halfLength = round(length(data.latitude)/2);

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



Hide the player and confirm that it is no longer visible.

```
hide(player)
isOpen(player)
```

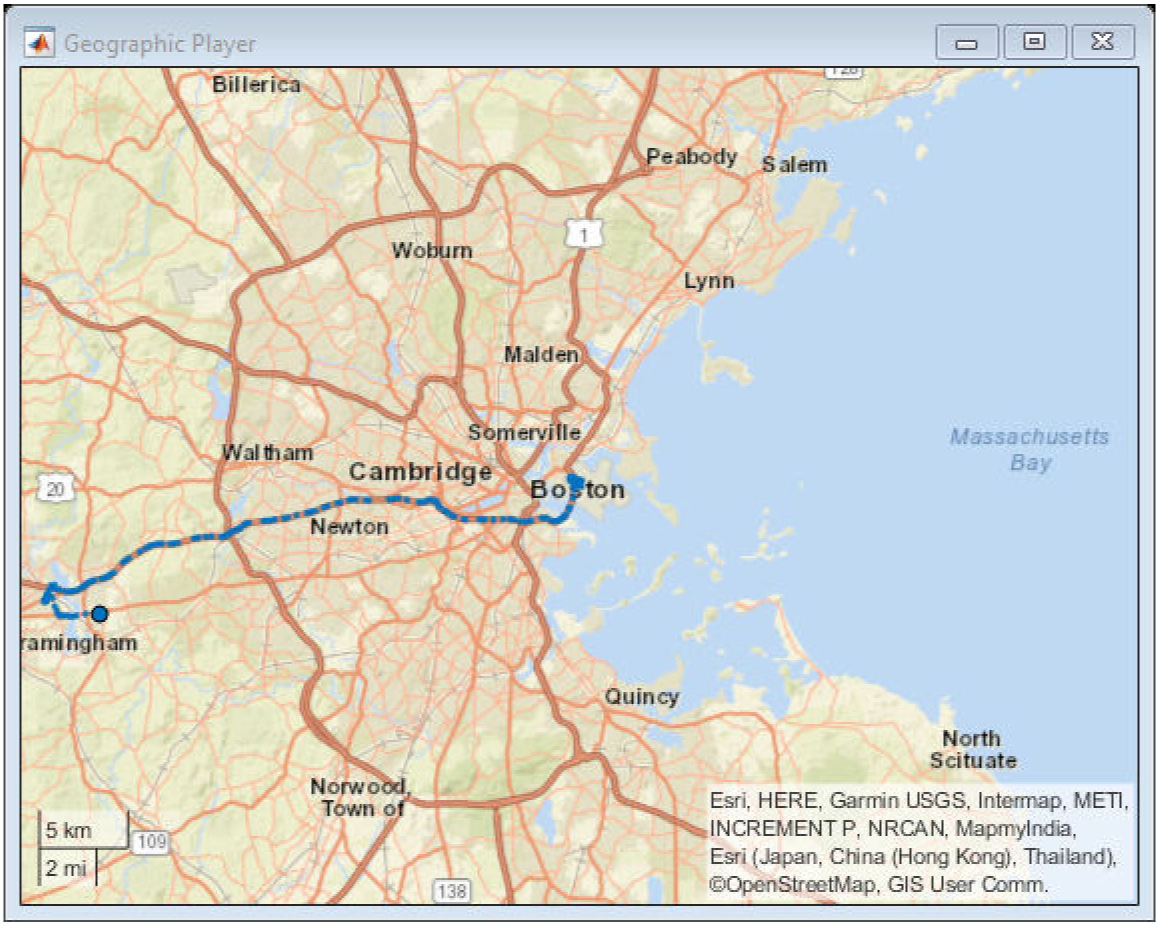
```
ans = logical
   0
```

Add the remaining half of the geographic coordinates to the map.

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

Show the player. The player now displays both halves of the route.

```
show(player)
```



Input Arguments

player — Streaming geographic player
geoplayer object

Streaming geographic player, specified as a geoplayer object.[8]

See Also

geoplayer | hide | isOpen

Introduced in R2018a

8. Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

hide

Make `geoplayer` figure invisible

Syntax

`hide(player)`

Description

`hide(player)` hides the `geoplayer` figure. To redisplay this figure, use `show(player)`.

Examples

Hide and Show Geographic Player

Load a sequence of latitude and longitude coordinates.

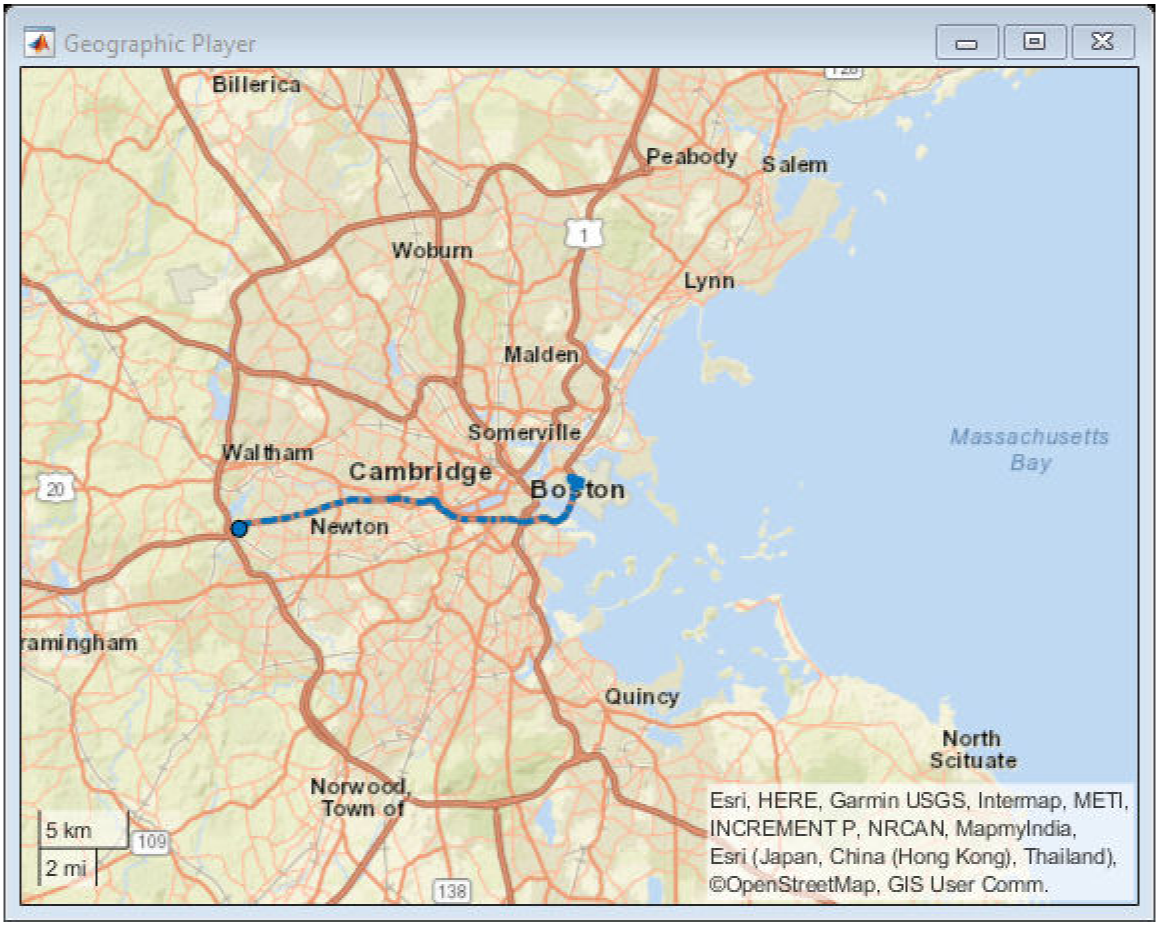
```
data = load('geoRoute.mat');
```

Create a geographic player with a zoom level of 10. Configure the player to show its complete history of plotted points.

```
player = geoplayer(data.latitude(1),data.longitude(1),10,'HistoryDepth',Inf);
```

Display the first half of the geographic coordinates in a sequence. The circle marker indicates the current position.

```
halfLength = round(length(data.latitude)/2);

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



Hide the player and confirm that it is no longer visible.

```
hide(player)
isOpen(player)
```

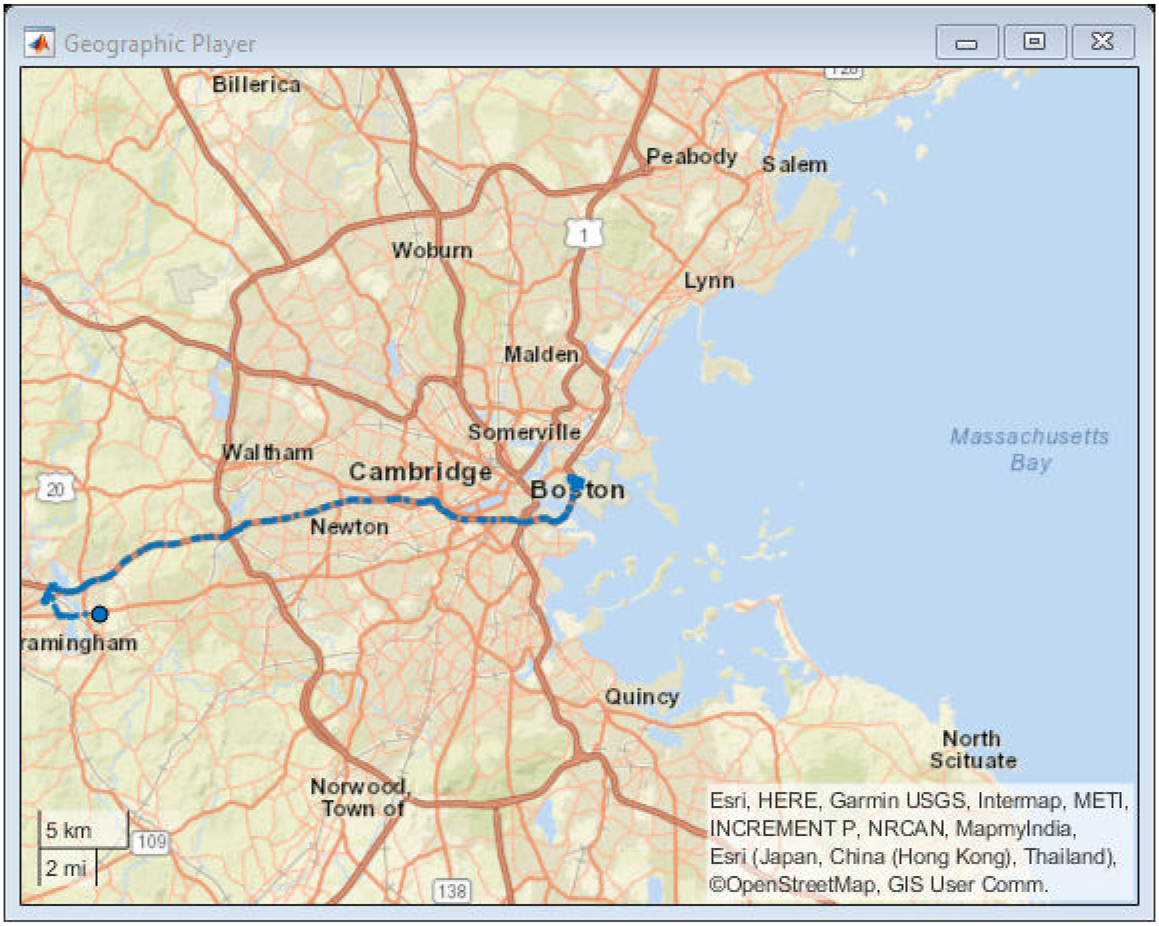
```
ans = logical
   0
```

Add the remaining half of the geographic coordinates to the map.

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

Show the player. The player now displays both halves of the route.

```
show(player)
```



Input Arguments

`player` — Streaming geographic player
`geoplayer` object

Streaming geographic player, specified as a `geoplayer` object.[9]

See Also

`geoplayer` | `isOpen` | `show`

Introduced in R2018a

9. Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

isOpen

Return true if `geoplayer` figure is visible

Syntax

```
tf = isOpen(player)
```

Description

`tf = isOpen(player)` returns logical `1` (true) if the `geoplayer` figure is visible. Otherwise, `isOpen` returns logical `0` (false).

Examples

Plot Points While Geographic Player Is Open

Load a sequence of latitude and longitude coordinates.

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

Create a geographic player with a zoom level of 12. Configure the player to display all points in its history.

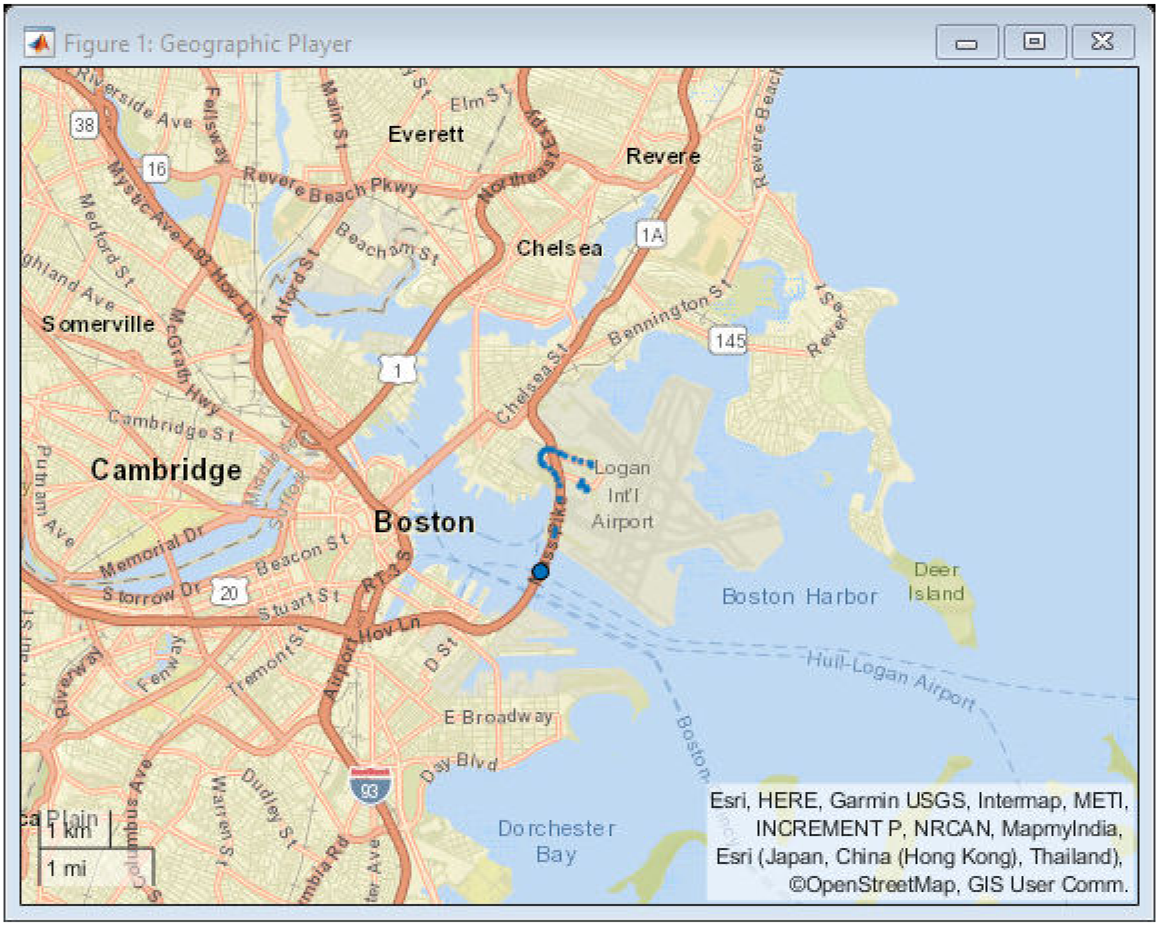
```
player = geoplayer(data.latitude(1),data.longitude(1),12,'HistoryDepth',Inf);
```

Display the geographic coordinates in a sequence by using the `plotPosition` function. Put the call to `plotPosition` inside a `while` loop, so that the player plots points only while the figure is open. You can exit the loop by closing the figure. If you do not close the figure, then the loop automatically exits when all points are plotted.

```
i = 1;
numPoints = length(data.latitude);
while isOpen(player) && i<=numPoints
    plotPosition(player,data.latitude(i),data.longitude(i))
    pause(0.1)
    i=i+1;
end
```

To make the figure visible again, use the `show` function.

```
show(player)
```



Input Arguments

player — Streaming geographic player
geoplayer object

Streaming geographic player, specified as a `geoplayer` object.[10]

Output Arguments

tf — Visibility of geographic player
1 (true) | 0 (false)

Visibility of geographic player, returned as logical `1` (true) when the `geoplayer` figure is open, and logical `0` (false) otherwise.

See Also

`geoplayer` | `hide` | `show`

10. Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Introduced in R2018a

hereHDLMReader

HERE HD Live Map reader

Description

Use a `hereHDLMReader` object to read high-definition map data for selected map tiles from the HERE HD Live Map[11] (HERE HDLM) web service, provided by HERE Technologies. HERE HDLM data provides highly detailed and accurate information about the vehicle environment, such as road and lane topology, and is suitable for developing automated driving applications.

You can select specific map tiles from which to read data or select map tiles based on the coordinates of a driving route. To read map data for tiles, use the `read` function and specify the reader as an input argument. For more details, see "Read and Visualize HERE HD Live Map Data".

Note Use of the `hereHDLMReader` object requires valid HERE HDLM credentials. If you have not previously set up 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**.

Creation

Syntax

```
reader = hereHDLMReader(lat,lon)
reader = hereHDLMReader(tileID)
reader = hereHDLMReader( ___ ,Name,Value)
```

Description

`reader = hereHDLMReader(lat,lon)` creates a HERE HDLM reader that can read map data for the HERE map tiles that correspond to a set of latitude and longitude coordinates. The map tiles are at a zoom level of 14.

`reader = hereHDLMReader(tileID)` creates a HERE HDLM reader that can read map data for the map tiles with the specified HERE tile IDs. These tile IDs are stored in the `TileIDs` property of the HERE HDLM reader.

`reader = hereHDLMReader( ___ ,Name,Value)` sets the `Configuration`, `WriteLocation`, and `CoordinateFormat` properties using one or more name-value pairs. For example, `hereHDLMReader(tileID,'Configuration',config)` creates a reader that is configured to read map tile data from a specific HERE HDLM production catalog or catalog version, where `config` is a `hereHDLMConfiguration` object.

11. 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.

Input Arguments

lat — Latitude coordinates
vector of real values in the range [–90, 90]

Latitude coordinates, specified as a vector of real values in the range [–90, 90].

Use this vector, along with `lon`, to specify the coordinates of a driving route that you want to read map data from.

`lat` and `lon` must be the same size.

Data Types: `double`

lon — Longitude coordinates
vector of real values in the range [–180, 180]

Longitude coordinates, specified as a vector of real values in the range [–180, 180].

Use this vector, along with `lat`, to specify the coordinates of a driving route that you want to read map data from.

`lat` and `lon` must be the same size.

Data Types: `double`

tileID — HERE tile IDs
vector of unsigned 32-bit integers

HERE tile IDs from which to read data, specified as a vector of unsigned 32-bit integers. These tile IDs are stored in the `TileIDs` property of the `hereHDLMReader` object. The specified map tiles must all come from the same HERE HDLM production catalog.

If you configure the `hereHDLMReader` object to read data from a specific catalog using the `hereHDLMConfiguration` object, then all tile IDs must be found within that catalog. Otherwise, the reader object returns an error.

Example: `uint32([386497368 386497369])`

Data Types: `uint32`

Properties

TileIDs — HERE tile IDs
vector of unsigned 32-bit integers

This property is read-only.

HERE tile IDs from which to read data, specified as a vector of unsigned 32-bit integers. These tiles correspond to either the specified `lat` and `lon` coordinates or the specified `tileID` tiles.

Example: `uint32([386497368 386497369])`

Data Types: `uint32`

Layers — Map data layers
string array

This property is read-only.

Map data layers available for the selected HERE tile IDs, specified as a string array of layer names. The available map layers vary depending on the geographic region.

To read data from these layers, specify these layer names as inputs to the `read` function.

`Configuration` — Catalog configuration
`hereHDLMConfiguration` object

This property is read-only.

Catalog configuration, specified as a `hereHDLMConfiguration` object. This configuration contains the specific HERE HDLM catalog and catalog version that the `hereHDLMReader` object reads data from.

If you do not specify a configuration at creation, the reader object computes the default configuration by searching the latest version of each production catalog. If all selected map tile IDs are found within a catalog, then the `hereHDLMReader` object is configured to read data from the latest version of that catalog.

You can set this property when you create the reader object. After you create the object, this property is read-only.

`WriteLocation` — Folder name of downloaded map data
`tempdir` (temporary directory) (default) | string scalar | character vector

This property is read-only.

Name of folder to which HERE HDLM data is downloaded, specified as a string scalar or character vector. The specified folder must exist and have write permissions.

By default, data from the HERE HDLM web service is downloaded to a temporary file location. This temporary file location is deleted at the end of your MATLAB session.

You can set this property when you create the reader object. After you create the object, this property is read-only.

Example: `"C:\Users\myName\HERE"`

`CoordinateFormat` — Type of coordinate encoding format
`'geographic'` (default) | `'raw'`

Type of coordinate encoding format to apply to geographic coordinate values, specified as either `'geographic'` or `'raw'`.

Format	Description	Example
`'geographic'`	Coordinate values are returned as (latitude, longitude) pairs with decimal degrees.	`[42.3743 -71.0266]`
`'raw'`	Coordinate values are returned in the default coordinate encoding format of the HERE HDLM service.	`int64(5978422612852408322)`

Object Functions

read   Read HERE HD Live Map layer data
plot   Plot HERE HD Live Map layer data

Examples

Plot and Stream Lane Topology Data from Driving Route

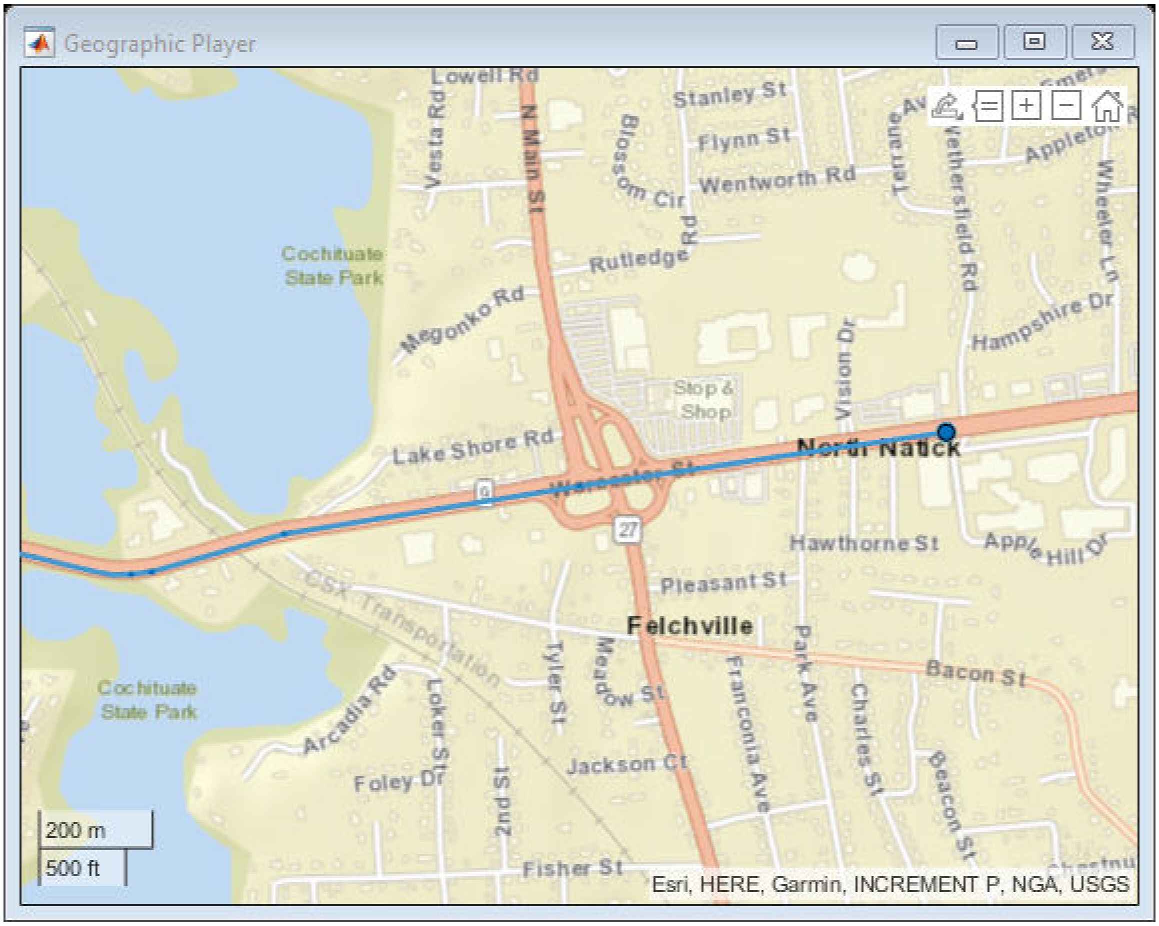
Use the HERE HD Live Map (HERE HDLM) service to read the lane topology data of a driving route and its surrounding area. Plot this data, and then stream the route on a geographic player.

Load the latitude and longitude coordinates of a driving route in Natick, Massachusetts, USA.

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

Stream the coordinates on a geographic player.

```
player = geoplayer(lat(1),lon(1),'HistoryDepth',5);
plotRoute(player,lat,lon)

for idx = 1:length(lat)
    plotPosition(player,lat(idx),lon(idx))
end
```

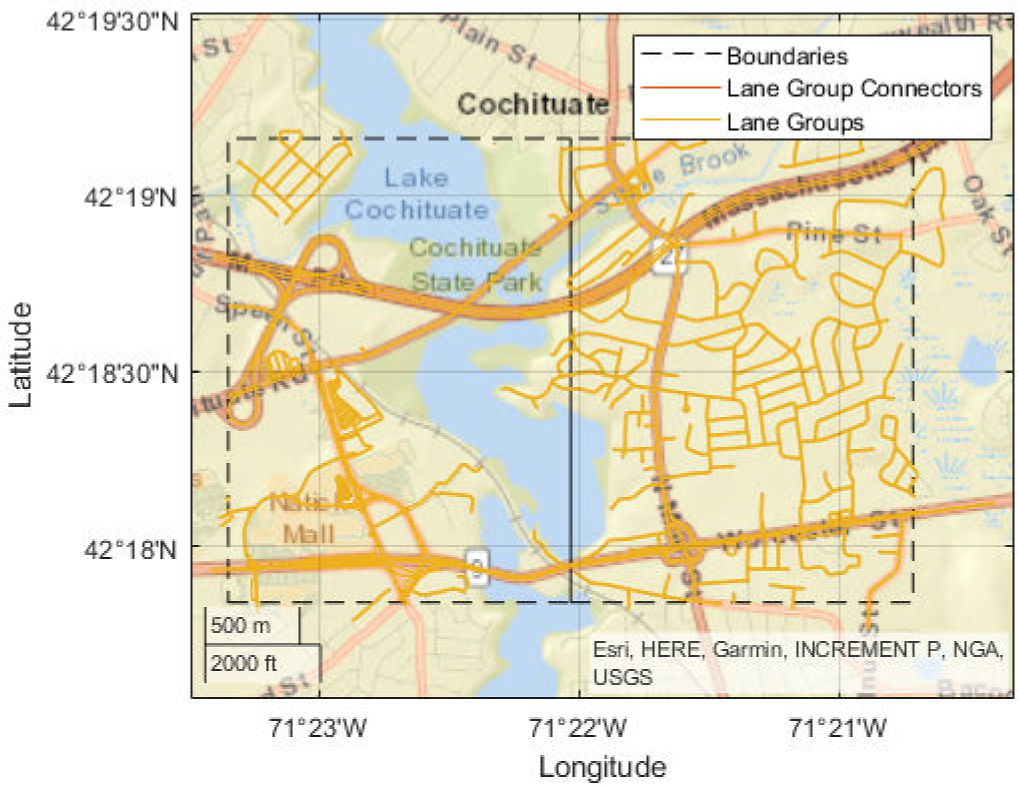


Create a HERE HDLM reader from the route coordinates. If you have not previously set up HERE HDLM credentials, a dialog box prompts you to enter them. The reader contains map data for the two map tiles that the route crosses.

```
reader = hereHDLMReader(lat,lon);
```

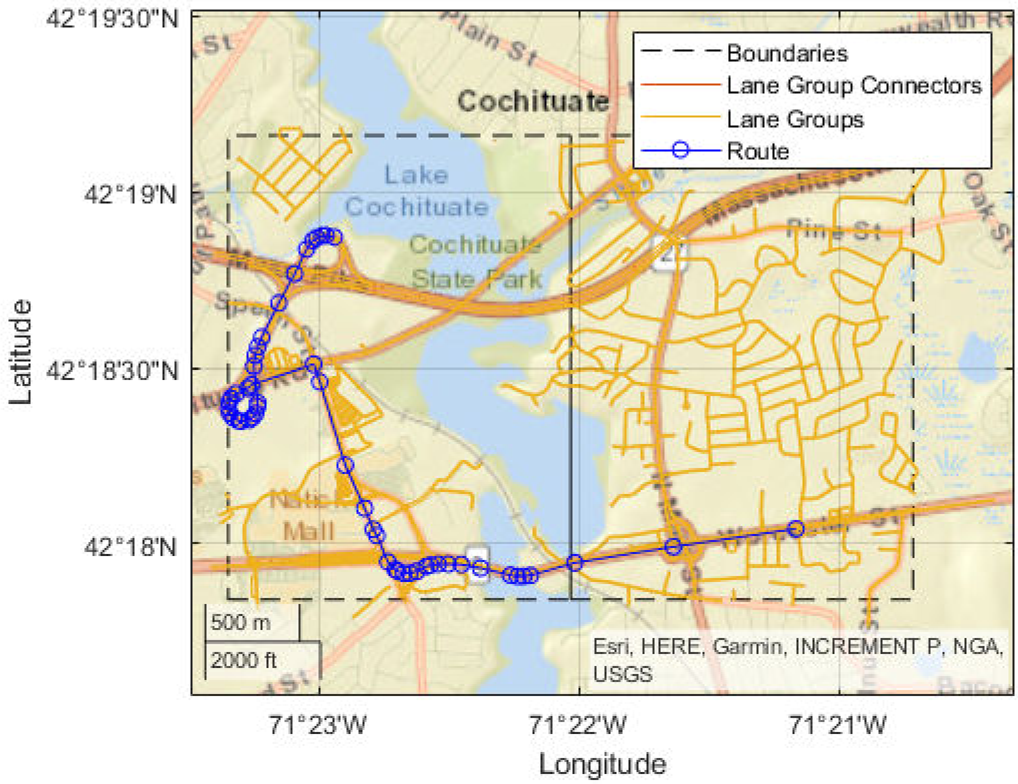
Read lane topology data from the `LaneTopology` layer of the map tiles. Plot the lane topology.

```
laneTopology = read(reader,'LaneTopology');
plot(laneTopology)
```



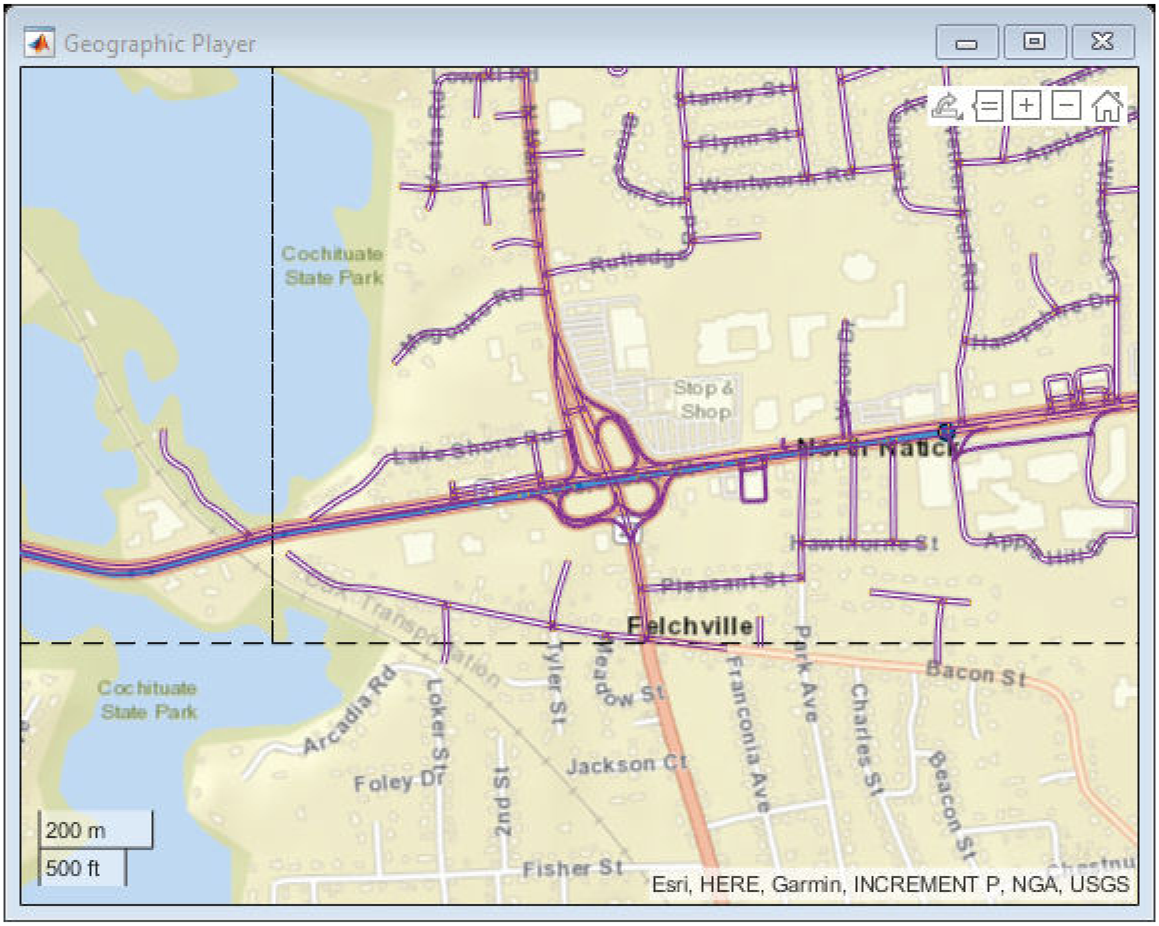
Overlay the route data on the plot.

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



Overlay the lane topology data on the geographic player. Stream the route again.

```
plot(laneTopology,'Axes',player.Axes)
for idx = 1:length(lat)
    plotPosition(player,lat(idx),lon(idx))
end
```



Plot 3-D Lane Geometry on Custom Basemap

Use the HERE HD Live Map (HERE HDLM) web service to read 3-D lane geometry data from a map tile. Then, plot the data on an OpenStreetMap® basemap.

Create a HERE HDLM reader for a map tile ID representing an area of Berlin, Germany. If you have not previously set up HERE HDLM credentials, a dialog box prompts you to enter them.

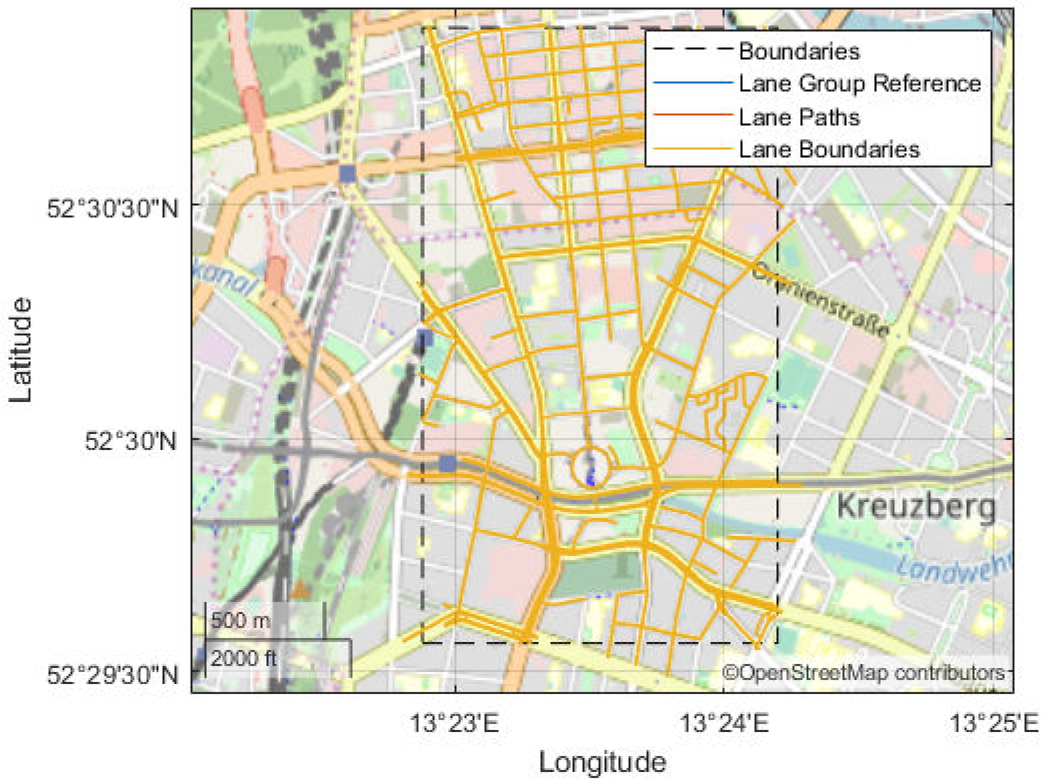
```
tileID = uint32(377894435);
reader = hereHDLMReader(tileID);
```

Add the OpenStreetMap basemap to the list of basemaps available for use with the HERE HDLM service. 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)
```

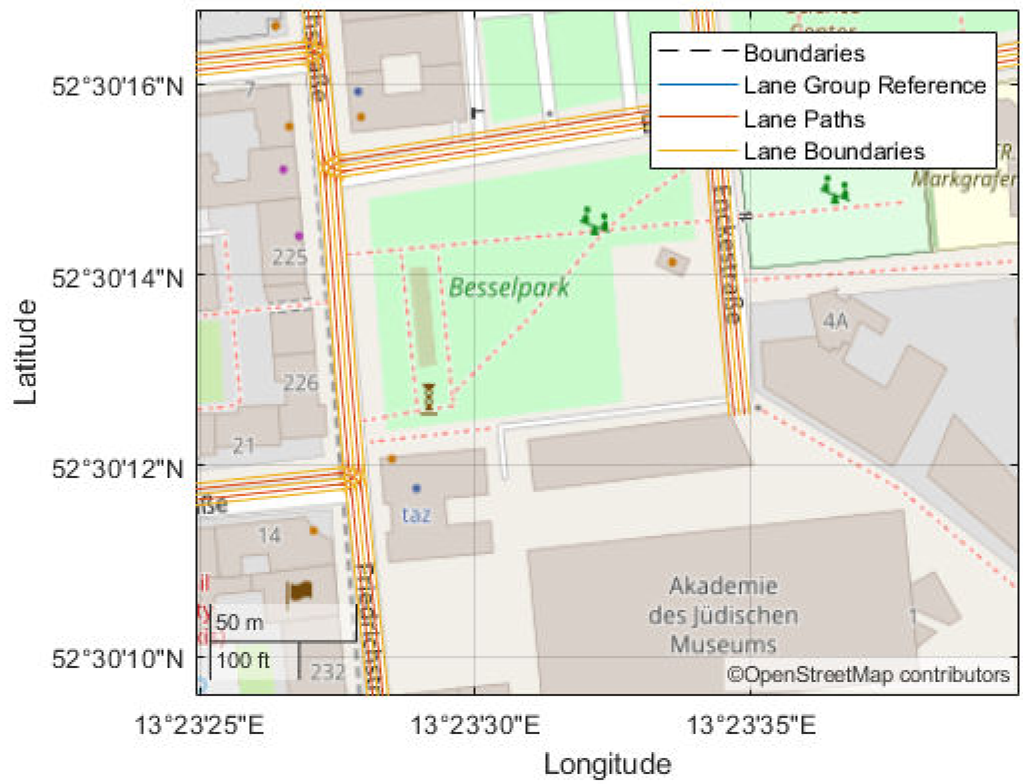
Read 3-D lane geometry data from the `LaneGeometryPolyline` layer of the map tile. Plot the lane geometry on the `openstreetmap` basemap.

```
laneGeometryPolyline = read(reader,'LaneGeometryPolyline');
gx = plot(laneGeometryPolyline);
geobasemap(gx,'openstreetmap')
```



Zoom in on the central coordinate of the map tile.

```
latcenter = laneGeometryPolyline.TileCenterHere3dCoordinate.Here2dCoordinate(1);
loncenter = laneGeometryPolyline.TileCenterHere3dCoordinate.Here2dCoordinate(2);

offset = 0.001;
latlim = [latcenter-offset,latcenter+offset];
lonlim = [loncenter-offset,loncenter+offset];

geolimits(latlim,lonlim)
```



Find Shortest Path Between Two Nodes

Use the HERE HD Live Map (HERE HDLM) web service to read the topology geometry data from a map tile. Use this data to find the shortest path between two nodes within the map tile.

Define a HERE tile ID for an area of Stockholm, Sweden.

```
tileID = uint32(378373553);
```

Create a HERE HDLM reader for the tile ID. Configure the reader to search for the tile in only the Western Europe catalog. If you have not previously set up HERE HDLM credentials, a dialog box prompts you to enter them. The reader contains map data for the specified map tile.

```
config = hereHDLMConfiguration('hrn:here:data::olp-here-had:here-hdlm-protobuf-weu-2');
reader = hereHDLMReader(tileID,'Configuration',config);
```

Read the link definitions from the `TopologyGeometry` layer of the map tile. The returned layer object contains the specified `LinksStartingInTile` field and the required map tile fields, such as the tile ID. The other fields are empty. Your map data and catalog version might differ from the ones shown here.

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

```
topology = 
  TopologyGeometry with properties:
```

```
  Data:
                 HereTileId: 378373553
        IntersectingLinkRefs: []
         LinksStartingInTile: [1249×1 struct]
                 NodesInTile: []
   TileCenterHere2dCoordinate: [59.3372 18.0505]

  Metadata:
                    Catalog: 'hrn:here:data::olp-here-had:here-hdlm-protobuf-weu-2'
             CatalogVersion: 5597

 Use plot to visualize TopologyGeometry data.
```

Find the start and end nodes for each link in the `LinksStartingInTile` field.

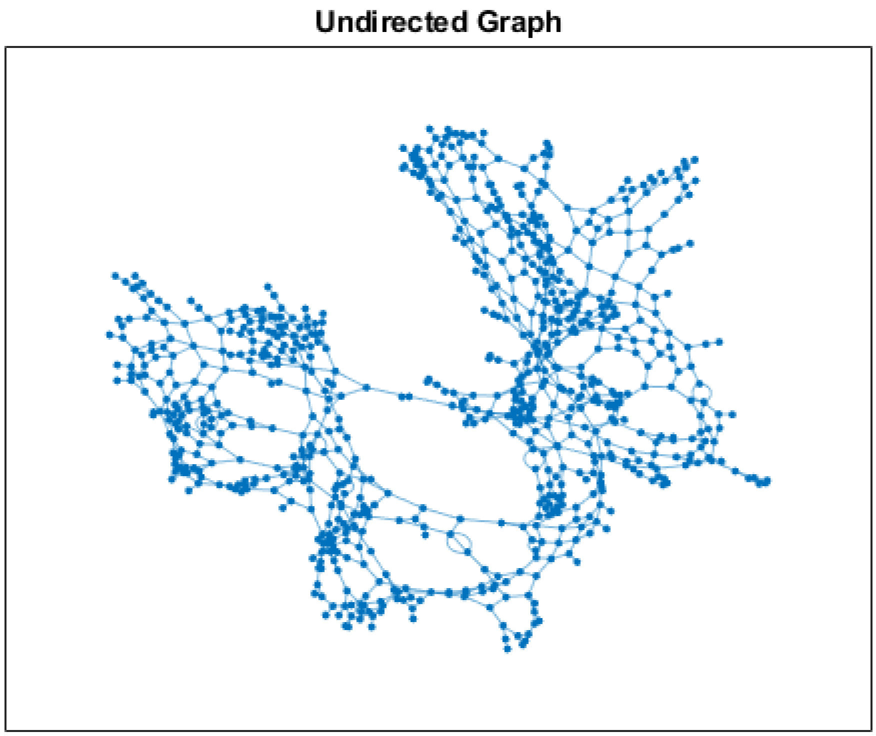
```
startNodes = [topology.LinksStartingInTile.StartNodeId];
endNodesRef = [topology.LinksStartingInTile.EndNodeRef];
endNodes = [endNodesRef.NodeId];
```

Find the length of each link in meters.

```
linkLengths = [topology.LinksStartingInTile.LinkLengthMeters];
```

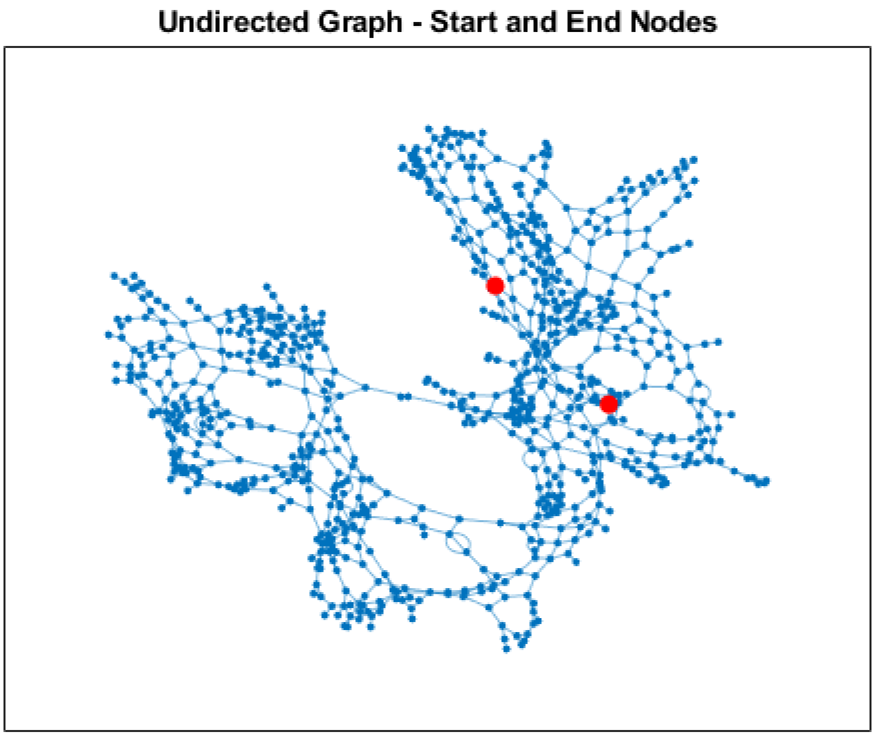
Create an undirected graph for the links in the map tile.

```
G = graph(string(startNodes),string(endNodes),double(linkLengths));
H = plot(G,'Layout','force');
title('Undirected Graph')
```



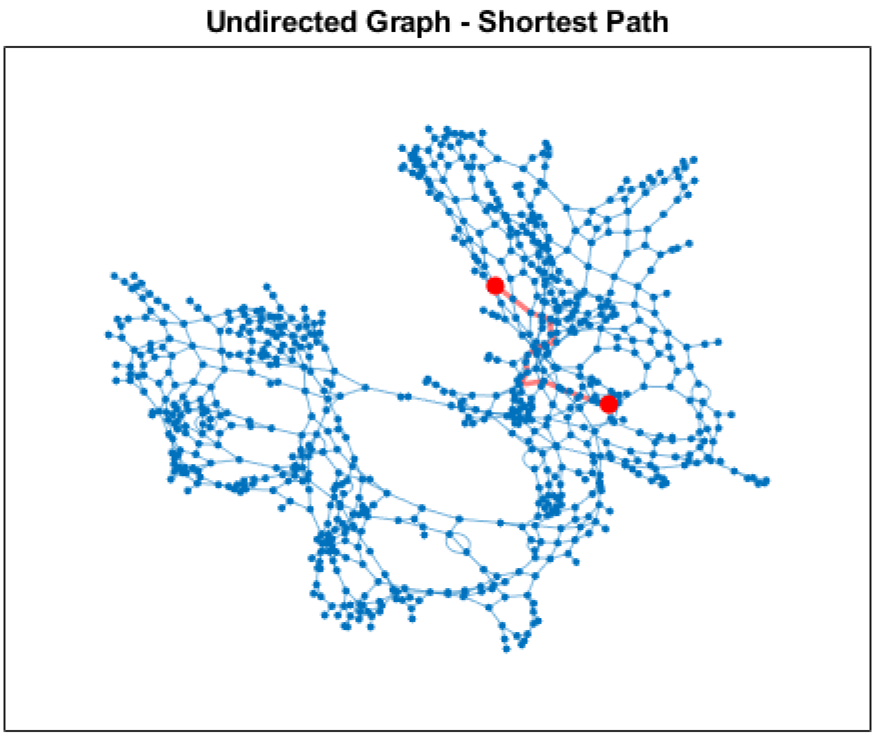
Specify a start and end node to find the shortest path between them. Use the first and last node in the graph as the start and end nodes, respectively. Overlay the nodes on the graph.

```
startNode = G.Nodes.Name(1);
endNode = G.Nodes.Name(end);

highlight(H,[startNode endNode],'NodeColor','red','MarkerSize',6)
title('Undirected Graph - Start and End Nodes')
```



Find the shortest path between the two nodes. Plot the path.

```
path = shortestpath(G,startNode,endNode);
highlight(H,path,'EdgeColor','red','LineWidth',2);
title('Undirected Graph - Shortest Path')
```



Limitations

- The HERE HDLM web service determines the geographic coverage of the map data. Map data is not available for all locations.

Tips

- To speed up the performance of the reader, when creating the reader, specify a `hereHDLMConfiguration` object for the `Configuration` property. This object configures the reader to search for the selected map tiles only a specific HERE HD Live Map production catalog. If you do not specify a configuration object when you create the reader, the reader searches for the map tiles across all catalogs.
- To save HERE HDLM credentials between MATLAB sessions, select the corresponding option in the HERE HD Live Map Credentials dialog box. To manage HERE HDLM credentials, use the `hereHDLMCredentials` function.

See Also

`geoplayer` | `geoplot` | `hereHDLMConfiguration` | `hereHDLMCredentials`

Topics

"Read and Visualize HERE HD Live Map Data"
"HERE HD Live Map Layers"

"Use HERE HD Live Map Data to Verify Lane Configurations"

Introduced in R2019a

read

Read HERE HD Live Map layer data

Syntax

```
layerData = read(reader,layerType)
layerData = read(reader,layerType,fields)
```

Description

`layerData = read(reader,layerType)` reads HERE HD Live Map[12] (HERE HDLM) data of a specified layer type from a `hereHDLMReader` object and returns an array of layer objects. These layer objects contain map layer data for the HERE map tiles whose IDs correspond to the IDs stored in the `TileIds` property of `reader`.

`layerData = read(reader,layerType,fields)` returns an array of layer objects containing data for only the required fields, such as the `HereTileId` field, and for the specified fields. All other fields in the returned layer objects are returned as empty: `[]`. If you do not require data from all fields within the layer objects, use this syntax to speed up performance of this function.

Examples

Plot and Stream Lane Topology Data from Driving Route

Use the HERE HD Live Map (HERE HDLM) service to read the lane topology data of a driving route and its surrounding area. Plot this data, and then stream the route on a geographic player.

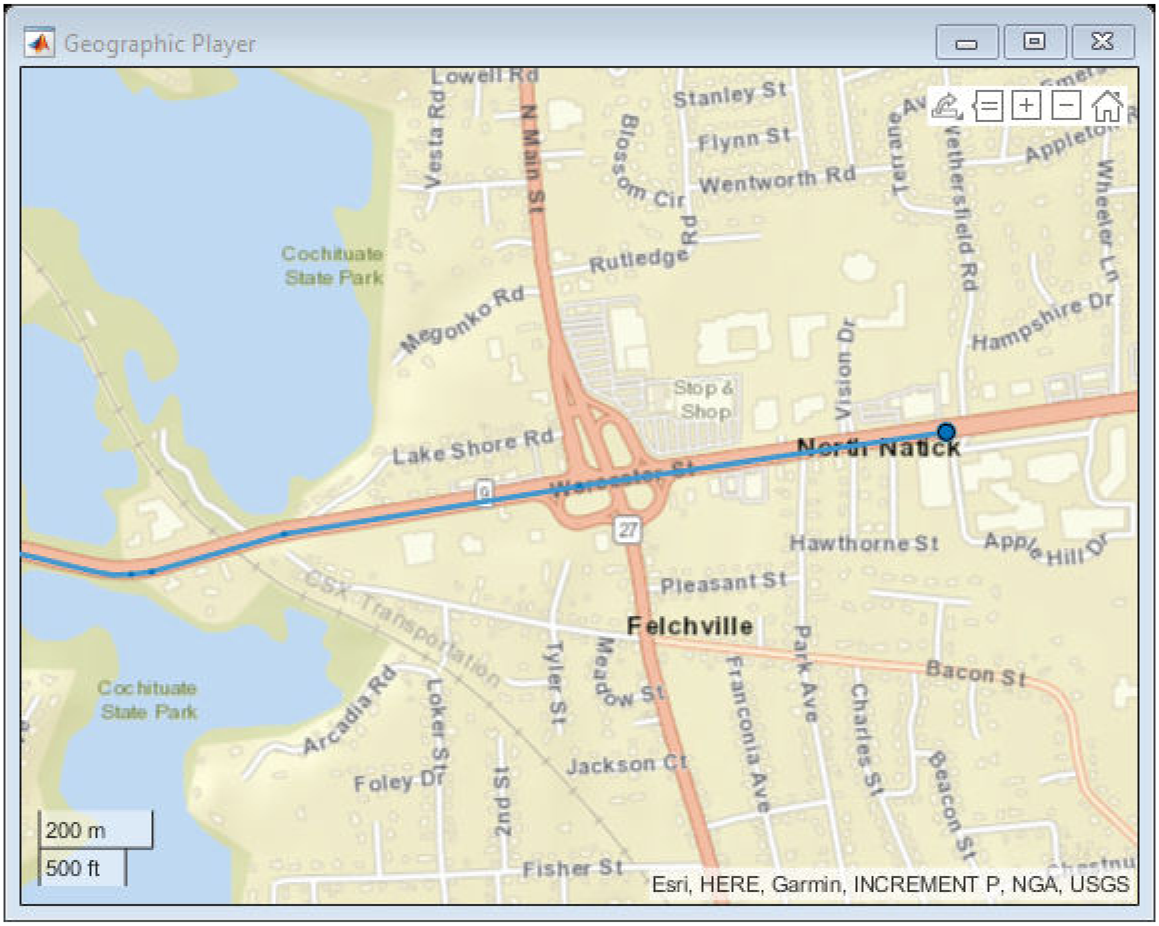
Load the latitude and longitude coordinates of a driving route in Natick, Massachusetts, USA.

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

Stream the coordinates on a geographic player.

```
player = geoplayer(lat(1),lon(1),'HistoryDepth',5);
plotRoute(player,lat,lon)

for idx = 1:length(lat)
    plotPosition(player,lat(idx),lon(idx))
end
```

12. 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.

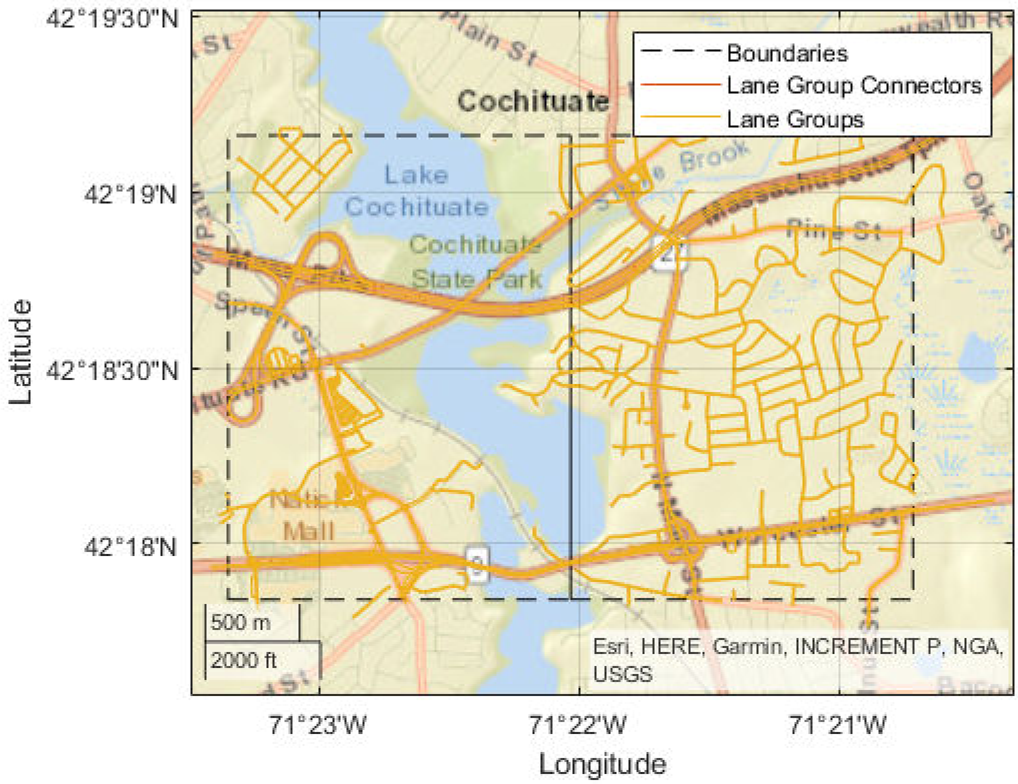


Create a HERE HDLM reader from the route coordinates. If you have not previously set up HERE HDLM credentials, a dialog box prompts you to enter them. The reader contains map data for the two map tiles that the route crosses.

```
reader = hereHDLMReader(lat,lon);
```

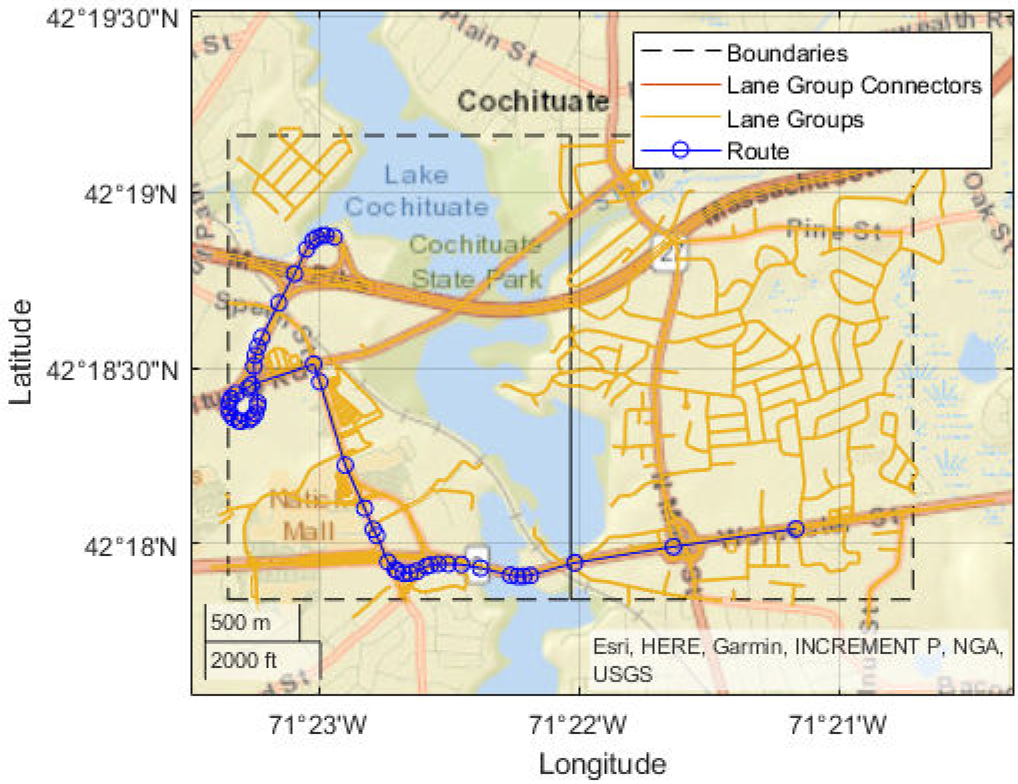
Read lane topology data from the `LaneTopology` layer of the map tiles. Plot the lane topology.

```
laneTopology = read(reader,'LaneTopology');
plot(laneTopology)
```



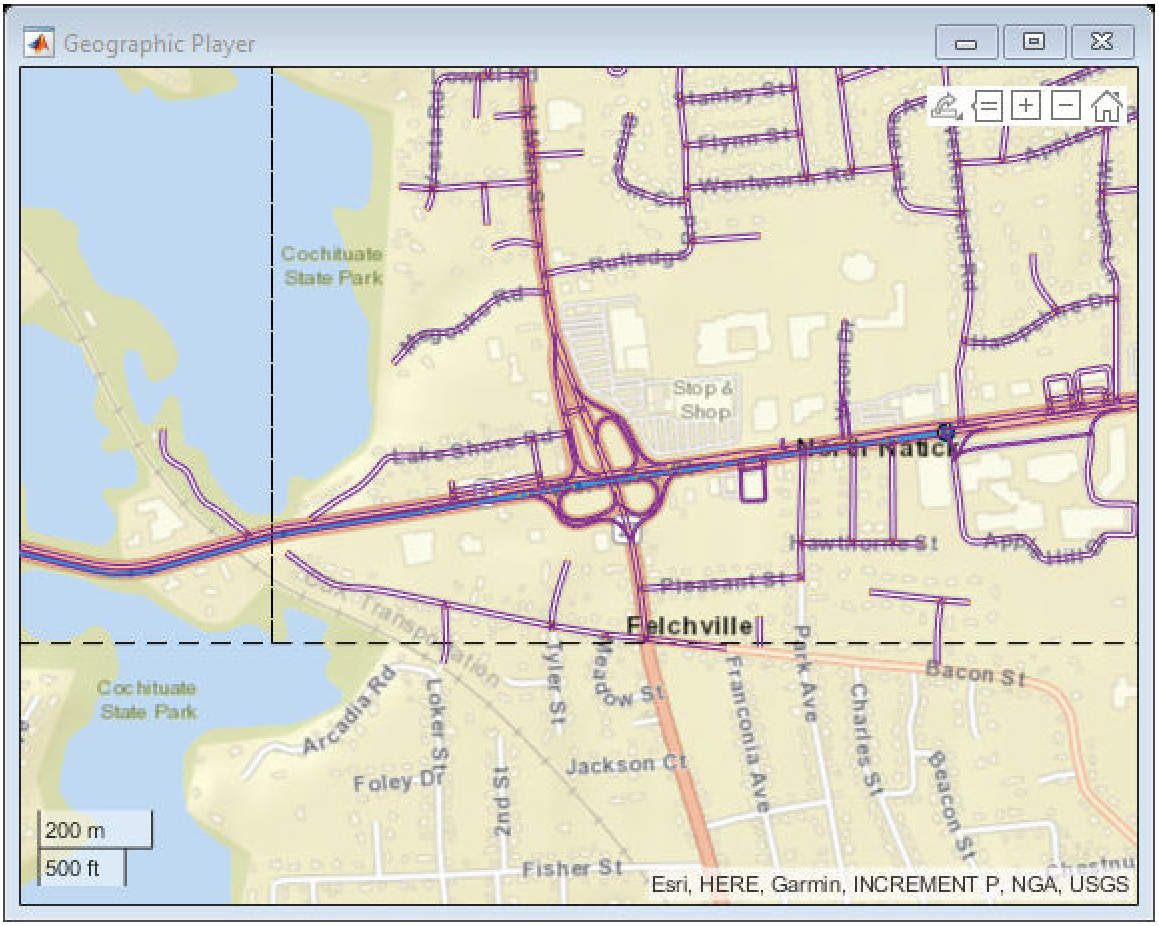
Overlay the route data on the plot.

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



Overlay the lane topology data on the geographic player. Stream the route again.

```
plot(laneTopology,'Axes',player.Axes)
for idx = 1:length(lat)
    plotPosition(player,lat(idx),lon(idx))
end
```



Find Shortest Path Between Two Nodes

Use the HERE HD Live Map (HERE HDLM) web service to read the topology geometry data from a map tile. Use this data to find the shortest path between two nodes within the map tile.

Define a HERE tile ID for an area of Stockholm, Sweden.

```
tileID = uint32(378373553);
```

Create a HERE HDLM reader for the tile ID. Configure the reader to search for the tile in only the Western Europe catalog. If you have not previously set up HERE HDLM credentials, a dialog box prompts you to enter them. The reader contains map data for the specified map tile.

```
config = hereHDLMConfiguration('hrn:here:data::olp-here-had:here-hdlm-protobuf-weu-2');
reader = hereHDLMReader(tileID,'Configuration',config);
```

Read the link definitions from the `TopologyGeometry` layer of the map tile. The returned layer object contains the specified `LinksStartingInTile` field and the required map tile fields, such as the tile ID. The other fields are empty. Your map data and catalog version might differ from the ones shown here.

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

```
topology = 
  TopologyGeometry with properties:

   Data:
                   HereTileId: 378373553
         IntersectingLinkRefs: []
          LinksStartingInTile: [1249×1 struct]
                  NodesInTile: []
    TileCenterHere2dCoordinate: [59.3372 18.0505]

   Metadata:
                      Catalog: 'hrn:here:data::olp-here-had:here-hdlm-protobuf-weu-2'
               CatalogVersion: 5597

  Use plot to visualize TopologyGeometry data.
```

Find the start and end nodes for each link in the `LinksStartingInTile` field.

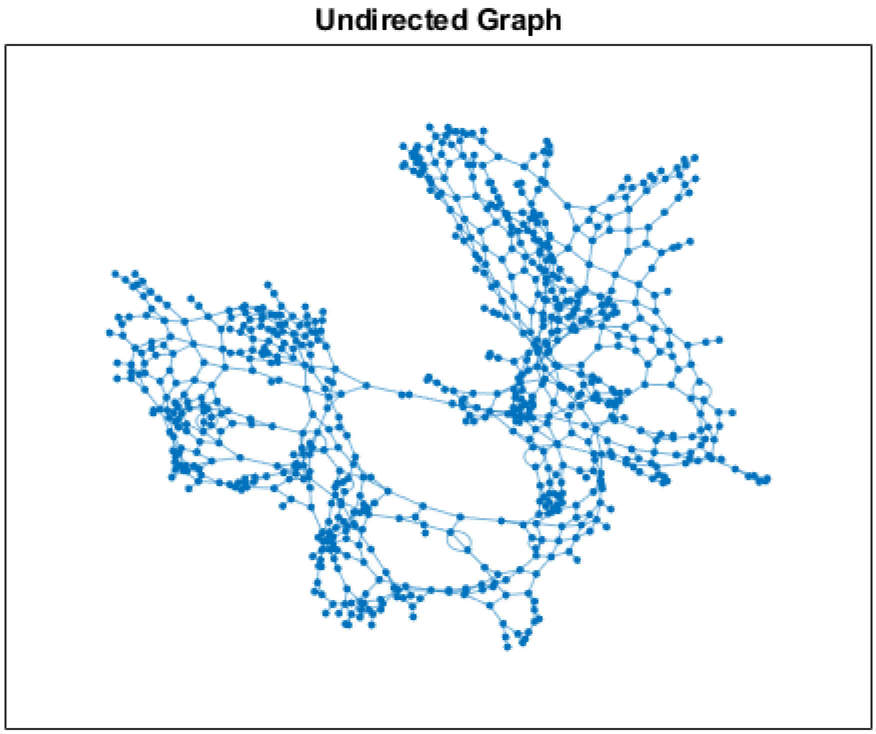
```
startNodes = [topology.LinksStartingInTile.StartNodeId];
endNodesRef = [topology.LinksStartingInTile.EndNodeRef];
endNodes = [endNodesRef.NodeId];
```

Find the length of each link in meters.

```
linkLengths = [topology.LinksStartingInTile.LinkLengthMeters];
```

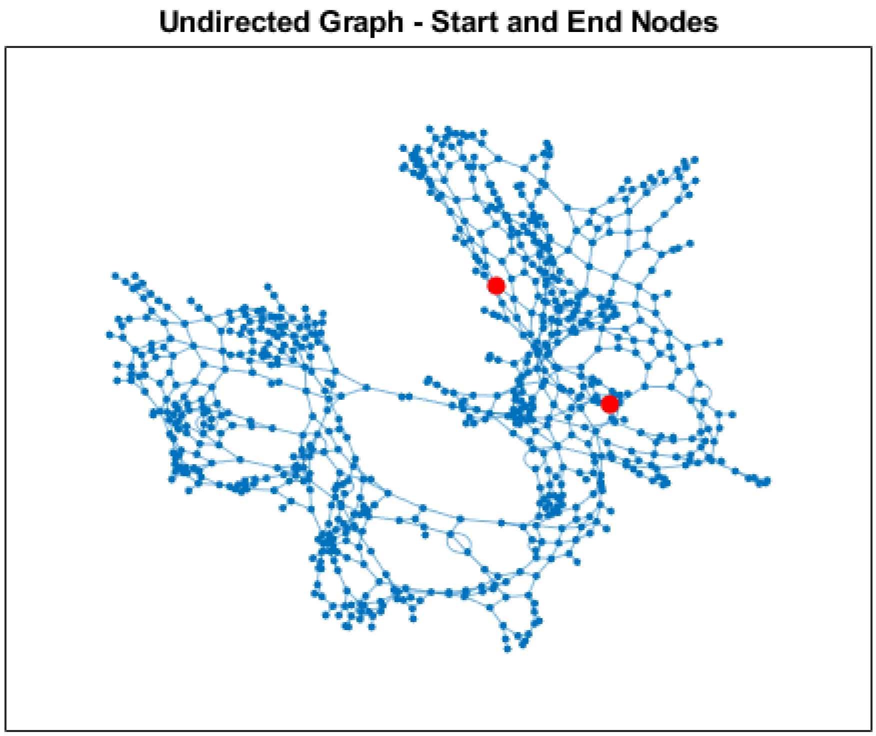
Create an undirected graph for the links in the map tile.

```
G = graph(string(startNodes),string(endNodes),double(linkLengths));
H = plot(G,'Layout','force');
title('Undirected Graph')
```



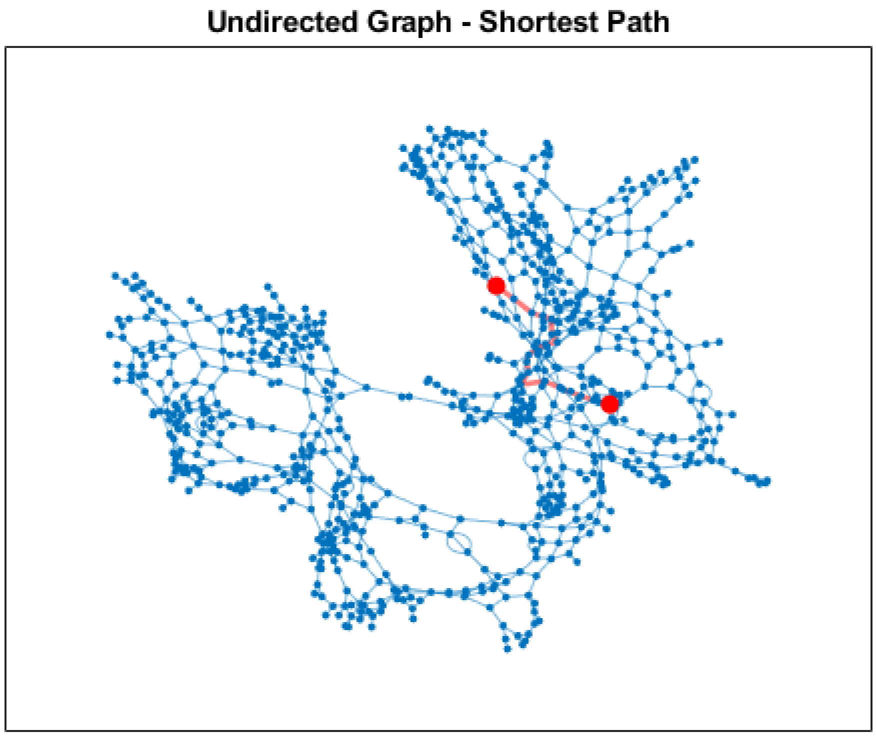
Specify a start and end node to find the shortest path between them. Use the first and last node in the graph as the start and end nodes, respectively. Overlay the nodes on the graph.

```
startNode = G.Nodes.Name(1);
endNode = G.Nodes.Name(end);

highlight(H,[startNode endNode],'NodeColor','red','MarkerSize',6)
title('Undirected Graph - Start and End Nodes')
```



Find the shortest path between the two nodes. Plot the path.

```
path = shortestpath(G,startNode,endNode);
highlight(H,path,'EdgeColor','red','LineWidth',2);
title('Undirected Graph - Shortest Path')
```



Input Arguments

reader — Input HERE HDLM reader
hereHDLMReader object

Input HERE HDLM reader, specified as a `hereHDLMReader` object.

layerType — Layer type
string scalar | character vector

Layer type from which to read data, specified as a string scalar or character vector. `layerType` must be a valid layer type for the map tiles stored in `reader`. To see the list of valid layers, use the `Layers` property of `reader`.

Example: `"AdasAttributes"`

Example: `'LaneTopology'`

fields — Layer object fields
string scalar | character vector | string array | cell array of character vectors

Layer object fields from which to read data, specified as a string scalar, character vector, string array, or cell array of character vectors. All fields must be valid fields of the layer specified by `layerType`. You can specify only the top-level fields of this layer. You cannot specify its metadata fields.

In the returned array of layer objects, only required fields, such as the HereTileId field, and the specified fields contain data. All other fields are returned as empty: [].

For a list of the valid top-level data fields for each layer type, see the data output argument.

Example: 'LinkAttribution'

Example: "NodeAttribution"

Example: ["LinkAttribution" "NodeAttribution"]

Example: {'LinkAttribution','NodeAttribution'}

Output Arguments

layerData — HERE HDLM layer data

T-by-1 array of layer objects

HERE HDLM layer data, returned as a *T*-by-1 array of layer objects. *T* is the number of map tile IDs stored in the TileIds property of the specified reader. Each layer object contains map data that is of type layerType for a HERE map tile that was read from reader. Such data can include:

- The geometry of links (streets) and nodes (intersections and dead ends) within map tiles
- Various road-level and lane-level attributes
- Landmark-based localization information, such as the barriers, signs, and poles along a road

The layer objects also contain metadata specifying the catalog name and catalog version from which the read function obtained the data.

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

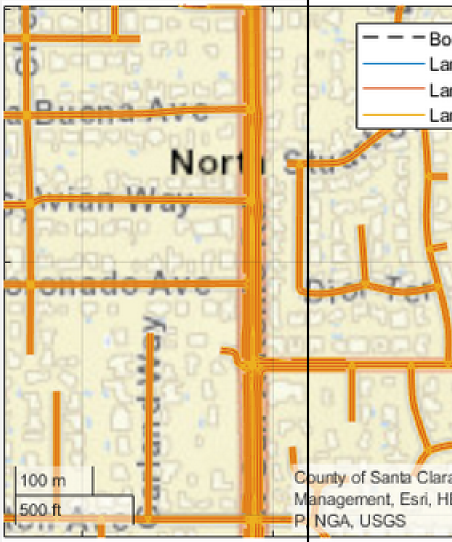
HERE HDLM Layer Fields	MATLAB Layer Object Property
here_tile_id	HereTileId
tile_center_here_2d_coordinate	TileCenterHere2dCoordinate
nodes_in_tile	NodesInTile

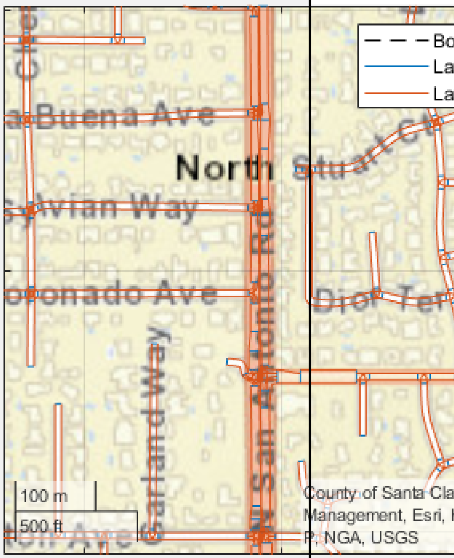
The layer objects are MATLAB structures whose properties correspond to structure fields. To access data from these fields, use dot notation.

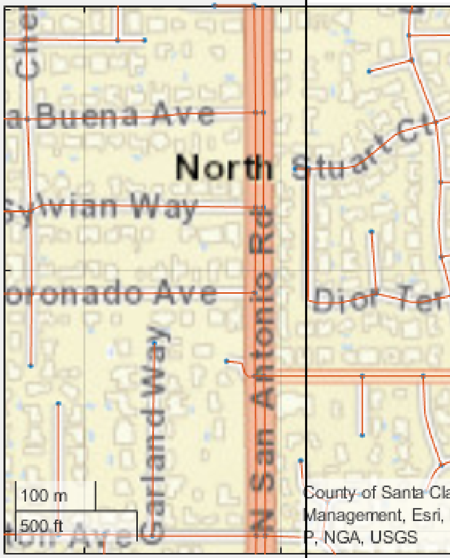
For example, this code selects the NodeId subfield from the NodeAttribution field of a layer:

```
layerData.NodeAttribution.NodeId
```

This table summarizes the valid types of layer objects and their top-level data fields. The available layers are for the Road Centerline Model, HD Lane Model, and HD Localization Model. For an overview of HERE HDLM layers and the models that they belong to, see “HERE HD Live Map Layers”.

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

Layer Object	Description	Top-Level Data Fields (Layer Object Properties)	Plot Support
`LaneTopology`	Topologies of the HD Lane model, including lane group, lane group connector, lane, and lane connector topologies. This layer also contains the simplified 2-D boundary geometry of the lane model for determining map tile affinity and overflow.	• `HereTileId` • `TileCenterHere2dCoordinate` • `LaneGroupsStartingInTile` • `LaneGroupConnectorsInTile` • `IntersectingLaneGroupRefs`	Available — Use the `plot` function. 
`LocalizationBarrier`	Positions, dimensions, and attributes of barriers such as guardrails and Jersey barriers found along roads	• `HereTileId` • `TileCenterHere3dCoordinate` • `Barriers` • `RoadToBarriersReferences` • `IntersectingBarrierRefs`	Not available
`LocalizationPole`	Positions, dimensions, and attributes of traffic signal poles and other poles found along or hanging over roads	• `HereTileId` • `TileCenterHere3dCoordinate` • `Signs` • `RoadToSignsReferences`	Not available
`LocalizationSign`	Positions, dimensions, and attributes of traffic-sign faces found along roads	• `HereTileId` • `TileCenterHere3dCoordinate` • `Poles` • `RoadToPolesReferences`	Not available

Layer Object	Description	Top-Level Data Fields (Layer Object Properties)	Plot Support
`RoutingAttributes`	Road attributes related to navigation and conditions. These attributes are mapped parametrically to the 2-D polyline geometry in the topology layer.	• `HereTileId` • `LinkAttribution` • `NodeAttribution` • `StrandAttribution` • `AttributionGroupList`	Not available
`RoutingLaneAttributes`	Core navigation lane attributes and conditions, such as the number of lanes in a road. These values are mapped parametrically to 2-D polylines along the road links.	• `HereTileId` • `LinkAttribution`	Not available
`SpeedAttributes`	Speed-related road attributes, such as speed limits. These attributes are mapped to the 2-D polyline geometry of the topology layer.	• `HereTileId` • `LinkAttribution`	Not available
`TopologyGeometry`	Topology and 2-D line geometry of the road. This layer also contains definitions of the nodes and links in the map tile.	• `HereTileId` • `TileCenterHere2dCoordinate` • `NodesInTile` • `LinksStartingInTile` • `IntersectingLinkRefs`	Available — Use the `plot` function. 

See Also
`hereHDLMConfiguration` | `hereHDLMCredentials` | `hereHDLMReader` | `plot`

Topics
"Read and Visualize HERE HD Live Map Data"
"HERE HD Live Map Layers"
"Use HERE HD Live Map Data to Verify Lane Configurations"

Introduced in R2019a

plot

Package: `driving.heremaps`

Plot HERE HD Live Map layer data

Syntax

```
plot(layerData)
plot(layerData,'Axes',gxIn)
gxOut = plot( ___ )
```

Description

`plot(layerData)` plots HERE HD Live Map[13] (HERE HDLM) layer data on a geographic axes. `layerData` is a map layer object that was read from the selected tiles of a `hereHDLMReader` object by using the `read` function.

`plot(layerData,'Axes',gxIn)` plots the layer data in the specified geographic axes, `gxIn`.

`gxOut = plot( ___ )` plots the layer data and returns the geographic axes on which the data was plotted, using the inputs from any of the preceding syntaxes. Use `gxOut` to modify properties of the geographic axes.

Examples

Plot Road Topology Data from Driving Route

Load a sequence of latitude and longitude coordinates from a driving route.

```
data = load('geoSequence.mat')
```

```
data = struct with fields:
     latitude: [1000×1 double]
    longitude: [1000×1 double]
```

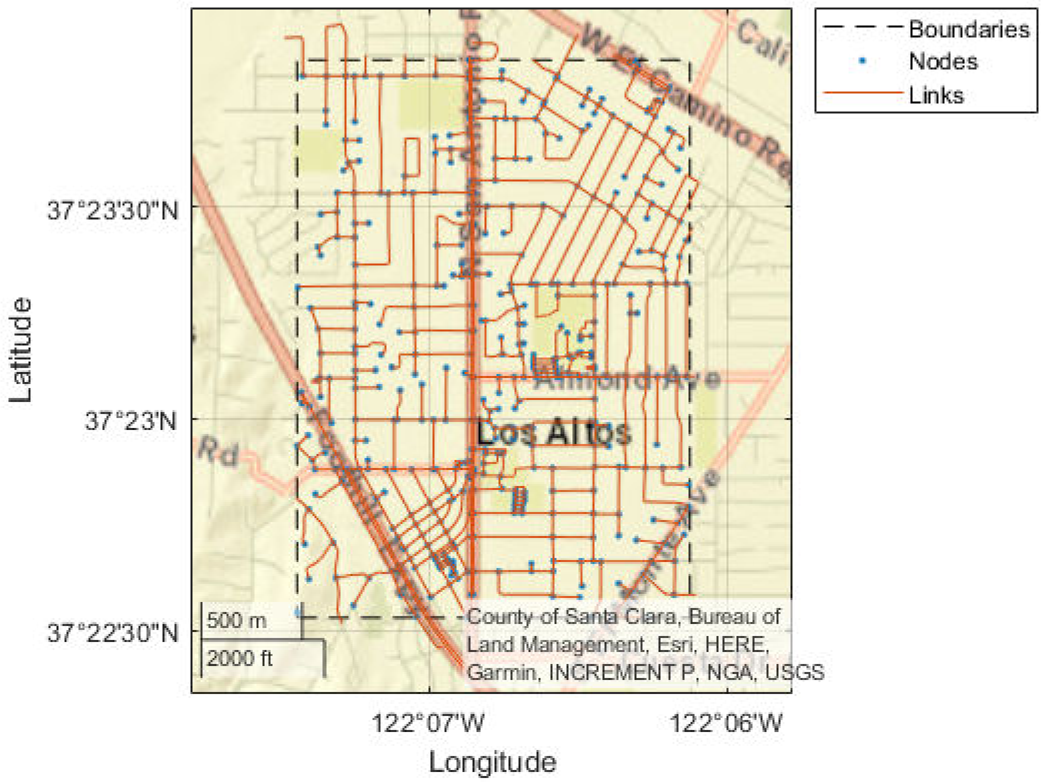
Create a HERE HD Live Map (HERE HDLM) reader from the specified coordinates. If you have not previously set up HERE HDLM credentials, a dialog box prompts you to enter them. The reader contains layered map data for the tile that the driving route is on.

```
reader = hereHDLMReader(data.latitude,data.longitude);
```

Read road topology data from the `TopologyGeometry` layer. Plot the data.

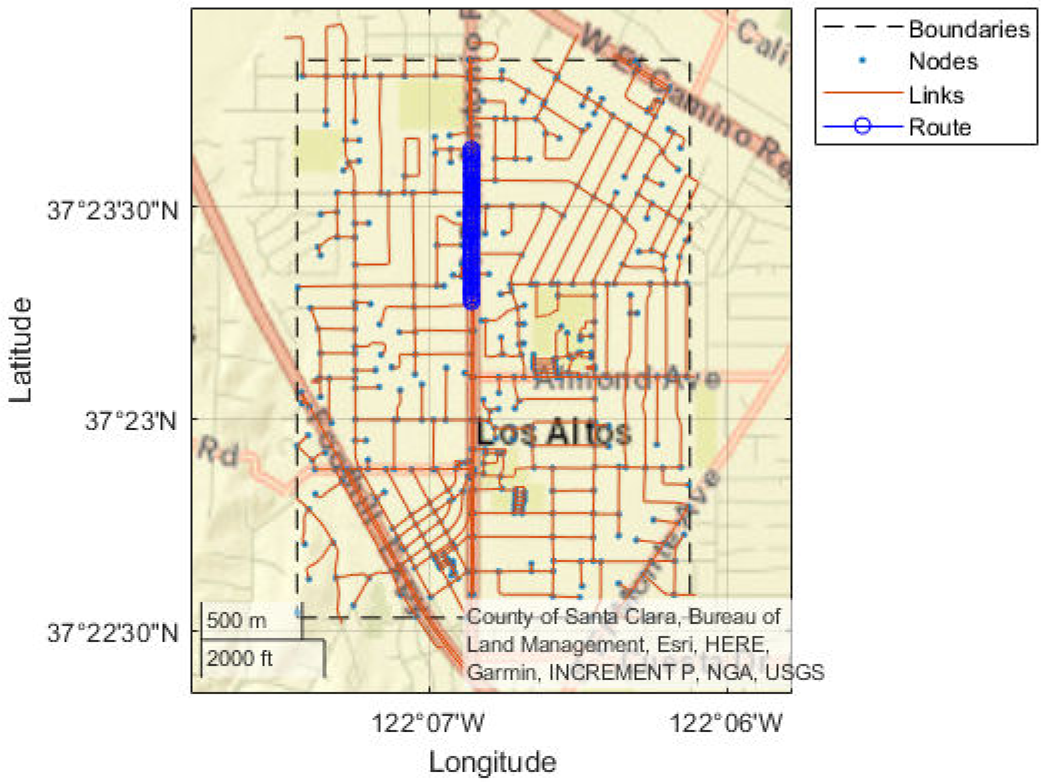
```
roadTopology = read(reader,'TopologyGeometry');
plot(roadTopology)
legend('Location','northeastoutside')
```

13. 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.



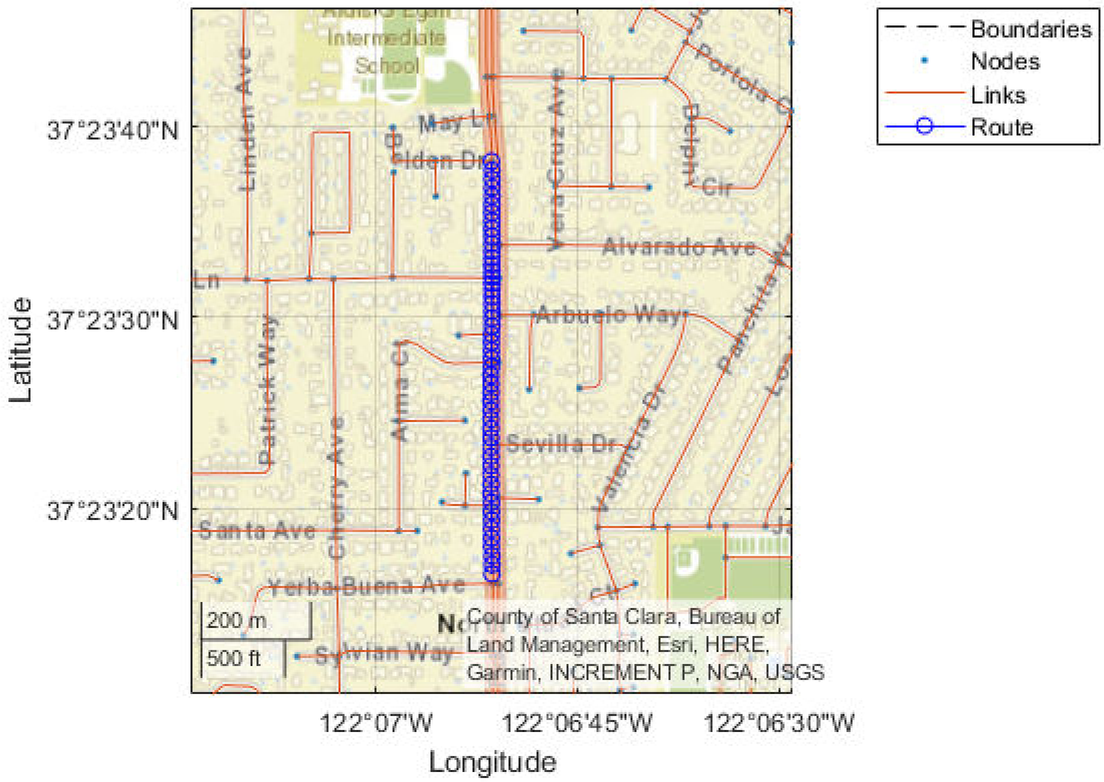
Overlay the driving route coordinates on the plot.

```
hold on
geoplot(data.latitude,data.longitude,'bo-','DisplayName','Route')
hold off
```



Zoom in on the route.

```
latcenter = median(data.latitude);
loncenter = median(data.longitude);

offset = 0.005;
latlim = [latcenter-offset,latcenter+offset];
lonlim = [loncenter-offset,loncenter+offset];

geolimits(latlim,lonlim)
```



Plot and Stream Lane Topology Data from Driving Route

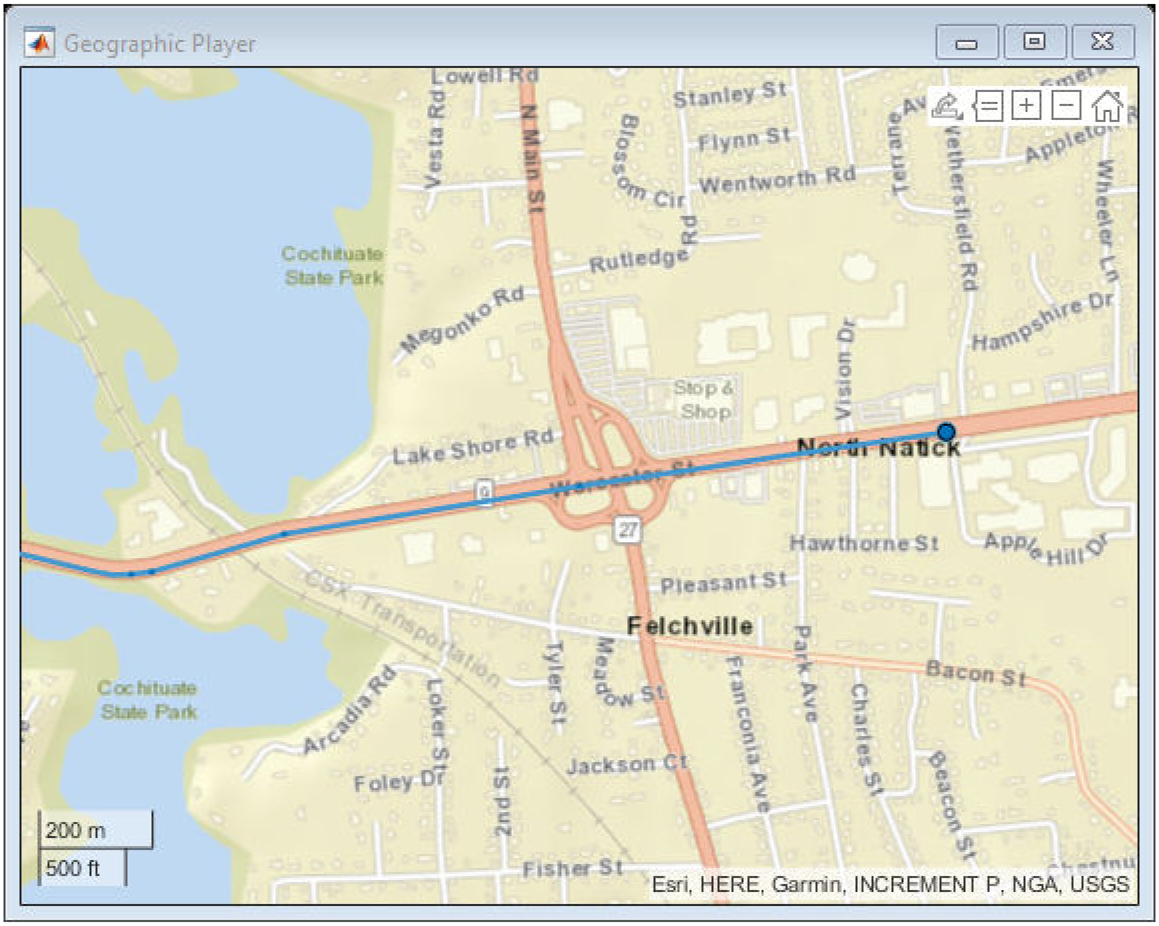
Use the HERE HD Live Map (HERE HDLM) service to read the lane topology data of a driving route and its surrounding area. Plot this data, and then stream the route on a geographic player.

Load the latitude and longitude coordinates of a driving route in Natick, Massachusetts, USA.

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

Stream the coordinates on a geographic player.

```
player = geoplayer(lat(1),lon(1),'HistoryDepth',5);
plotRoute(player,lat,lon)

for idx = 1:length(lat)
    plotPosition(player,lat(idx),lon(idx))
end
```

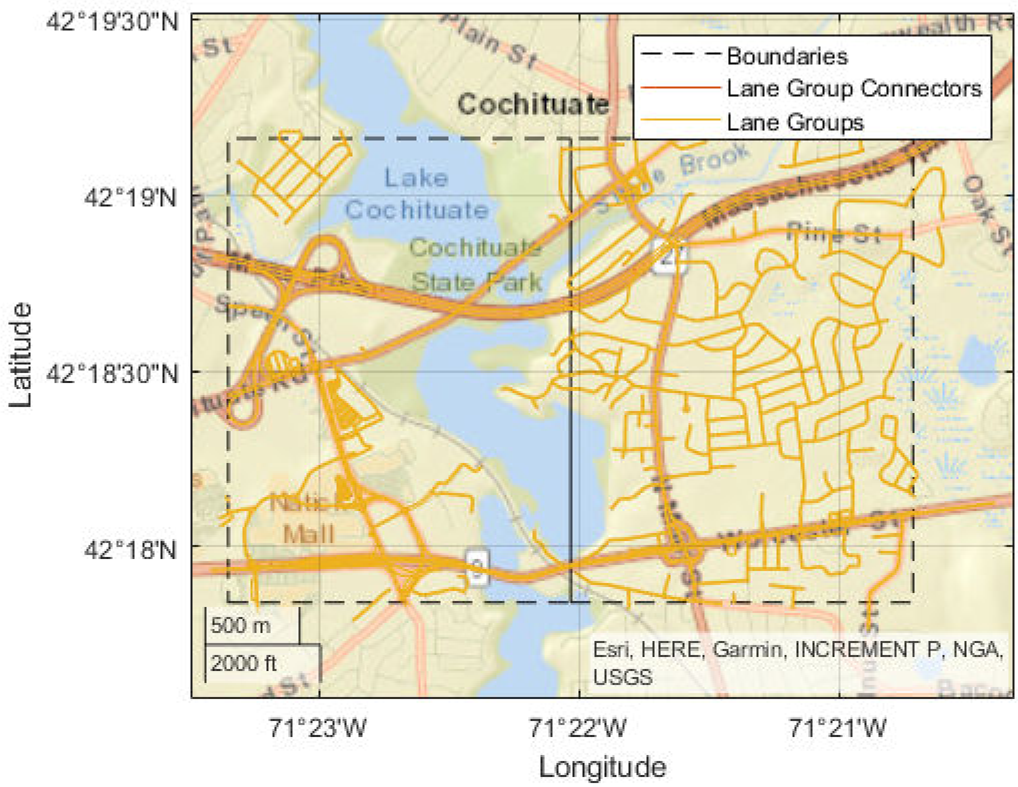


Create a HERE HDLM reader from the route coordinates. If you have not previously set up HERE HDLM credentials, a dialog box prompts you to enter them. The reader contains map data for the two map tiles that the route crosses.

```
reader = hereHDLMReader(lat,lon);
```

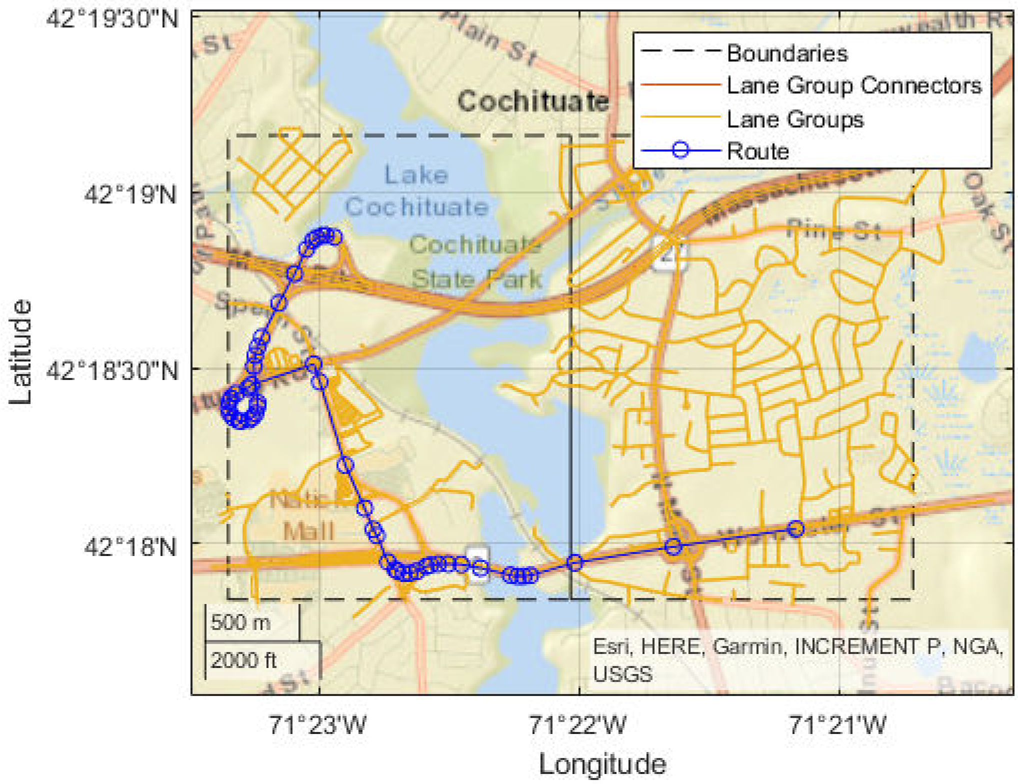
Read lane topology data from the `LaneTopology` layer of the map tiles. Plot the lane topology.

```
laneTopology = read(reader,'LaneTopology');
plot(laneTopology)
```



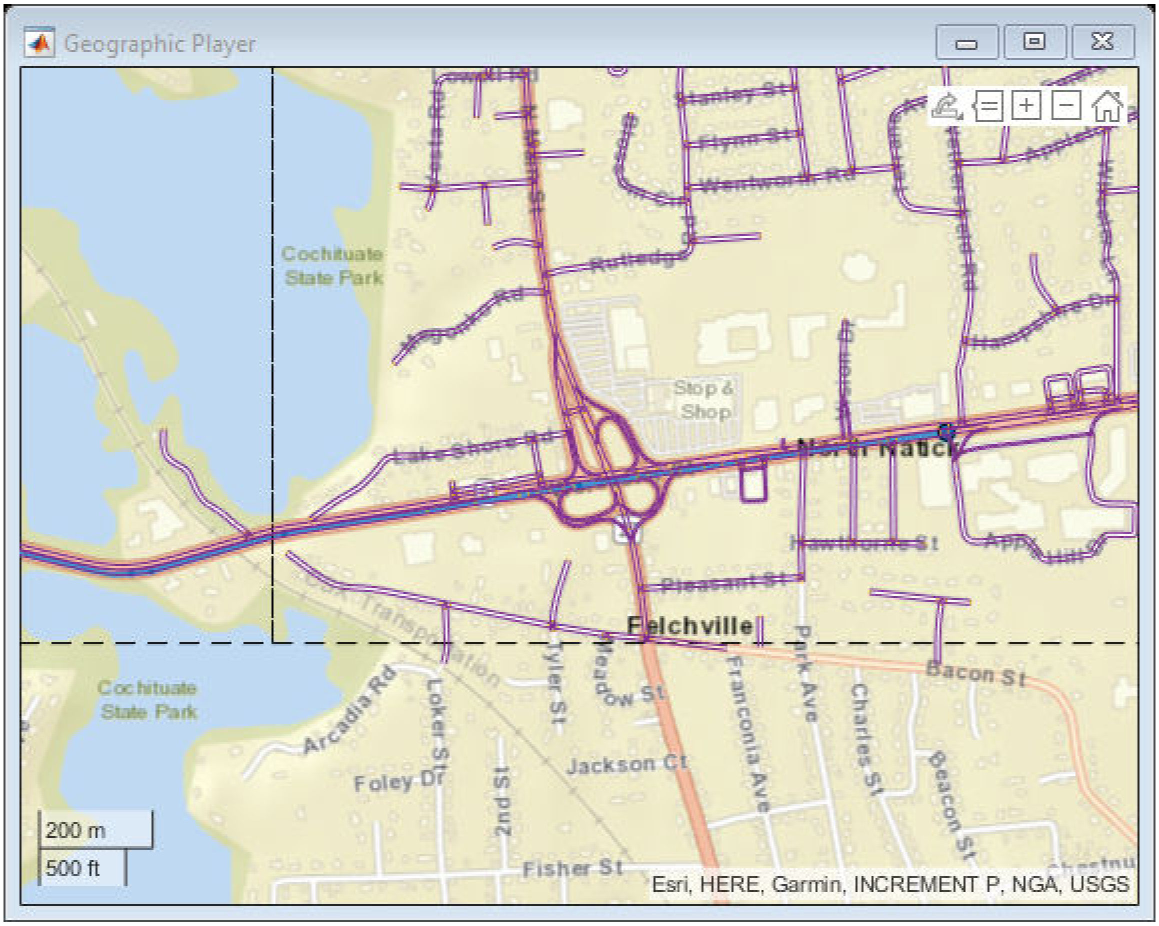
Overlay the route data on the plot.

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



Overlay the lane topology data on the geographic player. Stream the route again.

```
plot(laneTopology,'Axes',player.Axes)
for idx = 1:length(lat)
    plotPosition(player,lat(idx),lon(idx))
end
```



Plot 3-D Lane Geometry on Custom Basemap

Use the HERE HD Live Map (HERE HDLM) web service to read 3-D lane geometry data from a map tile. Then, plot the data on an OpenStreetMap® basemap.

Create a HERE HDLM reader for a map tile ID representing an area of Berlin, Germany. If you have not previously set up HERE HDLM credentials, a dialog box prompts you to enter them.

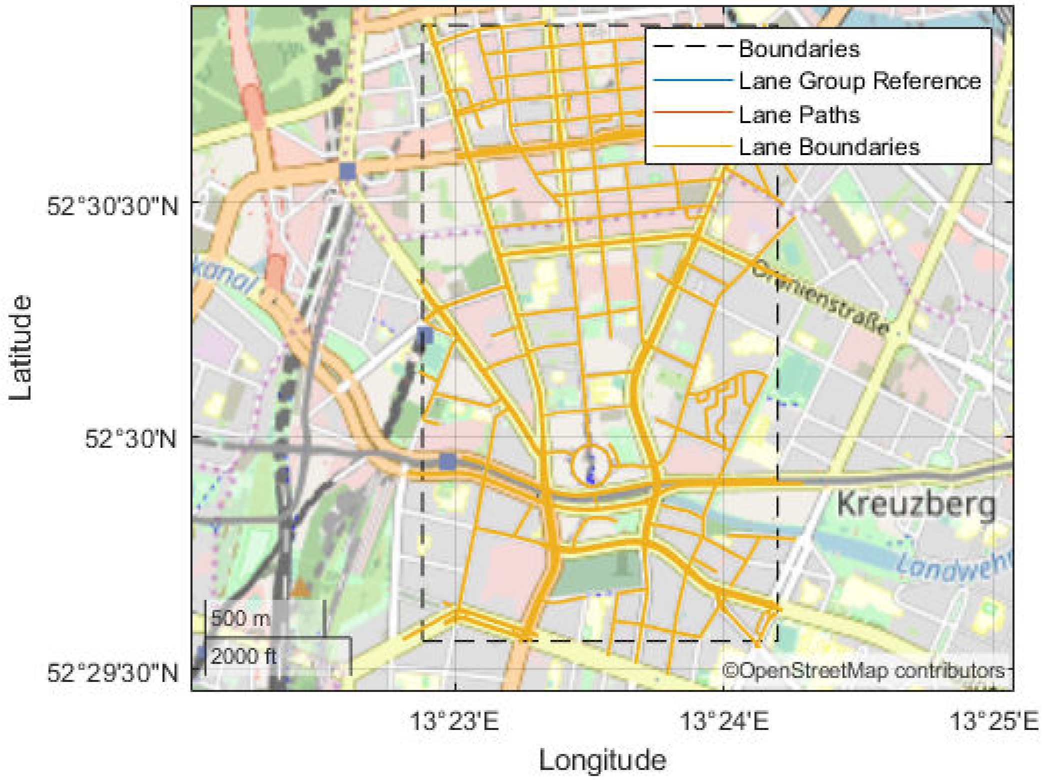
```
tileID = uint32(377894435);
reader = hereHDLMReader(tileID);
```

Add the OpenStreetMap basemap to the list of basemaps available for use with the HERE HDLM service. 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)
```

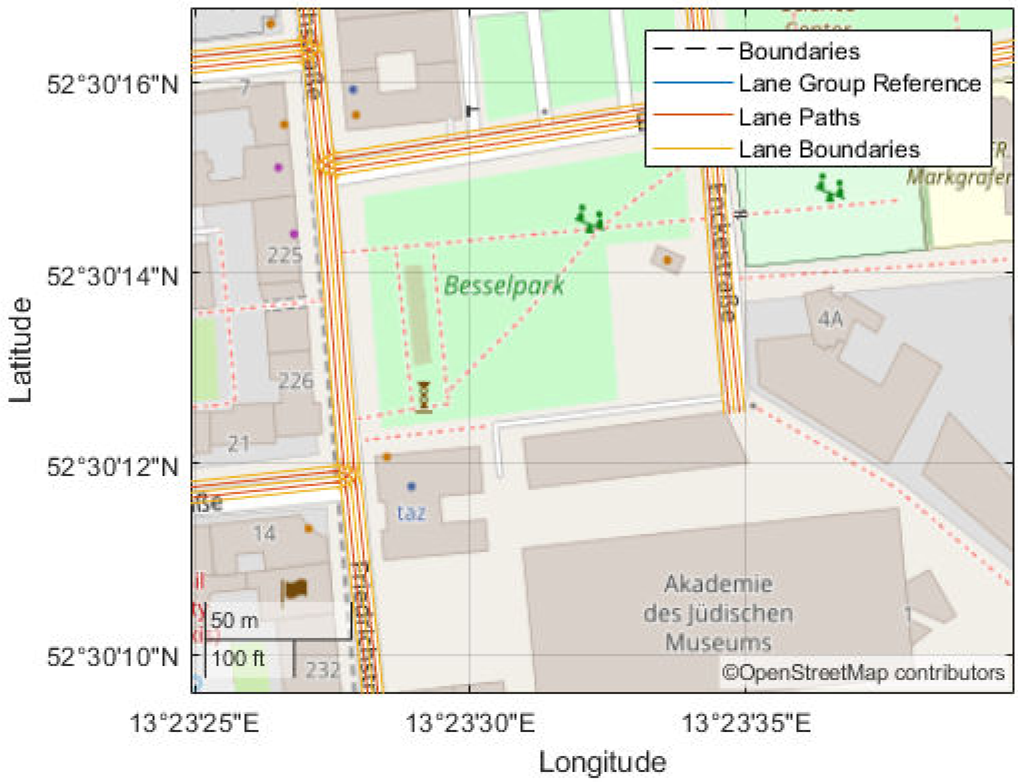
Read 3-D lane geometry data from the `LaneGeometryPolyline` layer of the map tile. Plot the lane geometry on the `openstreetmap` basemap.

```
laneGeometryPolyline = read(reader,'LaneGeometryPolyline');
gx = plot(laneGeometryPolyline);
geobasemap(gx,'openstreetmap')
```



Zoom in on the central coordinate of the map tile.

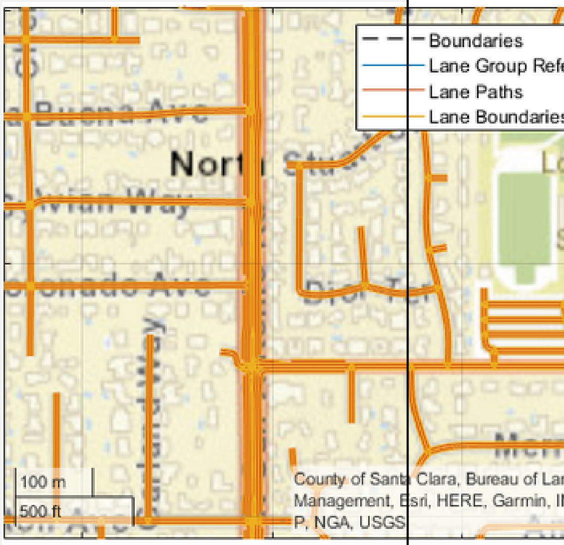
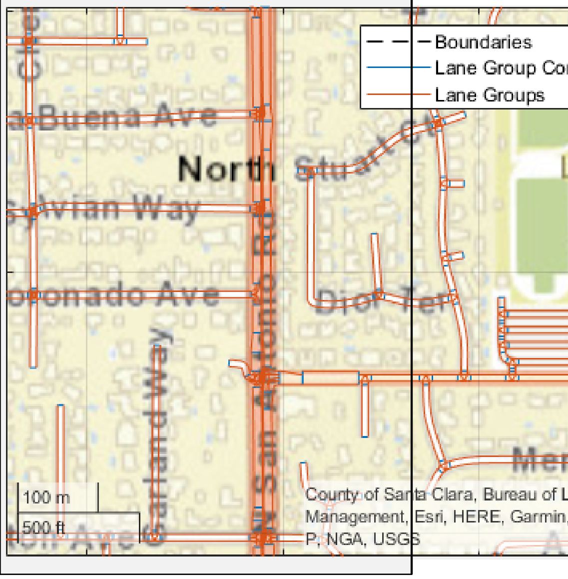
```
latcenter = laneGeometryPolyline.TileCenterHere3dCoordinate.Here2dCoordinate(1);
loncenter = laneGeometryPolyline.TileCenterHere3dCoordinate.Here2dCoordinate(2);

offset = 0.001;
latlim = [latcenter-offset,latcenter+offset];
lonlim = [loncenter-offset,loncenter+offset];

geolimits(latlim,lonlim)
```

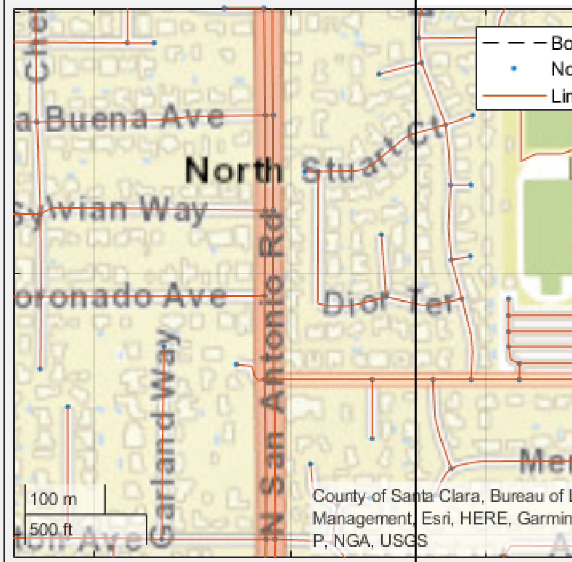


Input Arguments

layerData — HERE HDLM layer data
LaneGeometryPolyline object | LaneTopology object | TopologyGeometry object

HERE HDLM layer data to plot, specified as one of the layer objects shown in the table.

Layer Object	Description	Sample Plot
`LaneGeometryPolyline`	3-D lane geometry composed of a set of 3-D points joined into polylines.	
`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.	

Layer Object	Description	Sample Plot
`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.	

To obtain these layers from map tiles selected by a `hereHDLMReader` object, use the `read` function.

gxIn — Geographic axes on which to plot data
`GeographicAxes` object

Geographic axes on which to plot data, specified as a `GeographicAxes` object.[14]

Output Arguments

gxOut — Geographic axes on which data is plotted
`GeographicAxes` object

Geographic axes on which data is plotted, returned as a `GeographicAxes` object. Use this object to customize the map display. For more details, see GeographicAxes Properties.

See Also

`geoaxes` | `geobasemap` | `geoplayer` | `geoplot` | `hereHDLMReader` | `read`

Topics
GeographicAxes Properties
"Read and Visualize HERE HD Live Map Data"
"Use HERE HD Live Map Data to Verify Lane Configurations"

Introduced in R2019a

14. Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

hereHDLMConfiguration

Configure HERE HD Live Map reader

Description

A `hereHDLMConfiguration` object configures a `hereHDLMReader` object to search for map data in only a specific HERE HD Live Map[15] (HDLM) production catalog or catalog version. These catalogs roughly correspond to various geographic regions, such as Western Europe and North America. Using this configuration object can speed up the performance of the reader, so that it does not search unnecessary catalogs. The configuration object is stored in the `Configuration` property of a `hereHDLMReader` object.

Note Use of the `hereHDLMConfiguration` object requires valid HERE HDLM credentials. If you have not previously set up 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**.

Creation

Syntax

```
config = hereHDLMConfiguration(catalog)
config = hereHDLMConfiguration(catalog,catalogVersion)
```

Description

`config = hereHDLMConfiguration(catalog)` creates a `hereHDLMConfiguration` object for the latest version of the specified HERE HDLM catalog. A `hereHDLMReader` object with this configuration searches for the selected map tiles within only the catalog and version specified by that configuration.

`config = hereHDLMConfiguration(catalog,catalogVersion)` creates a `hereHDLMConfiguration` object for the specified version of the catalog.

Input Arguments

`catalog` — Name of HERE HDLM production catalog
string scalar | character vector

Name of HERE HDLM production catalog, specified as a string scalar or character vector. You can obtain production catalog names from HERE Technologies.

Example: `'hrn:here:data::olp-here-had:here-hdlm-protobuf-na-2'` specifies a catalog that roughly corresponds to the North America region.

Example: `'hrn:here:data::olp-here-had:here-hdlm-protobuf-weu-2'` specifies a catalog that roughly corresponds to the Western Europe region.

15. 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.

`catalogVersion` — Version number of HERE HDLM production catalog
positive integer

Version number of a HERE HDLM production catalog, specified as a positive integer. The HERE HDLM web service determines the availability of previous versions of the catalog. If you specify a version of a catalog that is not available, then `hereHDLMConfiguration` returns an error.

Properties

`Catalog` — Name of HERE HDLM production catalog
string scalar | character vector

This property is read-only.

Name of a HERE HDLM production catalog, specified as a string scalar or character vector. This property is set to the name of the catalog specified by the `catalog` input.

`CatalogVersion` — Version number of HERE HDLM production catalog
positive integer

This property is read-only.

Version number of a HERE HDLM production catalog, specified as a positive integer. The version number corresponds to the value specified in the `catalogVersion` input argument. If you do not specify `catalogVersion`, then this property is set to the latest version of the specified catalog.

Examples

Create Configuration for Specific Catalog

Define a HERE tile ID for an area of Berlin, Germany.

```
tileID = uint32(377894435);
```

Create a HERE HD Live Map (HERE HDLM) configuration object for the catalog that roughly corresponds to Western Europe. If you have not previously set up HERE HDLM credentials, a dialog box prompts you to enter them. Your catalog version might differ from the one shown here.

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

```
config = 
  hereHDLMConfiguration with properties:

           Catalog: 'hrn:here:data::olp-here-had:here-hdlm-protobuf-weu-2'
    CatalogVersion: 5597
```

Create a HERE HDLM reader using the specified HERE tile ID and configuration object. During creation, `hereHDLMReader` searches for the tile ID within only the Western Europe catalog. This reader is configured to read map data from only that catalog.

```
reader = hereHDLMReader(tileID,'Configuration',config);
```

Create Configuration for Specific Catalog Version

Create a HERE HD Live Map (HERE HDLM) configuration object for the previous version of a catalog.

Load a sequence of latitude and longitude coordinates for a driving route in Los Altos, California, USA.

```
data = load('geoSequence.mat')
```

```
data = struct with fields:
     latitude: [1000×1 double]
    longitude: [1000×1 double]
```

Create a HERE HDLM configuration object for the latest version of a catalog that roughly corresponds to North America. If you have not previously set up HERE HDLM credentials, a dialog box prompts you to enter them. Your catalog version might differ from the one shown here.

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

```
configLatest = 
  hereHDLMConfiguration with properties:

           Catalog: 'hrn:here:data::olp-here-had:here-hdlm-protobuf-na-2'
    CatalogVersion: 3320
```

Create a configuration object for the previous version of the catalog.

```
previousVersion = configLatest.CatalogVersion - 1;
config = hereHDLMConfiguration(catalog,previousVersion)
```

```
config = 
  hereHDLMConfiguration with properties:

           Catalog: 'hrn:here:data::olp-here-had:here-hdlm-protobuf-na-2'
    CatalogVersion: 3319
```

Create a HERE HDLM reader using the specified configuration object. The reader is configured to read data from only the previous version of the North America catalog.

```
reader = hereHDLMReader(data.latitude,data.longitude,'Configuration',config);
```

Tips

- To save HERE HDLM credentials between MATLAB sessions, select the **Save my credentials between MATLAB sessions** option in the HERE HD Live Map Credentials dialog box. To manage HERE HDLM credentials, use the `hereHDLMCredentials` function.

Compatibility Considerations

`hereHDLMConfiguration(region)` syntax will be removed
Warns starting in R2020b

In `hereHDLMConfiguration` objects, the syntax for configuring a `hereHDLMReader` object to search catalogs from a specific region, `hereHDLMConfiguration(region)`, will be removed in a future release. Instead, specify the catalog name that corresponds to that region by using the `hereHDLMConfiguration(catalog)` syntax.

Previously, the catalog names for regions such as North America were not available to customers. HERE Technologies now makes these catalog names available through the HERE HD Live Map Marketplace, making the region syntax unnecessary.

Update Code

The table shows a typical usage of the `hereHDLMConfiguration(region)` syntax. It also shows how to update your code using the `hereHDLMConfiguration(catalog)` syntax.

Discouraged Usage	Recommended Replacement
`catalog = hereHDLMConfiguration('North America')`	`catalog = hereHDLMConfiguration('hrn:here:data::olp-h`

See Also

`hereHDLMCredentials` | `hereHDLMReader`

Topics

"Read and Visualize HERE HD Live Map Data"

Introduced in R2019a

inflationCollisionChecker

Collision-checking configuration for costmap based on inflation

Description

The `inflationCollisionChecker` function creates an `InflationCollisionChecker` object, which holds the collision-checking configuration of a vehicle costmap. A vehicle costmap with this configuration inflates the size of obstacles in the vehicle environment. This inflation is based on the specified `InflationCollisionChecker` properties, such as the dimensions of the vehicle and the radius of circles required to enclose the vehicle. For more details, see “Algorithms” on page 4-703. Path planning algorithms, such as `pathPlannerRRT`, use this costmap collision-checking configuration to avoid inflated obstacles and plan collision-free paths through an environment.

Use the `InflationCollisionChecker` object to set the `CollisionChecker` property of your `vehicleCostmap` object. This collision-checking configuration affects the return values of the `checkFree` and `checkOccupied` functions used by `vehicleCostmap`. These values indicate whether a vehicle pose is *free* or *occupied*.

Creation

Syntax

```
ccConfig = inflationCollisionChecker
ccConfig = inflationCollisionChecker(vehicleDims)
ccConfig = inflationCollisionChecker(vehicleDims,numCircles)
ccConfig = inflationCollisionChecker( ___ ,Name,Value)
```

Description

`ccConfig = inflationCollisionChecker` creates an `InflationCollisionChecker` object, `ccConfig`, that holds the collision-checking configuration of a vehicle costmap. This object uses one circle to enclose the vehicle. The dimensions of the vehicle correspond to the values of a default `vehicleDimensions` object.

`ccConfig = inflationCollisionChecker(vehicleDims)` specifies the dimensions of the vehicle, where `vehicleDims` is a `vehicleDimensions` object. The `vehicleDims` input sets the `VehicleDimensions` property of `ccConfig`.

`ccConfig = inflationCollisionChecker(vehicleDims,numCircles)` also specifies the number of circles used to enclose the vehicle. The `numCircles` input sets the `NumCircles` property of `ccConfig`.

`ccConfig = inflationCollisionChecker( ___ ,Name,Value)` sets the `CenterPlacements` and `InflationRadius` properties using name-value pairs and the inputs from any of the preceding syntaxes. Enclose each property name in quotes.

Example: `inflationCollisionChecker('CenterPlacements',[0.2 0.5 0.8],'InflationRadius',1.2)`

Properties

NumCircles — Number of circles enclosing the vehicle
1 (default) | positive integer

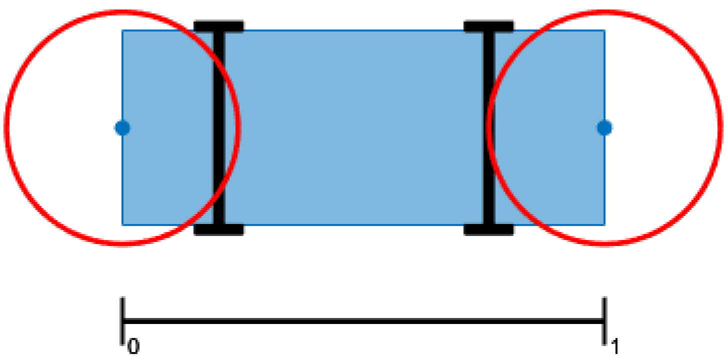
Number of circles used to enclose the vehicle and calculate the inflation radius, specified as a positive integer. Typical values are from 1 to 5.

- For faster but more conservative collision checking, decrease the number of circles. This approach improves performance because the path planning algorithm makes fewer collision checks.
- For slower but more precise collision checking, increase the number of circles. This approach is useful when planning a path around tight corners or through narrow corridors, such as in a parking lot.

CenterPlacements — Normalized placement of circle centers
1-by-NumCircles vector of real values in the range [0, 1]

Normalized placement of circle centers along the longitudinal axis of the vehicle, specified as a 1-by-NumCircles vector of real values in the range [0, 1].

- A value of 0 places a circle center at the rear of the vehicle.
- A value of 1 places a circle center at the front of the vehicle.



Specify CenterPlacements when you want to align the circles with exact positions on the vehicle. If you leave CenterPlacements unspecified, the object computes the center placements so that the circles completely enclose the vehicle. If you change the number of center placements, NumCircles is updated to the number of elements in CenterPlacements.

VehicleDimensions — Vehicle dimensions
vehicleDimensions object

Vehicle dimensions used to compute the inflation radius, specified as a vehicleDimensions object. By default, the InflationCollisionChecker object uses the dimensions of a default vehicleDimensions object. Vehicle dimensions are in world units.

InflationRadius — Inflation radius
nonnegative real number

Inflation radius, specified as a nonnegative real number. By default, the object computes the inflation radius based on the values of NumCircles, CenterPlacements, and VehicleDimensions. For more details, see “Algorithms” on page 4-703.

Object Functions

plot   Plot collision configuration

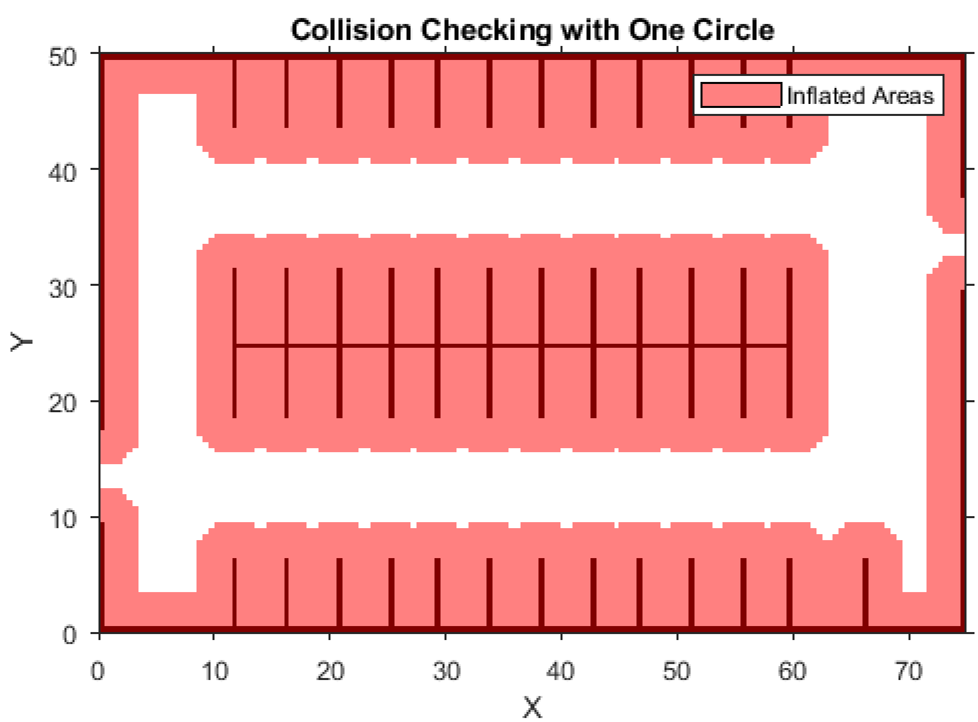
Examples

Plan Path Using Different Collision-Checking Configurations

Plan a vehicle path to a narrow parking spot by using the optimized rapidly exploring random tree (RRT*) algorithm. Try different collision-checking configurations in the costmap used by the RRT* path planner.

Load and display a costmap of a parking lot. The costmap is a `vehicleCostmap` object. By default, `vehicleCostmap` uses a collision-checking configuration that inflates obstacles based on a radius of only one circle enclosing the vehicle. The costmap overinflates the obstacles (the parking spot boundaries).

```
data = load('parkingLotCostmap.mat');
costmap = data.parkingLotCostmap;

figure
plot(costmap)
title('Collision Checking with One Circle')
```



Use `inflationCollisionChecker` to create a new collision-checking configuration for the costmap.

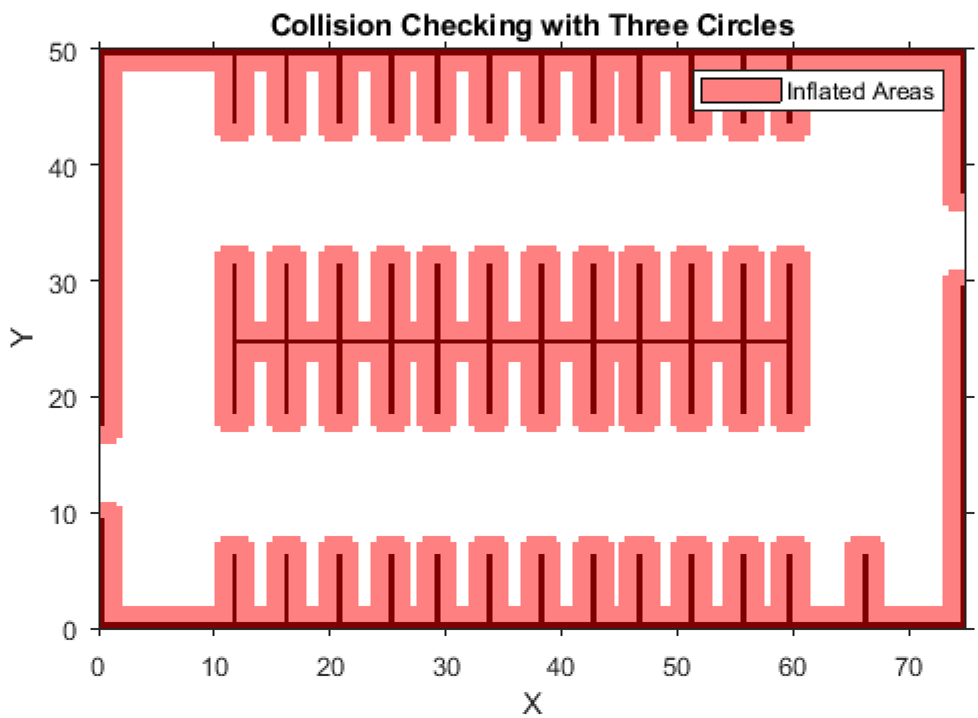
- To decrease inflation of the obstacles, increase the number of circles enclosing the vehicle.
- To specify the dimensions of the vehicle, use a `vehicleDimensions` object.

Specify the collision-checking configuration in the `CollisionChecker` property of the costmap.

```
vehicleDims = vehicleDimensions(4.5,1.7);  % 4.5 m long, 1.7 m wide
numCircles = 3;
ccConfig = inflationCollisionChecker(vehicleDims,numCircles);
costmap.CollisionChecker = ccConfig;
```

Display the costmap with the new collision-checking configuration. The inflated areas are reduced.

```
figure
plot(costmap)
title('Collision Checking with Three Circles')
```



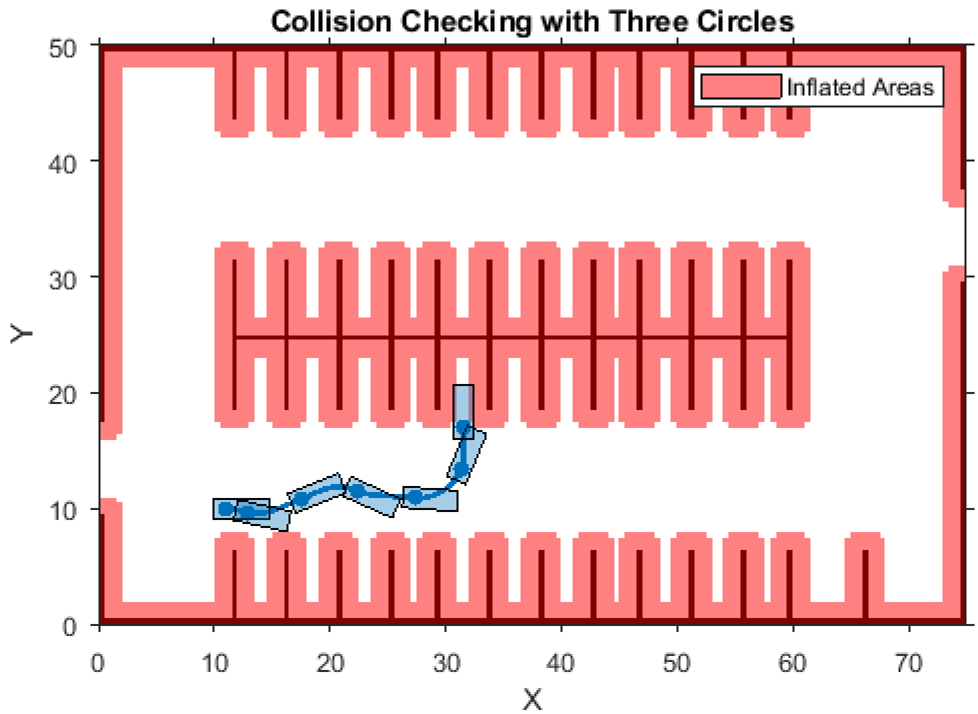
Define a planning problem: a vehicle starts near the left entrance of the parking lot and ends in a parking spot.

```
startPose = [11 10 0];   % [meters, meters, degrees]
goalPose = [31.5 17 90];
```

Use a `pathPlannerRRT` object to plan a path to the parking spot. Plot the planned path.

```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

```
hold on
plot(refPath)
hold off
```



Create Collision-Checking Configuration with Center Placements

Create a collision-checking configuration for a costmap. Manually specify the circle centers so that they fully enclose the vehicle.

Define the dimensions of a vehicle by using a `vehicleDimensions` object.

```
length = 5; % meters
width = 2; % meters
vehicleDims = vehicleDimensions(length,width);
```

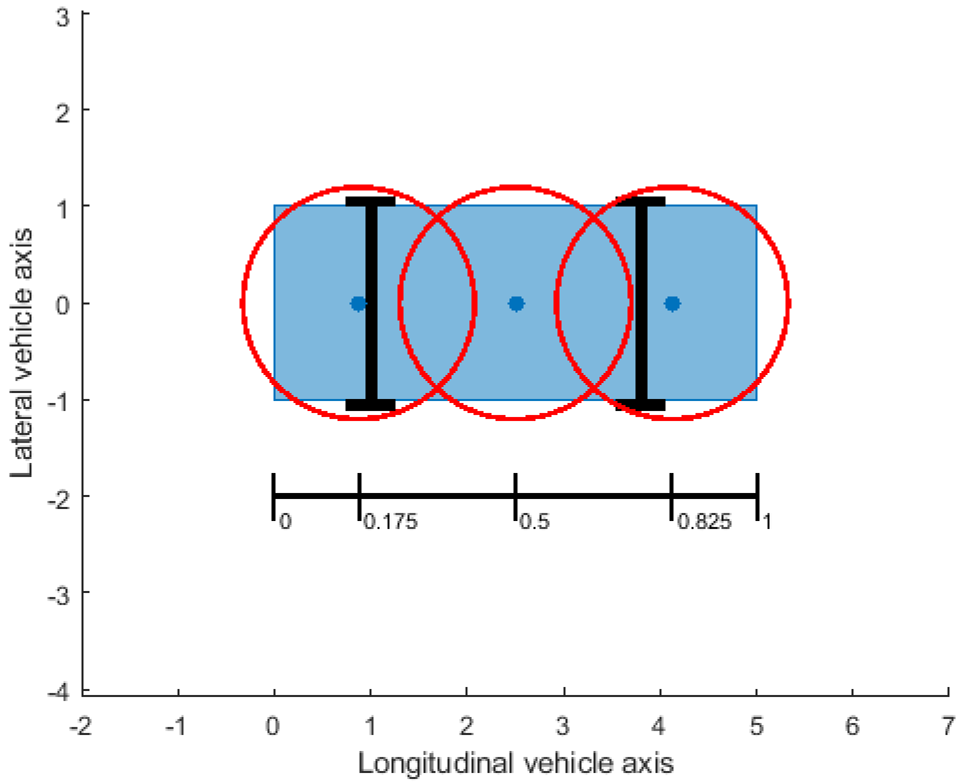
Define three circle centers and the inflation radius to use for collision checking. Place one center at the vehicle's midpoint. Offset the other two centers by an equal amount on either end of the vehicle.

```
distFromSide = 0.175;
centerPlacements = [distFromSide 0.5 1-distFromSide];
inflationRadius = 1.2;
```

Create and display the collision-checking configuration.

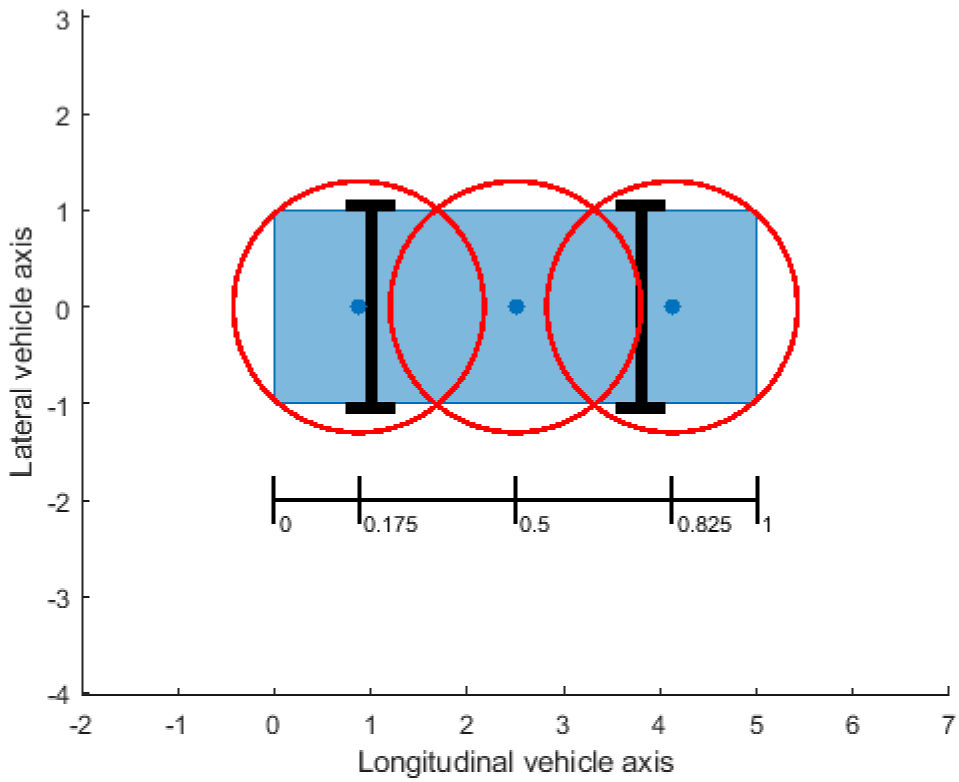
```
ccConfig = inflationCollisionChecker(vehicleDims, ...
    'CenterPlacements',centerPlacements,'InflationRadius',inflationRadius);
```

```
figure
plot(ccConfig)
```



In this configuration, the corners of the vehicle are not enclosed within the circles. To fully enclose the vehicle, increase the inflation radius. Display the updated configuration.

```
ccConfig.InflationRadius = 1.3;
plot(ccConfig)
```



Use this collision-checking configuration to create a 10-by-20 meter costmap.

```
costmap = vehicleCostmap(10,20,0.1,'CollisionChecker',ccConfig);
```

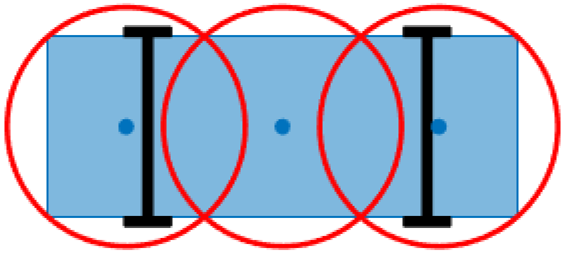
Tips

- To visually verify that the circles completely enclose the vehicle, use the `plot` function. If the circles do not completely enclose the vehicle, some of the free poses returned by `checkFree` (or unoccupied poses returned by `checkOccupied`) might actually be in collision.

Algorithms

The `InflationRadius` property of `InflationCollisionChecker` determines the amount, in world units, by which to inflate obstacles. By default, `InflationRadius` is equal to the radius of the smallest set of overlapping circles required to completely enclose the vehicle, as determined by the following properties:

- `NumCircles` — Number of circles used to enclose the vehicle
- `CenterPlacements` — Placements of the circle centers along the longitudinal axis of the vehicle
- `VehicleDimensions` — Dimensions of the vehicle



For more details about how this collision-checking configuration defines inflated areas in a costmap, see the “Algorithms” on page 4-913 section of `vehicleCostmap`.

References

[1] Ziegler, J., and C. Stiller. "Fast Collision Checking for Intelligent Vehicle Motion Planning." *IEEE Intelligent Vehicle Symposium*. June 21–24, 2010.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All inputs to `inflationCollisionChecker` must be compile-time constants.

See Also

Objects
`pathPlannerRRT` | `vehicleCostmap` | `vehicleDimensions`

Topics
“Automated Parking Valet”

Introduced in R2018b

plot

Plot collision configuration

Syntax

```
plot(ccConfig)
plot(ccConfig,Name,Value)
```

Description

`plot(ccConfig)` plots the collision-checking configuration of an `InflationCollisionChecker` object. Use `plot` to visually verify that the circles in the configuration fully enclose the vehicle.

`plot(ccConfig,Name,Value)` specifies options using one or more `Name,Value` pair arguments. For example, `plot(ccConfig,'Ruler','Off')` turns off the ruler that indicates the locations of the circle centers.

Examples

Create Collision-Checking Configuration with Center Placements

Create a collision-checking configuration for a costmap. Manually specify the circle centers so that they fully enclose the vehicle.

Define the dimensions of a vehicle by using a `vehicleDimensions` object.

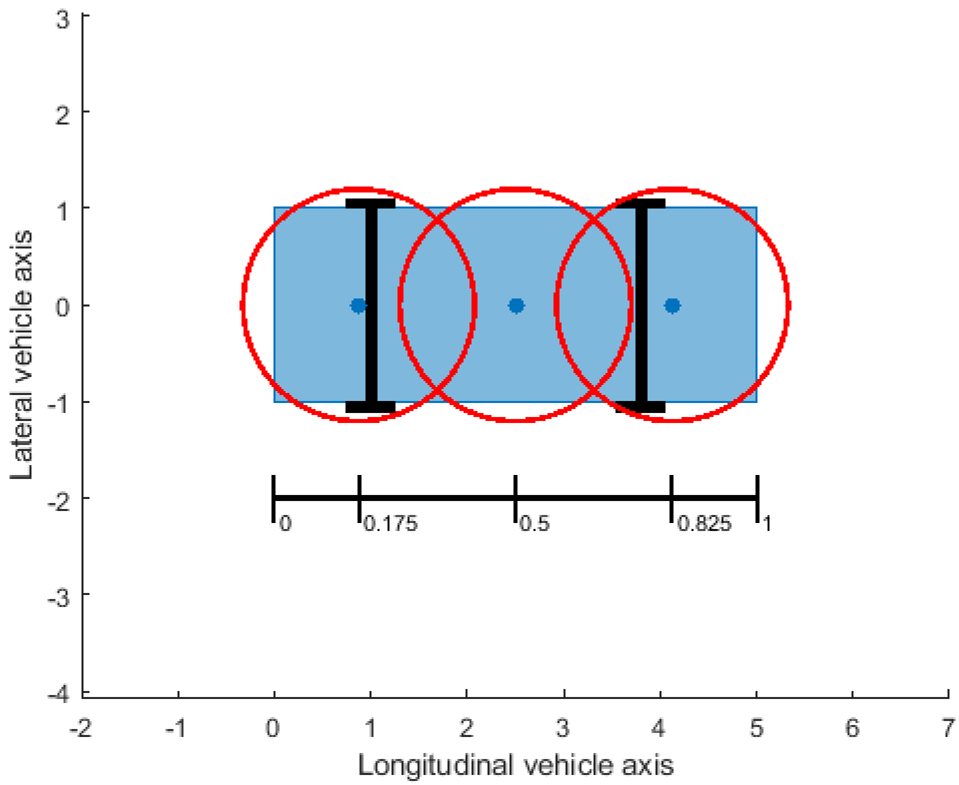
```
length = 5; % meters
width = 2; % meters
vehicleDims = vehicleDimensions(length,width);
```

Define three circle centers and the inflation radius to use for collision checking. Place one center at the vehicle's midpoint. Offset the other two centers by an equal amount on either end of the vehicle.

```
distFromSide = 0.175;
centerPlacements = [distFromSide 0.5 1-distFromSide];
inflationRadius = 1.2;
```

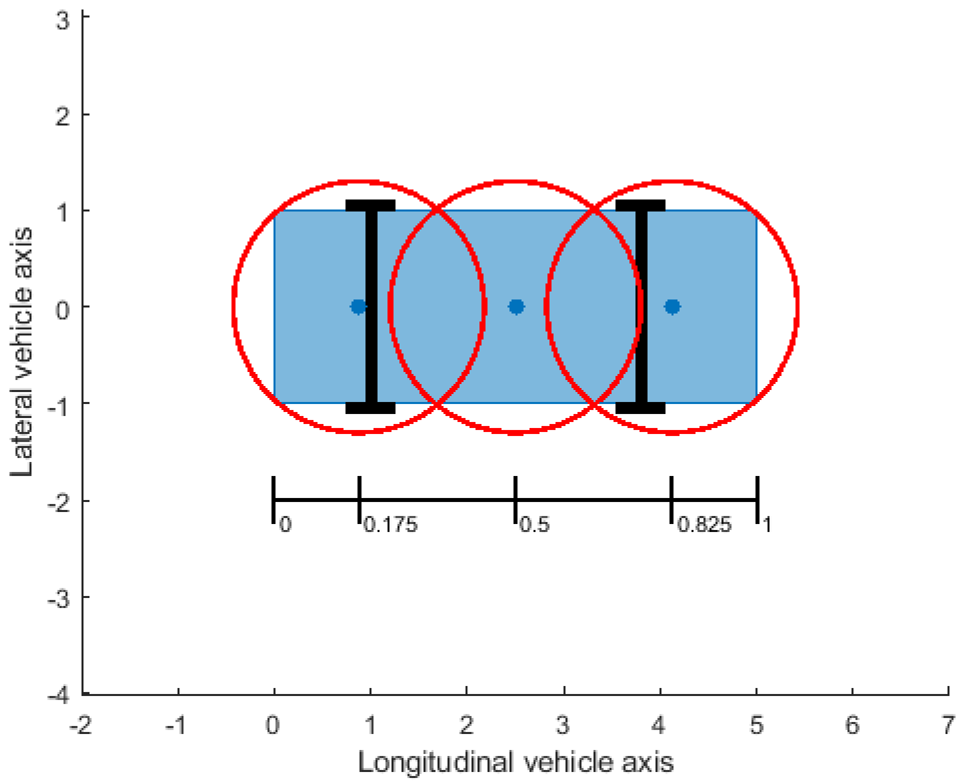
Create and display the collision-checking configuration.

```
ccConfig = inflationCollisionChecker(vehicleDims, ...
    'CenterPlacements',centerPlacements,'InflationRadius',inflationRadius);

figure
plot(ccConfig)
```



In this configuration, the corners of the vehicle are not enclosed within the circles. To fully enclose the vehicle, increase the inflation radius. Display the updated configuration.

```
ccConfig.InflationRadius = 1.3;
plot(ccConfig)
```



Use this collision-checking configuration to create a 10-by-20 meter costmap.

```
costmap = vehicleCostmap(10,20,0.1,'CollisionChecker',ccConfig);
```

Input Arguments

ccConfig — Collision-checking configuration
InflationCollisionChecker object

Collision-checking configuration, specified as an InflationCollisionChecker object. To create a collision-checking configuration, use the inflationCollisionChecker function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: plot(ccConfig,'Parent',ax) plots the collision configuration in axes ax.

Parent — Axes on which to plot collision configuration
Axes object

Axes on which to plot the collision configuration, specified as the comma-separated pair consisting of 'Parent' and an Axes object. To create an Axes object, use the axes function.

To plot the collision configuration in a new figure, leave `'Parent'` unspecified.

`Ruler` — Display ruler
`'on'` (default) | `'off'`

Display the ruler that shows the locations of the circle centers, specified as the comma-separated pair consisting of `'Ruler'` and `'on'` or `'off'`.

See Also
`inflationCollisionChecker`

Introduced in R2018b

parabolicLaneBoundary

Parabolic lane boundary model

Description

The `parabolicLaneBoundary` object contains information about a parabolic lane boundary model.

Creation

To generate parabolic lane boundary models that fit a set of boundary points and an approximate width, use the `findParabolicLaneBoundaries` function. If you already know your parabolic parameters, create lane boundary models by using the `parabolicLaneBoundary` function (described here).

Syntax

`boundaries = parabolicLaneBoundary(parabolicParameters)`

Description

`boundaries = parabolicLaneBoundary(parabolicParameters)` creates an array of parabolic lane boundary models from an array of `[A B C]` parameters for the parabolic equation $y = Ax^2 + Bx + C$. Points within the lane boundary models are in world coordinates.

Input Arguments

`parabolicParameters` — Coefficients for parabolic models
`[A B C]` real-valued vector | matrix of `[A B C]` values

Coefficients for parabolic models of the form $y = Ax^2 + Bx + C$, specified as an `[A B C]` real-valued vector or as a matrix of `[A B C]` values. Each row of `parabolicParameters` describes a separate parabolic lane boundary model.

Properties

`Parameters` — Coefficients for parabolic model
`[A B C]` real-valued vector

Coefficients for a parabolic model of the form $y = Ax^2 + Bx + C$, specified as an `[A B C]` real-valued vector.

`BoundaryType` — Type of boundary
`LaneBoundaryType`

Type of boundary, specified as a `LaneBoundaryType` of supported lane boundaries. The supported lane boundary types are:

- `Unmarked`
- `Solid`
- `Dashed`
- `BottsDots`
- `DoubleSolid`

Specify a lane boundary type as `LaneBoundaryType.`*`BoundaryType`*. For example:

```
LaneBoundaryType.BottsDots
```

`Strength` — Strength of boundary model
real scalar

Strength of the boundary model, specified as a real scalar. `Strength` is the ratio of the number of unique *x*-axis locations on the boundary to the length of the boundary specified by the `XExtent` property. A solid line without any breaks has a higher strength than a dotted line that has breaks along the full length of the boundary.

`XExtent` — Length of boundary along *x*-axis
`[minX maxX]` real-valued vector

Length of the boundary along the *x*-axis, specified as a `[minX maxX]` real-valued vector that describes the minimum and maximum *x*-axis locations.

Object Functions

computeBoundaryModel   Obtain y-coordinates of lane boundaries given x-coordinates

Examples

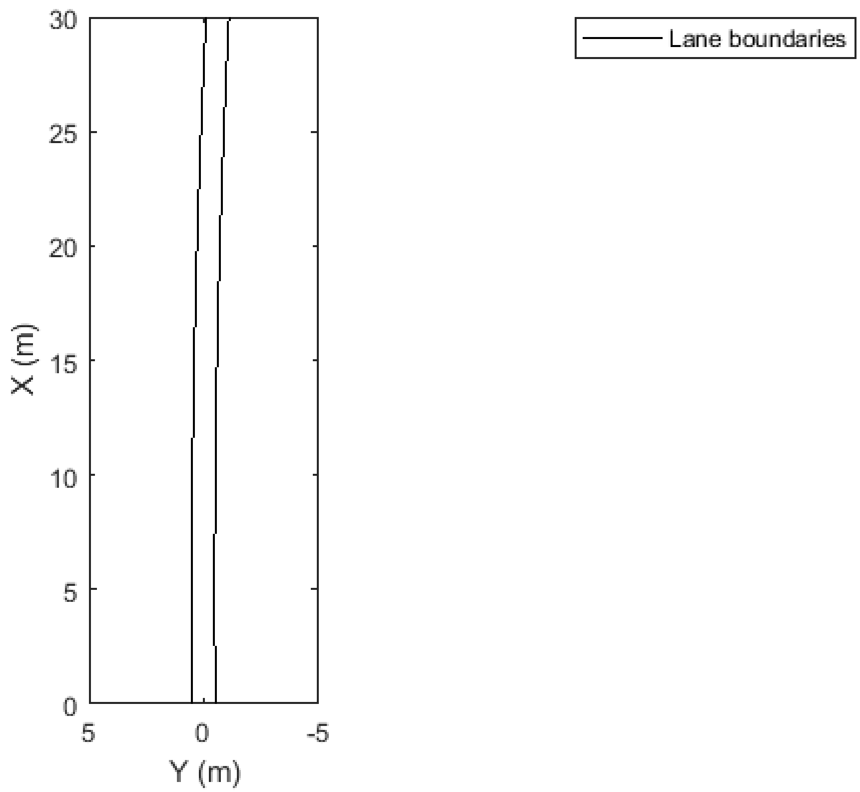
Create Parabolic Lane Boundaries

Create left-lane and right-lane parabolic boundary models.

```
llane = parabolicLaneBoundary([-0.001 0.01  0.5]);
rlane = parabolicLaneBoundary([-0.001 0.01 -0.5]);
```

Create a bird's-eye plot and lane boundary plotter. Plot the lane boundaries.

```
bep = birdsEyePlot('XLimits',[0 30],'YLimits',[-5 5]);
lbPlotter = laneBoundaryPlotter(bep,'DisplayName','Lane boundaries');
plotLaneBoundary(lbPlotter, [llane rlane]);
```



Find Parabolic Lane Boundaries in Bird's-Eye-View Image

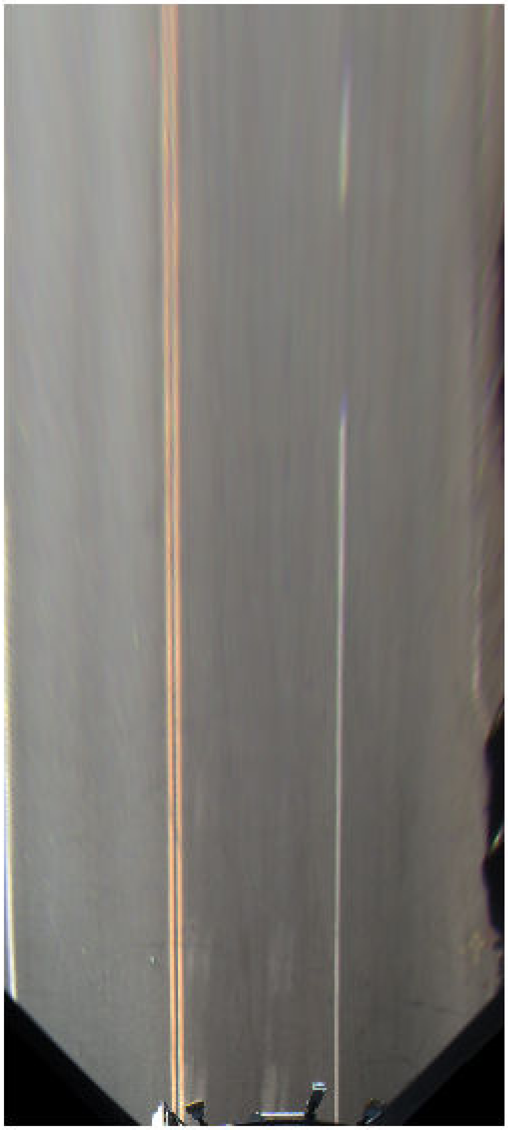
Find lanes in an image by using parabolic lane boundary models. Overlay the identified lanes on the original image and on a bird's-eye-view transformation of the image.

Load an image of a road with lanes. The image was obtained from a camera sensor mounted on the front of a vehicle.

```
I = imread('road.png');
```

Transform the image into a bird's-eye-view image by using a preconfigured sensor object. This object models the sensor that captured the original image.

```
bevSensor = load('birdsEyeConfig');
birdsEyeImage = transformImage(bevSensor.birdsEyeConfig,I);
imshow(birdsEyeImage)
```

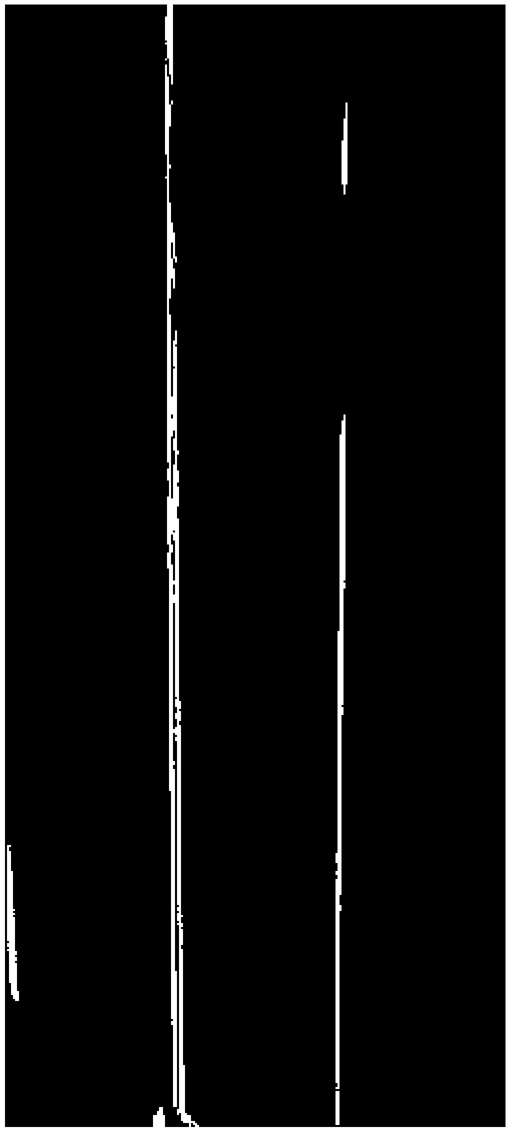


Set the approximate lane marker width in world units (meters).

```
approxBoundaryWidth = 0.25;
```

Detect lane features and display them as a black-and-white image.

```
birdsEyeBW = segmentLaneMarkerRidge(rgb2gray(birdsEyeImage), ...
    bevSensor.birdsEyeConfig,approxBoundaryWidth);
imshow(birdsEyeBW)
```



Obtain lane candidate points in world coordinates.

```
[imageX,imageY] = find(birdsEyeBW);
xyBoundaryPoints = imageToVehicle(bevSensor.birdsEyeConfig,[imageY,imageX]);
```

Find lane boundaries in the image by using the `findParabolicLaneBoundaries` function. By default, the function returns a maximum of two lane boundaries. The boundaries are stored in an array of `parabolicLaneBoundary` objects.

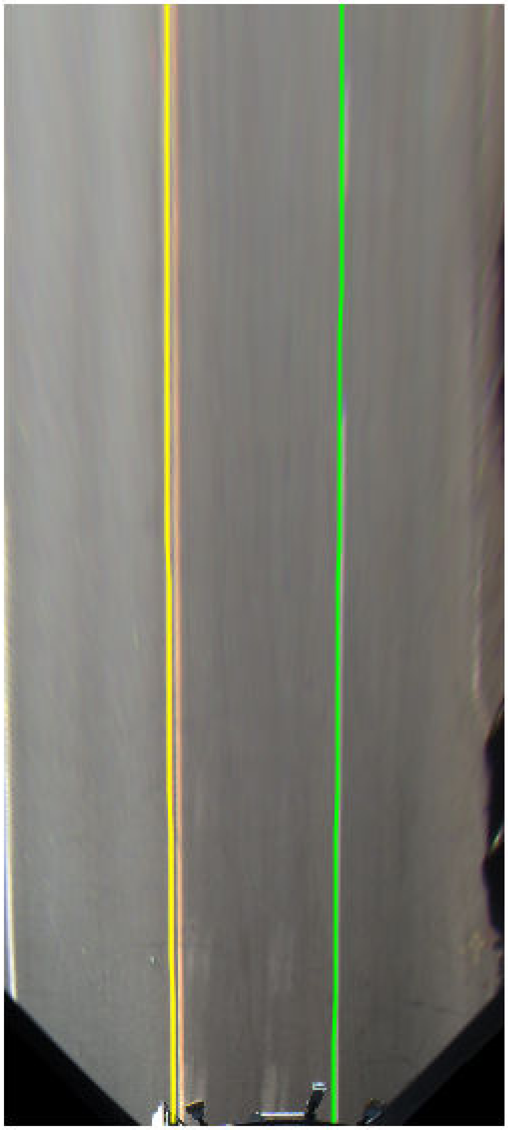
```
boundaries = findParabolicLaneBoundaries(xyBoundaryPoints,approxBoundaryWidth);
```

Use `insertLaneBoundary` to overlay the lanes on the original image. The `XPoints` vector represents the lane points, in meters, that are within range of the ego vehicle's sensor. Specify the lanes in different colors. By default, lanes are yellow.

```
XPoints = 3:30;

figure
sensor = bevSensor.birdsEyeConfig.Sensor;
lanesI = insertLaneBoundary(I,boundaries(1),sensor,XPoints);
lanesI = insertLaneBoundary(lanesI,boundaries(2),sensor,XPoints,'Color','green');
imshow(lanesI)
```

View the lanes in the bird's-eye-view image.

```
figure
BEconfig = bevSensor.birdsEyeConfig;
lanesBEI = insertLaneBoundary(birdsEyeImage,boundaries(1),BEconfig,XPoints);
lanesBEI = insertLaneBoundary(lanesBEI,boundaries(2),BEconfig,XPoints,'Color','green');
imshow(lanesBEI)
```



Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Apps
Ground Truth Labeler

Objects
`cubicLaneBoundary`

Functions
`evaluateLaneBoundaries` | `findParabolicLaneBoundaries` | `insertLaneBoundary`

Introduced in R2017a

cubicLaneBoundary

Cubic lane boundary model

Description

The `cubicLaneBoundary` object contains information about a cubic lane boundary model.

Creation

To generate cubic lane boundary models that fit a set of boundary points and an approximate width, use the `findCubicLaneBoundaries` function. If you already know your cubic parameters, create lane boundary models by using the `cubicLaneBoundary` function (described here).

Syntax

`boundaries = cubicLaneBoundary(cubicParameters)`

Description

`boundaries = cubicLaneBoundary(cubicParameters)` creates an array of cubic lane boundary models from an array of `[A B C D]` parameters for the cubic equation $y = Ax^3 + Bx^2 + Cx + D$. Points within the lane boundary models are in world coordinates.

Input Arguments

`cubicParameters` — Parameters for cubic models
`[A B C D]` real-valued vector | matrix of `[A B C D]` values

Parameters for cubic models of the form $y = Ax^3 + Bx^2 + Cx + D$, specified as an `[A B C D]` real-valued vector or as a matrix of `[A B C D]` values. Each row of `cubicParameters` describes a separate cubic lane boundary model.

Properties

`Parameters` — Coefficients for cubic model
`[A B C D]` real-valued vector

Coefficients for a cubic model of the form $y = Ax^3 + Bx^2 + Cx + D$, specified as an `[A B C D]` real-valued vector.

`BoundaryType` — Type of boundary
`LaneBoundaryType`

Type of boundary, specified as a `LaneBoundaryType` of supported lane boundaries. The supported lane boundary types are:

- `Unmarked`

- `Solid`
- `Dashed`
- `BottsDots`
- `DoubleSolid`

Specify a lane boundary type as `LaneBoundaryType.`*`BoundaryType`*. For example:

```
LaneBoundaryType.BottsDots
```

`Strength` — Strength of boundary model
real scalar

Strength of the boundary model, specified as a real scalar. `Strength` is the ratio of the number of unique *x*-axis locations on the boundary to the length of the boundary specified by the `XExtent` property. A solid line without any breaks has a higher strength than a dotted line that has breaks along the full length of the boundary.

`XExtent` — Length of boundary along *x*-axis
`[minX maxX]` real-valued vector

Length of the boundary along the *x*-axis, specified as a `[minX maxX]` real-valued vector that describes the minimum and maximum *x*-axis locations.

Object Functions

computeBoundaryModel   Obtain y-coordinates of lane boundaries given x-coordinates

Examples

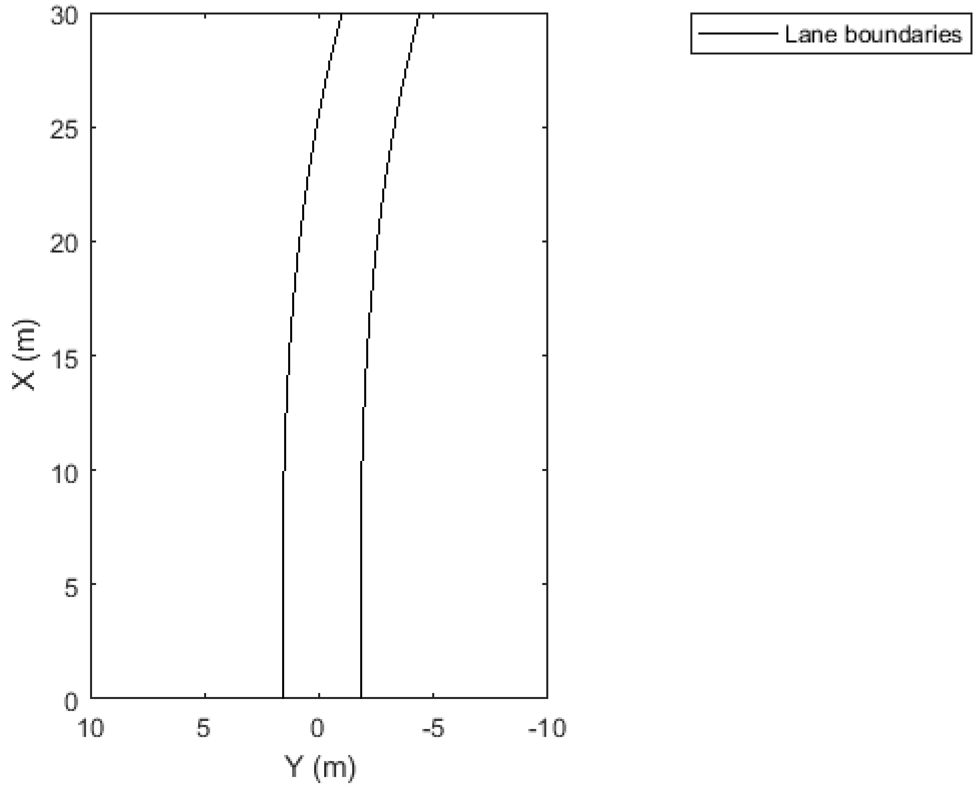
Create Cubic Lane Boundaries

Create left-lane and right-lane cubic boundary models.

```
llane = cubicLaneBoundary([-0.0001 0.0 0.003  1.6]);
rlane = cubicLaneBoundary([-0.0001 0.0 0.003 -1.8]);
```

Create a bird's-eye plot and lane boundary plotter. Plot the lane boundaries.

```
bep = birdsEyePlot('XLimits',[0 30],'YLimits',[-10 10]);
lbPlotter = laneBoundaryPlotter(bep,'DisplayName','Lane boundaries');

plotLaneBoundary(lbPlotter, [llane rlane]);
```



Find Cubic Lane Boundaries in Bird's-Eye-View Image

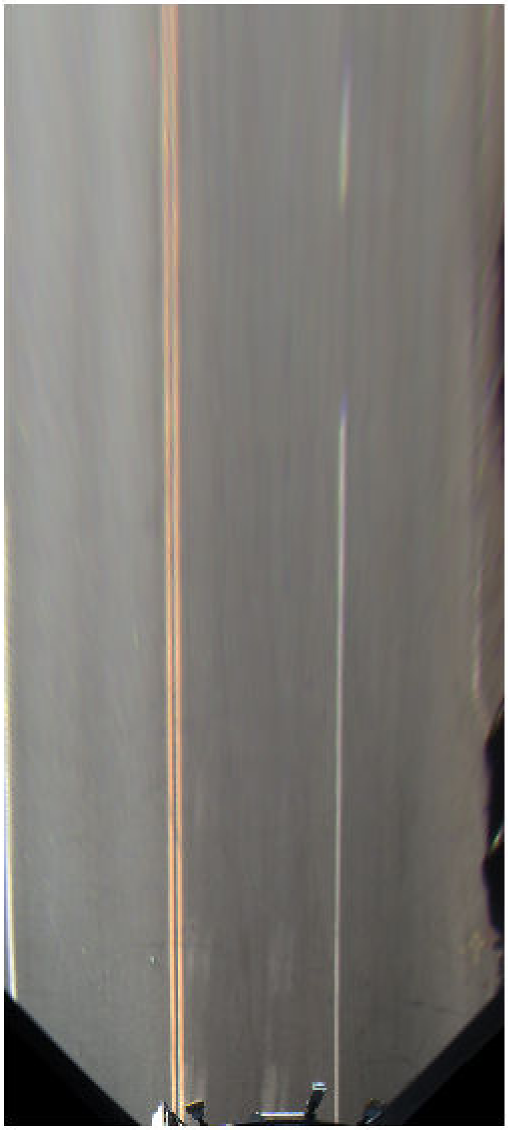
Find lanes in an image by using cubic lane boundary models. Overlay the identified lanes on the original image and on a bird's-eye-view transformation of the image.

Load an image of a road with lanes. The image was obtained from a camera sensor mounted on the front of a vehicle.

```
I = imread('road.png');
```

Transform the image into a bird's-eye-view image by using a preconfigured sensor object. This object models the sensor that captured the original image.

```
bevSensor = load('birdsEyeConfig');
birdsEyeImage = transformImage(bevSensor.birdsEyeConfig,I);
imshow(birdsEyeImage)
```

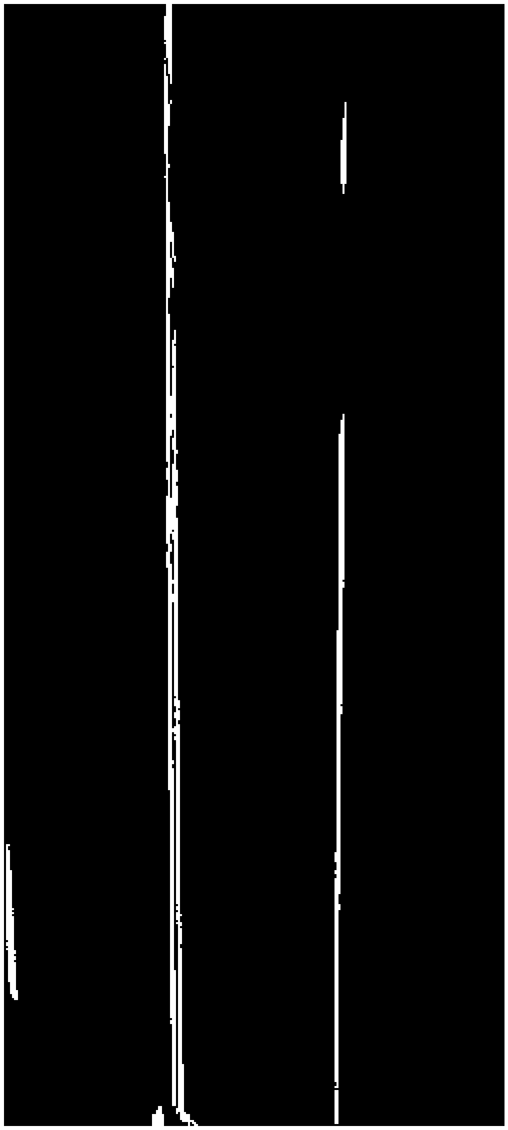


Set the approximate lane marker width in world units (meters).

```
approxBoundaryWidth = 0.25;
```

Detect lane features and display them as a black-and-white image.

```
birdsEyeBW = segmentLaneMarkerRidge(rgb2gray(birdsEyeImage), ...
    bevSensor.birdsEyeConfig,approxBoundaryWidth);
imshow(birdsEyeBW)
```



Obtain lane candidate points in world coordinates.

```
[imageX,imageY]  = find(birdsEyeBW);
xyBoundaryPoints = imageToVehicle(bevSensor.birdsEyeConfig,[imageY,imageX]);
```

Find lane boundaries in the image by using the `findCubicLaneBoundaries` function. By default, the function returns a maximum of two lane boundaries. The boundaries are stored in an array of `cubicLaneBoundary` objects.

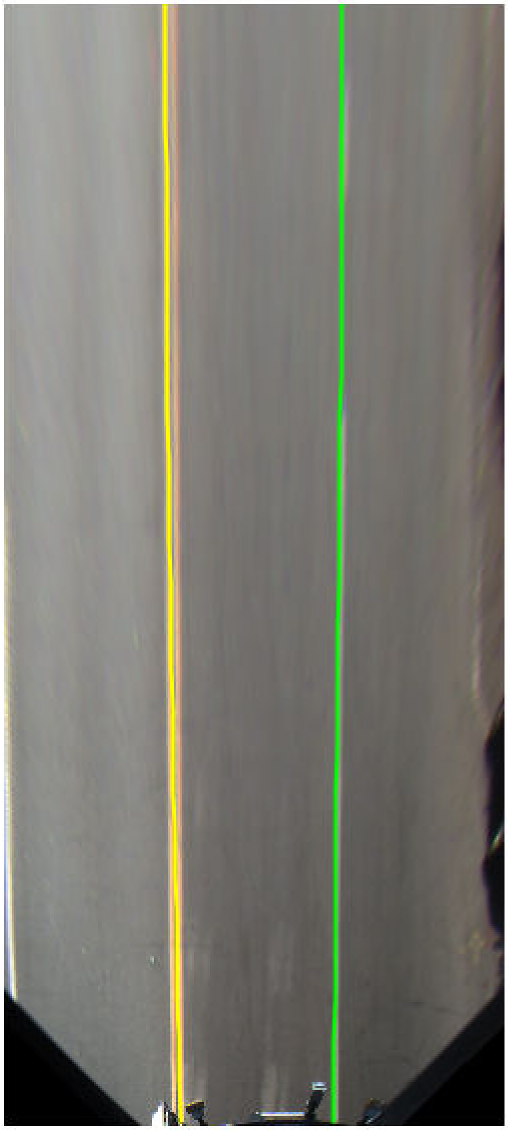
```
boundaries = findCubicLaneBoundaries(xyBoundaryPoints,approxBoundaryWidth);
```

Use `insertLaneBoundary` to overlay the lanes on the original image. The `XPoints` vector represents the lane points, in meters, that are within range of the ego vehicle's sensor. Specify the lanes in different colors. By default, lanes are yellow.

```
XPoints = 3:30;

figure
sensor = bevSensor.birdsEyeConfig.Sensor;
lanesI = insertLaneBoundary(I,boundaries(1),sensor,XPoints);
lanesI = insertLaneBoundary(lanesI,boundaries(2),sensor,XPoints,'Color','green');
imshow(lanesI)
```

View the lanes in the bird's-eye-view image.

```
figure
BEconfig = bevSensor.birdsEyeConfig;
lanesBEI = insertLaneBoundary(birdsEyeImage,boundaries(1),BEconfig,XPoints);
lanesBEI = insertLaneBoundary(lanesBEI,boundaries(2),BEconfig,XPoints,'Color','green');
imshow(lanesBEI)
```



Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Apps
Ground Truth Labeler

Objects
`parabolicLaneBoundary`

Functions
`evaluateLaneBoundaries` | `findCubicLaneBoundaries` | `insertLaneBoundary`

Introduced in R2018a

computeBoundaryModel

Obtain *y*-coordinates of lane boundaries given *x*-coordinates

Syntax

```
yWorld = computeBoundaryModel(boundaries,xWorld)
```

Description

`yWorld = computeBoundaryModel(boundaries,xWorld)` computes the *y*-axis world coordinates of lane boundary models at the specified *x*-axis world coordinates.

- If `boundaries` is a single lane boundary model, then `yWorld` is a vector of coordinates corresponding to the coordinates in `xWorld`.
- If `boundaries` is an array of lane boundary models, then `yWorld` is a matrix. Each row or column of `yWorld` corresponds to a lane boundary model computed at the *x*-coordinates in row or column vector `xWorld`.

Examples

Compute Lane Boundary

Create a `parabolicLaneBoundary` object to model a lane boundary. Compute the positions of the lane along a set of *x*-axis locations.

Specify the parabolic parameters and create a lane boundary model.

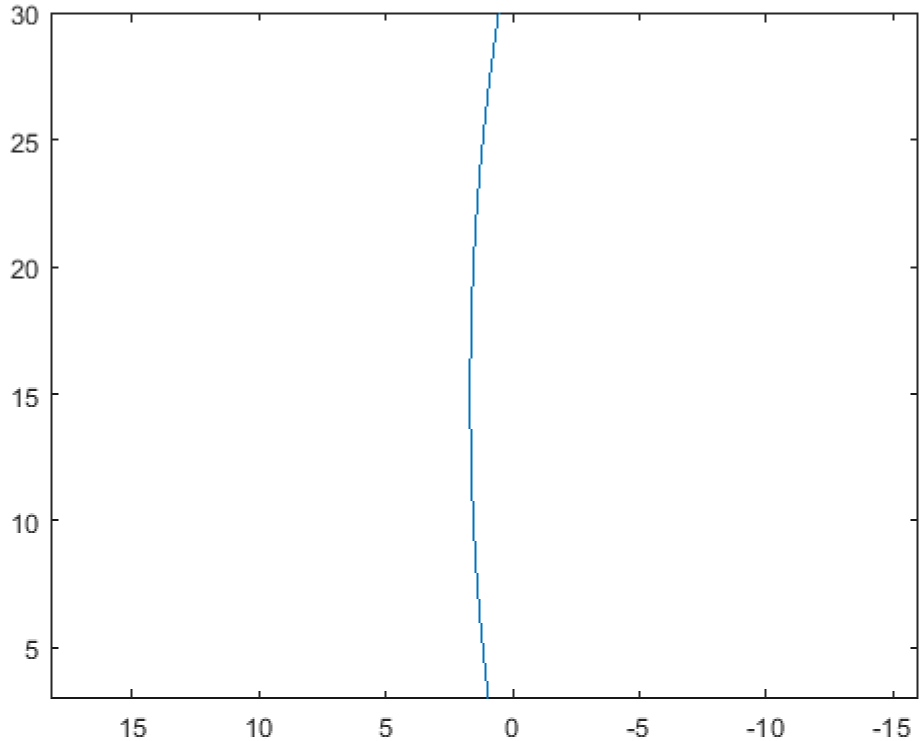
```
parabolicParams = [-0.005 0.15 0.55];
lb = parabolicLaneBoundary(parabolicParams);
```

Compute the *y*-axis locations for given *x*-axis locations within the range of a camera sensor mounted to the front of a vehicle.

```
xWorld = 3:30; % in meters
yWorld = computeBoundaryModel(lb,xWorld);
```

Plot the lane boundary points. To fit the coordinate system, flip the axis order and change the *x*-direction.

```
plot(yWorld,xWorld)
axis equal
set(gca,'XDir','reverse')
```



Plot Path of Ego Vehicle

Create a 3-meter-wide lane.

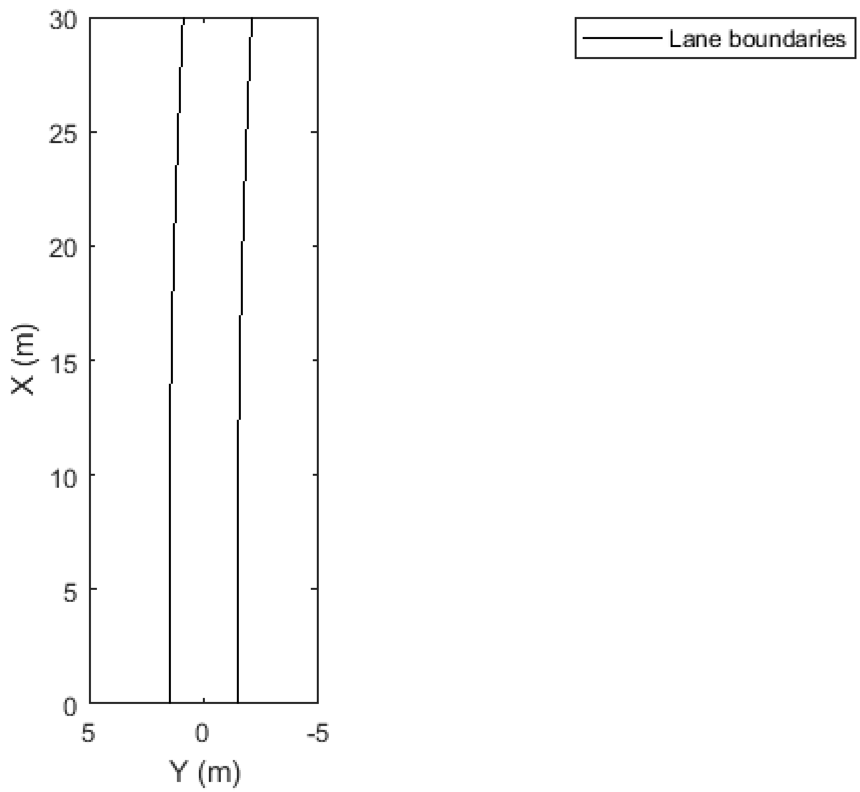
```
lb = parabolicLaneBoundary([-0.001,0.01,1.5]);
rb = parabolicLaneBoundary([-0.001,0.01,-1.5]);
```

Compute the lane boundary model manually from 0 to 30 meters along the *x*-axis.

```
xWorld = (0:30)';
yLeft = computeBoundaryModel(lb,xWorld);
yRight = computeBoundaryModel(rb,xWorld);
```

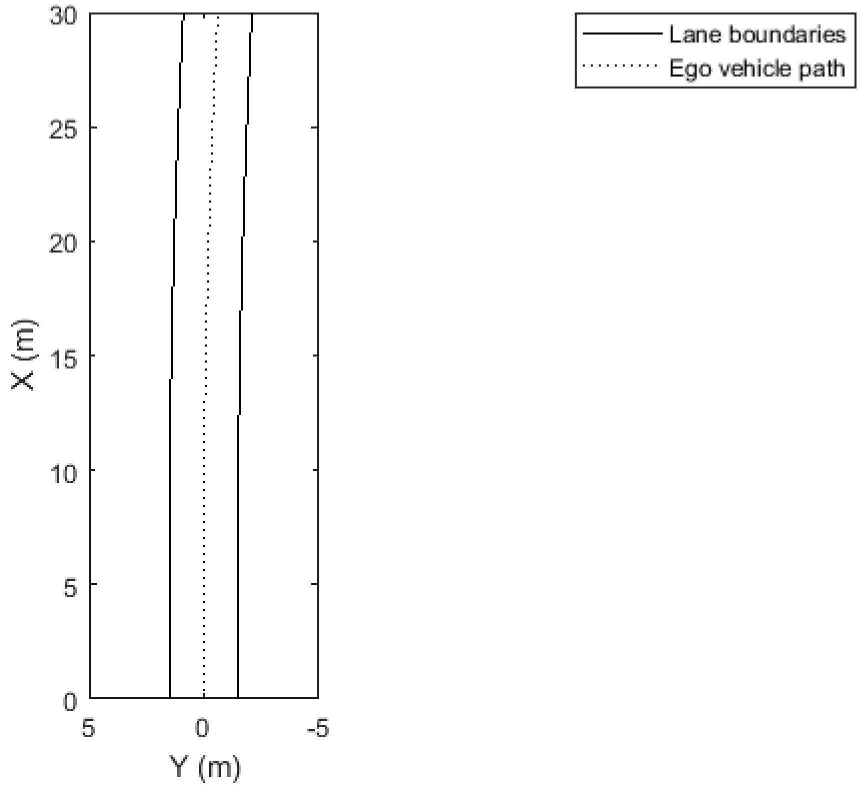
Create a bird's-eye plot and lane boundary plotter. Display the lane information on the bird's-eye plot.

```
bep = birdsEyePlot('XLimits',[0 30],'YLimits',[-5 5]);
lanePlotter = laneBoundaryPlotter(bep,'DisplayName','Lane boundaries');
plotLaneBoundary(lanePlotter,{[xWorld,yLeft],[xWorld,yRight]});
```



Create a path plotter. Create and display the path of an ego vehicle that travels through the center of the lane.

```
yCenter = (yLeft + yRight)/2;
egoPathPlotter = pathPlotter(bep,'DisplayName','Ego vehicle path');
plotPath(egoPathPlotter,{[xWorld,yCenter]});
```



Find Candidate Ego Lane Boundaries

Find candidate ego lane boundaries from an array of lane boundaries.

Create an array of cubic lane boundaries.

```
lbs = [cubicLaneBoundary([-0.0001, 0.0, 0.003,  1.6]), ...
       cubicLaneBoundary([-0.0001, 0.0, 0.003,  4.6]), ...
       cubicLaneBoundary([-0.0001, 0.0, 0.003, -1.6]), ...
       cubicLaneBoundary([-0.0001, 0.0, 0.003, -4.6])];
```

For each lane boundary, compute the *y*-axis location at which the *x*-coordinate is 0.

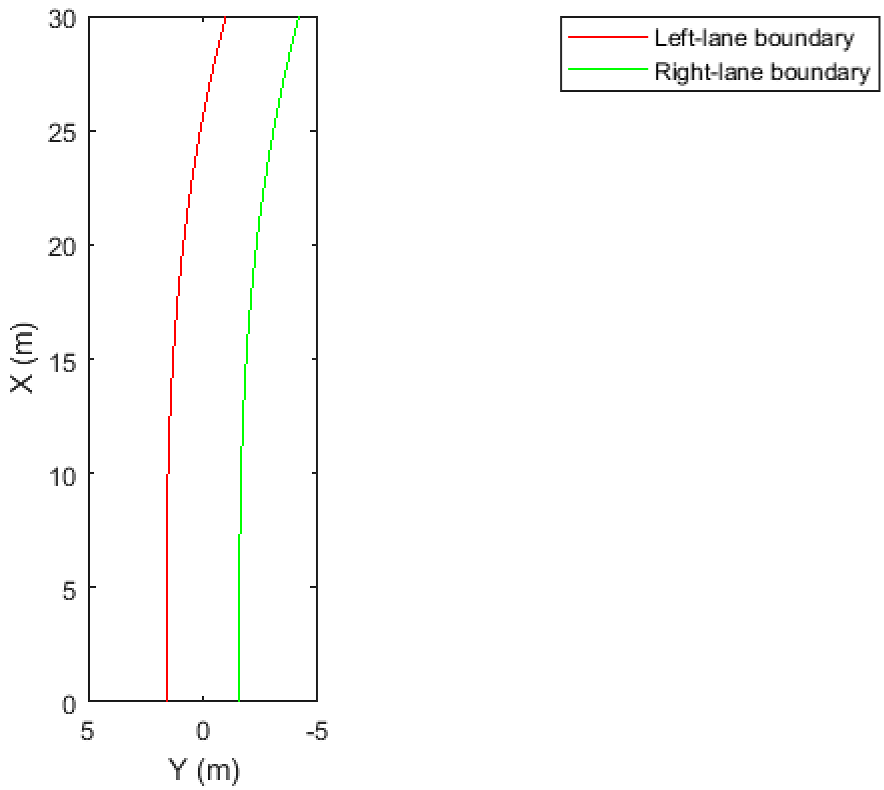
```
xWorld = 0; % meters
yWorld = computeBoundaryModel(lbs,0);
```

Use the computed locations to find the ego lane boundaries that best meet the criteria.

```
leftEgoBoundaryIndex = find(yWorld == min(yWorld(yWorld>0)));
rightEgoBoundaryIndex = find(yWorld == max(yWorld(yWorld<=0)));
leftEgoBoundary = lbs(leftEgoBoundaryIndex);
rightEgoBoundary = lbs(rightEgoBoundaryIndex);
```

Plot the boundaries using a bird's-eye plot and lane boundary plotter.

```
bep = birdsEyePlot('XLimits',[0 30],'YLimits',[-5 5]);
lbPlotter = laneBoundaryPlotter(bep,'DisplayName','Left-lane boundary','Color','r');
rbPlotter = laneBoundaryPlotter(bep,'DisplayName','Right-lane boundary','Color','g');
plotLaneBoundary(lbPlotter,leftEgoBoundary)
plotLaneBoundary(rbPlotter,rightEgoBoundary)
```



Input Arguments

boundaries — Lane boundary models
lane boundary object | array of lane boundary objects

Lane boundary models containing the parameters used to compute the *y*-axis coordinates, specified as a lane boundary object or an array of lane boundary objects. Valid objects are `parabolicLaneBoundary` and `cubicLaneBoundary`.

`xWorld` — *x*-axis locations of boundaries
real scalar | real-valued vector

x-axis locations of the boundaries in world coordinates, specified as a real scalar or real-valued vector.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
cubicLaneBoundary | parabolicLaneBoundary

Functions
insertLaneBoundary

Introduced in R2017a

monoCamera

Configure monocular camera sensor

Description

The `monoCamera` object holds information about the configuration of a monocular camera sensor. Configuration information includes the camera intrinsics, camera extrinsics such as its orientation (as described by pitch, yaw, and roll), and the camera location within the vehicle. To estimate the intrinsic and extrinsic camera parameters, see “Calibrate a Monocular Camera”.

For images captured by the camera, you can use the `imageToVehicle` and `vehicleToImage` functions to transform point locations between image coordinates and vehicle coordinates. These functions apply projective transformations (homography), which enable you to estimate distances from a camera mounted on the vehicle to locations on a flat road surface.

Creation

Syntax

```
sensor = monoCamera(intrinsics,height)
sensor = monoCamera(intrinsics,height,Name,Value)
```

Description

`sensor = monoCamera(intrinsics,height)` creates a `monoCamera` object that contains the configuration of a monocular camera sensor, given the intrinsic parameters of the camera and the height of the camera above the ground. `intrinsics` and `height` set the `Intrinsics` and `Height` properties of the camera.

`sensor = monoCamera(intrinsics,height,Name,Value)` sets properties on page 4-731 using one or more name-value pairs. For example, `monoCamera(intrinsics,1.5,'Pitch',1)` creates a monocular camera sensor that is 1.5 meters above the ground and has a 1-degree pitch toward the ground. Enclose each property name in quotes.

Properties

`Intrinsics` — Intrinsic camera parameters
`cameraIntrinsics` object | `cameraParameters` object

Intrinsic camera parameters, specified as either a `cameraIntrinsics` or `cameraParameters` object. The intrinsic camera parameters include the focal length and optical center of the camera, and the size of the image produced by the camera.

You can set this property when you create the object. After you create the object, this property is read-only.

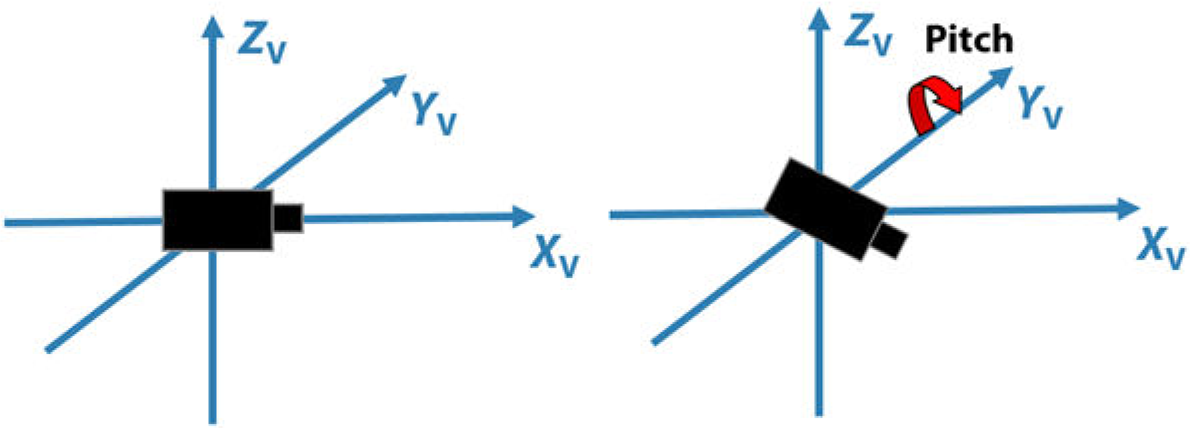
`Height` — Height from road surface to camera sensor
real scalar

Height from the road surface to the camera sensor, specified as a real scalar. The height is the perpendicular distance from the ground to the focal point of the camera. Specify the height in world units, such as meters. To estimate this value, use the `estimateMonoCameraParameters` function.

Pitch — Pitch angle
real scalar

Pitch angle between the horizontal plane of the vehicle and the optical axis of the camera, specified as a real scalar in degrees. To estimate this value, use the `estimateMonoCameraParameters` function.

`Pitch` uses the ISO convention for rotation, with a clockwise positive angle direction when looking in the positive direction of the vehicle's Y_V axis.

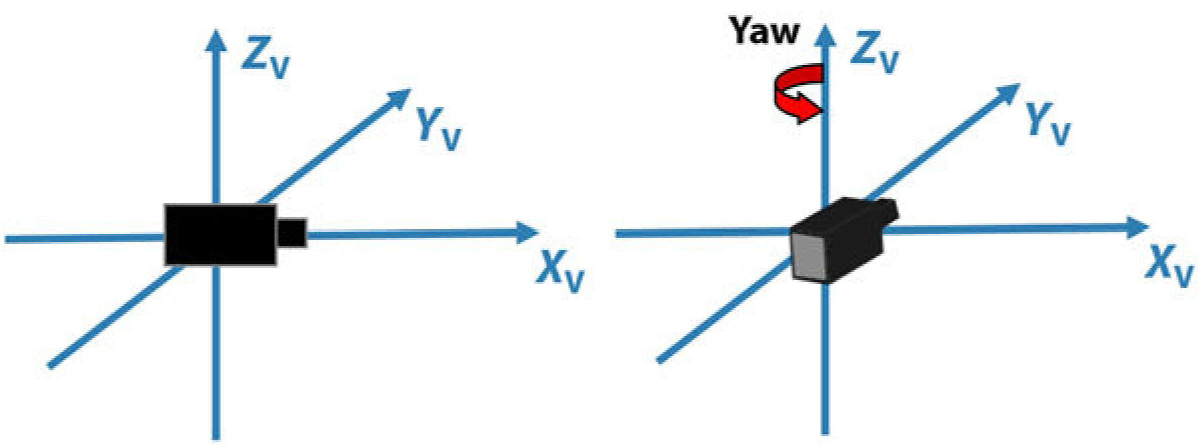


For more details, see “Angle Directions” on page 4-741.

Yaw — Yaw angle
real scalar

Yaw angle between the X_V axis of the vehicle and the optical axis of the camera, specified as a real scalar in degrees. To estimate this value, use the `estimateMonoCameraParameters` function.

`Yaw` uses the ISO convention for rotation, with a clockwise positive angle direction when looking in the positive direction of the vehicle's Z_V axis.



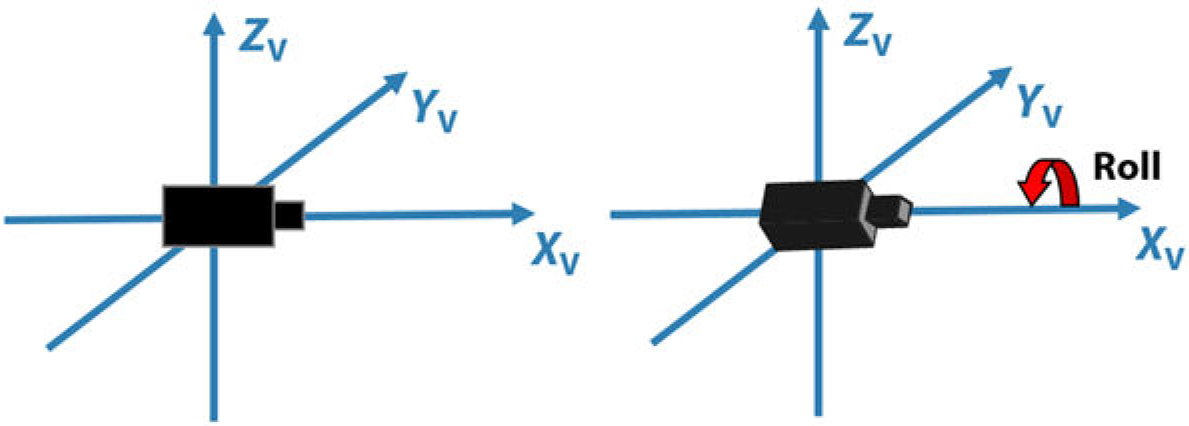
For more details, see “Angle Directions” on page 4-741.

`Roll` — Roll angle

real scalar

Roll angle of the camera around its optical axis, returned as a real scalar in degrees. To estimate this value, use the `estimateMonoCameraParameters` function.

`Roll` uses the ISO convention for rotation, with a clockwise positive angle direction when looking in the positive direction of the vehicle's X_V axis.



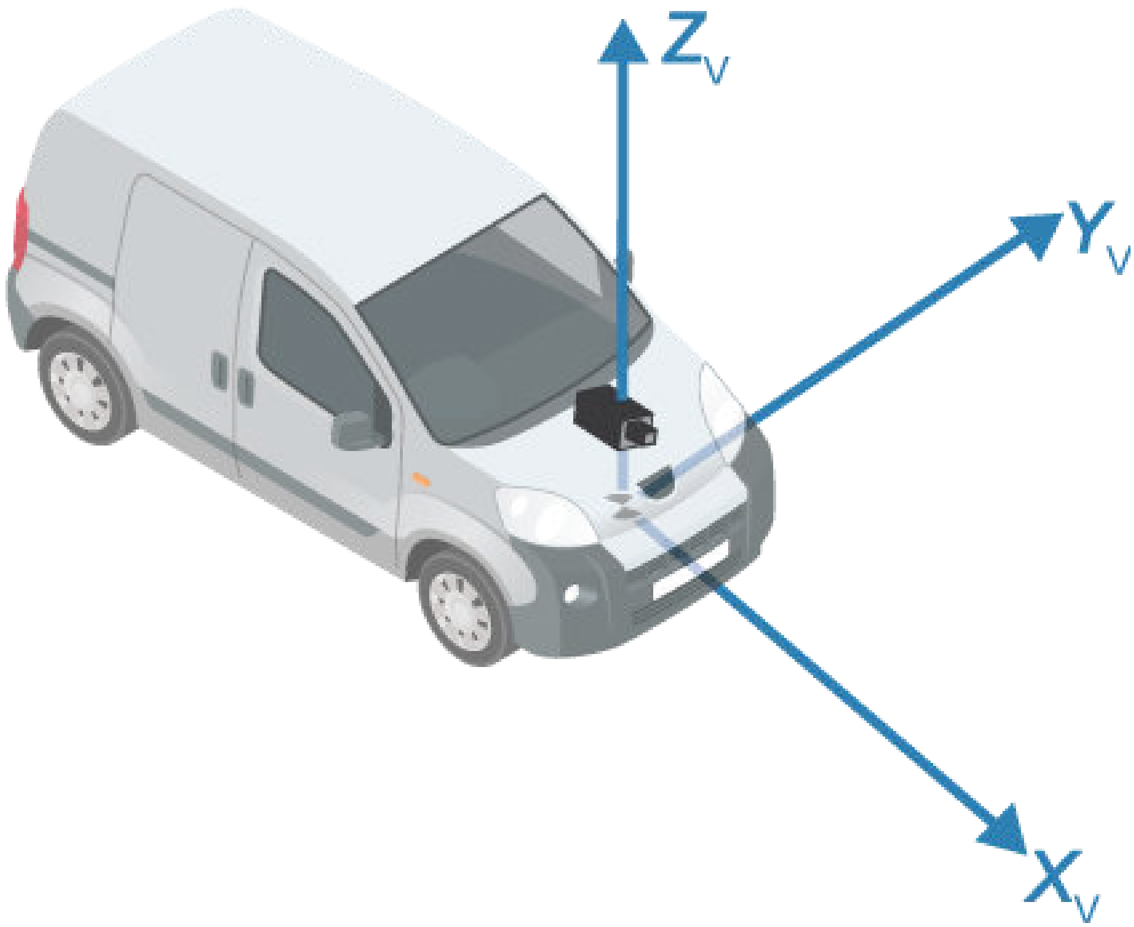
For more details, see “Angle Directions” on page 4-741.

`SensorLocation` — Location of center of camera sensor

`[0 0]` (default) | two-element vector

Location of the center of the camera sensor, specified as a two-element vector of the form `[x y]`. Use this property to change the placement of the camera. Units are in the vehicle coordinate system (X_V, Y_V, Z_V).

By default, the camera sensor is located at the (X_V, Y_V) origin, at the height specified by `Height`.



`WorldUnits` — World coordinate system units
`'meters'` | character vector | string scalar

World coordinate system units, specified as a character vector or string scalar. This property only stores the unit type and does not affect any calculations. Any text is valid.

You can set this property when you create the object. After you create the object, this property is read-only.

Object Functions

imageToVehicle	Convert image coordinates to vehicle coordinates
vehicleToImage	Convert vehicle coordinates to image coordinates

Examples

Create Monocular Camera Object

Create a forward-facing monocular camera sensor mounted on an ego vehicle. Examine an image captured from the camera and determine locations within the image in both vehicle and image coordinates.

Set the intrinsic parameters of the camera. Specify the focal length, the principal point of the image plane, and the output image size. Units are in pixels. Save the intrinsics as a `cameraIntrinsics` (Computer Vision Toolbox) object.

```
focalLength = [800 800];
principalPoint = [320 240];
imageSize = [480 640];

intrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);
```

Specify the position of the camera. Position the camera 2.18 meters above the ground with a 14-degree pitch toward the ground.

```
height = 2.18;
pitch = 14;
```

Define a monocular camera sensor using the intrinsic camera parameters and the position of the camera. Load an image from the camera.

```
sensor = monoCamera(intrinsics,height,'Pitch',pitch);

Ioriginal = imread('road.png');
figure
imshow(Ioriginal)
title('Original Image')
```

Original Image

Determine the image coordinates of a point 10 meters directly in front of the camera. The *X*-axis points forward from the camera and the *Y*-axis points to the left.

```
xyVehicleLoc1 = [10 0];
xyImageLoc1 = vehicleToImage(sensor,xyVehicleLoc1)
```

```
xyImageLoc1 = 1×2

  320.0000  216.2296
```

Display the point on the image.

```
IvehicleToImage = insertMarker(Ioriginal,xyImageLoc1);
IvehicleToImage = insertText(IvehicleToImage,xyImageLoc1 + 5,'10 meters');
figure
imshow(IvehicleToImage)
title('Vehicle-to-Image Point')
```

Determine the vehicle coordinates of a point that lies on the road surface in the image.

```
xyImageLoc2 = [300 300];
xyVehicleLoc2 = imageToVehicle(sensor,xyImageLoc2)
```

```
xyVehicleLoc2 = 1×2

    6.5959    0.1732
```

The point is about 6.6 meters in front of the vehicle and about 0.17 meters to the left of the vehicle center.

Display the vehicle coordinates of the point on the image.

```
IimageToVehicle = insertMarker(Ioriginal,xyImageLoc2);
displayText = sprintf('(%.2f m, %.2f m)',xyVehicleLoc2);
IimageToVehicle = insertText(IimageToVehicle,xyImageLoc2 + 5,displayText);

figure
imshow(IimageToVehicle)
title('Image-to-Vehicle Point')
```

Generate Visual Detections from Monocular Camera

Create a vision sensor by using a monocular camera configuration, and generate detections from that sensor.

Specify the intrinsic parameters of the camera and create a `monoCamera` object from these parameters. The camera is mounted on top of an ego vehicle at a height of 1.5 meters above the ground and a pitch of 1 degree toward the ground.

```
focalLength = [800 800];
principalPoint = [320 240];
imageSize = [480 640];
intrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);

height = 1.5;
pitch = 1;
monoCamConfig = monoCamera(intrinsics,height,'Pitch',pitch);
```

Create a vision detection generator using the monocular camera configuration.

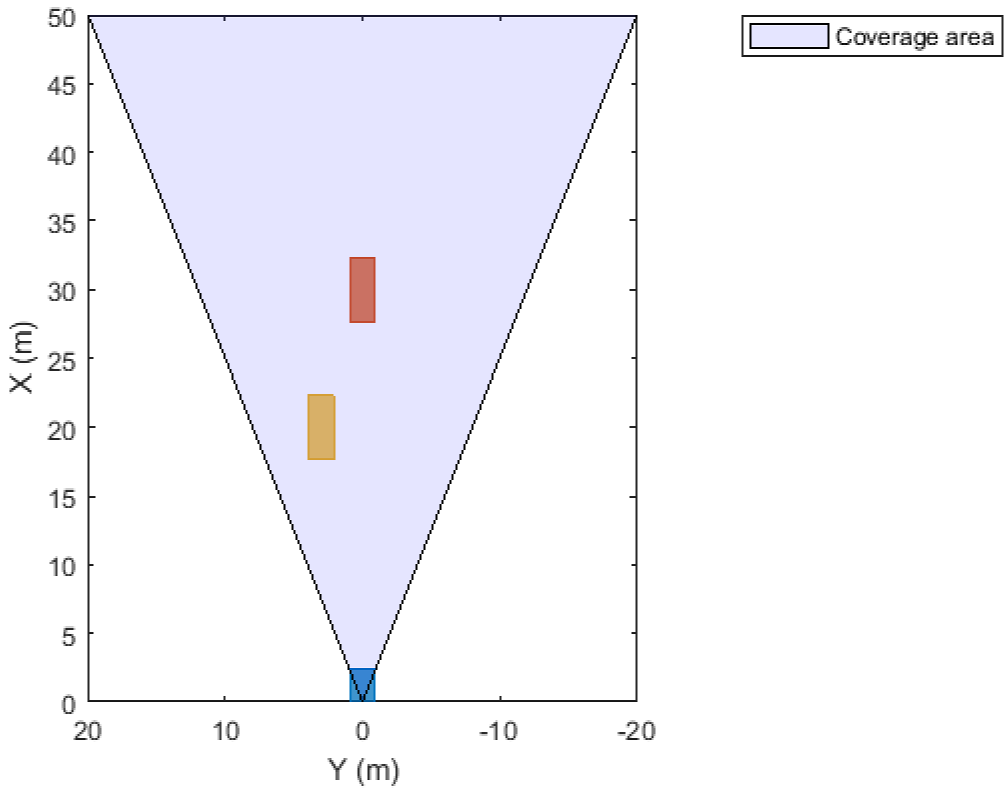
```
visionSensor = visionDetectionGenerator(monoCamConfig);
```

Generate a driving scenario with an ego vehicle and two target cars. Position the first target car 30 meters directly in front of the ego vehicle. Position the second target car 20 meters in front of the ego vehicle but offset to the left by 3 meters.

```
scenario = drivingScenario;
egoVehicle = vehicle(scenario,'ClassID',1);
targetCar1 = vehicle(scenario,'ClassID',1,'Position',[30 0 0]);
targetCar2 = vehicle(scenario,'ClassID',1,'Position',[20 3 0]);
```

Use a bird's-eye plot to display the vehicle outlines and sensor coverage area.

```
figure
bep = birdsEyePlot('XLim',[0 50],'YLim',[-20 20]);

olPlotter = outlinePlotter(bep);
[position,yaw,length,width,originOffset,color] = targetOutlines(egoVehicle);
plotOutline(olPlotter,position,yaw,length,width);

caPlotter = coverageAreaPlotter(bep,'DisplayName','Coverage area','FaceColor','blue');
plotCoverageArea(caPlotter,visionSensor.SensorLocation,visionSensor.MaxRange, ...
    visionSensor.Yaw,visionSensor.FieldOfView(1))
```



Obtain the poses of the target cars from the perspective of the ego vehicle. Use these poses to generate detections from the sensor.

```
poses = targetPoses(egoVehicle);
[dets,numValidDets] = visionSensor(poses,scenario.SimulationTime);
```

Display the (X,Y) positions of the valid detections. For each detection, the (X,Y) positions are the first two values of the `Measurement` field.

```
for i = 1:numValidDets
    XY = dets{i}.Measurement(1:2);
    detXY = sprintf('Detection %d: X = %.2f meters, Y = %.2f meters',i,XY);
    disp(detXY)
end
```

```
Detection 1: X = 19.09 meters, Y = 2.79 meters
Detection 2: X = 27.81 meters, Y = 0.08 meters
```

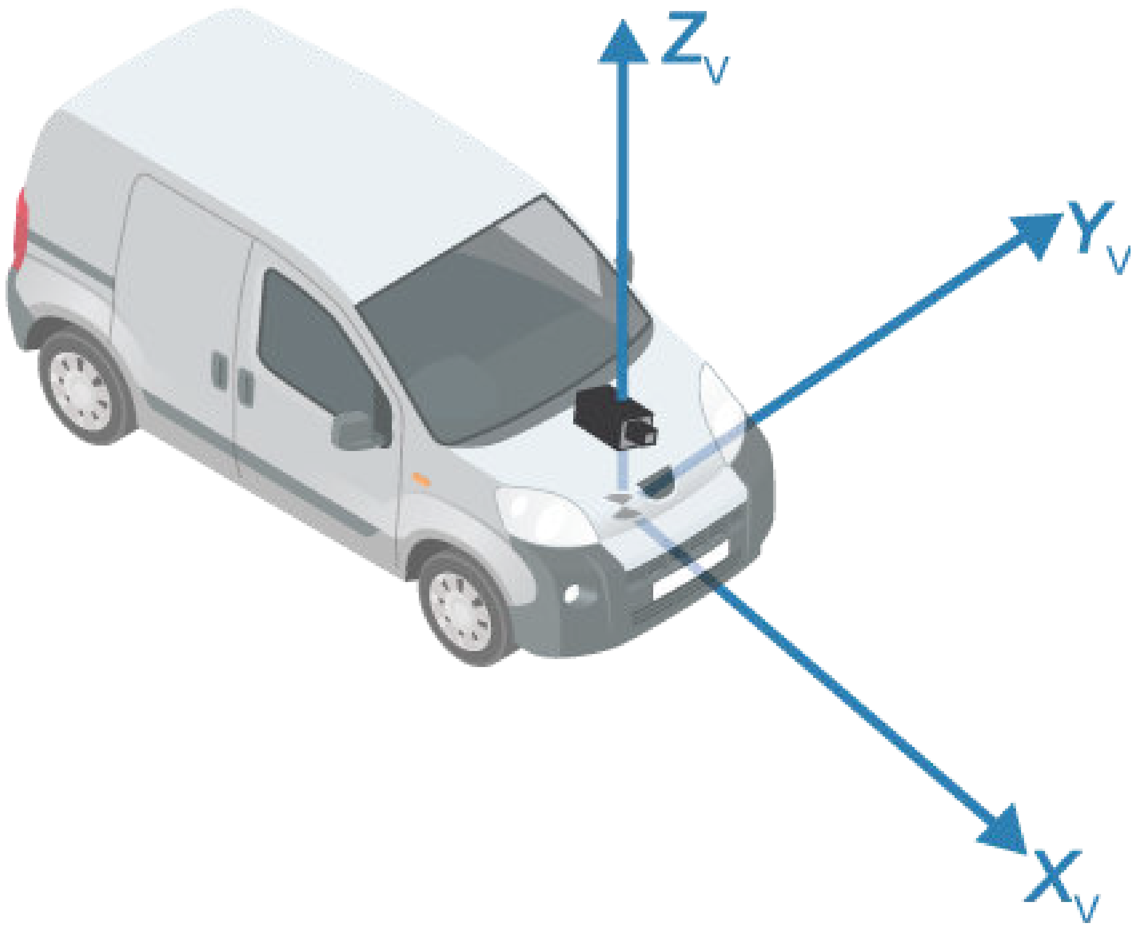
More About

Vehicle Coordinate System

In the vehicle coordinate system (X_V, Y_V, Z_V) defined by `monoCamera`:

- 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 default origin of this coordinate system is on the road surface, directly below the camera center. The focal point of the camera defines this center point.

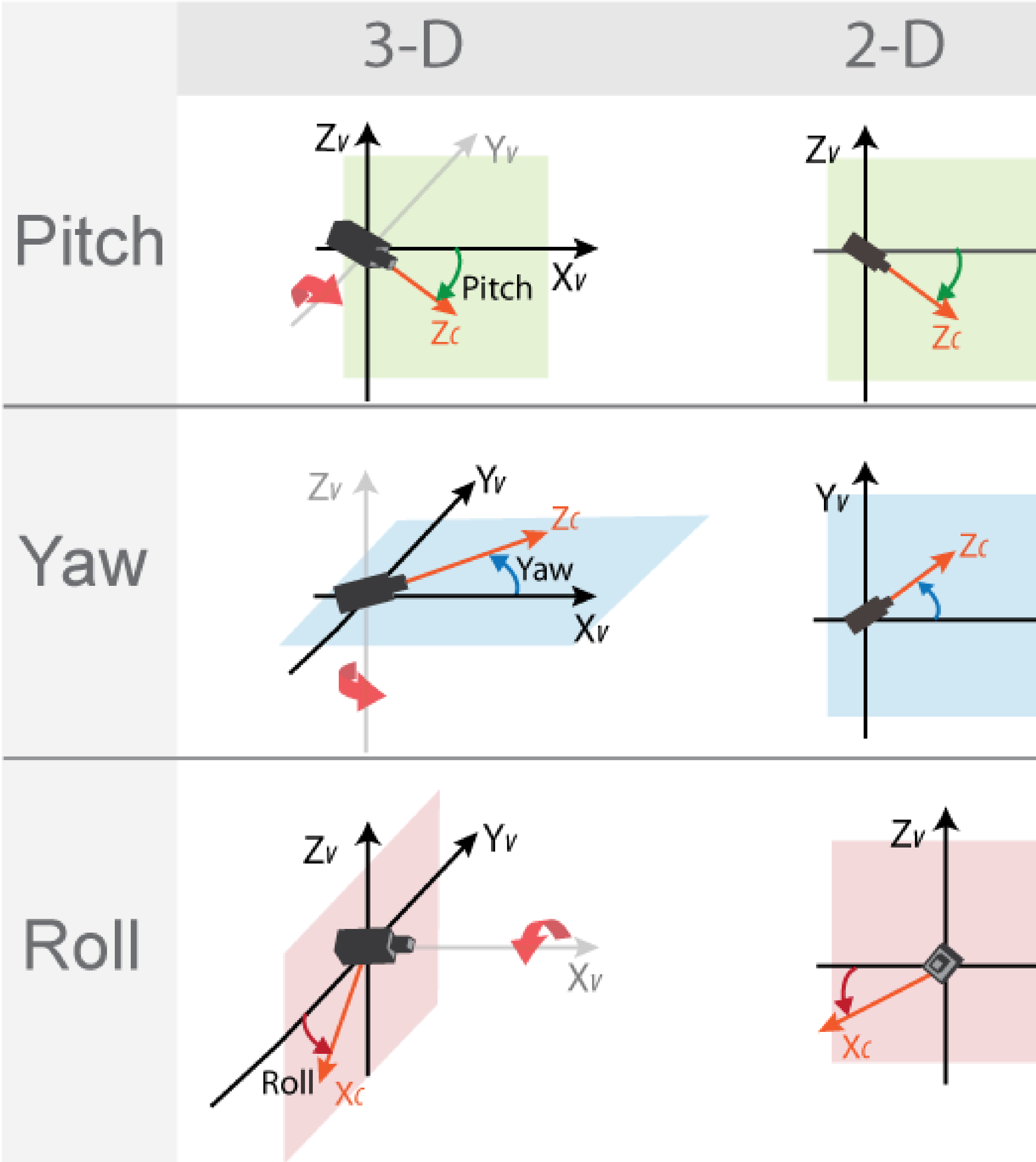


To change the placement of the origin within the vehicle coordinate system, update the `SensorLocation` property.

For more details about the vehicle coordinate system, see “Coordinate Systems in Automated Driving Toolbox”.

Angle Directions

The monocular camera sensor uses clockwise positive angle directions when looking in the positive direction of the *Z*-, *Y*-, and *X*-axes, respectively.


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

Compatibility Considerations

Direction of yaw angle rotation adjusted
Behavior changed in R2018a

Starting in R2018a, the `monoCamera` object uses the correct direction of rotation for the yaw angle. When you look in the positive direction of the vehicle's *Z*-axis, the yaw angle is now positive in the clockwise direction. Previously, this angle was positive in the counterclockwise direction.

If you are using R2017b or earlier, to use the correct direction of rotation, update the yaw angle to its negative value. For example, to update the yaw angle for a `monoCamera` object named `sensor`, use this code:

```
sensor.Yaw = -sensor.Yaw;
```

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Apps
Camera Calibrator

Functions
`estimateCameraParameters` | `estimateMonoCameraParameters` | `extrinsics`

Objects
`birdsEyeView` | `cameraIntrinsics` | `cameraParameters`

Topics
"Calibrate a Monocular Camera"
"Configure Monocular Fisheye Camera"
"Visual Perception Using Monocular Camera"
"Coordinate Systems in Automated Driving Toolbox"

Introduced in R2017a

vehicleToImage

Convert vehicle coordinates to image coordinates

Syntax

```
imagePoints = vehicleToImage(monoCam,vehiclePoints)
```

Description

`imagePoints = vehicleToImage(monoCam,vehiclePoints)` converts [*x y*] or [*x y z*] vehicle coordinates to [*x y*] image coordinates by applying a projective transformation. The monocular camera object, `monoCam`, contains the camera parameters.

Examples

Create Monocular Camera Object

Create a forward-facing monocular camera sensor mounted on an ego vehicle. Examine an image captured from the camera and determine locations within the image in both vehicle and image coordinates.

Set the intrinsic parameters of the camera. Specify the focal length, the principal point of the image plane, and the output image size. Units are in pixels. Save the intrinsics as a `cameraIntrinsics` (Computer Vision Toolbox) object.

```
focalLength = [800 800];
principalPoint = [320 240];
imageSize = [480 640];

intrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);
```

Specify the position of the camera. Position the camera 2.18 meters above the ground with a 14-degree pitch toward the ground.

```
height = 2.18;
pitch = 14;
```

Define a monocular camera sensor using the intrinsic camera parameters and the position of the camera. Load an image from the camera.

```
sensor = monoCamera(intrinsics,height,'Pitch',pitch);

Ioriginal = imread('road.png');
figure
imshow(Ioriginal)
title('Original Image')
```

Original Image

Determine the image coordinates of a point 10 meters directly in front of the camera. The *X*-axis points forward from the camera and the *Y*-axis points to the left.

```
xyVehicleLoc1 = [10 0];
xyImageLoc1 = vehicleToImage(sensor,xyVehicleLoc1)
```

```
xyImageLoc1 = 1×2

  320.0000   216.2296
```

Display the point on the image.

```
IvehicleToImage = insertMarker(Ioriginal,xyImageLoc1);
IvehicleToImage = insertText(IvehicleToImage,xyImageLoc1 + 5,'10 meters');
figure
imshow(IvehicleToImage)
title('Vehicle-to-Image Point')
```

Determine the vehicle coordinates of a point that lies on the road surface in the image.

```
xyImageLoc2 = [300 300];
xyVehicleLoc2 = imageToVehicle(sensor,xyImageLoc2)
```

```
xyVehicleLoc2 = 1×2

    6.5959    0.1732
```

The point is about 6.6 meters in front of the vehicle and about 0.17 meters to the left of the vehicle center.

Display the vehicle coordinates of the point on the image.

```
IimageToVehicle = insertMarker(Ioriginal,xyImageLoc2);
displayText = sprintf('(%.2f m, %.2f m)',xyVehicleLoc2);
IimageToVehicle = insertText(IimageToVehicle,xyImageLoc2 + 5,displayText);

figure
imshow(IimageToVehicle)
title('Image-to-Vehicle Point')
```

Input Arguments

monoCam — Monocular camera parameters
monoCamera object

Monocular camera parameters, specified as a monoCamera object.

vehiclePoints — Vehicle points
M-by-2 matrix | *M*-by-3 matrix

Vehicle points, specified as an *M*-by-2 or *M*-by-3 matrix containing *M* number of [*x y*] or [*x y z*] vehicle coordinates.

Output Arguments

imagePoints — Image points
M-by-2 matrix

Image points, returned as an *M*-by-2 matrix containing *M* number of [*x y*] image coordinates.

See Also

Objects
`monoCamera`

Functions
`imageToVehicle`

Topics
"Coordinate Systems in Automated Driving Toolbox"

Introduced in R2017a

imageToVehicle

Convert image coordinates to vehicle coordinates

Syntax

```
vehiclePoints = imageToVehicle(monoCam,imagePoints)
```

Description

`vehiclePoints = imageToVehicle(monoCam,imagePoints)` converts image coordinates to [*x* *y*] vehicle coordinates by applying a projective transformation. The monocular camera object, `monoCam`, contains the camera parameters.

Examples

Create Monocular Camera Object

Create a forward-facing monocular camera sensor mounted on an ego vehicle. Examine an image captured from the camera and determine locations within the image in both vehicle and image coordinates.

Set the intrinsic parameters of the camera. Specify the focal length, the principal point of the image plane, and the output image size. Units are in pixels. Save the intrinsics as a `cameraIntrinsics` (Computer Vision Toolbox) object.

```
focalLength = [800 800];
principalPoint = [320 240];
imageSize = [480 640];

intrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);
```

Specify the position of the camera. Position the camera 2.18 meters above the ground with a 14-degree pitch toward the ground.

```
height = 2.18;
pitch = 14;
```

Define a monocular camera sensor using the intrinsic camera parameters and the position of the camera. Load an image from the camera.

```
sensor = monoCamera(intrinsics,height,'Pitch',pitch);

Ioriginal = imread('road.png');
figure
imshow(Ioriginal)
title('Original Image')
```

Determine the image coordinates of a point 10 meters directly in front of the camera. The *X*-axis points forward from the camera and the *Y*-axis points to the left.

```
xyVehicleLoc1 = [10 0];
xyImageLoc1 = vehicleToImage(sensor,xyVehicleLoc1)
```

```
xyImageLoc1 = 1×2

  320.0000  216.2296
```

Display the point on the image.

```
IvehicleToImage = insertMarker(Ioriginal,xyImageLoc1);
IvehicleToImage = insertText(IvehicleToImage,xyImageLoc1 + 5,'10 meters');
figure
imshow(IvehicleToImage)
title('Vehicle-to-Image Point')
```

Vehicle-to-Image Point

Determine the vehicle coordinates of a point that lies on the road surface in the image.

```
xyImageLoc2 = [300 300];
xyVehicleLoc2 = imageToVehicle(sensor,xyImageLoc2)
```

```
xyVehicleLoc2 = 1×2

    6.5959    0.1732
```

The point is about 6.6 meters in front of the vehicle and about 0.17 meters to the left of the vehicle center.

Display the vehicle coordinates of the point on the image.

```
IimageToVehicle = insertMarker(Ioriginal,xyImageLoc2);
displayText = sprintf('(%.2f m, %.2f m)',xyVehicleLoc2);
IimageToVehicle = insertText(IimageToVehicle,xyImageLoc2 + 5,displayText);

figure
imshow(IimageToVehicle)
title('Image-to-Vehicle Point')
```

Input Arguments

`monoCam` — Monocular camera parameters
`monoCamera` object

Monocular camera parameters, specified as a `monoCamera` object.

`imagePoints` — Image points
M-by-2 matrix

Image points, specified as an *M*-by-2 matrix containing *M* number of [*x* *y*] image coordinates.

Output Arguments

`vehiclePoints` — Vehicle points
M-by-2 matrix

Vehicle points, returned as an *M*-by-2 matrix containing *M* number of [*x* *y*] vehicle coordinates.

See Also

Objects
`monoCamera`

Functions
`vehicleToImage`

Topics
"Coordinate Systems in Automated Driving Toolbox"

Introduced in R2017a

vision.labeler.loading.MultiSignalSource class

Package: `vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading`
Superclasses: `matlab.mixin.Heterogeneous`

Interface for loading signal data into Ground Truth Labeler app

Description

The `vision.labeler.loading.MultiSignalSource` class creates an interface for loading signals from a data source into the **Ground Truth Labeler** app. The data source can be a file format or any custom source.

The interface created using this class enables you to customize the panel for loading custom data sources in the Add/Remove Signal dialog box of the app. The figure shows a sample loading panel.

The class also provides an interface to read frames from loaded signals. The app renders these frames for labeling.

To define a custom class to load a data source into the app, follow these steps.

1. Create a class that inherits from the `vision.labeler.loading.MultiSignalSource` class. The class definition must have this format, where *`customSourceClass`* is the name of your custom data source class.

   ```
   classdef customSourceClass < vision.labeler.loading.MultiSignalSource
   ```
2. Save the class to this folder, where *`matlabroot`* is the full path to your MATLAB installation folder as returned by the `matlabroot` function.

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

   Alternatively, create a `+vision/+labeler/+loading` folder structure, add these folders to the MATLAB search path, and save the class to the `+vision/+labeler/+loading` folder. The **Ground Truth Labeler** app recognizes data source classes in folders with this path only.
3. Define the class properties and methods required to load the data source into the app. This table shows the predefined custom classes that you can use as starting points for defining these properties and methods.

Class	Data Source Loaded by Class	Command to View Class Source Code
vision.labeler.loading.VideoSource	Video file	edit vision.labeler.loading.VideoSour
vision.labeler.loading.ImageSequenceSource	Image sequence folder	edit vision.labeler.loading.ImageSequ
vision.labeler.loading.VelodyneLidarSource	Velodyne packet capture (PCAP) file	edit vision.labeler.loading.VelodyneL
vision.labeler.loading.RosbagSource	Rosbag file	edit vision.labeler.loading.RosbagSou
vision.labeler.loading.PointCloudSequenceSource	Point cloud sequence folder	edit vision.labeler.loading.PointClou
vision.labeler.loading.CustomImageSource	Custom image format	edit vision.labeler.loading.CustomIma

For an explanation of the required properties and methods used for defining a custom data source class, see the “Create Class for Loading Custom Ground Truth Data Sources” example.

The `vision.labeler.loading.MultiSignalSource` class is a `handle` class.

Class Attributes

`Abstract`	`true`

For information on class attributes, see “Class Attributes”.

Properties

Name — Name of source type
string scalar

Name of the type of source that this class loads, specified as a string scalar.

Attributes:

`GetAccess`	`public`
`Abstract`	`true`
`Constant`	`true`
`NonCopyable`	`true`

Description — Description of class functionality
string scalar

Description of the functionality that this class provides, specified as a string scalar.

Attributes:

GetAccess	public
Abstract	true
Constant	true
NonCopyable	true

SourceName — Name of data source
string scalar

Name of the data source, specified as a string scalar. Typically, `SourceName` is the name of the file from which the signal is loaded.

Attributes:

GetAccess	public
SetAccess	protected

SourceParams — Parameters for loading signals from data source
structure

Parameters for loading signals from the data source into the app, specified as a structure. The fields of this structure contain values that the `loadSource` method requires to load the signal.

Attributes:

GetAccess	public
SetAccess	protected

SignalName — Names of signals in data source
string vector

Names of the signals that can be loaded from the data source, specified as a string vector.

Attributes:

GetAccess	public
SetAccess	protected

SignalType — Types of signals in data source
vector of `vision.labeler.loading.SignalType` enumerations

Types of the signals that can be loaded from the data source, specified as a vector of `vision.labeler.loading.SignalType` enumerations. Each signal listed in the `SignalName` property is of the type in the corresponding position of `SignalType`.

Attributes:

GetAccess	public
SetAccess	protected

Timestamp — Timestamps of signals in data source
cell array of `duration` vectors

Timestamps of the signals that can be loaded from the data source, specified as a cell array of `duration` vectors. Each signal listed in the `SignalName` property has the timestamps in the corresponding position of `Timestamp`.

Attributes:

GetAccess	public
SetAccess	protected

NumSignals — Number of signals in data source
nonnegative integer

Number of signals that can be read from the data source, specified as a nonnegative integer. `NumSignals` is equal to the number of signals in the `SignalName` property.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true
NonCopyable	true

Methods

Public Methods

customizeLoadPanel	`customizeLoadPanel(sourceObj,panel)` Customize the loading panel for the data source object. In the loading dialog box of the app, this method is invoked when you select the data source type from the **Source Type** list. Abstract: true
getLoadPanelData	`[sourceName,sourceParams] = getLoadPanelData(sourceObj)` Obtain the data needed to load the data source object currently selected in the loading panel. In the loading dialog box of the app, this method is invoked when you add a source. The method returns these outputs. • `sourceName` is a string capturing the name of the data source object. • `sourceParams` is a structure with fields containing the parameters required to load the data source object. Both of these outputs are passed to the `loadSource` method. Abstract: true

`loadSource`	`loadSource(sourceObj,sourceName,sourceParams)` Load a data source object into the app. In the loading dialog box of the app, this method is invoked after you add a source and the `getLoadPanelData` method executes successfully. This method is also invoked when you load the data source object into the MATLAB workspace. When you load the data source object, MATLAB expects that the source has the name `sourceName` and parameters `sourceParams` that are needed to load that source and read data from it. Abstract | true
`readFrame`	`frame  = readFrame(sourceObj,signalName,tsIndex)` Read a frame of data from a signal contained in a data source object at the specified timestamp index. The index must be in the bounds of the length of the timestamps for that signal. Abstract | true
`loadPanelChecker`	`loadPanelChecker` Check the load panel for the loading dialog box of the app. This method opens a dialog box similar to the loading dialog box that you open from the **Open** menu on the app toolstrip. Use this method to preview how the `customizeLoadPanel` method populates the loading panel for the selected data source object. Static | true

See Also

Apps
Ground Truth Labeler

Classes
`driving.connector.Connector` | `vision.labeler.AutomationAlgorithm`

Topics
"Sources vs. Signals in Ground Truth Labeling"
"Create Class for Loading Custom Ground Truth Data Sources"

Introduced in R2020a

vision.labeler.loading.VideoSource class

Package: `vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading`
Superclasses: `vision.labeler.loading.MultiSignalSource`

Load signals from video sources into Ground Truth Labeler app

Description

The `vision.labeler.loading.VideoSource` class creates an interface for loading signals from video data sources into the **Ground Truth Labeler** app. In the Add/Remove Signal dialog box of the app, when **Source Type** is set to `Video`, this class controls the parameters in that dialog box.

To access this dialog box, in the app, select **Open** > **Add Signals**.

The default implementation of this class loads the video formats accepted by the `VideoReader` object.

The `vision.labeler.loading.VideoSource` class is a `handle` class.

Creation

When you export labels from a **Ground Truth Labeler** app session that contains video sources, the exported `groundTruthMultisignal` object stores instances of this class in its `DataSource` property.

To create a `VideoSource` object programmatically, such as when programmatically creating a `groundTruthMultisignal` object, use the `vision.labeler.loading.VideoSource` function (described here).

Syntax

```
vidSource = vision.labeler.loading.VideoSource
```

Description

`vidSource = vision.labeler.loading.VideoSource` creates a `VideoSource` object for loading signals from video data sources. To specify the data source and the parameters required to load the source, use the `loadSource` method.

Properties

`Name` — Name of source type

`"Video"` (default) | string scalar

Name of the type of source that this class loads, specified as a string scalar.

Attributes:

`GetAccess`	`public`
`Constant`	`true`
`NonCopyable`	`true`

`Description` — Description of class functionality

`"A video reader"` (default) | string scalar

Description of the functionality that this class provides, specified as a string scalar.

Attributes:

`GetAccess`	`public`
`Constant`	`true`
`NonCopyable`	`true`

`SourceName` — Name of data source

`[]` (default) | string scalar

Name of the data source, specified as a string scalar. Typically, `SourceName` is the name of the file from which the signal is loaded.

Attributes:

`GetAccess`	`public`
`SetAccess`	`protected`

`SourceParams` — Parameters for loading video signal from data source

`[]` (default) | structure

Parameters for loading a video signal from a data source, specified as a structure.

This table describes the required and optional fields of the `SourceParams` structure.

Field	Description	Required or Optional
Timestamps	Timestamps for the video signal, specified as a cell array containing a single duration vector of timestamps. In the Add/Remove Signal dialog box of the app, if you set the **Timestamps** parameter to From Workspace and read the timestamps from a variable in the MATLAB workspace, then the SourceParams property stores these timestamps in the Timestamps field.	Optional In the Add/Remove Signal dialog box of the app, if you set the **Timestamps** parameter to From File and read the timestamps from the video file, then the structure does not include this field, and the SourceParams property is empty, [].

Attributes:

GetAccess	public
SetAccess	protected

SignalName — Names of signals in data source

[] (default) | string vector

Names of the signals that can be loaded from the data source, specified as a string vector.

Attributes:

GetAccess	public
SetAccess	protected

SignalType — Types of signals in data source

[] (default) | vector of vision.labeler.loading.SignalType enumerations

Types of the signals that can be loaded from the data source, specified as a vector of vision.labeler.loading.SignalType enumerations. Each signal listed in the SignalName property is of the type in the corresponding position of SignalType.

Attributes:

GetAccess	public
SetAccess	protected

Timestamp — Timestamps of signals in data source

[] (default) | cell array of duration vectors

Timestamps of the signals that can be loaded from the data source, specified as a cell array of duration vectors. Each signal listed in the SignalName property has the timestamps in the corresponding position of Timestamp.

Attributes:

GetAccess	public
SetAccess	protected

`NumSignals` — Number of signals in data source
`0` (default) | integer

Number of signals that can be read from the data source, specified as a nonnegative integer. `NumSignals` is equal to the number of signals in the `SignalName` property.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`
`NonCopyable`	`true`

Methods

Public Methods

`customizeLoadPanel`	`customizeLoadPanel(sourceObj,panel)` Customize the loading panel for the data source object. In the loading dialog box of the app, this method is invoked when you select the data source type from the **Source Type** list.
`getLoadPanelData`	`[sourceName,sourceParams] = getLoadPanelData(sourceObj)` Obtain the data needed to load the data source object currently selected in the loading panel. In the loading dialog box of the app, this method is invoked when you add a source. The method returns these outputs. • `sourceName` is a string capturing the name of the data source object. • `sourceParams` is a structure with fields containing the parameters required to load the data source object. Both of these outputs are passed to the `loadSource` method.
`loadSource`	`loadSource(sourceObj,sourceName,sourceParams)` Load a data source object into the app. In the loading dialog box of the app, this method is invoked after you add a source and the `getLoadPanelData` method executes successfully. This method is also invoked when you load the data source object into the MATLAB workspace. When you load the data source object, MATLAB expects that the source has the name `sourceName` and parameters `sourceParams` that are needed to load that source and read data from it.

readFrame	`frame = readFrame(sourceObj,signalName,tsIndex)` Read a frame of data from a signal contained in a data source object at the specified timestamp index. The index must be in the bounds of the length of the timestamps for that signal.
loadPanelChecker	`loadPanelChecker` Check the load panel for the loading dialog box of the app. This method opens a dialog box similar to the loading dialog box that you open from the **Open** menu on the app toolstrip. Use this method to preview how the `customizeLoadPanel` method populates the loading panel for the selected data source object. Static: true

Examples

Create Video Source

Create a video source from a video on the MATLAB® search path. Load the source name into the `VideoSource` object. The video has no source parameters needed to load it, so `sourceParams` is empty.

```
sourceName = 'caltech_cordova1.avi';
sourceParams = [];

vidSource = vision.labeler.loading.VideoSource;
loadSource(vidSource,sourceName,sourceParams);
```

Read the first frame from the video. Display the frame.

```
signalName = vidSource.SignalName;
I = readFrame(vidSource,signalName,1);

figure
imshow(I)
```

Tips

- You can this class as a starting point for creating a custom data source loading class. To view the source code for this class, use this command:

```
edit vision.labeler.loading.VideoSource
```

See Also

Apps
Ground Truth Labeler

Classes
`vision.labeler.loading.CustomImageSource` | `vision.labeler.loading.ImageSequenceSource` | `vision.labeler.loading.MultiSignalSource` | `vision.labeler.loading.PointCloudSequenceSource` | `vision.labeler.loading.RosbagSource` | `vision.labeler.loading.VelodyneLidarSource`

Topics

"Sources vs. Signals in Ground Truth Labeling"
"Create Class for Loading Custom Ground Truth Data Sources"

Introduced in R2020a

vision.labeler.loading.ImageSequenceSource class

Package: vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading
Superclasses: vision.labeler.loading.MultiSignalSource

Load signals from image sequence sources into Ground Truth Labeler app

Description

The vision.labeler.loading.ImageSequenceSource class creates an interface for loading signals from image sequence data sources into the **Ground Truth Labeler** app. In the Add/Remove Signal dialog box of the app, when **Source Type** is set to Image Sequence, this class controls the parameters in that dialog box.

To access this dialog box, in the app, select **Open** > **Add Signals**.

The default implementation of this class loads the image formats that can be read from an ImageDatastore object.

The vision.labeler.loading.ImageSequenceSource class is a handle class.

Creation

When you export labels from a **Ground Truth Labeler** app session that contains image sequence sources, the exported groundTruthMultisignal object stores instances of this class in its DataSource property.

To create an ImageSequenceSource object programmatically, such as when programmatically creating a groundTruthMultisignal object, use the vision.labeler.loading.ImageSequenceSource function (described here).

Syntax

```
imseqSource = vision.labeler.loading.ImageSequenceSource
```

Description

`imseqSource = vision.labeler.loading.ImageSequenceSource` creates an `ImageSequenceSource` object for loading signals from image sequence data sources. To specify the data source and the parameters required to load the source, use the `loadSource` method.

Properties

`Name` — Name of source type

`"Image Sequence"` (default) | string scalar

Name of the type of source that this class loads, specified as a string scalar.

Attributes:

`GetAccess`	`public`
`Constant`	`true`
`NonCopyable`	`true`

`Description` — Description of class functionality

`"An image sequence reader"` (default) | string scalar

Description of the functionality that this class provides, specified as a string scalar.

Attributes:

`GetAccess`	`public`
`Constant`	`true`
`NonCopyable`	`true`

`SourceName` — Name of data source

`[ ]` (default) | string scalar

Name of the data source, specified as a string scalar. Typically, `SourceName` is the name of the file from which the signal is loaded.

Attributes:

`GetAccess`	`public`
`SetAccess`	`protected`

`SourceParams` — Parameters for loading image sequence signal from data source

`[ ]` (default) | structure

Parameters for loading an image sequence signal from a data source, specified as a structure.

This table describes the required and optional fields of the `SourceParams` structure.

Field	Description	Required or Optional
Timestamps	Timestamps for the image sequence signal, specified as a cell array containing a single duration vector of timestamps. In the Add/Remove Signal dialog box of the app, if you set the **Timestamps** parameter to From Workspace and read the timestamps from a variable in the MATLAB workspace, then the SourceParams property stores these timestamps in the Timestamps field.	Optional If you set the **Timestamps** parameter to Use Default and use the default timestamps for image sequence signals, then the structure does not include this field, and the SourceParams property is empty, []. For image sequence signals, the default timestamp duration vector has elements from 0 seconds to the number of valid image files minus 1. Units are in seconds.

Attributes:

```
GetAccess        public
SetAccess        protected
```

SignalName — Names of signals in data source

[] (default) | string vector

Names of the signals that can be loaded from the data source, specified as a string vector.

Attributes:

```
GetAccess        public
SetAccess        protected
```

SignalType — Types of signals in data source

[] (default) | vector of vision.labeler.loading.SignalType enumerations

Types of the signals that can be loaded from the data source, specified as a vector of vision.labeler.loading.SignalType enumerations. Each signal listed in the SignalName property is of the type in the corresponding position of SignalType.

Attributes:

```
GetAccess        public
SetAccess        protected
```

Timestamp — Timestamps of signals in data source

[] (default) | cell array of duration vectors

Timestamps of the signals that can be loaded from the data source, specified as a cell array of duration vectors. Each signal listed in the SignalName property has the timestamps in the corresponding position of Timestamp.

Attributes:

```
GetAccess        public
SetAccess        protected
```

`NumSignals` — Number of signals in data source
`0` (default) | integer

Number of signals that can be read from the data source, specified as a nonnegative integer. `NumSignals` is equal to the number of signals in the `SignalName` property.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`
`NonCopyable`	`true`

Methods

Public Methods

`customizeLoadPanel`	`customizeLoadPanel(sourceObj,panel)` Customize the loading panel for the data source object. In the loading dialog box of the app, this method is invoked when you select the data source type from the **Source Type** list.
`getLoadPanelData`	`[sourceName,sourceParams] = getLoadPanelData(sourceObj)` Obtain the data needed to load the data source object currently selected in the loading panel. In the loading dialog box of the app, this method is invoked when you add a source. The method returns these outputs. • `sourceName` is a string capturing the name of the data source object. • `sourceParams` is a structure with fields containing the parameters required to load the data source object. Both of these outputs are passed to the `loadSource` method.
`loadSource`	`loadSource(sourceObj,sourceName,sourceParams)` Load a data source object into the app. In the loading dialog box of the app, this method is invoked after you add a source and the `getLoadPanelData` method executes successfully. This method is also invoked when you load the data source object into the MATLAB workspace. When you load the data source object, MATLAB expects that the source has the name `sourceName` and parameters `sourceParams` that are needed to load that source and read data from it.

`readFrame`	`frame = readFrame(sourceObj,signalName,tsIndex)` Read a frame of data from a signal contained in a data source object at the specified timestamp index. The index must be in the bounds of the length of the timestamps for that signal.

`loadPanelChecker`	`loadPanelChecker`
	
	Check the load panel for the loading dialog box of the app. This method opens a dialog box similar to the loading dialog box that you open from the **Open** menu on the app toolstrip. Use this method to preview how the `customizeLoadPanel` method populates the loading panel for the selected data source object.
	
	`Static` | `true`

Examples

Create Image Sequence Source

Specify the path to a folder containing an image sequence.

```
imseqFolder = fullfile(toolboxdir('driving'),'drivingdata','roadSequence');
```

Load the timestamps corresponding to the sequence

```
load(fullfile(imseqFolder,'timeStamps.mat'))
```

Create an image sequence source. Load the folder path and timestamps into the `ImageSequenceSource` object.

```
sourceName = imseqFolder;
sourceParams = struct;
sourceParams.Timestamps = timeStamps;

imseqSource = vision.labeler.loading.ImageSequenceSource;
loadSource(imseqSource,sourceName,sourceParams);
```

Read the first frame in the sequence. Display the frame.

```
signalName = imseqSource.SignalName;
I = readFrame(imseqSource,signalName,1);

figure
imshow(I)
```

Tips

- You can this class as a starting point for creating a custom data source loading class. To view the source code for this class, use this command:

```
edit vision.labeler.loading.ImageSequenceSource
```

See Also

Apps
Ground Truth Labeler

Classes
`vision.labeler.loading.CustomImageSource` | `vision.labeler.loading.PointCloudSequenceSource` | `vision.labeler.loading.RosbagSource` | `vision.labeler.loading.VelodyneLidarSource` | `vision.labeler.loading.VideoSource`

Topics
"Sources vs. Signals in Ground Truth Labeling"

“Create Class for Loading Custom Ground Truth Data Sources”

Introduced in R2020a

vision.labeler.loading.VelodyneLidarSource class

Package: `vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading`
Superclasses: `vision.labeler.loading.MultiSignalSource`

Load signals from Velodyne lidar sources into Ground Truth Labeler app

Description

The `vision.labeler.loading.VelodyneLidarSource` class creates an interface for loading signals from Velodyne packet capture (PCAP) lidar data sources into the **Ground Truth Labeler** app. In the Add/Remove Signal dialog box of the app, when **Source Type** is set to `Velodyne Lidar`, this class controls the parameters in that dialog box.

To access this dialog box, in the app, select **Open** > **Add Signals**.

The default implementation of this class loads Velodyne PCAP files from the device models accepted by the `velodyneFileReader` function.

The `vision.labeler.loading.VelodyneLidarSource` class is a `handle` class.

Creation

When you export labels from a **Ground Truth Labeler** app session that contains Velodyne lidar sources, the exported `groundTruthMultisignal` object stores instances of this class in its `DataSource` property.

To create a `VelodyneLidarSource` object programmatically, such as when programmatically creating a `groundTruthMultisignal` object, use the `vision.labeler.loading.VelodyneLidarSource` function (described here).

Syntax

```
velodyneSource = vision.labeler.loading.VelodyneLidarSource
```

Description

`velodyneSource = vision.labeler.loading.VelodyneLidarSource` creates a `VelodyneLidarSource` object for loading signals from Velodyne lidar data sources. To specify the data source and the parameters required to load the source, use the `loadSource` method.

Properties

`Name` — Name of source type
`"Velodyne Lidar"` (default) | string scalar

Name of the type of source that this class loads, specified as a string scalar.

Attributes:

GetAccess	public
Constant	true
NonCopyable	true

`Description` — Description of class functionality
`"A Velodyne file reader"` (default) | string scalar

Description of the functionality that this class provides, specified as a string scalar.

Attributes:

GetAccess	public
Constant	true
NonCopyable	true

`SourceName` — Name of data source
`[]` (default) | string scalar

Name of the data source, specified as a string scalar. Typically, `SourceName` is the name of the file from which the signal is loaded.

Attributes:

GetAccess	public
SetAccess	protected

`SourceParams` — Parameters for loading Velodyne lidar signal from data source
`[]` (default) | structure

Parameters for loading a Velodyne lidar signal from a data source, specified as a structure.

This table describes the required and optional fields of the `SourceParams` structure.

Field	Description	Required or Optional
`Timestamps`	Timestamps for the Velodyne lidar signal, specified as a cell array containing a single `duration` vector of timestamps. In the Add/Remove Signal dialog box of the app, if you set the **Timestamps** parameter to `From Workspace` and read the timestamps from a variable in the MATLAB workspace, then the `SourceParams` property stores these timestamps in the `Timestamps` field.	Optional In the Add/Remove Signal dialog box of the app, if you set the **Timestamps** parameter to `From File` and read the timestamps from the Velodyne PCAP file, then the structure does not include this field.
`DeviceModel`	Velodyne device model name, specified as one of these options. If you specify the incorrect device model for your Velodyne PCAP file, the app loads an improperly calibrated point cloud. In the Add/Remove Signal dialog box of the app, select the device model from the **Device Model** parameter. The **Calibration File** parameter updates to the calibration file of the selected device model.	Required

Field	Description	Required or Optional

CalibrationFile	Name of the Velodyne calibration XML file, specified as a character vector or string scalar.
	
	To specify one of the calibration files included with your MATLAB installation, at the MATLAB command prompt, enter this code. Replace `<DeviceModel>` with the name of the device model that you specify in the `DeviceModel` field of this structure (without quotes).
	
	`calibrationFile = fullfile( ...`
	`matlabroot,'toolbox', ...`
	`'shared','pointclouds','utilities', ...`
	`'velodyneFileReaderConfiguration', ...`
	`'<DeviceModel>.xml')`
	
	By default, the `CalibrationFile` field is set to the full path to the `VLP16.xml` file, which is the calibration file for the VLP-16 device model.
	
	In the Add/Remove Signal dialog box of the app, when you change the **Device Model** parameter selection, the **Calibration File** parameter updates to the corresponding calibration file for the selected device model. You can also browse for or enter a path to a different calibration file in the **Calibration File** box.
		Required

For more details on device models and calibration files, see the `velodyneFileReader` object reference page.

Attributes:

GetAccess	public
SetAccess	protected

SignalName — Names of signals in data source

[ ] (default) | string vector

Names of the signals that can be loaded from the data source, specified as a string vector.

Attributes:

GetAccess	public
SetAccess	protected

SignalType — Types of signals in data source
[] (default) | vector of vision.labeler.loading.SignalType enumerations

Types of the signals that can be loaded from the data source, specified as a vector of vision.labeler.loading.SignalType enumerations. Each signal listed in the SignalName property is of the type in the corresponding position of SignalType.

Attributes:

GetAccess	public
SetAccess	protected

Timestamp — Timestamps of signals in data source
[] (default) | cell array of duration vectors

Timestamps of the signals that can be loaded from the data source, specified as a cell array of duration vectors. Each signal listed in the SignalName property has the timestamps in the corresponding position of Timestamp.

Attributes:

GetAccess	public
SetAccess	protected

NumSignals — Number of signals in data source
0 (default) | integer

Number of signals that can be read from the data source, specified as a nonnegative integer. NumSignals is equal to the number of signals in the SignalName property.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true
NonCopyable	true

Methods

Public Methods

customizeLoadPanel	customizeLoadPanel(sourceObj,panel) Customize the loading panel for the data source object. In the loading dialog box of the app, this method is invoked when you select the data source type from the **Source Type** list.

`getLoadPanelData`	`[sourceName,sourceParams] = getLoadPanelData(sourceObj)` Obtain the data needed to load the data source object currently selected in the loading panel. In the loading dialog box of the app, this method is invoked when you add a source. The method returns these outputs. • `sourceName` is a string capturing the name of the data source object. • `sourceParams` is a structure with fields containing the parameters required to load the data source object. Both of these outputs are passed to the `loadSource` method.
`loadSource`	`loadSource(sourceObj,sourceName,sourceParams)` Load a data source object into the app. In the loading dialog box of the app, this method is invoked after you add a source and the `getLoadPanelData` method executes successfully. This method is also invoked when you load the data source object into the MATLAB workspace. When you load the data source object, MATLAB expects that the source has the name `sourceName` and parameters `sourceParams` that are needed to load that source and read data from it.
`readFrame`	`frame  = readFrame(sourceObj,signalName,tsIndex)` Read a frame of data from a signal contained in a data source object at the specified timestamp index. The index must be in the bounds of the length of the timestamps for that signal.
`loadPanelChecker`	`loadPanelChecker` Check the load panel for the loading dialog box of the app. This method opens a dialog box similar to the loading dialog box that you open from the **Open** menu on the app toolstrip. Use this method to preview how the `customizeLoadPanel` method populates the loading panel for the selected data source object. Static: true

Examples

Create Velodyne Lidar Source

Specify the name of the Velodyne® lidar data source, a packet capture (PCAP) file.

```
sourceName = fullfile(toolboxdir('vision'),'visiondata', ...
    'lidarData_ConstructionRoad.pcap');
```

Specify information needed to load the source, including the device model of the lidar and the calibration file.

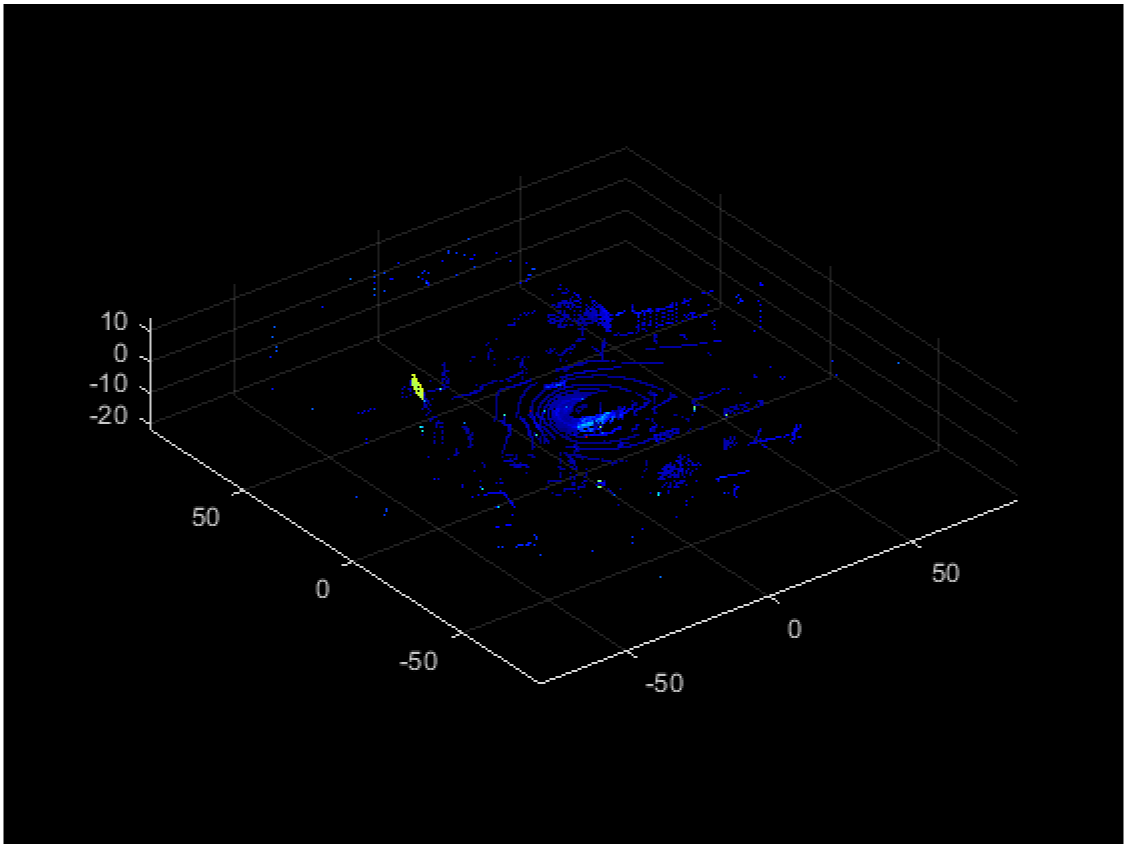
```
sourceParams = struct;
sourceParams.DeviceModel = 'HDL32E';
sourceParams.CalibrationFile = fullfile(matlabroot,'toolbox','shared', ...
    'pointclouds','utilities','velodyneFileReaderConfiguration', ...
    'HDL32E.xml');
```

Create the Velodyne lidar data source. Load the data source path, device model, and calibration file path into the `VelodyneLidarSource` object.

```
velodyneSource = vision.labeler.loading.VelodyneLidarSource;
loadSource(velodyneSource,sourceName,sourceParams);
```

Read the first frame from the source. Display the frame.

```
signalName = velodyneSource.SignalName;
pc = readFrame(velodyneSource,signalName,1);

figure
pcshow(pc)
```



Tips

- You can use this class as a starting point for creating a custom data source loading class. To view the source code for this class, use this command:

```
edit vision.labeler.loading.VelodyneLidarSource
```

See Also

Apps
Ground Truth Labeler

Classes
`vision.labeler.loading.CustomImageSource` | `vision.labeler.loading.ImageSequenceSource` | `vision.labeler.loading.PointCloudSequenceSource` | `vision.labeler.loading.RosbagSource` | `vision.labeler.loading.VideoSource`

Topics
"Sources vs. Signals in Ground Truth Labeling"
"Create Class for Loading Custom Ground Truth Data Sources"

Introduced in R2020a

vision.labeler.loading.RosbagSource class

Package: vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading
Superclasses: vision.labeler.loading.MultiSignalSource

Load signals from rosbag sources into Ground Truth Labeler app

Description

The vision.labeler.loading.RosbagSource class creates an interface for loading signals from rosbag files into the **Ground Truth Labeler** app. In the Add/Remove Signal dialog box of the app, when **Source Type** is set to Rosbag, this class controls the parameters in that dialog box.

To access this dialog box, in the app, select **Open** > **Add Signals**.

The default implementation of this class loads signals from these ROS message types:

- sensor_msgs/Image
- sensor_msgs/CompressedImage
- sensor_msgs/PointCloud2

Note This class requires ROS Toolbox.

The vision.labeler.loading.RosbagSource class is a handle class.

Creation

When you export labels from a **Ground Truth Labeler** app session that contains rosbag sources, the exported groundTruthMultisignal object stores instances of this class in its DataSource property.

To create a RosbagSource object programmatically, such as when programmatically creating a groundTruthMultisignal object, use the vision.labeler.loading.RosbagSource function (described here).

Syntax

`rosbagSource = vision.labeler.loading.RosbagSource`

Description

`rosbagSource = vision.labeler.loading.RosbagSource` creates a `RosbagSource` object for loading signals from rosbag data sources. To specify the data source and the parameters required to load the source, use the `loadSource` method.

Properties

`Name` — Name of source type
`"Rosbag"` (default) | string scalar

Name of the type of source that this class loads, specified as a string scalar.

Attributes:

`GetAccess`	`public`
`Constant`	`true`
`NonCopyable`	`true`

`Description` — Description of class functionality
`"A rosbag reader"` (default) | string scalar

Description of the functionality that this class provides, specified as a string scalar.

Attributes:

`GetAccess`	`public`
`Constant`	`true`
`NonCopyable`	`true`

`SourceName` — Name of data source
`[]` (default) | string scalar

Name of the data source, specified as a string scalar. Typically, `SourceName` is the name of the file from which the signal is loaded.

Attributes:

`GetAccess`	`public`
`SetAccess`	`protected`

`SourceParams` — Parameters for loading signals from rosbag data source
`[]` (default) | empty structure

Parameters for loading signals from a rosbag data source, specified as an empty structure. When you load image or lidar signals from a rosbag, do not specify the signal timestamps or any other parameters. The `loadSource` method reads these parameters from the rosbag.

Attributes:

GetAccess	public
SetAccess	protected

SignalName — Names of signals in data source
[ ] (default) | string vector

Names of the signals that can be loaded from the data source, specified as a string vector.

Attributes:

GetAccess	public
SetAccess	protected

SignalType — Types of signals in data source
[ ] (default) | vector of vision.labeler.loading.SignalType enumerations

Types of the signals that can be loaded from the data source, specified as a vector of vision.labeler.loading.SignalType enumerations. Each signal listed in the SignalName property is of the type in the corresponding position of SignalType.

Attributes:

GetAccess	public
SetAccess	protected

Timestamp — Timestamps of signals in data source
[ ] (default) | cell array of duration vectors

Timestamps of the signals that can be loaded from the data source, specified as a cell array of duration vectors. Each signal listed in the SignalName property has the timestamps in the corresponding position of Timestamp.

Attributes:

GetAccess	public
SetAccess	protected

NumSignals — Number of signals in data source
0 (default) | integer

Number of signals that can be read from the data source, specified as a nonnegative integer. NumSignals is equal to the number of signals in the SignalName property.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true
NonCopyable	true

Methods

Public Methods

`customizeLoadPanel`	`customizeLoadPanel(sourceObj,panel)` Customize the loading panel for the data source object. In the loading dialog box of the app, this method is invoked when you select the data source type from the **Source Type** list.
`getLoadPanelData`	`[sourceName,sourceParams] = getLoadPanelData(sourceObj)` Obtain the data needed to load the data source object currently selected in the loading panel. In the loading dialog box of the app, this method is invoked when you add a source. The method returns these outputs. • `sourceName` is a string capturing the name of the data source object. • `sourceParams` is a structure with fields containing the parameters required to load the data source object. Both of these outputs are passed to the `loadSource` method.
`loadSource`	`loadSource(sourceObj,sourceName,sourceParams)` Load a data source object into the app. In the loading dialog box of the app, this method is invoked after you add a source and the `getLoadPanelData` method executes successfully. This method is also invoked when you load the data source object into the MATLAB workspace. When you load the data source object, MATLAB expects that the source has the name `sourceName` and parameters `sourceParams` that are needed to load that source and read data from it.
`readFrame`	`frame  = readFrame(sourceObj,signalName,tsIndex)` Read a frame of data from a signal contained in a data source object at the specified timestamp index. The index must be in the bounds of the length of the timestamps for that signal.

loadPanelChecker	`loadPanelChecker` Check the load panel for the loading dialog box of the app. This method opens a dialog box similar to the loading dialog box that you open from the **Open** menu on the app toolstrip. Use this method to preview how the `customizeLoadPanel` method populates the loading panel for the selected data source object. Static: true

Tips

- You can this class as a starting point for creating a custom data source loading class. To view the source code for this class, use this command:

```
edit vision.labeler.loading.RosbagSource
```

See Also

Apps
Ground Truth Labeler

Classes
`vision.labeler.loading.CustomImageSource` | `vision.labeler.loading.ImageSequenceSource` | `vision.labeler.loading.PointCloudSequenceSource` | `vision.labeler.loading.VelodyneLidarSource` | `vision.labeler.loading.VideoSource`

Topics
"Sources vs. Signals in Ground Truth Labeling"
"Create Class for Loading Custom Ground Truth Data Sources"

Introduced in R2020a

vision.labeler.loading.PointCloudSequenceSource class

Package: vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading
Superclasses: vision.labeler.loading.MultiSignalSource

Load signals from point cloud sequence sources into Ground Truth Labeler app

Description

The vision.labeler.loading.PointCloudSequenceSource class creates an interface for loading signals from point cloud sequence data sources into the **Ground Truth Labeler** app. In the Add/Remove Signal dialog box of the app, when **Source Type** is set to Point Cloud Sequence, this class controls the parameters in that dialog box.

To access this dialog box, in the app, select **Open** > **Add Signals**.

The default implementation of this class loads point cloud sequences composed of PCD or PLY files.

The vision.labeler.loading.PointCloudSequenceSource class is a handle class.

Creation

When you export labels from a **Ground Truth Labeler** app session that contains point cloud sequence sources, the exported groundTruthMultisignal object stores instances of this class in its DataSource property.

To create a PointCloudSequenceSource object programmatically, such as when programmatically creating a groundTruthMultisignal object, use the vision.labeler.loading.PointCloudSequenceSource function (described here).

Syntax

```
pcseqSource = vision.labeler.loading.PointCloudSequenceSource
```

Description

`pcseqSource = vision.labeler.loading.PointCloudSequenceSource` creates a `PointCloudSequenceSource` object for loading signals from point cloud sequence data sources. To specify the data source and the parameters required to load the source, use the `loadSource` method.

Properties

`Name` — Name of source type

`"Point Cloud Sequence"` (default) | string scalar

Name of the type of source that this class loads, specified as a string scalar.

Attributes:

GetAccess	public
Constant	true
NonCopyable	true

`Description` — Description of class functionality

`"A PointCloud sequence reader"` (default) | string scalar

Description of the functionality that this class provides, specified as a string scalar.

Attributes:

GetAccess	public
Constant	true
NonCopyable	true

`SourceName` — Name of data source

`[]` (default) | string scalar

Name of the data source, specified as a string scalar. Typically, `SourceName` is the name of the file from which the signal is loaded.

Attributes:

GetAccess	public
SetAccess	protected

`SourceParams` — Parameters for loading point cloud sequence signal from data source

`[]` (default) | structure

Parameters for loading a point cloud sequence signal from a data source, specified as a structure.

This table describes the required and optional fields of the `SourceParams` structure.

Field	Description	Required or Optional
Timestamps	Timestamps for the point cloud sequence signal, specified as a cell array containing a single duration vector of timestamps. In the Add/Remove Signal dialog box of the app, if you set the **Timestamps** parameter to From Workspace and read the timestamps from a variable in the MATLAB workspace, then the SourceParams property stores these timestamps in the Timestamps field.	Optional If you set the **Timestamps** parameter to Use Default and use the default timestamps for point cloud sequence signals, then the structure does not include this field, and the SourceParams property is empty, []. For point cloud sequence signals, the default timestamp duration vector has elements from 0 to the number of valid point cloud files minus 1. Units are in seconds.

Attributes:

GetAccess	public
SetAccess	protected

SignalName — Names of signals in data source
[] (default) | string vector

Names of the signals that can be loaded from the data source, specified as a string vector.

Attributes:

GetAccess	public
SetAccess	protected

SignalType — Types of signals in data source
[] (default) | vector of vision.labeler.loading.SignalType enumerations

Types of the signals that can be loaded from the data source, specified as a vector of vision.labeler.loading.SignalType enumerations. Each signal listed in the SignalName property is of the type in the corresponding position of SignalType.

Attributes:

GetAccess	public
SetAccess	protected

Timestamp — Timestamps of signals in data source
[] (default) | cell array of duration vectors

Timestamps of the signals that can be loaded from the data source, specified as a cell array of duration vectors. Each signal listed in the SignalName property has the timestamps in the corresponding position of Timestamp.

Attributes:

GetAccess	public
SetAccess	protected

`NumSignals` — Number of signals in data source
`0` (default) | integer

Number of signals that can be read from the data source, specified as a nonnegative integer. `NumSignals` is equal to the number of signals in the `SignalName` property.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`
`NonCopyable`	`true`

Methods

Public Methods

`customizeLoadPanel`	`customizeLoadPanel(sourceObj,panel)` Customize the loading panel for the data source object. In the loading dialog box of the app, this method is invoked when you select the data source type from the **Source Type** list.
`getLoadPanelData`	`[sourceName,sourceParams] = getLoadPanelData(sourceObj)` Obtain the data needed to load the data source object currently selected in the loading panel. In the loading dialog box of the app, this method is invoked when you add a source. The method returns these outputs. • `sourceName` is a string capturing the name of the data source object. • `sourceParams` is a structure with fields containing the parameters required to load the data source object. Both of these outputs are passed to the `loadSource` method.
`loadSource`	`loadSource(sourceObj,sourceName,sourceParams)` Load a data source object into the app. In the loading dialog box of the app, this method is invoked after you add a source and the `getLoadPanelData` method executes successfully. This method is also invoked when you load the data source object into the MATLAB workspace. When you load the data source object, MATLAB expects that the source has the name `sourceName` and parameters `sourceParams` that are needed to load that source and read data from it.

`readFrame`	`frame = readFrame(sourceObj,signalName,tsIndex)` Read a frame of data from a signal contained in a data source object at the specified timestamp index. The index must be in the bounds of the length of the timestamps for that signal.
`loadPanelChecker`	`loadPanelChecker` Check the load panel for the loading dialog box of the app. This method opens a dialog box similar to the loading dialog box that you open from the **Open** menu on the app toolstrip. Use this method to preview how the `customizeLoadPanel` method populates the loading panel for the selected data source object. Static: true

Examples

Create Point Cloud Sequence Source

Specify the path to a folder containing a point cloud sequence.

```
pcSeqFolder = fullfile(toolboxdir('driving'),'drivingdata',...
    'lidarSequence');
```

Load the timestamps that correspond to the sequence.

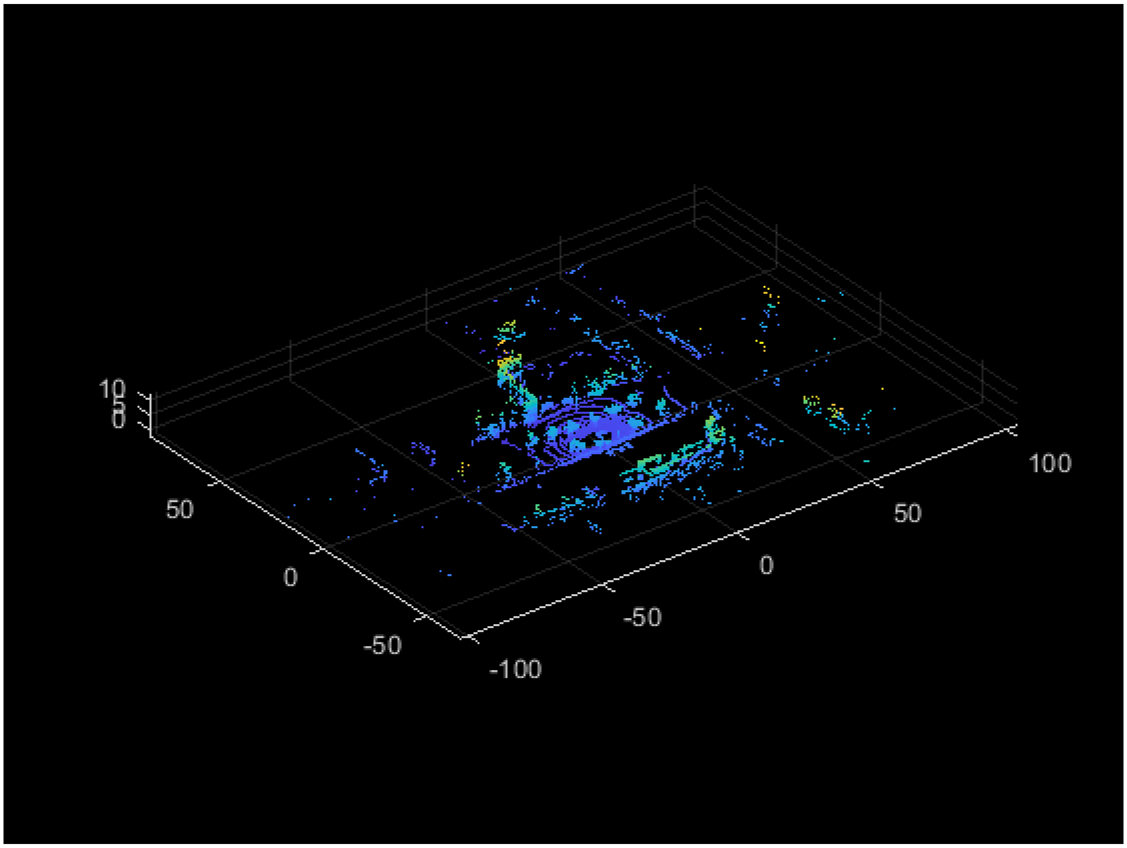
```
load(fullfile(pcSeqFolder,'timestamps.mat'));
```

Create a point cloud sequence source. Load the folder path and timestamps into the `PointCloudSequenceSource` object.

```
sourceName = pcSeqFolder;
sourceParams = struct;
sourceParams.Timestamps = timestamps;

pcseqSource = vision.labeler.loading.PointCloudSequenceSource;
loadSource(pcseqSource,sourceName,sourceParams);
```

Read the first frame in the sequence. Display the frame.

```
signalName = pcseqSource.SignalName;
pc = readFrame(pcseqSource,signalName,1);

figure
pcshow(pc)
```



Tips

- You can this class as a starting point for creating a custom data source loading class. To view the source code for this class, use this command:

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

See Also

Apps
Ground Truth Labeler

Classes
`vision.labeler.loading.CustomImageSource` | `vision.labeler.loading.ImageSequenceSource` | `vision.labeler.loading.RosbagSource` | `vision.labeler.loading.VelodyneLidarSource` | `vision.labeler.loading.VideoSource`

Topics
"Sources vs. Signals in Ground Truth Labeling"
"Create Class for Loading Custom Ground Truth Data Sources"

Introduced in R2020a

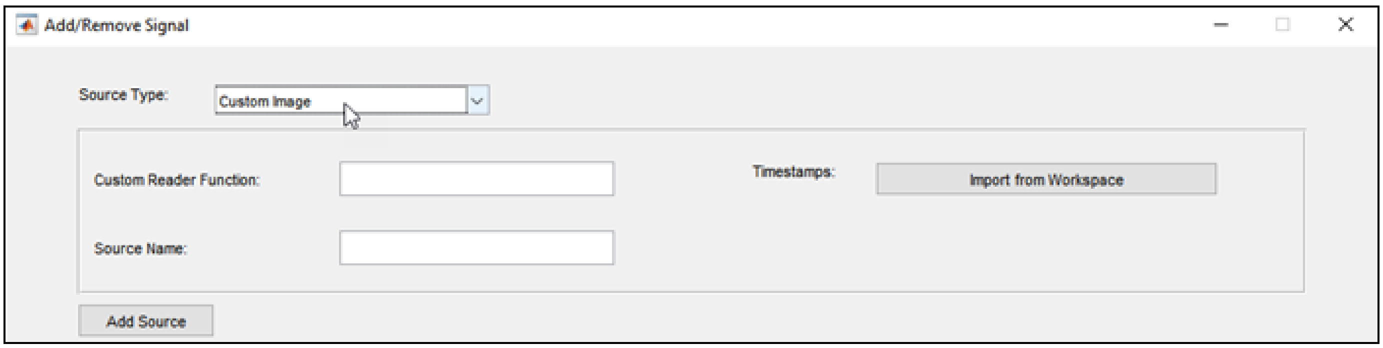
vision.labeler.loading.CustomImageSource class

Package: vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading
Superclasses: vision.labeler.loading.MultiSignalSource

Load signals from custom image sources into Ground Truth Labeler app

Description

The vision.labeler.loading.CustomImageSource class creates an interface for loading signals from custom image data sources into the **Ground Truth Labeler** app. In the Add/Remove Signal dialog box of the app, when **Source Type** is set to Custom Image, this class controls the parameters in that dialog box.



To access this dialog box, in the app, select **Open** > **Add Signals**.

The vision.labeler.loading.CustomImageSource class is a handle class.

Creation

When you export labels from a **Ground Truth Labeler** app session that contains custom image sources, the exported groundTruthMultisignal object stores instances of this class in its DataSource property.

To create a CustomImageSource object programmatically, such as when programmatically creating a groundTruthMultisignal object, use the vision.labeler.loading.CustomImageSource function (described here).

Syntax

customImgSource = vision.labeler.loading.CustomImageSource

Description

customImgSource = vision.labeler.loading.CustomImageSource creates a CustomImageSource object for loading signals from custom image data sources. To specify the data source and the parameters required to load the source, use the loadSource method.

Properties

`Name` — Name of source type
`"Custom Image"` (default) | string scalar

Name of the type of source that this class loads, specified as a string scalar.

Attributes:

`GetAccess`	`public`
`Constant`	`true`
`NonCopyable`	`true`

`Description` — Description of class functionality
`"A custom image source reader"` (default) | string scalar

Description of the functionality that this class provides, specified as a string scalar.

Attributes:

`GetAccess`	`public`
`Constant`	`true`
`NonCopyable`	`true`

`SourceName` — Name of data source
`[]` (default) | string scalar

Name of the data source, specified as a string scalar. Typically, `SourceName` is the name of the file from which the signal is loaded.

Attributes:

`GetAccess`	`public`
`SetAccess`	`protected`

`SourceParams` — Parameters for loading custom image signal from data source
`[]` (default) | structure

Parameters for loading a custom image signal from a data source, specified as a structure.

This table describes the required and optional fields of the `SourceParams` structure.

Field	Description	Required or Optional
`FunctionHandle`	Custom reader function for reading images from the data source, specified as a function handle. In the Add/Remove Signal dialog box of the app, specify this function handle in the **Custom Reader Function** parameter. For details on creating a custom reader function, see "Use Custom Image Source Reader for Labeling" (Computer Vision Toolbox).	Required
`Timestamps`	Timestamps for the custom image signal, specified as a cell array containing a single `duration` vector of timestamps. (For data sources that contain multiple signals, the `Timestamps` cell array contains one `duration` vector per signal with timestamps that are loaded from the MATLAB workspace.) In the Add/Remove Signal dialog box of the app, when you click the **Import from Workspace** button to read the timestamps from a variable in the MATLAB workspace, then the `SourceParams` property stores these timestamps in the `Timestamps` field.	Required

Attributes:

GetAccess	public
SetAccess	protected

`SignalName` — Names of signals in data source
[ ] (default) | string vector

Names of the signals that can be loaded from the data source, specified as a string vector.

Attributes:

GetAccess	public
SetAccess	protected

`SignalType` — Types of signals in data source
[ ] (default) | vector of `vision.labeler.loading.SignalType` enumerations

Types of the signals that can be loaded from the data source, specified as a vector of `vision.labeler.loading.SignalType` enumerations. Each signal listed in the `SignalName` property is of the type in the corresponding position of `SignalType`.

Attributes:

GetAccess	public
SetAccess	protected

`Timestamp` — Timestamps of signals in data source
`[]` (default) | cell array of `duration` vectors

Timestamps of the signals that can be loaded from the data source, specified as a cell array of `duration` vectors. Each signal listed in the `SignalName` property has the timestamps in the corresponding position of `Timestamp`.

Attributes:

GetAccess	public
SetAccess	protected

`NumSignals` — Number of signals in data source
`0` (default) | integer

Number of signals that can be read from the data source, specified as a nonnegative integer. `NumSignals` is equal to the number of signals in the `SignalName` property.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true
NonCopyable	true

Methods

Public Methods

customizeLoadPanel	`customizeLoadPanel(sourceObj,panel)` Customize the loading panel for the data source object. In the loading dialog box of the app, this method is invoked when you select the data source type from the **Source Type** list.

`getLoadPanelData`	`[sourceName,sourceParams] = getLoadPanelData(sourceObj)` Obtain the data needed to load the data source object currently selected in the loading panel. In the loading dialog box of the app, this method is invoked when you add a source. The method returns these outputs. • `sourceName` is a string capturing the name of the data source object. • `sourceParams` is a structure with fields containing the parameters required to load the data source object. Both of these outputs are passed to the `loadSource` method.
`loadSource`	`loadSource(sourceObj,sourceName,sourceParams)` Load a data source object into the app. In the loading dialog box of the app, this method is invoked after you add a source and the `getLoadPanelData` method executes successfully. This method is also invoked when you load the data source object into the MATLAB workspace. When you load the data source object, MATLAB expects that the source has the name `sourceName` and parameters `sourceParams` that are needed to load that source and read data from it.
`readFrame`	`frame  = readFrame(sourceObj,signalName,tsIndex)` Read a frame of data from a signal contained in a data source object at the specified timestamp index. The index must be in the bounds of the length of the timestamps for that signal.
`loadPanelChecker`	`loadPanelChecker` Check the load panel for the loading dialog box of the app. This method opens a dialog box similar to the loading dialog box that you open from the **Open** menu on the app toolstrip. Use this method to preview how the `customizeLoadPanel` method populates the loading panel for the selected data source object. Static: true

Examples

Create Custom Image Source

Specify the path to a folder containing a sequence of road images.

```
imageFolder = fullfile(toolboxdir('driving'),'drivingdata','roadSequence');
```

Store the images in an image datastore. The **Ground Truth Labeler** app and `groundTruthMultisignal` object do not natively support image datastores, so it is considered a custom image data source.

```
imds = imageDatastore(imageFolder);
```

Write a reader function, `readerFcn`, to read images from the datastore. The first input argument to the reader function, `sourceName`, is not used. The second input argument, `currentTimestamp`, is converted from a `duration` scalar to a 1-based index. This format is compatible with reading images from the datastore.

```
readerFcn = @(~,idx)readimage(imds,seconds(idx));
```

Create a custom image source. Load the source name, reader function, and first five timestamps of the datastore into the `CustomImageSource` object.

```
sourceName = imageFolder;
sourceParams = struct();
sourceParams.FunctionHandle = readerFcn;
sourceParams.Timestamps = seconds(1:5);
customImgSource = vision.labeler.loading.CustomImageSource;
loadSource(customImgSource,sourceName,sourceParams)
```

Read the first frame in the sequence. Display the frame.

```
signalName = customImgSource.SignalName;
I = readFrame(customImgSource,signalName,1);
figure
imshow(I)
```

Tips

- You can this class as a starting point for creating a custom data source loading class. To view the source code for this class, use this command:

```
edit vision.labeler.loading.CustomImageSource
```

See Also

Apps
Ground Truth Labeler

Classes
`vision.labeler.loading.ImageSequenceSource` | `vision.labeler.loading.PointCloudSequenceSource` | `vision.labeler.loading.RosbagSource` | `vision.labeler.loading.VelodyneLidarSource` | `vision.labeler.loading.VideoSource`

Topics
"Sources vs. Signals in Ground Truth Labeling"

"Create Class for Loading Custom Ground Truth Data Sources"

Introduced in R2020a

vision.labeler.loading.SignalType

Signal type enumerations for labeling

Description

The `vision.labeler.loading.SignalType` enumerations enable you to specify the types of signals used in the **Ground Truth Labeler** app. When selecting signals from a `groundTruthMultisignal` object by using the `selectLabelsBySignalType` function, use these enumerations to select labels of a specific signal type.

Creation

Syntax

```
vision.labeler.loading.SignalType.Image
vision.labeler.loading.SignalType.PointCloud
vision.labeler.loading.SignalType.Time
```

Description

`vision.labeler.loading.SignalType.Image` creates an enumeration of signal type `Image`. Use this enumeration to specify image signals obtained from sources such as videos or image sequences.

`vision.labeler.loading.SignalType.PointCloud` creates an enumeration of signal type `PointCloud`. Use this enumeration to specify lidar point cloud signals obtained from sources such as Velodyne packet capture (PCAP) files.

`vision.labeler.loading.SignalType.Time` creates an enumeration of signal type `Time`. Scene labels are `Time` signals and are of type `duration`. You cannot load `Time` signals into the **Ground Truth Labeler** app.

Examples

Select Ground Truth Labels by Signal Type

Select ground truth labels from a `groundTruthMultisignal` object by specifying a signal type.

Load a `groundTruthMultisignal` object containing ROI and scene label data for a video and corresponding lidar point cloud sequence. The helper function used to load this object is attached to the example as a supporting file.

```
gTruth = helperLoadGTruthVideoLidar;
```

Inspect the label definitions. The object contains definitions for image, point cloud, and time signals.

```
gTruth.LabelDefinitions
```

```
ans =

  5x7 table

      Name       SignalType    LabelType       Group         Description     LabelColor      Hiera
    _________    __________    _________    _____________    ___________    _____________    _____

    {'car'  }    Image         Rectangle    {'Vehicles'}     {0x0 char}     {1x3 double}     {1x1 s
    {'car'  }    PointCloud    Cuboid       {'Vehicles'}     {0x0 char}     {1x3 double}     {1x1 s
    {'truck'}    Image         Rectangle    {'Vehicles'}     {0x0 char}     {1x3 double}     {0x0 d
    {'truck'}    PointCloud    Cuboid       {'Vehicles'}     {0x0 char}     {1x3 double}     {0x0 d
    {'sunny'}    Time          Scene        {'None'    }     {0x0 char}     {1x3 double}     {0x0 d
```

Inspect the ROI labels. The object contains labels for the lidar point cloud sequence and the video.

```
gTruth.ROILabelData
```

```
ans =

  ROILabelData with properties:

    video_01_city_c2s_fcw_10s: [204x2 timetable]
                lidarSequence: [34x2 timetable]
```

Create a new `groundTruthMultisignal` object that contains labels for only point cloud signals.

```
signalTypes = vision.labeler.loading.SignalType.PointCloud;
gtLabel = selectLabelsBySignalType(gTruth,signalTypes);
```

For the original and new objects, inspect the first five rows of label data for the lidar point cloud sequence. Because lidar signals are of type `PointCloud`, the new object contains the same label data for the lidar sequence as the original object.

```
lidarLabels = gTruth.ROILabelData.lidarSequence;
lidarLabelsSelection = gtLabel.ROILabelData.lidarSequence;

numrows = 5;
head(lidarLabels,numrows)
head(lidarLabelsSelection,numrows)
```

```
ans =

  5x2 timetable

       Time           car             truck
    ___________    ____________    ____________

    0 sec          {1x1 struct}    {1x0 struct}
    0.29926 sec    {1x1 struct}    {1x0 struct}
    0.59997 sec    {1x1 struct}    {1x0 struct}
    0.8485 sec     {1x1 struct}    {1x0 struct}
    1.1484 sec     {1x1 struct}    {1x0 struct}
```

```
ans =

  5x2 timetable

       Time            car             truck
    ___________    ____________    ____________

    0 sec          {1x1 struct}    {1x0 struct}
    0.29926 sec    {1x1 struct}    {1x0 struct}
    0.59997 sec    {1x1 struct}    {1x0 struct}
    0.8485 sec     {1x1 struct}    {1x0 struct}
    1.1484 sec     {1x1 struct}    {1x0 struct}
```

For the original and new objects, inspect the first five rows of label data for the video. Because video signals are of type `Image`, the new object contains no label data for the video.

```
videoLabels = gTruth.ROILabelData.video_01_city_c2s_fcw_10s;
videoLabelsSelection = gtLabel.ROILabelData.video_01_city_c2s_fcw_10s;

head(videoLabels,numrows)
head(videoLabelsSelection,numrows)
```

```
ans =

  5x2 timetable

      Time          car             truck
    ________    ____________    ____________

    0 sec       {1x3 struct}    {1x0 struct}
    0.05 sec    {1x3 struct}    {1x0 struct}
    0.1 sec     {1x3 struct}    {1x0 struct}
    0.15 sec    {1x3 struct}    {1x0 struct}
    0.2 sec     {1x3 struct}    {1x0 struct}


ans =

  5x0 empty timetable
```

See Also

Apps
Ground Truth Labeler

Objects
`attributeType` | `groundTruthMultisignal` | `labelDefinitionCreatorMultisignal` | `labelType`

Functions
`selectLabelsBySignalType`

Topics
"Create Class for Loading Custom Ground Truth Data Sources"

Introduced in R2020a

sim3d.Editor

Interface to the Unreal Engine project

Description

Use the `sim3d.Editor` class to interface with the Unreal Editor.

To develop scenes with the Unreal Editor and co-simulate with Simulink, you need the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package. The support package contains an Unreal Engine project that allows you to customize the Automated Driving Toolbox scenes. For information about the support package, see “Customize Unreal Engine Scenes for Automated Driving”.

Creation

Syntax

`sim3d.Editor(project)`

Description

MATLAB creates an `sim3d.Editor` object for the Unreal Editor project specified in `sim3d.Editor( project)`.

Input Arguments

`project` — Project path and name
string array

Project path and name.

Example: `"C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject"`

Data Types: `string`

Properties

`Uproject` — Project path and name
string array

This property is read-only.

Project path and name with Unreal Engine project file extension.

Example: `"C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject"`

Data Types: `string`

Object Functions

open   Open the Unreal Editor

Examples

Open Project in Unreal Editor

Open an Unreal Engine project in the Unreal Editor.

Create an instance of the `sim3d.Editor` class for the Unreal Engine project located in `C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject`.

```
editor=sim3d.Editor(fullfile("C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject"))
```

Open the project in the Unreal Editor.

```
editor.open();
```

See Also

Topics
"Customize Unreal Engine Scenes for Automated Driving"

Introduced in R2020a

open

Open the Unreal Editor

Syntax

`[status,result]=open(sim3dEditorObj)`

Description

`[status,result]=open(sim3dEditorObj)` opens the Unreal Engine project in the Unreal Editor.

To develop scenes with the Unreal Editor and co-simulate with Simulink, you need the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package. The support package contains an Unreal Engine project that allows you to customize the Automated Driving Toolbox scenes. For information about the support package, see “Customize Unreal Engine Scenes for Automated Driving”.

Input Arguments

`sim3dEditorObj` — sim3d.Editor object

`sim3d.Editor` object

`sim3d.Editor` object for the Unreal Engine project.

Output Arguments

`status` — Command exit status

0 | nonzero integer

Command exit status, returned as either `0` or a nonzero integer. When the command is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

- If `command` includes the ampersand character (&), then `status` is the exit status when `command` starts
- If `command` does not include the ampersand character (&), then `status` is the exit status upon `command` completion.

`result` — Output of operating system command

character vector

Output of the operating system command, returned as a character vector. The system shell might not properly represent non-Unicode® characters.

See Also

`sim3d.Editor`

Topics

“Customize Unreal Engine Scenes for Automated Driving”

Introduced in R2020a

multiObjectTracker

Track objects using GNN assignment

Description

The multiObjectTracker System object initializes, confirms, predicts, corrects, and deletes the tracks of moving objects. Inputs to the multi-object tracker are detection reports generated by an objectDetection object, radarDetectionGenerator object, or visionDetectionGenerator object. The multi-object tracker accepts detections from multiple sensors and assigns them to tracks using a global nearest neighbor (GNN) criterion. Each detection is assigned to a separate track. If the detection cannot be assigned to any track, based on the AssignmentThreshold property, the tracker creates a new track. The tracks are returned in a structure array.

A new track starts in a *tentative* state. If enough detections are assigned to a tentative track, its status changes to *confirmed*. If the detection is a known classification (the ObjectClassID field of the returned track is nonzero), that track can be confirmed immediately. For details on the multi-object tracker properties used to confirm tracks, see "Algorithms" on page 4-819.

When a track is confirmed, the multi-object tracker considers that track to represent a physical object. If detections are not added to the track within a specifiable number of updates, the track is deleted.

The tracker also estimates the state vector and state vector covariance matrix for each track using a Kalman filter. These state vectors are used to predict a track's location in each frame and determine the likelihood of each detection being assigned to each track.

To track objects using a multi-object tracker:

1. Create the multiObjectTracker object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
tracker = multiObjectTracker
tracker = multiObjectTracker(Name,Value)
```

Description

tracker = multiObjectTracker creates a multiObjectTracker System object with default property values.

tracker = multiObjectTracker(Name,Value) sets properties on page 4-809 for the multi-object tracker using one or more name-value pairs. For example, multiObjectTracker('FilterInitializationFcn',@initcvukf,'MaxNumTracks',100) creates a multi-object tracker that uses a constant-velocity, unscented Kalman filter and maintains a maximum of 100 tracks. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`TrackerIndex` — Unique tracker identifier
`0` (default) | nonnegative integer

Unique tracker identifier, specified as a nonnegative integer. This property is used as the `SourceIndex` in the tracker outputs, and distinguishes tracks that come from different trackers in a multiple-tracker system. You must specify this property as a positive integer to use the track outputs as inputs to a track fuser.

Example: `1`

`FilterInitializationFcn` — Kalman filter initialization function
`@initcvkf` (default) | function handle | character vector | string scalar

Kalman filter initialization function, specified as a function handle or as a character vector or string scalar of the name of a valid Kalman filter initialization function.

Automated Driving Toolbox supplies several initialization functions that you can use to specify `FilterInitializationFcn`.

Initialization Function	Function Definition
`initcvekf`	Initialize constant-velocity extended Kalman filter.
`initcvkf`	Initialize constant-velocity linear Kalman filter.
`initcvukf`	Initialize constant-velocity unscented Kalman filter.
`initcaekf`	Initialize constant-acceleration extended Kalman filter.
`initcakf`	Initialize constant-acceleration linear Kalman filter.
`initcaukf`	Initialize constant-acceleration unscented Kalman filter.
`initctekf`	Initialize constant-turnrate extended Kalman filter.
`initctukf`	Initialize constant-turnrate unscented Kalman filter.

You can also write your own initialization function. The input to this function must be a detection report created by `objectDetection`. The output of this function must be a Kalman filter object:

trackingKF, trackingEKF, or trackingUKF. To guide you in writing this function, you can examine the details of the supplied functions from within MATLAB. For example:

```
type initcvkf
```

Data Types: function_handle | char | string

AssignmentThreshold — Detection assignment threshold
30*[1 Inf] (default) | positive scalar | 1-by-2 vector of positive values

Detection assignment threshold (or gating threshold), specified as a positive scalar or an 1-by-2 vector of [C_1,C_2], where $C_1 \leq C_2$. If specified as a scalar, the specified value, *val*, will be expanded to [*val*, Inf].

Initially, the tracker executes a *coarse* estimation for the normalized distance between all the tracks and detections. The tracker only calculates the accurate normalized distance for the combinations whose coarse normalized distance is less than C_2. Also, the tracker can only assign a detection to a track if their *accurate* normalized distance is less than C_1. See the distance function used with tracking filters (for example, trackingEKF) for an explanation of the distance calculation.

Tips:

- Increase the value of C_2 if there are combinations of track and detection that should be calculated for assignment but are not. Decrease it if cost calculation takes too much time.
- Increase the value of C_1 if there are detections that should be assigned to tracks but are not. Decrease it if there are detections that are assigned to tracks they should not be assigned to (too far away).

MaxNumTracks — Maximum number of tracks
200 (default) | positive integer

Maximum number of tracks that the tracker can maintain, specified as a positive integer.

Data Types: double

MaxNumSensors — Maximum number of sensors
20 (default) | positive integer

Maximum number of sensors that can be connected to the tracker, specified as a positive integer. When you specify detections as input to the multi-object tracker, MaxNumSensors must be greater than or equal to the highest SensorIndex value in the detections cell array of objectDetection objects used to update the multi-object tracker. This property determines how many sets of ObjectAttributes fields each output track can have.

Data Types: double

ConfirmationThreshold — Threshold for track confirmation
[2 3] (default) | two-element vector of non-decreasing positive integers

Threshold for track confirmation, specified as a two-element vector of non-decreasing positive integers, [M N], where M is less than or equal to N. A track is confirmed if it receives at least M detections in the last N updates.

- When setting M, take into account the probability of object detection for the sensors. The probability of detection depends on factors such as occlusion or clutter. You can reduce M when tracks fail to be confirmed or increase M when too many false detections are assigned to tracks.

- When setting `N`, consider the number of times you want the tracker to update before it makes a confirmation decision. For example, if a tracker updates every 0.05 seconds, and you allow 0.5 seconds to make a confirmation decision, set `N = 10`.

Example: `[3 5]`

Data Types: `double`

`DeletionThreshold` — Threshold for track deletion
`[5 5]` (default) | two-element vector of positive non-decreasing integers

Threshold for track deletion, specified as a two-element vector of positive non-decreasing integers `[P Q]`, where `P` is less than or equal to `Q`. If a confirmed track is not assigned to any detection `P` times in the last `Q` tracker updates, then the track is deleted.

- Decrease `Q` (or increase `P`) if tracks should be deleted earlier.
- Increase `Q` (or decrease `P`) if tracks should be kept for a longer time before deletion.

Example: `[3 5]`

Data Types: `single` | `double`

`HasCostMatrixInput` — Enable cost matrix input
`false` (default) | `true`

Enable a cost matrix as input to the `multiObjectTracker` System object or to the `updateTracks` function, specified as `false` or `true`.

Data Types: `logical`

`HasDetectableTrackIDsInput` — Enable input of detectable track IDs
`false` (default) | `true`

Enable the input of detectable track IDs at each object update, specified as `false` or `true`. Set this property to `true` if you want to provide a list of detectable track IDs. This list tells the tracker of all tracks that the sensors are expected to detect and, optionally, the probability of detection for each track.

Data Types: `logical`

`StateParameters` — Parameters of the track state reference frame
`struct([])` (default) | `struct` | `struct array`

Parameters of the track state reference frame, specified as a struct or a struct array. Use this property to define the track state reference frame and how to transform the track from the tracker (called source) coordinate system to the fuser coordinate system.

This property is tunable.

Data Types: `struct`

`NumTracks` — Number of tracks maintained by multi-object tracker
nonnegative integer

This property is read-only.

Number of tracks maintained by the multi-object tracker, specified as a nonnegative integer.

Data Types: double

NumConfirmedTracks — Number of confirmed tracks
nonnegative integer

This property is read-only.

Number of confirmed tracks, specified as a nonnegative integer. The IsConfirmed fields of the output track structures indicate which tracks are confirmed.

Data Types: double

Usage

To update tracks, call the created multi-object tracker with arguments, as if it were a function (described here). Alternatively, update tracks by using the updateTracks function, specifying the multi-object tracker as an input argument.

Syntax

```
confirmedTracks = tracker(detections,time)
[confirmedTracks,tentativeTracks] = tracker(detections,time)
[confirmedTracks,tentativeTracks,allTracks] = tracker(detections,time)
[ ___ ] = tracker(detections,time,costMatrix)
[ ___ ] = tracker( ___ ,detectableTrackIDs)
```

Description

confirmedTracks = tracker(detections,time) creates, updates, and deletes tracks in the multi-object tracker and returns details about the confirmed tracks. Updates are based on the specified list of detections, and all tracks are updated to the specified time. Each element in the returned confirmedTracks corresponds to a single track.

[confirmedTracks,tentativeTracks] = tracker(detections,time) also returns tentativeTracks containing details about the tentative tracks.

[confirmedTracks,tentativeTracks,allTracks] = tracker(detections,time) also returns allTracks containing details about all the confirmed and tentative tracks. The tracks are returned in the order by which the tracker internally maintains them. You can use this output to help you calculate the cost matrix, an optional input argument.

[___] = tracker(detections,time,costMatrix) specifies a cost matrix, returning any of the outputs from preceding syntaxes.

To specify a cost matrix, set the HasCostMatrixInput property of the multiObjectTracker System object to true.

[___] = tracker(___ ,detectableTrackIDs) also specifies a list of expected detectable tracks given by detectableTrackIDs. This argument can be used with any of the previous input syntaxes.

To enable this syntax, set the HasDetectableTrackIDsInput property to true.

Input Arguments

detections — Detection list
cell array of objectDetection objects

Detection list, specified as a cell array of objectDetection objects. The Time property value of each objectDetection object must be less than or equal to the current time of update, time, and greater than the previous time value used to update the multi-object tracker.

time — Time of update
real scalar

Time of update, specified as a real scalar. The multi-object tracker updates all tracks to this time. Units are in seconds.

time must be greater than or equal to the largest Time property value of the objectDetection objects in the input detections list. time must increase in value with each update to the multi-object tracker.

Data Types: double

costMatrix — Cost matrix
N_T-by-N_D matrix

Cost matrix, specified as a real-valued N_T-by-N_D matrix, where N_T is the number of existing tracks, and N_D is the number of current detections. The rows of the cost matrix correspond to the existing tracks. The columns correspond to the detections. Tracks are ordered as they appear in the list of tracks in the allTracks output argument of the previous update to the multi-object tracker.

In the first update to the multi-object tracker, or when the multi-object tracker has no previous tracks, assign the cost matrix a size of [0, N_D]. The cost must be calculated so that lower costs indicate a higher likelihood that the multi-object tracker assigns a detection to a track. To prevent certain detections from being assigned to certain tracks, use Inf.

Dependencies

To enable specification of the cost matrix when updating tracks, set the HasCostMatrixInput property of the multi-object tracker to true

Data Types: double

detectableTrackIDs — Detectable track IDs
real-valued M-by-1 vector | real-valued M-by-2 matrix

Detectable track IDs, specified as a real-valued M-by-1 vector or M-by-2 matrix. Detectable tracks are tracks that the sensors expect to detect. The first column of the matrix contains a list of track IDs that the sensors report as detectable. The optional second column contains the detection probability for the track. The detection probability is either reported by a sensor or, if not reported, obtained from the DetectionProbability property.

Tracks whose identifiers are not included in detectableTrackIDs are considered as undetectable. The track deletion logic does not count the lack of detection as a 'missed detection' for track deletion purposes.

Dependencies

To enable this input argument, set the detectableTrackIDs property to true.

Data Types: `single` | `double`

Output Arguments

`confirmedTracks` — Confirmed tracks
array of `objectTrack` objects | array of structures

Confirmed tracks, returned as an array of `objectTrack` objects in MATLAB, and returned as an array of structures in code generation. In code generation, the field names of the returned structure are same with the property names of `objectTrack`.

A track is confirmed if it satisfies the confirmation threshold specified in the `ConfirmationThreshold` property. In that case, the `IsConfirmed` property of the object or field of the structure is `true`.

Data Types: `struct` | `object`

`tentativeTracks` — Tentative tracks
array of `objectTrack` objects | array of structures

Tentative tracks, returned as an array of `objectTrack` objects in MATLAB, and returned as an array of structures in code generation. In code generation, the field names of the returned structure are same with the property names of `objectTrack`.

A track is tentative if it does not satisfy the confirmation threshold specified in the `ConfirmationThreshold` property. In that case, the `IsConfirmed` property of the object or field of the structure is `false`.

Data Types: `struct` | `object`

`allTracks` — All tracks
array of `objectTrack` objects | array of structures

All tracks, returned as an array of `objectTrack` objects in MATLAB, and returned as an array of structures in code generation. In code generation, the field names of the returned structure are same with the property names of `objectTrack`. All tracks consists of confirmed and tentative tracks.

Data Types: `struct` | `object`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `multiObjectTracker`

updateTracks	Update multi-object tracker with new detections
initializeTrack	Initialize new track
deleteTrack	Delete existing track
getTrackFilterProperties	Obtain filter properties of track from multi-object tracker
setTrackFilterProperties	Set filter properties of track from multi-object tracker
predictTracksToTime	Predict track state

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
clone	Create duplicate System object
isLocked	Determine if System object is in use
reset	Reset internal states of System object

Examples

Track Single Object Using Multi-Object Tracker

Create a `multiObjectTracker` System object™ using the default filter initialization function for a 2-D constant-velocity model. For this motion model, the state vector is [*x*;*vx*;*y*;*vy*].

```
tracker = multiObjectTracker('ConfirmationThreshold',[4 5], ...
    'DeletionThreshold',10);
```

Create a detection by specifying an `objectDetection` object. To use this detection with the multi-object tracker, enclose the detection in a cell array.

```
dettime = 1.0;
det = { ...
    objectDetection(dettime,[10; -1], ...
    'SensorIndex',1, ...
    'ObjectAttributes',{'ExampleObject',1}) ...
    };
```

Update the multi-object tracker with this detection. The time at which you update the multi-object tacker must be greater than or equal to the time at which the object was detected.

```
updatetime = 1.25;
[confirmedTracks,tentativeTracks,allTracks] = tracker(det,updatetime);
```

Create another detection of the same object and update the multi-object tracker. The tracker maintains only one track.

```
dettime = 1.5;
det = { ...
    objectDetection(dettime,[10.1; -1.1], ...
    'SensorIndex',1, ...
    'ObjectAttributes',{'ExampleObject',1}) ...
    };
updatetime = 1.75;
[confirmedTracks,tentativeTracks,allTracks] = tracker(det,updatetime);
```

Determine whether the track has been verified by checking the number of confirmed tracks.

```
numConfirmed = tracker.NumConfirmedTracks
```

```
numConfirmed = 0
```

Examine the position and velocity of the tracked object. Because the track has not been confirmed, get the position and velocity from the `tentativeTracks` structure.

```
positionSelector = [1 0 0 0; 0 0 1 0];
velocitySelector = [0 1 0 0; 0 0 0 1];
position = getTrackPositions(tentativeTracks,positionSelector)
```

```
position = 1×2

   10.1426   -1.1426
```

```
velocity = getTrackVelocities(tentativeTracks,velocitySelector)
```

```
velocity = 1×2

    0.1852   -0.1852
```

Confirm and Delete Track in Multi-Object Tracker

Create a sequence of detections of a moving object. Track the detections using a `multiObjectTracker` System object™. Observe how the tracks switch from tentative to confirmed and then to deleted.

Create a multi-object tracker using the `initcakf` filter initialization function. The tracker models 2-D constant-acceleration motion. For this motion model, the state vector is [*x*;*vx*;*ax*;*y*;*vy*;*ay*].

```
tracker = multiObjectTracker('FilterInitializationFcn',@initcakf, ...
    'ConfirmationThreshold',[3 4],'DeletionThreshold',[6 6]);
```

Create a sequence of detections of a moving target using `objectDetection`. To use these detections with the `multiObjectTracker`, enclose the detections in a cell array.

```
dt = 0.1;
pos = [10; -1];
vel = [10; 5];
for detno = 1:2
    time = (detno-1)*dt;
    det = { ...
        objectDetection(time,pos, ...
        'SensorIndex',1, ...
        'ObjectAttributes',{'ExampleObject',1}) ...
        };
    [confirmedTracks,tentativeTracks,allTracks] = tracker(det,time);
    pos = pos + vel*dt;
    meas = pos;
end
```

Verify that the track has not been confirmed yet by checking the number of confirmed tracks.

```
numConfirmed = tracker.NumConfirmedTracks
```

```
numConfirmed = 0
```

Because the track is not confirmed, get the position and velocity from the `tentativeTracks` structure.

```
positionSelector = [1 0 0 0 0 0; 0 0 0 1 0 0];
velocitySelector = [0 1 0 0 0 0; 0 0 0 0 1 0];
position = getTrackPositions(tentativeTracks,positionSelector)
```

```
position = 1×2

   10.6669   -0.6665
```

```
velocity = getTrackVelocities(tentativeTracks,velocitySelector)
```

```
velocity = 1×2

    3.3473    1.6737
```

Add more detections to confirm the track.

```
for detno = 3:5
    time = (detno-1)*dt;
    det = { ...
        objectDetection(time,pos, ...
        'SensorIndex',1, ...
        'ObjectAttributes',{'ExampleObject',1}) ...
        };
    [confirmedTracks,tentativeTracks,allTracks] = tracker(det,time);
    pos = pos + vel*dt;
    meas = pos;
end
```

Verify that the track has been confirmed, and display the position and velocity vectors for that track.

```
numConfirmed = tracker.NumConfirmedTracks
```

```
numConfirmed = 1
```

```
position = getTrackPositions(confirmedTracks,positionSelector)
```

```
position = 1×2

   13.8417    0.9208
```

```
velocity = getTrackVelocities(confirmedTracks,velocitySelector)
```

```
velocity = 1×2

    9.4670    4.7335
```

Let the tracker run but do not add new detections. The existing track is deleted.

```
for detno = 6:20
    time = (detno-1)*dt;
    det = {};
    [confirmedTracks,tentativeTracks,allTracks] = tracker(det,time);
    pos = pos + vel*dt;
    meas = pos;
end
```

Verify that the tracker has no tentative or confirmed tracks.

```
isempty(allTracks)
```

```
ans = logical
   1
```

Generate Radar Detections of Multiple Vehicles

Generate detections using a forward-facing automotive radar mounted on an ego vehicle. Assume that there are three targets:

- Vehicle 1 is in the center lane, directly in front of the ego vehicle, and driving at the same speed.
- Vehicle 2 is in the left lane and driving faster than the ego vehicle by 12 kilometers per hour.
- Vehicle 3 is in the right lane and driving slower than the ego vehicle by 5 kilometers per hour.

All positions, velocities, and measurements are relative to the ego vehicle. Run the simulation for ten steps.

```
dt = 0.1;
pos1 = [150 0 0];
pos2 = [160 10 0];
pos3 = [130 -10 0];
vel1 = [0 0 0];
vel2 = [12*1000/3600 0 0];
vel3 = [-5*1000/3600 0 0];
car1 = struct('ActorID',1,'Position',pos1,'Velocity',vel1);
car2 = struct('ActorID',2,'Position',pos2,'Velocity',vel2);
car3 = struct('ActorID',3,'Position',pos3,'Velocity',vel3);
```

Create an automotive radar sensor that is offset from the ego vehicle. By default, the sensor location is at (3.4,0) meters from the vehicle center and 0.2 meters above the ground plane. Turn off the range rate computation so that the radar sensor measures position only.

```
radar = radarDetectionGenerator('DetectionCoordinates','Sensor Cartesian', ...
    'MaxRange',200,'RangeResolution',10,'AzimuthResolution',10, ...
    'FieldOfView',[40 15],'UpdateInterval',dt,'HasRangeRate',false);
tracker = multiObjectTracker('FilterInitializationFcn',@initcvkf, ...
    'ConfirmationThreshold',[3 4],'DeletionThreshold',6);
```

Generate detections with the radar from the non-ego vehicles. The output detections form a cell array and can be passed directly in to the `multiObjectTracker`.

```
simTime = 0;
nsteps = 10;
for k = 1:nsteps
    dets = radar([car1 car2 car3],simTime);
    [confirmedTracks,tentativeTracks,allTracks] = tracker(dets,simTime);
```

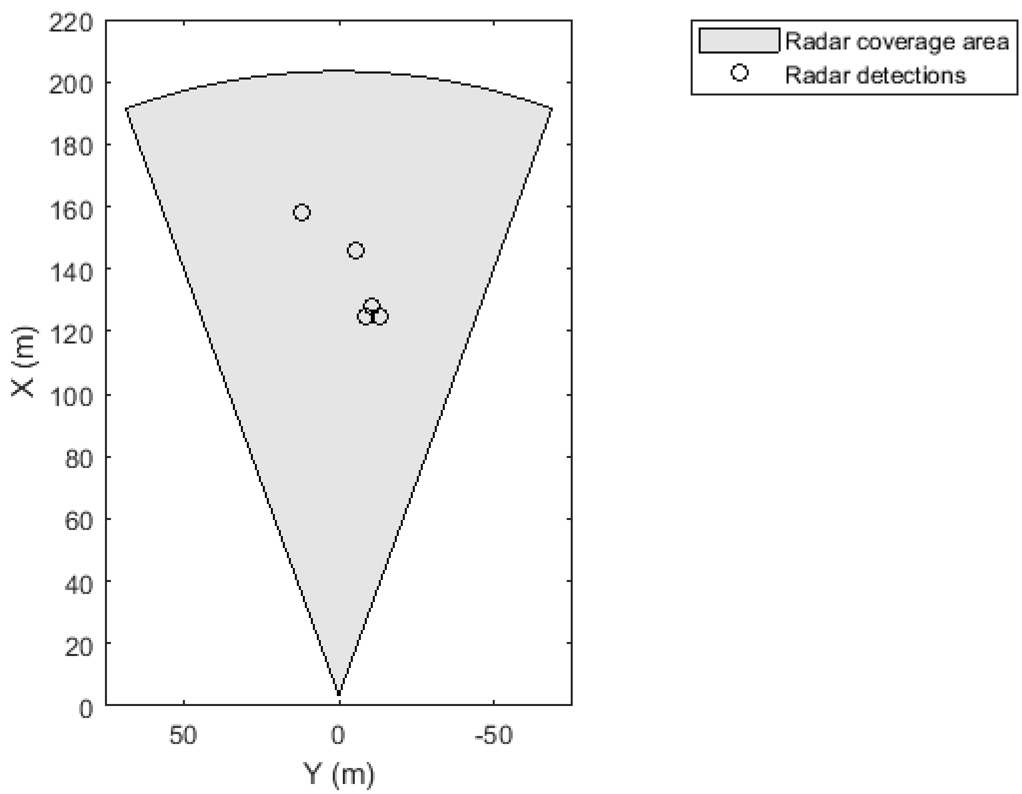
Move the cars one time step and update the multi-object tracker.

```
    simTime = simTime + dt;
    car1.Position = car1.Position + dt*car1.Velocity;
```

```
    car2.Position = car2.Position + dt*car2.Velocity;
    car3.Position = car3.Position + dt*car3.Velocity;
end
```

Use `birdsEyePlot` to create an overhead view of the detections. Plot the sensor coverage area. Extract the *X* and *Y* positions of the targets by converting the `Measurement` fields of the cell array into a MATLAB array. Display the detections on the bird's-eye plot.

```
BEplot = birdsEyePlot('XLim',[0 220],'YLim',[-75 75]);
caPlotter = coverageAreaPlotter(BEplot,'DisplayName','Radar coverage area');
plotCoverageArea(caPlotter,radar.SensorLocation,radar.MaxRange, ...
    radar.Yaw,radar.FieldOfView(1))
detPlotter = detectionPlotter(BEplot,'DisplayName','Radar detections');
detPos = cellfun(@(d)d.Measurement(1:2),dets,'UniformOutput',false);
detPos = cell2mat(detPos')';
if ~isempty(detPos)
    plotDetection(detPlotter,detPos)
end
```



Algorithms

When you pass detections into a multi-object tracker, the System object:

- Attempts to assign the input detections to existing tracks, based on the `AssignmentThreshold` property of the multi-object tracker.

- Creates new tracks from unassigned detections.
- Updates already assigned tracks and possibly confirms them, based on the `ConfirmationThreshold` property of the tracker.
- Deletes tracks that have no assigned detections, based on the `DeletionThreshold` property of the tracker.

Compatibility Considerations

Track output format changed
Behavior changed in R2020a

Starting from R2020a, the track output format of `multiObjectTracker` changes from track structure to `objectTrack`. As a result, when you load a `multiObjectTracker` created in an earlier version of MATLAB, you need to release the tracker first so that it can allow `objectTrack` as the track output format.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).
- All the detections used with a multi-object tracker must have properties with the same sizes and types.
- If you use the `ObjectAttributes` field within an `objectDetection` object, you must specify this field as a cell containing a structure. The structure for all detections must have the same fields and the values in these fields must always have the same size and type. The form of the structure cannot change during simulation.
- If `ObjectAttributes` are contained in the detection, the `SensorIndex` value of the detection cannot be greater than 10.
- The first update to the multi-object tracker must contain at least one detection.

See Also

Functions
`assignDetectionsToTracks` | `getTrackPositions` | `getTrackVelocities`

Objects
`drivingScenario` | `objectDetection` | `radarDetectionGenerator` | `trackingEKF` | `trackingKF` | `trackingUKF` | `visionDetectionGenerator`

Topics
“Multiple Object Tracking Tutorial”
“Track Multiple Vehicles Using a Camera”

Introduced in R2017a

deleteTrack

Delete existing track

Syntax

`deleted = deleteTrack(tracker,trackID)`

Description

`deleted = deleteTrack(tracker,trackID)` deletes the track specified by `trackID` in the `tracker`.

Examples

Delete track in `multiObjectTracker`

Create a track using detections in a `multiObjectTracker` tracker.

```
tracker = multiObjectTracker
```

```
tracker = 
  multiObjectTracker with properties:

                TrackerIndex: 0
     FilterInitializationFcn: 'initcvkf'
         AssignmentThreshold: [30 Inf]
                MaxNumTracks: 200
               MaxNumSensors: 20

       ConfirmationThreshold: [2 3]
           DeletionThreshold: [5 5]

          HasCostMatrixInput: false
  HasDetectableTrackIDsInput: false
             StateParameters: [1x1 struct]

                   NumTracks: 0
          NumConfirmedTracks: 0
```

```
detection1 = objectDetection(0,[1;1;1]);
detection2 = objectDetection(1,[1.1;1.2;1.1]);
tracker(detection1,0);
tracker(detection2,1)
```

```
ans = 
  objectTrack with properties:

           TrackID: 1
          BranchID: 0
       SourceIndex: 0
```

```
          UpdateTime: 1
                 Age: 2
               State: [6x1 double]
     StateCovariance: [6x6 double]
     StateParameters: [1x1 struct]
       ObjectClassID: 0
          TrackLogic: 'History'
     TrackLogicState: [1 1 0 0 0]
         IsConfirmed: 1
           IsCoasted: 0
      IsSelfReported: 1
    ObjectAttributes: [1x1 struct]
```

Delete the first track.

```
deleted1 = deleteTrack(tracker,1)
```

```
deleted1 = logical
   1
```

Uncomment the following to delete a nonexistent track. A warning will be issued.

```
% deleted2 = deleteTrack(tracker,2)
```

Input Arguments

tracker — Multi-object tracker
multiObjectTracker System object

Multi-object tracker, specified as a `multiObjectTracker` System object.

trackID — Track identifier
positive integer

Track identifier, specified as a positive integer.

Example: 21

Output Arguments

deleted — Indicate if track was successfully deleted
1 | 0

Indicate if the track was successfully deleted or not, returned as `1` or `0`. If the track specified by the `trackID` input existed and was successfully deleted, it returns as `1`. If the track did not exist, a warning is issued and it returns as `0`.

See Also

`initializeTrack` | `multiObjectTracker`

Introduced in R2020a

initializeTrack

Initialize new track

Syntax

```
trackID = initializeTrack(tracker,track)
trackID = initializeTrack(tracker,track,filter)
```

Description

`trackID = initializeTrack(tracker,track)` initializes a new track in the `tracker`. The tracker must be updated at least once before initializing a track. If the track is initialized successfully, the tracker assigns the output `trackID` to the track, sets the `UpdateTime` of the track equal to the last step time in the tracker, and synchronizes the data in the input `track` to the initialized track.

A warning is issued if the tracker already maintains the maximum number of tracks specified by its`MaxNumTracks` property. In this case, the `trackID` is returned as `0`, which indicates a failure to initialize the track.

`trackID = initializeTrack(tracker,track,filter)` initializes a new track in the `tracker`, using a specified tracking filter, `filter`.

Examples

Initialize Track in `multiObjectTracker`

Create a GNN tracker and update the tracker with detections at t = 0 and t = 1second.

```
tracker = multiObjectTracker
```

```
tracker = 
  multiObjectTracker with properties:

                TrackerIndex: 0
     FilterInitializationFcn: 'initcvkf'
         AssignmentThreshold: [30 Inf]
                MaxNumTracks: 200
               MaxNumSensors: 20

       ConfirmationThreshold: [2 3]
           DeletionThreshold: [5 5]

          HasCostMatrixInput: false
  HasDetectableTrackIDsInput: false
             StateParameters: [1x1 struct]

                   NumTracks: 0
          NumConfirmedTracks: 0
```

```
detection1 = objectDetection(0,[1;1;1]);
detection2 = objectDetection(1,[1.1;1.2;1.1]);
```

```
tracker(detection1,0);
currentTrack = tracker(detection2,1);
```

As seen from the `NumTracks` property, the tracker now maintains one track.

```
tracker
```

```
tracker =
  multiObjectTracker with properties:

                 TrackerIndex: 0
      FilterInitializationFcn: 'initcvkf'
          AssignmentThreshold: [30 Inf]
                 MaxNumTracks: 200
                MaxNumSensors: 20

        ConfirmationThreshold: [2 3]
            DeletionThreshold: [5 5]

           HasCostMatrixInput: false
   HasDetectableTrackIDsInput: false
              StateParameters: [1x1 struct]

                    NumTracks: 1
           NumConfirmedTracks: 1
```

Create a new track using the `objectTrack` object.

```
newTrack = objectTrack()
```

```
newTrack =
  objectTrack with properties:

             TrackID: 1
            BranchID: 0
         SourceIndex: 1
          UpdateTime: 0
                 Age: 1
               State: [6x1 double]
     StateCovariance: [6x6 double]
     StateParameters: [1x1 struct]
       ObjectClassID: 0
          TrackLogic: 'History'
     TrackLogicState: 1
         IsConfirmed: 1
           IsCoasted: 0
      IsSelfReported: 1
    ObjectAttributes: [1x1 struct]
```

Initialize a track in the GNN tracker object using the newly created track.

```
trackID = initializeTrack(tracker,newTrack)
```

```
trackID = uint32
    2
```

As seen from the `NumTracks` property, the tracker now maintains two tracks.

```
tracker

tracker = 
  multiObjectTracker with properties:

                TrackerIndex: 0
     FilterInitializationFcn: 'initcvkf'
         AssignmentThreshold: [30 Inf]
                MaxNumTracks: 200
               MaxNumSensors: 20

       ConfirmationThreshold: [2 3]
           DeletionThreshold: [5 5]

          HasCostMatrixInput: false
   HasDetectableTrackIDsInput: false
             StateParameters: [1x1 struct]

                   NumTracks: 2
          NumConfirmedTracks: 2
```

Input Arguments

tracker — Multi-object tracker
multiObjectTracker System object

Multi-object tracker, specified as a multiObjectTracker System object.

track — New track to be initialized
objectTrack object | structure

New track to be initialized, specified as an objectTrack object or a structure. If specified as a structure, the name, variable type, and data size of the fields of the structure must be the same as the name, variable type, and data size of the corresponding properties of the objectTrack object.

Data Types: struct | object

filter — Filter object
trackingKF | trackingEKF | trackingUKF

Filter object, specified as a trackingKF, trackingEKF, or trackingUKF object.

Output Arguments

trackID — Track identifier
nonnegative integer

Track identifier, returned as a nonnegative integer. trackID is returned as 0 if the track is not initialized successfully.

Example: 2

See Also
multiObjectTracker

Introduced in R2020a

getTrackFilterProperties

Obtain filter properties of track from multi-object tracker

Syntax

```
values = getTrackFilterProperties(tracker,trackID,property)
values = getTrackFilterProperties(tracker,trackID,property1,...,propertyN)
```

Description

`values = getTrackFilterProperties(tracker,trackID,property)` returns the tracking filter property values for a specific track within a multi-object tracker. `trackID` is the ID of that specific track.

`values = getTrackFilterProperties(tracker,trackID,property1,...,propertyN)` returns multiple property values. You can specify the properties in any order.

Examples

Display and Set Tracking Filter Properties in Multi-Object Tracker

Create a `multiObjectTracker` System object™ using a constant-acceleration, linear Kalman filter for all tracks.

```
tracker = multiObjectTracker('FilterInitializationFcn',@initcakf, ...
    'ConfirmationParameters',[4 5],'DeletionThreshold',[9 9]);
```

Create two detections and generate tracks for these detections.

```
detection1 = objectDetection(1.0,[10; 10]);
detection2 = objectDetection(1.0,[1000; 1000]);
[~,tracks] = tracker([detection1 detection2],1.1)
```

```
tracks=2×1 object
  2x1 objectTrack array with properties:

    TrackID
    BranchID
    SourceIndex
    UpdateTime
    Age
    State
    StateCovariance
    StateParameters
    ObjectClassID
    TrackLogic
    TrackLogicState
    IsConfirmed
    IsCoasted
    IsSelfReported
```

```
    ObjectAttributes
```

Get filter property values for the first track. Display the process noise values.

```
values = getTrackFilterProperties(tracker,1,'MeasurementNoise','ProcessNoise','MotionModel');
values{2}
```

```
ans = 6×6

    0.0000    0.0005    0.0050         0         0         0
    0.0005    0.0100    0.1000         0         0         0
    0.0050    0.1000    1.0000         0         0         0
         0         0         0    0.0000    0.0005    0.0050
         0         0         0    0.0005    0.0100    0.1000
         0         0         0    0.0050    0.1000    1.0000
```

Set new values for this property by doubling the process noise for the first track. Display the updated process noise values.

```
setTrackFilterProperties(tracker,1,'ProcessNoise',2*values{2});
values = getTrackFilterProperties(tracker,1,'ProcessNoise');
values{1}
```

```
ans = 6×6

    0.0001    0.0010    0.0100         0         0         0
    0.0010    0.0200    0.2000         0         0         0
    0.0100    0.2000    2.0000         0         0         0
         0         0         0    0.0001    0.0010    0.0100
         0         0         0    0.0010    0.0200    0.2000
         0         0         0    0.0100    0.2000    2.0000
```

Input Arguments

tracker — Multi-object tracker
multiObjectTracker System object

Multi-object tracker, specified as a multiObjectTracker System object.

trackID — Track ID
positive integer

Track ID, specified as a positive integer. trackID must be a valid track in tracker.

property — Tracking filter property
character vector | string scalar

Tracking filter property to return values for, specified as a character vector or string scalar. property must be a valid property of the tracking filter used by tracker. Valid tracking filters are trackingKF, trackingEKF, and trackingUKF.

You can specify additional properties in any order.

Example: 'MeasurementNoise','ProcessNoise'

Data Types: `char` | `string`

Output Arguments

`values` — Tracking filter property values
cell array

Tracking filter property values, returned as a cell array. Each element in the cell array corresponds to the values of a specified property. `getTrackFilterProperties` returns the values in the same order in which you specified the corresponding properties.

See Also

Objects
`multiObjectTracker` | `trackingEKF` | `trackingKF` | `trackingUKF`

Functions
`setTrackFilterProperties` | `updateTracks`

Introduced in R2017a

predictTracksToTime

Predict track state

Syntax

```
predictedtracks = predictTracksToTime(tracker,trackID,time)
predictedtracks = predictTracksToTime(tracker,category,time)
predictedtracks = predictTracksToTime(tracker,category,
time,'WithCovariance',tf)
```

Description

`predictedtracks = predictTracksToTime(tracker,trackID,time)` returns the predicted tracks, `predictedtracks`, of the `tracker`, at the specified time, `time`. The tracker or fuser must be updated at least once before calling this object function. Use `isLocked(tracker)` to test whether the tracker or fuser has been updated.

Note This function only outputs the predicted tracks and does not update the internal track states of the `tracker`.

`predictedtracks = predictTracksToTime(tracker,category,time)` returns all predicted tracks for a specified category, `category`, of tracked objects.

`predictedtracks = predictTracksToTime(tracker,category,
time,'WithCovariance',tf)` also allows you to specify whether to predict the state covariance of each track or not by setting the `tf` flag to `true` or `false`. Predicting the covariance slows down the prediction process and increases the computation cost, but it provides the predicted track state covariance in addition to the predicted state. The default is false.

Examples

Predict Track State in `multiObjectTracker`

Create a track from a detection at time t = 0 second and predict it to t = 1second.

```
tracker = multiObjectTracker;
detection = objectDetection(0,[0;0;0]);
tracker(detection,0);
predictedtracks = predictTracksToTime(tracker,'all',1)
```

```
predictedtracks = 
  objectTrack with properties:

              TrackID: 1
             BranchID: 0
          SourceIndex: 0
           UpdateTime: 1
                  Age: 1
```

```
             State: [6x1 double]
   StateCovariance: [6x6 double]
   StateParameters: [1x1 struct]
     ObjectClassID: 0
        TrackLogic: 'History'
   TrackLogicState: [1 0 0 0 0]
       IsConfirmed: 0
         IsCoasted: 0
    IsSelfReported: 1
  ObjectAttributes: [1x1 struct]
```

Input Arguments

`tracker` — Multi-object tracker
`multiObjectTracker` System object

Multi-object tracker, specified as a `multiObjectTracker` System object.

`trackID` — Track identifier
positive integer

Track identifier, specified as a positive integer. Only the track specified by the `trackID` is predicted in the tracker.

Example: `15`

Data Types: `single` | `double`

`time` — Prediction time
scalar

Prediction time, specified as a scalar. The states of tracks are predicted to this time. The time must be greater than the time input to the tracker in the previous track update. Units are in seconds.

Example: `1.0`

Data Types: `single` | `double`

`category` — Track categories
`'all'` | `'confirmed'` | `'tentative'`

Track categories, specified as `'all'`, `'confirmed'`, or `'tentative'`. You can choose to predict all tracks, only confirmed tracks, or only tentative tracks.

Data Types: `char`

Output Arguments

`predictedtracks` — List of predicted track or branch states
array of `objectTrack` objects | array of structures

List of tracks or branches, returned as:

- An array of `objectTrack` objects in the MATLAB interpreted mode.

- An array of structures in the code generation mode. The field names of the structures are the same as the names of properties in `objectTrack`.

Data Types: `struct` | `object`

See Also

`multiObjectTracker`

Introduced in R2020a

setTrackFilterProperties

Set filter properties of track from multi-object tracker

Syntax

```
setTrackFilterProperties(tracker,trackID,property,value)
setTrackFilterProperties(tracker,
trackID,property1,value1,...,propertyN,valueN)
```

Description

`setTrackFilterProperties(tracker,trackID,property,value)` sets the specified tracking filter property to the indicated value for a specific track within the multi-object tracker. `trackID` is the ID of that specific track.

`setTrackFilterProperties(tracker, trackID,property1,value1,...,propertyN,valueN)` sets multiple property values. You can specify the property-value pairs in any order.

Examples

Display and Set Tracking Filter Properties in Multi-Object Tracker

Create a `multiObjectTracker` System object™ using a constant-acceleration, linear Kalman filter for all tracks.

```
tracker = multiObjectTracker('FilterInitializationFcn',@initcakf, ...
    'ConfirmationParameters',[4 5],'DeletionThreshold',[9 9]);
```

Create two detections and generate tracks for these detections.

```
detection1 = objectDetection(1.0,[10; 10]);
detection2 = objectDetection(1.0,[1000; 1000]);
[~,tracks] = tracker([detection1 detection2],1.1)
```

```
tracks=2×1 object
  2x1 objectTrack array with properties:

    TrackID
    BranchID
    SourceIndex
    UpdateTime
    Age
    State
    StateCovariance
    StateParameters
    ObjectClassID
    TrackLogic
    TrackLogicState
    IsConfirmed
    IsCoasted
```

```
    IsSelfReported
    ObjectAttributes
```

Get filter property values for the first track. Display the process noise values.

```
values = getTrackFilterProperties(tracker,1,'MeasurementNoise','ProcessNoise','MotionModel');
values{2}
```

```
ans = 6×6

    0.0000    0.0005    0.0050         0         0         0
    0.0005    0.0100    0.1000         0         0         0
    0.0050    0.1000    1.0000         0         0         0
         0         0         0    0.0000    0.0005    0.0050
         0         0         0    0.0005    0.0100    0.1000
         0         0         0    0.0050    0.1000    1.0000
```

Set new values for this property by doubling the process noise for the first track. Display the updated process noise values.

```
setTrackFilterProperties(tracker,1,'ProcessNoise',2*values{2});
values = getTrackFilterProperties(tracker,1,'ProcessNoise');
values{1}
```

```
ans = 6×6

    0.0001    0.0010    0.0100         0         0         0
    0.0010    0.0200    0.2000         0         0         0
    0.0100    0.2000    2.0000         0         0         0
         0         0         0    0.0001    0.0010    0.0100
         0         0         0    0.0010    0.0200    0.2000
         0         0         0    0.0100    0.2000    2.0000
```

Input Arguments

tracker — Multi-object tracker
multiObjectTracker System object

Multi-object tracker, specified as a multiObjectTracker System object.

trackID — Track ID
positive integer

Track ID, specified as a positive integer. trackID must be a valid track in tracker.

property — Tracking filter property
character vector | string scalar

Tracking filter property to set values for, specified as a character vector or string scalar. property must be a valid property of the tracking filter used by tracker. Valid tracking filters are trackingKF, trackingEKF, and trackingUKF.

You can specify additional property-value pairs in any order.

Example: 'MeasurementNoise',eye(2,2),'MotionModel','2D Constant Acceleration'

Data Types: char | string

value — Value to set tracking filter property to
valid MATLAB expression

Value to set the corresponding tracking filter property to, specified as a MATLAB expression. value must be a valid value of the corresponding property.

You can specify additional property-value pairs in any order.

Example: 'MeasurementNoise',eye(2,2),'MotionModel','2D Constant Acceleration'

See Also

Objects
multiObjectTracker | trackingEKF | trackingKF | trackingUKF

Functions
getTrackFilterProperties | updateTracks

Introduced in R2017a

updateTracks

Update multi-object tracker with new detections

Syntax

```
confirmedTracks = updateTracks(tracker,detections,time)
[confirmedTracks,tentativeTracks] = updateTracks(tracker,detections,time)
[confirmedTracks,tentativeTracks,allTracks] = updateTracks(tracker,
detections,time)
[ ___ ] = updateTracks(tracker,detections,time,costMatrix)
[ ___ ] = updateTracks( ___ ,detectableTrackIDs)
```

Description

`confirmedTracks = updateTracks(tracker,detections,time)` creates, updates, and deletes tracks in the `multiObjectTracker` System object, `tracker`. Updates are based on the specified list of `detections`, and all tracks are updated to the specified `time`. Each element in the returned `confirmedTracks` corresponds to a single track.

`[confirmedTracks,tentativeTracks] = updateTracks(tracker,detections,time)` also returns `tentativeTracks` containing details about the tentative tracks.

`[confirmedTracks,tentativeTracks,allTracks] = updateTracks(tracker, detections,time)` also returns `allTracks` containing details about all confirmed and tentative tracks. The tracks are returned in the order by which the tracker internally maintains them. You can use this output to help you calculate the cost matrix, an optional input argument.

`[ ___ ] = updateTracks(tracker,detections,time,costMatrix)` specifies a cost matrix, returning any of the outputs from preceding syntaxes.

To specify a cost matrix, set the `HasCostMatrixInput` property of `tracker` to `true`.

`[ ___ ] = updateTracks( ___ ,detectableTrackIDs)` also specifies a list of expected detectable tracks given by `detectableTrackIDs`. This argument can be used with any of the previous input syntaxes.

To enable this syntax, set the `HasDetectableTrackIDsInput` property to `true`.

Examples

Generate Radar Detections of Multiple Vehicles

Generate detections using a forward-facing automotive radar mounted on an ego vehicle. Assume that there are three targets:

- Vehicle 1 is in the center lane, directly in front of the ego vehicle, and driving at the same speed.
- Vehicle 2 is in the left lane and driving faster than the ego vehicle by 12 kilometers per hour.
- Vehicle 3 is in the right lane and driving slower than the ego vehicle by 5 kilometers per hour.

All positions, velocities, and measurements are relative to the ego vehicle. Run the simulation for ten steps.

```
dt = 0.1;
pos1 = [150 0 0];
pos2 = [160 10 0];
pos3 = [130 -10 0];
vel1 = [0 0 0];
vel2 = [12*1000/3600 0 0];
vel3 = [-5*1000/3600 0 0];
car1 = struct('ActorID',1,'Position',pos1,'Velocity',vel1);
car2 = struct('ActorID',2,'Position',pos2,'Velocity',vel2);
car3 = struct('ActorID',3,'Position',pos3,'Velocity',vel3);
```

Create an automotive radar sensor that is offset from the ego vehicle. By default, the sensor location is at (3.4,0) meters from the vehicle center and 0.2 meters above the ground plane. Turn off the range rate computation so that the radar sensor measures position only.

```
radar = radarDetectionGenerator('DetectionCoordinates','Sensor Cartesian', ...
    'MaxRange',200,'RangeResolution',10,'AzimuthResolution',10, ...
    'FieldOfView',[40 15],'UpdateInterval',dt,'HasRangeRate',false);
tracker = multiObjectTracker('FilterInitializationFcn',@initcvkf, ...
    'ConfirmationThreshold',[3 4],'DeletionThreshold',[6 6]);
```

Generate detections with the radar from the non-ego vehicles. The output detections form a cell array and can be passed directly in to the `multiObjectTracker`.

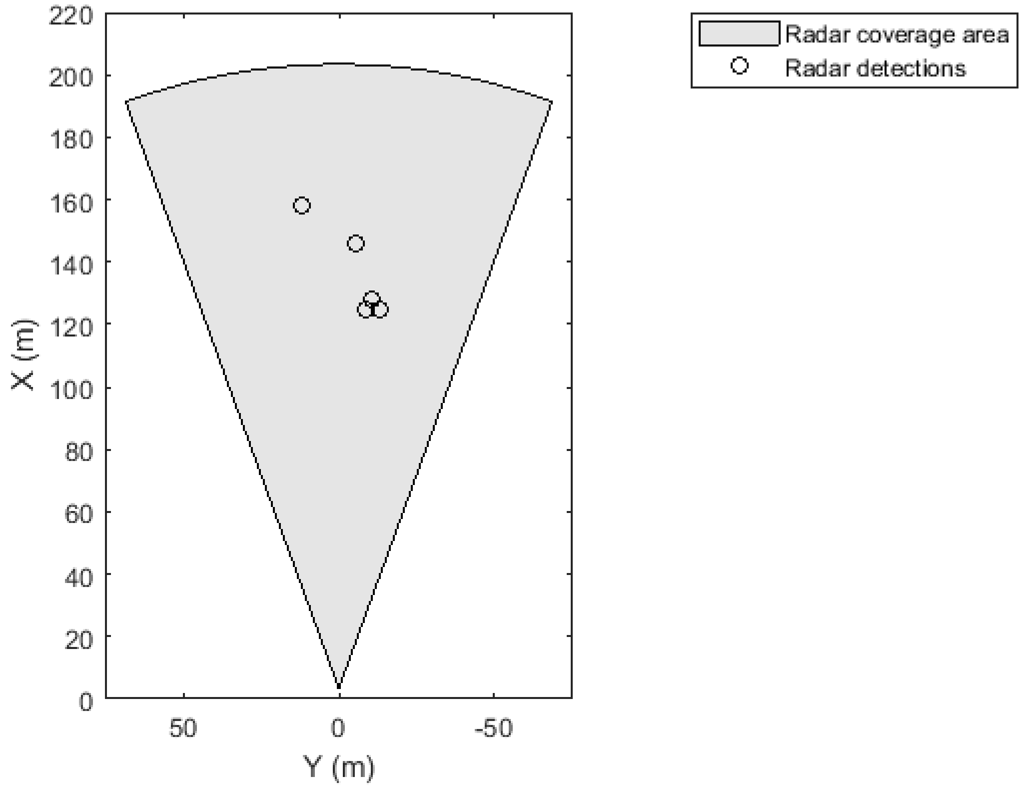
```
simTime = 0;
nsteps = 10;
for k = 1:nsteps
    dets = radar([car1 car2 car3],simTime);
    [confirmedTracks,tentativeTracks,allTracks] = updateTracks(tracker,dets,simTime);
```

Move the cars one time step and update the multi-object tracker.

```
    simTime = simTime + dt;
    car1.Position = car1.Position + dt*car1.Velocity;
    car2.Position = car2.Position + dt*car2.Velocity;
    car3.Position = car3.Position + dt*car3.Velocity;
end
```

Use `birdsEyePlot` to create an overhead view of the detections. Plot the sensor coverage area. Extract the *X* and *Y* positions of the targets by converting the `Measurement` fields of the cell array into a MATLAB array. Display the detections on the bird's-eye plot.

```
BEplot = birdsEyePlot('XLim',[0 220],'YLim',[-75 75]);
caPlotter = coverageAreaPlotter(BEplot,'DisplayName','Radar coverage area');
plotCoverageArea(caPlotter,radar.SensorLocation,radar.MaxRange, ...
    radar.Yaw,radar.FieldOfView(1))
detPlotter = detectionPlotter(BEplot,'DisplayName','Radar detections');
detPos = cellfun(@(d)d.Measurement(1:2),dets,'UniformOutput',false);
detPos = cell2mat(detPos')';
if ~isempty(detPos)
    plotDetection(detPlotter,detPos)
end
```



Input Arguments

`tracker` — Multi-object tracker
`multiObjectTracker` System object

Multi-object tracker, specified as a `multiObjectTracker` System object.

`detections` — Detection list
cell array of `objectDetection` objects

Detection list, specified as a cell array of `objectDetection` objects. The `Time` property value of each `objectDetection` object must be less than or equal to the current time of update, `time`, and greater than the previous time value used to update the multi-object tracker.

`time` — Time of update
real scalar

Time of update, specified as a real scalar. The multi-object tracker updates all tracks to this time. Units are in seconds.

`time` must be greater than or equal to the largest `Time` property value of the `objectDetection` objects in the input `detections` list. `time` must increase in value with each update to the multi-object tracker.

Data Types: `double`

costMatrix — Cost matrix
N_T-by-N_D matrix

Cost matrix, specified as a real-valued N_T-by-N_D matrix, where N_T is the number of existing tracks, and N_D is the number of current detections. The rows of the cost matrix correspond to the existing tracks. The columns correspond to the detections. Tracks are ordered as they appear in the list of tracks in the `allTracks` output argument of the previous update to the multi-object tracker.

In the first update to the multi-object tracker, or when the multi-object tracker has no previous tracks, assign the cost matrix a size of [0, N_D]. The cost must be calculated so that lower costs indicate a higher likelihood that the multi-object tracker assigns a detection to a track. To prevent certain detections from being assigned to certain tracks, use `Inf`.

Dependencies

To enable specification of the cost matrix when updating tracks, set the `HasCostMatrixInput` property of the multi-object tracker to `true`

Data Types: `double`

detectableTrackIDs — Detectable track IDs
real-valued *M*-by-1 vector | real-valued *M*-by-2 matrix

Detectable track IDs, specified as a real-valued *M*-by-1 vector or *M*-by-2 matrix. Detectable tracks are tracks that the sensors expect to detect. The first column of the matrix contains a list of track IDs that the sensors report as detectable. The optional second column contains the detection probability for the track. The detection probability is either reported by a sensor or, if not reported, obtained from the `DetectionProbability` property.

Tracks whose identifiers are not included in `detectableTrackIDs` are considered as undetectable. The track deletion logic does not count the lack of detection as a 'missed detection' for track deletion purposes.

Dependencies

To enable this input argument, set the `detectableTrackIDs` property to `true`.

Data Types: `single` | `double`

Output Arguments

confirmedTracks — Confirmed tracks
array of `objectTrack` objects | array of structures

Confirmed tracks, returned as an array of `objectTrack` objects in MATLAB, and returned as an array of structures in code generation. In code generation, the field names of the returned structure are same with the property names of `objectTrack`.

A track is confirmed if it satisfies the confirmation threshold specified in the `ConfirmationThreshold` property. In that case, the `IsConfirmed` property of the object or field of the structure is `true`.

Data Types: `struct` | `object`

tentativeTracks — Tentative tracks
array of `objectTrack` objects | array of structures

Tentative tracks, returned as an array of objectTrack objects in MATLAB, and returned as an array of structures in code generation. In code generation, the field names of the returned structure are same with the property names of objectTrack.

A track is tentative if it does not satisfy the confirmation threshold specified in the ConfirmationThreshold property. In that case, the IsConfirmed property of the object or field of the structure is false.

Data Types: struct | object

allTracks — All tracks
array of objectTrack objects | array of structures

All tracks, returned as an array of objectTrack objects in MATLAB, and returned as an array of structures in code generation. In code generation, the field names of the returned structure are same with the property names of objectTrack. All tracks consists of confirmed and tentative tracks.

Data Types: struct | object

Algorithms

When you pass detections into updateTracks, the function:

- Attempts to assign the input detections to existing tracks, based on the AssignmentThreshold property of the multi-object tracker.
- Creates new tracks from unassigned detections.
- Updates already assigned tracks and possibly confirms them, based on the ConfirmationThreshold property of the tracker.
- Deletes tracks that have no assigned detections, based on the DeletionThreshold property of the tracker.

See Also

Objects
multiObjectTracker | objectDetection

Functions
getTrackFilterProperties | setTrackFilterProperties

Introduced in R2017a

acfObjectDetectorMonoCamera

Detect objects in monocular camera using aggregate channel features

Description

The `acfObjectDetectorMonoCamera` contains information about an aggregate channel features (ACF) object detector that is configured for use with a monocular camera sensor. To detect objects in an image that was captured by the camera, pass the detector to the `detect` function.

Creation

1. Create an `acfObjectDetector` object by calling the `trainACFObjectDetector` function with training data.

   ```
   detector = trainACFObjectDetector(trainingData,...);
   ```

   Alternatively, create a pretrained detector using functions such as `vehicleDetectorACF` or `peopleDetectorACF`.
2. Create a `monoCamera` object to model the monocular camera sensor.

   ```
   sensor = monoCamera(...);
   ```
3. Create an `acfObjectDetectorMonoCamera` object by passing the detector and sensor as inputs to the `configureDetectorMonoCamera` function. The configured detector inherits property values from the original detector.

   ```
   configuredDetector = configureDetectorMonoCamera(detector,sensor,...);
   ```

Properties

`ModelName` — Name of classification model
character vector | string scalar

Name of the classification model, specified as a character vector or string scalar. By default, the name is set to the heading of the second column of the `trainingData` table specified in the `trainACFObjectDetector` function. You can modify this name after creating your `acfObjectDetectorMonoCamera` object.

Example: `'stopSign'`

`ObjectTrainingSize` — Size of training images
[*height width*] vector

This property is read-only.

Size of training images, specified as a [*height width*] vector.

Example: `[100 100]`

`NumWeakLearners` — Number of weak learners
integer

This property is read-only.

Number of weak learners used in the detector, specified as an integer. `NumWeakLearners` is less than or equal to the maximum number of weak learners for the last training stage. To restrict this maximum, you can use the `'MaxWeakLearners'` name-value pair in the `trainACFObjectDetector` function.

Camera — Camera configuration
`monoCamera` object

This property is read-only.

Camera configuration, specified as a `monoCamera` object. The object contains the camera intrinsics, the location, the pitch, yaw, and roll placement, and the world units for the parameters. Use the intrinsics to transform the object points in the image to world coordinates, which you can then compare to the values in the `WorldObjectSize` property.

`WorldObjectSize` — Range of object widths and lengths
[*minWidth maxWidth*] vector | [*minWidth maxWidth*; *minLength maxLength*] vector

Range of object widths and lengths in world units, specified as a [*minWidth maxWidth*] vector or [*minWidth maxWidth*; *minLength maxLength*] vector. Specifying the range of object lengths is optional.

Object Functions

detect Detect objects using ACF object detector configured for monocular camera

Examples

Detect Vehicles Using Monocular Camera and ACF

Configure an ACF object detector for use with a monocular camera mounted on an ego vehicle. Use this detector to detect vehicles within video frames captured by the camera.

Load an `acfObjectDetector` object pretrained to detect vehicles.

```
detector = vehicleDetectorACF;
```

Model a monocular camera sensor by creating a `monoCamera` object. This object contains the camera intrinsics and the location of the camera on the ego vehicle.

```
focalLength = [309.4362 344.2161];    % [fx fy]
principalPoint = [318.9034 257.5352]; % [cx cy]
imageSize = [480 640];                % [mrows ncols]
height = 2.1798;                      % height of camera above ground, in meters
pitch = 14;                           % pitch of camera, in degrees
intrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);

monCam = monoCamera(intrinsics,height,'Pitch',pitch);
```

Configure the detector for use with the camera. Limit the width of detected objects to a typical range for vehicle widths: 1.5–2.5 meters. The configured detector is an `acfObjectDetectorMonoCamera` object.

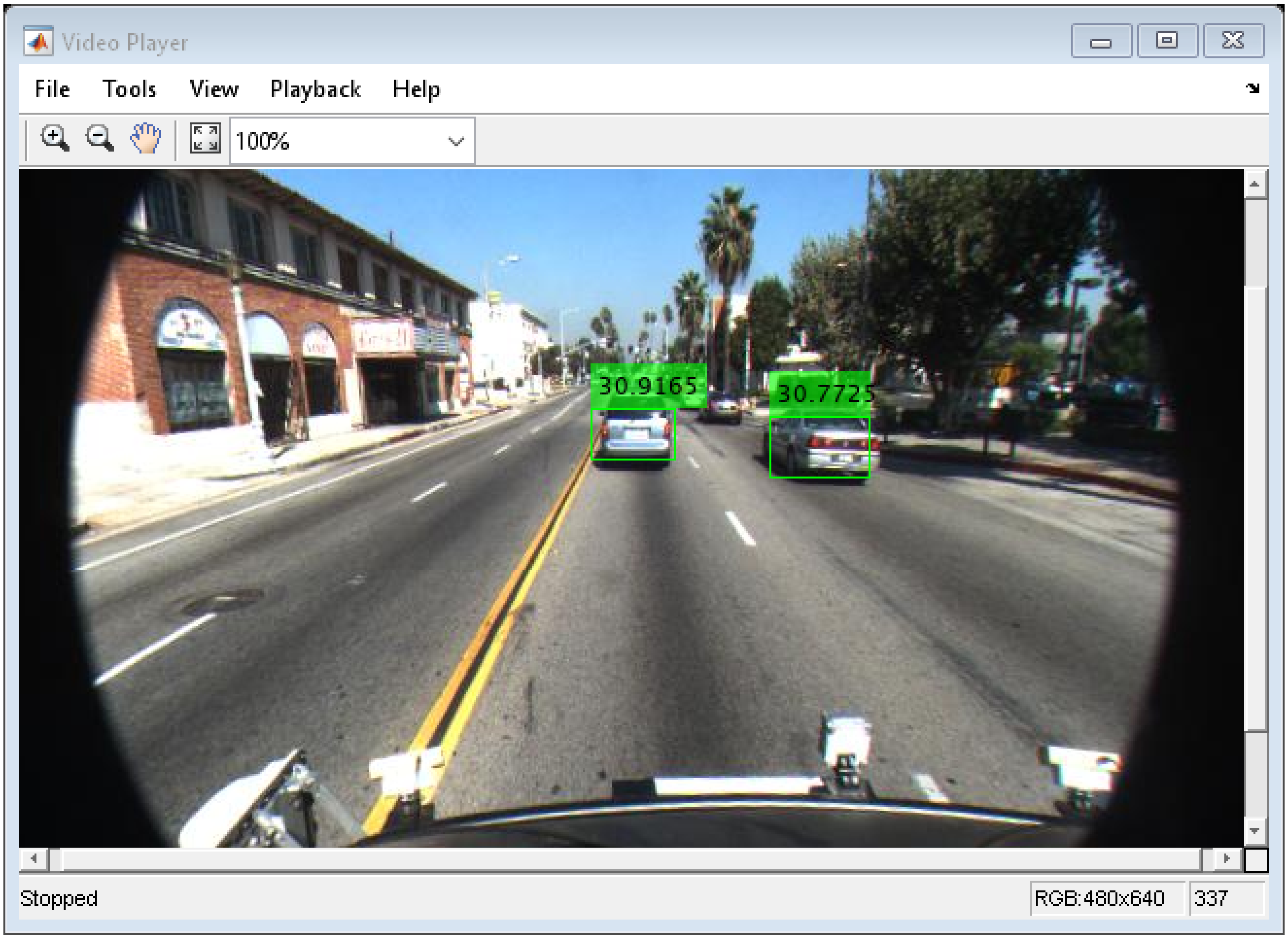
```
vehicleWidth = [1.5 2.5];
detectorMonoCam = configureDetectorMonoCamera(detector,monCam,vehicleWidth);
```

Load a video captured from the camera, and create a video reader and player.

```
videoFile = fullfile(toolboxdir('driving'),'drivingdata','caltech_washington1.avi');
reader = VideoReader(videoFile);
videoPlayer = vision.VideoPlayer('Position',[29 597 643 386]);
```

Run the detector in a loop over the video. Annotate the video with the bounding boxes for the detections and the detection confidence scores.

```
cont = hasFrame(reader);
while cont
   I = readFrame(reader);

   % Run the detector.
   [bboxes,scores] = detect(detectorMonoCam,I);
   if ~isempty(bboxes)
       I = insertObjectAnnotation(I, ...
                            'rectangle',bboxes, ...
                            scores, ...
                            'Color','g');
   end
   videoPlayer(I)
   % Exit the loop if the video player figure is closed.
   cont = hasFrame(reader) && isOpen(videoPlayer);
end

release(videoPlayer);
```



Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

This function supports C/C++ code generation with the limitations:

- Supports code generation (requires MATLAB Coder™) only in generic `MATLAB Host Computer` target platform.

See Also

Apps
Ground Truth Labeler

Functions
`configureDetectorMonoCamera` | `peopleDetectorACF` | `trainACFObjectDetector` | `vehicleDetectorACF`

Objects
`monoCamera`

Introduced in R2017a

detect

Detect objects using ACF object detector configured for monocular camera

Syntax

```
bboxes = detect(detector,I)
[bboxes,scores] = detect(detector,I)
[ ___ ]= detect(detector,I,roi)
[ ___ ] = detect( ___ ,Name,Value)
```

Description

`bboxes = detect(detector,I)` detects objects within image `I` using an aggregate channel features (ACF) object detector configured for a monocular camera. The locations of objects detected are returned as a set of bounding boxes.

`[bboxes,scores] = detect(detector,I)` also returns the detection confidence scores for each bounding box.

`[ ___ ]= detect(detector,I,roi)` detects objects within the rectangular search region specified by `roi`, using any of the preceding syntaxes.

`[ ___ ] = detect( ___ ,Name,Value)` specifies options using one or more `Name,Value` pair arguments. For example, `detect(detector,I,'WindowStride',2)` sets the stride of the sliding window used to detect objects to 2.

Examples

Detect Vehicles Using Monocular Camera and ACF

Configure an ACF object detector for use with a monocular camera mounted on an ego vehicle. Use this detector to detect vehicles within video frames captured by the camera.

Load an `acfObjectDetector` object pretrained to detect vehicles.

```
detector = vehicleDetectorACF;
```

Model a monocular camera sensor by creating a `monoCamera` object. This object contains the camera intrinsics and the location of the camera on the ego vehicle.

```
focalLength = [309.4362 344.2161];    % [fx fy]
principalPoint = [318.9034 257.5352]; % [cx cy]
imageSize = [480 640];                % [mrows ncols]
height = 2.1798;                      % height of camera above ground, in meters
pitch = 14;                           % pitch of camera, in degrees
intrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);

monCam = monoCamera(intrinsics,height,'Pitch',pitch);
```

Configure the detector for use with the camera. Limit the width of detected objects to a typical range for vehicle widths: 1.5–2.5 meters. The configured detector is an `acfObjectDetectorMonoCamera` object.

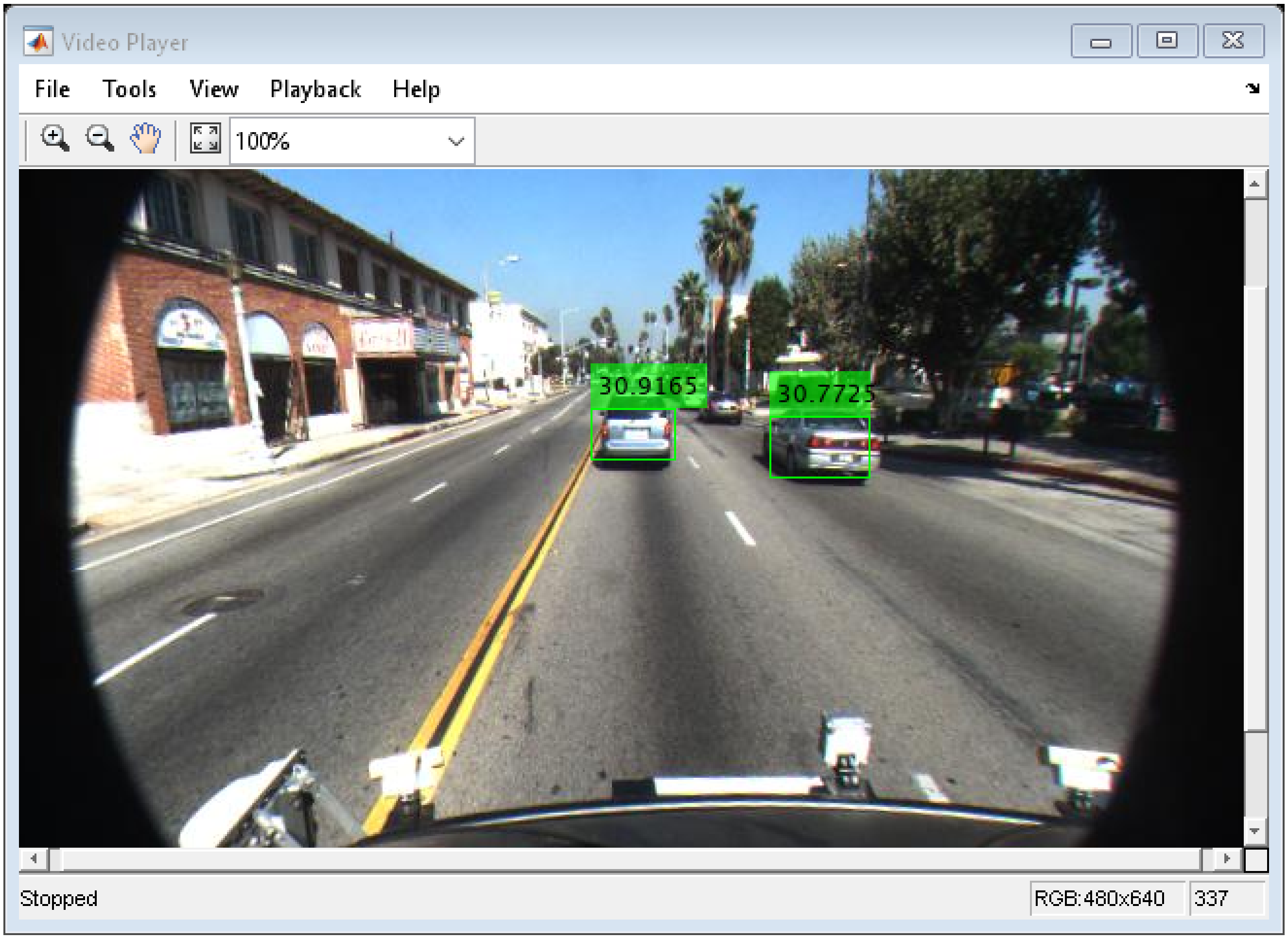
```
vehicleWidth = [1.5 2.5];
detectorMonoCam = configureDetectorMonoCamera(detector,monCam,vehicleWidth);
```

Load a video captured from the camera, and create a video reader and player.

```
videoFile = fullfile(toolboxdir('driving'),'drivingdata','caltech_washington1.avi');
reader = VideoReader(videoFile);
videoPlayer = vision.VideoPlayer('Position',[29 597 643 386]);
```

Run the detector in a loop over the video. Annotate the video with the bounding boxes for the detections and the detection confidence scores.

```
cont = hasFrame(reader);
while cont
   I = readFrame(reader);

   % Run the detector.
   [bboxes,scores] = detect(detectorMonoCam,I);
   if ~isempty(bboxes)
       I = insertObjectAnnotation(I, ...
                           'rectangle',bboxes, ...
                           scores, ...
                           'Color','g');
   end
   videoPlayer(I)
   % Exit the loop if the video player figure is closed.
   cont = hasFrame(reader) && isOpen(videoPlayer);
end

release(videoPlayer);
```



Input Arguments

`detector` — ACF object detector configured for monocular camera
`acfObjectDetectorMonoCamera` object

ACF object detector configured for a monocular camera, specified as an `acfObjectDetectorMonoCamera` object. To create this object, use the `configureDetectorMonoCamera` function with a `monoCamera` object and trained `acfObjectDetector` object as inputs.

`I` — Input image
grayscale image | RGB image

Input image, specified as a real, nonsparse, grayscale or RGB image.

Data Types: `uint8` | `uint16` | `int16` | `double` | `single` | `logical`

`roi` — Search region of interest
[*x y width height*] vector

Search region of interest, specified as an [*x y width height*] vector. The vector specifies the upper left corner and size of a region in pixels.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'NumScaleLevels',4`

`NumScaleLevels` — Number of scale levels per octave
8 (default) | positive integer

Number of scale levels per octave, specified as the comma-separated pair consisting of `'NumScaleLevels'` and a positive integer. Each octave is a power-of-two downscaling of the image. To detect people at finer scale increments, increase this number. Recommended values are in the range [4, 8].

`WindowStride` — Stride for sliding window
4 (default) | positive integer

Stride for the sliding window, specified as the comma-separated pair consisting of `'WindowStride'` and a positive integer. This value indicates the distance for the function to move the window in both the *x* and *y* directions. The sliding window scans the images for object detection.

`SelectStrongest` — Select strongest bounding box for each object
`true` (default) | `false`

Select the strongest bounding box for each detected object, specified as the comma-separated pair consisting of `'SelectStrongest'` and either `true` or `false`.

- `true` — Return the strongest bounding box per object. To select these boxes, `detect` calls the `selectStrongestBbox` function, which uses nonmaximal suppression to eliminate overlapping bounding boxes based on their confidence scores.
- `false` — Return all detected bounding boxes. You can then create your own custom operation to eliminate overlapping bounding boxes.

`MinSize` — Minimum region size
[*height width*] vector

Minimum region size that contains a detected object, specified as the comma-separated pair consisting of `'MinSize'` and a [*height width*] vector. Units are in pixels.

By default, `MinSize` is the smallest object that the trained `detector` can detect.

`MaxSize` — Maximum region size
`size(I)` (default) | [*height width*] vector

Maximum region size that contains a detected object, specified as the comma-separated pair consisting of `'MaxSize'` and a [*height width*] vector. Units are in pixels.

To reduce computation time, set this value to the known maximum region size for the objects being detected in the image. By default, `'MaxSize'` is set to the height and width of the input image, `I`.

`Threshold` — Classification accuracy threshold
–1 (default) | numeric scalar

Classification accuracy threshold, specified as the comma-separated pair consisting of `'Threshold'` and a numeric scalar. Recommended values are in the range [–1, 1]. During multiscale object detection, the threshold value controls the accuracy and speed for classifying image subregions as either objects or nonobjects. To speed up the performance at the risk of missing true detections, increase this threshold.

Output Arguments

`bboxes` — Location of objects detected within image
M-by-4 matrix

Location of objects detected within the input image, returned as an *M*-by-4 matrix, where *M* is the number of bounding boxes. Each row of `bboxes` contains a four-element vector of the form [*x y width height*]. This vector specifies the upper left corner and size of that corresponding bounding box in pixels.

`scores` — Detection confidence scores
M-by-1 vector

Detection confidence scores, returned as an *M*-by-1 vector, where *M* is the number of bounding boxes. A higher score indicates higher confidence in the detection.

See Also

Apps
Ground Truth Labeler

Functions
`configureDetectorMonoCamera` | `selectStrongestBbox` | `trainACFObjectDetector`

Objects
`acfObjectDetector` | `monoCamera`

Introduced in R2017a

fastRCNNObjectDetectorMonoCamera

Detect objects in monocular camera using Fast R-CNN deep learning detector

Description

The `fastRCNNObjectDetectorMonoCamera` object contains information about a Fast R-CNN (regions with convolutional neural networks) object detector that is configured for use with a monocular camera sensor. To detect objects in an image that was captured by the camera, pass the detector to the `detect` function. To classify image regions, pass the detector to the `classifyRegions` function.

When using `detect` or `classifyRegions` with `fastRCNNObjectDetectorMonoCamera`, use of a CUDA®-enabled NVIDIA® GPU with a compute capability of 3.0 or higher is highly recommended. The GPU reduces computation time significantly. Usage of the GPU requires Parallel Computing Toolbox™.

Creation

1. Create a `fastRCNNObjectDetector` object by calling the `trainFastRCNNObjectDetector` function with training data (requires Deep Learning Toolbox).

   ```
   detector = trainFastRCNNObjectDetector(trainingData,...);
   ```
2. Create a `monoCamera` object to model the monocular camera sensor.

   ```
   sensor = monoCamera(...);
   ```
3. Create a `fastRCNNObjectDetectorMonoCamera` object by passing the detector and sensor as inputs to the `configureDetectorMonoCamera` function. The configured detector inherits property values from the original detector.

   ```
   configuredDetector = configureDetectorMonoCamera(detector,sensor,...);
   ```

Properties

`ModelName` — Name of classification model
character vector | string scalar

Name of the classification model, specified as a character vector or string scalar. By default, the name is set to the heading of the second column of the `trainingData` table specified in the `trainFastRCNNObjectDetector` function. You can modify this name after creating your `fastRCNNObjectDetectorMonoCamera` object.

Example: `'stopSign'`

`Network` — Trained Fast R-CNN object detection network
object

This property is read-only.

Trained Fast R-CNN detection network, specified as an object. This object stores the layers that define the convolutional neural network used within the Fast R-CNN detector. This network classifies region proposals produced by the `RegionProposalFcn` property.

`RegionProposalFcn` — Region proposal method
function handle

Region proposal method, specified as a function handle.

`ClassNames` — Object class names
cell array

This property is read-only.

Names of the object classes that the Fast R-CNN detector was trained to find, specified as a cell array. This property is set by the `trainingData` input argument for the `trainFastRCNNObjectDetector` function. Specify the class names as part of the `trainingData` table.

`MinObjectSize` — Minimum object size supported
[*height width*] vector

This property is read-only.

Minimum object size supported by the Fast R-CNN network, specified as a [*height width*] vector. The minimum size depends on the network architecture.

`Camera` — Camera configuration
`monoCamera` object

This property is read-only.

Camera configuration, specified as a `monoCamera` object. The object contains the camera intrinsics, the location, the pitch, yaw, and roll placement, and the world units for the parameters. Use the intrinsics to transform the object points in the image to world coordinates, which you can then compare to the values in the `WorldObjectSize` property.

`WorldObjectSize` — Range of object widths and lengths
[*minWidth maxWidth*] vector | [*minWidth maxWidth*; *minLength maxLength*] vector

Range of object widths and lengths in world units, specified as a [*minWidth maxWidth*] vector or [*minWidth maxWidth*; *minLength maxLength*] vector. Specifying the range of object lengths is optional.

Object Functions

detect	Detect objects using Fast R-CNN object detector configured for monocular camera
classifyRegions	Classify objects in image regions using Fast R-CNN object detector configured for monocular camera

See Also

Apps
Ground Truth Labeler

Functions
`configureDetectorMonoCamera` | `trainFastRCNNObjectDetector`

Objects
`fastRCNNObjectDetector` | `monoCamera`

Topics
"Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN" (Computer Vision Toolbox)

Introduced in R2017a

detect

Detect objects using Fast R-CNN object detector configured for monocular camera

Syntax

```
bboxes = detect(detector,I)
[bboxes,scores] = detect(detector,I)
[___,labels] = detect(detector,I)
[___] = detect(___,roi)
detectionResults = detect(detector,ds)
[___] = detect(___,Name,Value)
```

Description

`bboxes = detect(detector,I)` detects objects within image `I` using a Fast R-CNN (regions with convolutional neural networks) object detector configured for a monocular camera. The locations of objects detected are returned as a set of bounding boxes.

When using this function, use of a CUDA-enabled NVIDIA GPU with a compute capability of 3.0 or higher is highly recommended. The GPU reduces computation time significantly. Usage of the GPU requires Parallel Computing Toolbox.

`[bboxes,scores] = detect(detector,I)` also returns the detection confidence scores for each bounding box.

`[___,labels] = detect(detector,I)` also returns a categorical array of labels assigned to the bounding boxes, using any of the preceding syntaxes. The labels used for object classes are defined during training using the `trainFastRCNNObjectDetector` function.

`[___] = detect(___,roi)` detects objects within the rectangular search region specified by `roi`.

`detectionResults = detect(detector,ds)` detects objects within the series of images returned by the `read` function of the input datastore.

`[___] = detect(___,Name,Value)` specifies options using one or more `Name,Value` pair arguments. For example, `detect(detector,I,'NumStongestRegions',1000)` limits the number of strongest region proposals to 1000.

Input Arguments

`detector` — Fast R-CNN object detector configured for monocular camera
`fastRCNNObjectDetectorMonoCamera` object

Fast R-CNN object detector configured for a monocular camera, specified as a `fastRCNNObjectDetectorMonoCamera` object. To create this object, use the `configureDetectorMonoCamera` function with a `monoCamera` object and trained `fastRCNNObjectDetector` object as inputs.

I — Input image
H-by-*W*-by-*C*-by-*B* numeric array of images

Input image, specified as an *H*-by-*W*-by-*C*-by-*B* numeric array of images Images must be real, nonsparse, grayscale or RGB image.

- *H*: Height
- *W*: Width
- *C*: The channel size in each image must be equal to the network's input channel size. For example, for grayscale images, *C* must be equal to `1`. For RGB color images, it must be equal to `3`.
- *B*: The number of images in the array.

The detector is sensitive to the range of the input image. Therefore, ensure that the input image range is similar to the range of the images used to train the detector. For example, if the detector was trained on `uint8` images, rescale this input image to the range [0, 255] by using the `im2uint8` or `rescale` function. The size of this input image should be comparable to the sizes of the images used in training. If these sizes are very different, the detector has difficulty detecting objects because the scale of the objects in the input image differs from the scale of the objects the detector was trained to identify. Consider whether you used the `SmallestImageDimension` property during training to modify the size of training images.

Data Types: `uint8` | `uint16` | `int16` | `double` | `single` | `logical`

ds — Datastore
`datastore` object

Datastore, specified as a datastore object containing a collection of images. Each image must be a grayscale, RGB, or multichannel image. The function processes only the first column of the datastore, which must contain images and must be cell arrays or tables with multiple columns.

`roi` — Search region of interest
[*x y width height*] vector

Search region of interest, specified as an [*x y width height*] vector. The vector specifies the upper left corner and size of a region in pixels.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'NumStrongestRegions',1000`

`NumStrongestRegions` — Maximum number of strongest region proposals
`2000` (default) | positive integer | `Inf`

Maximum number of strongest region proposals, specified as the comma-separated pair consisting of `'NumStrongestRegions'` and a positive integer. Reduce this value to speed up processing time at the cost of detection accuracy. To use all region proposals, specify this value as `Inf`.

`SelectStrongest` — Select strongest bounding box
`true` (default) | `false`

Select the strongest bounding box for each detected object, specified as the comma-separated pair consisting of `'SelectStrongest'` and either `true` or `false`.

- `true` — Return the strongest bounding box per object. To select these boxes, `detect` calls the `selectStrongestBboxMulticlass` function, which uses nonmaximal suppression to eliminate overlapping bounding boxes based on their confidence scores.

  For example:

  ```
  selectStrongestBboxMulticlass(bbox,scores, ...
              'RatioType','Min', ...
              'OverlapThreshold',0.5);
  ```
- `false` — Return all detected bounding boxes. You can then create your own custom operation to eliminate overlapping bounding boxes.

MinSize — Minimum region size
[*height width*] vector

Minimum region size that contains a detected object, specified as the comma-separated pair consisting of `'MinSize'` and a [*height width*] vector. Units are in pixels.

By default, `MinSize` is the smallest object that the trained `detector` can detect.

MaxSize — Maximum region size
`size(I)` (default) | [*height width*] vector

Maximum region size that contains a detected object, specified as the comma-separated pair consisting of `'MaxSize'` and a [*height width*] vector. Units are in pixels.

To reduce computation time, set this value to the known maximum region size for the objects being detected in the image. By default, `'MaxSize'` is set to the height and width of the input image, `I`.

MiniBatchSize — Minimum batch size
128 (default) | scalar

Minimum batch size, specified as the comma-separated pair consisting of `'MiniBatchSize'` and a scalar value. Use the `MiniBatchSize` to process a large collection of images. Images are grouped into minibatches and processed as a batch to improve computation efficiency. Increase the minibatch size to decrease processing time. Decrease the size to use less memory.

ExecutionEnvironment — Hardware resource
`'auto'` (default) | `'gpu'` | `'cpu'`

Hardware resource on which to run the detector, specified as the comma-separated pair consisting of `'ExecutionEnvironment'` and `'auto'`, `'gpu'`, or `'cpu'`.

- `'auto'` — Use a GPU if it is available. Otherwise, use the CPU.
- `'gpu'` — Use the GPU. To use a GPU, you must have Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with a compute capability of 3.0 or higher. If a suitable GPU is not available, the function returns an error.
- `'cpu'` — Use the CPU.

Output Arguments

`bboxes` — Location of objects detected
M-by-4 matrix | *B*-by-1 cell array

Location of objects detected within the input image or images, returned as an *M*-by-4 matrix or a *B*-by-1 cell array. *M* is the number of bounding boxes in an image, and *B* is the number of *M*-by-4 matrices when the input contains an array of images.

Each row of `bboxes` contains a four-element vector of the form [*x y width height*]. This vector specifies the upper left corner and size of that corresponding bounding box in pixels.

`scores` — Detection scores
M-by-1 vector | *B*-by-1 cell array

Detection confidence scores, returned as an *M*-by-1 vector or a *B*-by-1 cell array. *M* is the number of bounding boxes in an image, and *B* is the number of *M*-by-1 vectors when the input contains an array of images. A higher score indicates higher confidence in the detection.

`labels` — Labels for bounding boxes
M-by-1 categorical array | *B*-by-1 cell array

Labels for bounding boxes, returned as an *M*-by-1 categorical array or a *B*-by-1 cell array. *M* is the number of labels in an image, and *B* is the number of *M*-by-1 categorical arrays when the input contains an array of images. You define the class names used to label the objects when you train the input `detector`.

`detectionResults` — Detection results
3-column table

Detection results, returned as a 3-column table with variable names, *Boxes*, *Scores*, and *Labels*. The *Boxes* column contains *M*-by-4 matrices, of *M* bounding boxes for the objects found in the image. Each row contains a bounding box as a 4-element vector in the format [*x*,*y*,*width*,*height*]. The format specifies the upper-left corner location and size in pixels of the bounding box in the corresponding image.

See Also

Apps
Ground Truth Labeler

Functions
`configureDetectorMonoCamera` | `selectStrongestBboxMulticlass` | `trainFastRCNNObjectDetector`

Objects
`fastRCNNObjectDetectorMonoCamera` | `monoCamera`

Introduced in R2017a

classifyRegions

Classify objects in image regions using Fast R-CNN object detector configured for monocular camera

Syntax

```
[labels,scores] = classifyRegions(detector,I,rois)
[labels,scores,allScores] = classifyRegions(detector,I,rois)
[ ___ ] = classifyRegions( ___ ,'ExecutionEnvironment',resource)
```

Description

`[labels,scores] = classifyRegions(detector,I,rois)` classifies objects within the regions of interest of image `I`, using a Fast R-CNN (regions with convolutional neural networks) object detector configured for a monocular camera. For each region, `classifyRegions` returns the class label with the corresponding highest classification score.

When using this function, use of a CUDA enabled NVIDIA GPU with a compute capability of 3.0 or higher is highly recommended. The GPU reduces computation time significantly. Usage of the GPU requires Parallel Computing Toolbox.

`[labels,scores,allScores] = classifyRegions(detector,I,rois)` also returns all the classification scores of each region. The scores are returned in an *M*-by-*N* matrix of *M* regions and *N* class labels.

`[ ___ ] = classifyRegions( ___ ,'ExecutionEnvironment',resource)` specifies the hardware resource used to classify objects within image regions. You can use this name-value pair with any of the preceding syntaxes.

Input Arguments

`detector` — Fast R-CNN object detector configured for monocular camera
`fastRCNNObjectDetectorMonoCamera` object

Fast R-CNN object detector configured for a monocular camera, specified as a `fastRCNNObjectDetectorMonoCamera` object. To create this object, use the `configureDetectorMonoCamera` function with a `monoCamera` object and trained `fastRCNNObjectDetector` object as inputs.

`I` — Input image
grayscale image | RGB image

Input image, specified as a real, nonsparse, grayscale or RGB image.

Data Types: `uint8` | `uint16` | `int16` | `double` | `single` | `logical`

`rois` — Regions of interest
M-by-4 matrix

Regions of interest within the image, specified as an *M*-by-4 matrix defining *M* rectangular regions. Each row contains a four-element vector of the form [*x y width height*]. This vector specifies the upper left corner and size of a region in pixels.

`resource` — Hardware resource
`'auto'` (default) | `'gpu'` | `'cpu'`

Hardware resource used to classify image regions, specified as `'ExecutionEnvironment'` and `'auto'`, `'gpu'`, or `'cpu'`.

- `'auto'` — Use a GPU if it is available. Otherwise, use the CPU.
- `'gpu'` — Use the GPU. To use a GPU, you must have Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with a compute capability of 3.0 or higher. If a suitable GPU is not available, the function returns an error.
- `'cpu'` — Use the CPU.

Example: `'ExecutionEnvironment','cpu'`

Output Arguments

`labels` — Classification labels of regions
M-by-1 categorical array

Classification labels of regions, returned as an *M*-by-1 categorical array. *M* is the number of regions of interest in `rois`. Each class name in `labels` corresponds to a classification score in `scores` and a region of interest in `rois`. `classifyRegions` obtains the class names from the input `detector`.

`scores` — Highest classification score per region
M-by-1 vector of values in the range [0, 1]

Highest classification score per region, returned as an *M*-by-1 vector of values in the range [0, 1]. *M* is the number of regions of interest in `rois`. Each classification score in `scores` corresponds to a class name in `labels` and a region of interest in `rois`. A higher score indicates higher confidence in the classification.

`allScores` — All classification scores per region
M-by-*N* matrix of values in the range [0, 1]

All classification scores per region, returned as an *M*-by-*N* matrix of values in the range [0, 1]. *M* is the number of regions in `rois`. *N* is the number of class names stored in the input `detector`. Each row of classification scores in `allscores` corresponds to a region of interest in `rois`. A higher score indicates higher confidence in the classification.

See Also

Apps
Ground Truth Labeler

Functions
`configureDetectorMonoCamera` | `trainFastRCNNObjectDetector`

Objects
`fastRCNNObjectDetectorMonoCamera` | `monoCamera`

Introduced in R2017a

fasterRCNNObjectDetectorMonoCamera

Detect objects in monocular camera using Faster R-CNN deep learning detector

Description

The `fasterRCNNObjectDetectorMonoCamera` object contains information about a Faster R-CNN (regions with convolutional neural networks) object detector that is configured for use with a monocular camera sensor. To detect objects in an image that was captured by the camera, pass the detector to the `detect` function.

When using the `detect` function with `fasterRCNNObjectDetectorMonoCamera`, use of a CUDA enabled NVIDIA GPU with a compute capability of 3.0 or higher is highly recommended. The GPU reduces computation time significantly. Usage of the GPU requires Parallel Computing Toolbox.

Creation

1. Create a `fasterRCNNObjectDetector` object by calling the `trainFasterRCNNObjectDetector` function with training data (requires Deep Learning Toolbox).

   ```
   detector = trainFasterRCNNObjectDetector(trainingData,...);
   ```

   Alternatively, create a pretrained detector by using the `vehicleDetectorFasterRCNN` function.
2. Create a `monoCamera` object to model the monocular camera sensor.

   ```
   sensor = monoCamera(...);
   ```
3. Create a `fasterRCNNObjectDetectorMonoCamera` object by passing the detector and sensor as inputs to the `configureDetectorMonoCamera` function. The configured detector inherits property values from the original detector.

   ```
   configuredDetector = configureDetectorMonoCamera(detector,sensor,...);
   ```

Properties

`ModelName` — Name of classification model
character vector | string scalar

This property is read-only.

Name of the classification model, specified as a character vector or string scalar. By default, the name is set to the heading of the second column of the `trainingData` table specified in the `trainFasterRCNNObjectDetector` function. You can modify this name after creating your `fasterRCNNObjectDetectorMonoCamera` object.

`Network` — Trained Fast R-CNN object detection network
`DAGNetwork` object

This property is read-only.

Trained Fast R-CNN object detection network, specified as a `DAGNetwork` object. This object stores the layers that define the convolutional neural network used within the Faster R-CNN detector.

`AnchorBoxes` — Size of anchor boxes
M-by-2 matrix

This property is read-only.

Size of anchor boxes, specified as an *M*-by-2 matrix, where each row is in the format [*height width*]. This value is set during training.

`ClassNames` — Object class names
cell array

This property is read-only.

Names of the object classes that the Faster R-CNN detector was trained to find, specified as a cell array. This property is set by the `trainingData` input argument for the `trainFasterRCNNObjectDetector` function. Specify the class names as part of the `trainingData` table.

`MinObjectSize` — Minimum object size supported
[*height width*] vector

This property is read-only.

Minimum object size supported by the Faster R-CNN network, specified as a [*height width*] vector. The minimum size depends on the network architecture.

`Camera` — Camera configuration
`monoCamera` object

This property is read-only.

Camera configuration, specified as a `monoCamera` object. The object contains the camera intrinsics, the location, the pitch, yaw, and roll placement, and the world units for the parameters. Use the intrinsics to transform the object points in the image to world coordinates, which you can then compare to the values in the `WorldObjectSize` property.

`WorldObjectSize` — Range of object widths and lengths
[*minWidth maxWidth*] vector | [*minWidth maxWidth*; *minLength maxLength*] vector

Range of object widths and lengths in world units, specified as a [*minWidth maxWidth*] vector or [*minWidth maxWidth*; *minLength maxLength*] vector. Specifying the range of object lengths is optional.

Object Functions

detect   Detect objects using Faster R-CNN object detector configured for monocular camera

Examples

Detect Vehicles Using Monocular Camera and Faster R-CNN

Configure a Faster R-CNN object detector for use with a monocular camera mounted on an ego vehicle. Use this detector to detect vehicles within an image captured by the camera.

Load a `fasterRCNNObjectDetector` object pretrained to detect vehicles.

```
detector = vehicleDetectorFasterRCNN;
```

Model a monocular camera sensor by creating a `monoCamera` object. This object contains the camera intrinsics and the location of the camera on the ego vehicle.

```
focalLength = [309.4362 344.2161];    % [fx fy]
principalPoint = [318.9034 257.5352]; % [cx cy]
imageSize = [480 640];                % [mrows ncols]
height = 2.1798;                      % height of camera above ground, in meters
pitch = 14;                           % pitch of camera, in degrees
intrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);

monCam = monoCamera(intrinsics,height,'Pitch',pitch);
```

Configure the detector for use with the camera. Limit the width of detected objects to a typical range for vehicle widths: 1.5–2.5 meters. The configured detector is a `fasterRCNNObjectDetectorMonoCamera` object.

```
vehicleWidth = [1.5 2.5];
detectorMonoCam = configureDetectorMonoCamera(detector,monCam,vehicleWidth);
```

Read in an image captured by the camera.

```
I = imread('carsinfront.png');
imshow(I)
```

Detect the vehicles in the image by using the detector. Annotate the image with the bounding boxes for the detections and the detection confidence scores.

```
[bboxes,scores] = detect(detectorMonoCam,I);
I = insertObjectAnnotation(I,'rectangle',bboxes,scores,'Color','g');
imshow(I)
```

See Also

Apps
Ground Truth Labeler

Functions
`configureDetectorMonoCamera` | `trainFasterRCNNObjectDetector` | `vehicleDetectorFasterRCNN`

Objects
`fasterRCNNObjectDetector` | `monoCamera`

Topics
"Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN" (Computer Vision Toolbox)

Introduced in R2017a

detect

Detect objects using Faster R-CNN object detector configured for monocular camera

Syntax

```
bboxes = detect(detector,I)
[bboxes,scores] = detect(detector,I)
[___,labels] = detect(detector,I)
[___] = detect(___,roi)
detectionResults = detect(detector,ds)
[___] = detect(___,Name,Value)
```

Description

`bboxes = detect(detector,I)` detects objects within image `I` using a Faster R-CNN (regions with convolutional neural networks) object detector configured for a monocular camera. The locations of objects detected are returned as a set of bounding boxes.

When using this function, use of a CUDA-enabled NVIDIA GPU with a compute capability of 3.0 or higher is highly recommended. The GPU reduces computation time significantly. Usage of the GPU requires Parallel Computing Toolbox.

`[bboxes,scores] = detect(detector,I)` also returns the detection confidence scores for each bounding box.

`[___,labels] = detect(detector,I)` also returns a categorical array of labels assigned to the bounding boxes, using any of the preceding syntaxes. The labels used for object classes are defined during training using the `trainFasterRCNNObjectDetector` function.

`[___] = detect(___,roi)` detects objects within the rectangular search region specified by `roi`.

`detectionResults = detect(detector,ds)` detects objects within the series of images returned by the `read` function of the input datastore.

`[___] = detect(___,Name,Value)` specifies options using one or more `Name,Value` pair arguments. For example, `detect(detector,I,'NumStongestRegions',1000)` limits the number of strongest region proposals to 1000.

Examples

Detect Vehicles Using Monocular Camera and Faster R-CNN

Configure a Faster R-CNN object detector for use with a monocular camera mounted on an ego vehicle. Use this detector to detect vehicles within an image captured by the camera.

Load a `fasterRCNNObjectDetector` object pretrained to detect vehicles.

```
detector = vehicleDetectorFasterRCNN;
```

Model a monocular camera sensor by creating a `monoCamera` object. This object contains the camera intrinsics and the location of the camera on the ego vehicle.

```
focalLength = [309.4362 344.2161];    % [fx fy]
principalPoint = [318.9034 257.5352]; % [cx cy]
imageSize = [480 640];                % [mrows ncols]
height = 2.1798;                      % height of camera above ground, in meters
pitch = 14;                           % pitch of camera, in degrees
intrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);

monCam = monoCamera(intrinsics,height,'Pitch',pitch);
```

Configure the detector for use with the camera. Limit the width of detected objects to a typical range for vehicle widths: 1.5–2.5 meters. The configured detector is a `fasterRCNNObjectDetectorMonoCamera` object.

```
vehicleWidth = [1.5 2.5];
detectorMonoCam = configureDetectorMonoCamera(detector,monCam,vehicleWidth);
```

Read in an image captured by the camera.

```
I = imread('carsinfront.png');
imshow(I)
```

Detect the vehicles in the image by using the detector. Annotate the image with the bounding boxes for the detections and the detection confidence scores.

```
[bboxes,scores] = detect(detectorMonoCam,I);
I = insertObjectAnnotation(I,'rectangle',bboxes,scores,'Color','g');
imshow(I)
```

Input Arguments

`detector` — Faster R-CNN object detector configured for monocular camera
`fasterRCNNObjectDetectorMonoCamera` object

Faster R-CNN object detector configured for a monocular camera, specified as a `fasterRCNNObjectDetectorMonoCamera` object. To create this object, use the `configureDetectorMonoCamera` function with a `monoCamera` object and trained `fasterRCNNObjectDetector` object as inputs.

`I` — Input image
H-by-*W*-by-*C*-by-*B* numeric array of images

Input image, specified as an *H*-by-*W*-by-*C*-by-*B* numeric array of images Images must be real, nonsparse, grayscale or RGB image.

- *H*: Height
- *W*: Width
- *C*: The channel size in each image must be equal to the network's input channel size. For example, for grayscale images, *C* must be equal to `1`. For RGB color images, it must be equal to `3`.
- *B*: The number of images in the array.

The detector is sensitive to the range of the input image. Therefore, ensure that the input image range is similar to the range of the images used to train the detector. For example, if the detector was trained on `uint8` images, rescale this input image to the range [0, 255] by using the `im2uint8` or `rescale` function. The size of this input image should be comparable to the sizes of the images used in training. If these sizes are very different, the detector has difficulty detecting objects because the scale of the objects in the input image differs from the scale of the objects the detector was trained to identify. Consider whether you used the `SmallestImageDimension` property during training to modify the size of training images.

Data Types: `uint8` | `uint16` | `int16` | `double` | `single` | `logical`

ds — Datastore
`datastore` object

Datastore, specified as a datastore object containing a collection of images. Each image must be a grayscale, RGB, or multichannel image. The function processes only the first column of the datastore, which must contain images and must be cell arrays or tables with multiple columns.

roi — Search region of interest
[*x y width height*] vector

Search region of interest, specified as an [*x y width height*] vector. The vector specifies the upper left corner and size of a region in pixels.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'NumStrongestRegions',1000`

NumStrongestRegions — Maximum number of strongest region proposals
`2000` (default) | positive integer | `Inf`

Maximum number of strongest region proposals, specified as the comma-separated pair consisting of `'NumStrongestRegions'` and a positive integer. Reduce this value to speed up processing time at the cost of detection accuracy. To use all region proposals, specify this value as `Inf`.

SelectStrongest — Select strongest bounding box
`true` (default) | `false`

Select the strongest bounding box for each detected object, specified as the comma-separated pair consisting of `'SelectStrongest'` and either `true` or `false`.

- `true` — Return the strongest bounding box per object. To select these boxes, `detect` calls the `selectStrongestBboxMulticlass` function, which uses nonmaximal suppression to eliminate overlapping bounding boxes based on their confidence scores.

  For example:

  ```
  selectStrongestBboxMulticlass(bbox,scores, ...
              'RatioType','Min', ...
              'OverlapThreshold',0.5);
  ```
- `false` — Return all detected bounding boxes. You can then create your own custom operation to eliminate overlapping bounding boxes.

MinSize — Minimum region size
[*height width*] vector

Minimum region size that contains a detected object, specified as the comma-separated pair consisting of `'MinSize'` and a [*height width*] vector. Units are in pixels.

By default, `MinSize` is the smallest object that the trained `detector` can detect.

MaxSize — Maximum region size
`size(I)` (default) | [*height width*] vector

Maximum region size that contains a detected object, specified as the comma-separated pair consisting of `'MaxSize'` and a [*height width*] vector. Units are in pixels.

To reduce computation time, set this value to the known maximum region size for the objects being detected in the image. By default, `'MaxSize'` is set to the height and width of the input image, `I`.

MiniBatchSize — Minimum batch size
`128` (default) | scalar

Minimum batch size, specified as the comma-separated pair consisting of `'MiniBatchSize'` and a scalar value. Use the `MiniBatchSize` to process a large collection of images. Images are grouped into minibatches and processed as a batch to improve computation efficiency. Increase the minibatch size to decrease processing time. Decrease the size to use less memory.

ExecutionEnvironment — Hardware resource
`'auto'` (default) | `'gpu'` | `'cpu'`

Hardware resource on which to run the detector, specified as the comma-separated pair consisting of `'ExecutionEnvironment'` and `'auto'`, `'gpu'`, or `'cpu'`.

- `'auto'` — Use a GPU if it is available. Otherwise, use the CPU.
- `'gpu'` — Use the GPU. To use a GPU, you must have Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with a compute capability of 3.0 or higher. If a suitable GPU is not available, the function returns an error.
- `'cpu'` — Use the CPU.

Output Arguments

bboxes — Location of objects detected
M-by-4 matrix | *B*-by-1 cell array

Location of objects detected within the input image or images, returned as an *M*-by-4 matrix or a *B*-by-1 cell array. *M* is the number of bounding boxes in an image, and *B* is the number of *M*-by-4 matrices when the input contains an array of images.

Each row of `bboxes` contains a four-element vector of the form [*x y width height*]. This vector specifies the upper left corner and size of that corresponding bounding box in pixels.

`scores` — Detection scores
M-by-1 vector | *B*-by-1 cell array

Detection confidence scores, returned as an *M*-by-1 vector or a *B*-by-1 cell array. *M* is the number of bounding boxes in an image, and *B* is the number of *M*-by-1 vectors when the input contains an array of images. A higher score indicates higher confidence in the detection.

`labels` — Labels for bounding boxes
M-by-1 categorical array | *B*-by-1 cell array

Labels for bounding boxes, returned as an *M*-by-1 categorical array or a *B*-by-1 cell array. *M* is the number of labels in an image, and *B* is the number of *M*-by-1 categorical arrays when the input contains an array of images. You define the class names used to label the objects when you train the input `detector`.

`detectionResults` — Detection results
3-column table

Detection results, returned as a 3-column table with variable names, *Boxes*, *Scores*, and *Labels*. The *Boxes* column contains *M*-by-4 matrices, of *M* bounding boxes for the objects found in the image. Each row contains a bounding box as a 4-element vector in the format [*x*,*y*,*width*,*height*]. The format specifies the upper-left corner location and size in pixels of the bounding box in the corresponding image.

See Also

Apps
Ground Truth Labeler

Functions
`configureDetectorMonoCamera` | `selectStrongestBboxMulticlass` | `trainFasterRCNNObjectDetector`

Objects
`fasterRCNNObjectDetectorMonoCamera` | `monoCamera`

Introduced in R2017a

yolov2ObjectDetectorMonoCamera

Detect objects in monocular camera using YOLO v2 deep learning detector

Description

The `yolov2ObjectDetectorMonoCamera` object contains information about you only look once version 2 (YOLO v2) object detector that is configured for use with a monocular camera sensor. To detect objects in an image captured by the camera, pass the detector to the `detect` object function.

When using the `detect` object function with a `yolov2ObjectDetectorMonoCamera` object, use of a CUDA-enabled NVIDIA GPU with a compute capability of 3.0 or higher is highly recommended. The GPU reduces computation time significantly. Usage of the GPU requires Parallel Computing Toolbox.

Creation

1. Create a `yolov2ObjectDetector` object by calling the `trainYOLOv2ObjectDetector` function with training data (requires Deep Learning Toolbox).

   ```
   detector = trainYOLOv2ObjectDetector(trainingData,____);
   ```
2. Create a `monoCamera` object to model the monocular camera sensor.

   ```
   sensor = monoCamera(____);
   ```
3. Create a `yolov2ObjectDetectorMonoCamera` object by passing the detector and sensor as inputs to the `configureDetectorMonoCamera` function. The configured detector inherits property values from the original detector.

   ```
   configuredDetector = configureDetectorMonoCamera(detector,sensor,____);
   ```

Properties

`Camera` — Camera configuration
`monoCamera` object

This property is read-only.

Camera configuration, specified as a `monoCamera` object. The object contains the camera intrinsics, the location, the pitch, yaw, and roll placement, and the world units for the parameters. Use the intrinsics to transform the object points in the image to world coordinates, which you can then compare to the values in the `WorldObjectSize` property.

`WorldObjectSize` — Range of object widths and lengths
[*minWidth maxWidth*] vector | [*minWidth maxWidth*; *minLength maxLength*] vector

Range of object widths and lengths in world units, specified as a [*minWidth maxWidth*] vector or [*minWidth maxWidth*; *minLength maxLength*] vector. Specifying the range of object lengths is optional.

`ModelName` — Name of classification model
character vector | string scalar

Name of the classification model, specified as a character vector or string scalar. By default, the name is set to the heading of the second column of the `trainingData` table specified in the `trainYOLOv2ObjectDetector` function. You can modify this name after creating the `yolov2ObjectDetectorMonoCamera` object.

`Network` — Trained YOLO v2 object detection network
`DAGNetwork` object

This property is read-only.

Trained YOLO v2 object detection network, specified as a `DAGNetwork` object. This object stores the layers that are used within the YOLO v2 object detector.

`ClassNames` — Names of object classes
cell array of character vectors

This property is read-only.

Names of the object classes that the YOLO v2 object detector was trained to find, specified as a cell array of character vectors. This property is set by the `trainingData` input argument for the `trainYOLOv2ObjectDetector` function. Specify the class names as part of the `trainingData` table.

`AnchorBoxes` — Size of anchor boxes
M-by-2 matrix

This property is read-only.

Size of anchor boxes, specified as an *M*-by-2 matrix, where each row is of form [*height width*]. This value specifies the height and width of M anchor boxes. This property is set by the `AnchorBoxes` property of the output layer in the YOLO v2 network.

The anchor boxes are defined when creating the YOLO v2 network by using the `yolov2Layers` function. Alternatively, if you create the YOLO v2 network layer-by-layer, the anchor boxes are defined by using the `yolov2OutputLayer` function.

Object Functions

detect   Detect objects using YOLO v2 object detector configured for monocular camera

Examples

Detect Vehicles Using Monocular Camera and YOLO v2

Configure a YOLO v2 object detector for use with a monocular camera mounted on an ego vehicle. Use this detector to detect vehicles within an image captured by the camera.

Load a `yolov2ObjectDetector` object pretrained to detect vehicles.

```
detector = vehicleDetectorYOLOv2;
```

Model a monocular camera sensor by creating a `monoCamera` object. This object contains the camera intrinsics and the location of the camera on the ego vehicle.

```
focalLength = [309.4362 344.2161];    % [fx fy]
principalPoint = [318.9034 257.5352]; % [cx cy]
imageSize = [480 640];                % [mrows ncols]
height = 2.1798;                      % height of camera above ground, in meters
pitch = 14;                           % pitch of camera, in degrees
intrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);

sensor = monoCamera(intrinsics,height,'Pitch',pitch);
```

Configure the detector for use with the camera. Limit the width of detected objects to 2–3 meters. The configured detector is a `yolov2ObjectDetectorMonoCamera` object.

```
vehicleWidth = [2 3];
detectorMonoCam = configureDetectorMonoCamera(detector,sensor,vehicleWidth);
```

Read in an image captured by the camera.

```
I = imread('carsinfront.png');
```

Detect the vehicles in the image by using the detector. Annotate the image with the bounding boxes for the detections and the detection confidence scores.

```
[bboxes,scores,labels] = detect(detectorMonoCam,I);
I = insertObjectAnnotation(I,'rectangle',bboxes,scores,'Color','g');
imshow(I)
```

Display the labels for detected bounding boxes. The labels specify the class names of the detected objects.

```
disp(labels)
```

```
     vehicle
     vehicle
     vehicle
     vehicle
```

See Also

Apps
Ground Truth Labeler

Functions
`configureDetectorMonoCamera` | `trainYOLOv2ObjectDetector`

Objects
`monoCamera` | `yolov2ObjectDetector`

Topics
"Getting Started with YOLO v2" (Computer Vision Toolbox)
"Object Detection Using YOLO v2 Deep Learning" (Computer Vision Toolbox)

Introduced in R2019a

detect

Detect objects using YOLO v2 object detector configured for monocular camera

Syntax

```
bboxes = detect(detector,I)
[bboxes,scores] = detect(detector,I)
[___,labels] = detect(detector,I)
[___] = detect(___,roi)
detectionResults = detect(detector,ds)
[___] = detect(___,Name,Value)
```

Description

`bboxes = detect(detector,I)` detects objects within image `I` using you look only once version 2 (YOLO v2) object detector configured for a monocular camera. The locations of objects detected are returned as a set of bounding boxes.

When using this function, use of a CUDA-enabled NVIDIA GPU with a compute capability of 3.0 or higher is highly recommended. The GPU reduces computation time significantly. Usage of the GPU requires Parallel Computing Toolbox.

`[bboxes,scores] = detect(detector,I)` also returns the detection confidence scores for each bounding box.

`[___,labels] = detect(detector,I)` returns a categorical array of labels assigned to the bounding boxes in addition to the output arguments from the previous syntax. The labels used for object classes are defined during training using the `trainYOLOv2ObjectDetector` function.

`[___] = detect(___,roi)` detects objects within the rectangular search region specified by `roi`. Use output arguments from any of the previous syntaxes. Specify input arguments from any of the previous syntaxes.

`detectionResults = detect(detector,ds)` detects objects within the series of images returned by the `read` function of the input datastore.

`[___] = detect(___,Name,Value)` also specifies options using one or more `Name,Value` pair arguments in addition to the input arguments in any of the preceding syntaxes.

Examples

Detect Vehicles in Traffic Scenes from Monocular Video Using YOLO v2

Configure a YOLO v2 object detector for detecting vehicles within a video captured by a monocular camera.

Load a `yolov2ObjectDetector` object pretrained to detect vehicles.

```
vehicleDetector = load('yolov2VehicleDetector.mat','detector');
detector = vehicleDetector.detector;
```

Model a monocular camera sensor by creating a `monoCamera` object. This object contains the camera intrinsics and the location of the camera on the ego vehicle.

```
focalLength = [309.4362 344.2161];    % [fx fy]
principalPoint = [318.9034 257.5352]; % [cx cy]
imageSize = [480 640];                % [mrows ncols]
height = 2.1798;                      % Height of camera above ground, in meters
pitch = 14;                           % Pitch of camera, in degrees
intrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);

sensor = monoCamera(intrinsics,height,'Pitch',pitch);
```

Configure the detector for use with the camera. Limit the width of detected objects to 1.5-2.5 meters. The configured detector is a `yolov2ObjectDetectorMonoCamera` object.

```
vehicleWidth = [1.5 2.5];
detectorMonoCam = configureDetectorMonoCamera(detector,sensor,vehicleWidth);
```

Set up the video reader and read the input monocular video.

```
videoFile = '05_highway_lanechange_25s.mp4';
reader = VideoReader(videoFile);
```

Create a video player to display the video and the output detections.

```
videoPlayer = vision.DeployableVideoPlayer();
```

Detect vehicles in the video by using the detector. Specify the detection threshold as 0.6. Annotate the video with the bounding boxes for the detections, labels, and detection confidence scores.

```
cont = hasFrame(reader);
while cont
    I = readFrame(reader);
    [bboxes,scores,labels] = detect(detectorMonoCam,I,'Threshold',0.6); % Run the YOLO v2 object

    if ~isempty(bboxes)
        displayLabel = strcat(cellstr(labels),':',num2str(scores));
        I = insertObjectAnnotation(I,'rectangle',bboxes,displayLabel);
    end
    step(videoPlayer, I);
    cont = hasFrame(reader) && isOpen(videoPlayer); % Exit the loop if the video player figure w
end
```

Input Arguments

detector — YOLO v2 object detector configured for monocular camera
`yolov2ObjectDetectorMonoCamera` object

YOLO v2 object detector configured for monocular camera, specified as a `yolov2ObjectDetectorMonoCamera` object. To create this object, use the `configureDetectorMonoCamera` function with a `monoCamera` object and trained `yolov2ObjectDetector` object as inputs.

I — Input image
H-by-*W*-by-*C*-by-*B* numeric array of images

Input image, specified as an *H*-by-*W*-by-*C*-by-*B* numeric array of images Images must be real, nonsparse, grayscale or RGB image.

- *H*: Height
- *W*: Width
- *C*: The channel size in each image must be equal to the network's input channel size. For example, for grayscale images, *C* must be equal to `1`. For RGB color images, it must be equal to `3`.
- *B*: The number of images in the array.

The detector is sensitive to the range of the input image. Therefore, ensure that the input image range is similar to the range of the images used to train the detector. For example, if the detector was trained on `uint8` images, rescale this input image to the range [0, 255] by using the `im2uint8` or `rescale` function. The size of this input image should be comparable to the sizes of the images used in training. If these sizes are very different, the detector has difficulty detecting objects because the scale of the objects in the input image differs from the scale of the objects the detector was trained to identify. Consider whether you used the `SmallestImageDimension` property during training to modify the size of training images.

Data Types: `uint8` | `uint16` | `int16` | `double` | `single` | `logical`

ds — Datastore
`datastore` object

Datastore, specified as a datastore object containing a collection of images. Each image must be a grayscale, RGB, or multichannel image. The function processes only the first column of the datastore, which must contain images and must be cell arrays or tables with multiple columns.

`roi` — Search region of interest
[*x y width height*] vector

Search region of interest, specified as an [*x y width height*] vector. The vector specifies the upper left corner and size of a region in pixels.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `detect(detector,I,'Threshold',0.25)`

`Threshold` — Detection threshold
`0.5` (default) | scalar in the range [0, 1]

Detection threshold, specified as a comma-separated pair consisting of `'Threshold'` and a scalar in the range [0, 1]. Detections that have scores less than this threshold value are removed. To reduce false positives, increase this value.

`SelectStrongest` — Select strongest bounding box
`true` (default) | `false`

Select the strongest bounding box for each detected object, specified as the comma-separated pair consisting of `'SelectStrongest'` and `true` or `false`.

- `true` — Returns the strongest bounding box per object. The method calls the `selectStrongestBboxMulticlass` function, which uses nonmaximal suppression to eliminate overlapping bounding boxes based on their confidence scores.

  By default, the `selectStrongestBboxMulticlass` function is called as follows

  ```
  selectStrongestBboxMulticlass(bbox,scores,...
                                'RatioType','Min',...
                                'OverlapThreshold',0.5);
  ```
- `false` — Return all the detected bounding boxes. You can then write your own custom method to eliminate overlapping bounding boxes.

MinSize — Minimum region size
`[1 1]` (default) | vector of the form [*height width*]

Minimum region size, specified as the comma-separated pair consisting of `'MinSize'` and a vector of the form [*height width*]. Units are in pixels. The minimum region size defines the size of the smallest region containing the object.

By default, `'MinSize'` is 1-by-1.

MaxSize — Maximum region size
`size(I)` (default) | vector of the form [*height width*]

Maximum region size, specified as the comma-separated pair consisting of `'MaxSize'` and a vector of the form [*height width*]. Units are in pixels. The maximum region size defines the size of the largest region containing the object.

By default, `'MaxSize'` is set to the height and width of the input image, `I`. To reduce computation time, set this value to the known maximum region size for the objects that can be detected in the input test image.

ExecutionEnvironment — Hardware resource
`'auto'` (default) | `'gpu'` | `'cpu'`

Hardware resource on which to run the detector, specified as the comma-separated pair consisting of `'ExecutionEnvironment'` and `'auto'`, `'gpu'`, or `'cpu'`.

- `'auto'` — Use a GPU if it is available. Otherwise, use the CPU.
- `'gpu'` — Use the GPU. To use a GPU, you must have Parallel Computing Toolbox and a CUDA-enabled NVIDIA GPU with a compute capability of 3.0 or higher. If a suitable GPU is not available, the function returns an error.
- `'cpu'` — Use the CPU.

Acceleration — Performance optimization
`'auto'` (default) | `'mex'` | `'none'`

Performance optimization, specified as the comma-separated pair consisting of `'Acceleration'` and one of the following:

- `'auto'` — Automatically apply a number of optimizations suitable for the input network and hardware resource.

- `'mex'` — Compile and execute a MEX function. This option is available when using a GPU only. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. If Parallel Computing Toolbox or a suitable GPU is not available, then the function returns an error.
- `'none'` — Disable all acceleration.

The default option is `'auto'`. If `'auto'` is specified, MATLAB will apply a number of compatible optimizations. If you use the `'auto'` option, MATLAB does not ever generate a MEX function.

Using the `'Acceleration'` options `'auto'` and `'mex'` can offer performance benefits, but at the expense of an increased initial run time. Subsequent calls with compatible parameters are faster. Use performance optimization when you plan to call the function multiple times using new input data.

The `'mex'` option generates and executes a MEX function based on the network and parameters used in the function call. You can have several MEX functions associated with a single network at one time. Clearing the network variable also clears any MEX functions associated with that network.

The `'mex'` option is only available for input data specified as a numeric array, cell array of numeric arrays, table, or image datastore. No other types of datastore support the `'mex'` option.

The `'mex'` option is only available when you are using a GPU. You must also have a C/C++ compiler installed. For setup instructions, see "MEX Setup" (GPU Coder).

`'mex'` acceleration does not support all layers. For a list of supported layers, see "Supported Layers" (GPU Coder).

Output Arguments

bboxes — Location of objects detected

M-by-4 matrix | *B*-by-1 cell array

Location of objects detected within the input image or images, returned as an *M*-by-4 matrix or a *B*-by-1 cell array. *M* is the number of bounding boxes in an image, and *B* is the number of *M*-by-4 matrices when the input contains an array of images.

Each row of `bboxes` contains a four-element vector of the form [*x y width height*]. This vector specifies the upper left corner and size of that corresponding bounding box in pixels.

`scores` — Detection scores

M-by-1 vector | *B*-by-1 cell array

Detection confidence scores, returned as an *M*-by-1 vector or a *B*-by-1 cell array. *M* is the number of bounding boxes in an image, and *B* is the number of *M*-by-1 vectors when the input contains an array of images. A higher score indicates higher confidence in the detection.

`labels` — Labels for bounding boxes

M-by-1 categorical array | *B*-by-1 cell array

Labels for bounding boxes, returned as an *M*-by-1 categorical array or a *B*-by-1 cell array. *M* is the number of labels in an image, and *B* is the number of *M*-by-1 categorical arrays when the input contains an array of images. You define the class names used to label the objects when you train the input `detector`.

`detectionResults` — Detection results
3-column table

Detection results, returned as a 3-column table with variable names, *Boxes*, *Scores*, and *Labels*. The *Boxes* column contains *M*-by-4 matrices, of *M* bounding boxes for the objects found in the image. Each row contains a bounding box as a 4-element vector in the format [*x*,*y*,*width*,*height*]. The format specifies the upper-left corner location and size in pixels of the bounding box in the corresponding image.

See Also

Apps
Ground Truth Labeler

Functions
`configureDetectorMonoCamera` | `evaluateDetectionMissRate` | `evaluateDetectionPrecision` | `selectStrongestBboxMulticlass` | `trainYOLOv2ObjectDetector`

Objects
`monoCamera` | `yolov2ObjectDetectorMonoCamera`

Introduced in R2019a

ssdObjectDetectorMonoCamera

Detect objects in monocular camera using SSD deep learning detector

Description

The `ssdObjectDetectorMonoCamera` detects objects from an image, using a single shot detector (SSD) object detector. To detect objects in an image, pass the trained detector to the `detect` function.

Creation

1. Create a `ssdObjectDetector` object by calling the `trainSSDObjectDetector` function with training data (requires Deep Learning Toolbox).

   ```
   detector = trainSSDObjectDetector(trainingData,____);
   ```
2. Create a `monoCamera` object to model the monocular camera sensor.

   ```
   sensor = monoCamera(____);
   ```
3. Create a `ssdObjectDetectorMonoCamera` object by passing the detector and sensor as inputs to the `configureDetectorMonoCamera` function. The configured detector inherits property values from the original detector.

   ```
   configuredDetector = configureDetectorMonoCamera(detector,sensor,____);
   ```

Properties

`Camera` — Camera configuration
`monoCamera` object

This property is read-only.

Camera configuration, specified as a `monoCamera` object. The object contains the camera intrinsics, the location, the pitch, yaw, and roll placement, and the world units for the parameters. Use the intrinsics to transform the object points in the image to world coordinates, which you can then compare to the values in the `WorldObjectSize` property.

`WorldObjectSize` — Range of object widths and lengths
[*minWidth maxWidth*] vector | [*minWidth maxWidth*; *minLength maxLength*] vector

Range of object widths and lengths in world units, specified as a [*minWidth maxWidth*] vector or [*minWidth maxWidth*; *minLength maxLength*] vector. Specifying the range of object lengths is optional.

`ModelName` — Name of classification model
character vector | string scalar

Name of the classification model, specified as a character vector or string scalar. You can modify this name after creating the `ssdObjectDetectorMonoCamera` object.

`Network` — Trained SSD object detection network
`DAGNetwork` object

This property is read-only.

Trained SSD object detection network, specified as a `DAGNetwork` object. This object stores the layers that are used within the SSD object detector.

`AnchorBoxes` — Size of anchor boxes
P-by-1 cell array

This property is read-only.

Size of anchor boxes, specified as a *P*-by-1 cell array for *P* number of feature extraction layers used for object detection in the SSD network. Each element of the array contains an *M*-by-2 matrix of anchor box sizes, in the format [*height width*]. Each cell can contain a different number of anchor boxes. This value is set during training.

`ClassNames` — Names of object classes
cell array of character vectors

This property is read-only.

Names of the object classes that the SSD object detector was trained to find, specified as a cell array of character vectors. This property is set by the `trainingData` input argument for the `trainSSDObjectDetector` function. Specify the class names as part of the `trainingData` table.

Object Functions

detect   Detect objects using SSD multibox object detector

Examples

Detect Vehicles Using Monocular Camera and SSD

Configure a SSD object detector for use with a monocular camera mounted on an ego vehicle. Use this detector to detect vehicles within an image captured by the camera.

Load a `ssdObjectDetector` object pretrained to detect vehicles.

```
vehicleDetector = load('ssdVehicleDetector.mat','detector');
detector = vehicleDetector.detector;
```

Model a monocular camera sensor by creating a `monoCamera` object. This object contains the camera intrinsics and the location of the camera on the ego vehicle.

```
focalLength = [309.4362 344.2161];    % [fx fy]
principalPoint = [318.9034 257.5352]; % [cx cy]
imageSize = [480 640];                % [mrows ncols]
height = 2.1798;                      % height of camera above ground, in meters
pitch = 14;                           % pitch of camera, in degrees
intrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);

sensor = monoCamera(intrinsics,height,'Pitch',pitch);
```

Configure the detector for use with the camera. Limit the width of detected objects to 1.5 – 2.5 meters. The configured detector is a ssdObjectDetectorMonoCamera object.

```
vehicleWidth = [1.5 2.5];
detectorMonoCam = configureDetectorMonoCamera(detector,sensor,vehicleWidth);
```

Read an image captured by the camera.

```
I = imread('highwayCars.png');
```

Detect the vehicles in the image by using the detector. Annotate the image with the bounding boxes for the detections and the detection confidence scores.

```
[bboxes,scores,labels] = detect(detectorMonoCam,I,'Threshold',0.6);
I = insertObjectAnnotation(I,'rectangle',bboxes,scores,'Color','g');
imshow(I)
```

Display the labels for detected bounding boxes. The labels specify the class names of the detected objects.

```
disp(labels)
```

```
     vehicle      vehicle
```

See Also

Apps
Ground Truth Labeler

Functions
`configureDetectorMonoCamera` | `trainSSDObjectDetector`

Objects
`monoCamera` | `ssdObjectDetector`

Topics
"Getting Started with SSD Multibox Detection" (Computer Vision Toolbox)
"Create SSD Object Detection Network" (Computer Vision Toolbox)
"Object Detection Using SSD Deep Learning" (Computer Vision Toolbox)

Introduced in R2020a

detect

Detect objects using SSD object detector configured for monocular camera

Syntax

```
bboxes = detect(detector,I)
[bboxes,scores] = detect(detector,I)
[___,labels] = detect(detector,I)
[___] = detect(___,roi)

detectionResults = detect(detector,ds)

[___] = detect(___,Name,Value)
```

Description

`bboxes = detect(detector,I)` detects objects within image `I` using an SSD (singe shot detection convolutional neural networks) multibox object detector configured for a monocular camera. The locations of objects detected are returned as a set of bounding boxes.

When using this function, use of a CUDA-enabled NVIDIA GPU with a compute capability of 3.0 or higher is highly recommended. The GPU reduces computation time significantly. Usage of the GPU requires Parallel Computing Toolbox.

`[bboxes,scores] = detect(detector,I)` also returns the detection confidence scores for each bounding box.

`[___,labels] = detect(detector,I)` also returns a categorical array of labels assigned to the bounding boxes, using either of the preceding syntaxes. The labels used for object classes are defined during training using the `trainSSDObjectDetector` function.

`[___] = detect(___,roi)` detects objects within the rectangular search region specified by `roi`. Use output arguments from any of the previous syntaxes. Specify input arguments from any of the previous syntaxes.

`detectionResults = detect(detector,ds)` detects objects within the series of images returned by the `read` function of the input datastore.

`[___] = detect(___,Name,Value)` specifies options using one or more `Name,Value` pair arguments. For example, `detect(detector,I,'Threshold',0.75)` sets the detection score threshold to `0.75`. Any detections with a lower score are removed.

Examples

Detect Vehicles Using Monocular Camera and SSD

Configure a SSD object detector for use with a monocular camera mounted on an ego vehicle. Use this detector to detect vehicles within an image captured by the camera.

Load a `ssdObjectDetector` object pretrained to detect vehicles.

```
vehicleDetector = load('ssdVehicleDetector.mat','detector');
detector = vehicleDetector.detector;
```

Model a monocular camera sensor by creating a `monoCamera` object. This object contains the camera intrinsics and the location of the camera on the ego vehicle.

```
focalLength = [309.4362 344.2161];    % [fx fy]
principalPoint = [318.9034 257.5352]; % [cx cy]
imageSize = [480 640];                % [mrows ncols]
height = 2.1798;                      % height of camera above ground, in meters
pitch = 14;                           % pitch of camera, in degrees
intrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);

sensor = monoCamera(intrinsics,height,'Pitch',pitch);
```

Configure the detector for use with the camera. Limit the width of detected objects to 1.5 – 2.5 meters. The configured detector is a ssd`ObjectDetectorMonoCamera` object.

```
vehicleWidth = [1.5 2.5];
detectorMonoCam = configureDetectorMonoCamera(detector,sensor,vehicleWidth);
```

Read an image captured by the camera.

```
I = imread('highwayCars.png');
```

Detect the vehicles in the image by using the detector. Annotate the image with the bounding boxes for the detections and the detection confidence scores.

```
[bboxes,scores,labels] = detect(detectorMonoCam,I,'Threshold',0.6);
I = insertObjectAnnotation(I,'rectangle',bboxes,scores,'Color','g');
imshow(I)
```

Display the labels for detected bounding boxes. The labels specify the class names of the detected objects.

```
disp(labels)
```

```
     vehicle      vehicle
```

Input Arguments

detector — SSD multibox object detector
`SSDObjectDetector` object

SSD multibox object detector, specified as an `ssdObjectDetector` object. To create this object, call the `trainSSDObjectDetector` function with training data as input.

I — Input image
H-by-*W*-by-*C*-by-*B* numeric array of images

Input image, specified as an *H*-by-*W*-by-*C*-by-*B* numeric array of images Images must be real, nonsparse, grayscale or RGB image.

- *H*: Height
- *W*: Width
- *C*: The channel size in each image must be equal to the network's input channel size. For example, for grayscale images, *C* must be equal to `1`. For RGB color images, it must be equal to `3`.
- *B*: The number of images in the array.

The detector is sensitive to the range of the input image. Therefore, ensure that the input image range is similar to the range of the images used to train the detector. For example, if the detector was trained on `uint8` images, rescale this input image to the range [0, 255] by using the `im2uint8` or `rescale` function. The size of this input image should be comparable to the sizes of the images used in training. If these sizes are very different, the detector has difficulty detecting objects because the scale of the objects in the input image differs from the scale of the objects the detector was trained to identify. Consider whether you used the `SmallestImageDimension` property during training to modify the size of training images.

Data Types: `uint8` | `uint16` | `int16` | `double` | `single` | `logical`

ds — Datastore
`datastore` object

Datastore, specified as a datastore object containing a collection of images. Each image must be a grayscale, RGB, or multichannel image. The function processes only the first column of the datastore, which must contain images and must be cell arrays or tables with multiple columns.

`roi` — Search region of interest
[*x y width height*] vector

Search region of interest, specified as an [*x y width height*] vector. The vector specifies the upper left corner and size of a region in pixels.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'SelectStrongest',true`

`Threshold` — Detection threshold
`0.5` (default) | scalar

Detection threshold, specified as a scalar in the range [0, 1]. Detections that have scores less than this threshold value are removed. To reduce false positives, increase this value.

`SelectStrongest` — Select strongest bounding box
`true` (default) | `false`

Select the strongest bounding box for each detected object, specified as the comma-separated pair consisting of `'SelectStrongest'` and either `true` or `false`.

- `true` — Return the strongest bounding box per object. To select these boxes, `detect` calls the `selectStrongestBboxMulticlass` function, which uses nonmaximal suppression to eliminate overlapping bounding boxes based on their confidence scores.

  For example:

```
selectStrongestBboxMulticlass(bbox,scores, ...
            'RatioType','Min', ...
            'OverlapThreshold',0.5);
```

- `false` — Return all detected bounding boxes. You can then create your own custom operation to eliminate overlapping bounding boxes.

MaxSize — Maximum region size
`size(I)` (default) | [*height width*] vector

Maximum region size that contains a detected object, specified as the comma-separated pair consisting of `'MaxSize'` and a [*height width*] vector. Units are in pixels.

To reduce computation time, set this value to the known maximum region size for the objects being detected in the image. By default, `'MaxSize'` is set to the height and width of the input image, `I`.

MiniBatchSize — Minimum batch size
128 (default) | scalar

Minimum batch size, specified as the comma-separated pair consisting of `'MiniBatchSize'` and a scalar value. Use the `MiniBatchSize` to process a large collection of images. Images are grouped into minibatches and processed as a batch to improve computation efficiency. Increase the minibatch size to decrease processing time. Decrease the size to use less memory.

ExecutionEnvironment — Hardware resource
`'auto'` (default) | `'gpu'` | `'cpu'`

Hardware resource on which to run the detector, specified as the comma-separated pair consisting of `'ExecutionEnvironment'` and `'auto'`, `'gpu'`, or `'cpu'`.

- `'auto'` — Use a GPU if it is available. Otherwise, use the CPU.
- `'gpu'` — Use the GPU. To use a GPU, you must have Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with a compute capability of 3.0 or higher. If a suitable GPU is not available, the function returns an error.
- `'cpu'` — Use the CPU.

Output Arguments

bboxes — Location of objects detected
M-by-4 matrix | *B*-by-1 cell array

Location of objects detected within the input image or images, returned as an *M*-by-4 matrix or a *B*-by-1 cell array. *M* is the number of bounding boxes in an image, and *B* is the number of *M*-by-4 matrices when the input contains an array of images.

Each row of `bboxes` contains a four-element vector of the form [*x y width height*]. This vector specifies the upper left corner and size of that corresponding bounding box in pixels.

scores — Detection scores
M-by-1 vector | *B*-by-1 cell array

Detection confidence scores, returned as an *M*-by-1 vector or a *B*-by-1 cell array. *M* is the number of bounding boxes in an image, and *B* is the number of *M*-by-1 vectors when the input contains an array of images. A higher score indicates higher confidence in the detection.

`labels` — Labels for bounding boxes
M-by-1 categorical array | *B*-by-1 cell array

Labels for bounding boxes, returned as an *M*-by-1 categorical array or a *B*-by-1 cell array. *M* is the number of labels in an image, and *B* is the number of *M*-by-1 categorical arrays when the input contains an array of images. You define the class names used to label the objects when you train the input `detector`.

`detectionResults` — Detection results
3-column table

Detection results, returned as a 3-column table with variable names, *Boxes*, *Scores*, and *Labels*. The *Boxes* column contains *M*-by-4 matrices, of *M* bounding boxes for the objects found in the image. Each row contains a bounding box as a 4-element vector in the format [*x*,*y*,*width*,*height*]. The format specifies the upper-left corner location and size in pixels of the bounding box in the corresponding image.

See Also

Apps
Ground Truth Labeler

Functions
`configureDetectorMonoCamera` | `evaluateDetectionMissRate` | `evaluateDetectionPrecision` | `selectStrongestBboxMulticlass`

Objects
`monoCamera`

Topics
"Object Detection Using SSD Deep Learning" (Computer Vision Toolbox)
"Create SSD Object Detection Network" (Computer Vision Toolbox)
"Datastores for Deep Learning" (Deep Learning Toolbox)

Introduced in R2020a

pathPlannerRRT

Configure RRT* path planner

Description

The `pathPlannerRRT` object configures a vehicle path planner based on the optimal rapidly exploring random tree (RRT*) algorithm. An RRT* path planner explores the environment around the vehicle by constructing a tree of random collision-free poses.

Once the `pathPlannerRRT` object is configured, use the `plan` function to plan a path from the start pose to the goal.

Creation

Syntax

```
planner = pathPlannerRRT(costmap)
planner = pathPlannerRRT(costmap,Name,Value)
```

Description

`planner = pathPlannerRRT(costmap)` returns a `pathPlannerRRT` object for planning a vehicle path. `costmap` is a `vehicleCostmap` object specifying the environment around the vehicle. `costmap` sets the `Costmap` property value.

`planner = pathPlannerRRT(costmap,Name,Value)` sets properties on page 4-892 of the path planner by using one or more name-value pair arguments. For example, `pathPlanner(costmap,'GoalBias',0.5)` sets the `GoalBias` property to a probability of 0.5. Enclose each property name in quotes.

Properties

`Costmap` — Costmap of vehicle environment
`vehicleCostmap` object

Costmap of the vehicle environment, specified as a `vehicleCostmap` object. The costmap is used for collision checking of the randomly generated poses. Specify this costmap when creating your `pathPlannerRRT` object using the `costmap` input.

`GoalTolerance` — Tolerance around goal pose
`[0.5 0.5 5]` (default) | [*xTol*, *yTol*, *ΘTol*] vector

Tolerance around the goal pose, specified as an [*xTol*, *yTol*, *ΘTol*] vector. The path planner finishes planning when the vehicle reaches the goal pose within these tolerances for the (*x*, *y*) position and the orientation angle, *Θ*. The *xTol* and *yTol* values are in the same world units as the `vehicleCostmap`. *ΘTol* is in degrees.

GoalBias — Probability of selecting goal pose
0.1 (default) | real scalar in the range [0, 1]

Probability of selecting the goal pose instead of a random pose, specified as a real scalar in the range [0, 1]. Large values accelerate reaching the goal at the risk of failing to circumnavigate obstacles.

ConnectionMethod — Method used to connect poses
'Dubins' (default) | 'Reeds-Shepp'

Method used to calculate the connection between consecutive poses, specified as `'Dubins'` or `'Reeds-Shepp'`. Use `'Dubins'` if only forward motions are allowed.

The `'Dubins'` method contains a sequence of three primitive motions, each of which is one of these types:

- Straight (forward)
- Left turn at the maximum steering angle of the vehicle (forward)
- Right turn at the maximum steering angle of the vehicle (forward)

If you use this connection method, then the segments of the planned vehicle path are stored as an array of `driving.DubinsPathSegment` objects.

The `'Reeds-Shepp'` method contains a sequence of three to five primitive motions, each of which is one of these types:

- Straight (forward or reverse)
- Left turn at the maximum steering angle of the vehicle (forward or reverse)
- Right turn at the maximum steering angle of the vehicle (forward or reverse)

If you use this connection method, then the segments of the planned vehicle path are stored as an array of `driving.ReedsSheppPathSegment` objects.

The `MinTurningRadius` property determines the maximum steering angle.

ConnectionDistance — Maximum distance between poses
5 (default) | positive real scalar

Maximum distance between two connected poses, specified as a positive real scalar. `pathPlannerRRT` computes the connection distance along the path between the two poses, with turns included. Larger values result in longer path segments between poses.

MinTurningRadius — Minimum turning radius of vehicle
4 (default) | positive real scalar

Minimum turning radius of the vehicle, specified as a positive real scalar. This value corresponds to the radius of the turning circle at the maximum steering angle. Larger values limit the maximum steering angle for the path planner, and smaller values result in sharper turns. The default value is calculated using a wheelbase of 2.8 meters with a maximum steering angle of 35 degrees.

MinIterations — Minimum number of planner iterations
100 (default) | positive integer

Minimum number of planner iterations for exploring the costmap, specified as a positive integer. Increasing this value increases the sampling of alternative paths in the costmap.

`MaxIterations` — Maximum number of planner iterations
`10000` (default) | positive integer

Maximum number of planner iterations for exploring the costmap, specified as a positive integer. Increasing this value increases the number of samples for finding a valid path. If a valid path is not found, the path planner exits after exceeding this maximum.

`ApproximateSearch` — Enable approximate nearest neighbor search
`true` (default) | `false`

Enable approximate nearest neighbor search, specified as `true` or `false`. Set this value to `true` to use a faster, but approximate, search algorithm. Set this value to `false` to use an exact search algorithm at the cost of increased computation time.

Object Functions

plan Plan vehicle path using RRT* path planner
plot Plot path planned by RRT* path planner

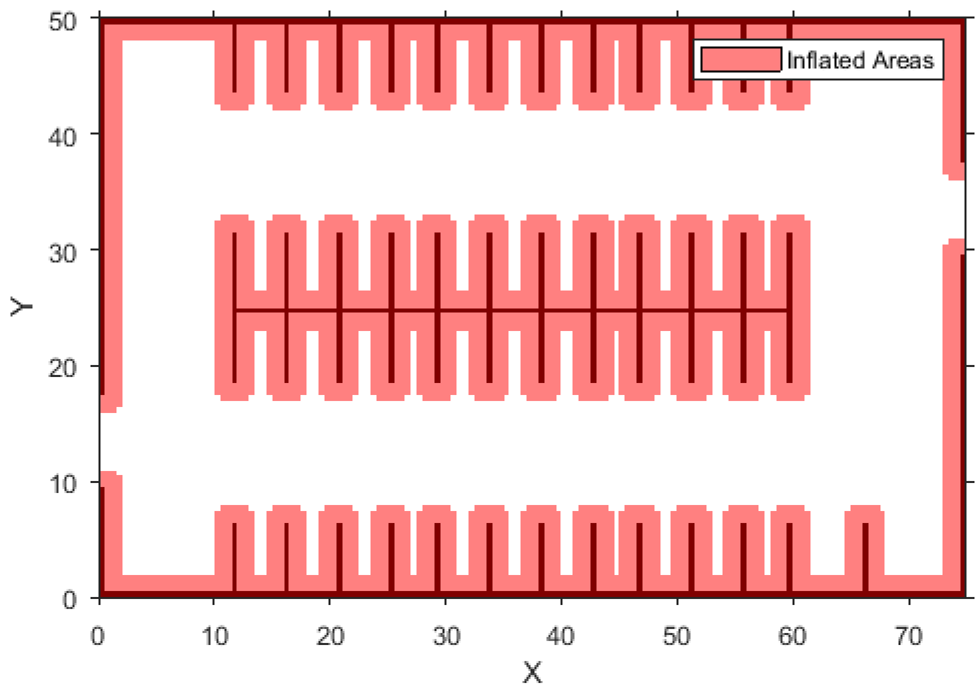
Examples

Plan Path to Parking Spot

Plan a vehicle path to a parking spot by using the RRT* algorithm.

Load a costmap of a parking lot. Plot the costmap to see the parking lot and inflated areas for the vehicle to avoid.

```
data = load('parkingLotCostmapReducedInflation.mat');
costmap = data.parkingLotCostmapReducedInflation;
plot(costmap)
```



Define start and goal poses for the path planner as [*x*, *y*, *Θ*] vectors. World units for the (*x*,*y*) locations are in meters. World units for the *Θ* orientation values are in degrees.

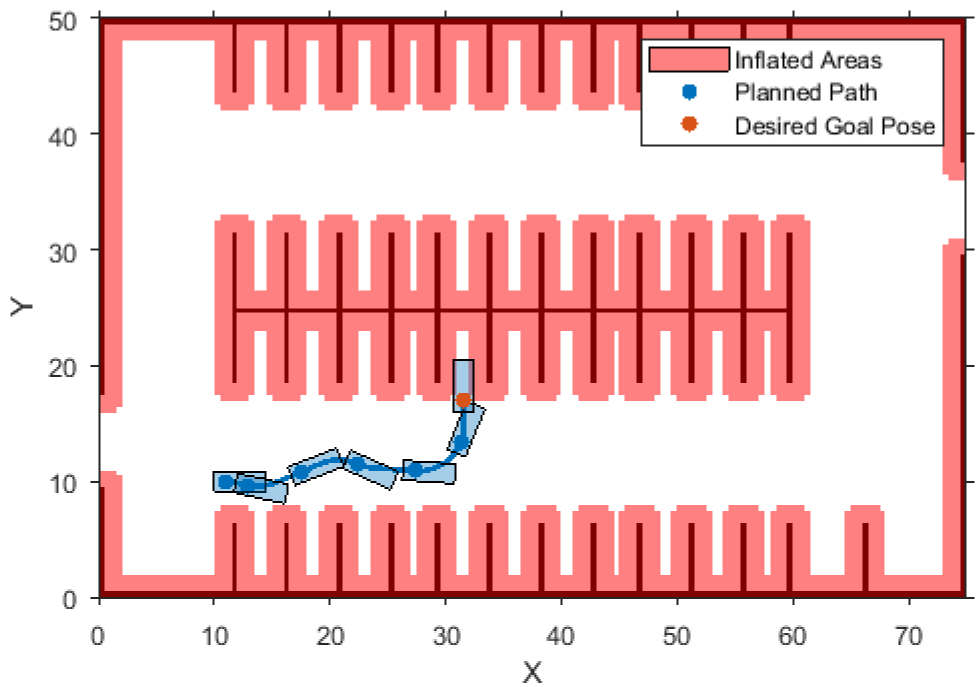
```
startPose = [11, 10, 0]; % [meters, meters, degrees]
goalPose  = [31.5, 17, 90];
```

Create an RRT* path planner to plan a path from the start pose to the goal pose.

```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Plot the planned path.

```
plot(planner)
```

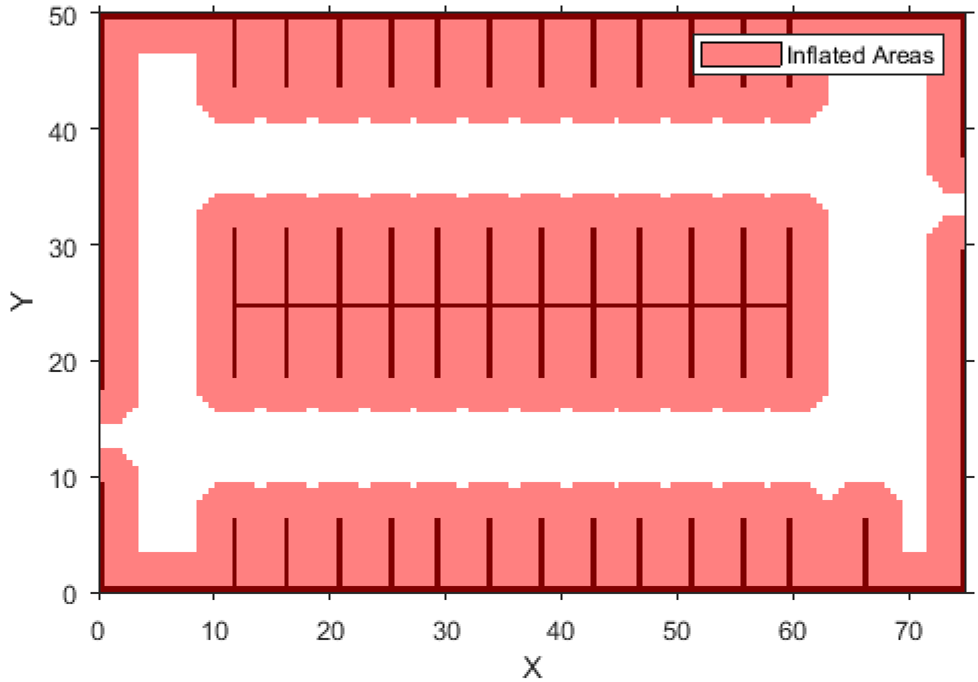


Plan Path and Check Its Validity

Plan a vehicle path through a parking lot by using the optimal rapidly exploring random tree (RRT*) algorithm. Check that the path is valid, and then plot the transition poses along the path.

Load a costmap of a parking lot. Plot the costmap to see the parking lot and inflated areas for the vehicle to avoid.

```
data = load('parkingLotCostmap.mat');
costmap = data.parkingLotCostmap;
plot(costmap)
```



Define start and goal poses for the vehicle as [*x*, *y*, *Θ*] vectors. World units for the (*x*,*y*) locations are in meters. World units for the *Θ* orientation angles are in degrees.

```
startPose = [4, 4, 90]; % [meters, meters, degrees]
goalPose = [30, 13, 0];
```

Use a `pathPlannerRRT` object to plan a path from the start pose to the goal pose.

```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Check that the path is valid.

```
isPathValid = checkPathValidity(refPath,costmap)
```

```
isPathValid = logical
   1
```

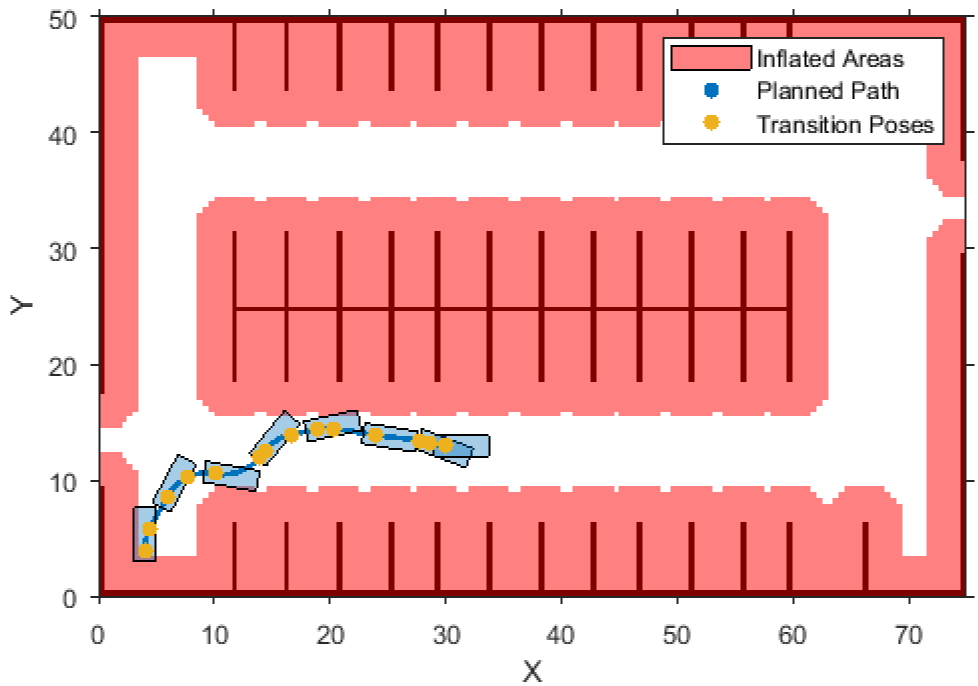
Interpolate the transition poses along the path.

```
transitionPoses = interpolate(refPath);
```

Plot the planned path and the transition poses on the costmap.

```
hold on
plot(refPath,'DisplayName','Planned Path')
scatter(transitionPoses(:,1),transitionPoses(:,2),[],'filled', ...
```

```
    'DisplayName','Transition Poses')
hold off
```



Tips

- Updating any of the properties of the planner clears the planned path from `pathPlannerRRT`. Calling `plot` displays only the costmap until a path is planned using `plan`.
- To improve performance, the `pathPlannerRRT` object uses an approximate nearest neighbor search. This search technique checks only `sqrt(N)` nodes, where `N` is the number of nodes to search. To use exact nearest neighbor search, set the `ApproximateSearch` property to `false`.
- The Dubins and Reeds-Shepp connection methods are assumed to be kinematically feasible and ignore inertial effects. These methods make the path planner suitable for low velocity environments, where inertial effects of wheel forces are small.

References

[1] Karaman, Sertac, and Emilio Frazzoli. "Optimal Kinodynamic Motion Planning Using Incremental Sampling-Based Methods." *49th IEEE Conference on Decision and Control (CDC)*. 2010.

[2] Shkel, Andrei M., and Vladimir Lumelsky. "Classification of the Dubins Set." *Robotics and Autonomous Systems*. Vol. 34, Number 4, 2001, pp. 179–202.

[3] Reeds, J. A., and L. A. Shepp. "Optimal paths for a car that goes both forwards and backwards." *Pacific Journal of Mathematics*. Vol. 145, Number 2, 1990, pp. 367–393.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `ConnectionMethod`, `MinIterations`, `MaxIterations`, and `ApproximateSearch` properties must be compile-time constants.

See Also

Functions
`checkPathValidity` | `lateralControllerStanley` | `plan` | `plot` | `smoothPathSpline`

Blocks
Lateral Controller Stanley

Objects
`driving.Path` | `vehicleCostmap`

Topics
"Automated Parking Valet"

Introduced in R2018a

plan

Plan vehicle path using RRT* path planner

Syntax

```
refPath = plan(planner,startPose,goalPose)
[refPath,tree] = plan(planner,startPose,goalPose)
```

Description

`refPath = plan(planner,startPose,goalPose)` plans a vehicle path from `startPose` to `goalPose` using the input `pathPlannerRRT` object. This object configures an optimal rapidly exploring random tree (RRT*) path planner.

`[refPath,tree] = plan(planner,startPose,goalPose)` also returns the exploration tree, `tree`.

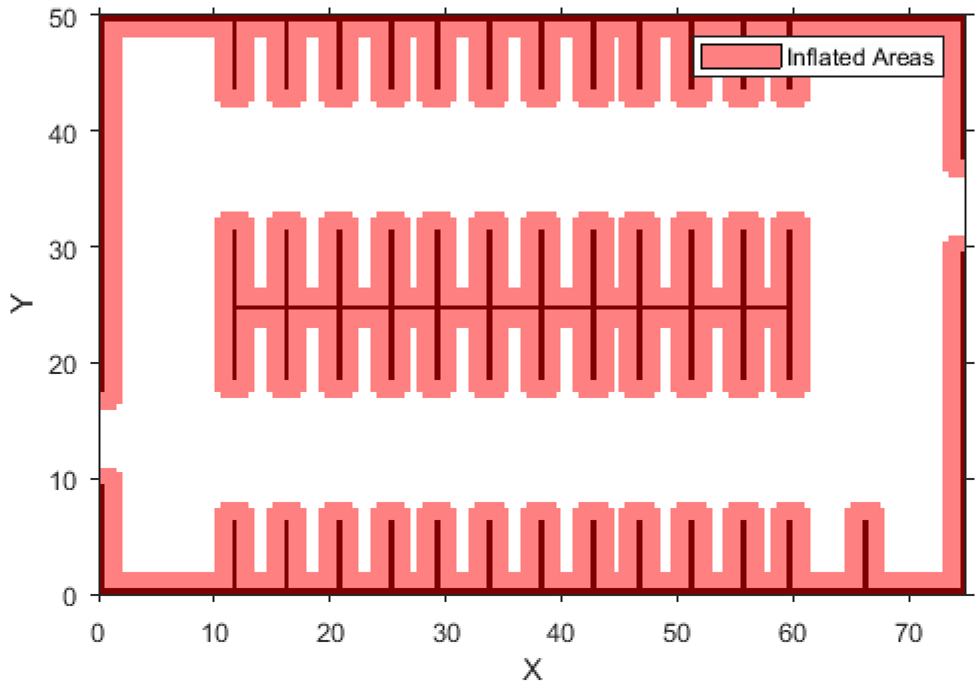
Examples

Plan Path to Parking Spot

Plan a vehicle path to a parking spot by using the RRT* algorithm.

Load a costmap of a parking lot. Plot the costmap to see the parking lot and inflated areas for the vehicle to avoid.

```
data = load('parkingLotCostmapReducedInflation.mat');
costmap = data.parkingLotCostmapReducedInflation;
plot(costmap)
```



Define start and goal poses for the path planner as [*x*, *y*, *Θ*] vectors. World units for the (*x*,*y*) locations are in meters. World units for the *Θ* orientation values are in degrees.

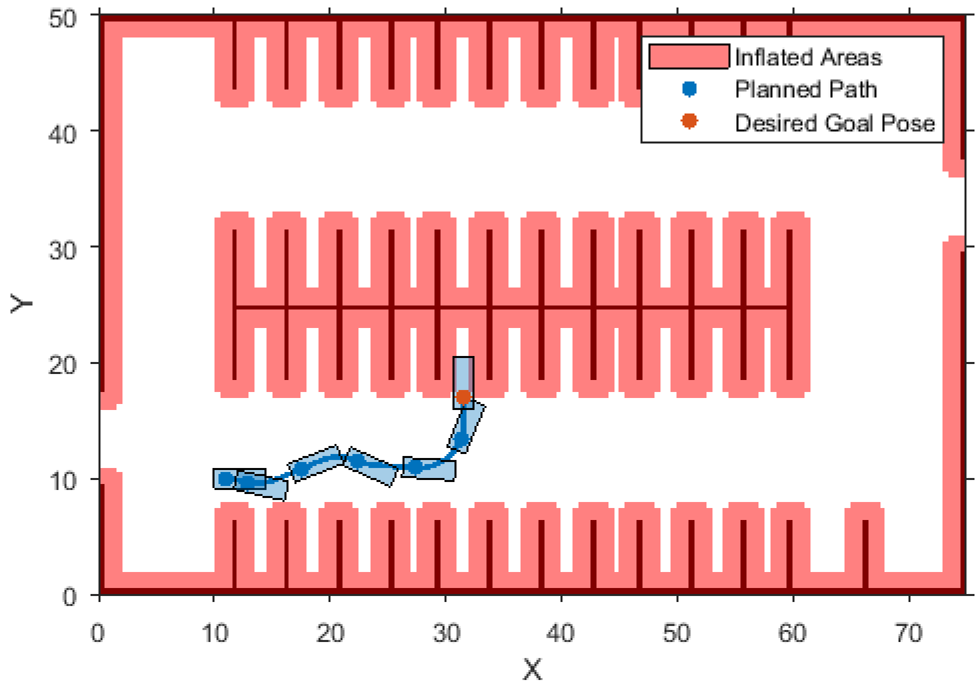
```
startPose = [11, 10, 0]; % [meters, meters, degrees]
goalPose  = [31.5, 17, 90];
```

Create an RRT* path planner to plan a path from the start pose to the goal pose.

```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Plot the planned path.

```
plot(planner)
```



Input Arguments

planner — RRT* path planner
pathPlannerRRT object

RRT* path planner, specified as a `pathPlannerRRT` object.

startPose — Initial pose of vehicle
[*x*, *y*, *Θ*] vector

Initial pose of the vehicle, specified as an [*x*, *y*, *Θ*] vector. *x* and *y* are in world units, such as meters. *Θ* is in degrees.

goalPose — Goal pose of vehicle
[*x*, *y*, *Θ*] vector

Goal pose of the vehicle, specified as an [*x*, *y*, *Θ*] vector. *x* and *y* are in world units, such as meters. *Θ* is in degrees.

The vehicle achieves its goal pose when the last pose in the path is within the `GoalTolerance` property of `planner`.

Output Arguments

`refPath` — Planned vehicle path
`driving.Path` object

Planned vehicle path, returned as a `driving.Path` object containing reference poses along the planned path. If planning was unsuccessful, the path has no poses. To check if the path is still valid due to costmap updates, use the `checkPathValidity` function.

`tree` — Exploration tree
`digraph` object

Exploration tree, returned as a `digraph` object. Nodes within `tree` represent explored vehicle poses. Edges within `tree` represent the distance between connected nodes.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The optional `tree` output argument, a digraph object, is not supported.

See Also

Functions
`checkPathValidity` | `plot`

Objects
`digraph` | `driving.Path` | `pathPlannerRRT` | `vehicleCostmap`

Topics
"Automated Parking Valet"

Introduced in R2018a

plot

Plot path planned by RRT* path planner

Syntax

```
plot(planner)
plot(planner,Name,Value)
```

Description

`plot(planner)` plots the path planned by the input `pathPlannerRRT` object. When specified as an input to the `plan` function, this object plans a path using the rapidly exploring random tree (RRT*) algorithm. If a path has not been planned using `plan`, or if properties of the `pathPlannerRRT` planner have changed since using `plan`, then `plot` displays only the costmap of `planner`.

`plot(planner,Name,Value)` specifies options using one or more name-value pair arguments. For example, `plot(planner,'Tree','on')` plots the poses explored by the RRT* path planner.

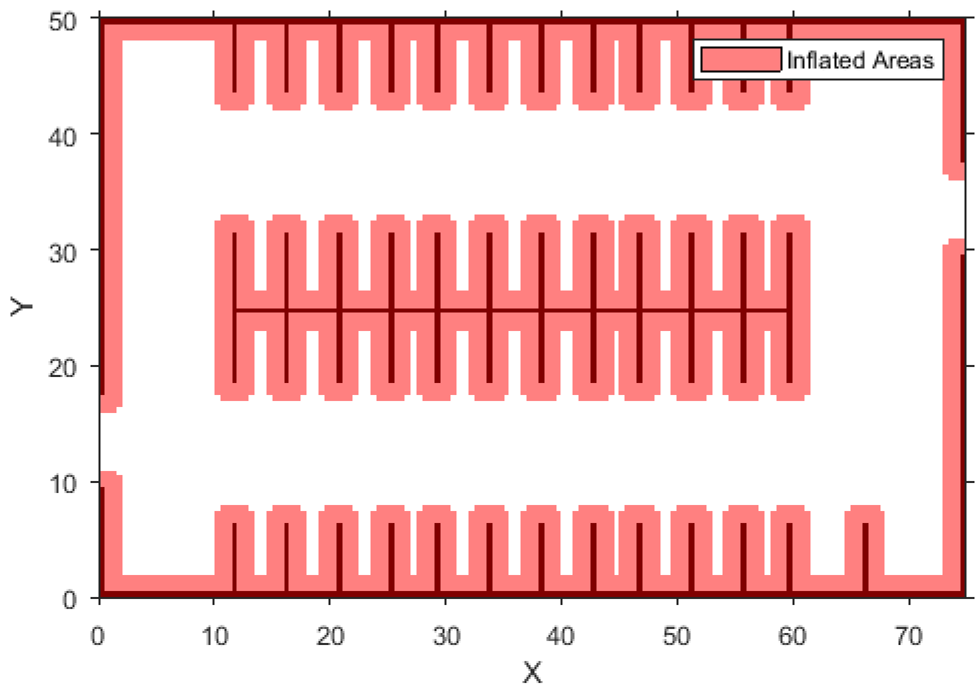
Examples

Plan Path to Parking Spot

Plan a vehicle path to a parking spot by using the RRT* algorithm.

Load a costmap of a parking lot. Plot the costmap to see the parking lot and inflated areas for the vehicle to avoid.

```
data = load('parkingLotCostmapReducedInflation.mat');
costmap = data.parkingLotCostmapReducedInflation;
plot(costmap)
```



Define start and goal poses for the path planner as [*x*, *y*, *Θ*] vectors. World units for the (*x*,*y*) locations are in meters. World units for the *Θ* orientation values are in degrees.

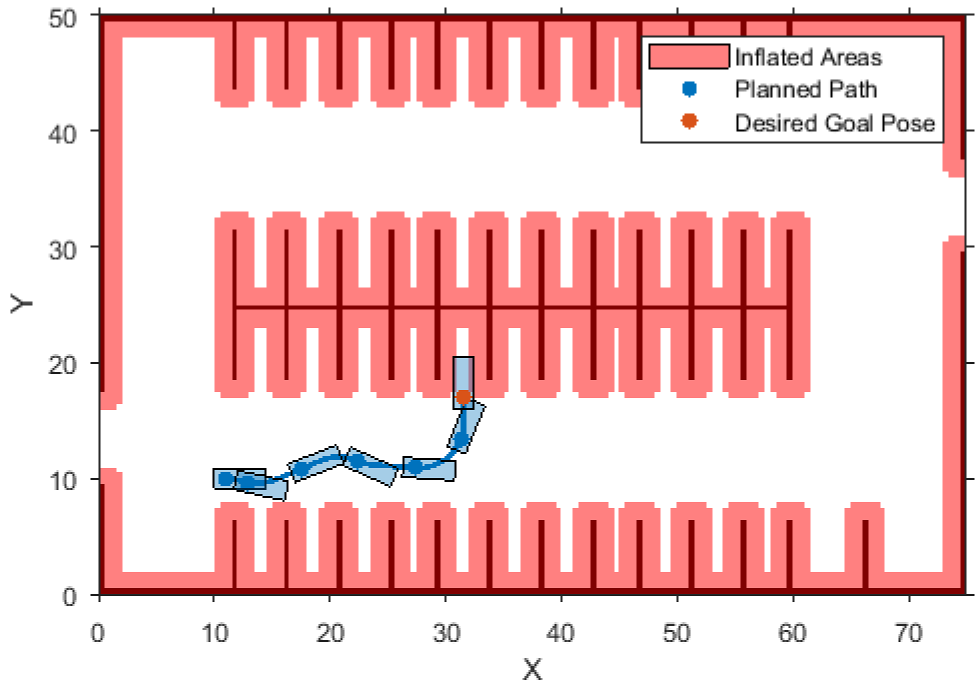
```
startPose = [11, 10, 0]; % [meters, meters, degrees]
goalPose  = [31.5, 17, 90];
```

Create an RRT* path planner to plan a path from the start pose to the goal pose.

```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Plot the planned path.

```
plot(planner)
```



Input Arguments

planner — RRT* path planner
`pathPlannerRRT` object

RRT* path planner, specified as a `pathPlannerRRT` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Vehicle','off'`

Parent — Axes object
`axes` object

Axes object in which to draw the plot, specified as the comma-separated pair consisting of `'Parent'` and an `axes` object. If you do not specify `Parent`, a new figure is created.

Tree — Display exploration tree
`'off'` (default) | `'on'`

Display exploration tree, specified as the comma-separated pair consisting of `'Tree'` and `'off'` or `'on'`. Setting this value to `'on'` displays the poses explored by the RRT* path planner, `planner`.

`Vehicle` — Display vehicle
`'on'` (default) | `'off'`

Display vehicle, specified as the comma-separated pair consisting of `'Vehicle'` and `'on'` or `'off'`. Setting this value to `'off'` disables the vehicle displayed along the path planned by the RRT* path planner, `planner`.

See Also

Functions
`checkPathValidity` | `plan`

Objects
`driving.Path` | `pathPlannerRRT` | `vehicleCostmap`

Topics
"Automated Parking Valet"

Introduced in R2018a

vehicleCostmap

Costmap representing planning space around vehicle

Description

The `vehicleCostmap` object creates a costmap that represents the planning search space around a vehicle. The costmap holds information about the environment, such as obstacles or areas that the vehicle cannot traverse. To check for collisions, the costmap inflates obstacles using the inflation radius specified in the `CollisionChecker` property. The costmap is used by path planning algorithms, such as `pathPlannerRRT`, to find collision-free paths for the vehicle to follow.

The costmap is stored as a 2-D grid of cells, often called an occupancy grid or occupancy map. Each grid cell in the costmap has a value in the range [0, 1] representing the cost of navigating through that grid cell. The state of each grid cell is free, occupied, or unknown, as determined by the `FreeThreshold` and `OccupiedThreshold` properties.

The following figure shows a costmap with sample costs and grid cell states.

Creation

Syntax

```
costmap = vehicleCostmap(C)
costmap = vehicleCostmap(mapWidth,mapLength)
costmap = vehicleCostmap(mapWidth,mapLength,costVal)
costmap = vehicleCostmap(occMap)
costmap = vehicleCostmap(___,'MapLocation',mapLocation)
costmap = vehicleCostmap(___,Name,Value)
```

Description

`costmap = vehicleCostmap(C)` creates a vehicle costmap using the cost values in matrix `C`.

costmap = vehicleCostmap(mapWidth,mapLength) creates a vehicle costmap representing an area of width mapWidth and length mapLength in world units. By default, each grid cell is in the unknown state.

costmap = vehicleCostmap(mapWidth,mapLength,costVal) also assigns a default cost, costVal, to each cell in the grid.

costmap = vehicleCostmap(occMap) creates a vehicle costmap from the occupancy map occMap. Use of this syntax requires Navigation Toolbox™.

costmap = vehicleCostmap(___ ,'MapLocation',mapLocation) specifies in mapLocation the bottom-left corner coordinates of the costmap. Specify 'MapLocation',mapLocation after any of the preceding inputs and in any order among the Name,Value pair arguments.

costmap = vehicleCostmap(___ ,Name,Value) uses Name,Value pair arguments to specify the FreeThreshold, OccupiedThreshold, CollisionChecker, and CellSize properties. For example, vehicleCostmap(C,'CollisionChecker',ccConfig) uses an inflationCollisionChecker object, ccConfig, to represent the vehicle shape and check for collisions. After you create the object, you can update all of these properties except CellSize.

Input Arguments

C — Cost values

matrix of real values in the range [0, 1]

Cost values, specified as a matrix of real values that are in the range [0, 1].

When creating a vehicleCostmap object, if you do not specify C or a uniform cost value, costVal, then the default cost value of each grid cell is (FreeThreshold + OccupiedThreshold)/2.

Data Types: single | double

mapWidth — Width of costmap

positive real scalar

Width of costmap, in world units, specified as a positive real scalar.

mapLength — Length of costmap

positive real scalar

Length of costmap, in world units, specified as a positive real scalar.

costVal — Uniform cost value

real scalar in the range [0, 1]

Uniform cost value applied to all cells in the costmap, specified as a real scalar in the range [0, 1].

When creating a vehicleCostmap object, if you do not specify costVal or a cost value matrix, C, then the default cost value of each grid cell is (FreeThreshold + OccupiedThreshold)/2.

occMap — Occupancy map

occupancyMap object | binaryOccupancyMap object

Occupancy map, specified as an occupancyMap or binaryOccupancyMap object. Use of this argument requires Navigation Toolbox.

`mapLocation` — Costmap location
`[0 0]` (default) | two-element real-valued vector of form [*mapX mapY*]

Costmap location, specified as a two-element real-valued vector of the form [*mapX mapY*]. This vector specifies the coordinate location of the bottom-left corner of the costmap.

Example: `'MapLocation',[8 8]`

Properties

`FreeThreshold` — Threshold below which grid cell is free
`0.2` (default) | real scalar in the range [0, 1]

Threshold below which a grid cell is free, specified as a real scalar in the range [0, 1].

A grid cell with cost *c* can have one of these states:

- If *c* < `FreeThreshold`, the grid cell state is *free*.
- If *c* ≥ `FreeThreshold` and *c* ≤ `OccupiedThreshold`, the grid cell state is *unknown*.
- If *c* > `OccupiedThreshold`, the grid cell state is *occupied*.

`OccupiedThreshold` — Threshold above which grid cell is occupied
`0.65` (default) | real scalar in the range [0, 1]

Threshold above which a grid cell is occupied, specified as a real scalar in the range [0, 1].

A grid cell with cost *c* can have one of these states:

- If *c* < `FreeThreshold`, the grid cell state is *free*.
- If *c* ≥ `FreeThreshold` and *c* ≤ `OccupiedThreshold`, the grid cell state is *unknown*.
- If *c* > `OccupiedThreshold`, the grid cell state is *occupied*.

`CollisionChecker` — Collision-checking configuration
`inflationCollisionChecker()` (default) | `InflationCollisionChecker` object

Collision-checking configuration, specified as an `InflationCollisionChecker` object. To create this object, use the `inflationCollisionChecker` function. Using the properties of the `InflationCollisionChecker` object, you can configure:

- The inflation radius used to inflate obstacles in the costmap
- The number of circles used to enclose the vehicle when calculating the inflation radius
- The placement of each circle along the longitudinal axis of the vehicle
- The dimensions of the vehicle

By default, `CollisionChecker` uses the default `InflationCollisionChecker` object, which is created using the syntax `inflationCollisionChecker()`. This collision-checking configuration encloses the vehicle in one circle.

`MapExtent` — Extent of costmap
four-element, nonnegative integer vector of form [*xmin xmax ymin ymax*]

This property is read-only.

Extent of costmap around the vehicle, specified as a four-element, nonnegative integer vector of the form [*xmin xmax ymin ymax*].

- *xmin* and *xmax* describe the length of the map in world coordinates.
- *ymin* and *ymax* describe the width of the map in world coordinates.

CellSize — Side length of each square cell
1 (default) | positive real scalar

Side length of each square cell, in world units, specified as a positive real scalar. For example, a side length of `1` implies a grid where each cell is a square of size 1-by-1 meters. Smaller values improve the resolution of the search space at the cost of increased memory consumption.

You can specify `CellSize` when you create the `vehicleCostmap` object. However, after you create the object, `CellSize` becomes read-only.

MapSize — Size of costmap grid
two-element, positive integer vector of form [*nrows ncols*]

This property is read-only.

Size of costmap grid, specified as a two-element, positive integer vector of the form [*nrows ncols*].

- *nrows* is the number of grid cell rows in the costmap.
- *ncols* is the number of grid cell columns in the costmap.

Object Functions

checkFree	Check vehicle costmap for collision-free poses or points
checkOccupied	Check vehicle costmap for occupied poses or points
getCosts	Get cost value of cells in vehicle costmap
setCosts	Set cost value of cells in vehicle costmap
plot	Plot vehicle costmap

Examples

Create and Populate a Vehicle Costmap

Create a 10-by-20 meter costmap that is divided into square cells of size 0.5-by-0.5 meters. Specify a default cost value of 0.5 for all cells.

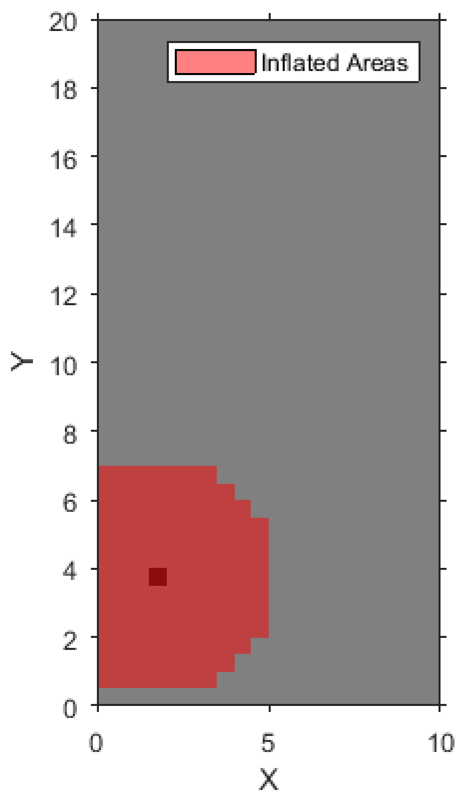
```
mapWidth = 10;
mapLength = 20;
costVal = 0.5;
cellSize = 0.5;

costmap = vehicleCostmap(mapWidth,mapLength,costVal,'CellSize',cellSize)
```

```
costmap = 
  vehicleCostmap with properties:

        FreeThreshold: 0.2000
    OccupiedThreshold: 0.6500
     CollisionChecker: [1x1 driving.costmap.InflationCollisionChecker]
```

```
      CellSize: 0.5000
       MapSize: [40 20]
     MapExtent: [0 10 0 20]
```

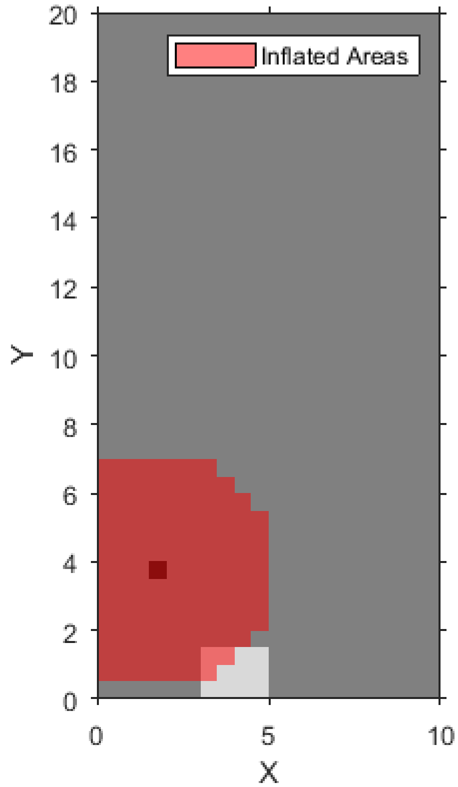
Mark an obstacle on the costmap. Display the costmap.

```
occupiedVal = 0.9;
xyPoint = [2,4];
setCosts(costmap,xyPoint,occupiedVal)

plot(costmap)
```



Mark an obstacle-free area on the costmap. Display the costmap again.

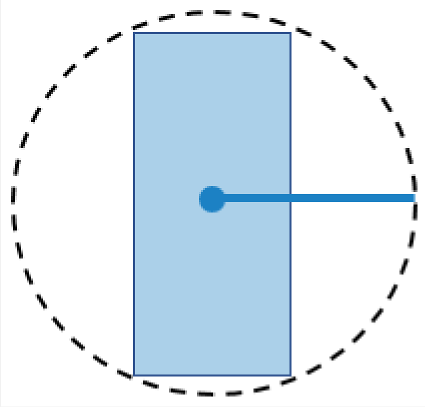
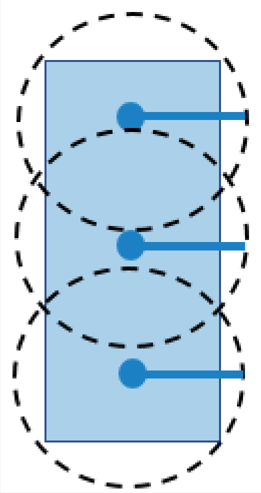
```
freeVal = 0.15;
[X,Y] = meshgrid(3.5:cellSize:5,0.5:cellSize:1.5);
setCosts(costmap,[X(:),Y(:)],freeVal)
plot(costmap)
```



Algorithms

To simplify checking for whether a vehicle pose is in collision, `vehicleCostmap` inflates the size of obstacles. The collision-checking algorithm follows these steps:

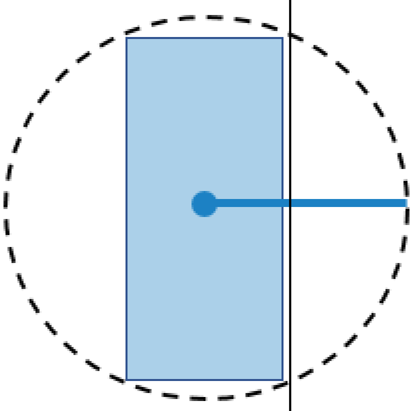
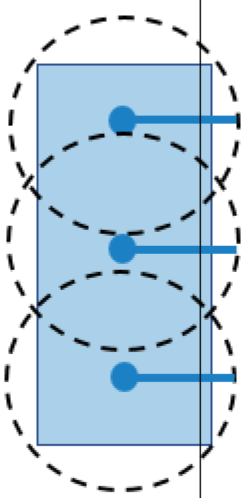
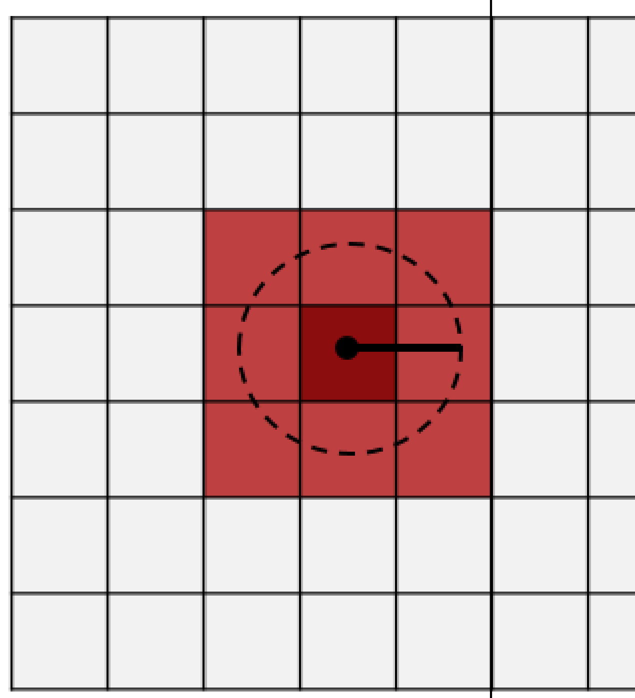
1. Calculate the inflation radius, in world units, from the vehicle dimensions. The default inflation radius is equal to the radius of the smallest set of overlapping circles required to completely enclose the vehicle. The center points of the circles lie along the longitudinal axis of the vehicle. Increasing the number of circles decreases the inflation radius, which enables more precise collision checking.

Inflation Radius, One Center	Inflation Radius, Three Centers
	

2 Convert the inflation radius to a number of grid cells, *R*. Round up noninteger values of *R* to the next largest integer.

3 Inflate the size of obstacles using *R*. Label all cells in the inflated area as occupied.

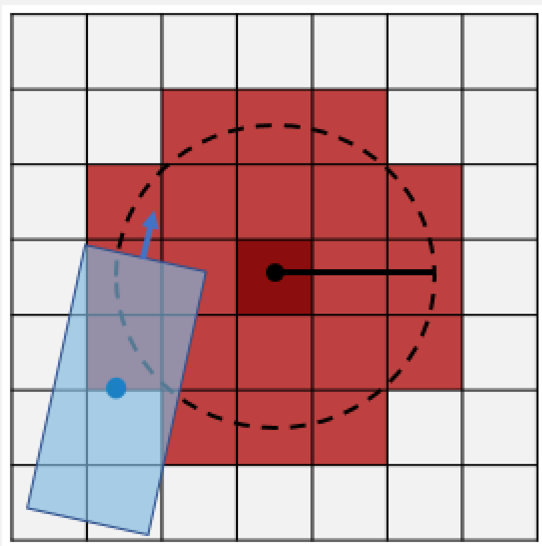
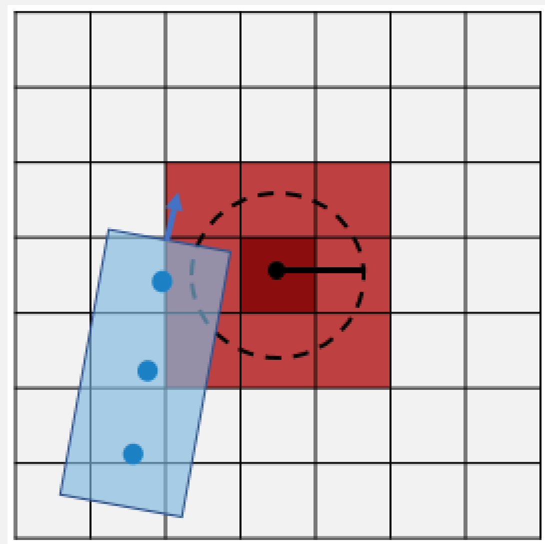
The diagrams show occupied cells in dark red. Cells in the inflated area are colored in light red. The solid black line shows the original inflation radius. In the diagram on the left, *R* is 3. In the diagram on the right, *R* is 2.

Inflated Grid Cells, One Center	Inflated Grid Cells, Three Centers
  	

4 Check whether the center points of the vehicle lie on inflated grid cells.

- If any center point lies on an inflated grid cell, then the vehicle pose is *occupied*. The `checkOccupied` function returns `true`. An occupied pose does not necessarily mean a collision. For example, the vehicle might lie on an inflated grid cell but not on the grid cell that is actually occupied.
- If no center points lie on inflated grid cells, and the cost value of each cell containing a center point is less than `FreeThreshold`, then the vehicle pose is *free*. The `checkFree` function returns `true`.
- If no center points lie on inflated grid cells, and the cost value of any cell containing a center point is greater than `FreeThreshold`, then the vehicle pose is *unknown*. Both `checkFree` and `checkOccupied` return `false`.

The following poses are considered in collision because at least one center point is on an inflated area.

Pose in Collision, One Center	Pose in Collision, Three Centers
	

Compatibility Considerations

`InflationRadius` and `VehicleDimensions` properties have been removed

Errors starting in R2020b

The `InflationRadius` and `VehicleDimensions` properties of the `vehicleCostmap` object have been removed. Follow this process instead:

1. Use the `inflationCollisionChecker` function to create an `InflationCollisionChecker` object, which has the `InflationRadius` and `VehicleDimensions` properties.
2. Specify this object as the value of the `CollisionChecker` property of `vehicleCostmap`.

If you do specify these properties for `vehicleCostmap`, the object returns an error.

When the `vehicleCostmap` object was introduced in R2018a, this object inflated obstacles based on the specified inflation radius and vehicle dimensions only. The `InflationCollisionChecker` object, which is specified in the `CollisionChecker` property of `vehicleCostmap`, provides additional configuration options for inflating obstacles. For example, you can specify the number of circles used to compute the inflation radius, enabling more precise collision checking.

Update Code

The table shows a typical usage of the `InflationRadius` and `VehicleDimensions` properties of `vehicleCostmap`. It also shows how to update your code using the corresponding properties of an `InflationCollisionChecker` object.

Invalid Usage	Recommended Replacement

`vehicleDims = vehicleDimensions(5,2);`
`inflationRadius = 1.2;`
`costmap = vehicleCostmap(C, ...`
`    'VehicleDimensions',vehicleDims, ...`
`    'InflationRadius',inflationRadius);`
	`vehicleDims = vehicleDimensions(5,2);`
	`inflationRadius = 1.2;`
	`ccConfig = inflationCollisionChecker(vehicleDims, ...`
	`    'InflationRadius',inflationRadius);`
	`costmap = vehicleCostmap(C, ...`
	`    'CollisionChecker',ccConfig);`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `occupancyMap` and `binaryOccupancyMap` inputs are not supported.
- The collision-checking configuration stored in the `CollisionChecker` property must be a compile-time constant.
- The `mapLocation` input argument must be a compile-time constant.

See Also
`inflationCollisionChecker` | `pathPlannerRRT`

Topics
"Automated Parking Valet"
"Create Occupancy Grid Using Monocular Camera and Semantic Segmentation"

Introduced in R2018a

checkFree

Check vehicle costmap for collision-free poses or points

Syntax

```
free = checkFree(costmap,vehiclePoses)
free = checkFree(costmap,xyPoints)
freeMat = checkFree(costmap)
```

Description

The `checkFree` function checks whether vehicle poses or points are free from obstacles on the vehicle costmap. Path planning algorithms use `checkFree` to check whether candidate vehicle poses along a path are navigable.

To simplify the collision check for a vehicle pose, `vehicleCostmap` inflates obstacles according to the vehicle's `InflationRadius`, as specified by the `CollisionChecker` property of the costmap. The collision checker calculates the inflation radius by enclosing the vehicle in a set of overlapping circles of radius *R*, where the centers of these circles lie along the longitudinal axis of the vehicle. The inflation radius is the minimum *R* needed to fully enclose the vehicle in these circles.

A vehicle pose is collision-free when the following conditions apply:

- None of the vehicle's circle centers lie on an inflated grid cell.
- The cost value of each containing a circle center is less than the `FreeThreshold` of the costmap.

For more details, see the algorithm on page 4-913 on the `vehicleCostmap` reference page.

`free = checkFree(costmap,vehiclePoses)` checks whether the vehicle poses are free from collision with obstacles on the costmap.

`free = checkFree(costmap,xyPoints)` checks whether (*x*, *y*) points in `xyPoints` are free from collision with obstacles on the costmap.

`freeMat = checkFree(costmap)` returns a logical matrix that indicates whether each cell of the costmap is free.

Examples

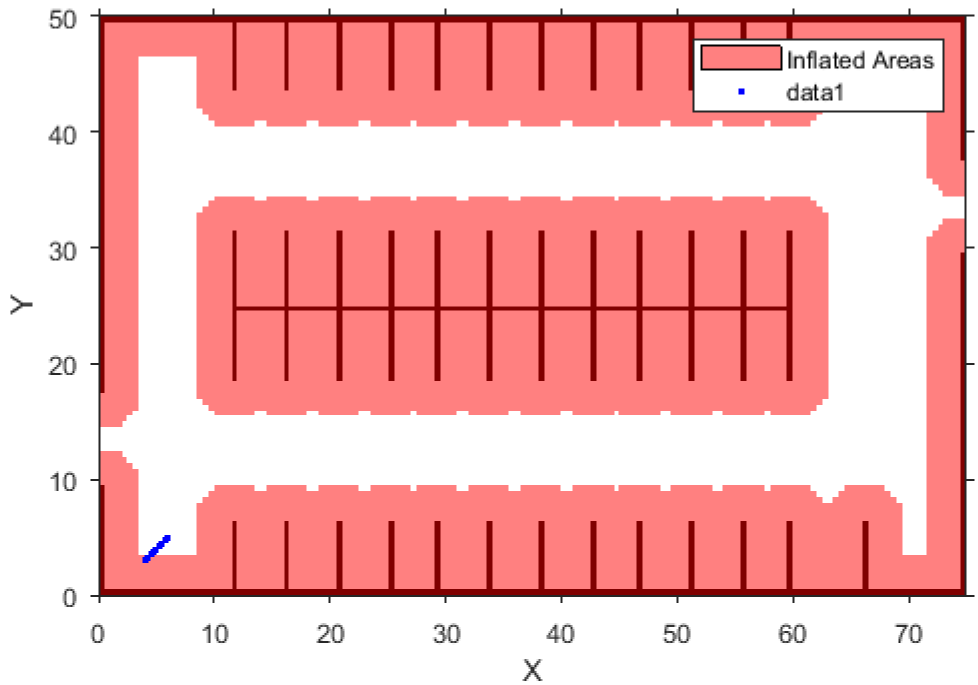
Check If Sequence of Poses Is Collision-Free

Load a costmap from a parking lot.

```
data = load('parkingLotCostmap.mat');
parkMap = data.parkingLotCostmap;
plot(parkMap)
```

Create vehicle poses following a straight-line path. `x` and `y` are the (*x*,*y*) coordinates of the rear axle of the vehicle. `theta` is the angle of the rear axle with respect to the *x*-axis. Note that the dimensions of the vehicle are stored in the `CollisionChecker.VehicleDimensions` property of the costmap, and that there is an offset between the rear axle of the vehicle and its center.

```
x = 4:0.25:6;
y = 3:0.25:5;
theta = repmat(45,size(x));
vehiclePoses = [x',y',theta'];
hold on
plot(x,y,'b.')
hold off
```



The first few (*x*,*y*) coordinates of the rear axle are within the inflated area. However, this does not imply a collision because the center of the vehicle may be outside the inflated area. Check if the poses are collision-free.

```
free = checkFree(parkMap,vehiclePoses)
```

```
free = 9x1 logical array

   1
   1
   1
   1
   1
   1
   1
   1
```

```
1
```

All values of `free` are 1 (`true`), so all poses are collision-free. The center of the vehicle does not enter the inflated area at any pose.

Input Arguments

`costmap` — Costmap
`vehicleCostmap` object

Costmap, specified as a `vehicleCostmap` object.

`vehiclePoses` — Vehicle poses
m-by-3 matrix of [*x*, *y*, *Θ*] vectors

Vehicle poses, specified as an *m*-by-3 matrix of [*x*, *y*, *Θ*] vectors. *m* is the number of poses.

x and *y* specify the location of the vehicle in world units, such as meters. This location is the center of the rear axle of the vehicle.

Θ specifies the orientation angle of the vehicle in degrees with respect to the *x*-axis. *Θ* is positive in the clockwise direction.

Example: `[3.4 2.6 0]` specifies a vehicle with the center of the rear axle at (3.4, 2.6) and an orientation angle of 0 degrees.

`xyPoints` — Points
M-by-2 real-valued matrix

Points, specified as an *M*-by-2 real-valued matrix that represents the (*x*, *y*) coordinates of *M* points.

Example: `[3.4 2.6]` specifies a single point at (3.4, 2.6)

Example: `[3 2;3 3;4 7]` specifies three points: (3, 2), (3, 3), and (4, 7)

Output Arguments

`free` — Vehicle pose or point is free
M-by-1 logical vector

Vehicle pose or point is free, returned as an *M*-by-1 logical vector. An element of `free` is `1` (`true`) when the corresponding vehicle pose in `vehiclePoses` or point in `xyPoints` is collision-free.

`freeMat` — Costmap cell is free
logical matrix

Costmap cell is free, returned as a logical matrix of the same size as the costmap grid. This size is specified by the `MapSize` property of the `costmap`. An element of `freeMat` is `1` (`true`) when the corresponding cell in `costmap` is unoccupied and the cost value of the cell is below the `FreeThreshold` of the costmap.

Tips

- If you specify a small value of `InflationRadius` that does not completely enclose the vehicle, then `checkFree` might report occupied poses as collision-free. To avoid this situation, the default value of `InflationRadius` completely encloses the vehicle.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`inflationCollisionChecker` | `pathPlannerRRT` | `vehicleCostmap`

Functions
`checkOccupied` | `checkPathValidity`

Introduced in R2018a

checkOccupied

Check vehicle costmap for occupied poses or points

Syntax

```
occ = checkOccupied(costmap,vehiclePoses)
occ = checkOccupied(costmap,xyPoints)
occMat = checkOccupied(costmap)
```

Description

The `checkOccupied` function checks whether vehicle poses or points are occupied by obstacles on the vehicle costmap. Path planning algorithms use `checkOccupied` to check whether candidate vehicle poses along a path are navigable.

To simplify the collision check for a vehicle pose, `vehicleCostmap` inflates obstacles according to the vehicle's `InflationRadius`, as specified by the `CollisionChecker` property of the costmap. The collision checker calculates the inflation radius by enclosing the vehicle in a set of overlapping circles of radius *R*, where the centers of these circles lie along the longitudinal axis of the vehicle. The inflation radius is the minimum *R* needed to fully enclose the vehicle in these circles. A vehicle pose is collision-free when none of the centers of these circles lie on an inflated grid cell. For more details, see the algorithm on page 4-913 on the `vehicleCostmap` reference page.

`occ = checkOccupied(costmap,vehiclePoses)` checks whether the vehicle poses are occupied.

`occ = checkOccupied(costmap,xyPoints)` checks whether (*x*, *y*) points in `xyPoints` are occupied.

`occMat = checkOccupied(costmap)` returns a logical matrix that indicates whether each cell of the costmap is occupied.

Examples

Check If Sequence of Poses Enters Occupied Cell

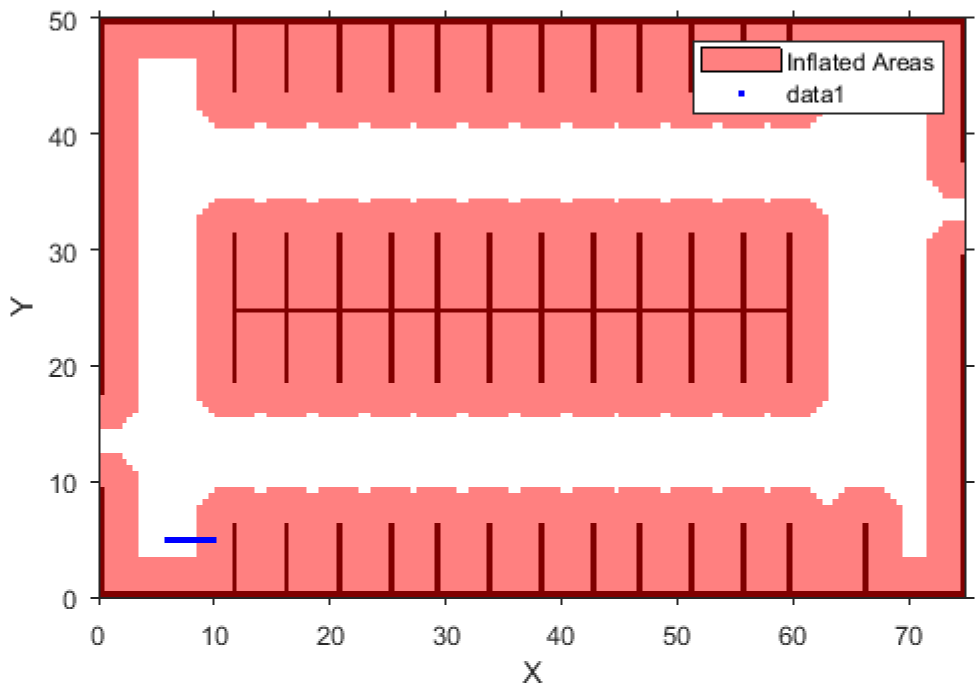
Load a costmap from a parking lot.

```
data = load('parkingLotCostmap.mat');
parkMap = data.parkingLotCostmap;
plot(parkMap)
```

Create vehicle poses following a straight-line path. `x` and `y` are the (*x*,*y*) coordinates of the rear axle of the vehicle. `theta` is the angle of the rear axle with respect to the *x*-axis. Note that the dimensions of the vehicle are stored in the `vehicleDimensions` property of the costmap, and that there is an offset between the rear axle of the vehicle and its center.

```
x = 6:0.25:10;
y = repmat(5,size(x));
theta = zeros(size(x));
```

```
vehiclePoses = [x',y',theta'];
hold on
plot(x,y,'b.')
```



Check if the poses are occupied.

```
occ = checkOccupied(parkMap,vehiclePoses)
```

```
occ = 17x1 logical array

   0
   0
   0
   0
   0
   1
   1
   1
   1
   1
      ⋮
```

The vehicle poses are occupied beginning with the sixth pose. In other words, the center of the vehicle in the sixth pose lies within the inflation radius of an occupied grid cell.

Input Arguments

`costmap` — Costmap
`vehicleCostmap` object

Costmap, specified as a `vehicleCostmap` object.

`vehiclePoses` — Vehicle poses
m-by-3 matrix of [*x*, *y*, *Θ*] vectors

Vehicle poses, specified as an *m*-by-3 matrix of [*x*, *y*, *Θ*] vectors. *m* is the number of poses.

x and *y* specify the location of the vehicle in world units, such as meters. This location is the center of the rear axle of the vehicle.

Θ specifies the orientation angle of the vehicle in degrees with respect to the *x*-axis. *Θ* is positive in the clockwise direction.

Example: `[3.4 2.6 0]` specifies a vehicle with the center of the rear axle at (3.4, 2.6) and an orientation angle of 0 degrees.

`xyPoints` — Points
M-by-2 real-valued matrix

Points, specified as an *M*-by-2 real-valued matrix that represents the (*x*, *y*) coordinates of *M* points.

Example: `[3.4 2.6]` specifies a single point at (3.4, 2.6)

Example: `[3 2;3 3;4 7]` specifies three points: (3, 2), (3, 3), and (4, 7)

Output Arguments

`occ` — Vehicle pose or point is occupied
M-by-1 logical vector

Vehicle pose or point is occupied, returned as an *M*-by-1 logical vector. An element of `occ` is `1` (`true`) when the corresponding vehicle pose in `vehiclePoses` or planar point in `xyPoints` is occupied.

`occMat` — Costmap cell is occupied
logical matrix

Costmap cell is occupied, returned as a logical matrix of the same size as the costmap grid. This size is specified by the `MapSize` property of the `costmap`. An element of `occMat` is `1` (`true`) when the corresponding cell in `costmap` is occupied.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`inflationCollisionChecker` | `pathPlannerRRT` | `vehicleCostmap`

Functions
`checkFree` | `checkPathValidity`

Introduced in R2018a

getCosts

Get cost value of cells in vehicle costmap

Syntax

```
costVals = getCosts(costmap,xyPoints)
costMat = getCosts(costmap)
```

Description

`costVals = getCosts(costmap,xyPoints)` returns a vector, `costVals`, that contains the costs for the (*x*, *y*) points in `xyPoints` in the vehicle costmap.

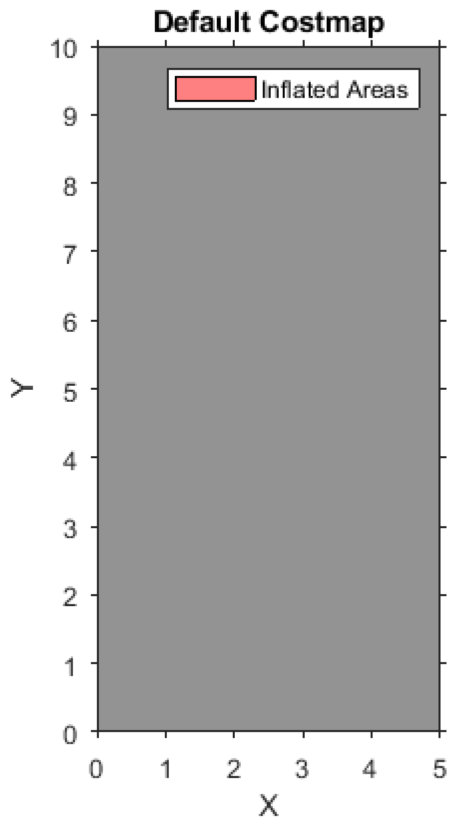
`costMat = getCosts(costmap)` returns a matrix, `costMat`, that contains the cost of each cell in the costmap.

Examples

Get Cost Matrix and Set Cost Values

Create a 5-by-10 meter vehicle costmap. Cells have side length 1, in the world units of meters. Set the inflation radius to 1. Plot the costmap, and get the default cost matrix.

```
costmap = vehicleCostmap(5,10);
costmap.CollisionChecker.InflationRadius = 1;
plot(costmap)
title('Default Costmap')
```

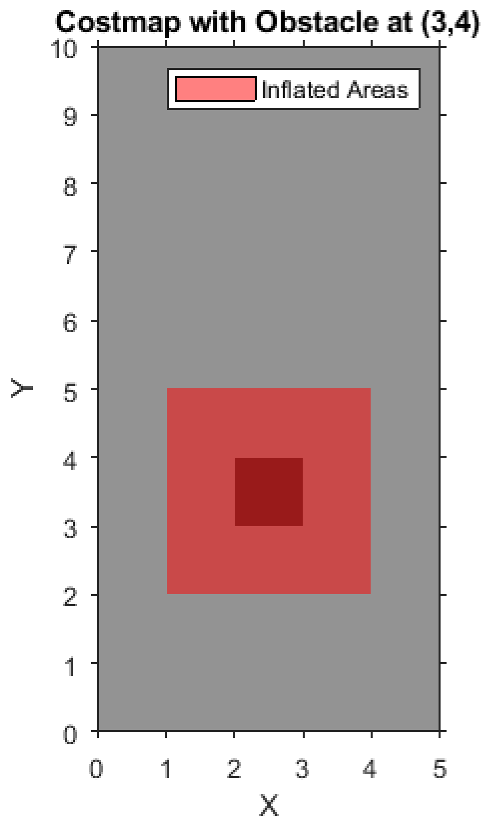



```
getCosts(costmap)
```

```
ans = 10×5

    0.4250    0.4250    0.4250    0.4250    0.4250
    0.4250    0.4250    0.4250    0.4250    0.4250
    0.4250    0.4250    0.4250    0.4250    0.4250
    0.4250    0.4250    0.4250    0.4250    0.4250
    0.4250    0.4250    0.4250    0.4250    0.4250
    0.4250    0.4250    0.4250    0.4250    0.4250
    0.4250    0.4250    0.4250    0.4250    0.4250
    0.4250    0.4250    0.4250    0.4250    0.4250
    0.4250    0.4250    0.4250    0.4250    0.4250
    0.4250    0.4250    0.4250    0.4250    0.4250
```

Mark an obstacle at the (x,y) coordinate (3,4) by increasing the cost of that cell.

```
setCosts(costmap,[3,4],0.8);
plot(costmap)
title('Costmap with Obstacle at (3,4)')
```



Get the cost of three cells: the cell with the obstacle, a cell adjacent to the obstacle, and a cell outside the inflation radius of the obstacle.

```
costVal = getCosts(costmap,[3 4;2 4;4 7])
```

```
costVal = 3×1

    0.8000
    0.4250
    0.4250
```

Although the plot of the costmap displays the cell with the obstacle and its adjacent cells in shades of red, only the cell with the obstacle has a higher cost value of 0.8. The other cells still have the default cost value of 0.425.

Input Arguments

`costmap` — Costmap
`vehicleCostmap` object

Costmap, specified as a `vehicleCostmap` object.

`xyPoints` — Points
M-by-2 real-valued matrix

Points, specified as an *M*-by-2 real-valued matrix that represents the (*x*, *y*) coordinates of *M* points.

Example: `[3.4 2.6]` specifies a single point at (3.4, 2.6)

Example: `[3 2;3 3;4 7]` specifies three points: (3, 2), (3, 3), and (4, 7)

Output Arguments

`costVals` — Cost of points
M-element real-valued vector

Cost of points in `xyPoints`, returned as an *M*-element real-valued vector.

`costMat` — Cost of all cells
real-valued matrix

Cost of all cells in `costmap`, returned as a real-valued matrix of the same size as the costmap grid. This size is specified by the `MapSize` property of the costmap.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`setCosts` | `vehicleCostmap`

Introduced in R2018a

plot

Plot vehicle costmap

Syntax

```
plot(costmap)
plot(costmap,Name,Value)
```

Description

The `plot` function displays a vehicle costmap. The darkness of each cell is proportional to the cost value of the cell. Cells with low cost are bright, and cells containing obstacles with high cost are dark. Inflated areas are displayed with a red hue, and cells outside the inflated area are displayed in grayscale.

`plot(costmap)` plots the vehicle costmap in the current axes.

`plot(costmap,Name,Value)` plots the vehicle costmap using name-value pair arguments to specify the parent axes or to adjust the display of inflated areas.

Examples

Display a Vehicle on a Costmap

Load a costmap from a parking lot. Display the costmap.

```
data = load('parkingLotCostmap.mat');
parkMap = data.parkingLotCostmap;
plot(parkMap)
```

Create a template polyshape object with the dimensions of the car.

```
carDims = parkMap.CollisionChecker.VehicleDimensions
```

```
carDims = 
  vehicleDimensions with properties:

           Length: 4.7000
            Width: 1.8000
           Height: 1.4000
        Wheelbase: 2.8000
     RearOverhang: 1
    FrontOverhang: 0.9000
       WorldUnits: 'meters'
```

```
ro = carDims.RearOverhang;
fo = carDims.FrontOverhang;
wb = carDims.Wheelbase;
hw = carDims.Width/2;
X = [-ro,wb+fo,wb+fo,-ro];
```

```
Y = [-hw,-hw,hw,hw];
templateShape = polyshape(X',Y');
```

Create a function handle to move the template to a specified vehicle pose. This `move` function translates the polyshape `s` to the coordinate (x,y) and then rotates the polyshape by an angle `theta` about the point (x,y).

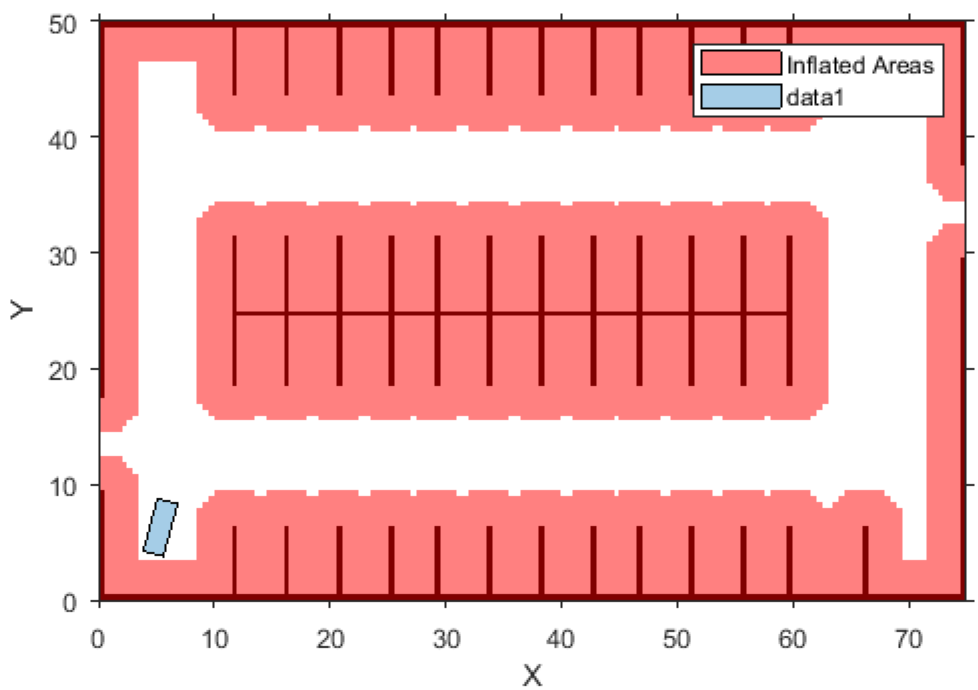
```
move = @(s,x,y,theta) rotate(translate(s,[x,y]), ...
    theta,[x,y]);
```

Move the car template to a pose.

```
carPose = [5,5,75];
carShape = move(templateShape,carPose(1),carPose(2),carPose(3));
```

Plot the car on the costmap.

```
hold on
plot(carShape)
```



Input Arguments

`costmap` — Costmap
`vehicleCostmap` object

Costmap, specified as a `vehicleCostmap` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Inflation','off'`

`Inflation` — Display inflated areas
`'on'` (default) | `'off'`

Display inflated areas, specified as the comma-separated pair consisting of `'Inflation'` and one of the following.

- `'on'`—Cells in the inflated area have a red hue.
- `'off'`—Cells containing obstacles have a red hue, but other cells in the inflated area are displayed in grayscale.

`Parent` — Axes on which to plot costmap
axes handle

Axes on which to plot the costmap, specified as the comma-separated pair consisting of `'Parent'` and an axes handle. By default, `plot` uses the current axes handle, which is returned by the `gca` function.

See Also
`polyshape` | `vehicleCostmap` | `vehicleDimensions`

Introduced in R2018a

setCosts

Set cost value of cells in vehicle costmap

Syntax

```
setCosts(costmap,xyPoints,costVals)
```

Description

`setCosts(costmap,xyPoints,costVals)` sets the costs, `costVals`, for the (*x*, *y*) points in `xyPoints` in the vehicle costmap.

Examples

Mark Rectangular Obstacle on Vehicle Costmap

Create a 10-by-15 meter vehicle costmap. Cells have a side length of 1 meter.

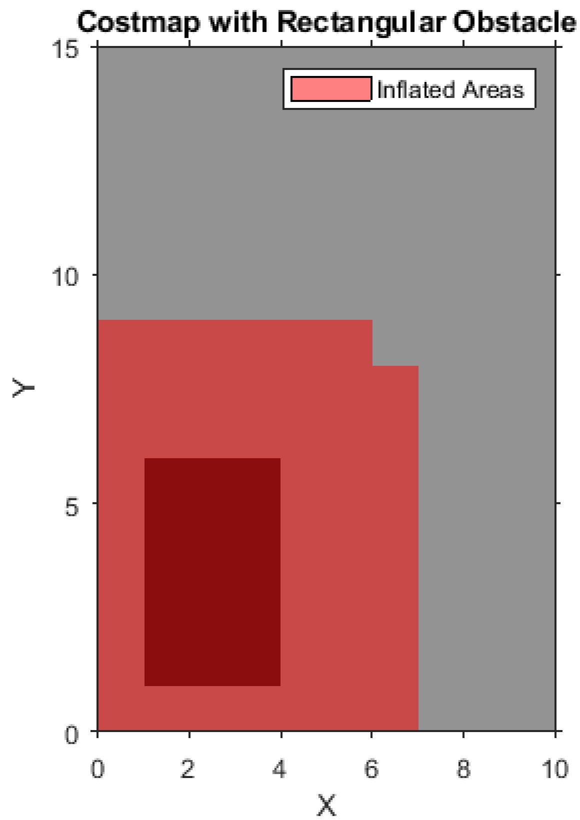
```
costmap = vehicleCostmap(10,15);
```

Define a set of (*x*,*y*) coordinates that correspond to a 3-by-5 meter rectangle.

```
[x,y] = meshgrid(2:4,2:6);
xyPoints = [x(:),y(:)];
```

Mark the rectangle as an obstacle by increasing the cost of its cells to 0.9.

```
costVal = 0.9;
setCosts(costmap,xyPoints,costVal);
plot(costmap)
title('Costmap with Rectangular Obstacle')
```



Input Arguments

`costmap` — Costmap
`vehicleCostmap` object

Costmap, specified as a `vehicleCostmap` object.

`xyPoints` — Points
M-by-2 real-valued matrix

Points, specified as an *M*-by-2 real-valued matrix that represents the (*x*, *y*) coordinates of *M* points.

Example: `[3.4 2.6]` specifies a single point at (3.4, 2.6)

Example: `[3 2;3 3;4 7]` specifies three points: (3, 2), (3, 3), and (4, 7)

`costVals` — Cost of points
M-element real-valued vector

Cost of points in `xyPoints`, specified as an *M*-element real-valued vector.

Example: `0.8` specifies the cost of a single point

Example: `[0.2 0.5 0.8]` specifies the cost of three points

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`getCosts` | `vehicleCostmap`

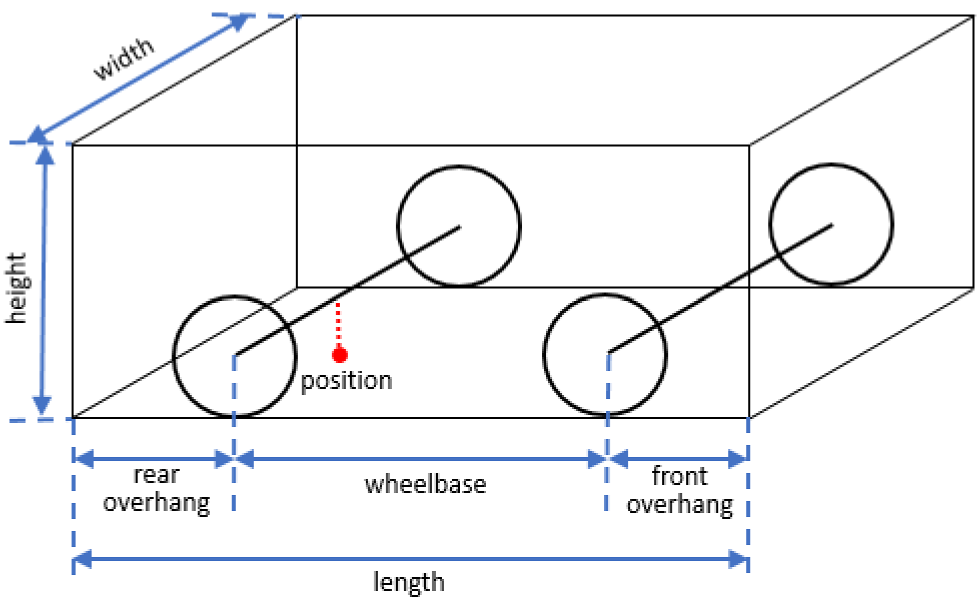
Introduced in R2018a

vehicleDimensions

Store vehicle dimensions

Description

The `vehicleDimensions` object stores vehicle dimensions. The figure shows the dimensions that are included in the `vehicleDimensions`.



The position of the vehicle is often represented as a single point located on the ground at the center of the rear axle, as indicated by the red dot in the figure. This position corresponds to the natural center of rotation of the vehicle.

The table lists typical vehicle types and their corresponding dimensions.

Vehicle Classification	Length	Width	Height	Wheelbase	Front Overhang	Rear Overhang
Automobile (sedan)	4.7 m	1.8 m	1.4 m	2.8 m	0.9 m	1.0 m
Motorcycle	2.2 m	0.6 m	1.5 m	1.51 m	0.37 m	0.32 m

Creation

Syntax

```
vdims = vehicleDimensions
```

```
vdims = vehicleDimensions(l,w,h)
vdims = vehicleDimensions(___,Name,Value)
```

Description

`vdims = vehicleDimensions` creates a `vehicleDimensions` object with a default length of 4.7 m, width of 1.8 m, and height of 1.4 m.

`vdims = vehicleDimensions(l,w,h)` creates a `vehicleDimensions` object and sets the `Length`, `Width`, and `Height` properties.

`vdims = vehicleDimensions(___,Name,Value)` uses one or more name-value pair arguments to set the `Wheelbase`, `FrontOverhang`, `RearOverhang`, and `WorldUnits` properties. `Name` is the property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Properties

`Length` — Length of vehicle
`4.7` (default) | positive real scalar

Length of vehicle, specified as a positive real scalar.

Data Types: `double`

`Width` — Width of vehicle
`1.8` (default) | positive real scalar

Width of vehicle, specified as a positive real scalar.

Data Types: `double`

`Height` — Height of vehicle
`1.4` (default) | positive real scalar

Height of vehicle, specified as a positive real scalar.

Data Types: `double`

`FrontOverhang` — Front overhang of vehicle
`0.9` (default) | real scalar

Front overhang of vehicle, specified as a real scalar. The front overhang is the distance between the front of the vehicle and the front axle. `FrontOverhang` can be negative.

Data Types: `double`

`RearOverhang` — Rear overhang of vehicle
`1.0` (default) | real scalar

Rear overhang of vehicle, specified as a real scalar. The rear overhang is the distance between the rear of the vehicle and the rear axle. `RearOverhang` can be negative.

Data Types: `double`

`Wheelbase` — Distance between axles
`2.8` (default) | positive real scalar

The distance between the front and rear axles of the vehicle, specified as a positive real scalar.

Data Types: `double`

`WorldUnits` — Units of measurement
`'meters'` (default) | character array

Units of measurement, specified as a character array. The units do not affect the values of measurements.

Examples

Specify Dimensions of a Motorcycle

Store the dimensions of a motorcycle with length 2.2, width 0.6, and height 1.5 meters. Also specify the distance that the motorcycle extends ahead of the front axle and behind the rear axle.

```
vdims = vehicleDimensions(2.2,0.6,1.5, ...
    'FrontOverhang',0.37,'RearOverhang',0.32)
```

```
vdims = 
  vehicleDimensions with properties:

           Length: 2.2000
            Width: 0.6000
           Height: 1.5000
        Wheelbase: 1.5100
     RearOverhang: 0.3200
    FrontOverhang: 0.3700
       WorldUnits: 'meters'
```

Tips

- The `Length` of the vehicle is the sum of the `Wheelbase`, `FrontOverhang`, and `RearOverhang`. If you change `FrontOverhang`, then the value of `Wheelbase` automatically adjusts to keep `Length` constant. Any change resulting in a negative wheelbase causes an error.
- You can use the vehicle dimensions to define a `vehicleCostmap` that represents the planning search space around a vehicle. Path planning algorithms, such as `pathPlannerRRT`, use vehicle dimensions to find a path for the vehicle to follow.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All inputs to `vehicleDimensions` must be compile-time constants.

See Also
`vehicle` | `vehicleCostmap`

Introduced in R2018a

objectDetection

Report for single object detection

Description

An objectDetection object contains an object detection report that was obtained by a sensor for a single object. You can use the objectDetection output as the input to trackers such as multiObjectTracker.

Creation

Syntax

```
detection = objectDetection(time,measurement)
detection = objectDetection( ___ ,Name,Value)
```

Description

detection = objectDetection(time,measurement) creates an object detection at the specified time from the specified measurement.

detection = objectDetection(___ ,Name,Value) creates a detection object with properties specified as one or more Name,Value pair arguments. Any unspecified properties have default values. You cannot specify the Time or Measurement properties using Name,Value pairs.

Input Arguments

time — Detection time
nonnegative real scalar

Detection time, specified as a nonnegative real scalar. This argument sets the Time property.

measurement — Object measurement
real-valued *N*-element vector

Object measurement, specified as a real-valued *N*-element vector. *N* is determined by the coordinate system used to report detections and other parameters that you specify in the MeasurementParameters property for the objectDetection object.

This argument sets the Measurement property.

Output Arguments

detection — Detection report
objectDetection object

Detection report for a single object, returned as an objectDetection object. An objectDetection object contains these properties:

Property	Definition
Time	Measurement time
Measurement	Object measurements
MeasurementNoise	Measurement noise covariance matrix
SensorIndex	Unique ID of the sensor
ObjectClassID	Object classification
ObjectAttributes	Additional information passed to tracker
MeasurementParameters	Parameters used by initialization functions of nonlinear Kalman tracking filters

Properties

Time — Detection time
nonnegative real scalar

Detection time, specified as a nonnegative real scalar. You cannot set this property as a name-value pair. Use the `time` input argument instead.

Example: `5.0`

Data Types: `double`

Measurement — Object measurement
real-valued *N*-element vector

Object measurement, specified as a real-valued *N*-element vector. You cannot set this property as a name-value pair. Use the `measurement` input argument instead.

Example: `[1.0;-3.4]`

Data Types: `double` | `single`

MeasurementNoise — Measurement noise covariance
scalar | real positive semi-definite symmetric *N*-by-*N* matrix

Measurement noise covariance, specified as a scalar or a real positive semi-definite symmetric *N*-by-*N* matrix. *N* is the number of elements in the measurement vector. For the scalar case, the matrix is a square diagonal *N*-by-*N* matrix having the same data interpretation as the measurement.

Example: `[5.0,1.0;1.0,10.0]`

Data Types: `double` | `single`

SensorIndex — Sensor identifier
`1` | positive integer

Sensor identifier, specified as a positive integer. The sensor identifier lets you distinguish between different sensors and must be unique to the sensor.

Example: `5`

Data Types: `double`

ObjectClassID — Object class identifier
`0` (default) | positive integer

Object class identifier, specified as a positive integer. Object class identifiers distinguish between different kinds of objects. The value 0 denotes an unknown object type. If the class identifier is nonzero, multiObjectTracker immediately creates a confirmed track from the detection.

Example: 1

Data Types: double

MeasurementParameters — Measurement function parameters
{} (default) | structure array | cell containing structure array | cell array

Measurement function parameters, specified as a structure array, a cell containing a structure array, or a cell array. The property contains all the arguments used by the measurement function specified by the MeasurementFcn property of a nonlinear tracking filter such as trackingEKF or trackingUKF.

The table shows sample fields for the MeasurementParameters structures.

Field	Description	Example
Frame	Frame used to report measurements, specified as one of these values: • 'rectangular' — Detections are reported in rectangular coordinates. • 'spherical' — Detections are reported in spherical coordinates.	'spherical'
OriginPosition	Position offset of the origin of the frame relative to the parent frame, specified as an [x y z] real-valued vector.	[0 0 0]
OriginVelocity	Velocity offset of the origin of the frame relative to the parent frame, specified as a [vx vy vz] real-valued vector.	[0 0 0]
Orientation	Frame rotation matrix, specified as a 3-by-3 real-valued orthonormal matrix.	[1 0 0; 0 1 0; 0 0 1]
HasAzimuth	Logical scalar indicating if azimuth is included in the measurement.	1
HasElevation	Logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if HasElevation is false, the reported measurements assume 0 degrees of elevation.	1

Field	Description	Example
HasRange	Logical scalar indicating if range is included in the measurement.	1
HasVelocity	Logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if HasVelocity is false, the measurements are reported as [x y z]. If HasVelocity is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if Orientation performs a frame rotation from the parent coordinate frame to the child coordinate frame. When IsParentToChild is false, then Orientation performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

ObjectAttributes — Object attributes
{} (default) | cell array

Object attributes passed through the tracker, specified as a cell array. These attributes are added to the output of the multiObjectTracker but not used by the tracker.

Example: {[10,20,50,100],'radar1'}

Examples

Create Detection from Position Measurement

Create a detection from a position measurement. The detection is made at a timestamp of one second from a position measurement of [100;250;10] in Cartesian coordinates.

```
detection = objectDetection(1,[100;250;10])
```

```
detection = 
  objectDetection with properties:

                     Time: 1
              Measurement: [3x1 double]
         MeasurementNoise: [3x3 double]
              SensorIndex: 1
            ObjectClassID: 0
    MeasurementParameters: {}
```

```
        ObjectAttributes: {}
```

Create Detection With Measurement Noise

Create an `objectDetection` from a time and position measurement. The detection is made at a time of one second for an object position measurement of `[100;250;10]`. Add measurement noise and set other properties using Name-Value pairs.

```
detection = objectDetection(1,[100;250;10],'MeasurementNoise',10, ...
    'SensorIndex',1,'ObjectAttributes',{'Example object',5})
```

```
detection = 
  objectDetection with properties:

                     Time: 1
              Measurement: [3x1 double]
         MeasurementNoise: [3x3 double]
              SensorIndex: 1
            ObjectClassID: 0
    MeasurementParameters: {}
         ObjectAttributes: {'Example object'  [5]}
```

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`multiObjectTracker` | `radarDetectionGenerator` | `trackingEKF` | `trackingKF` | `trackingUKF` | `visionDetectionGenerator`

Introduced in R2017a

trackingKF

Linear Kalman filter for object tracking

Description

A `trackingKF` object is a discrete-time linear Kalman filter used to track the positions and velocities of objects that can be encountered in an automated driving scenario. Such objects include automobiles, pedestrians, bicycles, and stationary structures or obstacles.

A Kalman filter is a recursive algorithm for estimating the evolving state of a process when measurements are made on the process. The filter is linear when the evolution of the state follows a linear motion model and the measurements are linear functions of the state. The filter assumes that both the process and measurements have additive noise. When the process noise and measurement noise are Gaussian, the Kalman filter is the optimal minimum mean squared error (MMSE) state estimator for linear processes.

You can use this object in these ways:

- Explicitly set the motion model. Set the motion model property, `MotionModel`, to `Custom`, and then use the `StateTransitionModel` property to set the state transition matrix.
- Set the `MotionModel` property to a predefined state transition model:

Motion Model
`'1D Constant Velocity'`
`'1D Constant Acceleration'`
`'2D Constant Velocity'`
`'2D Constant Acceleration'`
`'3D Constant Velocity'`
`'3D Constant Acceleration'`

Creation

Syntax

```
filter = trackingKF
filter = trackingKF(F,H)
filter = trackingKF(F,H,G)
filter = trackingKF('MotionModel',model)
filter = trackingKF(___,Name,Value)
```

Description

`filter = trackingKF` creates a linear Kalman filter object for a discrete-time, 2-D, constant-velocity moving object. The Kalman filter uses default values for the `StateTransitionModel`, MeasurementModel, and ControlModel properties. The function also sets the MotionModel property to '2D Constant Velocity'.

filter = trackingKF(F,H) specifies the state transition model, F, and the measurement model, H. With this syntax, the function also sets the MotionModel property to 'Custom'.

filter = trackingKF(F,H,G) also specifies the control model, G. With this syntax, the function also sets the MotionModel property to 'Custom'.

filter = trackingKF('MotionModel',model) sets the motion model property, MotionModel, to model.

filter = trackingKF(___ ,Name,Value) configures the properties of the Kalman filter by using one or more Name,Value pair arguments and any of the previous syntaxes. Any unspecified properties take default values.

Properties

State — Kalman filter state

0 (default) | real-valued scalar | real-valued *M*-element vector

Kalman filter state, specified as a real-valued *M*-element vector. *M* is the size of the state vector. Typical state vector sizes are described in the MotionModel property. When the initial state is specified as a scalar, the state is expanded into an *M*-element vector.

You can set the state to a scalar in these cases:

- When the MotionModel property is set to 'Custom', *M* is determined by the size of the state transition model.
- When the MotionModel property is set to '2D Constant Velocity', '3D Constant Velocity', '2D Constant Acceleration', or '3D Constant Acceleration', you must first specify the state as an *M*-element vector. You can use a scalar for all subsequent specifications of the state vector.

Example: [200;0.2;-40;-0.01]

Data Types: double

StateCovariance — State estimation error covariance

1 (default) | positive scalar | positive-definite real-valued *M*-by-*M* matrix

State error covariance, specified as a positive scalar or a positive-definite real-valued *M*-by-*M* matrix, where *M* is the size of the state. Specifying the value as a scalar creates a multiple of the *M*-by-*M* identity matrix. This matrix represents the uncertainty in the state.

Example: [20 0.1; 0.1 1]

Data Types: double

MotionModel — Kalman filter motion model

'Custom' (default) | '1D Constant Velocity' | '2D Constant Velocity' | '3D Constant Velocity' | '1D Constant Acceleration' | '2D Constant Acceleration' | '3D Constant Acceleration'

Kalman filter motion model, specified as 'Custom' or one of these predefined models. In this case, the state vector and state transition matrix take the form specified in the table.

Motion Model	Form of State Vector	Form of State Transition Model
'1D Constant Velocity'	[x;vx]	[1 dt; 0 1]
'2D Constant Velocity'	[x;vx;y;vy]	Block diagonal matrix with the [1 dt; 0 1] block repeated for the *x* and *y* spatial dimensions
'3D Constant Velocity'	[x;vx;y;vy;z;vz]	Block diagonal matrix with the [1 dt; 0 1] block repeated for the *x*, *y*, and *z* spatial dimensions.
'1D Constant Acceleration'	[x;vx;ax]	[1 dt 0.5*dt^2; 0 1 dt; 0 0 1]
'2D Constant Acceleration'	[x;vx;ax;y;vy;ay]	Block diagonal matrix with [1 dt 0.5*dt^2; 0 1 dt; 0 0 1] blocks repeated for the *x* and *y* spatial dimensions
'3D Constant Acceleration'	[x;vx,ax;y;vy;ay;z;vz;az]	Block diagonal matrix with the [1 dt 0.5*dt^2; 0 1 dt; 0 0 1] block repeated for the *x*, *y*, and *z* spatial dimensions

When the ControlModel property is defined, every nonzero element of the state transition model is replaced by dt.

When MotionModel is 'Custom', you must specify a state transition model matrix, a measurement model matrix, and optionally, a control model matrix as input arguments to the Kalman filter.

Data Types: char

StateTransitionModel — State transition model between time steps
[1 1 0 0; 0 1 0 0; 0 0 1 1; 0 0 0 1] (default) | real-valued *M*-by-*M* matrix

State transition model between time steps, specified as a real-valued *M*-by-*M* matrix. *M* is the size of the state vector. In the absence of controls and noise, the state transition model relates the state at any time step to the state at the previous step. The state transition model is a function of the filter time step size.

Example: [1 0; 1 2]

Dependencies

To enable this property, set MotionModel to 'Custom'.

Data Types: double

ControlModel — Control model
[] (default) | *M*-by-*L* real-valued matrix

Control model, specified as an *M*-by-*L* matrix. *M* is the dimension of the state vector and *L* is the number of controls or forces. The control model adds the effect of controls on the evolution of the state.

Example: [.01 0.2]

Data Types: `double`

`ProcessNoise` — Covariance of process noise
1 (default) | positive scalar | real-valued positive-definite *M*-by-*M* matrix

Covariance of process noise, specified as a positive scalar or an *M*-by-*M* matrix where *M* is the dimension of the state. If you specify this property as a scalar, the filter uses the value as a multiplier of the *M*-by-*M* identity matrix. Process noise expresses the uncertainty in the dynamic model and is assumed to be zero-mean Gaussian white noise.

Tip If you specify the `MotionModel` property as any of the predefined motion model, then the corresponding process noise is automatically generated during construction and updated during propagation. In this case, you do not need to specify the `ProcessNoise` property. In fact, the filter neglects your process noise input during object construction. If you want to specify the process noise other than the default values, use the `trackingEKF` object.

Data Types: `double`

`MeasurementModel` — Measurement model from state vector
`[1 0 0 0; 0 0 1 0]` (default) | real-valued *N*-by-*M* matrix

Measurement model from the state vector, specified as a real-valued *N*-by-*M* matrix, where *N* is the size of the measurement vector and *M* is the size of the state vector. The measurement model is a linear matrix that determines predicted measurements from the predicted state.

Example: `[1 0.5 0.01; 1.0 1 0]`

Data Types: `double`

`MeasurementNoise` — Measurement noise covariance
1 (default) | positive scalar | positive-definite real-valued *N*-by-*N* matrix

Covariance of the measurement noise, specified as a positive scalar or a positive-definite, real-valued *N*-by-*N* matrix, where *N* is the size of the measurement vector. If you specify this property as a scalar, the filter uses the value as a multiplier of the *N*-by-*N* identity matrix. Measurement noise represents the uncertainty of the measurement and is assumed to be zero-mean Gaussian white noise.

Example: `0.2`

Data Types: `double`

Object Functions

predict	Predict state and state estimation error covariance of linear Kalman filter
correct	Correct state and state estimation error covariance using tracking filter
correctjpda	Correct state and state estimation error covariance using tracking filter and JPDA
distance	Distances between current and predicted measurements of tracking filter
likelihood	Likelihood of measurement from tracking filter
clone	Create duplicate tracking filter
residual	Measurement residual and residual noise from tracking filter
initialize	Initialize state and covariance of tracking filter

Examples

Constant-Velocity Linear Kalman Filter

Create a linear Kalman filter that uses a `2D Constant Velocity` motion model. Assume that the measurement consists of the object's *x*-*y* location.

Specify the initial state estimate to have zero velocity.

```
x = 5.3;
y = 3.6;
initialState = [x;0;y;0];
KF = trackingKF('MotionModel','2D Constant Velocity','State',initialState);
```

Create the measured positions from a constant-velocity trajectory.

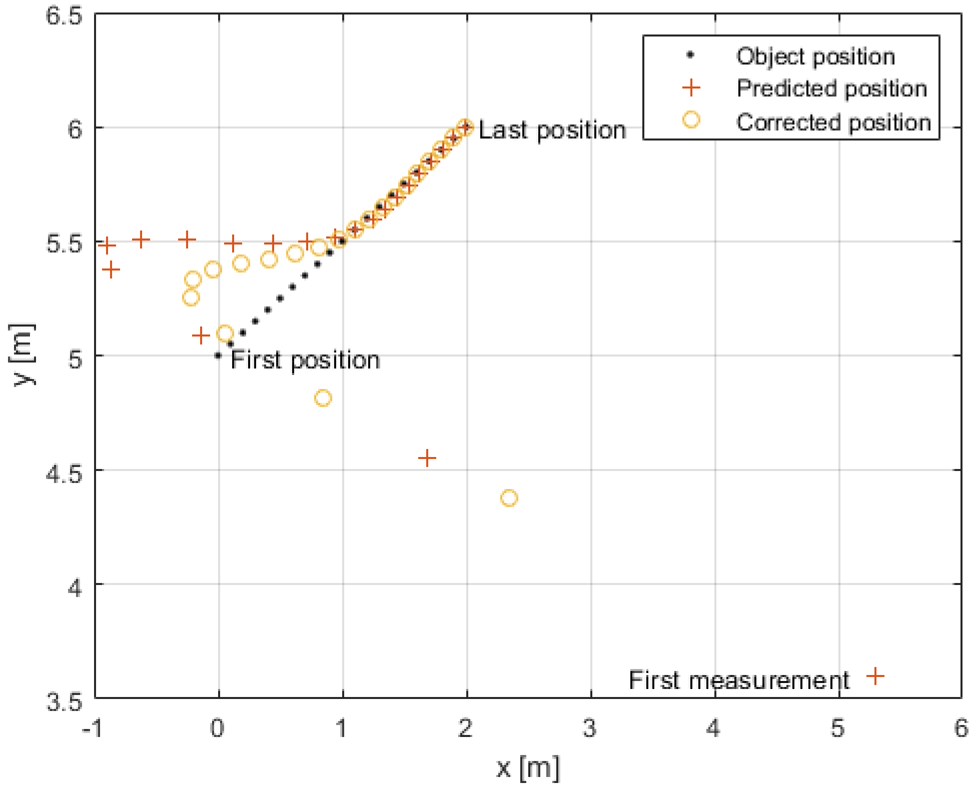
```
vx = 0.2;
vy = 0.1;
T  = 0.5;
pos = [0:vx*T:2;5:vy*T:6]';
```

Predict and correct the state of the object.

```
for k = 1:size(pos,1)
    pstates(k,:) = predict(KF,T);
    cstates(k,:) = correct(KF,pos(k,:));
end
```

Plot the tracks.

```
plot(pos(:,1),pos(:,2),'k.', pstates(:,1),pstates(:,3),'+', ...
    cstates(:,1),cstates(:,3),'o')
xlabel('x [m]')
ylabel('y [m]')
grid
xt  = [x-2 pos(1,1)+0.1 pos(end,1)+0.1];
yt = [y pos(1,2) pos(end,2)];
text(xt,yt,{'First measurement','First position','Last position'})
legend('Object position', 'Predicted position', 'Corrected position')
```



More About

Filter Parameters

This table relates the filter model parameters to the object properties. *M* is the size of the state vector. *N* is the size of the measurement vector. *L* is the size of the control model.

Model Parameter	Description	Filter Property	Size
F_k	State transition model that specifies a linear model of the force-free equations of motion of the object. This model, together with the control model, determines the state at time *k+1* as a function of the state at time *k*. The state transition model depends on the time step of the filter.	`StateTransitionModel`	*M*-by-*M*

Model Parameter	Description	Filter Property	Size
H_k	Measurement model that specifies how the measurements are linear functions of the state.	`MeasurementModel`	*N*-by-*M*
G_k	Control model describing the controls or forces acting on the object.	`ControlModel`	*M*-by-*L*
x_k	Estimate of the state of the object.	`State`	*M*-
P_k	Estimated covariance matrix of the state. The covariance represents the uncertainty in the values of the state.	`StateCovariance`	*M*-by-*M*
Q_k	Estimate of the process noise covariance matrix at step *k*. Process noise is a measure of the uncertainty in your dynamic model and is assumed to be zero-mean white Gaussian noise.	`ProcessNoise`	*M*-by-*M*
R_k	Estimate of the measurement noise covariance at step *k*. Measurement noise represents the uncertainty of the measurement and is assumed to be zero-mean white Gaussian noise.	`MeasurementNoise`	*N*-by-*N*

Algorithms

The Kalman filter describes the motion of an object by estimating its state. The state generally consists of object position and velocity and possibly its acceleration. The state can span one, two, or three spatial dimensions. Most frequently, you use the Kalman filter to model constant-velocity or constant-acceleration motion. A linear Kalman filter assumes that the process obeys the following linear stochastic difference equation:

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

x_k is the state at step k. F_k is the state transition model matrix. G_k is the control model matrix. u_k represents known generalized controls acting on the object. In addition to the specified equations of motion, the motion may be affected by random noise perturbations, v_k. The state, the state transition matrix, and the controls together provide enough information to determine the future motion of the object in the absence of noise.

In the Kalman filter, the measurements are also linear functions of the state,

$$z_k = H_k x_k + w_k$$

where H_k is the measurement model matrix. This model expresses the measurements as functions of the state. A measurement can consist of an object position, position and velocity, or its position, velocity, and acceleration, or some function of these quantities. The measurements can also include noise perturbations, w_k.

These equations, in the absence of noise, model the actual motion of the object and the actual measurements. The noise contributions at each step are unknown and cannot be modeled. Only the noise covariance matrices are known. The state covariance matrix is updated with knowledge of the noise covariance only.

For a brief description of the linear Kalman filter algorithm, see “Linear Kalman Filters” .

References

[1] Brown, R.G. and P.Y.C. Wang. *Introduction to Random Signal Analysis and Applied Kalman Filtering*. 3rd Edition. New York: John Wiley & Sons, 1997.

[2] Kalman, R. E. "A New Approach to Linear Filtering and Prediction Problems." *Transaction of the ASME–Journal of Basic Engineering*, Vol. 82, Series D, March 1960, pp. 35–45.

[3] Blackman, Samuel. *Multiple-Target Tracking with Radar Applications*. Artech House. 1986.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- When you create a `trackingKF` object, and you specify the `MotionModel` property as any value other than `'Custom'`, then you must specify the state vector explicitly at construction time using the `State` property. The choice of motion model determines the size of the state vector. However, motion models do not specify the data type, for example, double precision or single precision. Both size and data type are required for code generation.

See Also

Functions
`initcakf` | `initcvkf`

Objects
`multiObjectTracker` | `trackingABF` | `trackingEKF` | `trackingUKF`

Topics
“Linear Kalman Filters”

Introduced in R2017a

trackingABF

Alpha-beta filter for object tracking

Description

The `trackingABF` object represents an alpha-beta filter designed for object tracking for an object that follows a linear motion model and has a linear measurement model. Linear motion is defined by constant velocity or constant acceleration. Use the filter to predict the future location of an object, to reduce noise for a detected location, or to help associate multiple objects with their tracks.

Creation

Syntax

```
abf = trackingABF
abf = trackingABF(Name,Value)
```

Description

`abf = trackingABF` returns an alpha-beta filter for a discrete time, 2-D constant velocity system. The motion model is named `'2D Constant Velocity'` with the state defined as `[x; vx; y; vy]`.

`abf = trackingABF(Name,Value)` specifies the properties of the filter using one or more `Name,Value` pair arguments. Any unspecified properties take default values.

Properties

`MotionModel` — Model of target motion
`'2D Constant Velocity'` (default) | `'1D Constant Velocity'` | `'3D Constant Velocity'` | `'1D Constant Acceleration'` | `'2D Constant Acceleration'` | `'3D Constant Acceleration'`

Model of target motion, specified as a character vector or string. Specifying `1D`, `2D`, or `3D` specifies the dimension of the target's motion. Specifying `Constant Velocity` assumes that the target motion is a constant velocity at each simulation step. Specifying `Constant Acceleration` assumes that the target motion is a constant acceleration at each simulation step.

Data Types: `char` | `string`

`State` — Filter state
real-valued *M*-element vector | scalar

Filter state, specified as a real-valued *M*-element vector. A scalar input is extended to an *M*-element vector. The state vector is the concatenated states from each dimension. For example, if `MotionModel` is set to `'3D Constant Acceleration'`, the state vector is in the form: `[x; x'; x''; y; y'; y''; z; z'; z'']` where `'` and `''` indicate first and second order derivatives, respectively.

Example: `[200;0.2;150;0.1;0;0.25]`

Data Types: `double`

`StateCovariance` — State estimation error covariance
M-by-*M* matrix | scalar

State error covariance, specified as an *M*-by-*M* matrix, where *M* is the size of the filter state. A scalar input is extended to an *M*-by-*M* matrix. The covariance matrix represents the uncertainty in the filter state.

Example: `eye(6)`

`ProcessNoise` — Process noise covariance
D-by-*D* matrix | scalar

Process noise covariance, specified as a scalar or a *D*-by-*D* matrix, where *D* is the dimensionality of motion. For example, if `MotionModel` is `'2D Constant Velocity'`, then $D = 2$. A scalar input is extended to a *D*-by-*D* matrix.

Example: `[20 0.1; 0.1 1]`

`MeasurementNoise` — Measurement noise covariance
D-by-*D* matrix | scalar

Measurement noise covariance, specified as a scalar or a *D*-by-*D* matrix, where *D* is the dimensionality of motion. For example, if `MotionModel` is `'2D Constant Velocity'`, then $D = 2$. A scalar input is extended to a *M*-by-*M* matrix.

Example: `[20 0.1; 0.1 1]`

`Coefficients` — Alpha-beta filter coefficients
row vector | scalar

Alpha-beta filter coefficients, specified as a scalar or row vector. A scalar input is extended to a row vector. If you specify constant velocity in the `MotionModel` property, the coefficients are `[alpha beta]`. If you specify constant acceleration in the `MotionModel` property, the coefficients are `[alpha beta gamma]`.

Example: `[20 0.1]`

Object Functions

predict	Predict state and state estimation error covariance of tracking filter
correct	Correct state and state estimation error covariance using tracking filter
correctjpda	Correct state and state estimation error covariance using tracking filter and JPDA
distance	Distances between current and predicted measurements of tracking filter
likelihood	Likelihood of measurement from tracking filter
clone	Create duplicate tracking filter

Examples

Run `trackingABF` Filter

This example shows how to create and run a `trackingABF` filter. Call the `predict` and `correct` functions to track an object and correct the state estimation based on measurements.

Create the filter. Specify the initial state.

```
state = [1;2;3;4];
abf = trackingABF('State',state);
```

Call `predict` to get the predicted state and covariance of the filter. Use a 0.5 sec time step.

```
[xPred,pPred] = predict(abf, 0.5);
```

Call `correct` with a given measurement.

```
meas = [1;1];
[xCorr,pCorr] = correct(abf, meas);
```

Continue to predict the filter state. Specify the desired time step in seconds if necessary.

```
[xPred,pPred] = predict(abf);         % Predict over 1 second
[xPred,pPred] = predict(abf,2);       % Predict over 2 seconds
```

Modify the filter coefficients and correct again with a new measurement.

```
abf.Coefficients = [0.4 0.2];
[xCorr,pCorr] = correct(abf,[8;14]);
```

References

[1] Blackman, Samuel S. "*Multiple-target tracking with radar applications.*" Dedham, MA, Artech House, Inc., 1986, 463 p. (1986).

[2] Bar-Shalom, Yaakov, X. Rong Li, and Thiagalingam Kirubarajan. *Estimation with applications to tracking and navigation: theory algorithms and software*. John Wiley & Sons, 2004.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`multiObjectTracker` | `trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2020a

trackingEKF

Extended Kalman filter for object tracking

Description

A `trackingEKF` object is a discrete-time extended Kalman filter used to track the positions and velocities of objects that can be encountered in an automated driving scenario. Such objects include automobiles, pedestrians, bicycles, and stationary structures or obstacles.

A Kalman filter is a recursive algorithm for estimating the evolving state of a process when measurements are made on the process. The extended Kalman filter can model the evolution of a state when the state follows a nonlinear motion model, when the measurements are nonlinear functions of the state, or when both conditions apply. The extended Kalman filter is based on the linearization of the nonlinear equations. This approach leads to a filter formulation similar to the linear Kalman filter, `trackingKF`.

The process and measurements can have Gaussian noise, which you can include in these ways:

- Add noise to both the process and the measurements. In this case, the sizes of the process noise and measurement noise must match the sizes of the state vector and measurement vector, respectively.
- Add noise in the state transition function, the measurement model function, or in both functions. In these cases, the corresponding noise sizes are not restricted.

Creation

Syntax

```
filter = trackingEKF
filter = trackingEKF(transitionfcn,measurementfcn,state)
filter = trackingEKF( ___ ,Name,Value)
```

Description

`filter = trackingEKF` creates an extended Kalman filter object for a discrete-time system by using default values for the `StateTransitionFcn`, `MeasurementFcn`, and `State` properties. The process and measurement noises are assumed to be additive.

`filter = trackingEKF(transitionfcn,measurementfcn,state)` specifies the state transition function, `transitionfcn`, the measurement function, `measurementfcn`, and the initial state of the system, `state`.

`filter = trackingEKF( ___ ,Name,Value)` configures the properties of the extended Kalman filter object by using one or more `Name,Value` pair arguments and any of the previous syntaxes. Any unspecified properties have default values.

Properties

`State` — Kalman filter state
real-valued *M*-element vector

Kalman filter state, specified as a real-valued *M*-element vector, where *M* is the size of the filter state.

Example: `[200; 0.2]`

Data Types: `double`

`StateCovariance` — State estimation error covariance
positive-definite real-valued *M*-by-*M* matrix

State error covariance, specified as a positive-definite real-valued *M*-by-*M* matrix where *M* is the size of the filter state. The covariance matrix represents the uncertainty in the filter state.

Example: `[20 0.1; 0.1 1]`

`StateTransitionFcn` — State transition function
function handle

State transition function, specified as a function handle. This function calculates the state vector at time step `k` from the state vector at time step `k` – 1. The function can take additional input parameters, such as control inputs or time step size. The function can also include noise values.

The valid syntaxes for the state transition function depend on whether the filter has additive process noise. The table shows the valid syntaxes based on the value of the `HasAdditiveProcessNoise` property.

Valid Syntaxes (`HasAdditiveProcessNoise = true`)	Valid Syntaxes (`HasAdditiveProcessNoise = false`)

`x(k) = statetransitionfcn(x(k-1))`
`x(k) = statetransitionfcn(x(k-1),parameters)`
 	
• `x(k)` is the state at time `k`.
• `parameters` stands for all additional arguments required by the state transition function.
	`x(k) = statetransitionfcn(x(k-1),w(k-1))`
	`x(k) = statetransitionfcn(x(k-1),w(k-1),dt)`
	`x(k) = statetransitionfcn(__,parameters)`
	
	• `x(k)` is the state at time `k`.
	• `w(k)` is a value for the process noise at time `k`.
	• `dt` is the time step of the `trackingEKF` filter, `filter`, specified in the most recent call to the `predict` function. The `dt` argument applies when you use the filter within a tracker and call the `predict` function with the filter to predict the state of the tracker at the next time step. For the nonadditive process noise case, the tracker assumes that you explicitly specify the time step by using this syntax: `predict(filter,dt)`.
	• `parameters` stands for all additional arguments required by the state transition function.

Example: `@constacc`

Data Types: `function_handle`

`StateTransitionJacobianFcn` — Jacobian of state transition function

function handle

Jacobian of the state transition function, specified as a function handle. This function has the same input arguments as the state transition function.

The valid syntaxes for the Jacobian of the state transition function depend on whether the filter has additive process noise. The table shows the valid syntaxes based on the value of the `HasAdditiveProcessNoise` property.

Valid Syntaxes (`HasAdditiveProcessNoise = true`)	Valid Syntaxes (`HasAdditiveProcessNoise = false`)

`Jx(k) = statejacobianfcn(x(k))`
`Jx(k) = statejacobianfcn(x(k),parameters)`
 	
• `x(k)` is the state at time `k`.
• `Jx(k)` denotes the Jacobian of the predicted state with respect to the previous state. This Jacobian is an *M*-by-*M* matrix at time `k`. The Jacobian function can take additional input parameters, such as control inputs or time-step size.
• `parameters` stands for all additional arguments required by the Jacobian function, such as control inputs or time-step size.
	`[Jx(k),Jw(k)] = statejacobianfcn(x(k),w(k))`
	`[Jx(k),Jw(k)] = statejacobianfcn(x(k),w(k),dt)`
	`[Jx(k),Jw(k)] = statejacobianfcn(__,parameters)`
	
	• `x(k)` is the state at time `k`
	• `w(k)` is a sample *Q*-element vector of the process noise at time `k`. *Q* is the size of the process noise covariance. The process noise vector in the nonadditive case does not need to have the same dimensions as the state vector.
	• `Jx(k)` denotes the Jacobian of the predicted state with respect to the previous state. This Jacobian is an *M*-by-*M* matrix at time *k*. The Jacobian function can take additional input parameters, such as control inputs or time-step size.
	• `Jw(k)` denotes the *M*-by-*Q* Jacobian of the predicted state with respect to the process noise elements.
	• `dt` is the time step of the `trackingEKF` filter, `filter`, specified in the most recent call to the `predict` function. The `dt` argument applies when you use the filter within a tracker and call the `predict` function with the filter to predict the state of the tracker at the next time step. For the nonadditive process noise case, the tracker assumes that you explicitly specify the time step by using this syntax: `predict(filter,dt)`.
	• `parameters` stands for all additional arguments required by the Jacobian function, such as control inputs or time-step size.

If this property is not specified, the Jacobians are computed by numeric differencing at each call of the `predict` function. This computation can increase the processing time and numeric inaccuracy.

Example: `@constaccjac`

Data Types: `function_handle`

`ProcessNoise` — Process noise covariance
1 (default) | positive real scalar | positive-definite real-valued matrix

Process noise covariance, specified as a scalar or matrix.

- When `HasAdditiveProcessNoise` is `true`, specify the process noise covariance as a positive real scalar or a positive-definite real-valued *M*-by-*M* matrix. *M* is the dimension of the state vector. When specified as a scalar, the matrix is a multiple of the *M*-by-*M* identity matrix.
- When `HasAdditiveProcessNoise` is `false`, specify the process noise covariance as a *Q*-by-*Q* matrix. *Q* is the size of the process noise vector.

  You must specify `ProcessNoise` before any call to the `predict` function. In later calls to `predict`, you can optionally specify the process noise as a scalar. In this case, the process noise matrix is a multiple of the *Q*-by-*Q* identity matrix.

Example: `[1.0 0.05; 0.05 2]`

`HasAdditiveProcessNoise` — Model additive process noise
`true` (default) | `false`

Option to model process noise as additive, specified as `true` or `false`. When this property is `true`, process noise is added to the state vector. Otherwise, noise is incorporated into the state transition function.

`MeasurementFcn` — Measurement model function
function handle

Measurement model function, specified as a function handle. This function can be a nonlinear function that models measurements from the predicted state. Input to the function is the *M*-element state vector. The output is the *N*-element measurement vector. The function can take additional input arguments, such as sensor position and orientation.

- If `HasAdditiveMeasurementNoise` is `true`, specify the function using one of these syntaxes:

  ```
  z(k) = measurementfcn(x(k))
  ```

  ```
  z(k) = measurementfcn(x(k),parameters)
  ```

  `x(k)` is the state at time `k` and `z(k)` is the predicted measurement at time `k`. The `parameters` argument stands for all additional arguments required by the measurement function.
- If `HasAdditiveMeasurementNoise` is `false`, specify the function using one of these syntaxes:

  ```
  z(k) = measurementfcn(x(k),v(k))
  ```

  ```
  z(k) = measurementfcn(x(k),v(k),parameters)
  ```

  `x(k)` is the state at time `k` and `v(k)` is the measurement noise at time `k`. The `parameters` argument stands for all additional arguments required by the measurement function.

Example: `@cameas`

Data Types: `function_handle`

`MeasurementJacobianFcn` — Jacobian of measurement function
function handle

Jacobian of the measurement function, specified as a function handle. The function has the same input arguments as the measurement function. The function can take additional input parameters, such sensor position and orientation.

- If `HasAdditiveMeasurmentNoise` is `true`, specify the Jacobian function using one of these syntaxes:

  ```
  Jmx(k) = measjacobianfcn(x(k))
  ```

  ```
  Jmx(k) = measjacobianfcn(x(k),parameters)
  ```

  `x(k)` is the state at time `k`. `Jx(k)` denotes the *N*-by-*M* Jacobian of the measurement function with respect to the state. The `parameters` argument stands for all arguments required by the measurement function.
- If `HasAdditiveMeasurmentNoise` is `false`, specify the Jacobian function using one of these syntaxes:

  ```
  [Jmx(k),Jmv(k)] = measjacobianfcn(x(k),v(k))
  ```

  ```
  [Jmx(k),Jmv(k)] = measjacobianfcn(x(k),v(k),parameters)
  ```

  `x(k)` is the state at time `k` and `v(k)` is an *R*-dimensional sample noise vector. `Jmx(k)` denotes the *N*-by-*M* Jacobian of the measurement function with respect to the state. `Jmv(k)` denotes the Jacobian of the *N*-by-*R* measurement function with respect to the measurement noise. The `parameters` argument stands for all arguments required by the measurement function.

If not specified, measurement Jacobians are computed using numerical differencing at each call to the `correct` function. This computation can increase processing time and numerical inaccuracy.

Example: `@cameasjac`

Data Types: `function_handle`

`MeasurementNoise` — Measurement noise covariance
1 (default) | positive scalar | positive-definite real-valued matrix

Measurement noise covariance, specified as a positive scalar or positive-definite real-valued matrix.

- When `HasAdditiveMeasurementNoise` is `true`, specify the measurement noise covariance as a scalar or an *N*-by-*N* matrix. *N* is the size of the measurement vector. When specified as a scalar, the matrix is a multiple of the *N*-by-*N* identity matrix.
- When `HasAdditiveMeasurementNoise` is `false`, specify the measurement noise covariance as an *R*-by-*R* matrix. *R* is the size of the measurement noise vector.

  You must specify `MeasurementNoise` before any call to the `correct` function. After the first call to `correct`, you can optionally specify the measurement noise as a scalar. In this case, the measurement noise matrix is a multiple of the *R*-by-*R* identity matrix.

Example: `0.2`

`HasAdditiveMeasurmentNoise` — Model additive measurement noise
`true` (default) | `false`

Option to enable additive measurement noise, specified as `true` or `false`. When this property is `true`, noise is added to the measurement. Otherwise, noise is incorporated into the measurement function.

Object Functions

predict	Predict state and state estimation error covariance of tracking filter
correct	Correct state and state estimation error covariance using tracking filter
correctjpda	Correct state and state estimation error covariance using tracking filter and JPDA
distance	Distances between current and predicted measurements of tracking filter
likelihood	Likelihood of measurement from tracking filter
clone	Create duplicate tracking filter
residual	Measurement residual and residual noise from tracking filter
initialize	Initialize state and covariance of tracking filter

Examples

Constant-Velocity Extended Kalman Filter

Create a two-dimensional `trackingEKF` object and use name-value pairs to define the `StateTransitionJacobianFcn` and `MeasurementJacobianFcn` properties. Use the predefined constant-velocity motion and measurement models and their Jacobians.

```
EKF = trackingEKF(@constvel,@cvmeas,[0;0;0;0], ...
    'StateTransitionJacobianFcn',@constveljac, ...
    'MeasurementJacobianFcn',@cvmeasjac);
```

Run the filter. Use the `predict` and `correct` functions to propagate the state. You may call `predict` and `correct` in any order and as many times you want. Specify the measurement in Cartesian coordinates.

```
measurement = [1;1;0];
[xpred, Ppred] = predict(EKF);
[xcorr, Pcorr] = correct(EKF,measurement);
[xpred, Ppred] = predict(EKF);
[xpred, Ppred] = predict(EKF)
```

```
xpred = 4×1

    1.2500
    0.2500
    1.2500
    0.2500


Ppred = 4×4

   11.7500    4.7500         0         0
    4.7500    3.7500         0         0
         0         0   11.7500    4.7500
         0         0    4.7500    3.7500
```

More About

Filter Parameters

This table relates the filter model parameters to the object properties. *M* is the size of the state vector. *N* is the size of the measurement vector.

Filter Parameter	Description	Filter Property	Size
f	State transition function that specifies the equations of motion of the object. This function determines the state at time `k+1` as a function of the state and the controls at time `k`. The state transition function depends on the time-increment of the filter.	`StateTransitionFcn`	Function returns *M*-element vector
h	Measurement function that specifies how the measurements are functions of the state and measurement noise.	`MeasurementFcn`	Function returns *N*-element vector
x_k	Estimate of the object state.	`State`	*M*-element vector
P_k	State error covariance matrix representing the uncertainty in the values of the state.	`StateCovariance`	*M*-by-*M* matrix
Q_k	Estimate of the process noise covariance matrix at step `k`. Process noise is a measure of the uncertainty in the dynamic model. It is assumed to be zero-mean white Gaussian noise.	`ProcessNoise`	*M*-by-*M* matrix when `HasAdditiveProcessNoise` is `true`. *Q*-by-*Q* matrix when `HasAdditiveProcessNoise` is `false`
R_k	Estimate of the measurement noise covariance at step `k`. Measurement noise reflects the uncertainty of the measurement. It is assumed to be zero-mean white Gaussian noise.	`MeasurementNoise`	*N*-by-*N* matrix when `HasAdditiveMeasurementNoise` is `true`. *R*-by-*R* when `HasAdditiveMeasurementNoise` is `false`.

Filter Parameter	Description	Filter Property	Size
F	Function determining Jacobian of propagated state with respect to previous state.	`StateTransitionJacobianFcn`	M-by-M matrix
H	Function determining Jacobians of measurement with respect to the state and measurement noise.	`MeasurementJacobianFcn`	N-by-M for state vector Jacobian and N-by-R for measurement vector Jacobian

Algorithms

The extended Kalman filter estimates the state of a process governed by this nonlinear stochastic equation:

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

x_k is the state at step k. *f()* is the state transition function. Random noise perturbations, w_k, can affect the object motion. The filter also supports a simplified form,

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

To use the simplified form, set `HasAdditiveProcessNoise` to `true`.

In the extended Kalman filter, the measurements are also general functions of the state:

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

$h(x_k,v_k,t)$ is the measurement function that determines the measurements as functions of the state. Typical measurements are position and velocity or some function of position and velocity. The measurements can also include noise, represented by v_k. Again, the filter offers a simpler formulation.

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

To use the simplified form, set `HasAdditiveMeasurmentNoise` to `true`.

These equations represent the actual motion and the actual measurements of the object. However, the noise contribution at each step is unknown and cannot be modeled deterministically. Only the statistical properties of the noise are known.

References

[1] Brown, R.G. and P.Y.C. Wang. *Introduction to Random Signal Analysis and Applied Kalman Filtering*. 3rd Edition. New York: John Wiley & Sons, 1997.

[2] Kalman, R. E. “A New Approach to Linear Filtering and Prediction Problems.” *Transactions of the ASME–Journal of Basic Engineering*. Vol. 82, Series D, March 1960, pp. 35–45.

[3] Blackman, Samuel and R. Popoli. *Design and Analysis of Modern Tracking Systems*. Artech House.1999.

[4] Blackman, Samuel. *Multiple-Target Tracking with Radar Applications*. Artech House. 1986.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

cameas | cameasjac | constacc | constaccjac | constturn | constturnjac | constvel | constveljac | ctmeas | ctmeasjac | cvmeas | cvmeasjac | initcaekf | initctekf | initcvekf

Objects

multiObjectTracker | trackingABF | trackingKF | trackingUKF

Topics

"Extended Kalman Filters"

Introduced in R2017a

trackingUKF

Unscented Kalman filter for object tracking

Description

The `trackingUKF` object is a discrete-time unscented Kalman filter used to track the positions and velocities of objects that can be encountered in an automated driving scenario. Such objects include automobiles, pedestrians, bicycles, and stationary structures or obstacles.

An unscented Kalman filter is a recursive algorithm for estimating the evolving state of a process when measurements are made on the process. The unscented Kalman filter can model the evolution of a state that obeys a nonlinear motion model. The measurements can also be nonlinear functions of the state, and the process and measurements can have noise.

Use an unscented Kalman filter when one of both of these conditions apply:

- The current state is a nonlinear function of the previous state.
- The measurements are nonlinear functions of the state.

The unscented Kalman filter estimates the uncertainty about the state, and its propagation through the nonlinear state and measurement equations, by using a fixed number of sigma points. Sigma points are chosen by using the unscented transformation, as parameterized by the `Alpha`, `Beta`, and `Kappa` properties.

Creation

Syntax

```
filter = trackingUKF
filter = trackingUKF(transitionfcn,measurementfcn,state)
filter = trackingUKF( ___ ,Name,Value)
```

Description

`filter = trackingUKF` creates an unscented Kalman filter object for a discrete-time system by using default values for the `StateTransitionFcn`, `MeasurementFcn`, and `State` properties. The process and measurement noises are assumed to be additive.

`filter = trackingUKF(transitionfcn,measurementfcn,state)` specifies the state transition function, `transitionfcn`, the measurement function, `measurementfcn`, and the initial state of the system, `state`.

`filter = trackingUKF( ___ ,Name,Value)` configures the properties of the unscented Kalman filter object using one or more `Name,Value` pair arguments and any of the previous syntaxes. Any unspecified properties have default values.

Properties

`State` — Kalman filter state

real-valued *M*-element vector

Kalman filter state, specified as a real-valued *M*-element vector, where *M* is the size of the filter state.

Example: `[200; 0.2]`

Data Types: `double`

`StateCovariance` — State estimation error covariance

positive-definite real-valued *M*-by-*M* matrix

State error covariance, specified as a positive-definite real-valued *M*-by-*M* matrix where *M* is the size of the filter state. The covariance matrix represents the uncertainty in the filter state.

Example: `[20 0.1; 0.1 1]`

`StateTransitionFcn` — State transition function

function handle

State transition function, specified as a function handle. This function calculates the state vector at time step `k` from the state vector at time step `k` – 1. The function can take additional input parameters, such as control inputs or time step size. The function can also include noise values.

The valid syntaxes for the state transition function depend on whether the filter has additive process noise. The table shows the valid syntaxes based on the value of the `HasAdditiveProcessNoise` property.

Valid Syntaxes (`HasAdditiveProcessNoise = true`)	Valid Syntaxes (`HasAdditiveProcessNoise = false`)

`x(k) = statetransitionfcn(x(k-1))`
`x(k) = statetransitionfcn(x(k-1),parameters)`
 	
• `x(k)` is the state at time `k`.
• `parameters` stands for all additional arguments required by the state transition function.
	`x(k) = statetransitionfcn(x(k-1),w(k-1))`
	`x(k) = statetransitionfcn(x(k-1),w(k-1),dt)`
	`x(k) = statetransitionfcn(__,parameters)`
	
	• `x(k)` is the state at time `k`.
	• `w(k)` is a value for the process noise at time `k`.
	• `dt` is the time step of the `trackingUKF` filter, `filter`, specified in the most recent call to the `predict` function. The `dt` argument applies when you use the filter within a tracker and call the `predict` function with the filter to predict the state of the tracker at the next time step. For the nonadditive process noise case, the tracker assumes that you explicitly specify the time step by using this syntax: `predict(filter,dt)`.
	• `parameters` stands for all additional arguments required by the state transition function.

Example: `@constacc`

Data Types: `function_handle`

`ProcessNoise` — Process noise covariance
1 (default) | positive real scalar | positive-definite real-valued matrix

Process noise covariance, specified as a scalar or matrix.

- When `HasAdditiveProcessNoise` is `true`, specify the process noise covariance as a positive real scalar or a positive-definite real-valued *M*-by-*M* matrix. *M* is the dimension of the state vector. When specified as a scalar, the matrix is a multiple of the *M*-by-*M* identity matrix.
- When `HasAdditiveProcessNoise` is `false`, specify the process noise covariance as a *Q*-by-*Q* matrix. *Q* is the size of the process noise vector.

  You must specify `ProcessNoise` before any call to the `predict` function. In later calls to `predict`, you can optionally specify the process noise as a scalar. In this case, the process noise matrix is a multiple of the *Q*-by-*Q* identity matrix.

Example: `[1.0 0.05; 0.05 2]`

`HasAdditiveProcessNoise` — Model additive process noise
`true` (default) | `false`

Option to model process noise as additive, specified as `true` or `false`. When this property is `true`, process noise is added to the state vector. Otherwise, noise is incorporated into the state transition function.

`MeasurementFcn` — Measurement model function
function handle

Measurement model function, specified as a function handle. This function can be a nonlinear function that models measurements from the predicted state. Input to the function is the *M*-element state vector. The output is the *N*-element measurement vector. The function can take additional input arguments, such as sensor position and orientation.

- If `HasAdditiveMeasurementNoise` is `true`, specify the function using one of these syntaxes:

  ```
  z(k) = measurementfcn(x(k))
  ```

  ```
  z(k) = measurementfcn(x(k),parameters)
  ```

  `x(k)` is the state at time `k` and `z(k)` is the predicted measurement at time `k`. The `parameters` argument stands for all additional arguments required by the measurement function.
- If `HasAdditiveMeasurementNoise` is `false`, specify the function using one of these syntaxes:

  ```
  z(k) = measurementfcn(x(k),v(k))
  ```

  ```
  z(k) = measurementfcn(x(k),v(k),parameters)
  ```

  `x(k)` is the state at time `k` and `v(k)` is the measurement noise at time `k`. The `parameters` argument stands for all additional arguments required by the measurement function.

Example: `@cameas`

Data Types: `function_handle`

`MeasurementNoise` — Measurement noise covariance
1 (default) | positive scalar | positive-definite real-valued matrix

Measurement noise covariance, specified as a positive scalar or positive-definite real-valued matrix.

- When `HasAdditiveMeasurementNoise` is `true`, specify the measurement noise covariance as a scalar or an *N*-by-*N* matrix. *N* is the size of the measurement vector. When specified as a scalar, the matrix is a multiple of the *N*-by-*N* identity matrix.
- When `HasAdditiveMeasurementNoise` is `false`, specify the measurement noise covariance as an *R*-by-*R* matrix. *R* is the size of the measurement noise vector.

  You must specify `MeasurementNoise` before any call to the `correct` function. After the first call to `correct`, you can optionally specify the measurement noise as a scalar. In this case, the measurement noise matrix is a multiple of the *R*-by-*R* identity matrix.

Example: `0.2`

`HasAdditiveMeasurmentNoise` — Model additive measurement noise
`true` (default) | `false`

Option to enable additive measurement noise, specified as `true` or `false`. When this property is `true`, noise is added to the measurement. Otherwise, noise is incorporated into the measurement function.

`Alpha` — Sigma point spread around state
`1.0e-3` (default) | positive scalar greater than 0 and less than or equal to 1

Sigma point spread around state, specified as a positive scalar greater than 0 and less than or equal to 1.

`Beta` — Distribution of sigma points
2 (default) | nonnegative scalar

Distribution of sigma points, specified as a nonnegative scalar. This parameter incorporates knowledge of the noise distribution of states for generating sigma points. For Gaussian distributions, setting `Beta` to 2 is optimal.

`Kappa` — Secondary scaling factor for generating sigma points
`0` (default) | scalar from 0 to 3

Secondary scaling factor for generation of sigma points, specified as a scalar from 0 to 3. This parameter helps specify the generation of sigma points.

Object Functions

predict	Predict state and state estimation error covariance of tracking filter
correct	Correct state and state estimation error covariance using tracking filter
correctjpda	Correct state and state estimation error covariance using tracking filter and JPDA
distance	Distances between current and predicted measurements of tracking filter
likelihood	Likelihood of measurement from tracking filter
clone	Create duplicate tracking filter
residual	Measurement residual and residual noise from tracking filter
initialize	Initialize state and covariance of tracking filter

Examples

Constant-Velocity Unscented Kalman Filter

Create a `trackingUKF` object using the predefined constant-velocity motion model, `constvel`, and the associated measurement model, `cvmeas`. These models assume that the state vector has the form [x;vx;y;vy] and that the position measurement is in Cartesian coordinates, [x;y;z]. Set the sigma point spread property to 1e-2.

```
filter = trackingUKF(@constvel,@cvmeas,[0;0;0;0],'Alpha',1e-2);
```

Run the filter. Use the `predict` and `correct` functions to propagate the state. You can call `predict` and `correct` in any order and as many times as you want.

```
meas = [1;1;0];
[xpred, Ppred] = predict(filter);
[xcorr, Pcorr] = correct(filter,meas);
[xpred, Ppred] = predict(filter);
[xpred, Ppred] = predict(filter)
```

```
xpred = 4×1

    1.2500
    0.2500
    1.2500
    0.2500


Ppred = 4×4

   11.7500    4.7500   -0.0000    0.0000
    4.7500    3.7500    0.0000   -0.0000
   -0.0000    0.0000   11.7500    4.7500
    0.0000   -0.0000    4.7500    3.7500
```

More About

Filter Parameters

This table relates the filter model parameters to the object properties. *M* is the size of the state vector. *N* is the size of the measurement vector.

Model Parameter	Description	Filter Property	Size
f	State transition function that specifies the equations of motion of the object. This function determines the state at time `k+1` as a function of the state and the controls at time `k`. The state transition function depends on the time-increment of the filter.	`StateTransitionFcn`	Function returns *M*-element vector

Model Parameter	Description	Filter Property	Size
h	Measurement function that specifies how the measurements are functions of the state and measurement noise.	`MeasurementFcn`	Function returns *N*-element vector
x_k	Estimate of the object state.	`State`	*M*
P_k	State error covariance matrix representing the uncertainty in the values of the state	`StateCovariance`	*M*-by-*M*
Q_k	Estimate of the process noise covariance matrix at step `k`. Process noise is measure of the uncertainty in your dynamic model and is assumed to be zero-mean white Gaussian noise	`ProcessNoise`	*M*-by-*M* when `HasAdditiveProcessNoise` is `true`. *Q*-by-*Q* when `HasAdditiveProcessNoise`is `false`.
R_k	Estimate of the measurement noise covariance at step *k*. Measurement noise reflects the uncertainty of the measurement and is assumed to be zero-mean white Gaussian noise.	`MeasurementNoise`	*N*-by-*N* when `HasAdditiveMeasurementNoise` is `true`. *R*-by-*R* when `HasAdditiveMeasurementNoise` is `false`.
α	Determines spread of sigma points.	`Alpha`	scalar
β	*A priori* knowledge of sigma point distribution.	`Beta`	scalar
κ	Secondary scaling parameter.	`Kappa`	scalar

Algorithms

The unscented Kalman filter estimates the state of a process governed by a nonlinear stochastic equation

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

where x_k is the state at step k. *f()* is the state transition function, u_k are the controls on the process. The motion may be affected by random noise perturbations, w_k. The filter also supports a simplified form,

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

To use the simplified form, set `HasAdditiveProcessNoise` to `true`.

In the unscented Kalman filter, the measurements are also general functions of the state,

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

where $h(x_k,v_k,t)$ is the measurement function that determines the measurements as functions of the state. Typical measurements are position and velocity or some function of these. The measurements can include noise as well, represented by v_k. Again the class offers a simpler formulation

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

To use the simplified form, set `HasAdditiveMeasurmentNoise` to `true`.

These equations represent the actual motion of the object and the actual measurements. However, the noise contribution at each step is unknown and cannot be modeled exactly. Only statistical properties of the noise are known.

References

[1] Brown, R.G. and P.Y.C. Wang. *Introduction to Random Signal Analysis and Applied Kalman Filtering*. 3rd Edition. New York: John Wiley & Sons, 1997.

[2] Kalman, R. E. “A New Approach to Linear Filtering and Prediction Problems.” *Transactions of the ASME–Journal of Basic Engineering*. Vol. 82, Series D, March 1960, pp. 35–45.

[3] Wan, Eric A. and R. van der Merwe. “The Unscented Kalman Filter for Nonlinear Estimation”. *Adaptive Systems for Signal Processing, Communications, and Control*. AS-SPCC, IEEE, 2000, pp.153–158.

[4] Wan, Merle. “The Unscented Kalman Filter.” In *Kalman Filtering and Neural Networks*. Edited by Simon Haykin. John Wiley & Sons, Inc., 2001.

[5] Sarkka S. “Recursive Bayesian Inference on Stochastic Differential Equations.” Doctoral Dissertation. Helsinki University of Technology, Finland. 2006.

[6] Blackman, Samuel. *Multiple-Target Tracking with Radar Applications*. Artech House, 1986.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`cameas` | `cameasjac` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `constveljac` | `ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac` | `initcaukf` | `initctukf` | `initcvukf`

Objects
`multiObjectTracker` | `trackingABF` | `trackingEKF` | `trackingKF`

Introduced in R2017a

clone

Create duplicate tracking filter

Syntax

```
filterClone = clone(filter)
```

Description

`filterClone = clone(filter)` creates a copy of a tracking filter that has the same property values as the original filter.

Input Arguments

`filter` — Filter for object tracking

`trackingKF` object | `trackingEKF` object | `trackingUKF` object

Filter for object tracking, specified as one of these objects:

- `trackingKF` — Linear Kalman filter
- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter
- `trackingABF` — Alpha-beta filter

Output Arguments

`filterClone` — Cloned filter

tracking filter object

Cloned filter, returned as a tracking filter object of the same type as `filter`. The cloned filter has the same properties as the original filter.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`correct` | `correctjpda` | `distance` | `initialize` | `likelihood` | `predict` | `residual`

Introduced in R2017a

correct

Correct state and state estimation error covariance using tracking filter

Syntax

```
[xcorr,Pcorr] = correct(filter,zmeas)

[xcorr,Pcorr] = correct(filter,zmeas,measparams)

[xcorr,Pcorr] = correct(filter,zmeas,zcov)

[xcorr,Pcorr,zcorr] = correct(filter,zmeas)
[xcorr,Pcorr,zcorr] = correct(filter,zmeas,zcov)

correct(filter,___)
xcorr = correct(filter,___)
```

Description

`[xcorr,Pcorr] = correct(filter,zmeas)` returns the corrected state, `xcorr`, and the corrected state estimation error covariance, `Pcorr`, for the next time step of the input tracking filter based on the current measurement, `zmeas`. The corrected values overwrite the internal state and state estimation error covariance of `filter`.

`[xcorr,Pcorr] = correct(filter,zmeas,measparams)` specifies additional parameters used by the measurement function that is defined in the `MeasurementFcn` property of `filter`. You can return any of the outputs from preceding syntaxes.

If filter is a `trackingKF` or `trackingABF` object, then you cannot use this syntax.

`[xcorr,Pcorr] = correct(filter,zmeas,zcov)` specifies additional measurement covariance, `zcov`, used in the `MeasurementNoise` property of `filter`.

You can use this syntax only when `filter` is a `trackingKF` object.

`[xcorr,Pcorr,zcorr] = correct(filter,zmeas)` also returns the correction of measurements, `zcorr`.

You can use this syntax only when `filter` is a `trackingABF` object.

`[xcorr,Pcorr,zcorr] = correct(filter,zmeas,zcov)` returns the correction of measurements, `zcorr`, and also specifies additional measurement covariance, `zcov`, used in the `MeasurementNoise` property of `filter`.

You can use this syntax only when `filter` is a `trackingABF` object.

`correct(filter,___)` updates `filter` with the corrected state and state estimation error covariance without returning the corrected values. Specify the tracking filter and any of the input argument combinations from preceding syntaxes.

`xcorr = correct(filter,___)` updates `filter` with the corrected state and state estimation error covariance but returns only the corrected state, `xcorr`.

Examples

Constant-Velocity Extended Kalman Filter

Create a two-dimensional trackingEKF object and use name-value pairs to define the StateTransitionJacobianFcn and MeasurementJacobianFcn properties. Use the predefined constant-velocity motion and measurement models and their Jacobians.

```
EKF = trackingEKF(@constvel,@cvmeas,[0;0;0;0], ...
    'StateTransitionJacobianFcn',@constveljac, ...
    'MeasurementJacobianFcn',@cvmeasjac);
```

Run the filter. Use the predict and correct functions to propagate the state. You may call predict and correct in any order and as many times you want. Specify the measurement in Cartesian coordinates.

```
measurement = [1;1;0];
[xpred, Ppred] = predict(EKF);
[xcorr, Pcorr] = correct(EKF,measurement);
[xpred, Ppred] = predict(EKF);
[xpred, Ppred] = predict(EKF)
```

```
xpred = 4×1

    1.2500
    0.2500
    1.2500
    0.2500


Ppred = 4×4

   11.7500    4.7500         0         0
    4.7500    3.7500         0         0
         0         0   11.7500    4.7500
         0         0    4.7500    3.7500
```

Input Arguments

filter — Filter for object tracking

trackingKF object | trackingEKF object | trackingUKF object

Filter for object tracking, specified as one of these objects:

- trackingKF — Linear Kalman filter
- trackingEKF — Extended Kalman filter
- trackingUKF — Unscented Kalman filter
- trackingABF — Alpha-beta filter

zmeas — Measurement of filter

vector | matrix

Measurement of the tracked object, specified as a vector or matrix.

Data Types: single | double

measparams — Measurement parameters
comma-separated list of arguments

Measurement function arguments, specified as a comma-separated list of arguments. These arguments are the same ones that are passed into the measurement function specified by the MeasurementFcn property of the tracking filter. If filter is a trackingKF or trackingABF object, then you cannot specify measparams.

Suppose you set MeasurementFcn to @cameas, and then call correct:

```
[xcorr,Pcorr] = correct(filter,frame,sensorpos,sensorvel)
```

The correct function internally calls the following:

```
meas = cameas(state,frame,sensorpos,sensorvel)
```

zcov — Measurement covariance
M-by-*M* matrix

Measurement covariance, specified as an *M*-by-*M* matrix, where *M* is the dimension of the measurement. The same measurement covariance matrix is assumed for all measurements in zmeas.

Data Types: single | double

Output Arguments

xcorr — Corrected state of filter
vector | matrix

Corrected state of the filter, specified as a vector or matrix. The State property of the input filter is overwritten with this value.

Pcorr — Corrected state covariance of filter
vector | matrix

Corrected state covariance of the filter, specified as a vector or matrix. The StateCovariance property of the input filter is overwritten with this value.

zcorr — Corrected measurement of filter
vector | matrix

Corrected measurement of the filter, specified as a vector or matrix. You can return zcorr only when filter is a trackingABF object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

clone | correctjpda | distance | initialize | likelihood | predict | residual

Introduced in R2017a

correctjpda

Correct state and state estimation error covariance using tracking filter and JPDA

Syntax

```
[xcorr,Pcorr] = correctjpda(filter,zmeas)
```

```
[xcorr,Pcorr] = correctjpda(filter,zmeas,jpdacoeffs,measparams)
```

```
[xcorr,Pcorr] = correctjpda(filter,zmeas,jpdacoeffs,zcov)
```

```
[xcorr,Pcorr,zcorr] = correctjpda(filter,zmeas,jpdacoeffs)
[xcorr,Pcorr,zcorr] = correctjpda(filter,zmeas,jpdacoeffs,zcov)
```

```
correctjpda(filter, ___ )
xcorr = correctjpda(filter, ___ )
```

Description

`[xcorr,Pcorr] = correctjpda(filter,zmeas)` returns the corrected state, `xcorr`, and the corrected state estimation error covariance, `Pcorr`, for the next time step of the input tracking filter. The corrected values are based on a set of measurements, `zmeas`, and their joint probabilistic data association coefficients, `jpdacoeffs`. These values overwrite the internal state and state estimation error covariance of `filter`.

`[xcorr,Pcorr] = correctjpda(filter,zmeas,jpdacoeffs,measparams)` specifies additional parameters used by the measurement function that is defined in the `MeasurementFcn` property of the tracking filter object.

If filter is a `trackingKF` or `trackingABF` object, then you cannot use this syntax.

`[xcorr,Pcorr] = correctjpda(filter,zmeas,jpdacoeffs,zcov)` specifies additional measurement covariance, `zcov`, used in the `MeasurementNoise` property of `filter`.

You can use this syntax only when `filter` is a `trackingKF` object.

`[xcorr,Pcorr,zcorr] = correctjpda(filter,zmeas,jpdacoeffs)` also returns the correction of measurements, `zcorr`.

You can use this syntax only when `filter` is a `trackingABF` object.

`[xcorr,Pcorr,zcorr] = correctjpda(filter,zmeas,jpdacoeffs,zcov)` returns the correction of measurements, `zcorr`, and also specifies additional measurement covariance, `zcov`, used in the `MeasurementNoise` property of `filter`.

You can use this syntax only when `filter` is a `trackingABF` object.

`correctjpda(filter, ___ )` updates `filter` with the corrected state and state estimation error covariance without returning the corrected values. Specify the tracking filter and any of the input argument combinations from preceding syntaxes.

xcorr = correctjpda(filter,___) updates filter with the corrected state and state estimation error covariance but returns only the corrected state, xcorr.

Input Arguments

filter — Filter for object tracking
trackingKF object | trackingEKF object | trackingUKF object

Filter for object tracking, specified as one of these objects:

- trackingKF — Linear Kalman filter
- trackingEKF — Extended Kalman filter
- trackingUKF — Unscented Kalman filter
- trackingABF — Alpha-beta filter

zmeas — Measurements
M-by-*N* matrix

Measurements, specified as an *M*-by-*N* matrix, where *M* is the dimension of a single measurement, and *N* is the number of measurements.

Data Types: single | double

jpdacoeffs — Joint probabilistic data association coefficients
(*N*+1)-element vector

Joint probabilistic data association coefficients, specified as an (*N*+1)-element vector. The *i*th (i = 1, …, *N*) element of jpdacoeffs is the joint probability that the *i*th measurement in zmeas is associated with the filter. The last element of jpdacoeffs corresponds to the probability that no measurement is associated with the filter. The sum of all elements of jpdacoeffs must equal 1.

Data Types: single | double

zcov — Measurement covariance
M-by-*M* matrix

Measurement covariance, specified as an *M*-by-*M* matrix, where *M* is the dimension of the measurement. The same measurement covariance matrix is assumed for all measurements in zmeas.

Data Types: single | double

measparams — Measurement parameters
comma-separated list of arguments

Measurement function arguments, specified as a comma-separated list of arguments. These arguments are the same ones that are passed into the measurement function specified by the MeasurementFcn property of the tracking filter. If filter is a trackingKF or trackingABF object, then you cannot specify measparams.

Suppose you set MeasurementFcn to @cameas, and then call correctjpda:

```
[xcorr,Pcorr] = correctjpda(filter,frame,sensorpos,sensorvel)
```

The correctjpda function internally calls the following:

```
meas = cameas(state,frame,sensorpos,sensorvel)
```

Output Arguments

`xcorr` — Corrected state
P-element vector

Corrected state, returned as a *P*-element vector, where *P* is the dimension of the estimated state. The corrected state represents the *a posteriori* estimate of the state vector, taking into account the current measurements and their associated probabilities.

`Pcorr` — Corrected state error covariance
positive-definite *P*-by-*P* matrix

Corrected state error covariance, returned as a positive-definite *P*-by-*P* matrix, where *P* is the dimension of the state estimate. The corrected state covariance matrix represents the *a posteriori* estimate of the state covariance matrix, taking into account the current measurements and their associated probabilities.

`zcorr` — Corrected measurements
M-by-*N* matrix

Corrected measurements, returned as an *M*-by-*N* matrix, where *M* is the dimension of a single measurement, and *N* is the number of measurements. You can return `zcorr` only when `filter` is a `trackingABF` object.

More About

JPDA Correction Algorithm for Discrete Extended Kalman Filter

In the measurement update of a regular Kalman filter, the filter usually only needs to update the state and covariance based on one measurement. For instance, the equations for measurement update of a discrete extended Kalman filter can be given as

$$x_k^+ = x_k^- + K_k(y - h(x_k^-))$$
$$P_k^+ = P_k^- - K_k S_k K_k^T$$

where x_k^- and x_k^+ are the a priori and a posteriori state estimates, respectively, K_k is the Kalman gain, y is the actual measurement, and $h(x_k^-)$ is the predicted measurement. P_k^- and P_k^+ are the a priori and a posteriori state error covariance matrices, respectively. The innovation matrix S_k is defined as

$$S_k = H_k P_k^- H_k^T$$

where H_k is the Jacobian matrix for the measurement function h.

In the workflow of a JPDA tracker, the filter needs to process multiple probable measurements y_i (i = 1, …, N) with varied probabilities of association β_i (i = 0, 1, …, N). Note that β_0 is the probability that no measurements is associated with the filter. The measurement update equations for a discrete extended Kalman filter used for a JPDA tracker are

$$x_k^+ = x_k^- + K_k \sum_{i=1}^{N} \beta_i (y_i - h(x_k^-))$$
$$P_k^+ = P_k^- - (1 - \beta_0) K_k S_k K_k^T + P_k$$

where

$$P_k = K_k \sum_{i=1}^{N} \left[\beta_i (y_i - h(x_k -))(y_i - h(x_k -))^T - (\delta y)(\delta y)^T\right] K_k T$$

and

$$\delta y = \sum_{j=1}^{N} \beta_j \left(y_j - h(x_k -)\right)$$

Note that these equations only apply to `trackingEKF` and are not the exact equations used in other tracking filters.

References

[1] Fortmann, T., Y. Bar-Shalom, and M. Scheffe. "Sonar Tracking of Multiple Targets Using Joint Probabilistic Data Association." *IEEE Journal of Ocean Engineering.* Vol. 8, Number 3, 1983, pp. 173–184.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

`correctjpda` supports only double-precision code generation, not single-precision.

See Also

`clone` | `correct` | `distance` | `initialize` | `likelihood` | `predict` | `residual`

Introduced in R2019a

distance

Distances between current and predicted measurements of tracking filter

Syntax

```
dist = distance(filter,zmeas)
dist = distance(filter,zmeas,measparams)
```

Description

`dist = distance(filter,zmeas)` computes the normalized distances between one or more current object measurements, `zmeas`, and the corresponding predicted measurements computed by the input `filter`. Use this function to assign measurements to tracks.

This distance computation takes into account the covariance of the predicted state and the measurement noise.

`dist = distance(filter,zmeas,measparams)` specifies additional parameters that are used by the `MeasurementFcn` of the filter.

If filter is a `trackingKF` or `trackingABF` object, then you cannot use this syntax.

Input Arguments

`filter` — Filter for object tracking
`trackingKF` object | `trackingEKF` object | `trackingUKF` object

Filter for object tracking, specified as one of these objects:

- `trackingKF` — Linear Kalman filter
- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter
- `trackingABF` — Alpha-beta filter

`zmeas` — Measurements of tracked objects
matrix

Measurements of tracked objects, specified as a matrix. Each row of the matrix contains a measurement vector.

`measparams` — Parameters for measurement function
cell array

Parameters for measurement function, specified as a cell array. The parameters are passed to the measurement function that is defined in the `MeasurementFcn` property of the `filter`. If `filter` is a `trackingKF` or `trackingABF` object, then you cannot specify `measparams`.

Suppose you set the `MeasurementFcn` property of `filter` to `@cameas`, and then set these values:

```
measurementParams = {frame,sensorpos,sensorpos}
```

The `distance` function internally calls the following:

```
cameas(state,frame,sensorpos,sensorvel)
```

Output Arguments

`dist` — Distances between measurements
row vector

Distances between measurements, returned as a row vector. Each element corresponds to a distance between the predicted measurement in the input `filter` and a measurement contained in a row of `zmeas`.

Algorithms

The `distance` function computes the normalized distance between the filter object and a set of measurements. This distance computation is a variant of the Mahalanobis distance and takes into account the residual (the difference between the object measurement and the value predicted by the filter), the residual covariance, and the measurement noise.

Consider an extended Kalman filter with state x and measurement z. The equations used to compute the residual, z_{res}, and the residual covariance, S, are

$$z_{res} = z - h(x),$$
$$S = R + HPH^T,$$

where:

- h is the measurement function defined in the `MeasurementFcn` property of the filter.
- R is the measurement noise covariance defined in the `MeasurementNoise` property of the filter.
- H is the Jacobian of the measurement function defined in the `MeasurementJacobianFcn` property of the filter.

The residual covariance calculation for other filters can vary slightly from the one shown because tracking filters have different ways of propagating the covariance to the measurement space. For example, instead of using the Jacobian of the measurement function to propagate the covariance, unscented Kalman filters sample the covariance, and then propagate the sampled points.

The equation for the Mahalanobis distance, d^2, is

$$d^2 = z_{res}{}^T S^{-1} z,$$

The distance function computes the normalized distance, d_n, as

$$d_n = d^2 + \log(|S|),$$

where $\log(|S|)$ is the logarithm of the determinant of residual covariance S.

The $\log(|S|)$ term accounts for tracks that are coasted, meaning that they are predicted but have not had an update for a long time. Tracks in this state can make S very large, resulting in a smaller Mahalanobis distance relative to the updated tracks. This difference in distance values can cause the coasted tracks to incorrectly take detections from the updated tracks. The $\log(|S|)$ term compensates for this effect by penalizing such tracks, whose predictions are highly uncertain.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`clone` | `correct` | `correctjpda` | `initialize` | `likelihood` | `predict` | `residual`

Introduced in R2017a

initialize

Initialize state and covariance of tracking filter

Syntax

```
initialize(filter,state,statecov)
initialize(filter,state,statecov,Name,Value)
```

Description

`initialize(filter,state,statecov)` initializes the filter by setting the `State` and `StateCovariance` properties of the `filter` with the corresponding `state` and `statecov` inputs.

`initialize(filter,state,statecov,Name,Value)` also initializes properties of `filter` by using one or more name-value pairs. Specify the name of the filter property and the value to which you want to initialize it. You cannot change the size or type of the properties that you initialize.

Input Arguments

`filter` — Filter for object tracking
`trackingKF` object | `trackingEKF` object | `trackingUKF` object

Filter for object tracking, specified as one of these objects:

- `trackingKF` — Linear Kalman filter
- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter

`state` — Filter state
real-valued *M*-element vector

Filter state, specified as a real-valued *M*-element vector, where *M* is the size of the filter state.

Example: `[200; 0.2]`

Data Types: `double`

`statecov` — State estimation error covariance
positive-definite real-valued *M*-by-*M* matrix

State estimation error covariance, specified as a positive-definite real-valued *M*-by-*M* matrix. *M* is the size of the filter state. The covariance matrix represents the uncertainty in the filter state.

Example: `[20 0.1; 0.1 1]`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

clone | correct | correctjpda | distance | likelihood | predict | residual

Introduced in R2018b

likelihood

Likelihood of measurement from tracking filter

Syntax

```
measlikelihood = likelihood(filter,zmeas)
measlikelihood = likelihood(filter,zmeas,measparams)
```

Description

`measlikelihood = likelihood(filter,zmeas)` returns the likelihood of a measurement, `zmeas`, that was produced by the specified filter, `filter`.

`measlikelihood = likelihood(filter,zmeas,measparams)` specifies additional parameters that are used by the `MeasurementFcn` of the filter.

If filter is a `trackingKF` or `trackingABF` object, then you cannot use this syntax.

Input Arguments

`filter` — Filter for object tracking
`trackingKF` object | `trackingEKF` object | `trackingUKF` object

Filter for object tracking, specified as one of these objects:

- `trackingKF` — Linear Kalman filter
- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter
- `trackingABF` — Alpha-beta filter

`zmeas` — Current measurement of tracked object
vector | matrix

Current measurement of a tracked object, specified a vector or matrix.

`measparams` — Parameters for measurement function
cell array

Parameters for measurement function, specified as a cell array. The parameters are passed to the measurement function that is defined in the `MeasurementFcn` of the input `filter`. If `filter` is a `trackingKF` or `trackingABF` object, then you cannot specify `measparams`.

Output Arguments

`measlikelihood` — Likelihood of measurement
scalar

Likelihood of measurement, returned as a scalar.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

clone | correct | correctjpda | distance | initialize | predict | residual

Introduced in R2018a

predict

Predict state and state estimation error covariance of tracking filter

Syntax

```
[xpred,Ppred] = predict(filter)

[xpred,Ppred] = predict(filter,dt)
[xpred,Ppred] = predict(filter,predparams)

[xpred,Ppred,zpred] = predict(filter)
[xpred,Ppred,zpred] = predict(filter,dt)

predict(filter, ___ )
xpred = predict(filter, ___ )
```

Description

`[xpred,Ppred] = predict(filter)` returns the predicted state, `xpred`, and the predicted state estimation error covariance, `Ppred`, for the next time step of the input tracking filter. The predicted values overwrite the internal state and state estimation error covariance of `filter`.

`[xpred,Ppred] = predict(filter,dt)` specifies the time step as a positive scalar in seconds, and returns one or more of the outputs from the preceding syntaxes.

`[xpred,Ppred] = predict(filter,predparams)` specifies additional prediction parameters used by the state transition function. The state transition function is defined in the `StateTransitionFcn` property of `filter`.

`[xpred,Ppred,zpred] = predict(filter)` also returns the predicted measurement at the next time step.

You can use this syntax only when `filter` is a `trackingABF` object.

`[xpred,Ppred,zpred] = predict(filter,dt)` returns the predicted state, state estimation error covariance, and measurement at the specified time step.

You can use this syntax only when `filter` is a `trackingABF` object.

`predict(filter, ___ )` updates `filter` with the predicted state and state estimation error covariance without returning the predicted values. Specify the tracking filter and any of the input argument combinations from preceding syntaxes.

`xpred = predict(filter, ___ )` updates `filter` with the predicted state and state estimation error covariance but returns only the predicted state, `xpred`.

Examples

Constant-Velocity Extended Kalman Filter

Create a two-dimensional `trackingEKF` object and use name-value pairs to define the `StateTransitionJacobianFcn` and `MeasurementJacobianFcn` properties. Use the predefined constant-velocity motion and measurement models and their Jacobians.

```
EKF = trackingEKF(@constvel,@cvmeas,[0;0;0;0], ...
    'StateTransitionJacobianFcn',@constveljac, ...
    'MeasurementJacobianFcn',@cvmeasjac);
```

Run the filter. Use the `predict` and `correct` functions to propagate the state. You may call `predict` and `correct` in any order and as many times you want. Specify the measurement in Cartesian coordinates.

```
measurement = [1;1;0];
[xpred, Ppred] = predict(EKF);
[xcorr, Pcorr] = correct(EKF,measurement);
[xpred, Ppred] = predict(EKF);
[xpred, Ppred] = predict(EKF)
```

```
xpred = 4×1

    1.2500
    0.2500
    1.2500
    0.2500


Ppred = 4×4

   11.7500    4.7500         0         0
    4.7500    3.7500         0         0
         0         0   11.7500    4.7500
         0         0    4.7500    3.7500
```

Input Arguments

`filter` — Filter for object tracking

`trackingEKF` object | `trackingUKF` object

Filter for object tracking, specified as one of these objects:

- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter
- `trackingABF` — Alpha-beta filter

To use the `predict` function with a `trackingKF` linear Kalman filter, see `predict (trackingKF)`.

`dt` — Time step

positive scalar

Time step for next prediction, specified as a positive scalar in seconds.

`predparams` — Prediction parameters
comma-separated list of arguments

Prediction parameters used by the state transition function, specified as a comma-separated list of arguments. These arguments are the same arguments that are passed into the state transition function specified by the `StateTransitionFcn` property of the input `filter`.

Suppose you set the `StateTransitionFcn` property to `@constacc` and then call the `predict` function:

```
[xpred,Ppred] = predict(filter,dt)
```

The `predict` function internally calls the following:

```
state = constacc(state,dt)
```

Output Arguments

`xpred` — Predicted state of filter
vector | matrix

Predicted state of the filter, specified as a vector or matrix. The `State` property of the input `filter` is overwritten with this value.

`Ppred` — Predicted state covariance of filter
vector | matrix

Predicted state covariance of the filter, specified as a vector or matrix. The `StateCovariance` property of the input `filter` is overwritten with this value.

`zpred` — Predicted measurement
vector | matrix

Predicted measurement, specified as a vector or matrix. You can return `zpred` only when `filter` is a `trackingABF` object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`clone` | `correct` | `correctjpda` | `distance` | `initialize` | `likelihood` | `residual`

Introduced in R2017a

predict

Predict state and state estimation error covariance of linear Kalman filter

Syntax

```
[xpred,Ppred] = predict(filter)

[xpred,Ppred] = predict(filter,u)
[xpred,Ppred] = predict(filter,F)
[xpred,Ppred] = predict(filter,F,Q)
[xpred,Ppred] = predict(filter,u,F,G)
[xpred,Ppred] = predict(filter,u,F,G,Q)

[xpred,Ppred] = predict(filter,dt)
[xpred,Ppred] = predict(filter,u,dt)

predict(filter,___)
xpred = predict(filter,___)
```

Description

`[xpred,Ppred] = predict(filter)` returns the predicted state, `xpred`, and the predicted state estimation error covariance, `Ppred`, for the next time step of the input linear Kalman filter. The predicted values overwrite the internal state and state estimation error covariance of `filter`.

This syntax applies when you set the `ControlModel` property of `filter` to an empty matrix.

`[xpred,Ppred] = predict(filter,u)` specifies a control input, or force, `u`, and returns one or more of the outputs from the preceding syntaxes.

This syntax applies when you set the `ControlModel` property of `filter` to a nonempty matrix.

`[xpred,Ppred] = predict(filter,F)` specifies the state transition model, `F`. Use this syntax to change the state transition model during a simulation.

This syntax applies when you set the `ControlModel` property of `filter` to an empty matrix.

`[xpred,Ppred] = predict(filter,F,Q)` specifies the state transition model, `F`, and the process noise covariance, `Q`. Use this syntax to change the state transition model and process noise covariance during a simulation.

This syntax applies when you set the `ControlModel` property of `filter` to an empty matrix.

`[xpred,Ppred] = predict(filter,u,F,G)` specifies the force or control input, `u`, the state transition model, `F`, and the control model, `G`. Use this syntax to change the state transition model and control model during a simulation.

This syntax applies when you set the `ControlModel` property of `filter` to a nonempty matrix.

[xpred,Ppred] = predict(filter,u,F,G,Q) specifies the force or control input, u, the state transition model, F, the control model, G, and the process noise covariance, Q. Use this syntax to change the state transition model, control model, and process noise covariance during a simulation.

This syntax applies when you set the ControlModel property of filter to a nonempty matrix.

[xpred,Ppred] = predict(filter,dt) returns the predicted outputs after time step dt.

This syntax applies when the MotionModel property of filter is not set to 'Custom' and the ControlModel property is set to an empty matrix.

[xpred,Ppred] = predict(filter,u,dt) also specifies a force or control input, u.

This syntax applies when the MotionModel property of filter is not set to 'Custom' and the ControlModel property is set to a nonempty matrix.

predict(filter, ___) updates filter with the predicted state and state estimation error covariance without returning the predicted values. Specify the tracking filter and any of the input argument combinations from preceding syntaxes.

xpred = predict(filter, ___) updates filter with the predicted state and state estimation error covariance but returns only the predicted state, xpred.

Examples

Constant-Velocity Linear Kalman Filter

Create a linear Kalman filter that uses a 2D Constant Velocity motion model. Assume that the measurement consists of the object's *x*-*y* location.

Specify the initial state estimate to have zero velocity.

```
x = 5.3;
y = 3.6;
initialState = [x;0;y;0];
KF = trackingKF('MotionModel','2D Constant Velocity','State',initialState);
```

Create the measured positions from a constant-velocity trajectory.

```
vx = 0.2;
vy = 0.1;
T  = 0.5;
pos = [0:vx*T:2;5:vy*T:6]';
```

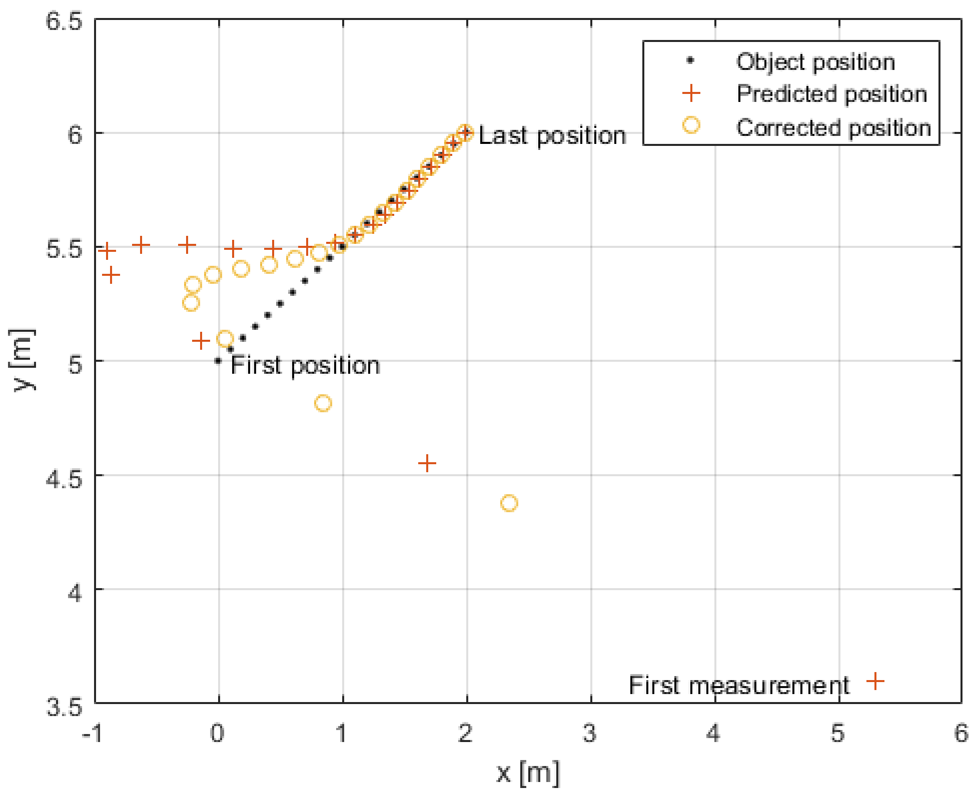
Predict and correct the state of the object.

```
for k = 1:size(pos,1)
    pstates(k,:) = predict(KF,T);
    cstates(k,:) = correct(KF,pos(k,:));
end
```

Plot the tracks.

```
plot(pos(:,1),pos(:,2),'k.', pstates(:,1),pstates(:,3),'+', ...
    cstates(:,1),cstates(:,3),'o')
```

```
xlabel('x [m]')
ylabel('y [m]')
grid
xt  = [x-2 pos(1,1)+0.1 pos(end,1)+0.1];
yt = [y pos(1,2) pos(end,2)];
text(xt,yt,{'First measurement','First position','Last position'})
legend('Object position', 'Predicted position', 'Corrected position')
```



Input Arguments

`filter` — Linear Kalman filter for object tracking
`trackingKF` object

Linear Kalman filter for object tracking, specified as a `trackingKF` object.

u — Control vector
real-valued *L*-element vector

Control vector, specified as a real-valued *L*-element vector.

F — State transition model
real-valued *M*-by-*M* matrix

State transition model, specified as a real-valued *M*-by-*M* matrix, where *M* is the size of the state vector.

`Q` — Process noise covariance matrix
positive-definite, real-valued *M*-by-*M* matrix

Process noise covariance matrix, specified as a positive-definite, real-valued *M*-by-*M* matrix, where *M* is the length of the state vector.

`G` — Control model
real-valued *M*-by-*L* matrix

Control model, specified as a real-valued *M*-by-*L* matrix. *M* is the size of the state vector. *L* is the number of independent controls.

`dt` — Time step
positive scalar

Time step, specified as a positive scalar. Units are in seconds.

Output Arguments

`xpred` — Predicted state
real-valued *M*-element vector

Predicted state, returned as a real-valued *M*-element vector. The predicted state represents the deducible estimate of the state vector, propagated from the previous state using the state transition and control models.

`Ppred` — Predicted state error covariance matrix
real-valued *M*-by-*M* matrix

Predicted state covariance matrix, specified as a real-valued *M*-by-*M* matrix. *M* is the size of the state vector. The predicted state covariance matrix represents the *deducible* estimate of the covariance matrix vector. The filter propagates the covariance matrix from the previous estimate.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`clone` | `correct` | `correctjpda` | `distance` | `initialize` | `likelihood` | `residual`

Introduced in R2017a

residual

Measurement residual and residual noise from tracking filter

Syntax

```
[zres,rescov] = residual(filter,zmeas)
[zres,rescov] = residual(filter,zmeas,measparams)
```

Description

`[zres,rescov] = residual(filter,zmeas)` computes the residual and residual covariance of the current given measurement, `zmeas`, with the predicted measurement in the tracking filter, `filter`. This function applies to filters that assume a Gaussian distribution for noise.

`[zres,rescov] = residual(filter,zmeas,measparams)` specifies additional parameters that are used by the `MeasurementFcn` of the filter.

If `filter` is a `trackingKF` object, then you cannot use this syntax.

Input Arguments

`filter` — Filter for object tracking
`trackingKF` object | `trackingEKF` object | `trackingUKF` object

Filter for object tracking, specified as one of these objects:

- `trackingKF` — Linear Kalman filter
- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter

`zmeas` — Current measurement of tracked object
vector | matrix

Current measurement of a tracked object, specified as a vector or matrix.

`measparams` — Parameters for measurement function
cell array

Parameters for measurement function, specified as a cell array. The parameters are passed to the measurement function that is defined in the `MeasurementFcn` property of the input `filter`. If `filter` is a `trackingKF` object, then you cannot specify `measparams`.

Output Arguments

`zres` — Residual between current and predicted measurement
matrix

Residual between current and predicted measurement, returned as a matrix.

rescov — Residual covariance
matrix

Residual covariance, returned as a matrix.

Algorithms

The residual is the difference between a measurement and the value predicted by the filter. For Kalman filters, the residual calculation depends on whether the filter is linear or nonlinear.

Linear Kalman Filters

Given a linear Kalman filter with a current measurement of z, the residual z_{res} is defined as

$$z_{res} = z - Hx,$$

where:

- H is the measurement model set by the `MeasurementModel` property of the filter.
- x is the current filter state.

The covariance of the residual, S, is defined as

$$S = R + HPH^T,$$

where:

- P is the state covariance matrix.
- R is the measurement noise matrix set by the `MeasurementNoise` property of the filter.

Nonlinear Kalman Filters

Given a nonlinear Kalman filter with a current measurement of z, the residual z_{res} is defined as:

$$z_{res} = z - h(x),$$

where:

- h is the measurement function set by the `MeasurementFcn` property.
- x is the current filter state.

The covariance of the residual, S, is defined as:

$$S = R + R_p,$$

where:

- R is the measurement noise matrix set by the `MeasurementNoise` property of the filter.
- R_p is the state covariance matrix projected onto the measurement space.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`clone` | `correct` | `correctjpda` | `distance` | `initialize` | `likelihood` | `predict`

Introduced in R2018a

quaternion

Create a quaternion array

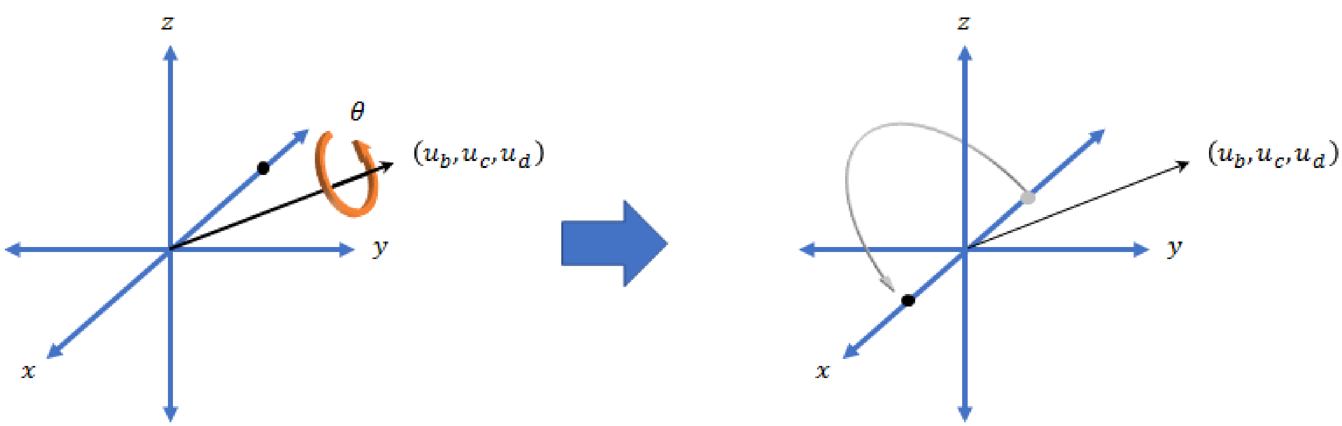
Description

A quaternion is a four-part hyper-complex number used in three-dimensional rotations and orientations.

A quaternion number is represented in the form $a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, where a, b, c, and d parts are real numbers, and i, j, and k are the basis elements, satisfying the equation: $\mathrm{i}^2 = \mathrm{j}^2 = \mathrm{k}^2 = \mathrm{ijk} = -1$.

The set of quaternions, denoted by **H**, is defined within a four-dimensional vector space over the real numbers, $\mathbf{R}^4$. Every element of **H** has a unique representation based on a linear combination of the basis elements, i, j, and k.

All rotations in 3-D can be described by an axis of rotation and angle about that axis. An advantage of quaternions over rotation matrices is that the axis and angle of rotation is easy to interpret. For example, consider a point in $\mathbf{R}^3$. To rotate the point, you define an axis of rotation and an angle of rotation.



The quaternion representation of the rotation may be expressed as $q = \cos\left(\theta/2\right) + \sin\left(\theta/2\right)(u_b\mathrm{i} + u_c\mathrm{j} + u_d\mathrm{k})$, where θ is the angle of rotation and $[u_b,\ u_c,\ \text{and}\ u_d]$ is the axis of rotation.

Creation

Syntax

```
quat = quaternion()
quat = quaternion(A,B,C,D)
quat = quaternion(matrix)
quat = quaternion(RV,'rotvec')
```

```
quat = quaternion(RV,'rotvecd')
quat = quaternion(RM,'rotmat',PF)
quat = quaternion(E,'euler',RS,PF)
quat = quaternion(E,'eulerd',RS,PF)
```

Description

`quat = quaternion()` creates an empty quaternion.

`quat = quaternion(A,B,C,D)` creates a quaternion array where the four quaternion parts are taken from the arrays `A`, `B`, `C`, and `D`. All the inputs must have the same size and be of the same data type.

`quat = quaternion(matrix)` creates an *N*-by-1 quaternion array from an *N*-by-4 matrix, where each column becomes one part of the quaternion.

`quat = quaternion(RV,'rotvec')` creates an *N*-by-1 quaternion array from an *N*-by-3 matrix of rotation vectors, `RV`. Each row of `RV` represents a rotation vector in radians.

`quat = quaternion(RV,'rotvecd')` creates an *N*-by-1 quaternion array from an *N*-by-3 matrix of rotation vectors, `RV`. Each row of `RV` represents a rotation vector in degrees.

`quat = quaternion(RM,'rotmat',PF)` creates an *N*-by-1 quaternion array from the 3-by-3-by-*N* array of rotation matrices, `RM`. `PF` can be either `'point'` if the Euler angles represent point rotations or `'frame'` for frame rotations.

`quat = quaternion(E,'euler',RS,PF)` creates an *N*-by-1 quaternion array from the *N*-by-3 matrix, `E`. Each row of `E` represents a set of Euler angles in radians. The angles in `E` are rotations about the axes in sequence `RS`.

`quat = quaternion(E,'eulerd',RS,PF)` creates an *N*-by-1 quaternion array from the *N*-by-3 matrix, `E`. Each row of `E` represents a set of Euler angles in degrees. The angles in `E` are rotations about the axes in sequence `RS`.

Input Arguments

`A,B,C,D` — Quaternion parts

comma-separated arrays of the same size

Parts of a quaternion, specified as four comma-separated scalars, matrices, or multi-dimensional arrays of the same size.

Example: `quat = quaternion(1,2,3,4)` creates a quaternion of the form 1 + 2i + 3j + 4k.

Example: `quat = quaternion([1,5],[2,6],[3,7],[4,8])` creates a 1-by-2 quaternion array where `quat(1,1) = 1 + 2i + 3j + 4k` and `quat(1,2) = 5 + 6i + 7j + 8k`

Data Types: `single` | `double`

`matrix` — Matrix of quaternion parts

N-by-4 matrix

Matrix of quaternion parts, specified as an *N*-by-4 matrix. Each row represents a separate quaternion. Each column represents a separate quaternion part.

Example: `quat = quaternion(rand(10,4))` creates a 10-by-1 quaternion array.

Data Types: `single` | `double`

RV — Matrix of rotation vectors
N-by-3 matrix

Matrix of rotation vectors, specified as an *N*-by-3 matrix. Each row of `RV` represents the [X Y Z] elements of a rotation vector. A rotation vector is a unit vector representing the axis of rotation scaled by the angle of rotation in radians or degrees.

To use this syntax, specify the first argument as a matrix of rotation vectors and the second argument as the `'rotvec'` or `'rotvecd'`.

Example: `quat = quaternion(rand(10,3),'rotvec')` creates a 10-by-1 quaternion array.

Data Types: `single` | `double`

RM — Rotation matrices
3-by-3 matrix | 3-by-3-by-*N* array

Array of rotation matrices, specified by a 3-by-3 matrix or 3-by-3-by-*N* array. Each page of the array represents a separate rotation matrix.

Example: `quat = quaternion(rand(3),'rotmat','point')`

Example: `quat = quaternion(rand(3),'rotmat','frame')`

Data Types: `single` | `double`

PF — Type of rotation matrix
`'point'` | `'frame'`

Type of rotation matrix, specified by `'point'` or `'frame'`.

Example: `quat = quaternion(rand(3),'rotmat','point')`

Example: `quat = quaternion(rand(3),'rotmat','frame')`

Data Types: `char` | `string`

E — Matrix of Euler angles
N-by-3 matrix

Matrix of Euler angles, specified by an *N*-by-3 matrix. If using the `'euler'` syntax, specify `E` in radians. If using the `'eulerd'` syntax, specify `E` in degrees.

Example: `quat = quaternion(E,'euler','YZY','point')`

Example: `quat = quaternion(E,'euler','XYZ','frame')`

Data Types: `single` | `double`

RS — Rotation sequence
character vector | scalar string

Rotation sequence, specified as a three-element character vector:

- `'YZY'`
- `'YXY'`
- `'ZYZ'`

- `'ZXZ'`
- `'XYX'`
- `'XZX'`
- `'XYZ'`
- `'YZX'`
- `'ZXY'`
- `'XZY'`
- `'ZYX'`
- `'YXZ'`

Assume you want to determine the new coordinates of a point when its coordinate system is rotated using frame rotation. The point is defined in the original coordinate system as:

```
point = [sqrt(2)/2,sqrt(2)/2,0];
```

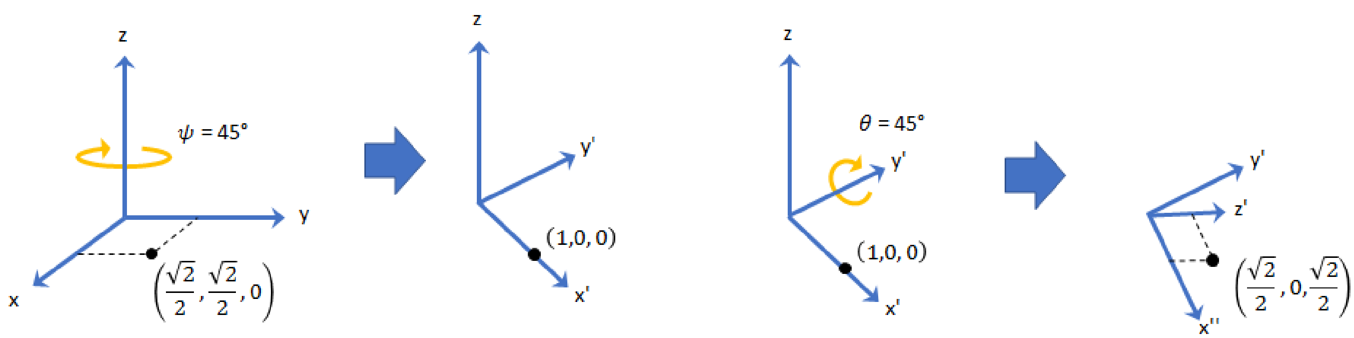
In this representation, the first column represents the *x*-axis, the second column represents the *y*-axis, and the third column represents the *z*-axis.

You want to rotate the point using the Euler angle representation [45,45,0]. Rotate the point using two different rotation sequences:

- If you create a quaternion rotator and specify the 'ZYX' sequence, the frame is first rotated 45° around the *z*-axis, then 45° around the new *y*-axis.

  ```
  quatRotator = quaternion([45,45,0],'eulerd','ZYX','frame');
  newPointCoordinate = rotateframe(quatRotator,point)
  ```

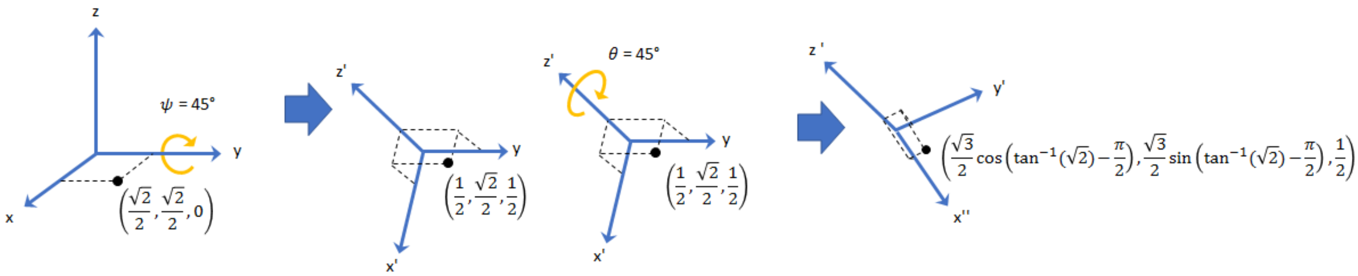
  ```
  newPointCoordinate =

      0.7071   -0.0000    0.7071
  ```



- If you create a quaternion rotator and specify the 'YZX' sequence, the frame is first rotated 45° around the *y*-axis, then 45° around the new *z*-axis.

  ```
  quatRotator = quaternion([45,45,0],'eulerd','YZX','frame');
  newPointCoordinate = rotateframe(quatRotator,point)
  ```

  ```
  newPointCoordinate =

      0.8536    0.1464    0.5000
  ```



Data Types: `char` | `string`

Object Functions

angvel	Angular velocity from quaternion array
classUnderlying	Class of parts within quaternion
compact	Convert quaternion array to N-by-4 matrix
conj	Complex conjugate of quaternion
ctranspose, '	Complex conjugate transpose of quaternion array
dist	Angular distance in radians
euler	Convert quaternion to Euler angles (radians)
eulerd	Convert quaternion to Euler angles (degrees)
exp	Exponential of quaternion array
ldivide, .\	Element-wise quaternion left division
log	Natural logarithm of quaternion array
meanrot	Quaternion mean rotation
minus, -	Quaternion subtraction
mtimes, *	Quaternion multiplication
norm	Quaternion norm
normalize	Quaternion normalization
ones	Create quaternion array with real parts set to one and imaginary parts set to zero
parts	Extract quaternion parts
power, .^	Element-wise quaternion power
prod	Product of a quaternion array
randrot	Uniformly distributed random rotations
rdivide, ./	Element-wise quaternion right division
rotateframe	Quaternion frame rotation
rotatepoint	Quaternion point rotation
rotmat	Convert quaternion to rotation matrix
rotvec	Convert quaternion to rotation vector (radians)
rotvecd	Convert quaternion to rotation vector (degrees)
slerp	Spherical linear interpolation
times, .*	Element-wise quaternion multiplication
transpose, .'	Transpose a quaternion array
uminus, -	Quaternion unary minus
zeros	Create quaternion array with all parts set to zero

Examples

Create Empty Quaternion

```
quat = quaternion()
```

```
quat =

  0x0 empty quaternion array
```

By default, the underlying class of the quaternion is a double.

```
classUnderlying(quat)
```

```
ans =
'double'
```

Create Quaternion by Specifying Individual Quaternion Parts

You can create a quaternion array by specifying the four parts as comma-separated scalars, matrices, or multidimensional arrays of the same size.

Define quaternion parts as scalars.

```
A = 1.1;
B = 2.1;
C = 3.1;
D = 4.1;
quatScalar = quaternion(A,B,C,D)
```

```
quatScalar = quaternion
     1.1 + 2.1i + 3.1j + 4.1k
```

Define quaternion parts as column vectors.

```
A = [1.1;1.2];
B = [2.1;2.2];
C = [3.1;3.2];
D = [4.1;4.2];
quatVector = quaternion(A,B,C,D)
```

```
quatVector=2×1 quaternion array
     1.1 + 2.1i + 3.1j + 4.1k
     1.2 + 2.2i + 3.2j + 4.2k
```

Define quaternion parts as matrices.

```
A = [1.1,1.3; ...
     1.2,1.4];
B = [2.1,2.3; ...
     2.2,2.4];
C = [3.1,3.3; ...
     3.2,3.4];
D = [4.1,4.3; ...
     4.2,4.4];
quatMatrix = quaternion(A,B,C,D)
```

```
quatMatrix=2×2 quaternion array
     1.1 + 2.1i + 3.1j + 4.1k     1.3 + 2.3i + 3.3j + 4.3k
     1.2 + 2.2i + 3.2j + 4.2k     1.4 + 2.4i + 3.4j + 4.4k
```

Define quaternion parts as three dimensional arrays.

```
A = randn(2,2,2);
B = zeros(2,2,2);
C = zeros(2,2,2);
D = zeros(2,2,2);
quatMultiDimArray = quaternion(A,B,C,D)
```

```
quatMultiDimArray = 2x2x2 quaternion array
quatMultiDimArray(:,:,1) =

     0.53767 +       0i +       0j +       0k     -2.2588 +       0i +       0j +       0k
      1.8339 +       0i +       0j +       0k     0.86217 +       0i +       0j +       0k


quatMultiDimArray(:,:,2) =

     0.31877 +       0i +       0j +       0k    -0.43359 +       0i +       0j +       0k
     -1.3077 +       0i +       0j +       0k     0.34262 +       0i +       0j +       0k
```

Create Quaternion by Specifying Quaternion Parts Matrix

You can create a scalar or column vector of quaternions by specify an *N*-by-4 matrix of quaternion parts, where columns correspond to the quaternion parts A, B, C, and D.

Create a column vector of random quaternions.

```
quatParts = rand(3,4)
```

```
quatParts = 3×4

    0.8147    0.9134    0.2785    0.9649
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706
```

```
quat = quaternion(quatParts)
```

```
quat=3×1 quaternion array
     0.81472 + 0.91338i +  0.2785j + 0.96489k
     0.90579 + 0.63236i + 0.54688j + 0.15761k
     0.12699 + 0.09754i + 0.95751j + 0.97059k
```

To retrieve the `quatParts` matrix from quaternion representation, use `compact`.

```
retrievedquatParts = compact(quat)
```

```
retrievedquatParts = 3×4

    0.8147    0.9134    0.2785    0.9649
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706
```

Create Quaternion by Specifying Rotation Vectors

You can create an *N*-by-1 quaternion array by specifying an *N*-by-3 matrix of rotation vectors in radians or degrees. Rotation vectors are compact spatial representations that have a one-to-one relationship with normalized quaternions.

Rotation Vectors in Radians

Create a scalar quaternion using a rotation vector and verify the resulting quaternion is normalized.

```
rotationVector = [0.3491,0.6283,0.3491];
quat = quaternion(rotationVector,'rotvec')
```

```
quat = quaternion
     0.92124 + 0.16994i + 0.30586j + 0.16994k
```

```
norm(quat)
```

```
ans = 1.0000
```

You can convert from quaternions to rotation vectors in radians using the `rotvec` function. Recover the `rotationVector` from the quaternion, `quat`.

```
rotvec(quat)
```

```
ans = 1×3

    0.3491    0.6283    0.3491
```

Rotation Vectors in Degrees

Create a scalar quaternion using a rotation vector and verify the resulting quaternion is normalized.

```
rotationVector = [20,36,20];
quat = quaternion(rotationVector,'rotvecd')
```

```
quat = quaternion
     0.92125 + 0.16993i + 0.30587j + 0.16993k
```

```
norm(quat)
```

```
ans = 1
```

You can convert from quaternions to rotation vectors in degrees using the `rotvecd` function. Recover the `rotationVector` from the quaternion, `quat`.

```
rotvecd(quat)
```

```
ans = 1×3

   20.0000   36.0000   20.0000
```

Create Quaternion by Specifying Rotation Matrices

You can create an N-by-1 quaternion array by specifying a 3-by-3-by-N array of rotation matrices. Each page of the rotation matrix array corresponds to one element of the quaternion array.

Create a scalar quaternion using a 3-by-3 rotation matrix. Specify whether the rotation matrix should be interpreted as a frame or point rotation.

```
rotationMatrix = [1 0         0; ...
                  0 sqrt(3)/2 0.5; ...
                  0 -0.5      sqrt(3)/2];
quat = quaternion(rotationMatrix,'rotmat','frame')
```

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

You can convert from quaternions to rotation matrices using the `rotmat` function. Recover the `rotationMatrix` from the quaternion, `quat`.

```
rotmat(quat,'frame')
```

```
ans = 3×3

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

Create Quaternion by Specifying Euler Angles

You can create an *N*-by-1 quaternion array by specifying an *N*-by-3 array of Euler angles in radians or degrees.

Euler Angles in Radians

Use the `euler` syntax to create a scalar quaternion using a 1-by-3 vector of Euler angles in radians. Specify the rotation sequence of the Euler angles and whether the angles represent a frame or point rotation.

```
E = [pi/2,0,pi/4];
quat = quaternion(E,'euler','ZYX','frame')
```

```
quat = quaternion
     0.65328 +  0.2706i +  0.2706j + 0.65328k
```

You can convert from quaternions to Euler angles using the `euler` function. Recover the Euler angles, `E`, from the quaternion, `quat`.

```
euler(quat,'ZYX','frame')
```

```
ans = 1×3

    1.5708         0    0.7854
```

Euler Angles in Degrees

Use the `eulerd` syntax to create a scalar quaternion using a 1-by-3 vector of Euler angles in degrees. Specify the rotation sequence of the Euler angles and whether the angles represent a frame or point rotation.

```
E = [90,0,45];
quat = quaternion(E,'eulerd','ZYX','frame')
```

```
quat = quaternion
     0.65328 +  0.2706i +  0.2706j + 0.65328k
```

You can convert from quaternions to Euler angles in degrees using the `eulerd` function. Recover the Euler angles, `E`, from the quaternion, `quat`.

```
eulerd(quat,'ZYX','frame')
```

```
ans = 1×3

   90.0000         0   45.0000
```

Quaternion Algebra

Quaternions form a noncommutative associative algebra over the real numbers. This example illustrates the rules of quaternion algebra.

Addition and Subtraction

Quaternion addition and subtraction occur part-by-part, and are commutative:

```
Q1 = quaternion(1,2,3,4)
```

```
Q1 = quaternion
     1 + 2i + 3j + 4k
```

```
Q2 = quaternion(9,8,7,6)
```

```
Q2 = quaternion
     9 + 8i + 7j + 6k
```

```
Q1plusQ2 = Q1 + Q2
```

```
Q1plusQ2 = quaternion
     10 + 10i + 10j + 10k
```

```
Q2plusQ1 = Q2 + Q1
```

```
Q2plusQ1 = quaternion
     10 + 10i + 10j + 10k
```

```
Q1minusQ2 = Q1 - Q2
```

```
Q1minusQ2 = quaternion
    -8 - 6i - 4j - 2k
```

```
Q2minusQ1 = Q2 - Q1
```

```
Q2minusQ1 = quaternion
     8 + 6i + 4j + 2k
```

You can also perform addition and subtraction of real numbers and quaternions. The first part of a quaternion is referred to as the *real* part, while the second, third, and fourth parts are referred to as the *vector*. Addition and subtraction with real numbers affect only the real part of the quaternion.

```
Q1plusRealNumber = Q1 + 5
```

```
Q1plusRealNumber = quaternion
     6 + 2i + 3j + 4k
```

```
Q1minusRealNumber = Q1 - 5
```

```
Q1minusRealNumber = quaternion
    -4 + 2i + 3j + 4k
```

Multiplication

Quaternion multiplication is determined by the products of the basis elements and the distributive law. Recall that multiplication of the basis elements, *i*, *j*, and *k*, are not commutative, and therefore quaternion multiplication is not commutative.

```
Q1timesQ2 = Q1 * Q2
```

```
Q1timesQ2 = quaternion
    -52 + 16i + 54j + 32k
```

```
Q2timesQ1 = Q2 * Q1
```

```
Q2timesQ1 = quaternion
    -52 + 36i + 14j + 52k
```

```
isequal(Q1timesQ2,Q2timesQ1)
```

```
ans = logical
   0
```

You can also multiply a quaternion by a real number. If you multiply a quaternion by a real number, each part of the quaternion is multiplied by the real number individually:

```
Q1times5 = Q1*5
```

```
Q1times5 = quaternion
      5 + 10i + 15j + 20k
```

Multiplying a quaternion by a real number is commutative.

```
isequal(Q1*5,5*Q1)
```

```
ans = logical
   1
```

Conjugation

The complex conjugate of a quaternion is defined such that each element of the vector portion of the quaternion is negated.

```
Q1
```

```
Q1 = quaternion
     1 + 2i + 3j + 4k
```

```
conj(Q1)
```

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

Multiplication between a quaternion and its conjugate is commutative:

```
isequal(Q1*conj(Q1),conj(Q1)*Q1)
```

```
ans = logical
   1
```

Quaternion Array Manipulation

You can organize quaternions into vectors, matrices, and multidimensional arrays. Built-in MATLAB® functions have been enhanced to work with quaternions.

Concatenate

Quaternions are treated as individual objects during concatenation and follow MATLAB rules for array manipulation.

```
Q1 = quaternion(1,2,3,4);
Q2 = quaternion(9,8,7,6);

qVector = [Q1,Q2]
```

```
qVector=1×2 quaternion array
     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
```

```
Q3 = quaternion(-1,-2,-3,-4);
Q4 = quaternion(-9,-8,-7,-6);

qMatrix = [qVector;Q3,Q4]
```

```
qMatrix=2×2 quaternion array
     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
```

```
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k
```

```
qMultiDimensionalArray(:,:,1) = qMatrix;
qMultiDimensionalArray(:,:,2) = qMatrix
```

```
qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) = 

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k


qMultiDimensionalArray(:,:,2) = 

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k
```

Indexing

To access or assign elements in a quaternion array, use indexing.

```
qLoc2 = qMultiDimensionalArray(2)
```

```
qLoc2 = quaternion
    -1 - 2i - 3j - 4k
```

Replace the quaternion at index two with a quaternion one.

```
qMultiDimensionalArray(2) = ones('quaternion')
```

```
qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) = 

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
     1 + 0i + 0j + 0k    -9 - 8i - 7j - 6k


qMultiDimensionalArray(:,:,2) = 

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k
```

Reshape

To reshape quaternion arrays, use the `reshape` function.

```
qMatReshaped = reshape(qMatrix,4,1)
```

```
qMatReshaped=4×1 quaternion array
     1 + 2i + 3j + 4k
    -1 - 2i - 3j - 4k
     9 + 8i + 7j + 6k
    -9 - 8i - 7j - 6k
```

Transpose

To transpose quaternion vectors and matrices, use the `transpose` function.

```
qMatTransposed = transpose(qMatrix)
```

```
qMatTransposed=2×2 quaternion array
     1 + 2i + 3j + 4k    -1 - 2i - 3j - 4k
     9 + 8i + 7j + 6k    -9 - 8i - 7j - 6k
```

Permute

To permute quaternion vectors, matrices, and multidimensional arrays, use the `permute` function.

```
qMultiDimensionalArray
```

```
qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
     1 + 0i + 0j + 0k    -9 - 8i - 7j - 6k


qMultiDimensionalArray(:,:,2) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k
```

```
qMatPermute = permute(qMultiDimensionalArray,[3,1,2])
```

```
qMatPermute = 2x2x2 quaternion array
qMatPermute(:,:,1) =

     1 + 2i + 3j + 4k     1 + 0i + 0j + 0k
     1 + 2i + 3j + 4k    -1 - 2i - 3j - 4k


qMatPermute(:,:,2) =

     9 + 8i + 7j + 6k    -9 - 8i - 7j - 6k
     9 + 8i + 7j + 6k    -9 - 8i - 7j - 6k
```

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Topics
“Rotations, Orientations, and Quaternions for Automated Driving”

Introduced in R2020a

objectTrack

Single object track report

Description

`objectTrack` captures the track information of a single object. `objectTrack` is the standard output format for trackers.

Creation

Syntax

```
track = objectTrack
track = objectTrack(Name,Value)
```

Description

`track = objectTrack` creates an `objectTrack` object with default property values. An `objectTrack` object contains information like the age and state of a single track.

`track = objectTrack(Name,Value)` allows you to set properties using one or more name-value pairs. Enclose each property name in single quotes.

Properties

`TrackID` — Unique track identifier
`1` (default) | nonnegative integer

Unique track identifier, specified as a nonnegative integer. This property distinguishes different tracks.

Example: 2

`BranchID` — Unique track branch identifier
`0` (default) | nonnegative integer

Unique track branch identifier, specified as a nonnegative integer. This property distinguishes different track branches.

Example: 1

`SourceIndex` — Index of source track reporting system
`1` (default) | nonnegative integer

Index of source track reporting system, specified as a nonnegative integer. This property identifies the source that reports the track.

Example: 3

ObjectClassID — Object class identifier
0 (default) | nonnegative integer

Object class identifier, specified as a nonnegative integer. This property distinguishes between different user-defined types of objects. For example, you can use 1 for objects of type "car", and 2 for objects of type "pedestrian". 0 is reserved for unknown classification.

Example: 3

UpdateTime — Update time of track
0 (default) | nonnegative real scalar

Time at which the track was updated by a tracker, specified as a nonnegative real scalar.

Example: 1.2

Data Types: single | double

Age — Number of times track was updated
1 (default) | positive integer

Number of times the track was updated, specified as a positive integer. When a track is initialized, its Age is equal to 1. Any subsequent update with a hit or miss increases the track Age by 1.

Example: 2

State — Current state of track
zeros(6,1) (default) | real-valued *N*-element vector

The current state of the track at the UpdateTime, specified as a real-valued *N*-element vector, where *N* is the dimension of the state. The format of track state depends on the model used to track the object. For example, for 3-D constant velocity model used with constvel, the state vector is $[x; v_x; y; v_y; z; v_z]$.

Example: [1 0.2 3 0.2]

Data Types: single | double

StateCovariance — Current state uncertainty covariance of track
eye(6,6) (default) | real positive semidefinite symmetric *N*-by-*N* matrix

The current state uncertainty covariance of the track, specified as a real positive semidefinite symmetric *N*-by-*N* matrix, where *N* is the dimension of state specified in the State property.

Data Types: single | double

TrackLogic — Track confirmation and deletion logic type
'History' (default) | 'Integrated' | 'Score'

Confirmation and deletion logic type, specified as:

- 'History' – Track confirmation and deletion is based on the number of times the track has been assigned to a detection in the latest tracker updates.
- 'Score' – Track confirmation and deletion is based on a log-likelihood track score. A high score means that the track is more likely to be valid. A low score means that the track is more likely to be a false alarm.
- 'Integrated' – Track confirmation and deletion is based on the integrated probability of track existence.

`TrackLogicState` — State of track logic
1-by-*M* logical vector | 1-by-2 real-valued scar | nonnegative scalar

The current state of the track logic type. Based on the logic type specified in the `TrackLogic` property, the logic state is specified as:

- `'History'` – A 1-by-*M* logical vector, where *M* is the number of latest track logical states recorded. `true` (1) values indicate hits, and `false` (0) values indicate misses. For example, `[1 0 1 1 1]` represents four hits and one miss in the last five updates. The default value for logic state is 1.
- `'Score'` – A 1-by-2 real-valued vector, [*cs*, *ms*]. *cs* is the current score, and *ms* is the maximum score. The default value is `[0, 0]`.
- `'Integrated'` – A nonnegative scalar. The scalar represents the integrated probability of existence of the track. The default value is 0.5.

`IsConfirmed` — Indicate if track is confirmed
`true` (default) | `false`

Indicate if the track is confirmed, specified as `true` or `false`.

Data Types: `logical`

`IsCoasted` — Indicate if track is coasted
`false` (default) | `true`

Indicate if the track is coasted, specified as `true` or `false`. A track is coasted if its latest update is based on prediction instead of correction using detections.

Data Types: `logical`

`IsSelfReported` — Indicate if track is self reported
`true` (default) | `false`

Indicate if the track is self reported, specified as `true` or `false`. A track is self reported if it is reported from internal sources (senors, trackers, or fusers). To limit the propagation of rumors in a tracking system, use the value `false` if the track was updated by an external source.

Example: `false`

Data Types: `logical`

`ObjectAttributes` — Object attributes
`struct()` (default) | structure

Object attributes passed by the tracker, specified as a structure.

`StateParameters` — Parameters of the track state reference frame
`struct()` (default) | structure | structure array

Parameters of the track state reference frame, specified as a structure or a structure array. Use this property to define the track state reference frame and how to transform the track from the source coordinate system to the fuser coordinate system.

Object Functions

toStruct   Convert objectTrack object to struct

Examples

Create Track Report using `objectTrack`

Create a report of a track using `objectTrack`.

```
x = (1:6)';
P = diag(1:6);
track = objectTrack('State',x,'StateCovariance',P);
disp(track)
```

```
  objectTrack with properties:

             TrackID: 1
            BranchID: 0
         SourceIndex: 1
          UpdateTime: 0
                 Age: 1
               State: [6x1 double]
     StateCovariance: [6x6 double]
     StateParameters: [1x1 struct]
       ObjectClassID: 0
          TrackLogic: 'History'
     TrackLogicState: 1
         IsConfirmed: 1
           IsCoasted: 0
      IsSelfReported: 1
    ObjectAttributes: [1x1 struct]
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

- The `TrackLogic` property can only be set during construction.

See Also

Introduced in R2020a

toStruct

Convert `objectTrack` object to `struct`

Syntax

`S = toStruct(objTrack)`

Description

`S = toStruct(objTrack)` converts an array of `objectTrack` objects, `objTrack`, to an array of structures whose fields are equivalent to the properties of `objTrack`.

Examples

Convert `objectTrack` to Struct

Create a report of a track using `objectTrack`.

```
x = (1:6)';
P = diag(1:6);
track = objectTrack('State', x, 'StateCovariance', P)
```

```
track = 
  objectTrack with properties:

              TrackID: 1
             BranchID: 0
          SourceIndex: 1
           UpdateTime: 0
                  Age: 1
                State: [6x1 double]
      StateCovariance: [6x6 double]
      StateParameters: [1x1 struct]
        ObjectClassID: 0
           TrackLogic: 'History'
      TrackLogicState: 1
          IsConfirmed: 1
            IsCoasted: 0
       IsSelfReported: 1
     ObjectAttributes: [1x1 struct]
```

Convert the track object to a structure.

```
S = toStruct(track)
```

```
S = struct with fields:
              TrackID: 1
             BranchID: 0
          SourceIndex: 1
           UpdateTime: 0
```

```
                 Age: 1
               State: [6x1 double]
     StateCovariance: [6x6 double]
     StateParameters: [1x1 struct]
       ObjectClassID: 0
          TrackLogic: 'History'
     TrackLogicState: 1
         IsConfirmed: 1
           IsCoasted: 0
      IsSelfReported: 1
    ObjectAttributes: [1x1 struct]
```

Input Arguments

objTrack — Reports of object track
array of objectTrack object

Reports of object tracks, specified as an array of objectTrack objects.

Output Arguments

S — Structures converted from objectTrack
array of structure

Structures converted from objectTrack, returned as an array of structures. The dimension of the returned structure is same with the dimension of the objTrack input. The fields of each structure are equivalent to the properties of objectTrack.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
objectTrack

Introduced in R2020a

classUnderlying

Class of parts within quaternion

Syntax

```
underlyingClass = classUnderlying(quat)
```

Description

`underlyingClass = classUnderlying(quat)` returns the name of the class of the parts of the quaternion `quat`.

Examples

Get Underlying Class of Quaternion

A quaternion is a four-part hyper-complex number used in three-dimensional representations. The four parts of the quaternion are of data type `single` or `double`.

Create two quaternions, one with an underlying data type of `single`, and one with an underlying data type of `double`. Verify the underlying data types by calling `classUnderlying` on the quaternions.

```
qSingle = quaternion(single([1,2,3,4]))
```

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

```
classUnderlying(qSingle)
```

```
ans =
'single'
```

```
qDouble = quaternion([1,2,3,4])
```

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

```
classUnderlying(qDouble)
```

```
ans =
'double'
```

You can separate quaternions into their parts using the `parts` function. Verify the parts of each quaternion are the correct data type. Recall that `double` is the default MATLAB® type.

```
[aS,bS,cS,dS] = parts(qSingle)
```

```
aS = single
    1
```

```
bS = single
    2

cS = single
    3

dS = single
    4
```

```
[aD,bD,cD,dD] = parts(qDouble)
```

```
aD = 1

bD = 2

cD = 3

dD = 4
```

Quaternions follow the same implicit casting rules as other data types in MATLAB. That is, a quaternion with underlying data type `single` that is combined with a quaternion with underlying data type `double` results in a quaternion with underlying data type `single`. Multiply `qDouble` and `qSingle` and verify the resulting underlying data type is `single`.

```
q = qDouble*qSingle;
classUnderlying(q)
```

```
ans =
'single'
```

Input Arguments

`quat` — Quaternion to investigate
scalar | vector | matrix | multi-dimensional array

Quaternion to investigate, specified as a quaternion or array of quaternions.

Data Types: `quaternion`

Output Arguments

`underlyingClass` — Underlying class of quaternion object
`'single'` | `'double'`

Underlying class of quaternion, returned as the character vector `'single'` or `'double'`.

Data Types: `char`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`compact` | `parts`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

angvel

Angular velocity from quaternion array

Syntax

```
AV = angvel(Q,dt,'frame')
AV = angvel(Q,dt,'point')
[AV,qf] = angvel(Q,dt,fp,qi)
```

Description

`AV = angvel(Q,dt,'frame')` returns the angular velocity array from an array of quaternions, `Q`. The quaternions in `Q` correspond to frame rotation. The initial quaternion is assumed to represent zero rotation.

`AV = angvel(Q,dt,'point')` returns the angular velocity array from an array of quaternions, `Q`. The quaternions in `Q` correspond to point rotation. The initial quaternion is assumed to represent zero rotation.

`[AV,qf] = angvel(Q,dt,fp,qi)` allows you to specify the initial quaternion, `qi`, and the type of rotation, `fp`. It also returns the final quaternion, `qf`.

Examples

Generate Angular Velocity From Quaternion Array

Create an array of quaternions.

```
eulerAngles = [(0:10:90).',zeros(numel(0:10:90),2)];
q = quaternion(eulerAngles,'eulerd','ZYX','frame');
```

Specify the time step and generate the angular velocity array.

```
dt = 1;
av = angvel(q,dt,'frame') % units in rad/s
```

```
av = 10×3

         0         0         0
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
```

Input Arguments

`Q` — Quaternions
N-by-1 vector of quaternions

Quaternions, specified as an *N*-by-1 vector of quaternions.

Data Types: `quaternion`

`dt` — Time step
nonnegative scalar

Time step, specified as a nonnegative scalar.

Data Types: `single` | `double`

`fp` — Type of rotation
`'frame'` | `'point'`

Type of rotation, specified as `'frame'` or `'point'`.

`qi` — Initial quaternion
quaternion

Initial quaternion, specified as a quaternion.

Data Types: `quaternion`

Output Arguments

`AV` — Angular velocity
N-by-3 real matrix

Angular velocity, returned as an *N*-by-3 real matrix. *N* is the number of quaternions given in the input *Q*. Each row of the matrix corresponds to an angular velocity vector.

`qf` — Final quaternion
quaternion

Final quaternion, returned as a quaternion. `qf` is the same as the last quaternion in the *Q* input.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

compact

Convert quaternion array to *N*-by-4 matrix

Syntax

```
matrix = compact(quat)
```

Description

`matrix = compact(quat)` converts the quaternion array, `quat`, to an *N*-by-4 matrix. The columns are made from the four quaternion parts. The i^{th} row of the matrix corresponds to `quat(i)`.

Examples

Convert Quaternion Array to Compact Representation of Parts

Create a scalar quaternion with random parts. Convert the parts to a 1-by-4 vector using `compact`.

```
randomParts = randn(1,4)
```

```
randomParts = 1×4

    0.5377    1.8339   -2.2588    0.8622
```

```
quat = quaternion(randomParts)
```

```
quat = quaternion
     0.53767 +  1.8339i -  2.2588j + 0.86217k
```

```
quatParts = compact(quat)
```

```
quatParts = 1×4

    0.5377    1.8339   -2.2588    0.8622
```

Create a 2-by-2 array of quaternions, then convert the representation to a matrix of quaternion parts. The output rows correspond to the linear indices of the quaternion array.

```
quatArray = [quaternion([1:4;5:8]),quaternion([9:12;13:16])]
```

```
quatArray=2×2 quaternion array
      1 +  2i +  3j +  4k      9 + 10i + 11j + 12k
      5 +  6i +  7j +  8k     13 + 14i + 15j + 16k
```

```
quatArrayParts = compact(quatArray)
```

```
quatArrayParts = 4×4
```

```
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

Input Arguments

`quat` — Quaternion to convert
scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

`matrix` — Quaternion in matrix form
N-by-4 matrix

Quaternion in matrix form, returned as an *N*-by-4 matrix, where *N* = `numel(quat)`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`classUnderlying` | `parts`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

conj

Complex conjugate of quaternion

Syntax

```
quatConjugate = conj(quat)
```

Description

`quatConjugate = conj(quat)` returns the complex conjugate of the quaternion, `quat`.

If $q = a + b\text{i} + c\text{j} + d\text{k}$, the complex conjugate of q is $q^* = a - b\text{i} - c\text{j} - d\text{k}$. Considered as a rotation operator, the conjugate performs the opposite rotation. For example,

```
q = quaternion(deg2rad([16 45 30]),'rotvec');
a = q*conj(q);
rotatepoint(a,[0,1,0])
```

```
ans =

     0     1     0
```

Examples

Complex Conjugate of Quaternion

Create a quaternion scalar and get the complex conjugate.

```
q = normalize(quaternion([0.9 0.3 0.3 0.25]))
```

```
q = quaternion
     0.87727 + 0.29242i + 0.29242j + 0.24369k
```

```
qConj = conj(q)
```

```
qConj = quaternion
     0.87727 - 0.29242i - 0.29242j - 0.24369k
```

Verify that a quaternion multiplied by its conjugate returns a quaternion one.

```
q*qConj
```

```
ans = quaternion
     1 + 0i + 0j + 0k
```

Input Arguments

quat — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion to conjugate, specified as a scalar, vector, matrix, or array of quaternions.

Data Types: `quaternion`

Output Arguments

quatConjugate — Quaternion conjugate
scalar | vector | matrix | multidimensional array

Quaternion conjugate, returned as a quaternion or array of quaternions the same size as `quat`.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`norm` | `times, .*`

Objects
`quaternion`

Topics
“Rotations, Orientations, and Quaternions for Automated Driving”

Introduced in R2020a

ctranspose, '

Complex conjugate transpose of quaternion array

Syntax

```
quatTransposed = quat'
```

Description

`quatTransposed = quat'` returns the complex conjugate transpose of the quaternion, `quat`.

Examples

Vector Complex Conjugate Transpose

Create a vector of quaternions and compute its complex conjugate transpose.

```
quat = quaternion(randn(4,4))
```

```
quat=4×1 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k
       1.8339 -   1.3077i +   2.7694j - 0.063055k
      -2.2588 -  0.43359i -   1.3499j +  0.71474k
      0.86217 +  0.34262i +   3.0349j -  0.20497k
```

```
quatTransposed = quat'
```

```
quatTransposed=1×4 quaternion array
      0.53767 -  0.31877i -   3.5784j -   0.7254k       1.8339 +   1.3077i -   2.7694j + 0.063055
```

Matrix Complex Conjugate Transpose

Create a matrix of quaternions and compute its complex conjugate transpose.

```
quat = [quaternion(randn(2,4)),quaternion(randn(2,4))]
```

```
quat=2×2 quaternion array
      0.53767 -   2.2588i +  0.31877j -  0.43359k       3.5784 -   1.3499i +   0.7254j +  0.7147
       1.8339 +  0.86217i -   1.3077j +  0.34262k       2.7694 +   3.0349i - 0.063055j -  0.2049
```

```
quatTransposed = quat'
```

```
quatTransposed=2×2 quaternion array
      0.53767 +   2.2588i -  0.31877j +  0.43359k       1.8339 -  0.86217i +   1.3077j -  0.3426
       3.5784 +   1.3499i -   0.7254j -  0.71474k       2.7694 -   3.0349i + 0.063055j +  0.2049
```

Input Arguments

`quat` — Quaternion to transpose
scalar | vector | matrix

Quaternion to transpose, specified as a vector or matrix or quaternions. The complex conjugate transpose is defined for 1-D and 2-D arrays.

Data Types: `quaternion`

Output Arguments

`quatTransposed` — Conjugate transposed quaternion
scalar | vector | matrix

Conjugate transposed quaternion, returned as an *N*-by-*M* array, where `quat` was specified as an *M*-by-*N* array.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`transpose, .'`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

dist

Angular distance in radians

Syntax

```
distance = dist(quatA,quatB)
```

Description

`distance = dist(quatA,quatB)` returns the angular distance in radians between two quaternions, `quatA` and `quatB`.

Examples

Calculate Quaternion Distance

Calculate the quaternion distance between a single quaternion and each element of a vector of quaternions. Define the quaternions using Euler angles.

```
q = quaternion([0,0,0],'eulerd','zyx','frame')
```

```
q = quaternion
     1 + 0i + 0j + 0k
```

```
qArray = quaternion([0,45,0;0,90,0;0,180,0;0,-90,0;0,-45,0],'eulerd','zyx','frame')
```

```
qArray=5×1 quaternion array
       0.92388 +         0i +   0.38268j +         0k
       0.70711 +         0i +   0.70711j +         0k
    6.1232e-17 +         0i +         1j +         0k
       0.70711 +         0i -   0.70711j +         0k
       0.92388 +         0i -   0.38268j +         0k
```

```
quaternionDistance = rad2deg(dist(q,qArray))
```

```
quaternionDistance = 5×1

   45.0000
   90.0000
  180.0000
   90.0000
   45.0000
```

If both arguments to `dist` are vectors, the quaternion distance is calculated between corresponding elements. Calculate the quaternion distance between two quaternion vectors.

```
angles1 = [30,0,15; ...
           30,5,15; ...
```

```
            30,10,15; ...
            30,15,15];
angles2 = [30,6,15; ...
            31,11,15; ...
            30,16,14; ...
            30.5,21,15.5];

qVector1 = quaternion(angles1,'eulerd','zyx','frame');
qVector2 = quaternion(angles2,'eulerd','zyx','frame');

rad2deg(dist(qVector1,qVector2))
```

```
ans = 4×1

    6.0000
    6.0827
    6.0827
    6.0287
```

Note that a quaternion represents the same rotation as its negative. Calculate a quaternion and its negative.

```
qPositive = quaternion([30,45,-60],'eulerd','zyx','frame')
```

```
qPositive = quaternion
     0.72332 - 0.53198i + 0.20056j +  0.3919k
```

```
qNegative = -qPositive
```

```
qNegative = quaternion
    -0.72332 + 0.53198i - 0.20056j -  0.3919k
```

Find the distance between the quaternion and its negative.

```
dist(qPositive,qNegative)
```

```
ans = 0
```

The components of a quaternion may look different from the components of its negative, but both expressions represent the same rotation.

Input Arguments

`quatA`,`quatB` — Quaternions to calculate distance between

scalar | vector | matrix | multidimensional array

Quaternions to calculate distance between, specified as comma-separated quaternions or arrays of quaternions. `quatA` and `quatB` must have compatible sizes:

- `size(quatA) == size(quatB)`, or
- `numel(quatA) == 1`, or
- `numel(quatB) == 1`, or

- if [Adim1,…,AdimN] = size(quatA) and [Bdim1,…,BdimN] = size(quatB), then for i = 1:N, either Adimi==Bdimi or Adim==1 or Bdim==1.

  If one of the quaternion arguments contains only one quaternion, then this function returns the distances between that quaternion and every quaternion in the other argument.

Data Types: quaternion

Output Arguments

distance — Angular distance (radians)
scalar | vector | matrix | multidimensional array

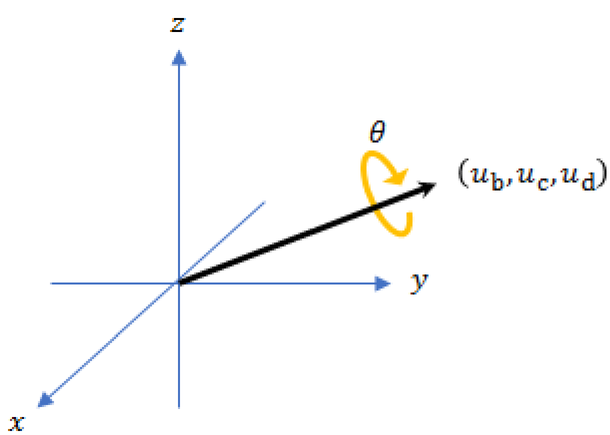
Angular distance in radians, returned as an array. The dimensions are the maximum of the union of size(quatA) and size(quatB).

Data Types: single | double

Algorithms

The dist function returns the angular distance between two quaternions.

A quaternion may be defined by an axis (u_b,u_c,u_d) and angle of rotation θ_q:
$q = \cos\left(\theta_q/2\right) + \sin\left(\theta_q/2\right)(u_b\mathrm{i} + u_c\mathrm{j} + u_d\mathrm{k})$.



Given a quaternion in the form, $q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, where a is the real part, you can solve for the angle of q as $\theta_q = 2\cos^{-1}(a)$.

Consider two quaternions, p and q, and the product $z = p*\mathrm{conjugate}(q)$. As p approaches q, the angle of z goes to 0, and z approaches the unit quaternion.

The angular distance between two quaternions can be expressed as $\theta_z = 2\cos^{-1}(\mathrm{real}(z))$.

Using the quaternion data type syntax, the angular distance is calculated as:

```
angularDistance = 2*acos(abs(parts(p*conj(q))));
```

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`conj` | `parts`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

euler

Convert quaternion to Euler angles (radians)

Syntax

```
eulerAngles = euler(quat,rotationSequence,rotationType)
```

Description

`eulerAngles = euler(quat,rotationSequence,rotationType)` converts the quaternion, `quat`, to an *N*-by-3 matrix of Euler angles.

Examples

Convert Quaternion to Euler Angles in Radians

Convert a quaternion frame rotation to Euler angles in radians using the 'ZYX' rotation sequence.

```
quat = quaternion([0.7071 0.7071 0 0]);
eulerAnglesRandians = euler(quat,'ZYX','frame')
```

```
eulerAnglesRandians = 1×3

         0         0    1.5708
```

Input Arguments

`quat` — Quaternion to convert to Euler angles
scalar | vector | matrix | multidimensional array

Quaternion to convert to Euler angles, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

`rotationSequence` — Rotation sequence
`'ZYX'` | `'ZYZ'` | `'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'XYZ'` | `'XYX'` | `'XZY'` | `'XZX'`

Rotation sequence of Euler representation, specified as a character vector or string.

The rotation sequence defines the order of rotations about the axes. For example, if you specify a rotation sequence of `'YZX'`:

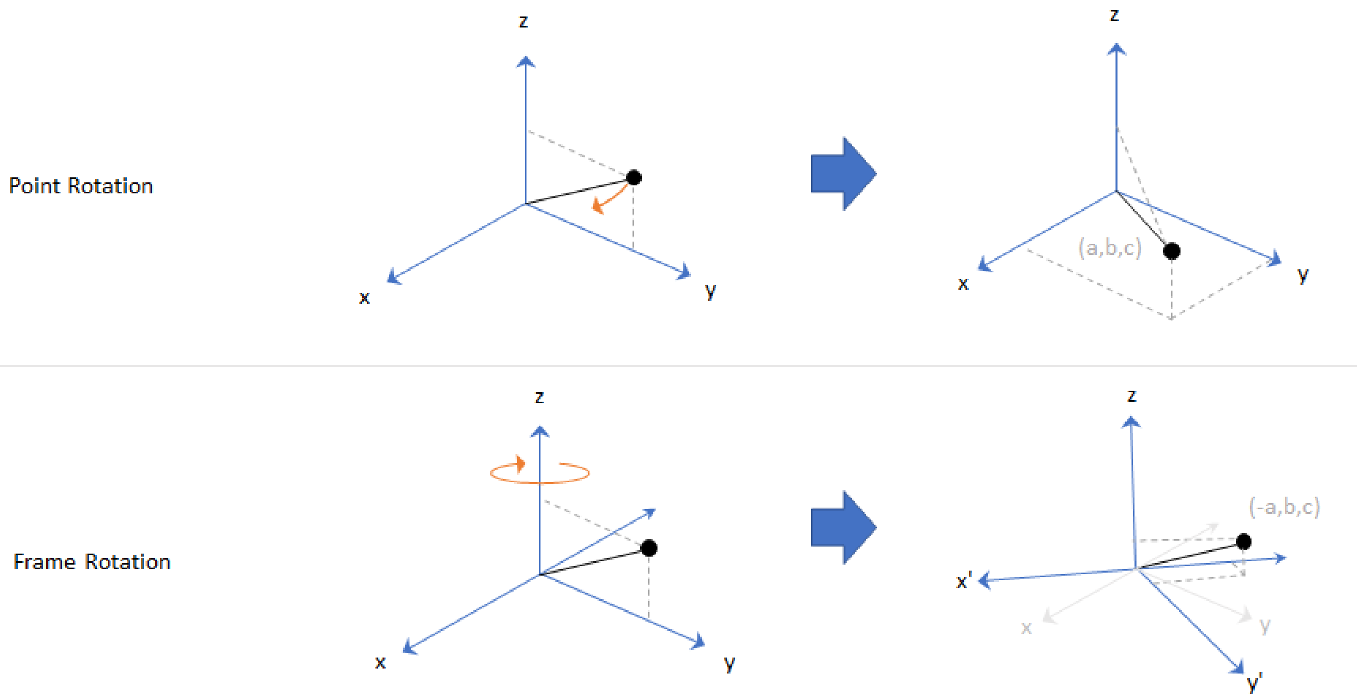
1. The first rotation is about the y-axis.
2. The second rotation is about the new z-axis.
3. The third rotation is about the new x-axis.

Data Types: `char` | `string`

rotationType — Type of rotation
'point' | 'frame'

Type of rotation, specified as 'point' or 'frame'.

In a point rotation, the frame is static and the point moves. In a frame rotation, the point is static and the frame moves. Point rotation and frame rotation define equivalent angular displacements but in opposite directions.



Data Types: char | string

Output Arguments

eulerAngles — Euler angle representation (radians)
N-by-3 matrix

Euler angle representation in radians, returned as a *N*-by-3 matrix. *N* is the number of quaternions in the quat argument.

For each row of eulerAngles, the first element corresponds to the first axis in the rotation sequence, the second element corresponds to the second axis in the rotation sequence, and the third element corresponds to the third axis in the rotation sequence.

The data type of the Euler angles representation is the same as the underlying data type of quat.

Data Types: single | double

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`eulerd` | `rotateframe` | `rotatepoint`

Objects

`quaternion`

Topics

"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

eulerd

Convert quaternion to Euler angles (degrees)

Syntax

```
eulerAngles = eulerd(quat,rotationSequence,rotationType)
```

Description

`eulerAngles = eulerd(quat,rotationSequence,rotationType)` converts the quaternion, `quat`, to an *N*-by-3 matrix of Euler angles in degrees.

Examples

Convert Quaternion to Euler Angles in Degrees

Convert a quaternion frame rotation to Euler angles in degrees using the 'ZYX' rotation sequence.

```
quat = quaternion([0.7071 0.7071 0 0]);
eulerAnglesDegrees = eulerd(quat,'ZYX','frame')
```

```
eulerAnglesDegrees = 1×3

         0         0   90.0000
```

Input Arguments

`quat` — Quaternion to convert to Euler angles
scalar | vector | matrix | multidimensional array

Quaternion to convert to Euler angles, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

`rotationSequence` — Rotation sequence
`'ZYX'` | `'ZYZ'` | `'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'XYZ'` | `'XYX'` | `'XZY'` | `'XZX'`

Rotation sequence of Euler angle representation, specified as a character vector or string.

The rotation sequence defines the order of rotations about the axes. For example, if you specify a rotation sequence of `'YZX'`:

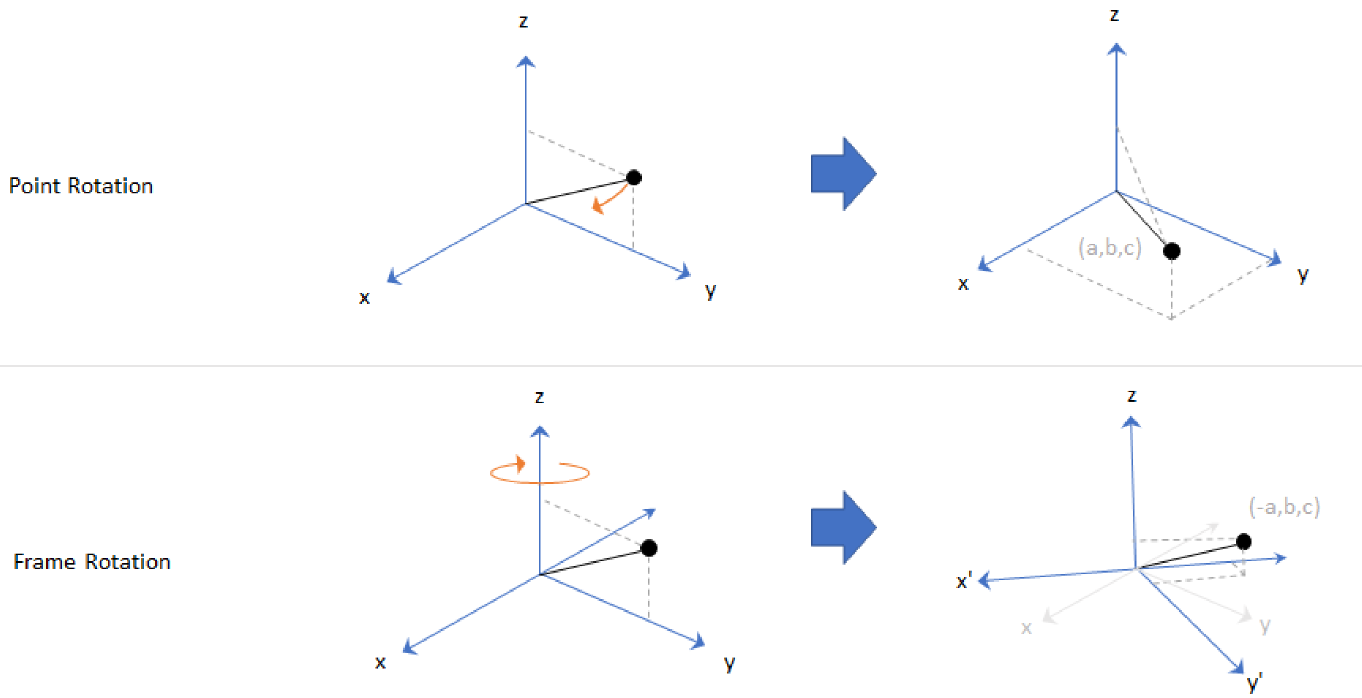
1. The first rotation is about the *y*-axis.
2. The second rotation is about the new *z*-axis.
3. The third rotation is about the new *x*-axis.

Data Types: `char` | `string`

rotationType — Type of rotation
'point' | 'frame'

Type of rotation, specified as 'point' or 'frame'.

In a point rotation, the frame is static and the point moves. In a frame rotation, the point is static and the frame moves. Point rotation and frame rotation define equivalent angular displacements but in opposite directions.



Data Types: char | string

Output Arguments

eulerAngles — Euler angle representation (degrees)
N-by-3 matrix

Euler angle representation in degrees, returned as a *N*-by-3 matrix. *N* is the number of quaternions in the quat argument.

For each row of eulerAngles, the first column corresponds to the first axis in the rotation sequence, the second column corresponds to the second axis in the rotation sequence, and the third column corresponds to the third axis in the rotation sequence.

The data type of the Euler angles representation is the same as the underlying data type of quat.

Data Types: single | double

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`euler` | `rotateframe` | `rotatepoint`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

exp

Exponential of quaternion array

Syntax

```
B = exp(A)
```

Description

`B = exp(A)` computes the exponential of the elements of the quaternion array `A`.

Examples

Exponential of Quaternion Array

Create a 4-by-1 quaternion array A.

```
A = quaternion(magic(4))
```

```
A=4×1 quaternion array
     16 +  2i +  3j + 13k
      5 + 11i + 10j +  8k
      9 +  7i +  6j + 12k
      4 + 14i + 15j +  1k
```

Compute the exponential of A.

```
B = exp(A)
```

```
B=4×1 quaternion array
     5.3525e+06 + 1.0516e+06i + 1.5774e+06j + 6.8352e+06k
        -57.359 -     89.189i -     81.081j -     64.865k
        -6799.1 +     2039.1i +     1747.8j +     3495.6k
          -6.66 +     36.931i +     39.569j +     2.6379k
```

Input Arguments

A — Input quaternion
scalar | vector | matrix | multidimensional array

Input quaternion, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Output Arguments

B — Result
scalar | vector | matrix | multidimensional array

Result of quaternion exponential, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Given a quaternion $A = a + b\text{i} + c\text{j} + d\text{k} = a + \bar{v}$, the exponential is computed by

$$\exp(A) = e^{a}\left(\cos\|\bar{v}\| + \frac{\bar{v}}{\|\bar{v}\|}\sin\|\bar{v}\|\right)$$

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`log` | `power, .^`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

ldivide, .\

Element-wise quaternion left division

Syntax

`C = A.\B`

Description

`C = A.\B` performs quaternion element-wise division by dividing each element of quaternion `B` by the corresponding element of quaternion `A`.

Examples

Divide a Quaternion Array by a Real Scalar

Create a 2-by-1 quaternion array, and divide it element-by-element by a real scalar.

```
A = quaternion([1:4;5:8])
```

```
A=2×1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

```
B = 2;
C = A.\B
```

```
C=2×1 quaternion array
     0.066667 -  0.13333i -      0.2j -  0.26667k
     0.057471 - 0.068966i -  0.08046j - 0.091954k
```

Divide a Quaternion Array by Another Quaternion Array

Create a 2-by-2 quaternion array, and divide it element-by-element by another 2-by-2 quaternion array.

```
q1 = quaternion([1:4;2:5;4:7;5:8]);
A = reshape(q1,2,2)
```

```
A=2×2 quaternion array
     1 + 2i + 3j + 4k     4 + 5i + 6j + 7k
     2 + 3i + 4j + 5k     5 + 6i + 7j + 8k
```

```
q2 = quaternion(magic(4));
B = reshape(q2,2,2)
```

```
B=2×2 quaternion array
     16 +  2i +  3j + 13k      9 +  7i +  6j + 12k
      5 + 11i + 10j +  8k      4 + 14i + 15j +  1k
```

```
C = A.\B
```

```
C=2×2 quaternion array
          2.7 -      1.9i -      0.9j -      1.7k       1.5159 -  0.37302i -  0.15079j -  0.0238
       2.2778 +  0.46296i -  0.57407j + 0.092593k       1.2471 +  0.91379i -  0.33908j -   0.109
```

Input Arguments

A — Divisor
scalar | vector | matrix | multidimensional array

Divisor, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: `quaternion` | `single` | `double`

B — Dividend
scalar | vector | matrix | multidimensional array

Dividend, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: `quaternion` | `single` | `double`

Output Arguments

C — Result
scalar | vector | matrix | multidimensional array

Result of quaternion division, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Quaternion Division

Given a quaternion $A = a_1 + a_2\mathrm{i} + a_3\mathrm{j} + a_4\mathrm{k}$ and a real scalar p,

$$C = p.\backslash A = \frac{a_1}{p} + \frac{a_2}{p}\mathrm{i} + \frac{a_3}{p}\mathrm{j} + \frac{a_4}{p}\mathrm{k}$$

Note For a real scalar p, $A./p = A.\backslash p$.

Quaternion Division by a Quaternion Scalar

Given two quaternions A and B of compatible sizes, then

$$C = A.\backslash B = A^{-1}.*B = \left(\frac{conj(A)}{norm(A)^2}\right).*B$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`conj` | `norm` | `rdivide, ./` | `times, .*`

Objects

`quaternion`

Topics

"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

log

Natural logarithm of quaternion array

Syntax

`B = log(A)`

Description

`B = log(A)` computes the natural logarithm of the elements of the quaternion array `A`.

Examples

Logarithmic Values of Quaternion Array

Create a 3-by-1 quaternion array A.

```
A = quaternion(randn(3,4))
```

```
A=3×1 quaternion array
     0.53767 + 0.86217i - 0.43359j +  2.7694k
      1.8339 + 0.31877i + 0.34262j -  1.3499k
     -2.2588 -  1.3077i +  3.5784j +  3.0349k
```

Compute the logarithmic values of A.

```
B = log(A)
```

```
B=3×1 quaternion array
      1.0925 + 0.40848i - 0.20543j +  1.3121k
      0.8436 + 0.14767i + 0.15872j - 0.62533k
      1.6807 - 0.53829i +   1.473j +  1.2493k
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Output Arguments

B — Logarithm values
scalar | vector | matrix | multidimensional array

Quaternion natural logarithm values, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Given a quaternion $A = a + \bar{v} = a + b\text{i} + c\text{j} + d\text{k}$, the logarithm is computed by

$$\log(A) = \log\|A\| + \frac{\bar{v}}{\|\bar{v}\|}\arccos\frac{a}{\|A\|}$$

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`exp` | `power, .^`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

meanrot

Quaternion mean rotation

Syntax

```
quatAverage = meanrot(quat)
quatAverage = meanrot(quat,dim)
quatAverage = meanrot( ___ ,nanflag)
```

Description

`quatAverage = meanrot(quat)` returns the average rotation of the elements of `quat` along the first array dimension whose size not does equal 1.

- If `quat` is a vector, `meanrot(quat)` returns the average rotation of the elements.
- If `quat` is a matrix, `meanrot(quat)` returns a row vector containing the average rotation of each column.
- If `quat` is a multidimensional array, then `mearot(quat)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors. This dimension becomes 1 while the sizes of all other dimensions remain the same.

The `meanrot` function normalizes the input quaternions, `quat`, before calculating the mean.

`quatAverage = meanrot(quat,dim)` return the average rotation along dimension `dim`. For example, if `quat` is a matrix, then `meanrot(quat,2)` is a column vector containing the mean of each row.

`quatAverage = meanrot( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `meanrot(quat,'includenan')` includes all `NaN` values in the calculation while `mean(quat,'omitnan')` ignores them.

Examples

Quaternion Mean Rotation

Create a matrix of quaternions corresponding to three sets of Euler angles.

```
eulerAngles = [40 20 10; ...
               50 10 5; ...
               45 70 1];

quat = quaternion(eulerAngles,'eulerd','ZYX','frame');
```

Determine the average rotation represented by the quaternions. Convert the average rotation to Euler angles in degrees for readability.

```
quatAverage = meanrot(quat)
```

```
quatAverage = quaternion
      0.88863 - 0.062598i +  0.27822j +  0.35918k
```

```
eulerAverage = eulerd(quatAverage,'ZYX','frame')
```

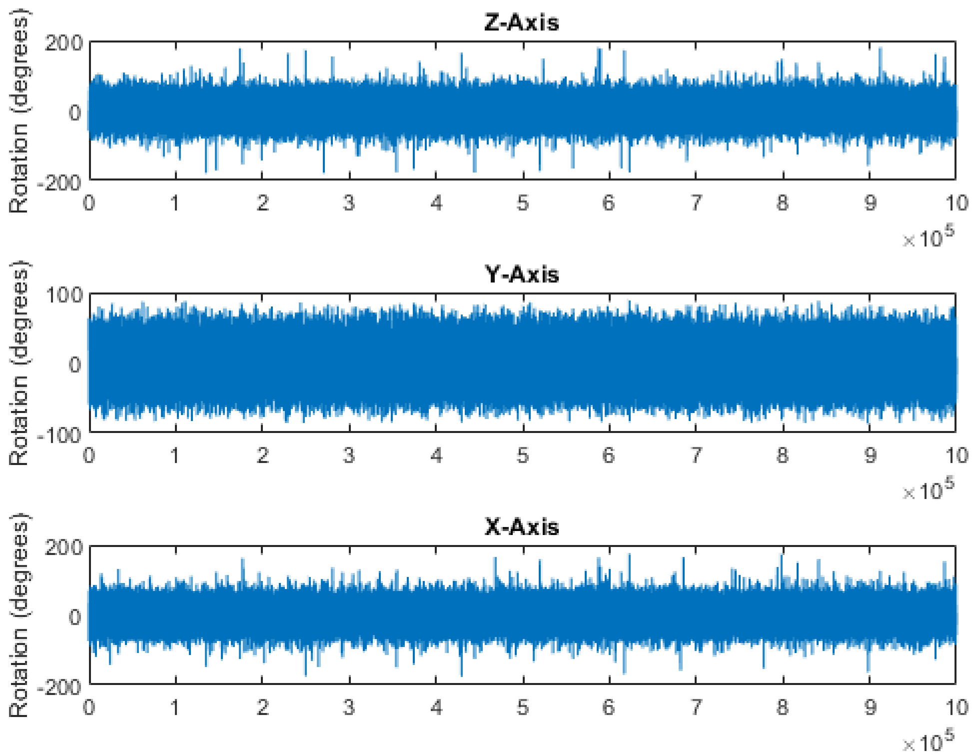
```
eulerAverage = 1×3

   45.7876   32.6452    6.0407
```

Average Out Rotational Noise

Use `meanrot` over a sequence of quaternions to average out additive noise.

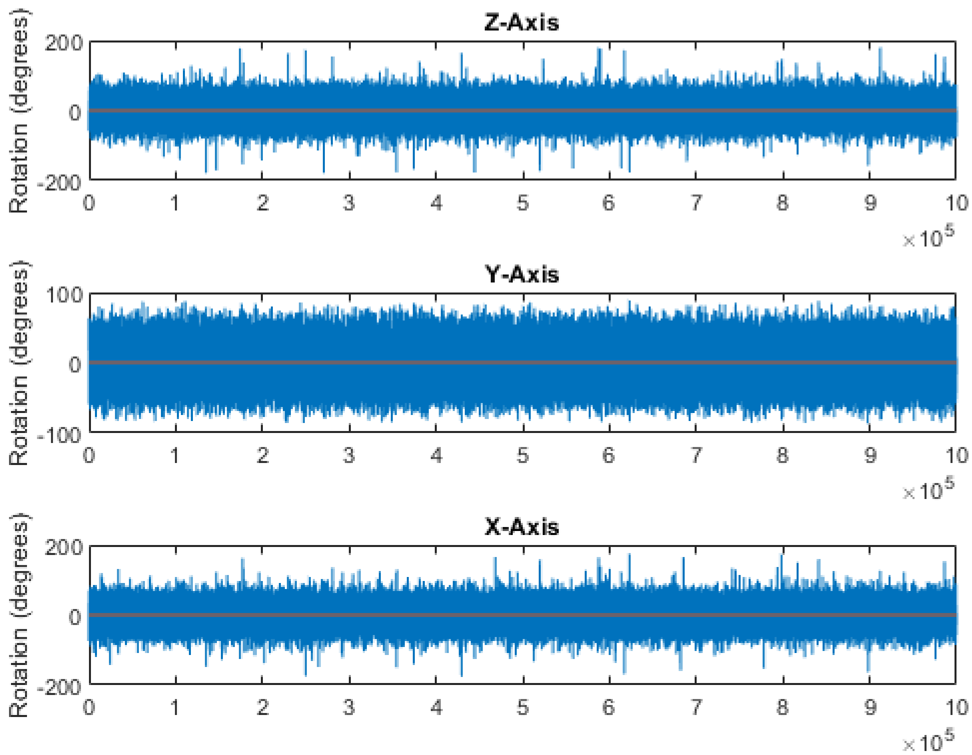
Create a vector of 1e6 quaternions whose distance, as defined by the `dist` function, from quaternion(1,0,0,0) is normally distributed. Plot the Euler angles corresponding to the noisy quaternion vector.

```
nrows = 1e6;
ax = 2*rand(nrows,3) - 1;
ax = ax./sqrt(sum(ax.^2,2));
ang = 0.5*randn(size(ax,1),1);
q = quaternion(ax.*ang ,'rotvec');

noisyEulerAngles = eulerd(q,'ZYX','frame');

figure(1)

subplot(3,1,1)
plot(noisyEulerAngles(:,1))
title('Z-Axis')
ylabel('Rotation (degrees)')
hold on

subplot(3,1,2)
plot(noisyEulerAngles(:,2))
title('Y-Axis')
ylabel('Rotation (degrees)')
hold on

subplot(3,1,3)
plot(noisyEulerAngles(:,3))
title('X-Axis')
ylabel('Rotation (degrees)')
hold on
```



Use `meanrot` to determine the average quaternion given the vector of quaternions. Convert to Euler angles and plot the results.

```
qAverage = meanrot(q);

qAverageInEulerAngles = eulerd(qAverage,'ZYX','frame');

figure(1)

subplot(3,1,1)
plot(ones(nrows,1)*qAverageInEulerAngles(:,1))
title('Z-Axis')

subplot(3,1,2)
plot(ones(nrows,1)*qAverageInEulerAngles(:,2))
title('Y-Axis')

subplot(3,1,3)
plot(ones(nrows,1)*qAverageInEulerAngles(:,3))
title('X-Axis')
```



The `meanrot` Algorithm and Limitations

The `meanrot` Algorithm

The `meanrot` function outputs a quaternion that minimizes the squared Frobenius norm of the difference between rotation matrices. Consider two quaternions:

- `q0` represents no rotation.
- `q90` represents a 90 degree rotation about the *x*-axis.

```
q0 = quaternion([0 0 0],'eulerd','ZYX','frame');
q90 = quaternion([0 0 90],'eulerd','ZYX','frame');
```

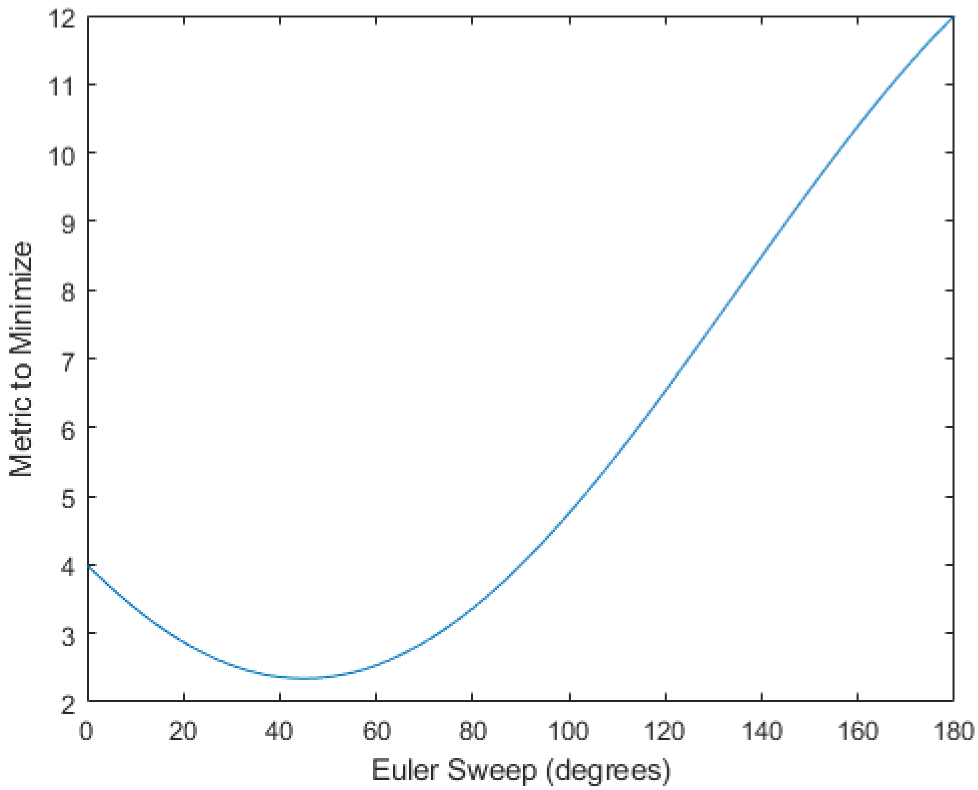
Create a quaternion sweep, `qSweep`, that represents rotations from 0 to 180 degrees about the *x*-axis.

```
eulerSweep = (0:1:180)';
qSweep = quaternion([zeros(numel(eulerSweep),2),eulerSweep], ...
    'eulerd','ZYX','frame');
```

Convert `q0`, `q90`, and `qSweep` to rotation matrices. In a loop, calculate the metric to minimize for each member of the quaternion sweep. Plot the results and return the value of the Euler sweep that corresponds to the minimum of the metric.

```
r0     = rotmat(q0,'frame');
r90    = rotmat(q90,'frame');
```

```
rSweep = rotmat(qSweep,'frame');

metricToMinimize = zeros(size(rSweep,3),1);
for i = 1:numel(qSweep)
    metricToMinimize(i) = norm((rSweep(:,:,i) - r0),'fro').^2 + ...
                          norm((rSweep(:,:,i) - r90),'fro').^2;
end

plot(eulerSweep,metricToMinimize)
xlabel('Euler Sweep (degrees)')
ylabel('Metric to Minimize')
```




```
[~,eulerIndex] = min(metricToMinimize);
eulerSweep(eulerIndex)
```

```
ans = 45
```

The minimum of the metric corresponds to the Euler angle sweep at 45 degrees. That is, `meanrot` defines the average between `quaterion([0 0 0],'ZYX','frame')` and `quaternion([0 0 90],'ZYX','frame')` as `quaternion([0 0 45],'ZYX','frame')`. Call `meanrot` with `q0` and `q90` to verify the same result.

```
eulerd(meanrot([q0,q90]),'ZYX','frame')
```

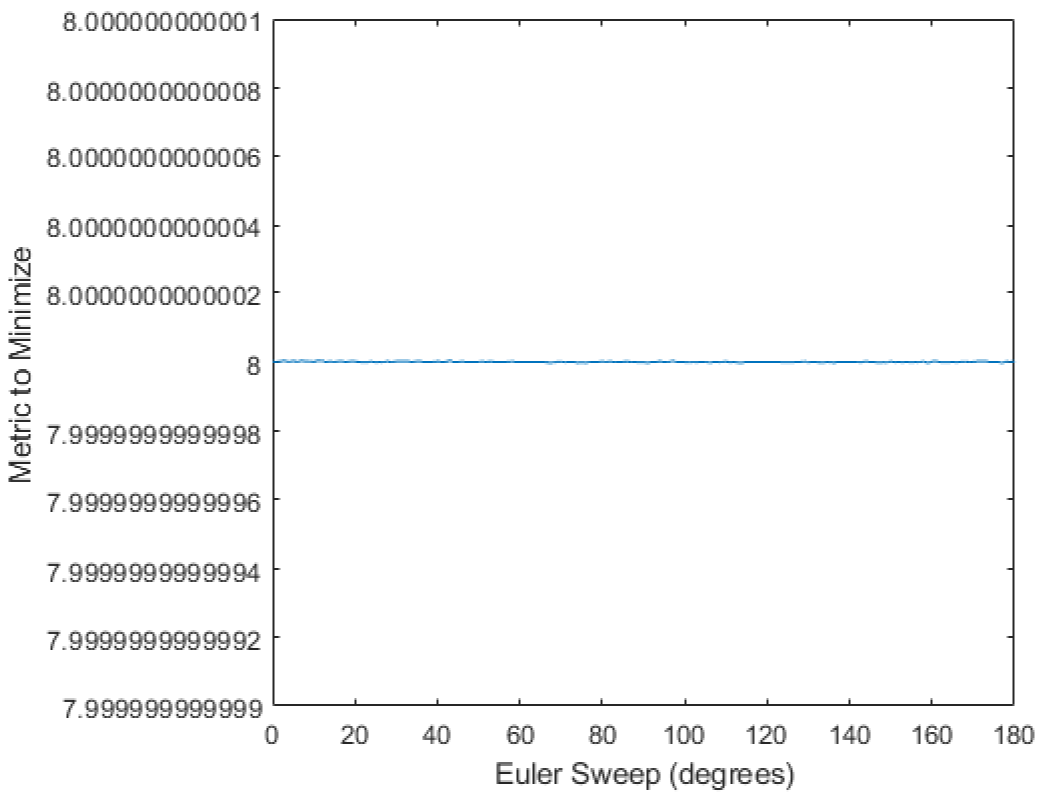
```
ans = 1×3
```

```
         0         0   45.0000
```

Limitations

The metric that `meanrot` uses to determine the mean rotation is not unique for quaternions significantly far apart. Repeat the experiment above for quaternions that are separated by 180 degrees.

```
q180 = quaternion([0 0 180],'eulerd','ZYX','frame');
r180 = rotmat(q180,'frame');

for i = 1:numel(qSweep)
    metricToMinimize(i) = norm((rSweep(:,:,i) - r0),'fro').^2 + ...
                          norm((rSweep(:,:,i) - r180),'fro').^2;
end

plot(eulerSweep,metricToMinimize)
xlabel('Euler Sweep (degrees)')
ylabel('Metric to Minimize')
```




```
[~,eulerIndex] = min(metricToMinimize);
eulerSweep(eulerIndex)
```

```
ans = 159
```

Quaternion means are usually calculated for rotations that are close to each other, which makes the edge case shown in this example unlikely in real-world applications. To average two quaternions that are significantly far apart, use the `slerp` function. Repeat the experiment using `slerp` and verify that the quaternion mean returned is more intuitive for large distances.

```
qMean = slerp(q0,q180,0.5);
q0_q180 = eulerd(qMean,'ZYX','frame')
```

```
q0_q180 = 1×3

         0         0   90.0000
```

Input Arguments

quat — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion for which to calculate the mean, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

dim — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to 1. The `size(quatAverage,dim)` is `1`, while the sizes of all other dimensions remain the same.

Data Types: `double` | `single`

nanflag — NaN condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` –– Include `NaN` values when computing the mean rotation, resulting in `NaN`.
- `'omitnan'` –– Ignore all `NaN` values in the input.

Data Types: `char` | `string`

Output Arguments

quatAverage — Quaternion average rotation
scalar | vector | matrix | multidimensional array

Quaternion average rotation, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Algorithms

`meanrot` determines a quaternion mean, $\bar{q}$, according to [1]. $\bar{q}$ is the quaternion that minimizes the squared Frobenius norm of the difference between rotation matrices:

$$\bar{q} = \arg \min_{q \in S^3} \sum_{i=1}^{n} \|A(q) - A(q_i)\|_F^2$$

References

[1] Markley, F. Landis, Yang Chen, John Lucas Crassidis, and Yaakov Oshman. "Average Quaternions." *Journal of Guidance, Control, and Dynamics*. Vol. 30, Issue 4, 2007, pp. 1193-1197.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`dist` | `slerp`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

minus, -

Quaternion subtraction

Syntax

`C = A - B`

Description

`C = A - B` subtracts quaternion `B` from quaternion `A` using quaternion subtraction. Either `A` or `B` may be a real number, in which case subtraction is performed with the real part of the quaternion argument.

Examples

Subtract a Quaternion from a Quaternion

Quaternion subtraction is defined as the subtraction of the corresponding parts of each quaternion. Create two quaternions and perform subtraction.

```
Q1 = quaternion([1,0,-2,7]);
Q2 = quaternion([1,2,3,4]);

Q1minusQ2 = Q1 - Q2
```

```
Q1minusQ2 = quaternion
     0 - 2i - 5j + 3k
```

Subtract a Real Number from a Quaternion

Addition and subtraction of real numbers is defined for quaternions as acting on the real part of the quaternion. Create a quaternion and then subtract 1 from the real part.

```
Q = quaternion([1,1,1,1])
```

```
Q = quaternion
     1 + 1i + 1j + 1k
```

```
Qminus1 = Q - 1
```

```
Qminus1 = quaternion
     0 + 1i + 1j + 1k
```

Input Arguments

A — Input
scalar | vector | matrix | multidimensional array

Input, specified as a quaternion, array of quaternions, real number, or array of real numbers.

Data Types: `quaternion` | `single` | `double`

B — Input
scalar | vector | matrix | multidimensional array

Input, specified as a quaternion, array of quaternions, real number, or array of real numbers.

Data Types: `quaternion` | `single` | `double`

Output Arguments

C — Result
scalar | vector | matrix | multidimensional array

Result of quaternion subtraction, returned as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`mtimes, *` | `times, .*` | `uminus, -`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

mtimes, *

Quaternion multiplication

Syntax

`quatC = A*B`

Description

`quatC = A*B` implements quaternion multiplication if either `A` or `B` is a quaternion. Either `A` or `B` must be a scalar.

You can use quaternion multiplication to compose rotation operators:

- To compose a sequence of frame rotations, multiply the quaternions in the order of the desired sequence of rotations. For example, to apply a *p* quaternion followed by a *q* quaternion, multiply in the order *pq*. The rotation operator becomes $(pq)^{*}v(pq)$, where *v* represents the object to rotate specified in quaternion form. * represents conjugation.
- To compose a sequence of point rotations, multiply the quaternions in the reverse order of the desired sequence of rotations. For example, to apply a *p* quaternion followed by a *q* quaternion, multiply in the reverse order, *qp*. The rotation operator becomes $(qp)v(qp)^{*}$.

Examples

Multiply Quaternion Scalar and Quaternion Vector

Create a 4-by-1 column vector, `A`, and a scalar, `b`. Multiply `A` times `b`.

```
A = quaternion(randn(4,4))
```

```
A=4×1 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k
       1.8339 -   1.3077i +   2.7694j - 0.063055k
      -2.2588 -  0.43359i -   1.3499j +  0.71474k
      0.86217 +  0.34262i +   3.0349j -  0.20497k
```

```
b = quaternion(randn(1,4))
```

```
b = quaternion
    -0.12414 +  1.4897i +   1.409j +  1.4172k
```

```
C = A*b
```

```
C=4×1 quaternion array
      -6.6117 +   4.8105i +  0.94224j -   4.2097k
      -2.0925 +   6.9079i +   3.9995j -   3.3614k
       1.8155 -   6.2313i -    1.336j -     1.89k
      -4.6033 +   5.8317i + 0.047161j -    2.791k
```

Input Arguments

A — Input
scalar | vector | matrix | multidimensional array

Input to multiply, specified as a quaternion, array of quaternions, real scalar, or array of real scalars.

If B is nonscalar, then A must be scalar.

Data Types: quaternion | single | double

B — Input
scalar | vector | matrix | multidimensional array

Input to multiply, specified as a quaternion, array of quaternions, real scalar, or array of real scalars.

If A is nonscalar, then B must be scalar.

Data Types: quaternion | single | double

Output Arguments

quatC — Quaternion product
scalar | vector | matrix | multidimensional array

Quaternion product, returned as a quaternion or array of quaternions.

Data Types: quaternion

Algorithms

Quaternion Multiplication by a Real Scalar

Given a quaternion

$$q = a_{\mathrm{q}} + b_{\mathrm{q}}\mathrm{i} + c_{\mathrm{q}}\mathrm{j} + d_{\mathrm{q}}\mathrm{k},$$

the product of q and a real scalar β is

$$\beta q = \beta a_{\mathrm{q}} + \beta b_{\mathrm{q}}\mathrm{i} + \beta c_{\mathrm{q}}\mathrm{j} + \beta d_{\mathrm{q}}\mathrm{k}$$

Quaternion Multiplication by a Quaternion Scalar

The definition of the basis elements for quaternions,

$$\mathrm{i}^2 = \mathrm{j}^2 = \mathrm{k}^2 = \mathrm{ijk} = -1\,,$$

can be expanded to populate a table summarizing quaternion basis element multiplication:

	1	**i**	**j**	**k**
1	1	i	j	k
i	i	−1	k	−j

j	j	−k	−1	i
k	k	j	−i	−1

When reading the table, the rows are read first, for example: ij = k and ji = −k.

Given two quaternions, $q = a_q + b_q\mathrm{i} + c_q\mathrm{j} + d_q\mathrm{k}$, and $p = a_p + b_p\mathrm{i} + c_p\mathrm{j} + d_p\mathrm{k}$, the multiplication can be expanded as:

$$\begin{aligned} z = pq &= (a_p + b_p\mathrm{i} + c_p\mathrm{j} + d_p\mathrm{k})(a_q + b_q\mathrm{i} + c_q\mathrm{j} + d_q\mathrm{k}) \\ &= a_pa_q + a_pb_q\mathrm{i} + a_pc_q\mathrm{j} + a_pd_q\mathrm{k} \\ &+ b_pa_q\mathrm{i} + b_pb_q\mathrm{i}^2 + b_pc_q\mathrm{ij} + b_pd_q\mathrm{ik} \\ &+ c_pa_q\mathrm{j} + c_pb_q\mathrm{ji} + c_pc_q\mathrm{j}^2 + c_pd_q\mathrm{jk} \\ &+ d_pa_qk + d_pb_q\mathrm{ki} + d_pc_q\mathrm{kj} + d_pd_q\mathrm{k}^2 \end{aligned}$$

You can simplify the equation using the quaternion multiplication table:

$$\begin{aligned} z = pq \ &= a_pa_q + a_pb_q\mathrm{i} + a_pc_q\mathrm{j} + a_pd_q\mathrm{k} \\ &+ b_pa_q\mathrm{i} - b_pb_q + b_pc_q\mathrm{k} - b_pd_q\mathrm{j} \\ &+ c_pa_q\mathrm{j} - c_pb_q\mathrm{k} - c_pc_q + c_pd_q\mathrm{i} \\ &+ d_pa_qk + d_pb_q\mathrm{j} - d_pc_q\mathrm{i} - d_pd_q \end{aligned}$$

References

[1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`times, .*`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

norm

Quaternion norm

Syntax

```
N = norm(quat)
```

Description

`N = norm(quat)` returns the norm of the quaternion, `quat`.

Given a quaternion of the form $Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, the norm of the quaternion is defined as $\mathrm{norm}(Q) = \sqrt{a^2 + b^2 + c^2 + d^2}$.

Examples

Calculate Quaternion Norm

Create a scalar quaternion and calculate its norm.

```
quat = quaternion(1,2,3,4);
norm(quat)
```

```
ans = 5.4772
```

The quaternion norm is defined as the square root of the sum of the quaternion parts squared. Calculate the quaternion norm explicitly to verify the result of the `norm` function.

```
[a,b,c,d] = parts(quat);
sqrt(a^2+b^2+c^2+d^2)
```

```
ans = 5.4772
```

Input Arguments

quat — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion for which to calculate the norm, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

N — Quaternion norm
scalar | vector | matrix | multidimensional array

Quaternion norm. If the input `quat` is an array, the output is returned as an array the same size as `quat`. Elements of the array are real numbers with the same data type as the underlying data type of the quaternion, `quat`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`conj` | `normalize` | `parts`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

normalize

Quaternion normalization

Syntax

```
quatNormalized = normalize(quat)
```

Description

`quatNormalized = normalize(quat)` normalizes the quaternion.

Given a quaternion of the form $Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, the normalized quaternion is defined as $Q/\sqrt{a^2 + b^2 + c^2 + d^2}$.

Examples

Normalize Elements of Quaternion Vector

Quaternions can represent rotations when normalized. You can use `normalize` to normalize a scalar, elements of a matrix, or elements of a multi-dimensional array of quaternions. Create a column vector of quaternions, then normalize them.

```
quatArray = quaternion([1,2,3,4; ...
                        2,3,4,1; ...
                        3,4,1,2]);
quatArrayNormalized = normalize(quatArray)
```

```
quatArrayNormalized=3×1 quaternion array
     0.18257 + 0.36515i + 0.54772j +  0.7303k
     0.36515 + 0.54772i +  0.7303j + 0.18257k
     0.54772 +  0.7303i + 0.18257j + 0.36515k
```

Input Arguments

quat — Quaternion to normalize
scalar | vector | matrix | multidimensional array

Quaternion to normalize, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

quatNormalized — Normalized quaternion
scalar | vector | matrix | multidimensional array

Normalized quaternion, returned as a quaternion or array of quaternions the same size as `quat`.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`conj` | `norm` | `times, .*`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

ones

Create quaternion array with real parts set to one and imaginary parts set to zero

Syntax

```
quatOnes = ones('quaternion')
quatOnes = ones(n,'quaternion')
quatOnes = ones(sz,'quaternion')
quatOnes = ones(sz1,...,szN,'quaternion')

quatOnes = ones( ___ ,'like',prototype,'quaternion')
```

Description

`quatOnes = ones('quaternion')` returns a scalar quaternion with the real part set to 1 and the imaginary parts set to 0.

Given a quaternion of the form $Q = a + b\text{i} + c\text{j} + d\text{k}$, a quaternion one is defined as $Q = 1 + 0\text{i} + 0\text{j} + 0\text{k}$.

`quatOnes = ones(n,'quaternion')` returns an `n`-by-`n` quaternion matrix with the real parts set to 1 and the imaginary parts set to 0.

`quatOnes = ones(sz,'quaternion')` returns an array of quaternion ones where the size vector, `sz`, defines `size(qOnes)`.

Example: `ones([1,4,2],'quaternion')` returns a 1-by-4-by-2 array of quaternions with the real parts set to `1` and the imaginary parts set to `0`.

`quatOnes = ones(sz1,...,szN,'quaternion')` returns a `sz1`-by-...-by-`szN` array of ones where `sz1,…,szN` indicates the size of each dimension.

`quatOnes = ones( ___ ,'like',prototype,'quaternion')` specifies the underlying class of the returned quaternion array to be the same as the underlying class of the quaternion prototype.

Examples

Quaternion Scalar One

Create a quaternion scalar one.

```
quatOnes = ones('quaternion')
```

```
quatOnes = quaternion
     1 + 0i + 0j + 0k
```

Square Matrix of Quaternion Ones

Create an n-by-n matrix of quaternion ones.

```
n = 3;
quatOnes = ones(n,'quaternion')
```

```
quatOnes=3×3 quaternion array
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
```

Multidimensional Array of Quaternion Ones

Create a multidimensional array of quaternion ones by defining array dimensions in order. In this example, you create a 3-by-1-by-2 array. You can specify dimensions using a row vector or comma-separated integers. Specify the dimensions using a row vector and display the results:

```
dims = [3,1,2];
quatOnesSyntax1 = ones(dims,'quaternion')
```

```
quatOnesSyntax1 = 3x1x2 quaternion array
quatOnesSyntax1(:,:,1) =

     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k


quatOnesSyntax1(:,:,2) =

     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k
```

Specify the dimensions using comma-separated integers, and then verify the equivalency of the two syntaxes:

```
quatOnesSyntax2 = ones(3,1,2,'quaternion');
isequal(quatOnesSyntax1,quatOnesSyntax2)
```

```
ans = logical
   1
```

Underlying Class of Quaternion Ones

A quaternion is a four-part hyper-complex number used in three-dimensional rotations and orientations. You can specify the underlying data type of the parts as `single` or `double`. The default is `double`.

Create a quaternion array of ones with the underlying data type set to `single`.

```
quatOnes = ones(2,'like',single(1),'quaternion')
```

```
quatOnes=2×2 quaternion array
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
```

Verify the underlying class using the `classUnderlying` function.

```
classUnderlying(quatOnes)
```

```
ans = 
'single'
```

Input Arguments

n — Size of square quaternion matrix
integer value

Size of square quaternion matrix, specified as an integer value.

If `n` is zero or negative, then `quatOnes` is returned as an empty matrix.

Example: `ones(4,'quaternion')` returns a 4-by-4 matrix of quaternions with the real parts set to `1` and the imaginary parts set to `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz — Output size
row vector of integer values

Output size, specified as a row vector of integer values. Each element of `sz` indicates the size of the corresponding dimension in `quatOnes`. If the size of any dimension is `0` or negative, then `quatOnes` is returned as an empty array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

prototype — Quaternion prototype
variable

Quaternion prototype, specified as a variable.

Example: `ones(2,'like',quat,'quaternion')` returns a 2-by-2 matrix of quaternions with the same underlying class as the prototype quaternion, `quat`.

Data Types: `quaternion`

sz1,...,szN — Size of each dimension
two or more integer values

Size of each dimension, specified as two or more integers. If the size of any dimension is `0` or negative, then `quatOnes` is returned as an empty array.

Example: `ones(2,3,'quaternion')` returns a 2-by-3 matrix of quaternions with the real parts set to `1` and the imaginary parts set to `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`quatOnes` — Quaternion ones
scalar | vector | matrix | multidimensional array

Quaternion ones, returned as a scalar, vector, matrix, or multidimensional array of quaternions.

Given a quaternion of the form $Q = a + b\text{i} + c\text{j} + d\text{k}$, a quaternion one is defined as $Q = 1 + 0\text{i} + 0\text{j} + 0\text{k}$.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`zeros`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

parts

Extract quaternion parts

Syntax

```
[a,b,c,d] = parts(quat)
```

Description

`[a,b,c,d] = parts(quat)` returns the parts of the quaternion array as arrays, each the same size as `quat`.

Examples

Convert Quaternion to Matrix of Quaternion Parts

Convert a quaternion representation to parts using the `parts` function.

Create a two-element column vector of quaternions by specifying the parts.

```
quat = quaternion([1:4;5:8])
```

```
quat=2×1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

Recover the parts from the quaternion matrix using the `parts` function. The parts are returned as separate output arguments, each the same size as the input 2-by-1 column vector of quaternions.

```
[qA,qB,qC,qD] = parts(quat)
```

```
qA = 2×1

     1
     5


qB = 2×1

     2
     6


qC = 2×1

     3
     7


qD = 2×1
```

```
     4
     8
```

Input Arguments

quat — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion, specified as a quaternion or array of quaternions.

Data Types: `quaternion`

Output Arguments

[a,b,c,d] — Quaternion parts
scalar | vector | matrix | multidimensional array

Quaternion parts, returned as four arrays: a, b, d, and d. Each part is the same size as `quat`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`classUnderlying` | `compact`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

power, .^

Element-wise quaternion power

Syntax

```
C = A.^b
```

Description

`C = A.^b` raises each element of `A` to the corresponding power in `b`.

Examples

Raise a Quaternion to a Real Scalar Power

Create a quaternion and raise it to a real scalar power.

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

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

```
b = 3;
C = A.^b
```

```
C = quaternion
     -86 -  52i -  78j - 104k
```

Raise a Quaternion Array to Powers from a Multidimensional Array

Create a 2-by-1 quaternion array and raise it to powers from a 2-D array.

```
A = quaternion([1:4;5:8])
```

```
A=2×1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

```
b = [1 0 2; 3 2 1]
```

```
b = 2×3

     1     0     2
     3     2     1
```

```
C = A.^b
```

```
C=2×3 quaternion array
        1 +    2i +    3j +    4k        1 +    0i +    0j +    0k      -28 +    4i +    6j +
    -2110 -  444i -  518j -  592k     -124 +   60i +   70j +   80k        5 +    6i +    7j +
```

Input Arguments

A — Base
scalar | vector | matrix | multidimensional array

Base, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion` | `single` | `double`

b — Exponent
scalar | vector | matrix | multidimensional array

Exponent, specified as a real scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Output Arguments

C — Result
scalar | vector | matrix | multidimensional array

Each element of quaternion A raised to the corresponding power in b, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

The polar representation of a quaternion $A = a + b\text{i} + c\text{j} + d\text{k}$ is given by

$$A = \|A\|(\cos\theta + \widehat{u}\sin\theta)$$

where θ is the angle of rotation, and $\hat{u}$ is the unit quaternion.

Quaternion A raised by a real exponent b is given by

$$P = A\,.\hat{}\,b = \|A\|^{b}(\cos(b\theta) + \widehat{u}\sin(b\theta))$$

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`exp` | `log`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

prod

Product of a quaternion array

Syntax

```
quatProd = prod(quat)
quatProd = prod(quat,dim)
```

Description

`quatProd = prod(quat)` returns the quaternion product of the elements of the array.

`quatProd = prod(quat,dim)` calculates the quaternion product along dimension `dim`.

Examples

Product of Quaternions in Each Column

Create a 3-by-3 array whose elements correspond to their linear indices.

```
A = reshape(quaternion(randn(9,4)),3,3)
```

```
A=3×3 quaternion array
      0.53767 +   2.7694i +    1.409j -  0.30344k      0.86217 +   0.7254i -   1.2075j +   0.888
       1.8339 -   1.3499i +   1.4172j +  0.29387k      0.31877 - 0.063055i +  0.71724j -   1.147
      -2.2588 +   3.0349i +   0.6715j -  0.78728k      -1.3077 +  0.71474i +   1.6302j -   1.068
```

Find the product of the quaternions in each column. The length of the first dimension is `1`, and the length of the second dimension matches `size(A,2)`.

```
B = prod(A)
```

```
B=1×3 quaternion array
    -19.837 -  9.1521i +  15.813j -  19.918k     -5.4708 - 0.28535i +   3.077j -  1.2295k
```

Product of Specified Dimension of Quaternion Array

You can specify which dimension of a quaternion array to take the product of.

Create a 2-by-2-by-2 quaternion array.

```
A = reshape(quaternion(randn(8,4)),2,2,2);
```

Find the product of the elements in each page of the array. The length of the first dimension matches `size(A,1)`, the length of the second dimension matches `size(A,2)`, and the length of the third dimension is `1`.

```
dim = 3;
B = prod(A,dim)

B=2×2 quaternion array
     -2.4847 +  1.1659i - 0.37547j +  2.8068k     0.28786 - 0.29876i - 0.51231j -  4.2972k
     0.38986 -  3.6606i -  2.0474j -   6.047k      -1.741 - 0.26782i +  5.4346j +  4.1452k
```

Input Arguments

quat — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion, specified as scalar, vector, matrix, or multidimensional array of quaternions.

Example: `qProd = prod(quat)` calculates the quaternion product along the first non-singleton dimension of `quat`.

Data Types: `quaternion`

dim — Dimension
first non-singleton dimension (default) | positive integer

Dimension along which to calculate the quaternion product, specified as a positive integer. If `dim` is not specified, `prod` operates along the first non-singleton dimension of `quat`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

quatProd — Quaternion product
positive integer

Quaternion product, returned as quaternion array with one less non-singleton dimension than `quat`.

For example, if `quat` is a 2-by-2-by-5 array,

- `prod(quat,1)` returns a 1-by-2-by-5 array.
- `prod(quat,2)` returns a 2-by-1-by-5 array.
- `prod(quat,3)` returns a 2-by-2 array.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`mtimes, *` | `times, .*`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

rdivide, ./

Element-wise quaternion right division

Syntax

```
C = A./B
```

Description

`C = A./B` performs quaternion element-wise division by dividing each element of quaternion `A` by the corresponding element of quaternion `B`.

Examples

Divide a Quaternion Array by a Real Scalar

Create a 2-by-1 quaternion array, and divide it element-by-element by a real scalar.

```
A = quaternion([1:4;5:8])
```

```
A=2×1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

```
B = 2;
C = A./B
```

```
C=2×1 quaternion array
     0.5 +   1i + 1.5j +   2k
     2.5 +   3i + 3.5j +   4k
```

Divide a Quaternion Array by Another Quaternion Array

Create a 2-by-2 quaternion array, and divide it element-by-element by another 2-by-2 quaternion array.

```
q1 = quaternion(magic(4));
A = reshape(q1,2,2)
```

```
A=2×2 quaternion array
     16 +  2i +  3j + 13k      9 +  7i +  6j + 12k
      5 + 11i + 10j +  8k      4 + 14i + 15j +  1k
```

```
q2 = quaternion([1:4;3:6;2:5;4:7]);
B = reshape(q2,2,2)
```

```
B=2×2 quaternion array
     1 + 2i + 3j + 4k     2 + 3i + 4j + 5k
     3 + 4i + 5j + 6k     4 + 5i + 6j + 7k
```

```
C = A./B
```

```
C=2×2 quaternion array
          2.7 -      0.1i -      2.1j -      1.7k       2.2778 + 0.092593i -  0.46296j -  0.5740
       1.8256 - 0.081395i +  0.45349j -  0.24419k       1.4524 -      0.5i +   1.0238j -   0.261
```

Input Arguments

A — Dividend
scalar | vector | matrix | multidimensional array

Dividend, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

B — Divisor
scalar | vector | matrix | multidimensional array

Divisor, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

Output Arguments

C — Result
scalar | vector | matrix | multidimensional array

Result of quaternion division, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

Algorithms

Quaternion Division

Given a quaternion $A = a_1 + a_2\mathrm{i} + a_3\mathrm{j} + a_4\mathrm{k}$ and a real scalar p,

$$C = A./p = \frac{a_1}{p} + \frac{a_2}{p}\mathrm{i} + \frac{a_3}{p}\mathrm{j} + \frac{a_4}{p}\mathrm{k}$$

Note For a real scalar p, $A./p = A.\backslash p$.

Quaternion Division by a Quaternion Scalar

Given two quaternions A and B of compatible sizes,

$$C = A./B = A.*B^{-1} = A.*\left(\frac{conj(B)}{norm(B)^2}\right)$$

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`conj` | `ldivide, .\` | `norm` | `times, .*`

Objects
`quaternion`

Topics
“Rotations, Orientations, and Quaternions for Automated Driving”

Introduced in R2020a

randrot

Uniformly distributed random rotations

Syntax

```
R = randrot
R = randrot(m)
R = randrot(m1,...,mN)
R = randrot([m1,...,mN])
```

Description

`R = randrot` returns a unit quaternion drawn from a uniform distribution of random rotations.

`R = randrot(m)` returns an `m`-by-`m` matrix of unit quaternions drawn from a uniform distribution of random rotations.

`R = randrot(m1,...,mN)` returns an `m1`-by-...-by-`mN` array of random unit quaternions, where `m1`, ..., `mN` indicate the size of each dimension. For example, `randrot(3,4)` returns a 3-by-4 matrix of random unit quaternions.

`R = randrot([m1,...,mN])` returns an `m1`-by-...-by-`mN` array of random unit quaternions, where `m1`,..., `mN` indicate the size of each dimension. For example, `randrot([3,4])` returns a 3-by-4 matrix of random unit quaternions.

Examples

Matrix of Random Rotations

Generate a 3-by-3 matrix of uniformly distributed random rotations.

```
r = randrot(3)
```

```
r=3×3 quaternion array
      0.17446 +  0.59506i -  0.73295j +  0.27976k      0.69704 - 0.060589i +  0.68679j -  0.1969
      0.21908 -  0.89875i -    0.298j +  0.23548k    -0.049744 +  0.59691i +  0.56459j +  0.5678
       0.6375 +  0.49338i -  0.24049j +  0.54068k       0.2979 -  0.53568i +  0.31819j +  0.7232
```

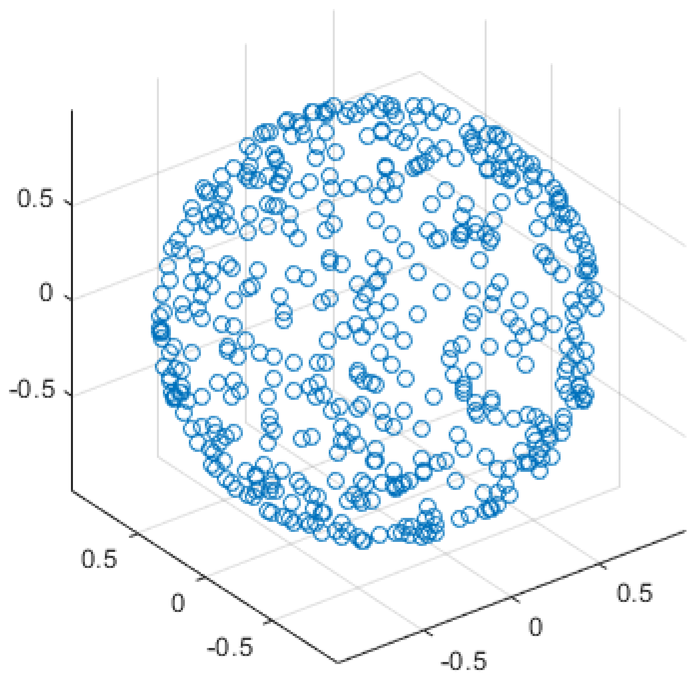
Create Uniform Distribution of Random Rotations

Create a vector of 500 random quaternions. Use rotatepoint on page 4-1088 to visualize the distribution of the random rotations applied to point (1, 0, 0).

```
q = randrot(500,1);
```

```
pt = rotatepoint(q, [1 0 0]);
```

```
figure
scatter3(pt(:,1), pt(:,2), pt(:,3))
axis equal
```



Input Arguments

m — Size of square matrix
integer

Size of square quaternion matrix, specified as an integer value. If m is 0 or negative, then R is returned as an empty matrix.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

m1,...,mN — Size of each dimension
two or more integer values

Size of each dimension, specified as two or more integer values. If the size of any dimension is 0 or negative, then R is returned as an empty array.

Example: randrot(2,3) returns a 2-by-3 matrix of random quaternions.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

[m1,...,mN] — Vector of size of each dimension
row vector of integer values

Vector of size of each dimension, specified as a row vector of two or more integer values. If the size of any dimension is `0` or negative, then `R` is returned as an empty array.

Example: `randrot([2,3])` returns a 2-by-3 matrix of random quaternions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

R — Random quaternions
scalar | vector | matrix | multidimensional array

Random quaternions, returned as a quaternion or array of quaternions.

Data Types: `quaternion`

References

[1] Shoemake, K. "Uniform Random Rotations." *Graphics Gems III* (K. David, ed.). New York: Academic Press, 1992.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

rotateframe

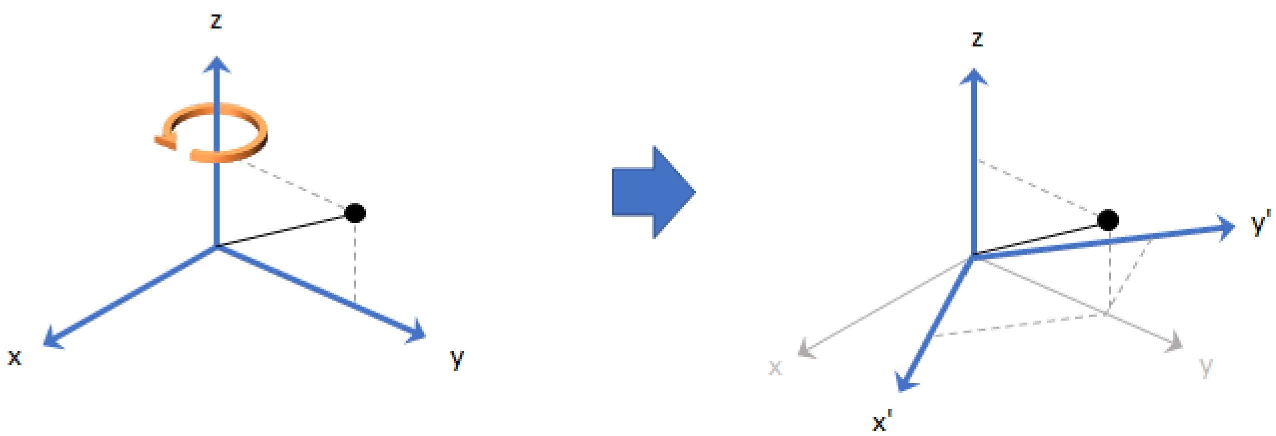
Quaternion frame rotation

Syntax

```
rotationResult = rotateframe(quat,cartesianPoints)
```

Description

`rotationResult = rotateframe(quat,cartesianPoints)` rotates the frame of reference for the Cartesian points using the quaternion, `quat`. The elements of the quaternion are normalized before use in the rotation.

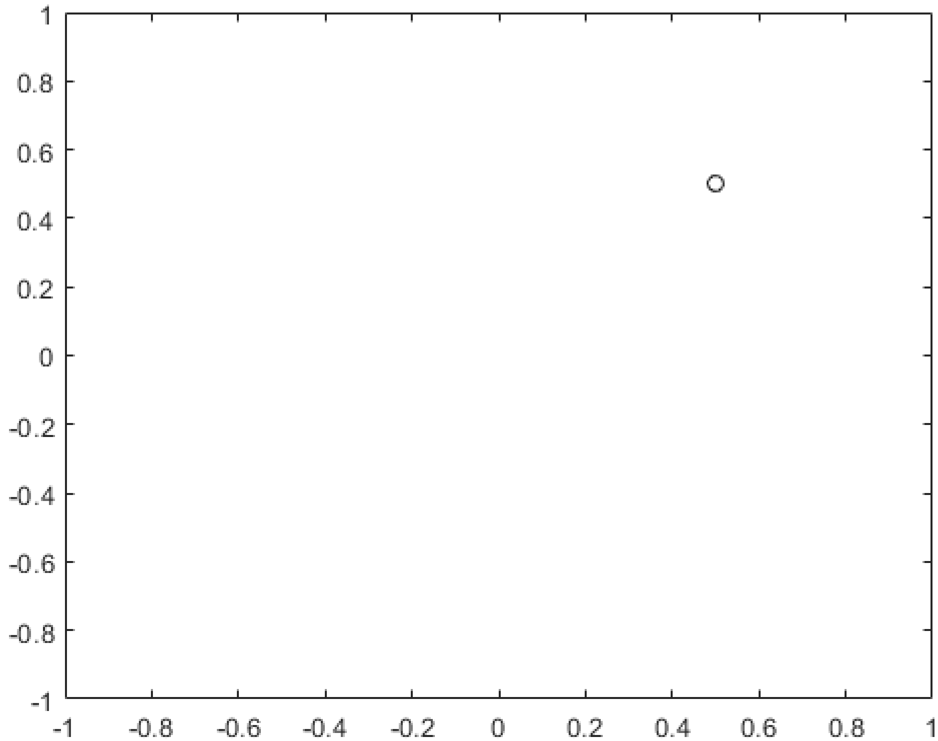


Examples

Rotate Frame Using Quaternion Vector

Define a point in three dimensions. The coordinates of a point are always specified in the order *x*, *y*, and *z*. For convenient visualization, define the point on the *x*-*y* plane.

```
x = 0.5;
y = 0.5;
z = 0;
plot(x,y,'ko')
hold on
axis([-1 1 -1 1])
```



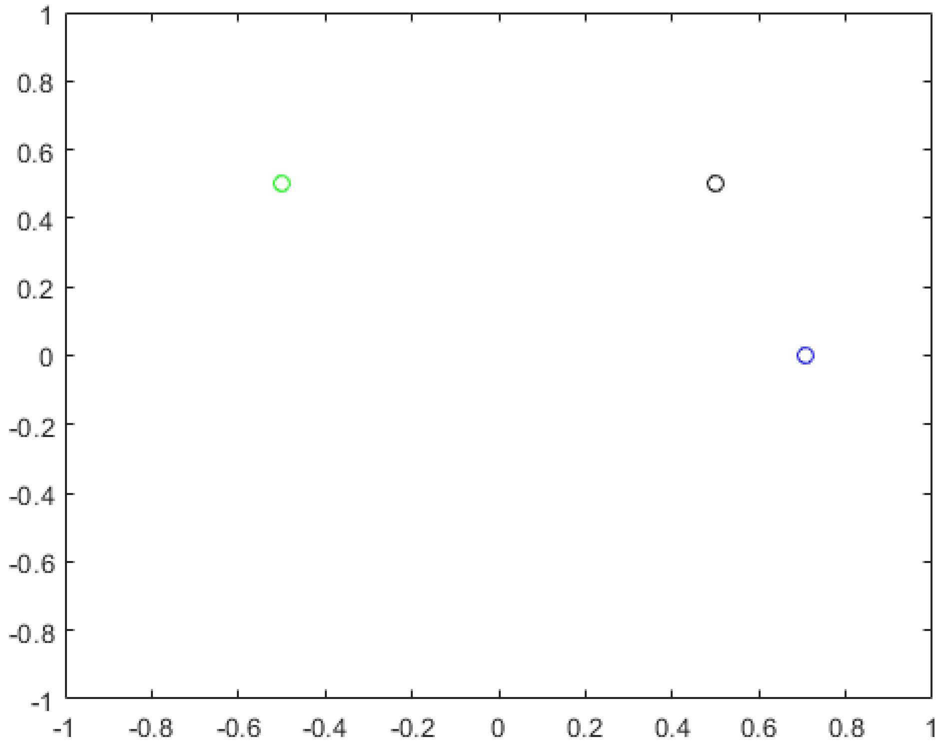
Create a quaternion vector specifying two separate rotations, one to rotate the frame 45 degrees and another to rotate the point -90 degrees about the *z*-axis. Use `rotateframe` to perform the rotations.

```
quat = quaternion([0,0,pi/4; ...
                   0,0,-pi/2],'euler','XYZ','frame');

rereferencedPoint = rotateframe(quat,[x,y,z])
```

```
rereferencedPoint = 2×3

    0.7071   -0.0000         0
   -0.5000    0.5000         0
```

Plot the rereferenced points.

```
plot(rereferencedPoint(1,1),rereferencedPoint(1,2),'bo')
plot(rereferencedPoint(2,1),rereferencedPoint(2,2),'go')
```



Rereference Group of Points using Quaternion

Define two points in three-dimensional space. Define a quaternion to rereference the points by first rotating the reference frame about the *z*-axis 30 degrees and then about the new *y*-axis 45 degrees.

```
a = [1,0,0];
b = [0,1,0];
quat = quaternion([30,45,0],'eulerd','ZYX','point');
```

Use `rotateframe` to reference both points using the quaternion rotation operator. Display the result.

```
rP = rotateframe(quat,[a;b])
```

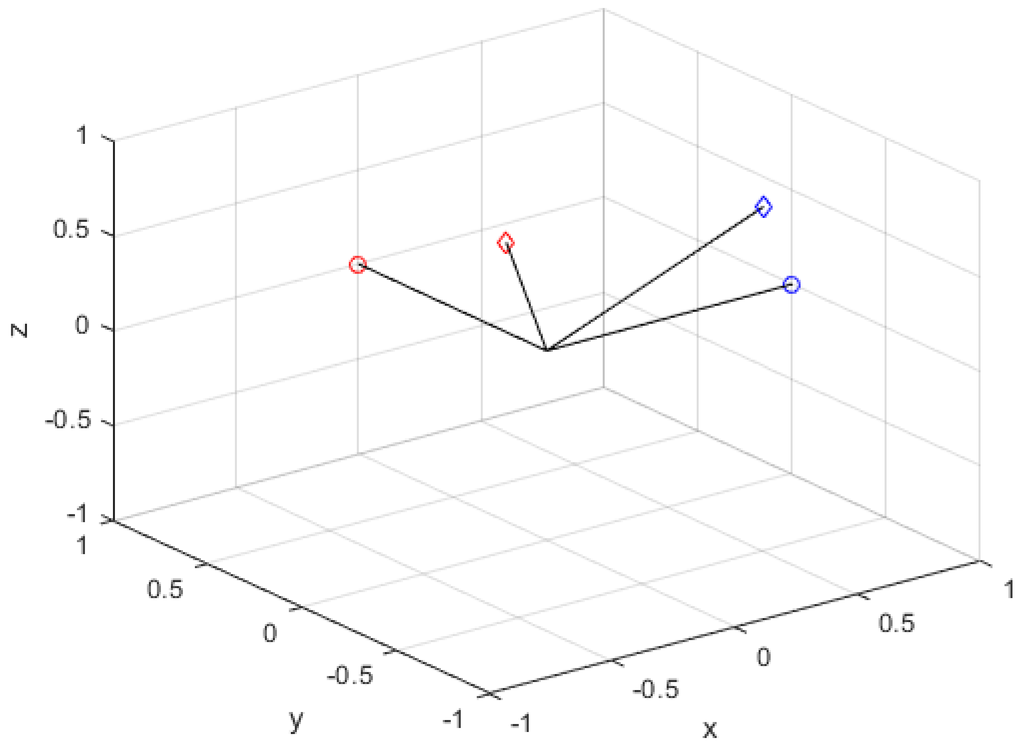
```
rP = 2×3

    0.6124   -0.3536    0.7071
    0.5000    0.8660   -0.0000
```

Visualize the original orientation and the rotated orientation of the points. Draw lines from the origin to each of the points for visualization purposes.

```
plot3(a(1),a(2),a(3),'bo');
```

```
hold on
```

```
grid on
axis([-1 1 -1 1 -1 1])
xlabel('x')
ylabel('y')
zlabel('z')

plot3(b(1),b(2),b(3),'ro');
plot3(rP(1,1),rP(1,2),rP(1,3),'bd')
plot3(rP(2,1),rP(2,2),rP(2,3),'rd')

plot3([0;rP(1,1)],[0;rP(1,2)],[0;rP(1,3)],'k')
plot3([0;rP(2,1)],[0;rP(2,2)],[0;rP(2,3)],'k')
plot3([0;a(1)],[0;a(2)],[0;a(3)],'k')
plot3([0;b(1)],[0;b(2)],[0;b(3)],'k')
```



Input Arguments

quat — Quaternion that defines rotation
scalar | vector

Quaternion that defines rotation, specified as a scalar quaternion or vector of quaternions.

Data Types: `quaternion`

cartesianPoints — Three-dimensional Cartesian points
1-by-3 vector | *N*-by-3 matrix

Three-dimensional Cartesian points, specified as a 1-by-3 vector or *N*-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`rotationResult` — Re-referenced Cartesian points
vector | matrix

Cartesian points defined in reference to rotated reference frame, returned as a vector or matrix the same size as `cartesianPoints`.

The data type of the re-referenced Cartesian points is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

Quaternion frame rotation re-references a point specified in $\mathbf{R}^3$ by rotating the original frame of reference according to a specified quaternion:

$$L_q(u) = q^* u q$$

where q is the quaternion, * represents conjugation, and u is the point to rotate, specified as a quaternion.

For convenience, the `rotateframe` function takes a point in $\mathbf{R}^3$ and returns a point in $\mathbf{R}^3$. Given a function call with some arbitrary quaternion, $q = a + b\text{i} + c\text{j} + d\text{k}$, and arbitrary coordinate, $[x,y,z]$,

```
point = [x,y,z];
rereferencedPoint = rotateframe(q,point)
```

the `rotateframe` function performs the following operations:

1. Converts point $[x,y,z]$ to a quaternion:

   $$u_q = 0 + xi + yj + zk$$

2. Normalizes the quaternion, q:

   $$q_n = \frac{q}{\sqrt{a^2 + b^2 + c^2 + d^2}}$$

3. Applies the rotation:

   $$v_q = q^* u_q q$$

4. Converts the quaternion output, v_q, back to $\mathbf{R}^3$

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`rotatepoint`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

rotatepoint

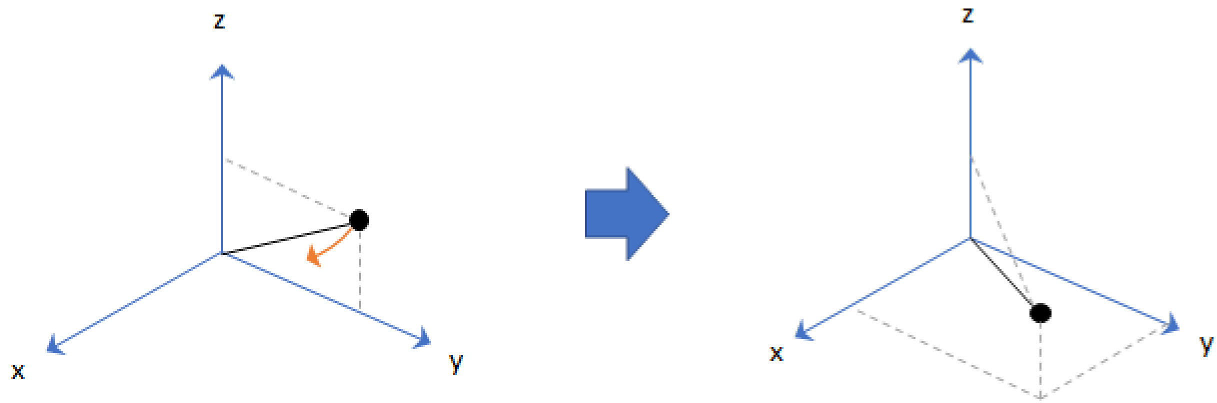
Quaternion point rotation

Syntax

```
rotationResult = rotatepoint(quat,cartesianPoints)
```

Description

`rotationResult = rotatepoint(quat,cartesianPoints)` rotates the Cartesian points using the quaternion, `quat`. The elements of the quaternion are normalized before use in the rotation.

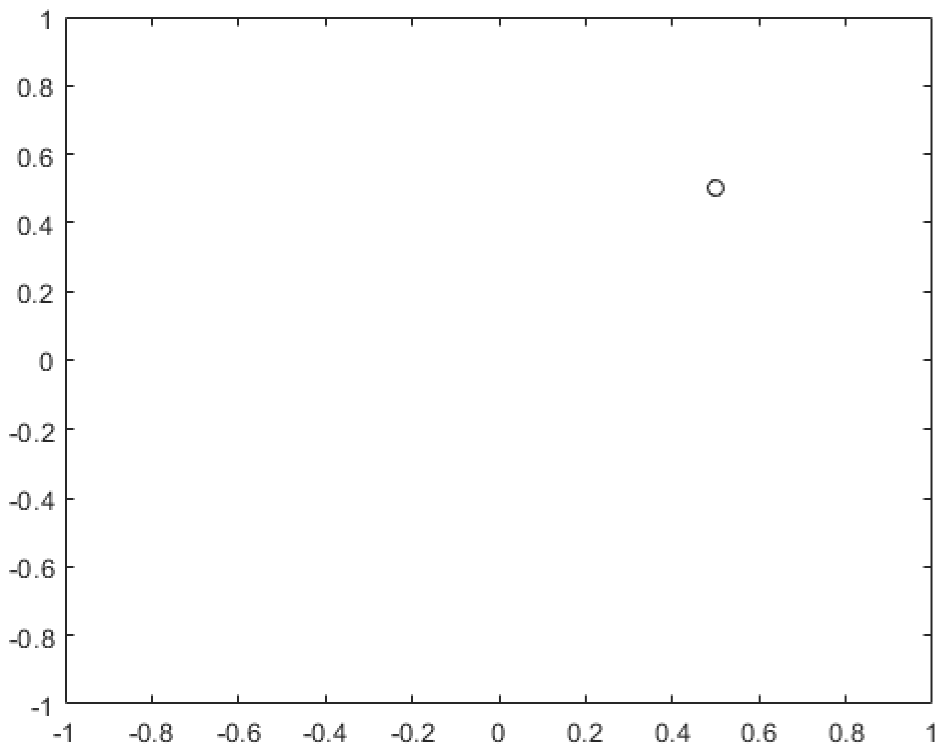


Examples

Rotate Point Using Quaternion Vector

Define a point in three dimensions. The coordinates of a point are always specified in order *x*, *y*, *z*. For convenient visualization, define the point on the *x*-*y* plane.

```
x = 0.5;
y = 0.5;
z = 0;

plot(x,y,'ko')
hold on
axis([-1 1 -1 1])
```



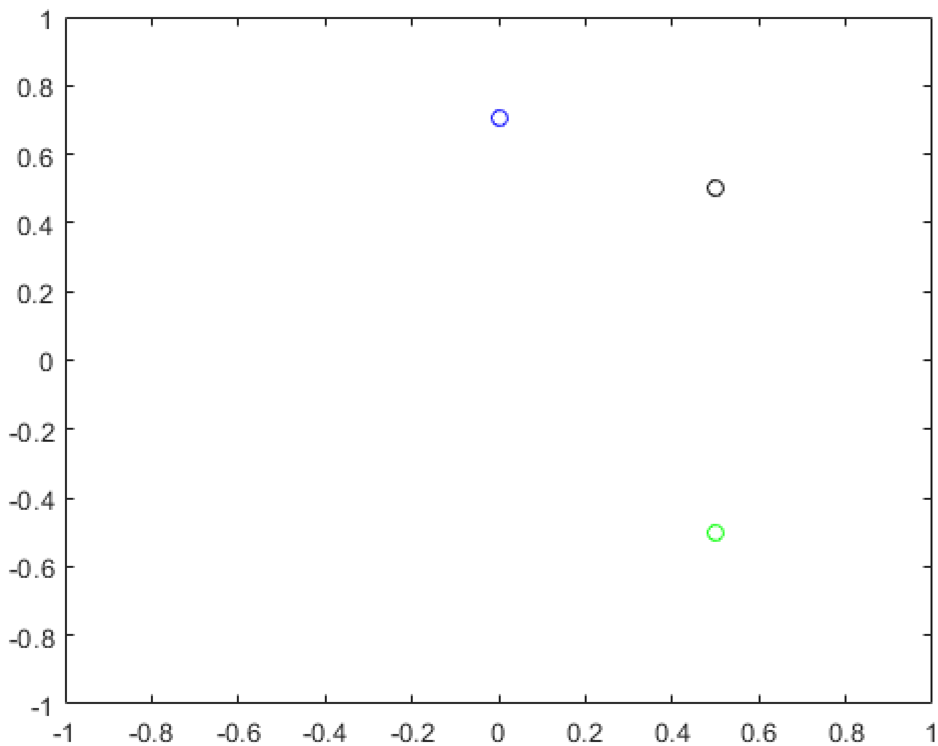
Create a quaternion vector specifying two separate rotations, one to rotate the point 45 and another to rotate the point -90 degrees about the *z*-axis. Use `rotatepoint` to perform the rotation.

```
quat = quaternion([0,0,pi/4; ...
                   0,0,-pi/2],'euler','XYZ','point');

rotatedPoint = rotatepoint(quat,[x,y,z])
```

```
rotatedPoint = 2×3

   -0.0000    0.7071         0
    0.5000   -0.5000         0
```

Plot the rotated points.

```
plot(rotatedPoint(1,1),rotatedPoint(1,2),'bo')
plot(rotatedPoint(2,1),rotatedPoint(2,2),'go')
```



Rotate Group of Points Using Quaternion

Define two points in three-dimensional space. Define a quaternion to rotate the point by first rotating about the *z*-axis 30 degrees and then about the new *y*-axis 45 degrees.

```
a = [1,0,0];
b = [0,1,0];
quat = quaternion([30,45,0],'eulerd','ZYX','point');
```

Use `rotatepoint` to rotate both points using the quaternion rotation operator. Display the result.

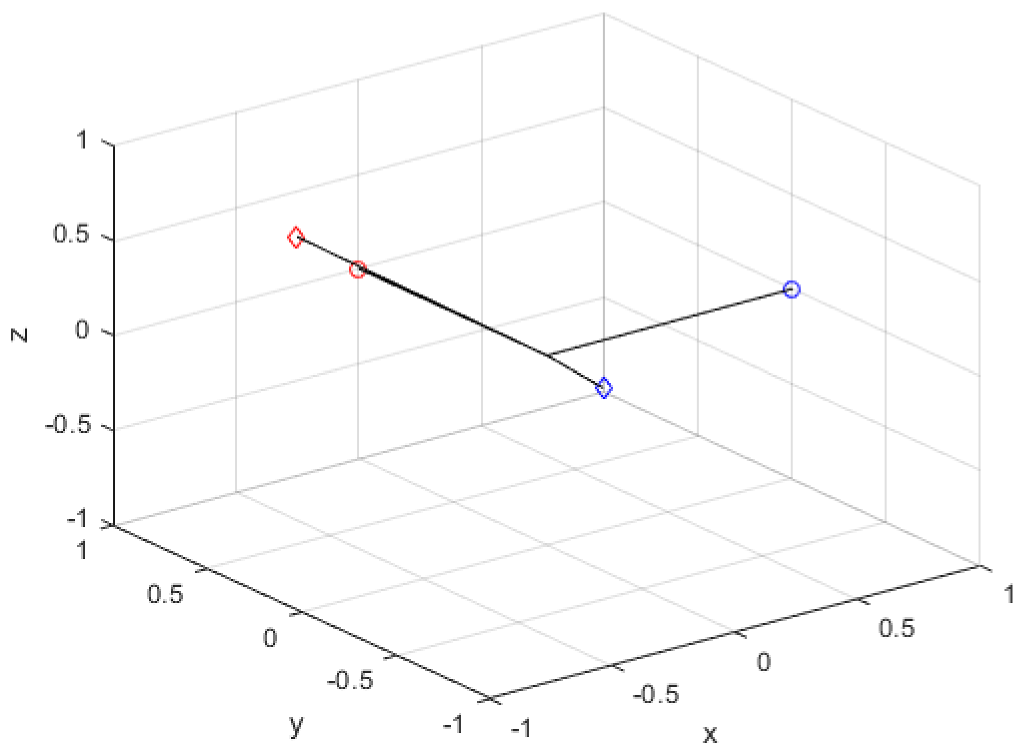
```
rP = rotatepoint(quat,[a;b])
```

```
rP = 2×3

    0.6124    0.5000   -0.6124
   -0.3536    0.8660    0.3536
```

Visualize the original orientation and the rotated orientation of the points. Draw lines from the origin to each of the points for visualization purposes.

```
plot3(a(1),a(2),a(3),'bo');

hold on
```

```
grid on
axis([-1 1 -1 1 -1 1])
xlabel('x')
ylabel('y')
zlabel('z')

plot3(b(1),b(2),b(3),'ro');
plot3(rP(1,1),rP(1,2),rP(1,3),'bd')
plot3(rP(2,1),rP(2,2),rP(2,3),'rd')

plot3([0;rP(1,1)],[0;rP(1,2)],[0;rP(1,3)],'k')
plot3([0;rP(2,1)],[0;rP(2,2)],[0;rP(2,3)],'k')
plot3([0;a(1)],[0;a(2)],[0;a(3)],'k')
plot3([0;b(1)],[0;b(2)],[0;b(3)],'k')
```



Input Arguments

quat — Quaternion that defines rotation
scalar | vector

Quaternion that defines rotation, specified as a scalar quaternion, row vector of quaternions, or column vector of quaternions.

Data Types: `quaternion`

`cartesianPoints` — Three-dimensional Cartesian points
1-by-3 vector | *N*-by-3 matrix

Three-dimensional Cartesian points, specified as a 1-by-3 vector or *N*-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`rotationResult` — Repositioned Cartesian points
vector | matrix

Rotated Cartesian points defined using the quaternion rotation, returned as a vector or matrix the same size as `cartesianPoints`.

Data Types: `single` | `double`

Algorithms

Quaternion point rotation rotates a point specified in $\mathbf{R}^3$ according to a specified quaternion:

$$L_q(u) = quq^*$$

where q is the quaternion, * represents conjugation, and u is the point to rotate, specified as a quaternion.

For convenience, the `rotatepoint` function takes in a point in $\mathbf{R}^3$ and returns a point in $\mathbf{R}^3$. Given a function call with some arbitrary quaternion, $q = a + b\text{i} + c\text{j} + d\text{k}$, and arbitrary coordinate, $[x,y,z]$, for example,

```
rereferencedPoint = rotatepoint(q,[x,y,z])
```

the `rotatepoint` function performs the following operations:

1. Converts point $[x,y,z]$ to a quaternion:

   $$u_q = 0 + xi + yj + zk$$

2. Normalizes the quaternion, q:

   $$q_n = \frac{q}{\sqrt{a^2 + b^2 + c^2 + d^2}}$$

3. Applies the rotation:

   $$v_q = qu_qq^*$$

4. Converts the quaternion output, v_q, back to $\mathbf{R}^3$

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
rotateframe

Objects
quaternion

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

rotmat

Convert quaternion to rotation matrix

Syntax

```
rotationMatrix = rotmat(quat,rotationType)
```

Description

`rotationMatrix = rotmat(quat,rotationType)` converts the quaternion, `quat`, to an equivalent rotation matrix representation.

Examples

Convert Quaternion to Rotation Matrix for Point Rotation

Define a quaternion for use in point rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma],'eulerd','ZYX','point')
```

```
quat = quaternion
       0.8924 +  0.23912i +  0.36964j + 0.099046k
```

Convert the quaternion to a rotation matrix.

```
rotationMatrix = rotmat(quat,'point')
```

```
rotationMatrix = 3×3

    0.7071   -0.0000    0.7071
    0.3536    0.8660   -0.3536
   -0.6124    0.5000    0.6124
```

To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the *y*- and *x*-axes. Multiply the rotation matrices and compare to the output of `rotmat`.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)   0           sind(theta) ; ...
      0             1           0           ; ...
     -sind(theta)   0           cosd(theta)];

rx = [1             0           0           ;      ...
      0             cosd(gamma) -sind(gamma) ;     ...
      0             sind(gamma) cosd(gamma)];

rotationMatrixVerification = rx*ry
```

```
rotationMatrixVerification = 3×3

    0.7071         0    0.7071
    0.3536    0.8660   -0.3536
   -0.6124    0.5000    0.6124
```

Convert Quaternion to Rotation Matrix for Frame Rotation

Define a quaternion for use in frame rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma],'eulerd','ZYX','frame')
```

```
quat = quaternion
       0.8924 +  0.23912i +  0.36964j - 0.099046k
```

Convert the quaternion to a rotation matrix.

```
rotationMatrix = rotmat(quat,'frame')
```

```
rotationMatrix = 3×3

    0.7071   -0.0000   -0.7071
    0.3536    0.8660    0.3536
    0.6124   -0.5000    0.6124
```

To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the *y*- and *x*-axes. Multiply the rotation matrices and compare to the output of `rotmat`.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)   0           -sind(theta) ; ...
      0             1           0            ; ...
     sind(theta)   0           cosd(theta)];

rx = [1             0           0            ;      ...
      0             cosd(gamma) sind(gamma) ;     ...
      0             -sind(gamma) cosd(gamma)];

rotationMatrixVerification = rx*ry
```

```
rotationMatrixVerification = 3×3

    0.7071         0   -0.7071
    0.3536    0.8660    0.3536
    0.6124   -0.5000    0.6124
```

Convert Quaternion Vector to Rotation Matrices

Create a 3-by-1 normalized quaternion vector.

```
qVec = normalize(quaternion(randn(3,4)));
```

Convert the quaternion array to rotation matrices. The pages of `rotmatArray` correspond to the linear index of `qVec`.

```
rotmatArray = rotmat(qVec,'frame');
```

Assume `qVec` and `rotmatArray` correspond to a sequence of rotations. Combine the quaternion rotations into a single representation, then apply the quaternion rotation to arbitrarily initialized Cartesian points.

```
loc = normalize(randn(1,3));
quat = prod(qVec);
rotateframe(quat,loc)
```

```
ans = 1×3

    0.9524    0.5297    0.9013
```

Combine the rotation matrices into a single representation, then apply the rotation matrix to the same initial Cartesian points. Verify the quaternion rotation and rotation matrix result in the same orientation.

```
totalRotMat = eye(3);
for i = 1:size(rotmatArray,3)
    totalRotMat = rotmatArray(:,:,i)*totalRotMat;
end
totalRotMat*loc'
```

```
ans = 3×1

    0.9524
    0.5297
    0.9013
```

Input Arguments

quat — Quaternion to convert
scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

`rotationType` — Type or rotation
`'frame'` | `'point'`

Type of rotation represented by the `rotationMatrix` output, specified as `'frame'` or `'point'`.

Data Types: `char` | `string`

Output Arguments

`rotationMatrix` — Rotation matrix representation

3-by-3 matrix | 3-by-3-by-*N* multidimensional array

Rotation matrix representation, returned as a 3-by-3 matrix or 3-by-3-by-*N* multidimensional array.

- If `quat` is a scalar, `rotationMatrix` is returned as a 3-by-3 matrix.
- If `quat` is non-scalar, `rotationMatrix` is returned as a 3-by-3-by-*N* multidimensional array, where `rotationMatrix(:,:,i)` is the rotation matrix corresponding to `quat(i)`.

The data type of the rotation matrix is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

Given a quaternion of the form

$$q = a + bi + cj + dk\ ,$$

the equivalent rotation matrix for frame rotation is defined as

$$\begin{bmatrix} 2a^2 - 1 + 2b^2 & 2bc + 2ad & 2bd - 2ac \\ 2bc - 2ad & 2a^2 - 1 + 2c^2 & 2cd + 2ab \\ 2bd + 2ac & 2cd - 2ab & 2a^2 - 1 + 2d^2 \end{bmatrix}.$$

The equivalent rotation matrix for point rotation is the transpose of the frame rotation matrix:

$$\begin{bmatrix} 2a^2 - 1 + 2b^2 & 2bc - 2ad & 2bd + 2ac \\ 2bc + 2ad & 2a^2 - 1 + 2c^2 & 2cd - 2ab \\ 2bd - 2ac & 2cd + 2ab & 2a^2 - 1 + 2d^2 \end{bmatrix}.$$

References

[1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`euler` | `eulerd` | `rotvec` | `rotvecd`

Objects

`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

rotvec

Convert quaternion to rotation vector (radians)

Syntax

```
rotationVector = rotvec(quat)
```

Description

`rotationVector = rotvec(quat)` converts the quaternion array, `quat`, to an *N*-by-3 matrix of equivalent rotation vectors in radians. The elements of `quat` are normalized before conversion.

Examples

Convert Quaternion to Rotation Vector in Radians

Convert a random quaternion scalar to a rotation vector in radians

```
quat = quaternion(randn(1,4));
rotvec(quat)
```

```
ans = 1×3

    1.6866   -2.0774    0.7929
```

Input Arguments

`quat` — Quaternion to convert
scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar quaternion, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

`rotationVector` — Rotation vector (radians)
N-by-3 matrix

Rotation vector representation, returned as an *N*-by-3 matrix of rotations vectors, where each row represents the [X Y Z] angles of the rotation vectors in radians. The *i*th row of `rotationVector` corresponds to the element `quat(i)`.

The data type of the rotation vector is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

All rotations in 3-D can be represented by a three-element axis of rotation and a rotation angle, for a total of four elements. If the rotation axis is constrained to be unit length, the rotation angle can be distributed over the vector elements to reduce the representation to three elements.

Recall that a quaternion can be represented in axis-angle form

$$q = \cos\left(\theta/2\right) + \sin\left(\theta/2\right)(x\mathrm{i} + y\mathrm{j} + z\mathrm{k}),$$

where θ is the angle of rotation and $[x,y,z]$ represent the axis of rotation.

Given a quaternion of the form

$$q = a + bi + cj + dk\ ,$$

you can solve for the rotation angle using the axis-angle form of quaternions:

$$\theta = 2\cos^{-1}(a)\ .$$

Assuming a normalized axis, you can rewrite the quaternion as a rotation vector without loss of information by distributing θ over the parts b, c, and d. The rotation vector representation of q is

$$q_{\mathrm{rv}} = \frac{\theta}{\sin\left(\theta/2\right)}[b, c, d]\ .$$

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`euler` | `eulerd` | `rotvecd`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

rotvecd

Convert quaternion to rotation vector (degrees)

Syntax

`rotationVector = rotvecd(quat)`

Description

`rotationVector = rotvecd(quat)` converts the quaternion array, `quat`, to an *N*-by-3 matrix of equivalent rotation vectors in degrees. The elements of `quat` are normalized before conversion.

Examples

Convert Quaternion to Rotation Vector in Degrees

Convert a random quaternion scalar to a rotation vector in degrees.

```
quat = quaternion(randn(1,4));
rotvecd(quat)
```

```
ans = 1×3

   96.6345 -119.0274   45.4312
```

Input Arguments

`quat` — Quaternion to convert
scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

`rotationVector` — Rotation vector (degrees)
N-by-3 matrix

Rotation vector representation, returned as an *N*-by-3 matrix of rotation vectors, where each row represents the [*x y z*] angles of the rotation vectors in degrees. The *i*th row of `rotationVector` corresponds to the element `quat(i)`.

The data type of the rotation vector is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

All rotations in 3-D can be represented by four elements: a three-element axis of rotation and a rotation angle. If the rotation axis is constrained to be unit length, the rotation angle can be distributed over the vector elements to reduce the representation to three elements.

Recall that a quaternion can be represented in axis-angle form

$$q = \cos\left(\theta/2\right) + \sin\left(\theta/2\right)(\mathrm{xi} + y\mathrm{j} + z\mathrm{k}),$$

where θ is the angle of rotation in degrees, and $[x,y,z]$ represent the axis of rotation.

Given a quaternion of the form

$$q = a + bi + cj + dk\,,$$

you can solve for the rotation angle using the axis-angle form of quaternions:

$$\theta = 2\cos^{-1}(a)\,.$$

Assuming a normalized axis, you can rewrite the quaternion as a rotation vector without loss of information by distributing θ over the parts b, c, and d. The rotation vector representation of q is

$$q_{\mathrm{rv}} = \frac{\theta}{\sin\left(\theta/2\right)}[b, c, d]\,.$$

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
euler | eulerd | rotvec

Objects
quaternion

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

slerp

Spherical linear interpolation

Syntax

```
q0 = slerp(q1,q2,T)
```

Description

`q0 = slerp(q1,q2,T)` spherically interpolates between `q1` and `q2` by the interpolation coefficient `T`.

Examples

Interpolate Between Two Quaternions

Create two quaternions with the following interpretation:

1. `a` = 45 degree rotation around the *z*-axis
2. `c` = -45 degree rotation around the *z*-axis

```
a = quaternion([45,0,0],'eulerd','ZYX','frame');
c = quaternion([-45,0,0],'eulerd','ZYX','frame');
```

Call `slerp` with the quaternions `a` and `c` and specify an interpolation coefficient of 0.5.

```
interpolationCoefficient = 0.5;

b = slerp(a,c,interpolationCoefficient);
```

The output of `slerp`, `b`, represents an average rotation of `a` and `c`. To verify, convert `b` to Euler angles in degrees.

```
averageRotation = eulerd(b,'ZYX','frame')
```

```
averageRotation = 1×3

     0     0     0
```

The interpolation coefficient is specified as a normalized value between `0` and `1`, inclusive. An interpolation coefficient of `0` corresponds to the `a` quaternion, and an interpolation coefficient of `1` corresponds to the `c` quaternion. Call `slerp` with coefficients `0` and `1` to confirm.

```
b = slerp(a,c,[0,1]);
eulerd(b,'ZYX','frame')
```

```
ans = 2×3

   45.0000         0         0
```

```
  -45.0000         0         0
```

You can create smooth paths between quaternions by specifying arrays of equally spaced interpolation coefficients.

```
path = 0:0.1:1;

interpolatedQuaternions = slerp(a,c,path);
```

For quaternions that represent rotation only about a single axis, specifying interpolation coefficients as equally spaced results in quaternions equally spaced in Euler angles. Convert `interpolatedQuaternions` to Euler angles and verify that the difference between the angles in the path is constant.

```
k = eulerd(interpolatedQuaternions,'ZYX','frame');
abc = abs(diff(k))
```

```
abc = 10×3

    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
```

Alternatively, you can use the `dist` function to verify that the distance between the interpolated quaternions is consistent. The `dist` function returns angular distance in radians; convert to degrees for easy comparison.

```
def = rad2deg(dist(interpolatedQuaternions(2:end),interpolatedQuaternions(1:end-1)))
```

```
def = 1×10

    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.(
```

SLERP Minimizes Great Circle Path

The SLERP algorithm interpolates along a great circle path connecting two quaternions. This example shows how the SLERP algorithm minimizes the great circle path.

Define three quaternions:

1. `q0` - quaternion indicating no rotation from the global frame
2. `q179` - quaternion indicating a 179 degree rotation about the *z*-axis
3. `q180` - quaternion indicating a 180 degree rotation about the *z*-axis

4 `q181` - quaternion indicating a 181 degree rotation about the *z*-axis

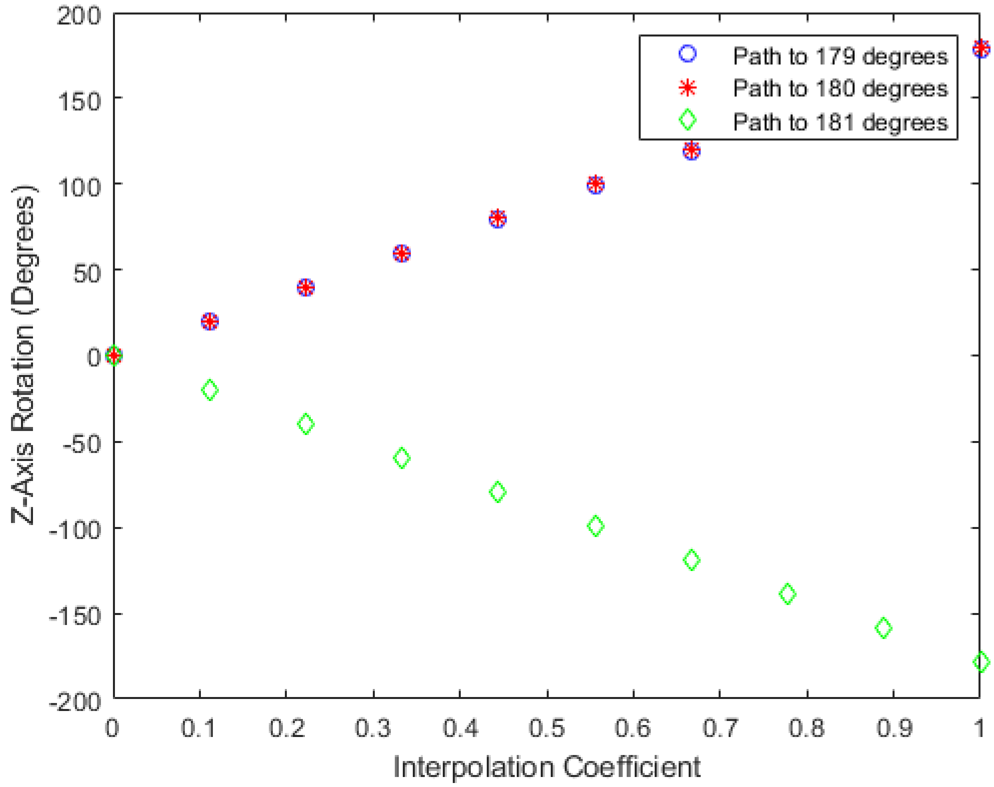
```
q0 = ones(1,'quaternion');

q179 = quaternion([179,0,0],'eulerd','ZYX','frame');

q180 = quaternion([180,0,0],'eulerd','ZYX','frame');

q181 = quaternion([181,0,0],'eulerd','ZYX','frame');
```

Use `slerp` to interpolate between `q0` and the three quaternion rotations. Specify that the paths are traveled in 10 steps.

```
T = linspace(0,1,10);

q179path = slerp(q0,q179,T);
q180path = slerp(q0,q180,T);
q181path = slerp(q0,q181,T);
```

Plot each path in terms of Euler angles in degrees.

```
q179pathEuler = eulerd(q179path,'ZYX','frame');
q180pathEuler = eulerd(q180path,'ZYX','frame');
q181pathEuler = eulerd(q181path,'ZYX','frame');

plot(T,q179pathEuler(:,1),'bo', ...
     T,q180pathEuler(:,1),'r*', ...
     T,q181pathEuler(:,1),'gd');
legend('Path to 179 degrees', ...
       'Path to 180 degrees', ...
       'Path to 181 degrees')
xlabel('Interpolation Coefficient')
ylabel('Z-Axis Rotation (Degrees)')
```



The path between `q0` and `q179` is clockwise to minimize the great circle distance. The path between `q0` and `q181` is counterclockwise to minimize the great circle distance. The path between `q0` and `q180` can be either clockwise or counterclockwise, depending on numerical rounding.

Input Arguments

q1 — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion to interpolate, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

`q1`, `q2`, and `T` must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of them is 1.

Data Types: `quaternion`

q2 — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion to interpolate, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

`q1`, `q2`, and `T` must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of the dimension sizes is 1.

Data Types: `quaternion`

T — Interpolation coefficient
scalar | vector | matrix | multidimensional array

Interpolation coefficient, specified as a scalar, vector, matrix, or multidimensional array of numbers with each element in the range [0,1].

`q1`, `q2`, and `T` must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of the dimension sizes is 1.

Data Types: `single` | `double`

Output Arguments

q0 — Interpolated quaternion
scalar | vector | matrix | multidimensional array

Interpolated quaternion, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Quaternion **s**pherical **l**inear int**erp**olation (SLERP) is an extension of linear interpolation along a plane to spherical interpolation in three dimensions. The algorithm was first proposed in [1]. Given two quaternions, q_1 and q_2, SLERP interpolates a new quaternion, q_0, along the great circle that connects q_1 and q_2. The interpolation coefficient, T, determines how close the output quaternion is to either q_1 and q_2.

The SLERP algorithm can be described in terms of sinusoids:

$$q_0 = \frac{\sin((1-T)\theta)}{\sin(\theta)} q_1 + \frac{\sin(T\theta)}{\sin(\theta)} q_2$$

where q_1 and q_2 are normalized quaternions, and θ is half the angular distance between q_1 and q_2.

References

[1] Shoemake, Ken. "Animating Rotation with Quaternion Curves." *ACM SIGGRAPH Computer Graphics* Vol. 19, Issue 3, 1985, pp. 345–354.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`dist` | `meanrot`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

times, .*

Element-wise quaternion multiplication

Syntax

quatC = A.*B

Description

quatC = A.*B returns the element-by-element quaternion multiplication of quaternion arrays.

You can use quaternion multiplication to compose rotation operators:

- To compose a sequence of frame rotations, multiply the quaternions in the same order as the desired sequence of rotations. For example, to apply a *p* quaternion followed by a *q* quaternion, multiply in the order *pq*. The rotation operator becomes $(pq)^{*}v(pq)$, where *v* represents the object to rotate in quaternion form. * represents conjugation.
- To compose a sequence of point rotations, multiply the quaternions in the reverse order of the desired sequence of rotations. For example, to apply a *p* quaternion followed by a *q* quaternion, multiply in the reverse order, *qp*. The rotation operator becomes $(qp)v(qp)^{*}$.

Examples

Multiply Two Quaternion Vectors

Create two vectors, A and B, and multiply them element by element.

```
A = quaternion([1:4;5:8]);
B = A;
C = A.*B
```

```
C=2×1 quaternion array
     -28 +   4i +   6j +   8k
    -124 +  60i +  70j +  80k
```

Multiply Two Quaternion Arrays

Create two 3-by-3 arrays, A and B, and multiply them element by element.

```
A = reshape(quaternion(randn(9,4)),3,3);
B = reshape(quaternion(randn(9,4)),3,3);
C = A.*B
```

```
C=3×3 quaternion array
     0.60169 +  2.4332i -  2.5844j + 0.51646k    -0.49513 +  1.1722i +  4.4401j -   1.217k      2
     -4.2329 +  2.4547i +  3.7768j + 0.77484k    -0.65232 - 0.43112i -  1.4645j - 0.90073k     -1
```

```
    -4.4159 +  2.1926i +  1.9037j -  4.0303k     -2.0232 +  0.4205i - 0.17288j +  3.8529k     -2
```

Note that quaternion multiplication is not commutative:

```
isequal(C,B.*A)
```

```
ans = logical
   0
```

Multiply Quaternion Row and Column Vectors

Create a row vector `a` and a column vector `b`, then multiply them. The 1-by-3 row vector and 4-by-1 column vector combine to produce a 4-by-3 matrix with all combinations of elements multiplied.

```
a = [zeros('quaternion'),ones('quaternion'),quaternion(randn(1,4))]
```

```
a=1×3 quaternion array
           0 +       0i +       0j +       0k           1 +       0i +       0j +       0k     0
```

```
b = quaternion(randn(4,4))
```

```
b=4×1 quaternion array
      0.31877 +   3.5784i +   0.7254j -  0.12414k
      -1.3077 +   2.7694i - 0.063055j +   1.4897k
     -0.43359 -   1.3499i +  0.71474j +    1.409k
      0.34262 +   3.0349i -  0.20497j +   1.4172k
```

```
a.*b
```

```
ans=4×3 quaternion array
            0 +       0i +        0j +        0k      0.31877 +   3.5784i +   0.7254j -  0.12414
            0 +       0i +        0j +        0k      -1.3077 +   2.7694i - 0.063055j +   1.4897
            0 +       0i +        0j +        0k     -0.43359 -   1.3499i +  0.71474j +    1.409
            0 +       0i +        0j +        0k      0.34262 +   3.0349i -  0.20497j +   1.4172
```

Input Arguments

A — Array to multiply
scalar | vector | matrix | multidimensional array

Array to multiply, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

`A` and `B` must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of them is 1.

Data Types: `quaternion` | `single` | `double`

B — Array to multiply
scalar | vector | matrix | multidimensional array

Array to multiply, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

`A` and `B` must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of them is 1.

Data Types: `quaternion` | `single` | `double`

Output Arguments

`quatC` — Quaternion product
scalar | vector | matrix | multidimensional array

Quaternion product, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Quaternion Multiplication by a Real Scalar

Given a quaternion,

$$q = a_{\mathrm{q}} + b_{\mathrm{q}}\mathrm{i} + c_{\mathrm{q}}\mathrm{j} + d_{\mathrm{q}}\mathrm{k},$$

the product of q and a real scalar β is

$$\beta q = \beta a_{\mathrm{q}} + \beta b_{\mathrm{q}}\mathrm{i} + \beta c_{\mathrm{q}}\mathrm{j} + \beta d_{\mathrm{q}}\mathrm{k}$$

Quaternion Multiplication by a Quaternion Scalar

The definition of the basis elements for quaternions,

$$\mathrm{i}^2 = \mathrm{j}^2 = \mathrm{k}^2 = \mathrm{ijk} = -1\,,$$

can be expanded to populate a table summarizing quaternion basis element multiplication:

	1	**i**	**j**	**k**
1	1	i	j	k
i	i	−1	k	−j
j	j	−k	−1	i
k	k	j	−i	−1

When reading the table, the rows are read first, for example: ij = k and ji = −k.

Given two quaternions, $q = a_{\mathrm{q}} + b_{\mathrm{q}}\mathrm{i} + c_{\mathrm{q}}\mathrm{j} + d_{\mathrm{q}}\mathrm{k}$, and $p = a_{\mathrm{p}} + b_{\mathrm{p}}\mathrm{i} + c_{\mathrm{p}}\mathrm{j} + d_{\mathrm{p}}\mathrm{k}$, the multiplication can be expanded as:

$$
\begin{aligned}
z = pq &= \left(a_\mathrm{p} + b_\mathrm{p}\mathrm{i} + c_\mathrm{p}\mathrm{j} + d_\mathrm{p}\mathrm{k}\right)\left(a_\mathrm{q} + b_\mathrm{q}\mathrm{i} + c_\mathrm{q}\mathrm{j} + d_\mathrm{q}\mathrm{k}\right) \\
&= a_\mathrm{p}a_\mathrm{q} + a_\mathrm{p}b_\mathrm{q}\mathrm{i} + a_\mathrm{p}c_\mathrm{q}\mathrm{j} + a_\mathrm{p}d_\mathrm{q}\mathrm{k} \\
&\quad + b_\mathrm{p}a_\mathrm{q}\mathrm{i} + b_\mathrm{p}b_\mathrm{q}\mathrm{i}^2 + b_\mathrm{p}c_\mathrm{q}\mathrm{ij} + b_\mathrm{p}d_\mathrm{q}\mathrm{ik} \\
&\quad + c_\mathrm{p}a_\mathrm{q}\mathrm{j} + c_\mathrm{p}b_\mathrm{q}\mathrm{ji} + c_\mathrm{p}c_\mathrm{q}\mathrm{j}^2 + c_\mathrm{p}d_\mathrm{q}\mathrm{jk} \\
&\quad + d_\mathrm{p}a_\mathrm{q}k + d_\mathrm{p}b_\mathrm{q}\mathrm{ki} + d_\mathrm{p}c_\mathrm{q}\mathrm{kj} + d_\mathrm{p}d_\mathrm{q}\mathrm{k}^2
\end{aligned}
$$

You can simplify the equation using the quaternion multiplication table.

$$
\begin{aligned}
z = pq &= a_\mathrm{p}a_\mathrm{q} + a_\mathrm{p}b_\mathrm{q}\mathrm{i} + a_\mathrm{p}c_\mathrm{q}\mathrm{j} + a_\mathrm{p}d_\mathrm{q}\mathrm{k} \\
&\quad + b_\mathrm{p}a_\mathrm{q}\mathrm{i} - b_\mathrm{p}b_\mathrm{q} + b_\mathrm{p}c_\mathrm{q}\mathrm{k} - b_\mathrm{p}d_\mathrm{q}\mathrm{j} \\
&\quad + c_\mathrm{p}a_\mathrm{q}\mathrm{j} - c_\mathrm{p}b_\mathrm{q}\mathrm{k} - c_\mathrm{p}c_\mathrm{q} + c_\mathrm{p}d_\mathrm{q}\mathrm{i} \\
&\quad + d_\mathrm{p}a_\mathrm{q}k + d_\mathrm{p}b_\mathrm{q}\mathrm{j} - d_\mathrm{p}c_\mathrm{q}\mathrm{i} - d_\mathrm{p}d_\mathrm{q}
\end{aligned}
$$

References

[1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`mtimes, *` | `prod`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

transpose, .'

Transpose a quaternion array

Syntax

```
Y = quat.'
```

Description

`Y = quat.'` returns the non-conjugate transpose of the quaternion array, `quat`.

Examples

Vector Transpose

Create a vector of quaternions and compute its nonconjugate transpose.

```
quat = quaternion(randn(4,4))
```

```
quat=4×1 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k
       1.8339 -   1.3077i +   2.7694j - 0.063055k
      -2.2588 -  0.43359i -   1.3499j +  0.71474k
      0.86217 +  0.34262i +   3.0349j -  0.20497k
```

```
quatTransposed = quat.'
```

```
quatTransposed=1×4 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k       1.8339 -   1.3077i +   2.7694j - 0.06305
```

Matrix Transpose

Create a matrix of quaternions and compute its nonconjugate transpose.

```
quat = [quaternion(randn(2,4)),quaternion(randn(2,4))]
```

```
quat=2×2 quaternion array
      0.53767 -   2.2588i +  0.31877j -  0.43359k       3.5784 -   1.3499i +   0.7254j +  0.7147
       1.8339 +  0.86217i -   1.3077j +  0.34262k       2.7694 +   3.0349i - 0.063055j -  0.2049
```

```
quatTransposed = quat.'
```

```
quatTransposed=2×2 quaternion array
      0.53767 -   2.2588i +  0.31877j -  0.43359k       1.8339 +  0.86217i -   1.3077j +  0.3426
       3.5784 -   1.3499i +   0.7254j +  0.71474k       2.7694 +   3.0349i - 0.063055j -  0.2049
```

Input Arguments

quat — Quaternion array to transpose
vector | matrix

Quaternion array to transpose, specified as a vector or matrix of quaternions. `transpose` is defined for 1-D and 2-D arrays. For higher-order arrays, use `permute`.

Data Types: `quaternion`

Output Arguments

Y — Transposed quaternion array
vector | matrix

Transposed quaternion array, returned as an *N*-by-*M* array, where `quat` was specified as an *M*-by-*N* array.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`ctranspose, '`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

uminus, -

Quaternion unary minus

Syntax

`mQuat = -quat`

Description

`mQuat = -quat` negates the elements of `quat` and stores the result in `mQuat`.

Examples

Negate Elements of Quaternion Matrix

Unary minus negates each part of a the quaternion. Create a 2-by-2 matrix, `Q`.

```
Q = quaternion(randn(2),randn(2),randn(2),randn(2))
```

```
Q=2×2 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k      -2.2588 -  0.43359i -   1.3499j +  0.7147
       1.8339 -   1.3077i +   2.7694j - 0.063055k      0.86217 +  0.34262i +   3.0349j -  0.2049
```

Negate the parts of each quaternion in `Q`.

```
R = -Q
```

```
R=2×2 quaternion array
     -0.53767 -  0.31877i -   3.5784j -   0.7254k       2.2588 +  0.43359i +   1.3499j -  0.7147
      -1.8339 +   1.3077i -   2.7694j + 0.063055k     -0.86217 -  0.34262i -   3.0349j +  0.2049
```

Input Arguments

`quat` — Quaternion array
scalar | vector | matrix | multidimensional array

Quaternion array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Output Arguments

`mQuat` — Negated quaternion array
scalar | vector | matrix | multidimensional array

Negated quaternion array, returned as the same size as `quat`.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`minus, -`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

zeros

Create quaternion array with all parts set to zero

Syntax

```
quatZeros = zeros('quaternion')
quatZeros = zeros(n,'quaternion')
quatZeros = zeros(sz,'quaternion')
quatZeros = zeros(sz1,...,szN,'quaternion')

quatZeros = zeros( ___ ,'like',prototype,'quaternion')
```

Description

`quatZeros = zeros('quaternion')` returns a scalar quaternion with all parts set to zero.

`quatZeros = zeros(n,'quaternion')` returns an `n`-by-`n` matrix of quaternions.

`quatZeros = zeros(sz,'quaternion')` returns an array of quaternions where the size vector, `sz`, defines `size(quatZeros)`.

`quatZeros = zeros(sz1,...,szN,'quaternion')` returns a `sz1`-by-...-by-`szN` array of quaternions where `sz1,…,szN` indicates the size of each dimension.

`quatZeros = zeros( ___ ,'like',prototype,'quaternion')` specifies the underlying class of the returned quaternion array to be the same as the underlying class of the quaternion prototype.

Examples

Quaternion Scalar Zero

Create a quaternion scalar zero.

```
quatZeros = zeros('quaternion')
```

```
quatZeros = quaternion
     0 + 0i + 0j + 0k
```

Square Matrix of Quaternions

Create an n-by-n array of quaternion zeros.

```
n = 3;
quatZeros = zeros(n,'quaternion')
```

```
quatZeros=3×3 quaternion array
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
```

```
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
```

Multidimensional Array of Quaternion Zeros

Create a multidimensional array of quaternion zeros by defining array dimensions in order. In this example, you create a 3-by-1-by-2 array. You can specify dimensions using a row vector or comma-separated integers.

Specify the dimensions using a row vector and display the results:

```
dims = [3,1,2];
quatZerosSyntax1 = zeros(dims,'quaternion')
```

```
quatZerosSyntax1 = 3x1x2 quaternion array
quatZerosSyntax1(:,:,1) =

     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k


quatZerosSyntax1(:,:,2) =

     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k
```

Specify the dimensions using comma-separated integers, and then verify the equivalence of the two syntaxes:

```
quatZerosSyntax2 = zeros(3,1,2,'quaternion');
isequal(quatZerosSyntax1,quatZerosSyntax2)
```

```
ans = logical
   1
```

Underlying Class of Quaternion Zeros

A quaternion is a four-part hyper-complex number used in three-dimensional representations. You can specify the underlying data type of the parts as `single` or `double`. The default is `double`.

Create a quaternion array of zeros with the underlying data type set to `single`.

```
quatZeros = zeros(2,'like',single(1),'quaternion')
```

```
quatZeros=2×2 quaternion array
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
```

Verify the underlying class using the `classUnderlying` function.

```
classUnderlying(quatZeros)
```

```
ans =
'single'
```

Input Arguments

n — Size of square quaternion matrix
integer value

Size of square quaternion matrix, specified as an integer value. If `n` is `0` or negative, then `quatZeros` is returned as an empty matrix.

Example: `zeros(4,'quaternion')` returns a 4-by-4 matrix of quaternion zeros.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz — Output size
row vector of integer values

Output size, specified as a row vector of integer values. Each element of `sz` indicates the size of the corresponding dimension in `quatZeros`. If the size of any dimension is `0` or negative, then `quatZeros` is returned as an empty array.

Example: `zeros([1,4,2],'quaternion')` returns a 1-by-4-by-2 array of quaternion zeros.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

prototype — Quaternion prototype
variable

Quaternion prototype, specified as a variable.

Example: `zeros(2,'like',quat,'quaternion')` returns a 2-by-2 matrix of quaternions with the same underlying class as the prototype quaternion, `quat`.

Data Types: `quaternion`

sz1,...,szN — Size of each dimension
two or more integer values

Size of each dimension, specified as two or more integers.

- If the size of any dimension is `0`, then `quatZeros` is returned as an empty array.
- If the size of any dimension is negative, then it is treated as `0`.

Example: `zeros(2,3,'quaternion')` returns a 2-by-3 matrix of quaternion zeros.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

quatZeros — Quaternion zeros
scalar | vector | matrix | multidimensional array

Quaternion zeros, returned as a quaternion or array of quaternions.

Given a quaternion of the form $Q = a + b\text{i} + c\text{j} + d\text{k}$, a quaternion zero is defined as $Q = 0 + 0\text{i} + 0\text{j} + 0\text{k}$.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`ones`

Objects
`quaternion`

Topics
"Rotations, Orientations, and Quaternions for Automated Driving"

Introduced in R2020a

trackHistoryLogic

Confirm and delete tracks based on recent track history

Description

The `trackHistoryLogic` object determines if a track should be confirmed or deleted based on the track history. A track should be confirmed if there are at least *Mc* hits in the recent *Nc* updates. A track should be deleted if there are at least *Md* misses in the recent *Nd* updates.

The confirmation and deletion decisions contribute to the track management by a `multiObjectTracker` object.

Creation

Syntax

```
logic = trackHistoryLogic
logic = trackHistoryLogic(Name,Value,...)
```

Description

`logic = trackHistoryLogic` creates a `trackHistoryLogic` object with default confirmation and deletion thresholds.

`logic = trackHistoryLogic(Name,Value,...)` specifies the properties of the track history logic object using one or more `Name,Value` pair arguments. Any unspecified properties take default values.

Properties

`ConfirmationThreshold` — Confirmation threshold
`[2 3]` (default) | positive integer scalar | 2-element vector of positive integers

Confirmation threshold, specified as a positive integer scalar or 2-element vector of positive integers. If the logic score is above this threshold, the track is confirmed. `ConfirmationThreshold` has the form [*Mc Nc*], where *Mc* is the number of hits required for confirmation in the recent *Nc* updates. When specified as a scalar, then *Mc* and *Nc* have the same value.

Example: `[3 5]`

Data Types: `single` | `double`

`DeletionThreshold` — Deletion threshold
`[6 6]` (default) | positive integer scalar | 2-element vector of positive integers

Deletion threshold, specified as a positive integer scalar or 2-element vector of positive integers. If the logic score is above this threshold, the track is deleted. `DeletionThreshold` has the form [*Md Nd*], where *Md* is the number of misses required for deletion in the recent *Nd* updates. When specified as a scalar, then *Md* and *Nd* have the same value.

Example: [5 5]

Data Types: single | double

History — Track history
logical vector

This property is read-only.

Track history, specified as a logical vector of length *N*, where *N* is the larger of the second element in the ConfirmationThreshold and the second element in the DeletionThreshold. The first element is the most recent update. A true value indicates a hit and a false value indicates a miss.

Object Functions

init	Initialize track logic with first hit
hit	Update track logic with subsequent hit
miss	Update track logic with miss
checkConfirmation	Check if track should be confirmed
checkDeletion	Check if track should be deleted
output	Get current state of track logic
reset	Reset state of track logic
clone	Create copy of track logic

Examples

Create and Update History-Based Logic

Create a history-based logic. Specify confirmation threshold values *Mc* and *Nc* as the vector [3 5]. Specify deletion threshold values *Md* and *Nd* as the vector [6 7].

```
historyLogic = trackHistoryLogic('ConfirmationThreshold',[3 5], ...
    'DeletionThreshold',[6 7])
```

```
historyLogic = 
  trackHistoryLogic with properties:

    ConfirmationThreshold: [3 5]
        DeletionThreshold: [6 7]
                  History: [0 0 0 0 0 0 0]
```

Initialize the logic, which records a hit as the first update to the logic.

```
init(historyLogic)
history = historyLogic.History;
disp(['History: [',num2str(history),'].']);
```

```
History: [1  0  0  0  0  0  0].
```

Update the logic four more times, where only the odd updates register a hit. The confirmation flag is true by the end of the fifth update, because three hits (*Mc*) are counted in the most recent five updates (*Nc*).

```
for i = 2:5
    isOdd = logical(mod(i,2));
```

```
    if isOdd
        hit(historyLogic)
    else
        miss(historyLogic)
    end

    history = historyLogic.History;
    confFlag = checkConfirmation(historyLogic);
    delFlag = checkDeletion(historyLogic,true,i);
    disp(['History: [',num2str(history),']. Confirmation Flag: ',num2str(confFlag), ...
        '. Deletion Flag: ',num2str(delFlag)']);
end
```

```
History: [0  1  0  0  0  0  0]. Confirmation Flag: 0. Deletion Flag: 0
History: [1  0  1  0  0  0  0]. Confirmation Flag: 0. Deletion Flag: 0
History: [0  1  0  1  0  0  0]. Confirmation Flag: 0. Deletion Flag: 0
History: [1  0  1  0  1  0  0]. Confirmation Flag: 1. Deletion Flag: 0
```

Update the logic with a miss six times. The deletion flag is `true` by the end of the fifth update, because six misses (*Md*) are counted in the most recent seven updates (*Nd*).

```
for i = 1:6
    miss(historyLogic);

    history = historyLogic.History;
    confFlag = checkConfirmation(historyLogic);
    delFlag = checkDeletion(historyLogic);
    disp(['History: [',num2str(history),']. Confirmation Flag: ',num2str(confFlag), ...
        '. Deletion Flag: ',num2str(delFlag)']);
end
```

```
History: [0  1  0  1  0  1  0]. Confirmation Flag: 0. Deletion Flag: 0
History: [0  0  1  0  1  0  1]. Confirmation Flag: 0. Deletion Flag: 0
History: [0  0  0  1  0  1  0]. Confirmation Flag: 0. Deletion Flag: 0
History: [0  0  0  0  1  0  1]. Confirmation Flag: 0. Deletion Flag: 0
History: [0  0  0  0  0  1  0]. Confirmation Flag: 0. Deletion Flag: 1
History: [0  0  0  0  0  0  1]. Confirmation Flag: 0. Deletion Flag: 1
```

References

[1] Blackman, S., and R. Popoli. *Design and Analysis of Modern Tracking Systems*. Boston, MA: Artech House, 1999.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`multiObjectTracker`

Introduced in R2020a

checkConfirmation

Check if track should be confirmed

Syntax

```
tf = checkConfirmation(historyLogic)
```

Description

`tf = checkConfirmation(historyLogic)` returns a flag that is `true` when at least *Mc* out of *Nc* recent updates of the track history logic object `historyLogic` are `true`.

Examples

Check Confirmation of History-Based Logic

Create a history-based logic. Specify confirmation threshold values *Mc* and *Nc* as the vector [2 3]. Specify deletion threshold values *Md* and *Nd* as the vector [3 3].

```
historyLogic = trackHistoryLogic('ConfirmationThreshold',[2 3], ...
    'DeletionThreshold',[3 3])
```

```
historyLogic = 
  trackHistoryLogic with properties:

    ConfirmationThreshold: [2 3]
        DeletionThreshold: [3 3]
                  History: [0 0 0]
```

Initialize the logic, which records a hit as the first update to the logic. The confirmation flag is `false` because the number of hits is less than two (*Mc*).

```
init(historyLogic)
history = output(historyLogic);
confFlag = checkConfirmation(historyLogic);
disp(['History: [',num2str(history),']. Confirmation Flag: ',num2str(confFlag)]);
```

```
History: [1  0  0]. Confirmation Flag: 0
```

Update the logic with a hit. The confirmation flag is `true` because two hits (*Mc*) are counted in the most recent three updates (*Nc*).

```
hit(historyLogic)
history = output(historyLogic);
confFlag = checkConfirmation(historyLogic);
disp(['History: [',num2str(history),']. Confirmation Flag: ',num2str(confFlag)]);
```

```
History: [1  1  0]. Confirmation Flag: 1
```

Input Arguments

historyLogic — Track history logic
trackHistoryLogic

Track history logic, specified as a trackHistoryLogic object.

Output Arguments

tf — Track should be confirmed
true | false

Track should be confirmed, returned as true or false.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
trackHistoryLogic

Introduced in R2020a

checkDeletion

Check if track should be deleted

Syntax

```
tf = checkDeletion(historyLogic)
tf = checkDeletion(historyLogic,tentativeTrack,age)
```

Description

`tf = checkDeletion(historyLogic)` returns a flag that is `true` when at least *Md* out of *Nd* recent updates of the track history logic object `historyLogic` are `false`.

`tf = checkDeletion(historyLogic,tentativeTrack,age)` returns a flag that is `true` when the track is tentative and there are not enough detections to allow it to confirm. Use the logical flag `tentativeTrack` to indicate if the track is tentative and provide `age` as a numeric scalar.

Examples

Check Deletion of History-Based Logic

Create a history-based logic. Specify confirmation threshold values *Mc* and *Nc* as the vector [2 3]. Specify deletion threshold values *Md* and *Nd* as the vector [4 5].

```
historyLogic = trackHistoryLogic('ConfirmationThreshold',[2 3], ...
    'DeletionThreshold',[4 5])
```

```
historyLogic = 
  trackHistoryLogic with properties:

    ConfirmationThreshold: [2 3]
        DeletionThreshold: [4 5]
                  History: [0 0 0 0 0]
```

Initialize the logic, which records a hit as the first update to the logic. The confirmation flag is `false` because the number of hits is less than two (*Mc*).

```
init(historyLogic)
history = output(historyLogic);
checkConfirmation(historyLogic)
```

```
ans = logical
   0
```

```
delFlag = checkDeletion(historyLogic);
disp(['History: [',num2str(history),']. Deletion Flag: ',num2str(delFlag)]);
```

```
History: [1  0  0  0  0]. Deletion Flag: 1
```

Update the logic with a hit. The confirmation flag is `true` because two hits (*Mc*) are counted in the most recent three updates (*Nc*).

```
hit(historyLogic)
history = output(historyLogic);
checkConfirmation(historyLogic)
```

```
ans = logical
   1
```

```
delFlag = checkDeletion(historyLogic);
disp(['History: [',num2str(history),']. Deletion Flag: ',num2str(delFlag)]);
```

```
History: [1  1  0  0  0]. Deletion Flag: 0
```

```
miss(historyLogic)
history = output(historyLogic);
checkConfirmation(historyLogic)
```

```
ans = logical
   1
```

```
delFlag = checkDeletion(historyLogic);
disp(['History: [',num2str(history),']. Deletion Flag: ',num2str(delFlag)]);
```

```
History: [0  1  1  0  0]. Deletion Flag: 0
```

```
miss(historyLogic)
history = output(historyLogic);
delFlag = checkDeletion(historyLogic);
checkConfirmation(historyLogic)
```

```
ans = logical
   0
```

```
disp(['History: [',num2str(history),']. Deletion Flag: ',num2str(delFlag)]);
```

```
History: [0  0  1  1  0]. Deletion Flag: 0
```

Check Deletion of Tentative Track

Create a history-based logic. Specify confirmation threshold values *Mc* and *Nc* as the vector [2 3]. Specify deletion threshold values *Md* and *Nd* as the vector [4 5].

```
historyLogic = trackHistoryLogic('ConfirmationThreshold',[2 3], ...
    'DeletionThreshold',5)
```

```
historyLogic = 
  trackHistoryLogic with properties:

    ConfirmationThreshold: [2 3]
        DeletionThreshold: [5 5]
                  History: [0 0 0 0 0]
```

Initialize the logic, which records a hit as the first update to the logic. Then, record two misses.

```
init(historyLogic)
miss(historyLogic)
miss(historyLogic)
history = output(historyLogic)
```

```
history = 1x5 logical array

   0   0   1   0   0
```

The confirmation flag is `false` because the number of hits in the most recent 3 updates (*Nc*) is less than 2 (*Mc*).

```
confirmationFlag = checkConfirmation(historyLogic)
```

```
confirmationFlag = logical
   0
```

Check the deletion flag as if the track were not tentative. The deletion flag is `false` because the number of misses in the most recent 5 updates (*Nm*) is less than 4 (*Mc*).

```
deletionFlag = checkDeletion(historyLogic)
```

```
deletionFlag = logical
   0
```

Recheck the deletion flag, treating the track as tentative with an age of 3. The tentative deletion flag is `true` because there are not enough detections to allow the track to confirm.

```
tentativeDeletionFlag = checkDeletion(historyLogic,true,3)
```

```
tentativeDeletionFlag = logical
   1
```

Input Arguments

`historyLogic` — Track history logic
`trackHistoryLogic`

Track history logic, specified as a `trackHistoryLogic` object.

`tentativeTrack` — Track is tentative
`false` | `true`

Track is tentative, specified as `false` or `true`. Use `tentativeTrack` to indicate if the track is tentative.

age — Number of updates
numeric scalar

Number of updates since track initialization, specified as a numeric scalar.

Output Arguments

`tf` — Track can be deleted
`true` | `false`

Track can be deleted, returned as `true` or `false`.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`trackHistoryLogic`

Introduced in R2020a

clone

Create copy of track logic

Syntax

```
clonedLogic = clone(logic)
```

Description

`clonedLogic = clone(logic)` returns a copy of the current track logic object, `logic`.

Examples

Clone Track History Logic

Create a history-based logic. Specify confirmation threshold values *Mc* and *Nc* as the vector [3 5]. Specify deletion threshold values *Md* and *Nd* as the vector [6 7].

```
historyLogic = trackHistoryLogic('ConfirmationThreshold',[3 5], ...
    'DeletionThreshold',[6 7])
```

```
historyLogic = 
  trackHistoryLogic with properties:

    ConfirmationThreshold: [3 5]
        DeletionThreshold: [6 7]
                  History: [0 0 0 0 0 0 0]
```

Initialize the logic, which records a hit as the first update to the logic.

```
init(historyLogic)
```

Update the logic four more times, where only the odd updates register a hit.

```
for i = 2:5
    isOdd = logical(mod(i,2));
    if isOdd
        hit(historyLogic)
    else
        miss(historyLogic)
    end
end
```

Get the current state of the logic.

```
history = output(historyLogic)
```

```
history = 1x7 logical array

   1   0   1   0   1   0   0
```

Create a copy of the logic. The clone has the same confirmation threshold, deletion threshold, and history as the original history logic.

```
clonedLogic = clone(historyLogic)
```

```
clonedLogic = 
  trackHistoryLogic with properties:

    ConfirmationThreshold: [3 5]
        DeletionThreshold: [6 7]
                  History: [1 0 1 0 1 0 0]
```

Input Arguments

logic — Track history logic
trackHistoryLogic object

Track history logic, specified as a trackHistoryLogic object.

Output Arguments

clonedLogic — Cloned track logic
trackHistoryLogic object

Cloned track logic, returned as a trackHistoryLogic object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

trackHistoryLogic

Introduced in R2020a

hit

Update track logic with subsequent hit

Syntax

```
hit(historyLogic)
```

Description

`hit(historyLogic)` updates the track history with a hit.

Examples

Update History Logic with Hit

Create a history-based logic with the default confirmation and deletion thresholds.

```
historyLogic = trackHistoryLogic;
```

Initialize the logic, which records a hit as the first update to the logic. The first element of the 'History' property, which indicates the most recent update, is 1.

```
init(historyLogic)
history = historyLogic.History;
disp(['History: [',num2str(history),'].']);
```

```
History: [1  0  0  0  0  0].
```

Update the logic with a hit. The first two elements of the 'History' property are 1.

```
hit(historyLogic)
history = historyLogic.History;
disp(['History: [',num2str(history),'].']);
```

```
History: [1  1  0  0  0  0].
```

Input Arguments

`historyLogic` — Track history logic
`trackHistoryLogic`

Track history logic, specified as a `trackHistoryLogic` object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`trackHistoryLogic`

Introduced in R2020a

init

Initialize track logic with first hit

Syntax

`init(historyLogic)`

Description

`init(historyLogic)` initializes the track history logic with the first hit.

Examples

Initialize History-Based Logic

Create a history-based logic with default confirmation and deletion thresholds.

```
historyLogic = trackHistoryLogic
```

```
historyLogic = 
  trackHistoryLogic with properties:

    ConfirmationThreshold: [2 3]
        DeletionThreshold: [6 6]
                  History: [0 0 0 0 0 0]
```

Initialize the logic, which records a hit as the first update to the logic.

```
init(historyLogic)
history = historyLogic.History;
disp(['History: [',num2str(history),'].']);
```

```
History: [1  0  0  0  0  0].
```

Input Arguments

`historyLogic` — Track history logic
`trackHistoryLogic` object

Track history logic, specified as a `trackHistoryLogic` object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`trackHistoryLogic`

Introduced in R2020a

miss

Update track logic with miss

Syntax

```
miss(historyLogic)
```

Description

`miss(historyLogic)` updates the track history with a miss.

Examples

Update History Logic with Miss

Create a history-based logic with the default confirmation and deletion thresholds.

```
historyLogic = trackHistoryLogic;
```

Initialize the logic, which records a hit as the first update to the logic. The first element of the 'History' property, which indicates the most recent update, is 1.

```
init(historyLogic)
history = historyLogic.History;
disp(['History: [',num2str(history),'].']);
```

```
History: [1  0  0  0  0  0].
```

Update the logic with a miss. The first element of the 'History' property is `0`.

```
miss(historyLogic)
history = historyLogic.History;
disp(['History: [',num2str(history),'].']);
```

```
History: [0  1  0  0  0  0].
```

Input Arguments

`historyLogic` — Track history logic
`trackHistoryLogic`

Track history logic, specified as a `trackHistoryLogic` object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`trackHistoryLogic`

Introduced in R2020a

output

Get current state of track logic

Syntax

```
history = output(historyLogic)
```

Description

`history = output(historyLogic)` returns the recent history updates of the track history logic object, `historyLogic`.

Examples

Get Recent History of History-Based Logic

Create a history-based logic. Specify confirmation threshold values *Mc* and *Nc* as the vector [3 5]. Specify deletion threshold values *Md* and *Nd* as the vector [6 7].

```
historyLogic = trackHistoryLogic('ConfirmationThreshold',[3 5], ...
    'DeletionThreshold',[6 7]);
```

Get the recent history of the logic. The history vector has a length of 7, which is the greater of *Nc* and *Nd*. All values are `0` because the logic is not initialized.

```
h = output(historyLogic)
```

```
h = 1x7 logical array

   0   0   0   0   0   0   0
```

Initialize the logic, then get the recent history of the logic. The first element, which indicates the most recent update, is 1.

```
init(historyLogic);
h = output(historyLogic)
```

```
h = 1x7 logical array

   1   0   0   0   0   0   0
```

Update the logic with a hit, then get the recent history of the logic.

```
hit(historyLogic);
h = output(historyLogic)
```

```
h = 1x7 logical array

   1   1   0   0   0   0   0
```

Input Arguments

historyLogic — Track history logic
trackHistoryLogic

Track history logic, specified as a trackHistoryLogic object.

Output Arguments

history — Recent history
logical vector

Recent track history of historyLogic, returned as a logical vector. The length of the vector is the same as the length of the History property of the historyLogic. The first element is the most recent update. A true value indicates a hit and a false value indicates a miss.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

trackHistoryLogic

Introduced in R2020a

reset

Reset state of track logic

Syntax

```
reset(logic)
```

Description

`reset(logic)` resets the track logic object, `logic`.

Examples

Reset Track History Logic

Create a history-based logic using the default confirmation threshold and deletion threshold. Get the current state of the logic. The current and maximum score are both `0`.

```
historyLogic = trackHistoryLogic;
history = output(historyLogic)
```

```
history = 1x6 logical array

   0   0   0   0   0   0
```

Initialize the logic, then get the current state of the logic.

```
volume = 1.3;
beta = 0.1;
init(historyLogic);
history = output(historyLogic)
```

```
history = 1x6 logical array

   1   0   0   0   0   0
```

Reset the logic, then get the current state of the logic.

```
reset(historyLogic)
history = output(historyLogic)
```

```
history = 1x6 logical array

   0   0   0   0   0   0
```

Input Arguments

`logic` — Track history logic
`trackHistoryLogic` object

Track history logic, specified as a `trackHistoryLogic` object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`trackHistoryLogic`

Introduced in R2020a

labelDefinitionCreatorMultisignal

Object for storing, modifying, and creating label definitions table for multisignal workflow

Description

The `labelDefinitionCreatorMultisignal` object stores definitions of labels, sublabels, and attributes to label ground truth data for a multisignal workflow. Use "Object Functions" on page 4-1143 to add, remove, modify, or display label definitions. Use the `create` object function to create a label definitions table from the `labelDefinitionCreatorMultisignal` object. You can use this label definitions table with the **Ground Truth Labeler** app.

Creation

Syntax

```
ldc = labelDefinitionCreatorMultisignal
ldc = labelDefinitionCreatorMultisignal(labelDefs)
```

Description

`ldc = labelDefinitionCreatorMultisignal` creates an empty label definition creator object `ldc` for a multisignal workflow. Add label definitions to this object using "Object Functions" on page 4-1143. Use the `info` function to inspect details of stored labels, sublabels, and attributes.

`ldc = labelDefinitionCreatorMultisignal(labelDefs)` creates a label definition creator object `ldc` for a multisignal workflow and stores definitions from the label definitions table `labelDefs`. Use "Object Functions" on page 4-1143 to add new label definitions or modify the existing label definitions. Use the `info` function to inspect details of stored labels, sublabels, and attributes.

Input Arguments

`labelDefs` — Label definitions
table

Label definitions, specified as a table with up to eight columns. The possible columns are *Name*, *SignalType*, *LabelType*, *Group*, *Description*, *LabelColor*, *PixelLabelID*, and *Hierarchy*. This table specifies the definitions of labels, sublabels, and attributes for labeling ground truth data. For more details, see LabelDefinitions property of `groundTruthMultisignal` object.

Output Arguments

`ldc` — Label definition creator for multisignal workflow
`labelDefinitionCreatorMultisignal` object

Label definition creator for the multisignal workflow, returned as a `labelDefinitionCreatorMultisignal` object that contains information about label definitions associated with ground truth data.

Object Functions

addLabel	Add label to label definition creator object for multisignal workflow
addSublabel	Add sublabel to label in label definition creator object for multisignal workflow
addAttribute	Add attributes to label or sublabel in label definition creator object for multisignal workflow
removeLabel	Remove label from label definition creator object for multisignal workflow
removeSublabel	Remove sublabel from label in label definition creator object for multisignal workflow
removeAttribute	Remove attribute from label or sublabel in label definition creator object for multisignal workflow
editLabelGroup	Modify label group name in label definition creator object for multisignal workflow
editGroupName	Change group name in label definition creator object for multisignal workflow
editLabelDescription	Modify label or sublabel description in label definition creator object for multisignal workflow
editAttributeDescription	Modify attribute description in label definition creator object for multisignal workflow
create	Create label definitions table from label definition creator object for multisignal workflow
info	Display label, sublabel, or attribute information stored in label definition creator object for multisignal workflow

Examples

Create Label Definition Creator Object for Multisignal Workflow and Add Label Definitions

Create an empty `labelDefinitionCreatorMultisignal` object.

```
ldc = labelDefinitionCreatorMultisignal
```

```
ldc = 
labelDefinitionCreatorMultisignal
```

Add a label with the name `'Vehicle'`. Specify the type as `'Rectangle'`. Adding a `'Rectangle'` also adds a `'Cuboid'` entry to the label definitions table.

```
addLabel(ldc,'Vehicle','Rectangle')
```

Add an attribute with the name `'Color'` to the label `'Vehicle'`. Specify the attribute type as a string with the value `'Red'`.

```
addAttribute(ldc,'Vehicle','Color',attributeType.String,'Red')
```

Add a sublabel with the name `'Wheel'` to the label `'Vehicle'`. Specify the type of the sublabel as `'Rectangle'`.

```
addSublabel(ldc,'Vehicle','Wheel','Rectangle')
```

Add an attribute called `'Diameter'` to the sublabel `'Wheel'`. Specify the attribute value as a `'Numeric'` scalar.

```
addAttribute(ldc,'Vehicle/Wheel','Diameter','Numeric',14)
```

Display the details of the updated `labelDefinitionCreatorMultisignal` object.

```
ldc
```

```
ldc = 
labelDefinitionCreatorMultisignal contains the following labels:

    Vehicle with 1 sublabels and 1 attributes and belongs to None group.    (info)

For more details about attributes and sublabels, use the info method.
```

Create a label definitions table from the definitions stored in the object.

```
labelDefs = create(ldc)
```

```
labelDefs=2×7 table
       Name         SignalType     LabelType      Group       Description     LabelColor     Hierarchy
    ___________    __________    _________    ________    ___________    __________    _________

    {'Vehicle'}    Image         Rectangle    {'None'}       {' '}       {0x0 char}    {1x1 struc
    {'Vehicle'}    PointCloud    Cuboid       {'None'}       {' '}       {0x0 char}    {1x1 struc
```

Create Label Definition Creator Object for Multisignal Workflow from Existing Label Definitions Table

Load an existing multisignal label definitions table into the workspace.

```
labelDefFile = fullfile(toolboxdir('driving'),'drivingdata','labelDefsMultiSignal.mat');
ld = load(labelDefFile)
```

```
ld = struct with fields:
    labelDefs: [6x6 table]
```

Create a `labelDefinitionCreatorMultisignal` object from the label definitions table.

```
ldc = labelDefinitionCreatorMultisignal(ld.labelDefs)
```

```
ldc = 
labelDefinitionCreatorMultisignal contains the following labels:

    Car with 0 sublabels and 0 attributes and belongs to None group.    (info)
    LeftLane with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Road with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Sunny with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Urban with 0 sublabels and 0 attributes and belongs to None group.    (info)

For more details about attributes and sublabels, use the info method.
```

Add a new attribute to the label `'Car'`.

```
addAttribute(ldc,'Car','Color','List',{'Red','Green','Blue'})
```

Display the details of the updated `labelDefinitionCreatorMultisignal` object.

```
ldc
```

```
ldc = 
labelDefinitionCreatorMultisignal contains the following labels:
```

```
    Car with 0 sublabels and 1 attributes and belongs to None group.    (info)
    LeftLane with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Road with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Sunny with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Urban with 0 sublabels and 0 attributes and belongs to None group.    (info)

For more details about attributes and sublabels, use the info method.
```

See Also

Apps
Ground Truth Labeler

Objects
`attributeType` | `groundTruthMultisignal` | `labelType`

Introduced in R2020a

addAttribute

Add attributes to label or sublabel in label definition creator object for multisignal workflow

Syntax

```
addAttribute(ldc,labelName,attributeName,typeOfAttribute,attributeDefault)
addAttribute( ___ ,Name,Value)
```

Description

`addAttribute(ldc,labelName,attributeName,typeOfAttribute,attributeDefault)` adds an attribute with specified name and type to the indicated label or sublabel. The attribute is added under the hierarchy for the specified label or sublabel in the `labelDefinitionCreatorMultisignal` object `ldc`.

`addAttribute( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Add Attributes to Label and Sublabel in Label Definition Creator Object for Multisignal Workflow

Create an empty `labelDefinitionCreatorMultisignal` object.

```
ldc = labelDefinitionCreatorMultisignal;
```

Add a label with the name `'Car'`. Specify the type of label as `'Rectangle'`. Adding a `'Rectangle'` also adds a `'Cuboid'` entry to the label definitions table.

```
addLabel(ldc,'Car','Rectangle');
```

Add an attribute `'Color'` to the label `'Car'`. Specify the attribute type as `'String'` with the value `'Red'`.

```
addAttribute(ldc,'Car','Color','String','Red')
```

Add a label with the name `'TrafficLight'`. Specify the type of the label as `'Rectangle'`. Add a description to the label.

```
addLabel(ldc,'TrafficLight','Rectangle','Description','Bounding boxes for stop signs');
```

Add a sublabel with the name `'RedLight'` to the label `'TrafficLight'`. Specify the type of the sublabel as `'Rectangle'`.

```
addSublabel(ldc,'TrafficLight','RedLight','Rectangle');
```

Add an attribute `'isOn'` to the sublabel `'RedLight'` in the label `'TrafficLight'`. Specify the attribute type for the sublabel as `'logical'` with the value `false`.

```
addAttribute(ldc,'TrafficLight/RedLight','isOn','logical',false);
```

Display the details of the updated `labelDefinitionCreatorMultisignal` object.

```
ldc
```

```
ldc = 
labelDefinitionCreatorMultisignal contains the following labels:

    Car with 0 sublabels and 1 attributes and belongs to None group.    (info)
    TrafficLight with 1 sublabels and 0 attributes and belongs to None group.    (info)

For more details about attributes and sublabels, use the info method.
```

Display information about the attribute under the label `'Car'` using the object function `info`.

```
info(ldc,'Car')
```

```
           Name: "Car"
     SignalType: Image
      LabelType: Rectangle
          Group: "None"
     LabelColor: {''}
     Attributes: "Color"
      Sublabels: []
    Description: ' '

           Name: "Car"
     SignalType: PointCloud
      LabelType: Cuboid
          Group: "None"
     LabelColor: {''}
     Attributes: "Color"
      Sublabels: []
    Description: ' '
```

Display information about the attribute under the label `'TrafficLight'` using the object function `info`.

```
info(ldc,'TrafficLight')
```

```
           Name: "TrafficLight"
     SignalType: Image
      LabelType: Rectangle
          Group: "None"
     LabelColor: {''}
     Attributes: []
      Sublabels: "RedLight"
    Description: 'Bounding boxes for stop signs'

           Name: "TrafficLight"
     SignalType: PointCloud
      LabelType: Cuboid
          Group: "None"
     LabelColor: {''}
     Attributes: []
      Sublabels: "RedLight"
    Description: 'Bounding boxes for stop signs'
```

Display information about the attribute under the sublabel `'RedLight'` in the label `'TrafficLight'` using the object function `info`.

```
info(ldc,'TrafficLight/RedLight')
```

```
          Name: "RedLight"
          Type: Rectangle
    LabelColor: ''
    Attributes: "isOn"
     Sublabels: []
   Description: ' '
```

Display information about the attribute `'isOn'` under the sublabel `'RedLight'` in the label `'TrafficLight'` using the object function `info`.

```
info(ldc,'TrafficLight/RedLight/isOn')
```

```
            Name: "isOn"
            Type: Logical
    DefaultValue: 0
     Description: ' '
```

Input Arguments

`ldc` — Label definition creator for multisignal workflow
`labelDefinitionCreatorMultisignal` object

Label definition creator for the multisignal workflow, specified as a `labelDefinitionCreatorMultisignal` object.

`labelName` — Label or sublabel name
character vector | string scalar

Label or sublabel name, specified as a character vector or string scalar that uniquely identifies the label or sublabel to which the attribute is to be added.

- To specify a label, use the form '*labelName*'.

Example: `addAttribute(ldc,'Car','Color')`

- To specify a sublabel, use the form '*labelName/sublabelName*'. In this case, the attribute associates with the sublabel.

Example: `addAttribute(ldc,'TrafficLight/RedLight','isOn')`

`attributeName` — Attribute name
character vector | string scalar

Attribute name, specified as a character vector or string scalar that identifies the attribute to be added to the label or sublabel.

`typeOfAttribute` — Type of attribute
`attributeType` enumeration | character vector | string scalar

Type of attribute, specified as one of these values:

- `attributeType` enumeration — The type of the attribute must be one of these `attributeType` enumerators: `Numeric`, `Logical`, `String`, or `List`.

Example: `addAttribute(ldc,'Car','Color',attributeType.String,'Red');`

- Character vector or string scalar — This value must partially or fully match one of the enumerators in the `attributeType` enumeration.

Example: `addAttribute(ldc,'Car','Color','Str','Red');`

`attributeDefault` — Default value of attribute
numeric scalar | logical scalar | character vector | string scalar | cell array of character vectors | cell array of string scalars

Default value of the attribute, specified as one of these:

- Numeric scalar — Specify this value when `typeOfAttribute` is `Numeric`.
- Logical scalar — Specify this value when `typeOfAttribute` is `Logical`.
- Character vector or string scalar — Specify this value when `typeOfAttribute` is `String`.
- Cell array of character vectors or cell array of string scalars — Specify this value when `typeOfAttribute` is `List`. The first entry in the cell array is the default value.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `addAttribute(ldc,'Car/Wheel','Outsidediameter',attributeType.Numeric,740,'Description','Outside diameter in mm');`

`Description` — Attribute description
`' '` (default) | character vector | string scalar

Attribute description, specified as a comma-separated pair consisting of `'Description'` and a character vector or string scalar. Use this name-value pair to describe the attribute.

See Also

Objects
`attributeType` | `labelDefinitionCreatorMultisignal`

Functions
`addLabel` | `addSublabel` | `editAttributeDescription` | `removeAttribute`

Introduced in R2020a

addLabel

Add label to label definition creator object for multisignal workflow

Syntax

```
addLabel(ldc,labelName,typeOfLabel)
addLabel( ___ ,Name,Value)
```

Description

`addLabel(ldc,labelName,typeOfLabel)` adds a label with the specified name and type to the `labelDefinitionCreatorMultisignal` object `ldc`.

`addLabel( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Add Label Using Label Definition Creator for Multisignal Workflow

Create an empty `labelDefinitionCreatorMultisignal` object.

```
ldc = labelDefinitionCreatorMultisignal;
```

Add a label named `'Car'`. Specify the type of label as `'Cuboid'`. Adding a `'Cuboid'` also adds a `'Rectangle'` entry to the label definitions table.

```
addLabel(ldc,'Car','Cuboid');
```

Add another label named `'StopSign'` in a group named `'TrafficSign'`. Specify the type of label as a `'Rectangle'`. Adding `'Rectangle'` also adds a `'Cuboid'` entry to the label definitions table. Add a description to the label.

```
addLabel(ldc,'StopSign','Rectangle','Group','TrafficSign','Description','Bounding boxes for stop
```

Display the details of the updated `labelDefinitionCreatorMultisignal` object.

```
ldc
```

```
ldc = 
labelDefinitionCreatorMultisignal contains the following labels:

    Car with 0 sublabels and 0 attributes and belongs to None group.    (info)
    StopSign with 0 sublabels and 0 attributes and belongs to TrafficSign group.    (info)

For more details about attributes and sublabels, use the info method.
```

Display information about the label `'Car'` using the object function `info`.

```
info(ldc,'Car')
```

```
       Name: "Car"
 SignalType: Image
  LabelType: Rectangle
      Group: "None"
 LabelColor: {''}
 Attributes: []
  Sublabels: []
Description: ' '

       Name: "Car"
 SignalType: PointCloud
  LabelType: Cuboid
      Group: "None"
 LabelColor: {''}
 Attributes: []
  Sublabels: []
Description: ' '
```

Display information about the label `'StopSign'` using the object function `info`.

```
info(ldc,'StopSign')
```

```
       Name: "StopSign"
 SignalType: Image
  LabelType: Rectangle
      Group: "TrafficSign"
 LabelColor: {''}
 Attributes: []
  Sublabels: []
Description: 'Bounding boxes for stop signs'

       Name: "StopSign"
 SignalType: PointCloud
  LabelType: Cuboid
      Group: "TrafficSign"
 LabelColor: {''}
 Attributes: []
  Sublabels: []
Description: 'Bounding boxes for stop signs'
```

Input Arguments

`ldc` — Label definition creator for multisignal workflow

`labelDefinitionCreatorMultisignal` object

Label definition creator for the multisignal workflow, specified as a `labelDefinitionCreatorMultisignal` object.

`labelName` — Label name

character vector | string scalar

Label name, specified as a character vector or string scalar that uniquely identifies the label to be added.

`typeOfLabel` — Type of label

`labelType` enumeration | character vector | string scalar

Type of label, specified as one of these values:

- `labelType` enumeration — You can use any of these `labelType` enumerators to specify the type of label: `Cuboid`, `Rectangle`, `Line`, `PixelLabel`, `Scene`, or `Custom`.

  Note Adding a `Cuboid` or `Rectangle` also adds a `Rectangle` or `Cuboid` entry, respectively, to the label definitions table.

Example: `addLabel(ldc,'Car',labelType.Cuboid);`

- Character vector or string scalar — This value must partially or fully match one of the `labelType` enumerators.

Example: `addLabel(ldc,'Car','Cub');`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`addLabel(ldc,'StopSign','Rectangle','Group','TrafficSign','Description','Bounding boxes for stop signs');`

`Group` — Group name

`'None'` (default) | character vector | string scalar

Group name, specified as a comma-separated pair consisting of `'Group'` and a character vector or string scalar. Use this name-value pair to specify a name for a group of labels.

`Description` — Label description

`' '` (default) | character vector | string scalar

Label description, specified as a comma-separated pair consisting of `'Description'` and a character vector or string scalar. Use this name-value pair to describe the label.

See Also

Objects
`labelDefinitionCreatorMultisignal` | `labelType`

Functions
`addAttribute` | `addSublabel` | `editLabelDescription` | `removeLabel`

Introduced in R2020a

addSublabel

Add sublabel to label in label definition creator object for multisignal workflow

Syntax

```
addSublabel(ldc,labelName,sublabelName,typeOfSublabel)
addSublabel( ___ ,Name,Value)
```

Description

`addSublabel(ldc,labelName,sublabelName,typeOfSublabel)` adds a sublabel with the specified name and type to the indicated label. The sublabel is added under the hierarchy for the specified label in the `labelDefinitionCreatorMultisignal` object `ldc`.

`addSublabel( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Add Sublabels to Labels in Label Definition Creator Object for Multisignal Workflow

Create an empty `labelDefinitionCreatorMultisignal` object.

```
ldc = labelDefinitionCreatorMultisignal;
```

Add a label with the name `'Vehicle'`. Specify the type as `'Rectangle'`. Adding a `'Rectangle'` also adds a `'Cuboid'` entry to the label definitions table.

```
addLabel(ldc,'Vehicle','Rectangle');
```

Add a sublabel with the name `'Wheel'` to the label `'Vehicle'`. Specify the type of the sublabel as `'Rectangle'`. Add a description to the sublabel.

```
addSublabel(ldc,'Vehicle','Wheel','rect','Description','Bounding boxes for wheel');
```

Display the details of the updated `labelDefinitionCreatorMultisignal` object.

```
ldc
```

```
ldc = 
labelDefinitionCreatorMultisignal contains the following labels:

    Vehicle with 1 sublabels and 0 attributes and belongs to None group.    (info)

For more details about attributes and sublabels, use the info method.
```

Display information about the label `'Vehicle'` using the object function `info`.

```
info(ldc,'Vehicle')
```

```
          Name: "Vehicle"
    SignalType: Image
```

```
      LabelType: Rectangle
          Group: "None"
     LabelColor: {''}
     Attributes: []
      Sublabels: "Wheel"
    Description: ' '

           Name: "Vehicle"
     SignalType: PointCloud
      LabelType: Cuboid
          Group: "None"
     LabelColor: {''}
     Attributes: []
      Sublabels: "Wheel"
    Description: ' '
```

Display information about the sublabel `'Wheel'` in the label `'Vehicle'` using the object function `info`.

```
info(ldc,'Vehicle/Wheel')
```

```
           Name: "Wheel"
           Type: Rectangle
     LabelColor: ''
     Attributes: []
      Sublabels: []
    Description: 'Bounding boxes for wheel'
```

Add another label with the name `'TrafficLight'`. Specify the type as `'Rectangle'`. Add a description to the label.

```
addLabel(ldc,'TrafficLight','Rectangle','Description','Bounding boxes for traffic light');
```

Add sublabels called `'RedLight'` and `'GreenLight'` to the label `'TrafficLight'`. Specify the type of the sublabels as `'Rectangle'`.

```
addSublabel(ldc,'TrafficLight','RedLight','Rectangle');
addSublabel(ldc,'TrafficLight','GreenLight','Rectangle');
```

Display the details of the updated `labelDefinitionCreatorMultisignal` object.

```
ldc
```

```
ldc =
labelDefinitionCreatorMultisignal contains the following labels:

    Vehicle with 1 sublabels and 0 attributes and belongs to None group.    (info)
    TrafficLight with 2 sublabels and 0 attributes and belongs to None group.    (info)

For more details about attributes and sublabels, use the info method.
```

Display information about the label `'TrafficLight'` using the object function `info`.

```
info(ldc,'TrafficLight')
```

```
           Name: "TrafficLight"
     SignalType: Image
      LabelType: Rectangle
          Group: "None"
```

```
     LabelColor: {''}
     Attributes: []
      Sublabels: ["RedLight"    "GreenLight"]
    Description: 'Bounding boxes for traffic light'

           Name: "TrafficLight"
     SignalType: PointCloud
      LabelType: Cuboid
          Group: "None"
     LabelColor: {''}
     Attributes: []
      Sublabels: ["RedLight"    "GreenLight"]
    Description: 'Bounding boxes for traffic light'
```

Input Arguments

`ldc` — Label definition creator for multisignal workflow
`labelDefinitionCreatorMultisignal` object

Label definition creator for the multisignal workflow, specified as a `labelDefinitionCreatorMultisignal` object.

`labelName` — Label name
character vector | string scalar

Label name, specified as a character vector or string scalar that uniquely identifies the label with which the sublabel is associated.

`sublabelName` — Sublabel name
character vector | string scalar

Sublabel name, specified as a character vector or string scalar that identifies the sublabel to be added.

`typeOfSublabel` — Type of sublabel
`labelType` enumeration | character vector | string scalar

Type of sublabel, specified as one of these values:

- `labelType` enumeration — The type of the sublabel must be one of these `labelType` enumerators: `Rectangle` or `Line`.

Example: `addSublabel(ldc,'Car','Wheel',labelType.Rectangle);`

- Character vector or string scalar — This value must partially or fully match one of these `labelType` enumerators: `Rectangle` or `Line`.

Example: `addSublabel(ldc,'Car','Wheel','Rec');`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `addSublabel(ldc,'Car','Wheel','Rec','Description','Bounding box for Wheel');`

`Description` — Sublabel description
`' '` (default) | character vector | string scalar

Sublabel description, specified as a comma-separated pair consisting of `'Description'` and a character vector or string scalar. Use this name-value pair to describe the sublabel.

See Also

Objects
`labelDefinitionCreatorMultisignal` | `labelType`

Functions
`addAttribute` | `addLabel` | `removeSublabel`

Introduced in R2020a

create

Create label definitions table from label definition creator object for multisignal workflow

Syntax

```
labelDefs = create(ldc)
```

Description

`labelDefs = create(ldc)` creates a label definitions table, `labelDefs`, from the `labelDefinitionCreatorMultisignal` object `ldc`. You can import the `labelDefs` table into the **Ground Truth Labeler** app to label ground truth data.

Examples

Create Label Definitions Table from Label Definition Creator Object for Multisignal Workflow

Create an empty `labelDefinitionCreatorMultisignal` object.

```
ldc = labelDefinitionCreatorMultisignal;
```

Add a label called `'Vehicle'`. Specify the label type as `'Rectangle'` and add a description to the label.

```
addLabel(ldc,'Vehicle','Rectangle','Description','Bounding box for the vehicle. Use this label fo
```

Add an attribute `'IsCar'` to the label `'Vehicle'`. Specify the attribute type as `'logical'` with value the `true` and add a description for the attribute.

```
addAttribute(ldc,'Vehicle','IsCar','logical',true,'Description','Type of vehicle')
```

Add an attribute `'IsBus'` to the label `'Vehicle'`. Specify the attribute type as `'logical'` with value the `false` and add a description for the attribute.

```
addAttribute(ldc,'Vehicle','IsBus','logical',false,'Description','Type of vehicle')
```

Create a label definitions table from the definitions stored in the object.

```
labelDefs = create(ldc)
```

```
labelDefs=2×7 table
        Name         SignalType     LabelType      Group                                              Descriptio
    ____________    __________    __________    _________    ________________________________________________

    {'Vehicle'}    Image         Rectangle     {'None'}     {'Bounding box for the vehicle. Use thi
    {'Vehicle'}    PointCloud    Cuboid        {'None'}     {'Bounding box for the vehicle. Use thi
```

Input Arguments

`ldc` — Label definition creator for multisignal workflow
`labelDefinitionCreatorMultisignal` object

Label definition creator for the multisignal workflow, specified as a `labelDefinitionCreatorMultisignal` object. The object defines the labels, sublabels, and attributes used for generating the label definitions table `labelDefs`.

Output Arguments

`labelDefs` — Label definitions
table

Label definitions, returned as a table with up to eight columns. The possible columns are *Name*, *SignalType*, *LabelType*, *Group*, *Description*, *LabelColor*, *PixelLabelID*, and *Hierarchy*. This table specifies the definitions of labels, sublabels, and attributes for labeling ground truth data. For more details, see LabelDefinitions property of `groundTruthMultisignal` object.

See Also

Objects
`labelDefinitionCreatorMultisignal`

Functions
`addAttribute` | `addLabel` | `addSublabel` | `info`

Introduced in R2020a

editAttributeDescription

Modify attribute description in label definition creator object for multisignal workflow

Syntax

`editAttributeDescription(ldc,labelName,attributeName,description)`

Description

`editAttributeDescription(ldc,labelName,attributeName,description)` modifies the description of an attribute under the label or sublabel identified by `labelName`. The label or sublabel must be associated with the `labelDefinitionCreatorMultisignal` object `ldc`.

Examples

Modify Attribute Description in Label Definition Creator Object for Multisignal Workflow

Create an empty `labelDefinitionCreatorMultisignal` object.

```
ldc = labelDefinitionCreatorMultisignal;
```

Add a label called `'TrafficLight'`.

```
addLabel(ldc,'TrafficLight',labelType.Rectangle);
```

Add a sublabel named `'RedLight'` for label `'TrafficLight'`.

```
addSublabel(ldc,'TrafficLight','RedLight',labelType.Rectangle);
```

Add an attribute named `'Active'` to the label `'TrafficLight'`. Set the attribute type as `'Logical'` with the default value `true`.

```
addAttribute(ldc,'TrafficLight','Active',attributeType.Logical,true);
```

Add an attribute called `'isOn'` to the sublabel `'RedLight'`. Set the attribute type as `'Logical'` with the default value `false`.

```
addAttribute(ldc,'TrafficLight/RedLight','isOn',attributeType.Logical,false);
```

Modify the Attribute Description Under a Label

Display information about the label `'TrafficLight'`.

```
info(ldc,'TrafficLight')
```

```
          Name: "TrafficLight"
    SignalType: Image
     LabelType: Rectangle
         Group: "None"
    LabelColor: {''}
    Attributes: "Active"
```

```
      Sublabels: "RedLight"
    Description: ' '

           Name: "TrafficLight"
     SignalType: PointCloud
      LabelType: Cuboid
          Group: "None"
     LabelColor: {''}
     Attributes: "Active"
      Sublabels: "RedLight"
    Description: ' '
```

Modify the description of the attribute `'Active'` under the label `'TrafficLight'`.

```
editAttributeDescription(ldc,'TrafficLight','Active','Is Active: true (DefaultValue: 1), false (D
```

Display information about the label `'TrafficLight'` to verify the modified attribute description.

```
info(ldc,'TrafficLight/Active')
```

```
            Name: "Active"
            Type: Logical
    DefaultValue: 1
     Description: 'Is Active: true (DefaultValue: 1), false (DefaultValue: 0)'
```

Modify the Attribute Description Under a Sublabel

Display information about the sublabel `'RedLight'`.

```
info(ldc,'TrafficLight/RedLight')
```

```
           Name: "RedLight"
           Type: Rectangle
     LabelColor: ''
     Attributes: "isOn"
      Sublabels: []
    Description: ' '
```

Modify the description of the attribute `'isOn'` under the sublabel `'RedLight'`.

```
editAttributeDescription(ldc,'TrafficLight/RedLight','isOn','Is On: true (DefaultValue: 1), false
```

Display information about the sublabel `'RedLight'` to verify the modified attribute description.

```
info(ldc,'TrafficLight/RedLight/isOn')
```

```
            Name: "isOn"
            Type: Logical
    DefaultValue: 0
     Description: 'Is On: true (DefaultValue: 1), false (DefaultValue: 0)'
```

Input Arguments

ldc — Label definition creator for multisignal workflow
`labelDefinitionCreatorMultisignal` object

Label definition creator for the multisignal workflow, specified as a `labelDefinitionCreatorMultisignal` object.

`labelName` — Label or sublabel name
character vector | string scalar

Label or sublabel name, specified as a character vector or string scalar that uniquely identifies the label or sublabel to which the attribute is associated.

- To specify a label, use the form '*labelName*'.

Example: `editAttributeDescription(ldc,'TrafficLight','Active','Is Active: true (DefaultValue: 1), false (DefaultValue: 0)')`

- To specify a sublabel, use the form '*labelName/sublabelName*'. In this case, the attribute is associated with the sublabel.

Example: `editAttributeDescription(ldc,'TrafficLight/RedLight','isOn','Is On: true (DefaultValue: 1), false (DefaultValue: 0)')`

`attributeName` — Attribute name
character vector | string scalar

Attribute name, specified as a character vector or string scalar that identifies the attribute for which the description is to be modified.

`description` — Description
character vector | string scalar

Description, specified as a character vector or string scalar that contains a new description for the attribute identified by `attributeName`.

See Also

Objects
`labelDefinitionCreatorMultisignal`

Functions
`editLabelDescription`

Introduced in R2020a

editGroupName

Change group name in label definition creator object for multisignal workflow

Syntax

```
editGroupName(ldc,oldname,newname)
```

Description

`editGroupName(ldc,oldname,newname)` changes the group name from `oldname` to `newname`. This function changes the group name in all the label definitions that have the `oldname`.

Examples

Rename Label Group in Label Definition Creator Object for Multisignal Workflow

Create an empty `labelDefinitionCreatorMultisignal` object.

```
ldc = labelDefinitionCreatorMultisignal;
```

Add labels named `'Car'` and `'Truck'` in a group named `'Vehicle'`.

```
addLabel(ldc,'Car',labelType.Rectangle,'Group','Vehicle');
addLabel(ldc,'Truck',labelType.Rectangle,'Group','Vehicle');
```

Change the `'Vehicle'` group name `'FourWheeler'`.

```
editGroupName(ldc,'Vehicle','FourWheeler');
```

Display the details of the updated `labelDefinitionCreatorMultisignal` object.

```
ldc
```

```
ldc =
labelDefinitionCreatorMultisignal contains the following labels:

    Car with 0 sublabels and 0 attributes and belongs to FourWheeler group.    (info)
    Truck with 0 sublabels and 0 attributes and belongs to FourWheeler group.    (info)

For more details about attributes and sublabels, use the info method.
```

Input Arguments

ldc — Label definition creator for multisignal workflow
`labelDefinitionCreatorMultisignal` object

Label definition creator for the multisignal workflow, specified as a `labelDefinitionCreatorMultisignal` object.

`oldname` — Old group name
character vector | string scalar

Old group name, specified as a character vector or string scalar that uniquely identifies group name you want to modify.

`newname` — New group name
character vector | string scalar

New group name, specified as a character vector or string scalar that uniquely identifies the new group name.

See Also

Objects
`labelDefinitionCreatorMultisignal`

Functions
`editLabelDescription` | `editLabelGroup`

Introduced in R2020a

editLabelDescription

Modify label or sublabel description in label definition creator object for multisignal workflow

Syntax

`editLabelDescription(ldc,labelName,description)`

Description

`editLabelDescription(ldc,labelName,description)` modifies the description of a label or sublabel identified by `labelName`. The label or sublabel must be associated with the `labelDefinitionCreatorMultisignal` object `ldc`.

Examples

Modify Description of Label and Sublabel in Label Definition Creator Object for Multisignal Workflow

Create an empty `labelDefinitionCreatorMultisignal` object.

```
ldc = labelDefinitionCreatorMultisignal;
```

Add a label with the name 'TrafficLight'. Specify the type of label as 'Rectangle'.

```
addLabel(ldc,'TrafficLight','Rectangle')
```

Add a sublabel called 'Light' to the label 'TrafficLight'. Specify the type of the sublabel as 'Rectangle'.

```
addSublabel(ldc,'TrafficLight','Light','Rectangle')
```

Modify Label Description

Display information about the label 'TrafficLight'.

```
info(ldc,'TrafficLight')
```

```
           Name: "TrafficLight"
     SignalType: Image
      LabelType: Rectangle
          Group: "None"
     LabelColor: {''}
     Attributes: []
      Sublabels: "Light"
    Description: ' '

           Name: "TrafficLight"
     SignalType: PointCloud
      LabelType: Cuboid
          Group: "None"
     LabelColor: {''}
     Attributes: []
```

```
      Sublabels: "Light"
    Description: ' '
```

Modify the description for the label 'TrafficLight'.

```
editLabelDescription(ldc,'TrafficLight','Bounding box for the traffic light')
```

Display information about the label 'TrafficLight' to verify the modified label description.

```
info(ldc,'TrafficLight')
```

```
           Name: "TrafficLight"
     SignalType: Image
      LabelType: Rectangle
          Group: "None"
     LabelColor: {''}
     Attributes: []
      Sublabels: "Light"
    Description: 'Bounding box for the traffic light'

           Name: "TrafficLight"
     SignalType: PointCloud
      LabelType: Cuboid
          Group: "None"
     LabelColor: {''}
     Attributes: []
      Sublabels: "Light"
    Description: 'Bounding box for the traffic light'
```

Modify Sublabel Description

Display information about the sublabel 'Light' under the label 'TrafficLight'.

```
info(ldc,'TrafficLight/Light')
```

```
           Name: "Light"
           Type: Rectangle
     LabelColor: ''
     Attributes: []
      Sublabels: []
    Description: ' '
```

Modify the description for the sublabel 'Light'.

```
editLabelDescription(ldc,'TrafficLight/Light','Bounding box around each light of the Traffic ligh
```

Display information about the sublabel 'Light' under the label 'TrafficLight' to verify the modified sublabel description.

```
info(ldc,'TrafficLight/Light')
```

```
           Name: "Light"
           Type: Rectangle
     LabelColor: ''
     Attributes: []
      Sublabels: []
    Description: 'Bounding box around each light of the Traffic light'
```

Input Arguments

`ldc` — Label definition creator for multisignal workflow
`labelDefinitionCreatorMultisignal` object

Label definition creator for the multisignal workflow, specified as a `labelDefinitionCreatorMultisignal` object.

`labelName` — Label or sublabel name
character vector | string scalar

Label or sublabel name, specified as a character vector or string scalar that uniquely identifies the label or sublabel for which the description is to be modified.

- To specify a label, use the form '*labelName*'.

Example: `editLabelDescription(ldc,'TrafficLight','Bounding box for the traffic light')`

- To specify a sublabel, use the form '*labelName/sublabelName*'.

Example: `editLabelDescription(ldc,'TrafficLight/Light','Bounding box around each light of the Traffic light')`

`description` — Description
character vector | string scalar

Description, specified as a character vector or string scalar that contains the new description for the label or sublabel identified by `labelName`.

See Also

Objects
`groundTruthMultisignal` | `labelDefinitionCreatorMultisignal`

Functions
`editAttributeDescription`

Introduced in R2020a

editLabelGroup

Modify label group name in label definition creator object for multisignal workflow

Syntax

```
editLabelGroup(ldc,labelName,groupName)
```

Description

`editLabelGroup(ldc,labelName,groupName)` modifies the group name that corresponds to the label identified by `labelName`. The label must be associated with the `labelDefinitionCreatorMultisignal` object `ldc`.

Examples

Modify Group Name for Labels in Label Definition Creator Object for Multisignal Workflow

Create an empty `labelDefinitionCreatorMultisignal` object.

```
ldc = labelDefinitionCreatorMultisignal;
```

Add a label named `'Car'` in a group named `'Vehicle'`. Set the type of the label as `'Rectangle'`. Adding a `'Rectangle'` also adds a `'Cuboid'` entry to the label definitions table.

```
addLabel(ldc,'Car','Rectangle','Group','Vehicle')
```

Display information about the group name of the label `'Car'` using the object function `info`.

```
info(ldc,'Car')
```

```
           Name: "Car"
     SignalType: Image
      LabelType: Rectangle
          Group: "Vehicle"
     LabelColor: {''}
     Attributes: []
      Sublabels: []
    Description: ' '

           Name: "Car"
     SignalType: PointCloud
      LabelType: Cuboid
          Group: "Vehicle"
     LabelColor: {''}
     Attributes: []
      Sublabels: []
    Description: ' '
```

Add a label named `'Truck'` to group named `'FourWheeler'`. Set the type of the label as `'Rectangle'`.

```
addLabel(ldc,'Truck',labelType.Rectangle,'Group','FourWheeler')
```

Move the `'Car'` label into the `'FourWheeler'` group.

```
editLabelGroup(ldc,'Car','FourWheeler')
```

Display information about the label `'Car'` to confirm the group name of the label is changed from `'Vehicle'` to `'FourWheeler'` using the object function `info`.

```
info(ldc,'Car')
```

```
          Name: "Car"
    SignalType: Image
     LabelType: Rectangle
         Group: "FourWheeler"
    LabelColor: {''}
    Attributes: []
     Sublabels: []
   Description: ' '

          Name: "Car"
    SignalType: PointCloud
     LabelType: Cuboid
         Group: "FourWheeler"
    LabelColor: {''}
    Attributes: []
     Sublabels: []
   Description: ' '
```

Input Arguments

ldc — Label definition creator for multisignal workflow
`labelDefinitionCreatorMultisignal` object

Label definition creator for the multisignal workflow, specified as a `labelDefinitionCreatorMultisignal` object.

labelName — Label name
character vector | string scalar

Label name, specified as a character vector or string scalar that uniquely identifies the label that corresponds to the `groupName` you want to modify.

groupName — Group name
character vector | string scalar

Group name, specified as a character vector or string scalar that identifies the group you want to modify, which corresponds to the label specified by `labelName`.

See Also

Objects
`labelDefinitionCreatorMultisignal`

Functions
`editGroupName` | `editLabelDescription`

Introduced in R2020a

info

Display label, sublabel, or attribute information stored in label definition creator object for multisignal workflow

Syntax

```
info(ldc,name)
infoStruct = info(ldc,name)
```

Description

`info(ldc,name)` displays information about the specified label, sublabel, or attribute stored in the `labelDefinitionCreatorMultisignal` object `ldc`.

`infoStruct = info(ldc,name)` returns the information as a structure.

Examples

Display Information On Definitions Stored in Label Definition Creator Object for Multisignal Workflow

Load an existing label definition table.

```
labelDefFile = fullfile(toolboxdir('driving'), 'drivingdata', 'labelDefsMultiSignal.mat');
ld = load(labelDefFile)
```

```
ld = struct with fields:
    labelDefs: [6x6 table]
```

Create a `labelDefinitionCreatorMultisignal` object from the label definitions table.

```
ldc = labelDefinitionCreatorMultisignal(ld.labelDefs)
```

```
ldc =
labelDefinitionCreatorMultisignal contains the following labels:

    Car with 0 sublabels and 0 attributes and belongs to None group.    (info)
    LeftLane with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Road with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Sunny with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Urban with 0 sublabels and 0 attributes and belongs to None group.    (info)

For more details about attributes and sublabels, use the info method.
```

Add an attribute `'Color'` to the label `'Car'`. Specify the attribute type as `'List'` and add items to the list.

```
addAttribute(ldc,'Car','Color','List',{'Red','Green','Blue'});
```

Display the details of the updated `labelDefinitionCreatorMultisignal` object.

```
ldc
```

```
ldc = 
labelDefinitionCreatorMultisignal contains the following labels:

    Car with 0 sublabels and 1 attributes and belongs to None group.    (info)
    LeftLane with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Road with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Sunny with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Urban with 0 sublabels and 0 attributes and belongs to None group.    (info)

For more details about attributes and sublabels, use the info method.
```

Display information about the attribute `'Color'` under the label `'Car'`.

```
colorStruct = info(ldc,'Car/Color')
```

```
colorStruct = struct with fields:
           Name: "Color"
           Type: List
      ListItems: {'Red'  'Green'  'Blue'}
    Description: ' '
```

Display the field `ListItems` in the `'Color'` attribute of the label `'Car'`.

```
colorStruct.ListItems
```

```
ans = 1x3 cell
    {'Red'}    {'Green'}    {'Blue'}
```

Input Arguments

ldc — Label definition creator for multisignal workflow
`labelDefinitionCreatorMultisignal` object

Label definition creator for the multisignal workflow, specified as a `labelDefinitionCreatorMultisignal` object.

name — Name of label, sublabel, or attribute
character vector | string scalar

Name of label, sublabel, or attribute in the `ldc` object, specified as a character vector or string scalar whose form depends on the type of name you specify.

- To specify a label, use the form '*labelName*'.

Example: `info(ldc,'TrafficLight')`

- To specify a sublabel, use the form '*labelName/sublabelName*'.

Example: `info(ldc,'TrafficLight/RedLight')`

- To specify an attribute, use the form '*labelName/attributeName*' or '*labelName/sublabelName/attributeName*'.

Example: `info(ldc,'TrafficLight/Active')`

Example: `info(ldc,'TrafficLight/RedLight/isOn')`

Output Arguments

`infoStruct` — Information structure
structure

Information structure, returned as a structure that contains the fields `Name`, `SignalType` (for labels), `LabelType` (for labels), `Type` (for sublabels and attributes), `Description`, `Attributes` (when pertinent), `Sublabels` (when pertinent), `DefaultValue` (for attributes), and `ListItems` (for List attributes).

See Also

Objects
`labelDefinitionCreatorMultisignal`

Functions
`addLabel` | `create`

Introduced in R2020a

removeAttribute

Remove attribute from label or sublabel in label definition creator object for multisignal workflow

Syntax

`removeAttribute(ldc,labelName,attributeName)`

Description

`removeAttribute(ldc,labelName,attributeName)` removes the specified attribute from the indicated label or sublabel in the `labelDefinitionCreatorMultisignal` object `ldc`.

Examples

Remove Attributes from Label and Sublabel in Label Definition Creator Object for Multisignal Workflow

Create an empty `labelDefinitionCreatorMultisignal` object.

```
ldc = labelDefinitionCreatorMultisignal;
```

Add a label with the name `'TrafficLight'`. Specify the type of label as `'Rectangle'`. Adding a `'Rectangle'` also adds a `'Cuboid'` entry to the label definitions table.

```
addLabel(ldc,'TrafficLight','Rectangle')
```

Add attribute `'Active'` to the label. Specify the attribute type as `'Logical'` with the value `true`.

```
addAttribute(ldc,'TrafficLight','Active','Logical',true)
```

Display information about the attributes under the label `'TrafficLight'` using the object function `info`.

```
info(ldc,'TrafficLight')
```

```
        Name: "TrafficLight"
  SignalType: Image
   LabelType: Rectangle
       Group: "None"
  LabelColor: {''}
  Attributes: "Active"
   Sublabels: []
 Description: ' '

        Name: "TrafficLight"
  SignalType: PointCloud
   LabelType: Cuboid
       Group: "None"
  LabelColor: {''}
  Attributes: "Active"
   Sublabels: []
 Description: ' '
```

Remove the attribute 'Active' from the label 'TrafficLight'.

```
removeAttribute(ldc,'TrafficLight','Active')
```

Add a sublabel called 'RedLight' to the label 'TrafficLight'. Specify the type of the sublabel as 'Rectangle'.

```
addSublabel(ldc,'TrafficLight','RedLight','Rectangle')
```

Add an attribute 'isOn' to the sublabel 'RedLight'. Specify the type for the attribute 'isOn' as 'Logical' with the value false.

```
addAttribute(ldc,'TrafficLight/RedLight','isOn','Logical',false)
```

Display information about the attributes under the sublabel 'RedLight' in the label 'TrafficLight' using the object function info.

```
info(ldc,'TrafficLight/RedLight')
```

```
           Name: "RedLight"
           Type: Rectangle
     LabelColor: ''
     Attributes: "isOn"
      Sublabels: []
    Description: ' '
```

Remove the attribute 'isOn' from the sublabel 'RedLight'.

```
removeAttribute(ldc,'TrafficLight/RedLight','isOn')
```

Display information about the label 'TrafficLight' using the object function info, to confirm that the attribute 'Active' has been removed from the label definitions.

```
info(ldc,'TrafficLight')
```

```
           Name: "TrafficLight"
     SignalType: Image
      LabelType: Rectangle
          Group: "None"
     LabelColor: {''}
     Attributes: []
      Sublabels: "RedLight"
    Description: ' '

           Name: "TrafficLight"
     SignalType: PointCloud
      LabelType: Cuboid
          Group: "None"
     LabelColor: {''}
     Attributes: []
      Sublabels: "RedLight"
    Description: ' '
```

Display information about the sublabel 'RedLight' in the label 'TrafficLight' using the object function info, to confirm that the attribute 'isOn' has been removed from the label definitions.

```
info(ldc,'TrafficLight/RedLight')
```

```
           Name: "RedLight"
           Type: Rectangle
```

```
     LabelColor: ''
     Attributes: []
      Sublabels: []
    Description: ' '
```

Input Arguments

ldc — Label definition creator for multisignal workflow
`labelDefinitionCreatorMultisignal` object

Label definition creator for the multisignal workflow, specified as a `labelDefinitionCreatorMultisignal` object.

labelName — Label or sublabel name
character vector | string scalar

Label or sublabel name, specified as a character vector or string scalar that uniquely identifies the label or sublabel from which the attribute is to be removed.

- To specify a label, use the form '*labelName*'.

Example: `removeAttribute(ldc,'TrafficLight','Active')`

- To specify a sublabel, use the form '*labelName/sublabelName*'. In this case, the attribute associates with the sublabel.

Example: `removeAttribute(ldc,'TrafficLight/RedLight','isOn')`

attributeName — Attribute name
character vector | string scalar

Attribute name, specified as a character vector or string scalar that identifies the attribute to be removed from the label or sublabel indicated by `labelName`.

See Also

Objects
`labelDefinitionCreatorMultisignal`

Functions
`addAttribute` | `addLabel` | `removeLabel`

Introduced in R2020a

removeLabel

Remove label from label definition creator object for multisignal workflow

Syntax

```
removeLabel(ldc,labelName)
```

Description

`removeLabel(ldc,labelName)` removes the specified label from the `labelDefinitionCreatorMultisignal` object `ldc`.

Note Removing a label also removes any sublabels or attributes associated with that label.

Examples

Remove Label from Label Definition Creator Object for Multisignal Workflow

Load an existing label definitions table into the workspace.

```
labelDefFile = fullfile(toolboxdir('driving'),'drivingdata','labelDefsMultiSignal.mat');
ld = load(labelDefFile)
```

```
ld = struct with fields:
    labelDefs: [6x6 table]
```

Create a `labelDefinitionCreatorMultisignal` object from the label definitions table.

```
ldc = labelDefinitionCreatorMultisignal(ld.labelDefs)
```

```
ldc = 
labelDefinitionCreatorMultisignal contains the following labels:

    Car with 0 sublabels and 0 attributes and belongs to None group.    (info)
    LeftLane with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Road with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Sunny with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Urban with 0 sublabels and 0 attributes and belongs to None group.    (info)

For more details about attributes and sublabels, use the info method.
```

Remove the label called `'Car'`.

```
removeLabel(ldc,'Car');
```

Display the details of the updated `labelDefinitionCreatorMultisignal` object to confirm that the label has been removed.

```
ldc
```

```
ldc = 
labelDefinitionCreatorMultisignal contains the following labels:

    LeftLane with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Road with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Sunny with 0 sublabels and 0 attributes and belongs to None group.    (info)
    Urban with 0 sublabels and 0 attributes and belongs to None group.    (info)

For more details about attributes and sublabels, use the info method.
```

Input Arguments

ldc — Label definition creator for multisignal workflow
labelDefinitionCreatorMultisignal object

Label definition creator for the multisignal workflow, specified as a labelDefinitionCreatorMultisignal object.

labelName — Label name
character vector | string scalar

Label name, specified as a character vector or string scalar that uniquely identifies the label to be removed from the ldc object.

See Also

Objects
labelDefinitionCreatorMultisignal

Functions
addLabel | removeAttribute | removeSublabel

Introduced in R2020a

removeSublabel

Remove sublabel from label in label definition creator object for multisignal workflow

Syntax

`removeSublabel(ldc,labelName,sublabelName)`

Description

`removeSublabel(ldc,labelName,sublabelName)` removes the specified sublabel from the indicated label. This label must be associated with the `labelDefinitionCreatorMultisignal` object `ldc`.

Note Removing a sublabel also removes any attributes associated with that sublabel.

Examples

Remove Sublabel from Label in Label Definition Creator Object for Multisignal Workflow

Create an empty `labelDefinitionCreatorMultisignal` object.

```
ldc = labelDefinitionCreatorMultisignal;
```

Add a label with the name `'TrafficLight'`. Specify the type of label as `'Rectangle'` and add a description. Adding a `'Rectangle'` also adds a `'Cuboid'` entry to the label definitions table.

```
addLabel(ldc,'TrafficLight',labelType.Rectangle,'Description','Bounding boxes for traffic light')
```

Add sublabels called `'RedLight'`, `'GreenLight'` and `'YellowLight'` to the label `'TrafficLight'`. Specify the type of the sublabels as `'Rectangle'`.

```
addSublabel(ldc,'TrafficLight','RedLight','Rectangle')
addSublabel(ldc,'TrafficLight','GreenLight','rect')
addSublabel(ldc,'TrafficLight','YellowLight',labelType.Rectangle)
```

Display information about the label 'TrafficLight' using the object function `info`, to confirm that the sublabels have been added to the label definitions.

```
info(ldc,'TrafficLight')
```

```
        Name: "TrafficLight"
  SignalType: Image
   LabelType: Rectangle
       Group: "None"
  LabelColor: {''}
  Attributes: []
   Sublabels: ["RedLight"    "GreenLight"    "YellowLight"]
 Description: 'Bounding boxes for traffic light'

        Name: "TrafficLight"
```

```
     SignalType: PointCloud
      LabelType: Cuboid
          Group: "None"
     LabelColor: {''}
     Attributes: []
      Sublabels: ["RedLight"    "GreenLight"    "YellowLight"]
    Description: 'Bounding boxes for traffic light'
```

Remove the sublabel 'YellowLight' from the label 'TrafficLight'.

```
removeSublabel(ldc,'TrafficLight','YellowLight')
```

Display information about the label 'TrafficLight' using the object function `info`, to confirm that the sublabel 'YellowLight' has been removed from the label definitions.

```
info(ldc,'TrafficLight')
```

```
           Name: "TrafficLight"
     SignalType: Image
      LabelType: Rectangle
          Group: "None"
     LabelColor: {''}
     Attributes: []
      Sublabels: ["RedLight"    "GreenLight"]
    Description: 'Bounding boxes for traffic light'

           Name: "TrafficLight"
     SignalType: PointCloud
      LabelType: Cuboid
          Group: "None"
     LabelColor: {''}
     Attributes: []
      Sublabels: ["RedLight"    "GreenLight"]
    Description: 'Bounding boxes for traffic light'
```

Input Arguments

`ldc` — Label definition creator for multisignal workflow
`labelDefinitionCreatorMultisignal` object

Label definition creator for the multisignal workflow, specified as a `labelDefinitionCreatorMultisignal` object.

`labelName` — Label name
character vector | string scalar

Label name, specified as a character vector or string scalar that uniquely identifies the label with which the sublabel is associated.

`sublabelName` — Sublabel name
character vector | string scalar

Sublabel name, specified as a character vector or string scalar that identifies the sublabel to be removed from the indicated label `labelName`.

See Also

Objects
`labelDefinitionCreatorMultisignal`

Functions
`addLabel` | `addSublabel` | `removeAttribute` | `removeLabel`

Introduced in R2020a

ROILabelData

Ground truth data for ROI labels

Description

The ROILabelData object stores ground truth data for region of interest (ROI) label definitions for each signal in a groundTruthMultisignal object.

Creation

When you export a groundTruthMultisignal object from a **Ground Truth Labeler** app session, the ROILabelData property of the exported object stores the ROI labels as an ROILabelData object. To create an ROILabelData object programmatically, use the vision.labeler.labeldata.ROILabelData function (described here).

Syntax

```
roiLabelData = vision.labeler.labeldata.ROILabelData(signalNames,labelData)
```

Description

roiLabelData = vision.labeler.labeldata.ROILabelData(signalNames,labelData) creates an object containing ROI label data for multiple signals. The created object, roiLabelData, contains properties with the signal names listed in signalNames. These properties store the corresponding ROI label data specified by labelData.

Input Arguments

signalNames — Signal names
string array

Signal names, specified as a string array. Specify the names of all signals present in the groundTruthMultisignal object you are creating. You can get the signal names from an existing groundTruthMultisignal object by accessing the DataSource property of that object. Use this command and replace gTruth with the name of your groundTruthMultisignal object variable.

```
gTruth.DataSource.SignalName
```

In an exported groundTruthMultisignal object, the ROILabelData object contains a label data property for each signal, even if some signals do not have ROI label data.

The properties of the created ROILabelData object have the names specified by signalNames.

Example: ["video_01_city_c2s_fcw_10s" "lidarSequence"]

labelData — ROI label data for each signal
cell array of timetables

ROI label data for each signal, specified as a cell array of timetables. Each timetable in the cell array contains data for the signal in the corresponding position of the `signalNames` input. The `ROILabelData` object stores each timetable in a property that has the same name as that signal.

The timetable format for each signal depends on data from the `groundTruthMultisignal` object that you exported or are creating.

Each timetable contains one column per label definition stored in the `LabelDefinitions` property of the `groundTruthMultisignal` object. Label definitions that the signal type does not support are excluded. For example, suppose you define a `Line` ROI label named `'lane'`. The timetable for a lidar point cloud signal does not include a `lane` column, because these signals do not support `Line` ROI labels. In the `DataSource` property of the `groundTruthMultisignal` object, the `SignalType` property of each data source lists the valid signal types.

The height of the timetable is defined by the number of timestamps in the signal. In the `DataSource` property of the `groundTruthMultisignal` object, the `Timestamp` property of each data source lists the signal timestamps.

For each label definition, all ROI labels marked at that timestamps are combined into a single cell in the table. Consider the ROI label data for a video signal stored in a `groundTruthMultisignal` object, `gTruth`. At each timestamp, `car` contains three labels, `truck` contains one label, and `lane` contains two labels.

```
gTruth.ROILabelData.video_01_city_c2s_fcw_10s
```

```
ans =

  5×4 timetable

      Time           car             truck            lane
    _________    ____________    ____________    ____________
    0 sec        {3×4 double}    {1×4 double}    {2×1 cell  }
    0.05 sec     {3×4 double}    {1×4 double}    {2×1 cell  }
    0.1 sec      {3×4 double}    {1×4 double}    {2×1 cell  }
    0.15 sec     {3×4 double}    {1×4 double}    {2×1 cell  }
    0.2 sec      {3×4 double}    {1×4 double}    {2×1 cell  }
```

The storage format for ROI label data depends on the label type.

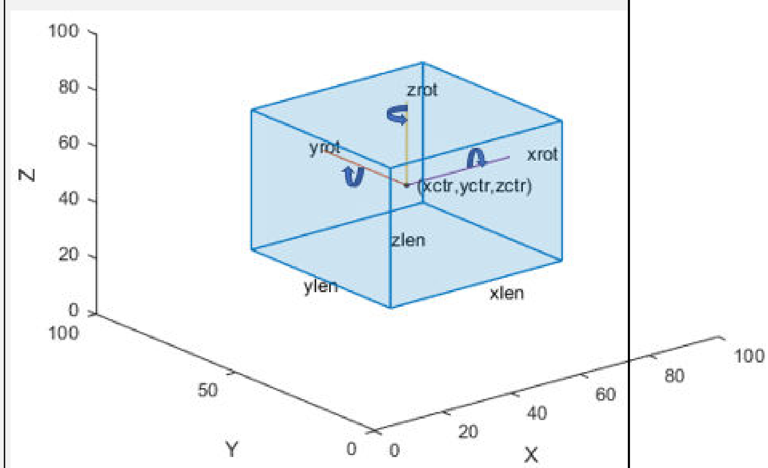
Label Type	Storage Format for Labels at Each Timestamp
`labelType.Rectangle`	`M`-by-4 numeric matrix of the form `[x, y, w, h]`, where: • `M` is the number of labels in the frame. • `x` and `y` specify the upper-left corner of the rectangle. • `w` specifies the width of the rectangle, which is its length along the *x*-axis. • `h` specifies the height of the rectangle, which is its length along the *y*-axis.

<table>
<tr><th>Label Type</th><th>Storage Format for Labels at Each Timestamp</th></tr>
<tr><td><code>labelType.Cuboid</code>

<code>labelType.ProjectedCuboid</code></td><td><code>M</code>-by-9 numeric matrix with rows of the form <code>[xctr, yctr, zctr, xlen, ylen, zlen, xrot, yrot, zrot]</code>, where:

• <code>M</code> is the number of labels in the frame.
• <code>xctr</code>, <code>yctr</code>, and <code>zctr</code> specify the center of the cuboid.
• <code>xlen</code>, <code>ylen</code>, and <code>zlen</code> specify the length of the cuboid along the x-axis, y-axis, and z-axis, respectively, before rotation has been applied.
• <code>xrot</code>, <code>yrot</code>, and <code>zrot</code> specify the rotation angles for the cuboid along the x-axis, y-axis, and z-axis, respectively. These angles are clockwise-positive when looking in the forward direction of their corresponding axes.

The figure shows how these values determine the position of a cuboid.


</td></tr>
<tr><td><code>labelType.Line</code></td><td><code>M</code>-by-1 vector of cell arrays, where <code>M</code> is the number of labels in the frame. Each cell array contains an <code>N</code>-by-2 numeric matrix of the form <code>[x1 y1; x2 y2; ... ; xN yN]</code> for <code>N</code> points in the polyline.</td></tr>
</table>

Label Type	Storage Format for Labels at Each Timestamp
`labelType.PixelLabel`	Label data for all pixel label definitions is stored in a single `PixelLabelData` column as a categorical label matrix. The label matrix must be stored on disk as a `uint8` image. When creating an `ROILabelData` object with pixel label data, the image file name must be specified as a character vector in the `labelData` input. The label matrix must contain 1 or 3 channels. For a 3-channel matrix, the RGB pixel values represent label IDs.
`labelType.Custom`	Labels are stored exactly as they are specified in the timetable. If you import a `groundTruthMultisignal` object containing custom label data into the **Ground Truth Labeler** app, this data is not imported into the app. Use custom data when gathering label data for training and combining it with data labeled in the app.

If the ROI label data includes sublabels or attributes, then the labels at each timestamp must be specified as structures instead. The structure includes these fields.

Label Structure Field	Description
`Position`	Positions of the parent labels at the given timestamp The format of `Position` depends on the label type. These formats are described in the previous table.
`AttributeName1,...,AttributeNameN`	Attributes of the parent labels Each defined sublabel has its own field, where the name of the field corresponds to the attribute name. The attribute value is a character vector for a `List` or `String` attribute, a numeric scalar for a `Numeric` attribute, or a logical scalar for a `Logical` attribute. If the attribute is unspecified, then the attribute value is an empty vector.

<table>
<tr><th>Label Structure Field</th><th>Description</th></tr>
<tr><td><code>SublabelName1,...,SublabelNameN</code></td><td>Sublabels of the parent labels

Each defined sublabel has its own field, where the name of the field corresponds to the sublabel name. The value of each sublabel field is a structure containing the data for all marked sublabels with that name at the given timestamp.

This table describes the format of this sublabel structure.
<table>
<tr><th>Sublabel Structure Field</th><th>Description</th></tr>
<tr><td><code>Position</code></td><td>Positions of the sublabels at the given timestamp

The format of <code>Position</code> depends on the label type. These formats are described in the previous table.</td></tr>
<tr><td><code>AttributeName1,...,AttributeNameN</code></td><td>Attributes of the sublabels

Each defined sublabel has its own field, where the name of the field corresponds to the attribute name. The attribute value is a character vector for a <code>List</code> or <code>String</code> attribute, a numeric scalar for a <code>Numeric</code> attribute, or a logical scalar for a <code>Logical</code> attribute. If you leave an attribute unspecified, then the attribute value is an empty vector.</td></tr>
</table>
</td></tr>
</table>

Properties

`SignalName1,...,SignalNameN` — ROI label data for each signal (as separate properties)

timetables

ROI label data, specified as timetables. The `ROILabelData` object contains one property per signal, where each property contains a timetable of ROI label data corresponding to that signal.

When exporting an ROILabelData object from a **Ground Truth Labeler** app session, the property names correspond to the signal names stored in the DataSource property of the exported groundTruthMultisignal object.

When creating an ROILabelData object programmatically, the signalNames and labelData input arguments define the property names and values of the created object.

Suppose you want to create a groundTruthMultisignal object containing a video signal and a lidar point cloud sequence signal. Specify the signals in a string array, signalNames.

```
signalNames = ["video_01_city_c2s_fcw_10s" "lidarSequence"];
```

Store the video ROI labels, videoData, and lidar point cloud sequence ROI labels, lidarData, in a cell array of timetables, labelData. Each timetable contains the data for the corresponding signal in signalNames.

```
labelData = {videoData,lidarData}
```

```
  1×2 cell array

    {204×2 timetable}    {34×1 timetable}
```

The ROILabelData object, roiData, stores this data in the property with the corresponding signal name. You can specify roiData in the ROILabelData property of a groundTruthMultisignal object.

```
roiData = vision.labeler.labeldata.ROILabelData(signalNames,labelData)
```

```
roiData = 

  ROILabelData with properties:

    video_01_city_c2s_fcw_10s: [204×2 timetable]
                lidarSequence: [34×1 timetable]
```

Examples

Create Ground Truth from Multiple Signals

Create ground truth data for a video signal and a lidar point cloud sequence signal that captures the same driving scene. Specify the signal sources, label definitions, and ROI and scene label data.

Create the video data source from an MP4 file.

```
sourceName = '01_city_c2s_fcw_10s.mp4';
sourceParams = [];
vidSource = vision.labeler.loading.VideoSource;
vidSource.loadSource(sourceName,sourceParams);
```

Create the point cloud sequence source from a folder of point cloud data (PCD) files.

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

```
lidarSourceData = load(fullfile(pcSeqFolder,'timestamps.mat'));

sourceName = pcSeqFolder;
sourceParams = struct;
sourceParams.Timestamps = timestamps;

pcseqSource = vision.labeler.loading.PointCloudSequenceSource;
pcseqSource.loadSource(sourceName,sourceParams);
```

Combine the signal sources into an array.

```
dataSource = [vidSource pcseqSource]
```

```
dataSource = 

  1x2 heterogeneous MultiSignalSource (VideoSource, PointCloudSequenceSource) array with propert

    SourceName
    SourceParams
    SignalName
    SignalType
    Timestamp
    NumSignals
```

Create a table of label definitions for the ground truth data by using a `labelDefinitionCreatorMultisignal` object.

- The `Car` label definition appears twice. Even though `Car` is defined as a rectangle, you can draw rectangles only for image signals, such as videos. The `labelDefinitionCreatorMultisignal` object creates an additional row for lidar point cloud signals. In these signal types, you can draw `Car` labels as cuboids only.
- The label definitions have no descriptions and no assigned colors, so the `Description` and `LabelColor` columns are empty.
- The label definitions have no assigned groups, so for all label definitions, the corresponding cell in the `Group` column is set to `'None'`.
- `Road` is a pixel label definition, so the table includes a `PixelLabelID` column.
- No label definitions have sublabels or attributes, so the table does not include a `Hierarchy` column for storing such information.

```
ldc = labelDefinitionCreatorMultisignal;
addLabel(ldc,'Car','Rectangle');
addLabel(ldc,'Truck','ProjectedCuboid');
addLabel(ldc,'Lane','Line');
addLabel(ldc,'Road','PixelLabel');
addLabel(ldc,'Sunny','Scene');
labelDefs = create(ldc)
```

```
labelDefs =

  6x7 table

      Name       SignalType        LabelType          Group       Description     LabelColor     PixelL
    __________    ___________    _______________    _________    ____________    ___________    ______
```

```
    {'Car'  }    Image         Rectangle          {'None'}       {' '}       {0x0 char}    {0x0 c
    {'Car'  }    PointCloud    Cuboid             {'None'}       {' '}       {0x0 char}    {0x0 c
    {'Truck'}    Image         ProjectedCuboid    {'None'}       {' '}       {0x0 char}    {0x0 c
    {'Lane' }    Image         Line               {'None'}       {' '}       {0x0 char}    {0x0 c
    {'Road' }    Image         PixelLabel         {'None'}       {' '}       {0x0 char}    {[
    {'Sunny'}    Time          Scene              {'None'}       {' '}       {0x0 char}    {0x0 c
```

Create ROI label data for the first frame of the video.

```
numVideoFrames = numel(vidSource.Timestamp{1});
carData = cell(numVideoFrames,1);
laneData = cell(numVideoFrames,1);
truckData = cell(numVideoFrames,1);
carData{1} = [304 212 37 33];
laneData{1} = [70 458; 311 261];
truckData{1} = [309,215,33,24,330,211,33,24];
videoData = timetable(vidSource.Timestamp{1},carData,laneData, ...
                      'VariableNames',{'Car','Lane'});
```

Create ROI label data for the first point cloud in the sequence.

```
numPCFrames = numel(pcseqSource.Timestamp{1});
carData = cell(numPCFrames, 1);
carData{1} = [27.35 18.32 -0.11 4.25 4.75 3.45 0 0 0];
lidarData = timetable(pcseqSource.Timestamp{1},carData,'VariableNames',{'Car'});
```

Combine the ROI label data for both sources.

```
signalNames = [dataSource.SignalName];
roiData = vision.labeler.labeldata.ROILabelData(signalNames,{videoData,lidarData})
```

```
roiData = 

  ROILabelData with properties:

    video_01_city_c2s_fcw_10s: [204x2 timetable]
                lidarSequence: [34x1 timetable]
```

Create scene label data for the first 10 seconds of the driving scene.

```
sunnyData = seconds([0 10]);
labelNames = ["Sunny"];
sceneData = vision.labeler.labeldata.SceneLabelData(labelNames,{sunnyData})
```

```
sceneData = 

  SceneLabelData with properties:

    Sunny: [0 sec    10 sec]
```

Create a ground truth object from the signal sources, label definitions, and ROI and scene label data. You can import this object into the **Ground Truth Labeler** app for manual labeling or to run a labeling automation algorithm on it. You can also extract training data from this object for deep learning models by using the `gatherLabelData` function.

```
gTruth = groundTruthMultisignal(dataSource,labelDefs,roiData,sceneData)
```

```
gTruth = 

  groundTruthMultisignal with properties:

          DataSource: [1x2 vision.labeler.loading.MultiSignalSource]
    LabelDefinitions: [6x7 table]
        ROILabelData: [1x1 vision.labeler.labeldata.ROILabelData]
      SceneLabelData: [1x1 vision.labeler.labeldata.SceneLabelData]
```

See Also

Apps
Ground Truth Labeler

Objects
`SceneLabelData` | `groundTruthMultisignal`

Introduced in R2020a

SceneLabelData

Ground truth data for scene labels

Description

The `SceneLabelData` object stores ground truth data for scene label definitions defined in a `groundTruthMultisignal` object.

Creation

When you export a `groundTruthMultisignal` object from a **Ground Truth Labeler** app session, the `SceneLabelData` property of the exported object stores the scene labels as a `SceneLabelData` object. To create a `SceneLabelData` object programmatically, use the `vision.labeler.labeldata.SceneLabelData` function (described here).

Syntax

`sceneLabelData = vision.labeler.labeldata.SceneLabelData(labelNames, labelData)`

Description

`sceneLabelData = vision.labeler.labeldata.SceneLabelData(labelNames, labelData)` creates an object containing scene label data for multiple signals. The created object, `sceneLabelData`, contains properties with the scene label names listed in `labelNames`. These properties store the corresponding scene label data specified by `labelData`.

Input Arguments

`labelNames` — Scene label names
string array

Scene label names, specified as a string array. Specify the names of all scene labels present in the `groundTruthMultisignal` object you are creating. You can get the scene label names from an existing `groundTruthMultisignal` object by accessing the `LabelDefinitions` property of that object. Use this code and replace `gTruth` with the name of a `groundTruthMultisignal` object variable.

```
isSceneLabel = gTruth.LabelDefinitions.LabelType == 'Scene';
gTruth.LabelDefinitions.Name(isSceneLabel)
```

In an exported `groundTruthMultisignal` object, the `SceneLabelData` object contains a label data property for every scene label, even if some scene labels do not have label data.

The properties of the created `SceneLabelData` object have the names specified by `labelNames`.

Example: `["sunny" "rainy" "urban" "rural"]`

labelData — Scene label data for each label
cell array of `duration` matrices

Scene label data for each label, specified as a cell array of `duration` matrices. Each matrix in the cell array contains data for the scene label in the corresponding position of the `labelNames` input. The `SceneLabelData` object stores each matrix in a property that has the same name as that signal.

Each scene label matrix is of size *N*-by-2. Each row in this matrix corresponds to a time range for which that scene label has been applied. *N* is the number of time ranges. Rows in the matrix are of the form `[rangeStart, rangeEnd]`, where `rangeStart` and `rangeEnd` specify the start and end of a time range for an applied scene label.

Row elements are of type `duration` and must be within the range of the minimum and maximum of all the timestamps in the `groundTruthMultisignal` object. If a scene label is not applied, then specify an empty matrix.

Example: `seconds([0 5; 10 20])` specifies a `duration` matrix corresponding to one scene label in a `groundTruthMultisignal` object. Units are in seconds. The scene label has been applied from 0 to 5 seconds and again from 10 to 20 seconds, across all signals in the object. Specify this matrix as part of a cell array containing matrices for additional scene labels.

Properties

SceneLabelName1,...,SceneLabelNameN — Scene label data for each label (as separate properties)
`duration` matrices

Scene label data, specified as `duration` matrices. The `SceneLabelData` object contains one property per scene label definition, where each property contains a `duration` matrix of scene label data corresponding to that scene label.

When exporting a `SceneLabelData` object from a **Ground Truth Labeler** app session, the property names correspond to the scene label names stored in the `LabelDefinitions` property of the exported `groundTruthMultisignal` object.

When creating a `SceneLabelData` object programmatically, the `labelNames` and `labelData` input arguments define the property names and values of the created object.

Suppose you want to create a `groundTruthMultisignal` object containing scene labels that describe whether the scene is sunny, rainy, urban, or rural. Specify the scene labels in a string array, `labelNames`.

```
labelNames = ["sunny" "rainy" "urban" "rural"];
```

Store the label data for each scene label in a cell array of matrices, `labelData`. Each matrix contains the data for the corresponding scene label in `labelNames`.

```
labelData = {sunnyData,rainyData,urbanData,ruralData}

  1×4 cell array

    {1×2 duration}    {2×2 duration}    {0×0 duration}    {4×2 duration}
```

The `SceneLabelData` object, `sceneData`, stores this data in the property with the corresponding signal name. You can specify `sceneData` in the `SceneLabelData` property of a `groundTruthMultisignal` object.

```
sceneData = vision.labeler.labeldata.SceneLabelData(labelNames,labelData)
```

```
sceneData = 

  SceneLabelData with properties:

    rainy: [2×2 duration]
    sunny: [0 sec    10.15 sec]
    rural: [4×2 duration]
    urban: [0×0 duration]
```

Object Functions

labelDefinitionsAtTime	Get scene label definition names at specified timestamp
labelDataAtTime	Get scene label data at specified timestamps

Examples

Create Ground Truth from Multiple Signals

Create ground truth data for a video signal and a lidar point cloud sequence signal that captures the same driving scene. Specify the signal sources, label definitions, and ROI and scene label data.

Create the video data source from an MP4 file.

```
sourceName = '01_city_c2s_fcw_10s.mp4';
sourceParams = [];
vidSource = vision.labeler.loading.VideoSource;
vidSource.loadSource(sourceName,sourceParams);
```

Create the point cloud sequence source from a folder of point cloud data (PCD) files.

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

lidarSourceData = load(fullfile(pcSeqFolder,'timestamps.mat'));

sourceName = pcSeqFolder;
sourceParams = struct;
sourceParams.Timestamps = timestamps;

pcseqSource = vision.labeler.loading.PointCloudSequenceSource;
pcseqSource.loadSource(sourceName,sourceParams);
```

Combine the signal sources into an array.

```
dataSource = [vidSource pcseqSource]
```

```
dataSource = 

  1x2 heterogeneous MultiSignalSource (VideoSource, PointCloudSequenceSource) array with propert

    SourceName
```

```
    SourceParams
    SignalName
    SignalType
    Timestamp
    NumSignals
```

Create a table of label definitions for the ground truth data by using a `labelDefinitionCreatorMultisignal` object.

- The `Car` label definition appears twice. Even though `Car` is defined as a rectangle, you can draw rectangles only for image signals, such as videos. The `labelDefinitionCreatorMultisignal` object creates an additional row for lidar point cloud signals. In these signal types, you can draw `Car` labels as cuboids only.
- The label definitions have no descriptions and no assigned colors, so the `Description` and `LabelColor` columns are empty.
- The label definitions have no assigned groups, so for all label definitions, the corresponding cell in the `Group` column is set to `'None'`.
- `Road` is a pixel label definition, so the table includes a `PixelLabelID` column.
- No label definitions have sublabels or attributes, so the table does not include a `Hierarchy` column for storing such information.

```
ldc = labelDefinitionCreatorMultisignal;
addLabel(ldc,'Car','Rectangle');
addLabel(ldc,'Truck','ProjectedCuboid');
addLabel(ldc,'Lane','Line');
addLabel(ldc,'Road','PixelLabel');
addLabel(ldc,'Sunny','Scene');
labelDefs = create(ldc)
```

```
labelDefs =

  6x7 table

      Name       SignalType       LabelType         Group      Description    LabelColor     PixelI
    _________    __________    _______________    _________    ___________    __________    ______

    {'Car'  }    Image         Rectangle          {'None'}        {' '}       {0x0 char}    {0x0 c
    {'Car'  }    PointCloud    Cuboid             {'None'}        {' '}       {0x0 char}    {0x0 c
    {'Truck'}    Image         ProjectedCuboid    {'None'}        {' '}       {0x0 char}    {0x0 c
    {'Lane' }    Image         Line               {'None'}        {' '}       {0x0 char}    {0x0 c
    {'Road' }    Image         PixelLabel         {'None'}        {' '}       {0x0 char}    {[
    {'Sunny'}    Time          Scene              {'None'}        {' '}       {0x0 char}    {0x0 c
```

Create ROI label data for the first frame of the video.

```
numVideoFrames = numel(vidSource.Timestamp{1});
carData = cell(numVideoFrames,1);
laneData = cell(numVideoFrames,1);
truckData = cell(numVideoFrames,1);
carData{1} = [304 212 37 33];
laneData{1} = [70 458; 311 261];
truckData{1} = [309,215,33,24,330,211,33,24];
```

```
videoData = timetable(vidSource.Timestamp{1},carData,laneData, ...
                      'VariableNames',{'Car','Lane'});
```

Create ROI label data for the first point cloud in the sequence.

```
numPCFrames = numel(pcseqSource.Timestamp{1});
carData = cell(numPCFrames, 1);
carData{1} = [27.35 18.32 -0.11 4.25 4.75 3.45 0 0 0];
lidarData = timetable(pcseqSource.Timestamp{1},carData,'VariableNames',{'Car'});
```

Combine the ROI label data for both sources.

```
signalNames = [dataSource.SignalName];
roiData = vision.labeler.labeldata.ROILabelData(signalNames,{videoData,lidarData})
```

```
roiData = 

  ROILabelData with properties:

    video_01_city_c2s_fcw_10s: [204x2 timetable]
                lidarSequence: [34x1 timetable]
```

Create scene label data for the first 10 seconds of the driving scene.

```
sunnyData = seconds([0 10]);
labelNames = ["Sunny"];
sceneData = vision.labeler.labeldata.SceneLabelData(labelNames,{sunnyData})
```

```
sceneData = 

  SceneLabelData with properties:

    Sunny: [0 sec    10 sec]
```

Create a ground truth object from the signal sources, label definitions, and ROI and scene label data. You can import this object into the **Ground Truth Labeler** app for manual labeling or to run a labeling automation algorithm on it. You can also extract training data from this object for deep learning models by using the `gatherLabelData` function.

```
gTruth = groundTruthMultisignal(dataSource,labelDefs,roiData,sceneData)
```

```
gTruth = 

  groundTruthMultisignal with properties:

          DataSource: [1x2 vision.labeler.loading.MultiSignalSource]
    LabelDefinitions: [6x7 table]
        ROILabelData: [1x1 vision.labeler.labeldata.ROILabelData]
```

```
    SceneLabelData: [1x1 vision.labeler.labeldata.SceneLabelData]
```

Tips

- To create a `groundTruthMultisignal` object containing ROI label data but no scene label data, specify the `SceneLabelData` property as an empty array. To create this array, at the MATLAB command prompt, enter this code.

  ```
  sceneData = vision.labeler.labeldata.SceneLabelData.empty
  ```

See Also

Apps
Ground Truth Labeler

Objects
`ROILabelData` | `groundTruthMultisignal`

Introduced in R2020a

labelDefinitionsAtTime

Get scene label definition names at specified timestamp

Syntax

```
labelNames = labelDefinitionsAtTime(sceneData,timestamp)
```

Description

`labelNames = labelDefinitionsAtTime(sceneData,timestamp)` returns the scene label definition names that are applied at the specified timestamp in a `SceneLabelData` object, `sceneData`.

Examples

Get Scene Label Definition Names at Timestamp

Get the scene label definition names that are applied at the first timestamp of a `SceneLabelData` object.

Create a `SceneLabelData` object. The object has labels for specifying whether a scene is sunny, rainy, urban, or rural. The scene labels are applied at these time ranges.

- `"sunny"` — 0 to 5 seconds
- `"rainy"` — 6 to 10 seconds
- `"urban"` — 0 to 8 seconds
- `"rural"` — 9 to 10 seconds

```
labelNames = ["sunny" "rainy" "urban" "rural"];

sunnyData = seconds([0 5]);
rainyData = seconds([6 10]);
urbanData = seconds([0 8]);
ruralData = seconds([9 10]);

labelData = {sunnyData rainyData urbanData ruralData};
sceneData = vision.labeler.labeldata.SceneLabelData(labelNames,labelData);
```

Get the scene labels that are applied at the start of the time range, that is, the first timestamp.

```
tsStart = 0;
labelNamesAtStart = labelDefinitionsAtTime(sceneData,tsStart)
```

```
labelNamesAtStart = 1x2 string
    "sunny"    "urban"
```

Input Arguments

`sceneData` — Scene label data
`SceneLabelData` object

Scene label data, specified as a `SceneLabelData` object.

`timestamp` — Timestamp
`duration` scalar

Timestamp, specified as a `duration` scalar.

Example: `seconds(9.5)` specifies a `duration` scalar of 9.5 seconds.

Output Arguments

`labelNames` — Scene label definition names
string vector

Scene label definition names, returned as a string vector. The vector contains the names of scene label definitions at the input `timestamp` in the input `sceneData`.

See Also

`SceneLabelData` | `labelDataAtTime`

Introduced in R2020a

labelDataAtTime

Get scene label data at specified timestamps

Syntax

```
labelData = labelDataAtTime(sceneData,labelNames,timestamps)
```

Description

`labelData = labelDataAtTime(sceneData,labelNames,timestamps)` returns the scene label data at the specified timestamps and for the specified label names present in a `SceneLabelData` object, `sceneData`.

Examples

Get Scene Label Data at Timestamps

Get the scene label data present at a specific time range in a `SceneLabelData` object.

Create a `SceneLabelData` object. The object has labels for specifying whether a scene is sunny, rainy, urban, or rural. The scene labels are applied at these time ranges.

- `"sunny"` — 0 to 5 seconds
- `"rainy"` — 6 to 10 seconds
- `"urban"` — 0 to 8 seconds
- `"rural"` — 9 to 10 seconds

```
labelNames = ["sunny" "rainy" "urban" "rural"];

sunnyData = seconds([0 5]);
rainyData = seconds([6 10]);
urbanData = seconds([0 8]);
ruralData = seconds([9 10]);

labelData = {sunnyData rainyData urbanData ruralData};
sceneData = vision.labeler.labeldata.SceneLabelData(labelNames,labelData);
```

Get the label data for the weather-related scene labels (`"sunny"` and `"rainy"`) over the time range that the `"urban"` scene label is applied.

```
weatherLabelNames = ["sunny" "rainy"];
urbanTimestamps = seconds(0:8);
weatherLabelData = labelDataAtTime(sceneData,weatherLabelNames,urbanTimestamps)
```

```
weatherLabelData=9×2 timetable
    timeStamps    sunny    rainy
    __________    _____    _____

    0 sec         true     false
    1 sec         true     false
```

```
    2 sec           true     false
    3 sec           true     false
    4 sec           true     false
    5 sec           true     false
    6 sec           false    true
    7 sec           false    true
    8 sec           false    true
```

Input Arguments

sceneData — Scene label data
SceneLabelData object

Scene label data, specified as a SceneLabelData object.

labelNames — Scene label names
string vector

Scene label names, specified as a string vector. The scene label names must be present in the sceneData input.

Example: ["sunny" "rainy"]

timestamps — Timestamps
duration vector

Timestamps, specified as a duration vector.

Example: seconds(5:10) specifies a duration vector from 5 to 10 seconds.

Output Arguments

labelData — Scene label data at specified timestamps
timetable

Scene label data at specified timestamps, returned as a timetable. The first column contains the timestamps specified by the timestamps input. The remaining columns correspond to the scene labels specified by the labelNames input. These columns contain logical 1 (true) and logical 0 (false) values that specify the scene labels present at each timestamp in the input sceneData.

See Also
SceneLabelData | labelDefinitionsAtTime

Introduced in R2020a

extendedObjectMesh

Mesh representation of extended object

Description

The `extendedObjectMesh` represents the 3-D geometry of an object. The 3-D geometry is represented by faces and vertices. Use these object meshes to specify the geometry of an `actor` for simulating lidar sensor data using `lidarPointCloudGenerator`.

Creation

Syntax

```
mesh = extendedObjectMesh('cuboid')
mesh = extendedObjectMesh('cylinder')
mesh = extendedObjectMesh('cylinder',n)
mesh = extendedObjectMesh('sphere')
mesh = extendedObjectMesh('sphere',n)
mesh = extendedObjectMesh(vertices,faces)
```

Description

`mesh = extendedObjectMesh('cuboid')` returns an `extendedObjectMesh` object, that defines a cuboid with unit dimensions. The origin of the cuboid is located at its geometric center.

`mesh = extendedObjectMesh('cylinder')` returns a hollow cylinder mesh with unit dimensions. The cylinder mesh has 20 equally spaced vertices around its circumference. The origin of the cylinder is located at its geometric center. The height is aligned with the *z*-axis.

`mesh = extendedObjectMesh('cylinder',n)` returns a cylinder mesh with `n` equally spaced vertices around its circumference.

`mesh = extendedObjectMesh('sphere')` returns a sphere mesh with unit dimensions. The sphere mesh has 119 vertices and 180 faces. The origin of the sphere is located at its center.

`mesh = extendedObjectMesh('sphere',n)` additionally allows you to specify the resolution, `n`, of the spherical mesh. The sphere mesh has $(n + 1)^2 - 2$ vertices and $2n(n - 1)$ faces.

`mesh = extendedObjectMesh(vertices,faces)` returns a mesh from faces and vertices. `vertices` and `faces` set the `Vertices` and `Faces` properties respectively.

Properties

`Vertices` — Vertices of defined object
N-by-3 matrix of real scalar

Vertices of the defined object, specified as an *N*-by-3 matrix of real scalars. *N* is the number of vertices. The first, second, and third element of each row represents the x-, y-, and z-position of each vertex, respectively.

Faces — Faces of defined object
M-by-3 matrix of positive integer

Faces of the defined object, specified as a *M*-by-3 array of positive integers. *M* is the number of faces. The three elements in each row are the vertex IDs of the three vertices forming the triangle face. The ID of the vertex is its corresponding row number specified in the `Vertices` property.

Object Functions

Use the object functions to develop new meshes.

translate	Translate mesh along coordinate axes
rotate	Rotate mesh about coordinate axes
scale	Scale mesh in each dimension
applyTransform	Apply forward transformation to mesh vertices
join	Join two object meshes
scaleToFit	Auto-scale object mesh to match specified cuboid dimensions
show	Display the mesh as a patch on the current axes

Examples

Create and Translate Cuboid Mesh

This example shows how to create an `extendedObjectMesh` object and translate the object.

Construct a cuboid mesh.

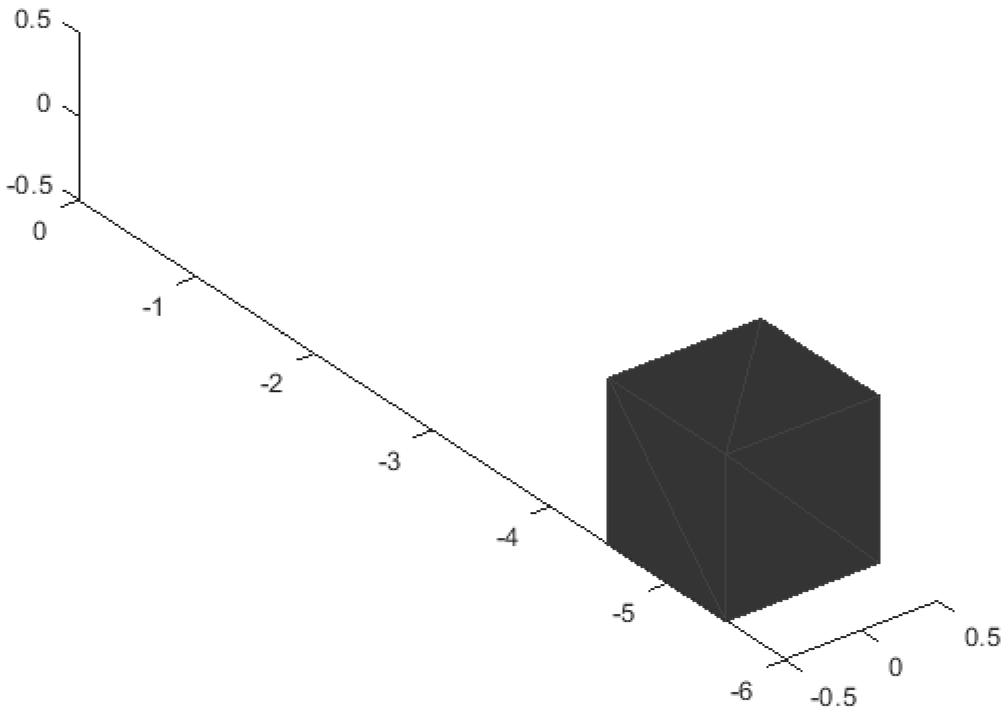
```
mesh = extendedObjectMesh('cuboid');
```

Translate the mesh by 5 units along the negative *y* axis.

```
mesh = translate(mesh,[0 -5 0]);
```

Visualize the mesh.

```
ax = show(mesh);
ax.YLim = [-6 0];
```



Create and Visualize Cylinder Mesh

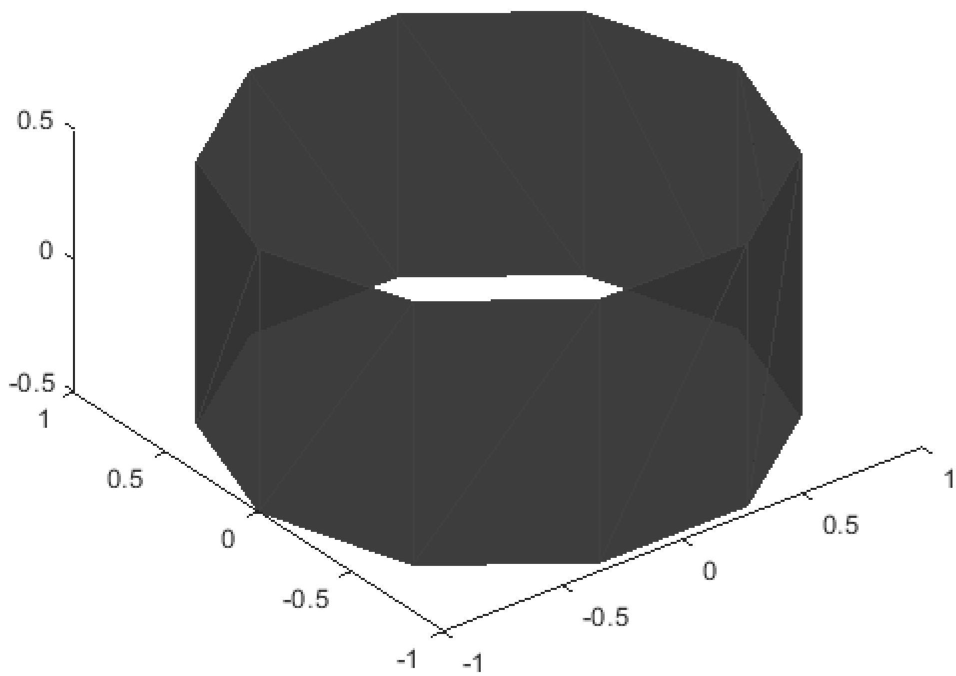
This example shows how to create an `extendedObjectMesh` object and visualize the object.

Construct a cylinder mesh.

```
mesh = extendedObjectMesh('cylinder');
```

Visualize the mesh.

```
ax = show(mesh);
```



Create and Auto-scale Sphere Mesh

This example shows how to create an `extendedObjectMesh` object and auto-scale the object to the required dimensions.

Construct a sphere mesh of unit dimensions.

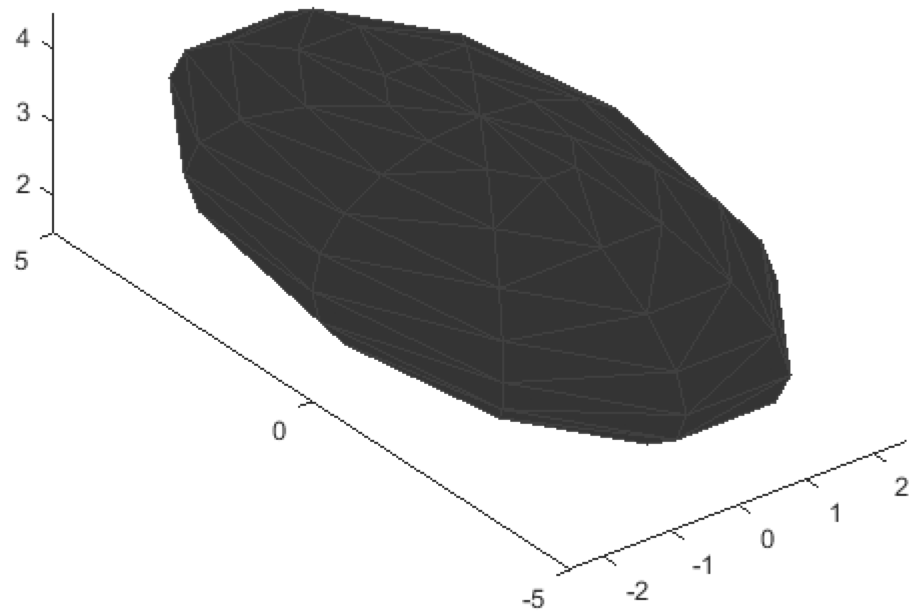
```
sph = extendedObjectMesh('sphere');
```

Auto-scale the mesh to the dimensions in `dims`.

```
dims = struct('Length',5,'Width',10,'Height',3,'OriginOffset',[0 0 -3]);
sph = scaleToFit(sph,dims);
```

Visualize the mesh.

```
show(sph);
```



Pre-built Meshes

You can use the prebuilt meshes as a starting point to develop your own meshes. The table lists the details of the meshes.

`driving.scenario.bicycleMesh`	Mesh representation of bicycle in driving scenario
`driving.scenario.carMesh`	Mesh representation of car in driving scenario.
`driving.scenario.pedestrianMesh`	Mesh representation of pedestrian in driving scenario.
`driving.scenario.truckMesh`	Mesh representation of truck in driving scenario.

You can view the source files of the meshes to understand how to develop new meshes. At the MATLAB command line, enter:

```
edit driving.scenario.XXXXMesh
```

Replace `XXXXMesh` with the name of the mesh.

See Also

Objects
`drivingScenario` | `lidarPointCloudGenerator`

Functions
`actor` | `driving.scenario.bicycleMesh` | `driving.scenario.carMesh` | `driving.scenario.pedestrianMesh` | `driving.scenario.truckMesh` | `roadMesh` | `vehicle`

Introduced in R2020a

applyTransform

Apply forward transformation to mesh vertices

Syntax

```
transformedMesh = applyTransform(mesh,T)
```

Description

`transformedMesh = applyTransform(mesh,T)` applies the forward transformation matrix `T` to the vertices of the object mesh.

Examples

Create and Transform Cuboid Mesh

This example shows how to create an `extendedObjectMesh` object and transform the object in a way defined by a given transformation matrix.

Create a cuboid mesh of unit dimensions.

```
cuboid = extendedObjectMesh('cuboid');
```

Create a transformation matrix that is a combination of a translation, a scaling, and a rotation.

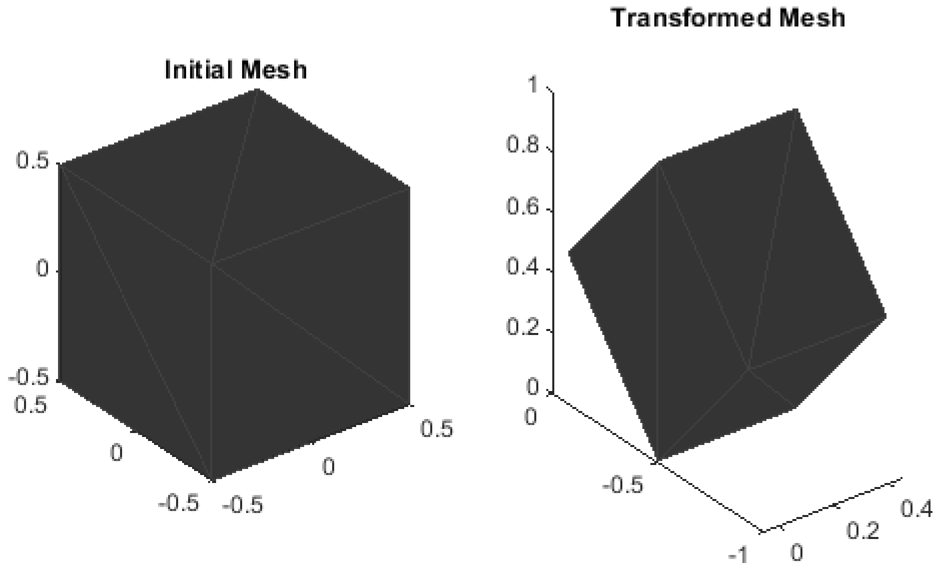
```
T = makehgtform('translate',[0.2 -0.5 0.5],'scale',[0.5 0.6 0.7],'xrotate',pi/4);
```

Transform the mesh.

```
transformedCuboid = applyTransform(cuboid,T);
```

Visualize the mesh.

```
subplot(1,2,1);
show(cuboid);
title('Initial Mesh');
subplot(1,2,2);
show(transformedCuboid);
title('Transformed Mesh');
```



Input Arguments

mesh — Extended object mesh
extendedObjectMesh object

Extended object mesh, specified as an extendedObjectMesh object.

T — Transformation matrix
4-by-4 matrix

Transformation matrix applied on the object mesh, specified as a 4-by-4 matrix. The 3-D coordinates of each point in the object mesh is transformed according to this formula:

```
[xT; yT; zT; 1] = T*[x; y; z; 1]
```

xT, yT, and zT are the transformed 3-D coordinates of the point.

Data Types: single | double

Output Arguments

transformedMesh — Transformed object mesh
extendedObjectMesh object

Transformed object mesh, returned as an extendedObjectMesh object.

See Also

Objects
`extendedObjectMesh`

Functions
`join` | `rotate` | `scale` | `scaleToFit` | `show` | `translate`

Introduced in R2020a

join

Join two object meshes

Syntax

```
joinedMesh = join(mesh1,mesh2)
```

Description

`joinedMesh = join(mesh1,mesh2)` joins the object meshes `mesh1` and `mesh2` and returns `joinedMesh` with the combined objects.

Examples

Create and Join Two Object Meshes

This example shows how to create `extendedObjectMesh` objects and join them together.

Construct two meshes of unit dimensions.

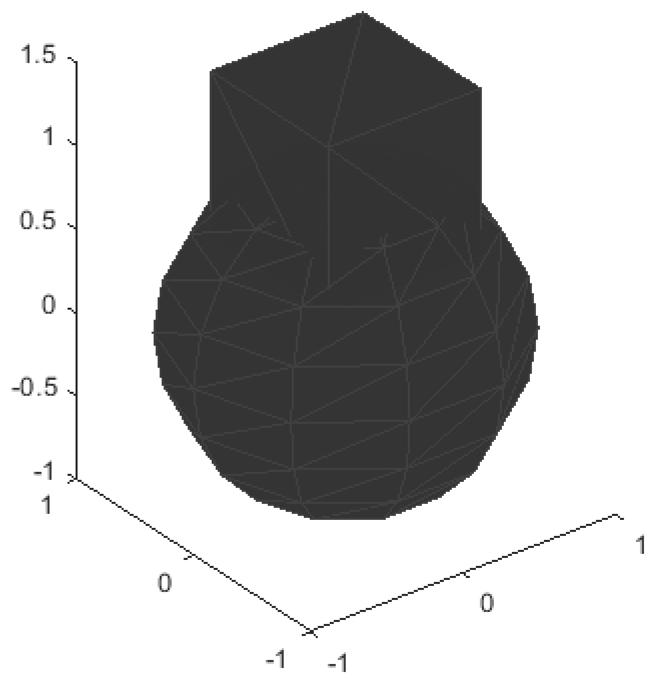
```
sph = extendedObjectMesh('sphere');
cub = extendedObjectMesh('cuboid');
```

Join the two meshes.

```
cub = translate(cub,[0 0 1]);
sphCub = join(sph,cub);
```

Visualize the final mesh.

```
show(sphCub);
```



Input Arguments

`mesh1` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`mesh2` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

Output Arguments

`joinedMesh` — Joined object mesh
`extendedObjectMesh` object

Joined object mesh, specified as an `extendedObjectMesh` object.

See Also

Objects
`extendedObjectMesh`

Functions
`applyTransform` | `rotate` | `scale` | `scaleToFit` | `show` | `translate`

Introduced in R2020a

rotate

Rotate mesh about coordinate axes

Syntax

`rotatedMesh = rotate(mesh,orient)`

Description

`rotatedMesh = rotate(mesh,orient)` rotate the `mesh` object by an orientation, `orient`.

Examples

Create and Rotate Cuboid Mesh

This example shows how to create an `extendedObjectMesh` object and rotate the object.

Construct a cuboid mesh.

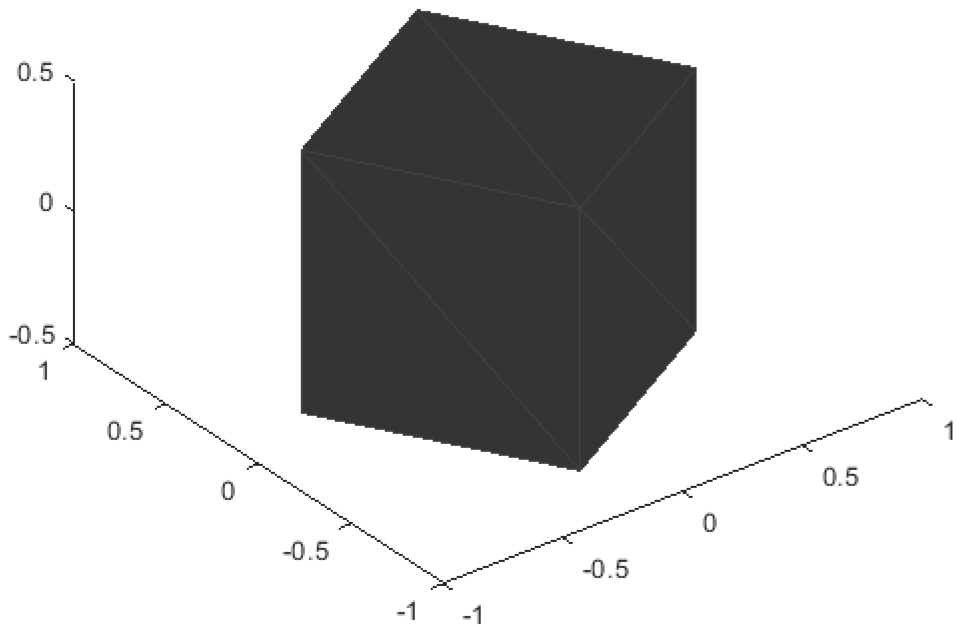
```
mesh = extendedObjectMesh('cuboid');
```

Rotate the mesh by 30 degrees around the *z* axis.

```
mesh = rotate(mesh,[30 0 0]);
```

Visualize the mesh.

```
ax = show(mesh);
```



Input Arguments

`mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`orient` — Description of rotation
3-by-3 orthonormal matrix | quaternion | 1-by-3 vector

Description of rotation for an object mesh, specified as:

- 3-by-3 orthonormal rotation matrix
- `quaternion`
- 1-by-3 vector, where the elements are positive rotations in degrees about the *z*, *y*, and *x* axes, in that order.

Output Arguments

`rotatedMesh` — Rotated object mesh
`extendedObjectMesh` object

Rotated object mesh, returned as an `extendedObjectMesh` object.

See Also

Objects
extendedObjectMesh

Functions
applyTransform | join | scale | scaleToFit | show | translate

Introduced in R2020a

scale

Scale mesh in each dimension

Syntax

```
scaledMesh = scale(mesh,scaleFactor)
scaledMesh = scale(mesh,[sx sy sz])
```

Description

`scaledMesh = scale(mesh,scaleFactor)` scales the object mesh by `scaleFactor`. `scaleFactor` can be the same for all dimensions or defined separately as elements of a 1-by-3 vector in the order *x*, *y*, and *z*.

`scaledMesh = scale(mesh,[sx sy sz])` scales the object mesh along the dimensions *x*, *y*, and *z* by the scaling factors `sx`, `sy`, and `sz`.

Examples

Create and Scale Cuboid Mesh

This example shows how to create an `extendedObjectMesh` object and scale the object.

Construct a cuboid mesh of unit dimensions.

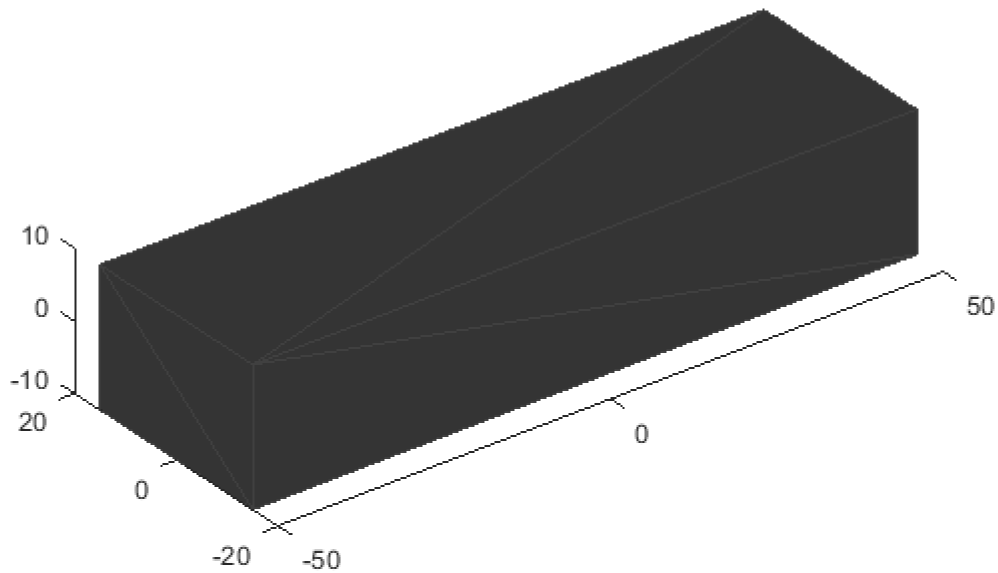
```
cuboid = extendedObjectMesh('cuboid');
```

Scale the mesh by different factors along each of the three axes.

```
scaledCuboid = scale(cuboid,[100 30 20]);
```

Visualize the mesh.

```
show(scaledCuboid);
```



Input Arguments

`mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`scaleFactor` — Scaling factor
positive real scalar | 1-by-3 vector

Scaling factor for the object mesh, specified as a single positive real value or as a 1-by-3 vector in the order *x*, *y*, and *z*.

Data Types: `single` | `double`

`sx` — Scaling factor for *x*-axis
positive real scalar

Scaling factor for *x*-axis, specified as a positive real scalar.

Data Types: `single` | `double`

`sy` — Scaling factor for *y*-axis
positive real scalar

Scaling factor for *y*-axis, specified as a positive real scalar.

Data Types: `single` | `double`

`sz` — Scaling factor for *z*-axis
positive real scalar

Scaling factor for *z*-axis, specified as a positive real scalar.

Data Types: `single` | `double`

Output Arguments

`scaledMesh` — Scaled object mesh
`extendedObjectMesh` object

Scaled object mesh, returned as an `extendedObjectMesh` object.

See Also

Objects
`extendedObjectMesh`

Functions
`applyTransform` | `join` | `rotate` | `scaleToFit` | `show` | `translate`

Introduced in R2020a

scaleToFit

Auto-scale object mesh to match specified cuboid dimensions

Syntax

```
scaledMesh = scaleToFit(mesh,dims)
```

Description

`scaledMesh = scaleToFit(mesh,dims)` auto-scales the object mesh to match the dimensions of a cuboid specified in the structure `dims`.

Examples

Create and Auto-scale Sphere Mesh

This example shows how to create an `extendedObjectMesh` object and auto-scale the object to the required dimensions.

Construct a sphere mesh of unit dimensions.

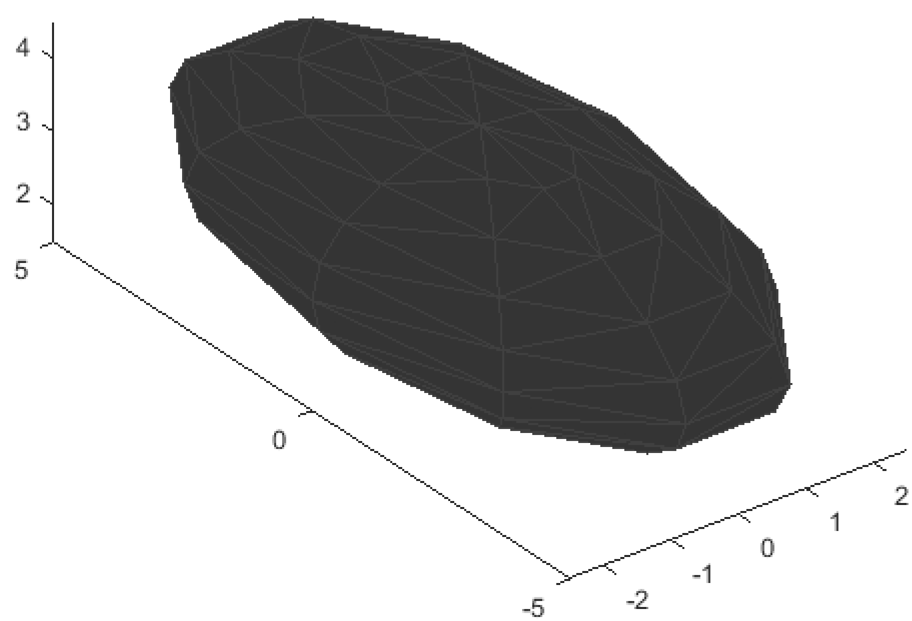
```
sph = extendedObjectMesh('sphere');
```

Auto-scale the mesh to the dimensions in `dims`.

```
dims = struct('Length',5,'Width',10,'Height',3,'OriginOffset',[0 0 -3]);
sph = scaleToFit(sph,dims);
```

Visualize the mesh.

```
show(sph);
```



Input Arguments

`mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh`

`dims` — Cuboid dimensions
struct

Dimensions of the cuboid to scale an object mesh, specified as a `struct` with these fields:

- `Length` – Length of the cuboid
- `Width` – Width of the cuboid
- `Height` – Height of the cuboid
- `OriginOffset` – Origin offset in 3-D coordinates

All the dimensions are in meters.

Data Types: `struct`

Output Arguments

`scaledMesh` — Scaled object mesh
`extendedObjectMesh` object

Scaled object mesh, returned as an `extendedObjectMesh` object.

See Also

Objects
`extendedObjectMesh`

Functions
`applyTransform` | `join` | `rotate` | `scale` | `show` | `translate`

Introduced in R2020a

show

Display the mesh as a patch on the current axes

Syntax

```
show(mesh)
show(mesh,ax)
ax = show(mesh)
```

Description

`show(mesh)` displays the `extendedObjectMesh` as a patch on the current axes. If there are no active axes, the function creates new axes.

`show(mesh,ax)` displays the object mesh as a patch on the axes `ax`.

`ax = show(mesh)` optionally outputs the handle to the axes where the mesh was plotted.

Examples

Create and Translate Cuboid Mesh

This example shows how to create an `extendedObjectMesh` object and translate the object.

Construct a cuboid mesh.

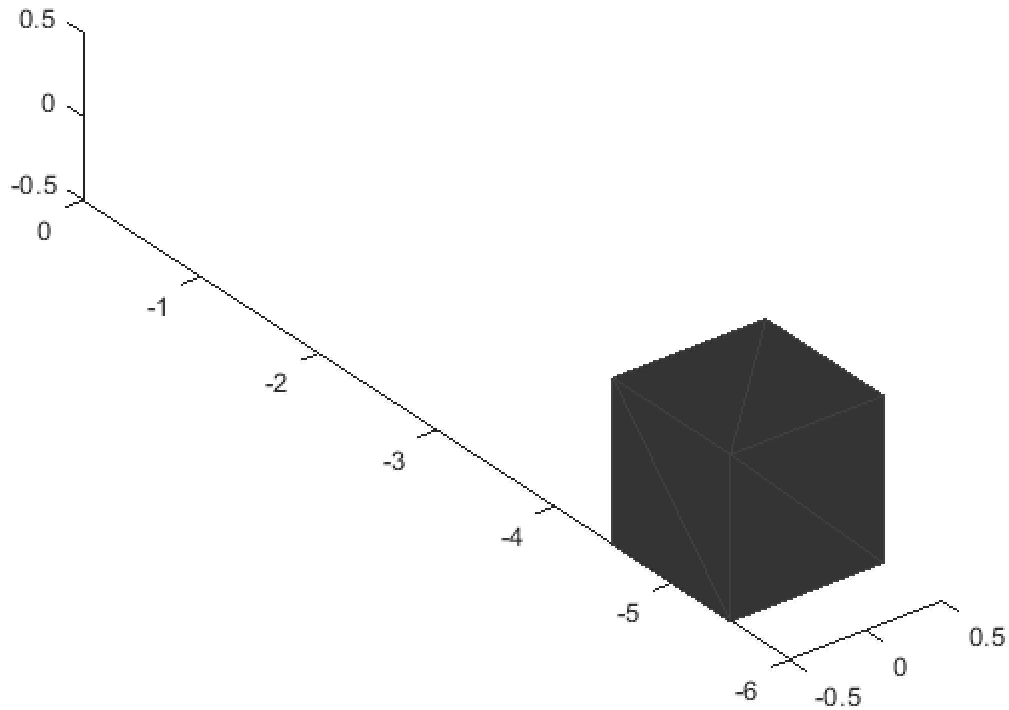
```
mesh = extendedObjectMesh('cuboid');
```

Translate the mesh by 5 units along the negative *y* axis.

```
mesh = translate(mesh,[0 -5 0]);
```

Visualize the mesh.

```
ax = show(mesh);
ax.YLim = [-6 0];
```



Input Arguments

`mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

ax — Current axes
`axes`

Current axes, specified as an `axes` object.

See Also

Objects
`extendedObjectMesh`

Functions
`applyTransform` | `join` | `rotate` | `scale` | `scaleToFit` | `translate`

Introduced in R2020a

translate

Translate mesh along coordinate axes

Syntax

`translatedMesh = translate(mesh,deltaPos)`

Description

`translatedMesh = translate(mesh,deltaPos)` translates the object mesh by the distances specified by `deltaPos` along the coordinate axes.

Examples

Create and Translate Cuboid Mesh

This example shows how to create an `extendedObjectMesh` object and translate the object.

Construct a cuboid mesh.

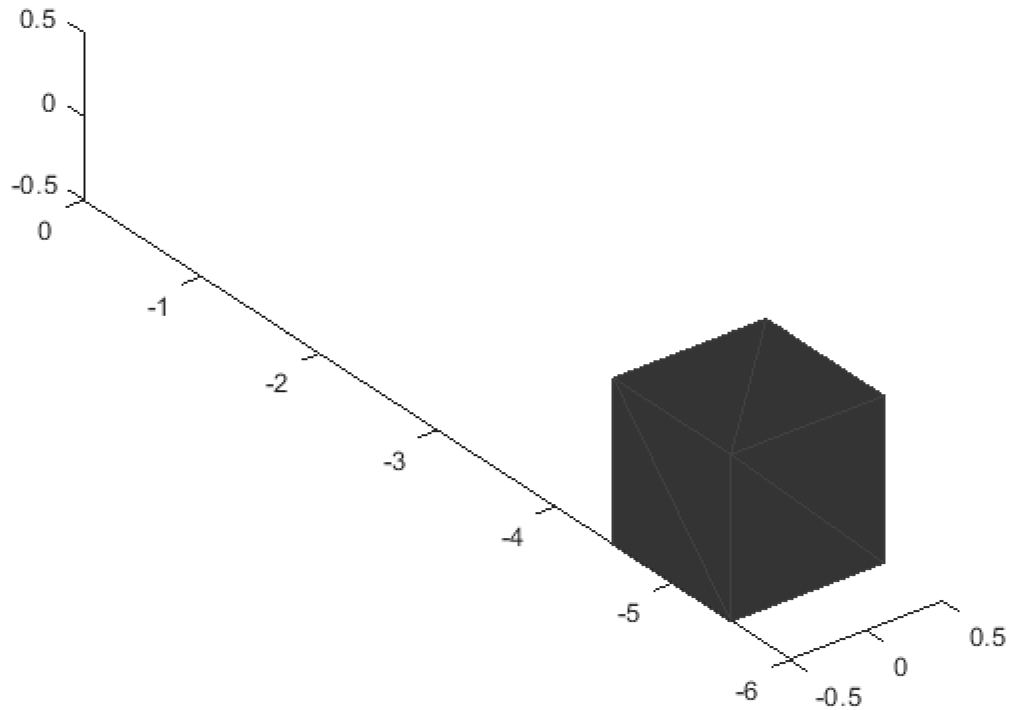
```
mesh = extendedObjectMesh('cuboid');
```

Translate the mesh by 5 units along the negative *y* axis.

```
mesh = translate(mesh,[0 -5 0]);
```

Visualize the mesh.

```
ax = show(mesh);
ax.YLim = [-6 0];
```



Input Arguments

`mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`deltaPos` — Translation vector
three-element, real-valued vector

Translation vector for an object mesh, specified as a three-element, real-valued vector. The three elements in the vector define the translation along the *x*, *y*, and *z* axes.

Data Types: `single` | `double`

Output Arguments

`translatedMesh` — Translated object mesh
`extendedObjectMesh` object

Translated object mesh, returned as an `extendedObjectMesh` object.

See Also

Objects
`extendedObjectMesh`

Functions
`applyTransform` | `join` | `rotate` | `scale` | `scaleToFit` | `show`

Introduced in R2020a

Scene Dimensions

Curved Road

Curved road 3D environment

Description

The **Curved Road** scene is a 3D environment of a curved highway loop. The scene is rendered using the Unreal Engine from Epic Games.

To simulate a driving algorithm in this scene:

1. Add a Simulation 3D Scene Configuration block to your Simulink model.
2. In this block, set the **Scene source** parameter to `Default Scenes`.
3. Set the enabled **Scene name** parameter to `Curved road`.

Explore Curved Road Scene

Explore the 3D Curved Road scene and inspect its dimensions by using a corresponding 2D top-view image of the scene.

You can use this image to inspect the scene before simulation and choose starting coordinates for vehicles. For details on using these images to select waypoints for path-following applications, see the “Select Waypoints for Unreal Engine Simulation” example.

Load the 2D spatial referencing object that corresponds to the scene. This `imref2d` (Image Processing Toolbox) object describes the relationship between the pixels in the image and the world coordinates of the scene.

```
data = load('sim3d_SpatialReferences.mat');
spatialRef = data.spatialReference.CurvedRoad
```

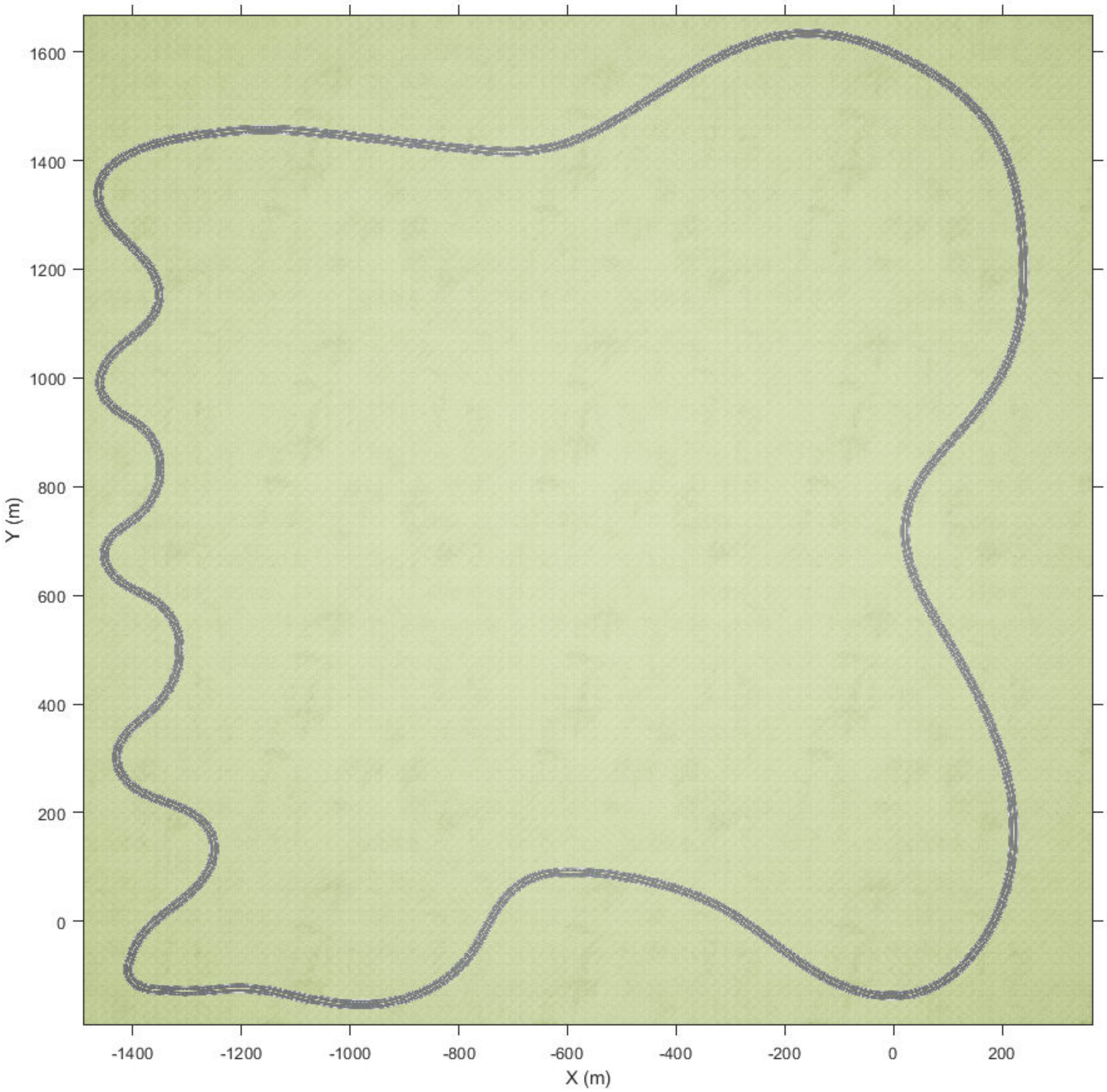
```
spatialRef = 
  imref2d with properties:
```

```
        XWorldLimits: [-1.4918e+03 367.9000]
        YWorldLimits: [-191.4200 1.6683e+03]
           ImageSize: [4845 4845]
PixelExtentInWorldX: 0.3838
PixelExtentInWorldY: 0.3838
ImageExtentInWorldX: 1.8597e+03
ImageExtentInWorldY: 1.8597e+03
   XIntrinsicLimits: [0.5000 4.8455e+03]
   YIntrinsicLimits: [0.5000 4.8455e+03]
```

Display the image corresponding to the scene. Use the spatial referencing object to display the axes in the world coordinates of the scene. Units are in meters.

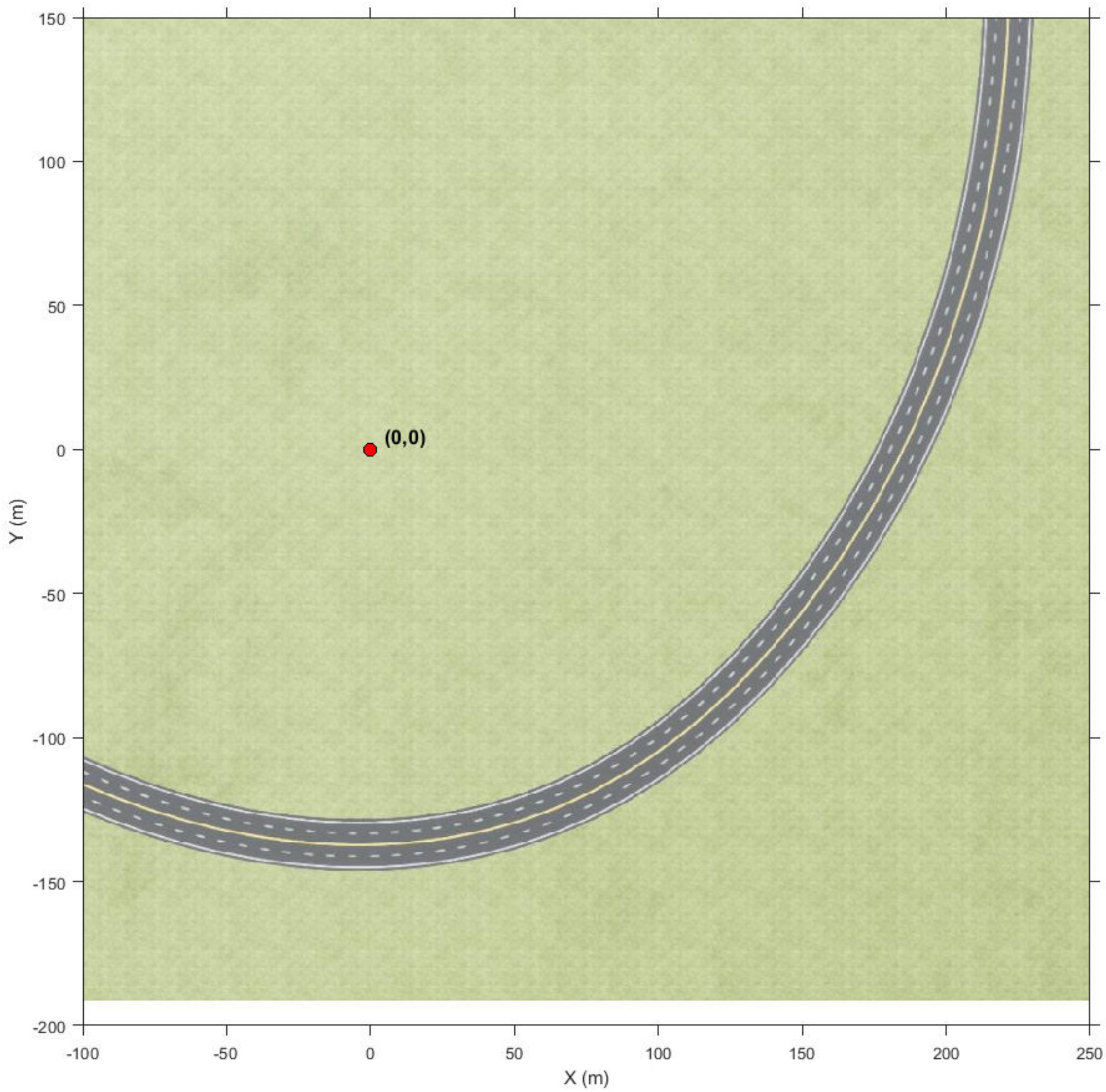
By default, the `imshow` function displays *Y*-axis values that increase from top to bottom. To align with the Automated Driving Toolbox™ world coordinate system, set the *Y*-direction to `'normal'` so that *Y*-axis values increase from bottom to top.

```
figure
fileName = 'sim3d_CurvedRoad.jpg';
I = imshow(fileName,spatialRef);
set(gca,'YDir','normal')
xlabel('X (m)')
ylabel('Y (m)')
```



Zoom in on the origin of the scene. Place a marker at the origin.

```
xlim([-100 250])
ylim([-200 150])

hold on
plot(0,0,'o','MarkerFaceColor','r','MarkerEdgeColor','k','MarkerSize',8)
offset = 5; % px
text(offset,offset,'(0,0)','Color','k','FontWeight','bold','FontSize',12)
hold off
```



Tips

- If you have the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, then you can modify this scene. In the Unreal Engine project file that comes with the support package, this scene is named `HwCurve`.

  For more details on customizing scenes, see “Customize Unreal Engine Scenes for Automated Driving”.

See Also

Double Lane Change | Large Parking Lot | Open Surface | Parking Lot | Straight Road | US City Block | US Highway | Virtual Mcity

Topics

"Unreal Engine Simulation for Automated Driving"
"Unreal Engine Simulation Environment Requirements and Limitations"
"Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox"
"Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer"

Double Lane Change

Double lane change 3D environment

Description

The **Double Lane Change** scene is a 3D environment of a straight road containing cones, traffic signs, and barrels. The cones are set up for a vehicle to perform a double lane change maneuver. The scene is rendered using the Unreal Engine from Epic Games.

To simulate a driving algorithm in this scene:

1. Add a Simulation 3D Scene Configuration block to your Simulink model.
2. In this block, set the **Scene source** parameter to `Default Scenes`.
3. Set the enabled **Scene name** parameter to `Double lane change`.

Explore Double Lane Change Scene

Explore the 3D Double Lane Change scene and inspect its dimensions by using a corresponding 2D top-view image of the scene.

You can use this image to inspect the scene before simulation and choose starting coordinates for vehicles. For details on using these images to select waypoints for path-following applications, see the “Select Waypoints for Unreal Engine Simulation” example.

Load the 2D spatial referencing object that corresponds to the scene. This `imref2d` (Image Processing Toolbox) object describes the relationship between the pixels in the image and the world coordinates of the scene.

```
data = load('sim3d_SpatialReferences.mat');
spatialRef = data.spatialReference.DoubleLaneChange
```

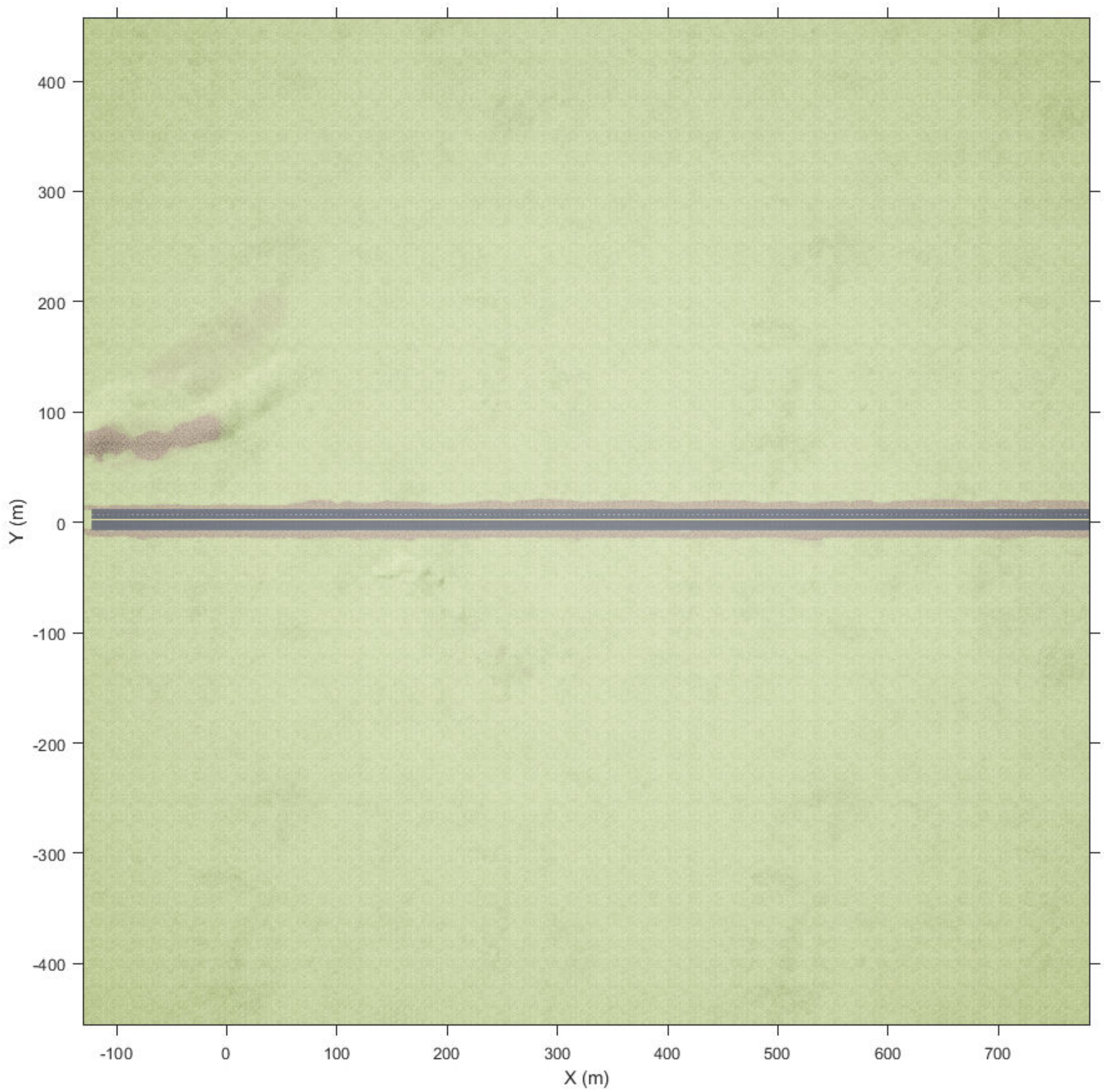
```
spatialRef = 
  imref2d with properties:

           XWorldLimits: [-130.5500 783.3500]
           YWorldLimits: [-456.1500 457.7500]
              ImageSize: [4845 4845]
    PixelExtentInWorldX: 0.1886
    PixelExtentInWorldY: 0.1886
    ImageExtentInWorldX: 913.9000
    ImageExtentInWorldY: 913.9000
       XIntrinsicLimits: [0.5000 4.8455e+03]
       YIntrinsicLimits: [0.5000 4.8455e+03]
```

Display the image corresponding to the scene. Use the spatial referencing object to display the axes in the world coordinates of the scene. Units are in meters.

By default, the `imshow` function displays *Y*-axis values that increase from top to bottom. To align with the Automated Driving Toolbox™ world coordinate system, set the *Y*-direction to `'normal'` so that *Y*-axis values increase from bottom to top.

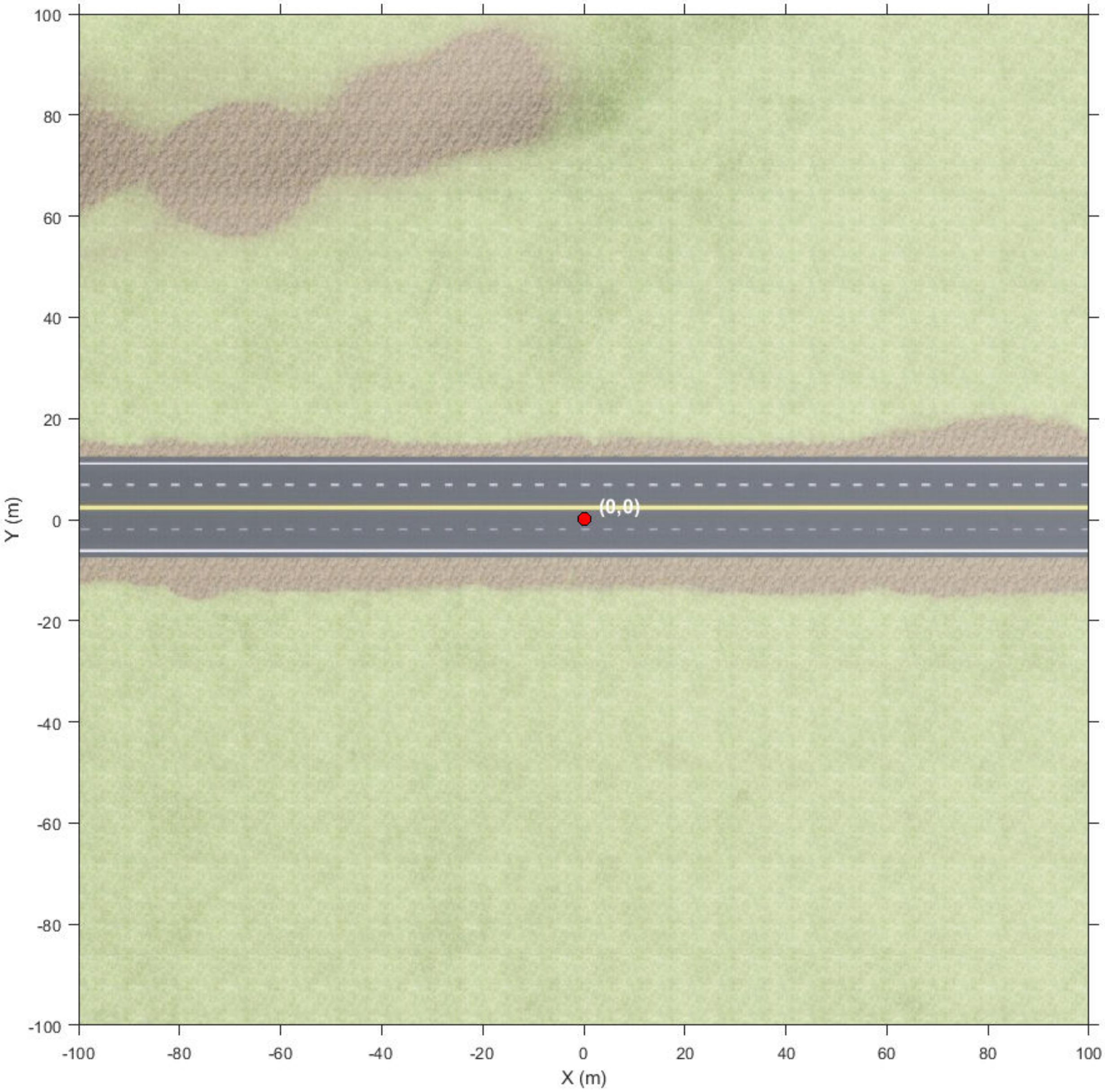
The image displays only the area of the scene containing the parking lot. The full scene has a length and width of 2016 meters.

```
figure
fileName = 'sim3d_DoubleLaneChange.jpg';
I = imshow(fileName,spatialRef);
set(gca,'YDir','normal')
xlabel('X (m)')
ylabel('Y (m)')
```



Zoom in on the origin of the scene. Place a marker at the origin. If you place a vehicle at the scene origin and set the vehicle's yaw angle to `0`, the traffic cones for performing the double lane change maneuver are directly in front of the vehicle.

```
xlim([-100 100])
ylim([-100 100])

hold on
plot(0,0,'o','MarkerFaceColor','r','MarkerEdgeColor','k','MarkerSize',8)
offset = 3; % px
text(offset,offset,'(0,0)','Color','w','FontWeight','bold','FontSize',12)
hold off
```



Tips

- If you have the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, then you can modify this scene. In the Unreal Engine project file that comes with the support package, this scene is named `DblLnChng`.

  For more details on customizing scenes, see “Customize Unreal Engine Scenes for Automated Driving”.

See Also

Curved Road | Large Parking Lot | Open Surface | Parking Lot | Straight Road | US City Block | US Highway | Virtual Mcity

Topics

"Unreal Engine Simulation for Automated Driving"
"Unreal Engine Simulation Environment Requirements and Limitations"
"Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox"
"Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer"

Large Parking Lot

Large parking lot 3D environment

Description

The **Large Parking Lot** scene is a 3D environment of a large parking lot that contains cones, curbs, traffic signs, and parked vehicles. The scene is rendered using the Unreal Engine from Epic Games.

To simulate a driving algorithm in this scene:

1. Add a Simulation 3D Scene Configuration block to your Simulink model.
2. In this block, set the **Scene source** parameter to `Default Scenes`.
3. Set the enabled **Scene name** parameter to `Large parking lot`.

Explore Large Parking Lot Scene

Explore the 3D Large Parking Lot scene and inspect its dimensions by using a corresponding 2D top-view image of the scene.

You can use this image to inspect the scene before simulation and choose starting coordinates for vehicles. For details on using these images to select waypoints for path-following applications, see the “Select Waypoints for Unreal Engine Simulation” example.

Load the 2D spatial referencing object that corresponds to the scene. This `imref2d` (Image Processing Toolbox) object describes the relationship between the pixels in the image and the world coordinates of the scene.

```
data = load('sim3d_SpatialReferences.mat');
spatialRef = data.spatialReference.LargeParkingLot
```

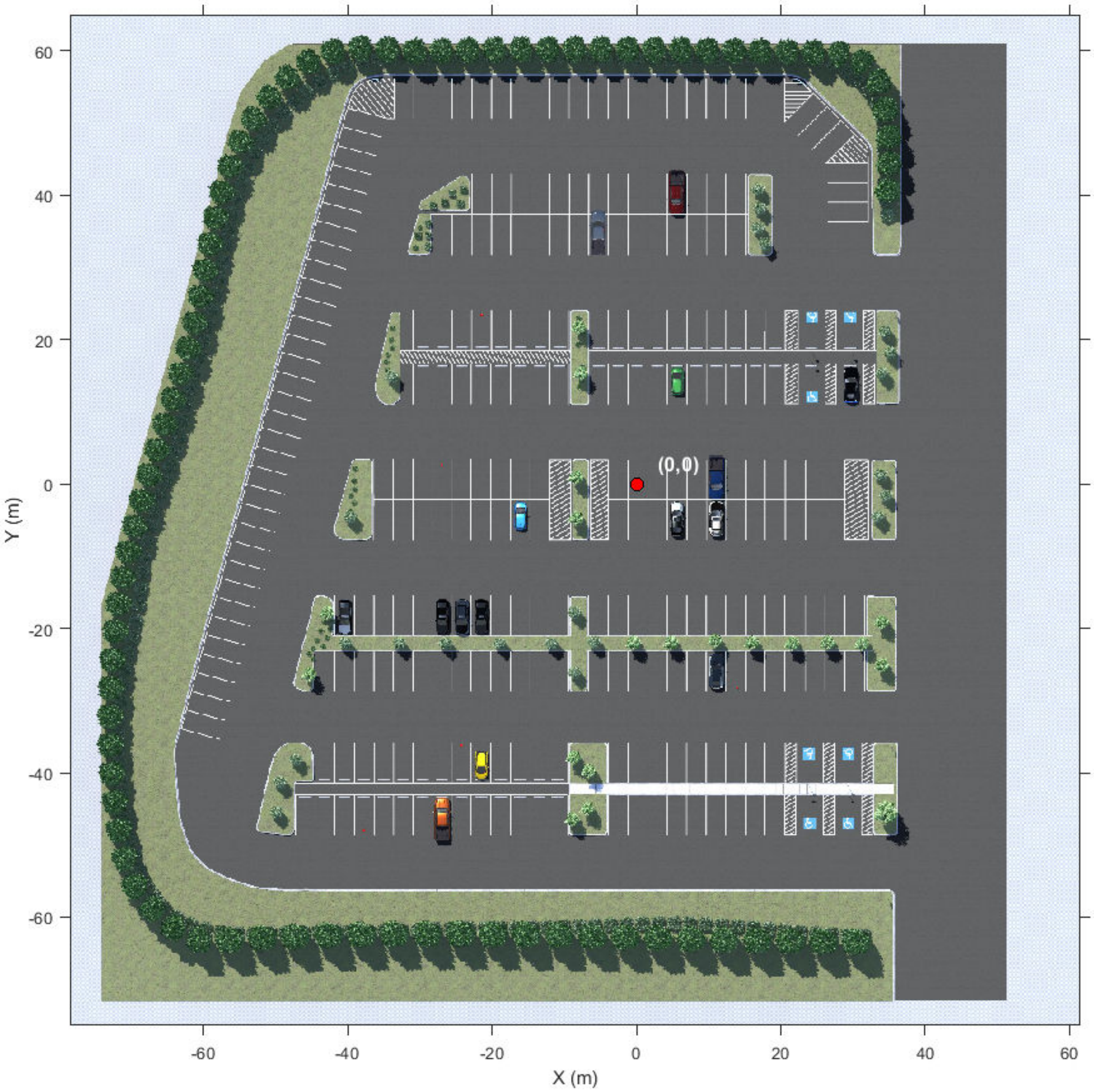
```
spatialRef = 
  imref2d with properties:

           XWorldLimits: [-78.5000 61.5000]
           YWorldLimits: [-75 65]
              ImageSize: [4845 4845]
    PixelExtentInWorldX: 0.0289
    PixelExtentInWorldY: 0.0289
    ImageExtentInWorldX: 140
    ImageExtentInWorldY: 140
       XIntrinsicLimits: [0.5000 4.8455e+03]
       YIntrinsicLimits: [0.5000 4.8455e+03]
```

Display the image corresponding to the scene. Use the spatial referencing object to display the axes in the world coordinates of the scene. Units are in meters.

By default, the `imshow` function displays *Y*-axis values that increase from top to bottom. To align with the Automated Driving Toolbox™ world coordinate system, set the *Y*-direction to `'normal'` so that *Y*-axis values increase from bottom to top.

Place a marker at the origin of the scene.

```
figure
fileName = 'sim3d_LargeParkingLot.jpg';
I = imshow(fileName,spatialRef);
set(gca,'YDir','normal')
xlabel('X (m)')
ylabel('Y (m)')

hold on
plot(0,0,'o','MarkerFaceColor','r','MarkerEdgeColor','k','MarkerSize',8)
offset = 3; % px
text(offset,offset,'(0,0)','Color','w','FontWeight','bold','FontSize',12)
hold off
```



Tips

- If you have the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, then you can modify this scene. In the Unreal Engine project file that comes with the support package, this scene is named `LargeParkingLot`.

  For more details on customizing scenes, see "Customize Unreal Engine Scenes for Automated Driving".

See Also

Curved Road | Double Lane Change | Open Surface | Parking Lot | Straight Road | US City Block | US Highway | Virtual Mcity

Topics

"Unreal Engine Simulation for Automated Driving"
"Unreal Engine Simulation Environment Requirements and Limitations"
"Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox"

Open Surface

Open surface 3D environment

Description

The **Open Surface** scene contains a 3D environment of an open, black road surface. The scene is rendered using the Unreal Engine from Epic Games.

To simulate a driving algorithm in this scene:

1. Add a Simulation 3D Scene Configuration block to your Simulink model.
2. In this block, set the **Scene source** parameter to `Default Scenes`.
3. Set the enabled **Scene name** parameter to `Open surface`.

Explore Open Surface Scene

Explore the 3D Open Surface scene and inspect its dimensions by using a corresponding 2D top-view image of the scene.

You can use this image to inspect the scene before simulation and choose starting coordinates for vehicles. For details on using these images to select waypoints for path-following applications, see the “Select Waypoints for Unreal Engine Simulation” example.

Load the 2D spatial referencing object that corresponds to the scene. This `imref2d` (Image Processing Toolbox) object describes the relationship between the pixels in the image and the world coordinates of the scene.

```
data = load('sim3d_SpatialReferences.mat');
spatialRef = data.spatialReference.OpenSurface
```

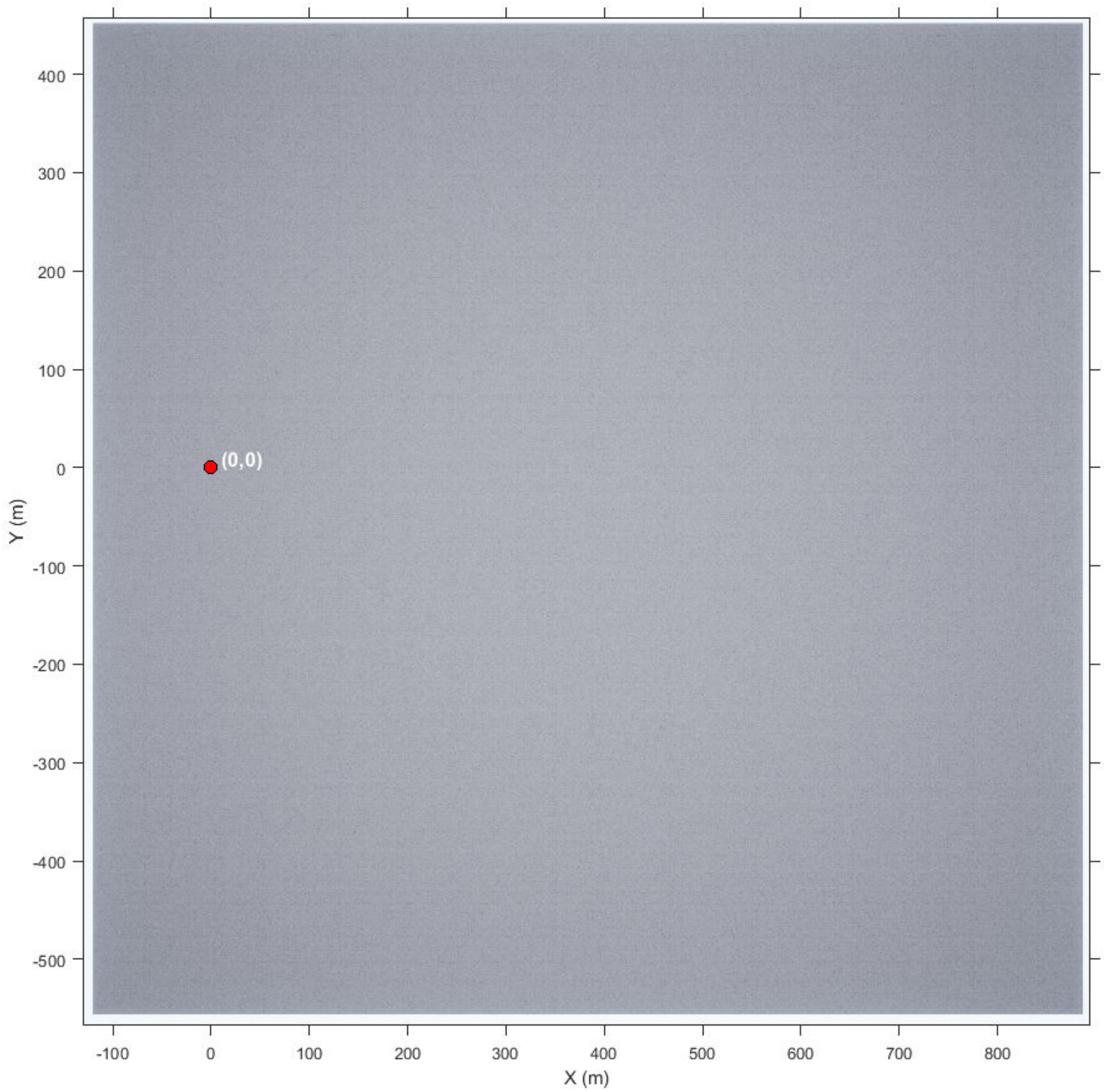
```
spatialRef = 
  imref2d with properties:

           XWorldLimits: [-130.5500 894.4500]
           YWorldLimits: [-567.2500 457.7500]
              ImageSize: [4845 4845]
    PixelExtentInWorldX: 0.2116
    PixelExtentInWorldY: 0.2116
    ImageExtentInWorldX: 1025
    ImageExtentInWorldY: 1025
       XIntrinsicLimits: [0.5000 4.8455e+03]
       YIntrinsicLimits: [0.5000 4.8455e+03]
```

Display the image corresponding to the scene. Use the spatial referencing object to display the axes in the world coordinates of the scene. Units are in meters.

By default, the `imshow` function displays *Y*-axis values that increase from top to bottom. To align with the Automated Driving Toolbox™ world coordinate system, set the *Y*-direction to `'normal'` so that *Y*-axis values increase from bottom to top.

Place a marker at the origin of the scene.

```
figure
fileName = 'sim3d_OpenSurface.jpg';
I = imshow(fileName,spatialRef);
set(gca,'YDir','normal')
xlabel('X (m)')
ylabel('Y (m)')

hold on
plot(0,0,'o','MarkerFaceColor','r','MarkerEdgeColor','k','MarkerSize',8)
offset = 10; % px
text(offset,offset,'(0,0)','Color','w','FontWeight','bold','FontSize',12)
hold off
```



Tips

- If you have the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, then you can modify this scene. In the Unreal Engine project file that comes with the support package, this scene is named `BlackLake`.

  For more details on customizing scenes, see “Customize Unreal Engine Scenes for Automated Driving”.

See Also
Curved Road | Double Lane Change | Large Parking Lot | Parking Lot | Straight Road | US City Block | US Highway | Virtual Mcity

Topics
"Unreal Engine Simulation for Automated Driving"
"Unreal Engine Simulation Environment Requirements and Limitations"
"Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox"

Parking Lot

Parking lot 3D environment

Description

The **Parking Lot** scene is a 3D environment of a parking lot. The scene is rendered using the Unreal Engine from Epic Games.

To simulate a driving algorithm in this scene:

1. Add a Simulation 3D Scene Configuration block to your Simulink model.
2. In this block, set the **Scene source** parameter to `Default Scenes`.
3. Set the enabled **Scene name** parameter to `Parking lot`.

Explore Parking Lot Scene

Explore the 3D Parking Lot scene and inspect its dimensions by using a corresponding 2D top-view image of the scene.

You can use this image to inspect the scene before simulation and choose starting coordinates for vehicles. For details on using these images to select waypoints for path-following applications, see the "Select Waypoints for Unreal Engine Simulation" example.

Load the 2D spatial referencing object that corresponds to the scene. This `imref2d` (Image Processing Toolbox) object describes the relationship between the pixels in the image and the world coordinates of the scene.

```
data = load('sim3d_SpatialReferences.mat');
spatialRef = data.spatialReference.ParkingLot
```

```
spatialRef = 
  imref2d with properties:
```

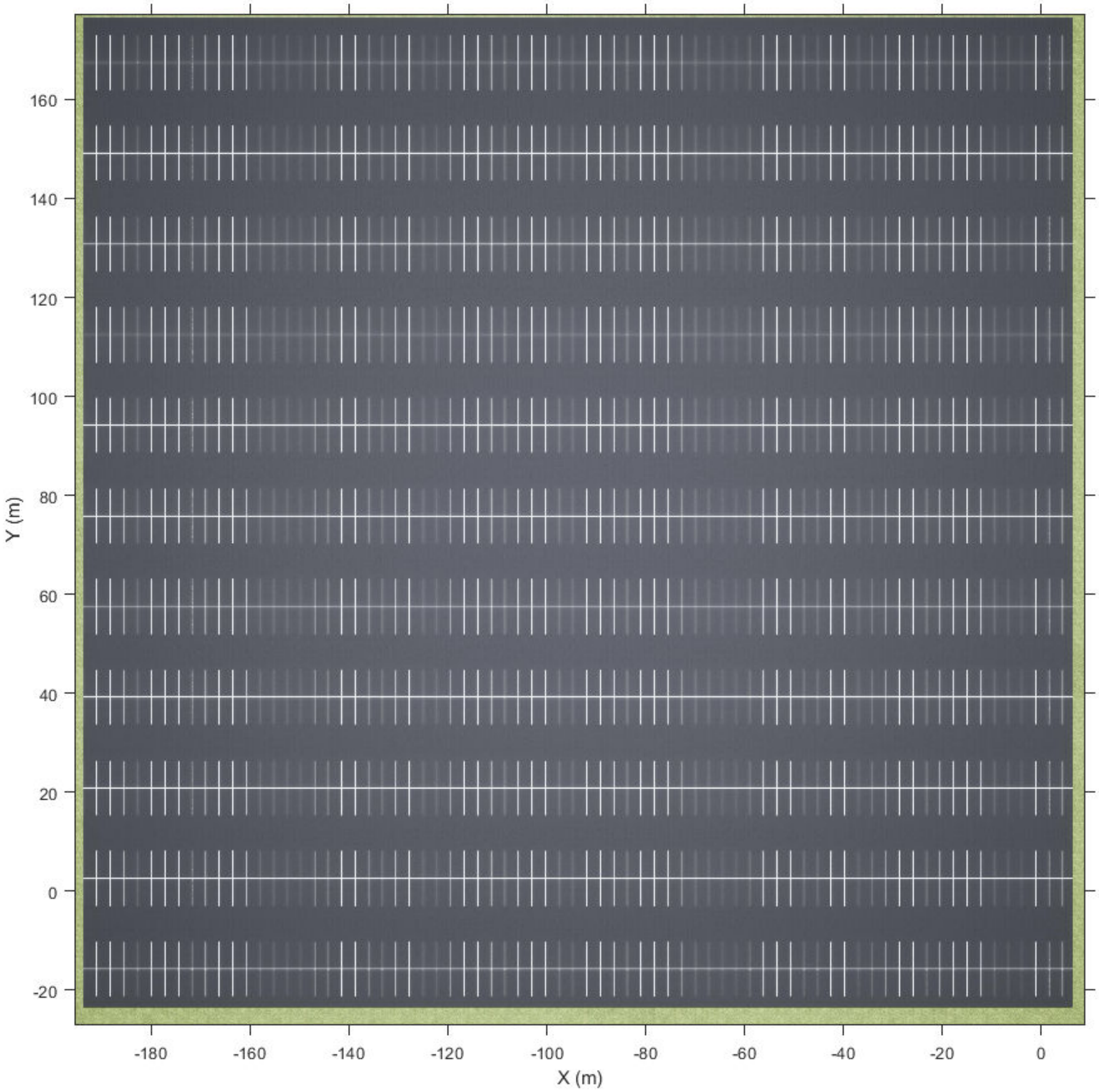
```
         XWorldLimits: [-195.5000 8.9000]
         YWorldLimits: [-27.1000 177.3000]
            ImageSize: [4845 4845]
    PixelExtentInWorldX: 0.0422
    PixelExtentInWorldY: 0.0422
    ImageExtentInWorldX: 204.4000
    ImageExtentInWorldY: 204.4000
       XIntrinsicLimits: [0.5000 4.8455e+03]
       YIntrinsicLimits: [0.5000 4.8455e+03]
```

Display the image corresponding to the scene. Use the spatial referencing object to display the axes in the world coordinates of the scene. Units are in meters.

By default, the `imshow` function displays *Y*-axis values that increase from top to bottom. To align with the Automated Driving Toolbox™ world coordinate system, set the *Y*-direction to `'normal'` so that *Y*-axis values increase from bottom to top.

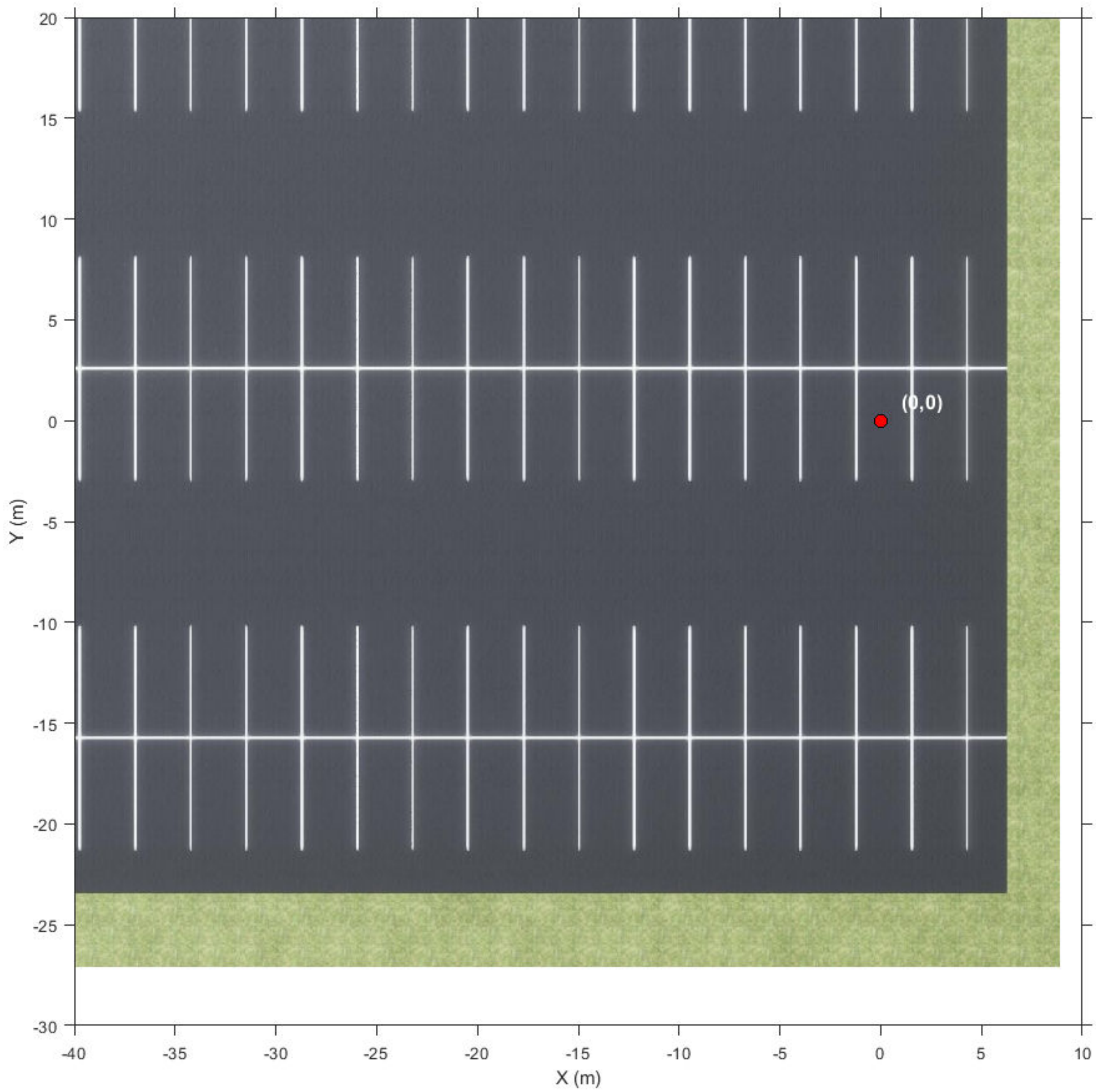
The image displays only the area of the scene containing the parking lot. The full scene has a length and width of 705.6 meters.

```
figure
fileName = 'sim3d_ParkingLot.jpg';
I = imshow(fileName,spatialRef);
set(gca,'YDir','normal')
xlabel('X (m)')
ylabel('Y (m)')
```



Zoom in on the origin of the scene. Place a marker at the origin.

```
xlim([-40 10])
ylim([-30 20])

hold on
plot(0,0,'o','MarkerFaceColor','r','MarkerEdgeColor','k','MarkerSize',8)
offset = 1; % px
text(offset,offset,'(0,0)','Color','w','FontWeight','bold','FontSize',12)
hold off
```



Tips

- If you have the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, then you can modify this scene. In the Unreal Engine project file that comes with the support package, this scene is named `SimpleLot`.

  For more details on customizing scenes, see "Customize Unreal Engine Scenes for Automated Driving".

See Also

Curved Road | Double Lane Change | Large Parking Lot | Open Surface | Straight Road | US City Block | US Highway | Virtual Mcity

Topics

“Unreal Engine Simulation for Automated Driving”
“Unreal Engine Simulation Environment Requirements and Limitations”
“Coordinate Systems in Automated Driving Toolbox”

Straight Road

Straight road 3D environment

Description

The **Straight Road** scene is a 3D environment of a straight four-lane divided highway. The scene is rendered using the Unreal Engine from Epic Games.

To simulate a driving algorithm in this scene:

1. Add a Simulation 3D Scene Configuration block to your Simulink model.
2. In this block, set the **Scene source** parameter to `Default Scenes`.
3. Set the enabled **Scene name** parameter to `Straight road`.

Explore Straight Road Scene

Explore the 3D Straight Road scene and inspect its dimensions by using a corresponding 2D top-view image of the scene.

You can use this image to inspect the scene before simulation and choose starting coordinates for vehicles. For details on using these images to select waypoints for path-following applications, see the "Select Waypoints for Unreal Engine Simulation" example.

Load the 2D spatial referencing object that corresponds to the scene. This `imref2d` (Image Processing Toolbox) object describes the relationship between the pixels in the image and the world coordinates of the scene.

```
data = load('sim3d_SpatialReferences.mat');
spatialRef = data.spatialReference.StraightRoad
```

```
spatialRef = 
  imref2d with properties:
```

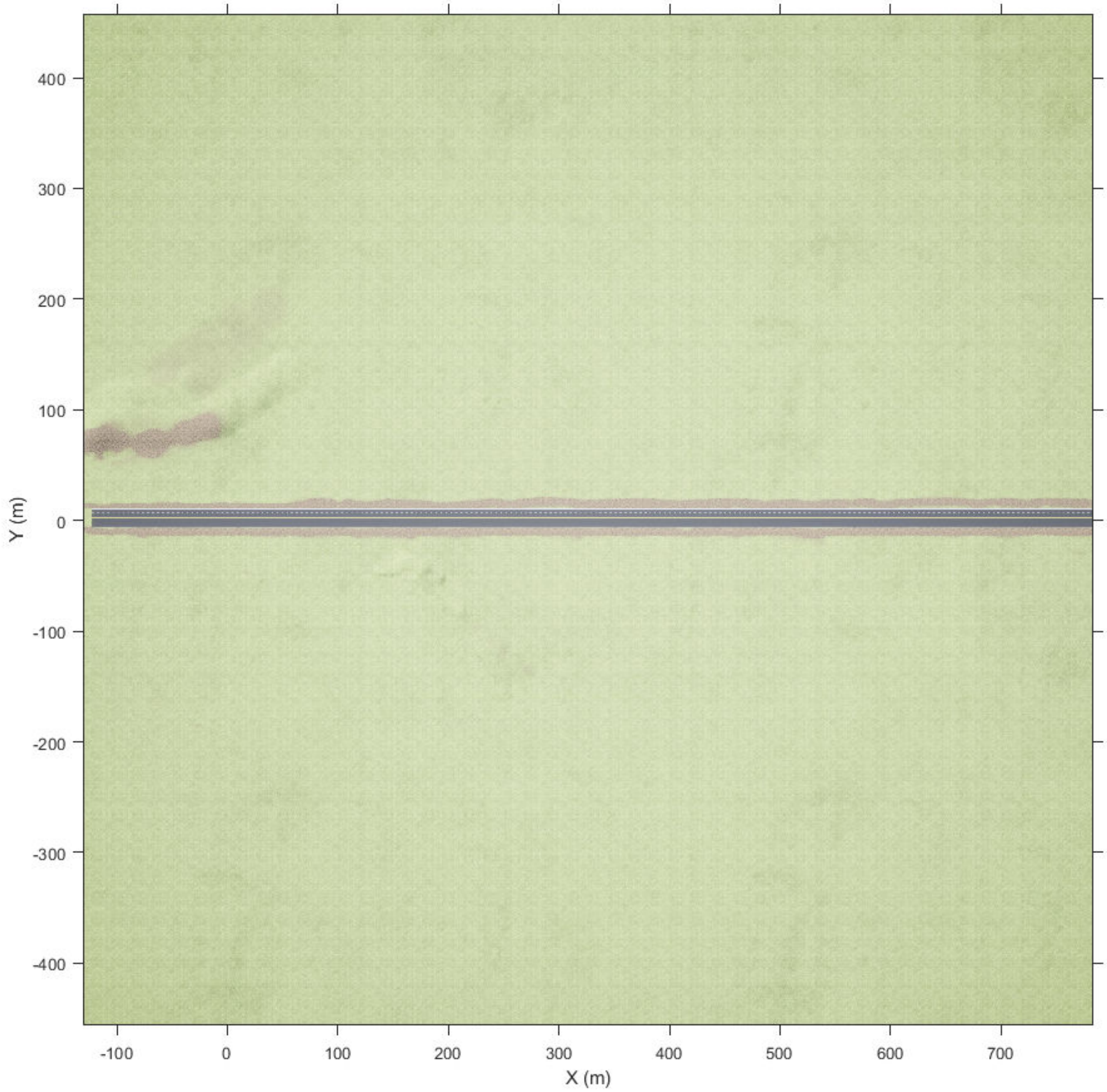
```
          XWorldLimits: [-130.5500 783.3500]
          YWorldLimits: [-456.1500 457.7500]
             ImageSize: [4845 4845]
   PixelExtentInWorldX: 0.1886
   PixelExtentInWorldY: 0.1886
   ImageExtentInWorldX: 913.9000
   ImageExtentInWorldY: 913.9000
      XIntrinsicLimits: [0.5000 4.8455e+03]
      YIntrinsicLimits: [0.5000 4.8455e+03]
```

Display the image corresponding to the scene. Use the spatial referencing object to display the axes in the world coordinates of the scene. Units are in meters.

By default, the `imshow` function displays *Y*-axis values that increase from top to bottom. To align with the Automated Driving Toolbox™ world coordinate system, set the *Y*-direction to `'normal'` so that *Y*-axis values increase from bottom to top.

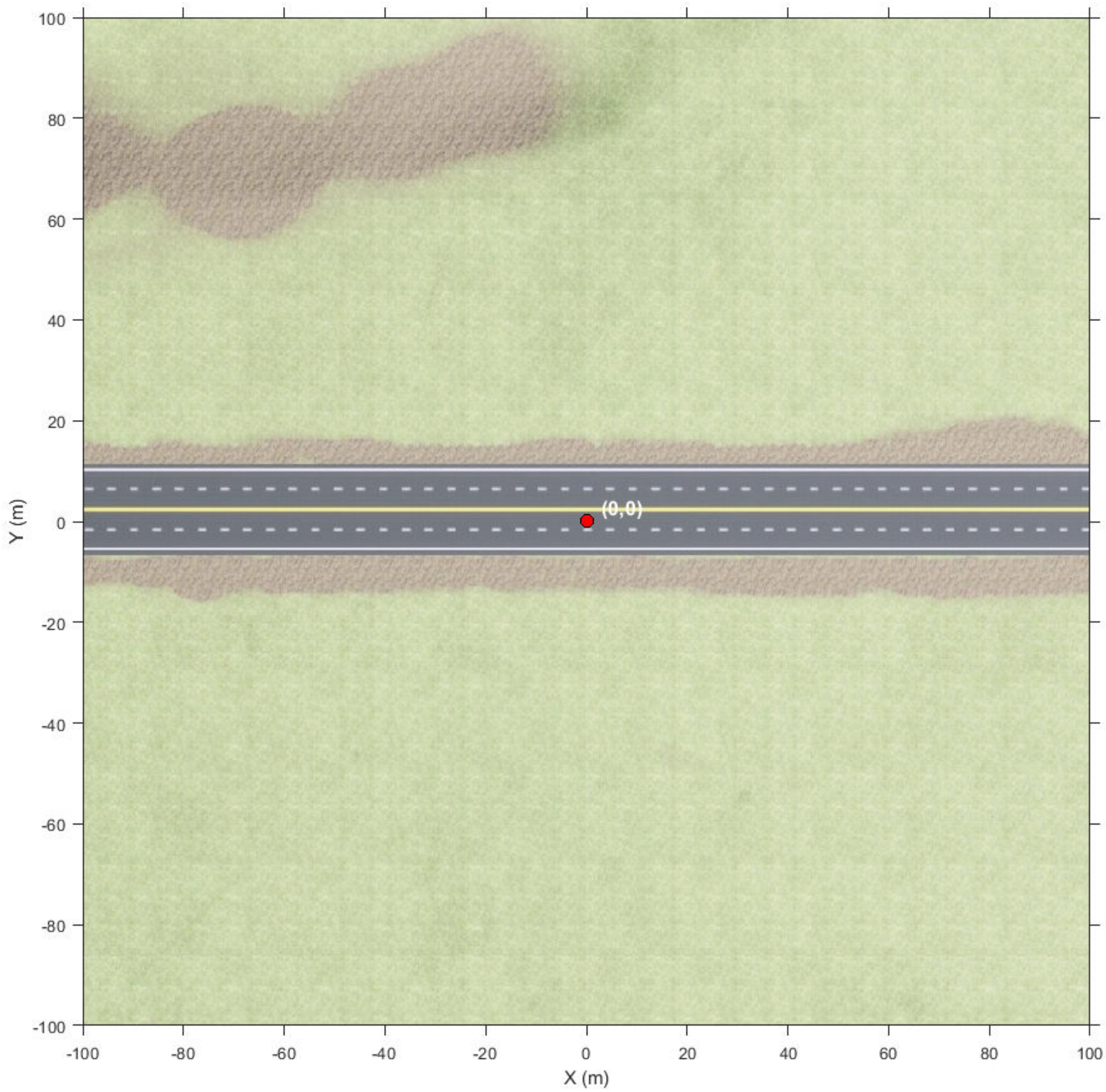
The image displays only the area of the scene containing the straight road. The full scene has a length and width of 2016 meters.

```
figure
fileName = 'sim3d_StraightRoad.jpg';
I = imshow(fileName,spatialRef);
set(gca,'YDir','normal')
xlabel('X (m)')
ylabel('Y (m)')
```



Zoom in on the origin of the scene. Place a marker at the origin.

```
xlim([-100 100])
ylim([-100 100])

hold on
plot(0,0,'o','MarkerFaceColor','r','MarkerEdgeColor','k','MarkerSize',8)
offset = 3; % px
text(offset,offset,'(0,0)','Color','w','FontWeight','bold','FontSize',12)
hold off
```



Tips

- If you have the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, then you can modify this scene. In the Unreal Engine project file that comes with the support package, this scene is named `HwStrght`.

  For more details on customizing scenes, see "Customize Unreal Engine Scenes for Automated Driving".

See Also
Curved Road | Double Lane Change | Large Parking Lot | Open Surface | Parking Lot | US City Block | US Highway | Virtual Mcity

Topics
"Unreal Engine Simulation for Automated Driving"
"Unreal Engine Simulation Environment Requirements and Limitations"
"Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox"
"Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer"

US City Block

US city block 3D environment

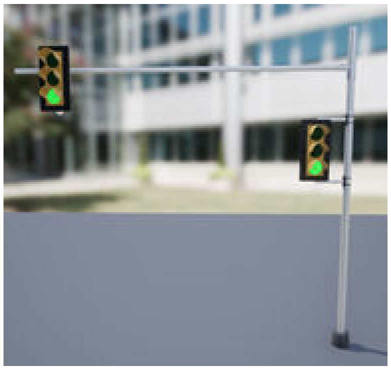
Description

The **US City Block** scene is a 3D environment of a US city block that contains 15 intersections and 30 traffic lights. The scene is rendered using the Unreal Engine from Epic Games.

To simulate a driving algorithm in this scene:

1. Add a Simulation 3D Scene Configuration block to your Simulink model.
2. In this block, set the **Scene source** parameter to `Default Scenes`.
3. Set the enabled **Scene name** parameter to `US city block`.

Traffic Lights



The US City Scene contains 30 traffic lights, two at each of the 15 intersections. Each intersection has a traffic light group. If you use the “Traffic Light Negotiation with Unreal Engine Visualization” example, you can control the timing of the traffic lights.

This table provides the traffic light names and locations in the world coordinate system. Dimensions are in m. Only one of the traffic lights in the group can be green at a time. The traffic lights are green for 10 s and yellow for 3 s. At the start of the simulation, the first traffic lights in the group are green (for example, `SM_TrafficLights1_3` and `SM_TrafficLights2_3`). The second lights in the group are red (for example, `SM_TrafficLights1_4` and `SM_TrafficLights2_4`).

Intersection	Unreal Engine Editor Name		Location					
	Traffic Light Group	Traffic Light	X	Y	Z	Roll	Pitch	Yaw
1	TrafficLightGroup	SM_TrafficLights1_3	-196.55	100.65	0	0	0	-90°
		SM_TrafficLights1_4	-210.20	113.40	0	0	0	0
2	TrafficLightGroup2	SM_TrafficLights2_3	-106.35	-98.35	0	0	0	90°
		SM_TrafficLights2_4	-120.40	113.50	0	0	0	0
3	TrafficLightGroup3	SM_TrafficLights3_1	-13.10	116.20	0.2	0	0	-90°

Intersection	Unreal Engine Editor Name		Location					
	Traffic Light Group	Traffic Light	X	Y	Z	Roll	Pitch	Yaw
		SM_TrafficLights3_4	-30.60	113.80	0	0	0	0
4	TrafficLightGroup4	SM_TrafficLights4_3	71.40	100.30	0	0	0	100°
		SM_TrafficLights4_4	64.80	113.0	0	0	0	0
5	TrafficLightGroup5	SM_TrafficLights5_1	171.50	115.70	0	0	0	-90°
		SM_TrafficLights5_4	157.40	113.50	0	0	0	0
6	TrafficLightGroup6	SM_TrafficLights6_2	-177.30	-5.70	0	0	0	-180°
		SM_TrafficLights6_3	-189.60	-7.40	0	0	0	90°
7	TrafficLightGroup7	SM_TrafficLights7_2	-105.20	-5.50	0	0	0	-180°
		SM_TrafficLights7_3	-117.80	-7.70	0.2	0	0	90°
8	TrafficLightGroup8	SM_TrafficLights8_1	-13.10	7.60	0.1	0	0	-90°

Intersection	Unreal Engine Editor Name		Location					
	Traffic Light Group	Traffic Light	X	Y	Z	Roll	Pitch	Yaw
		SM_TrafficLights8_2	-10.90	-5.60	0	0	0	-180°
9	TrafficLightGroup9	SM_TrafficLights9_2	85.90	-7.60	0.2	0	0	-180°
		SM_TrafficLights9_3	70.90	-9.20	0	0	0	90°
10	TrafficLightGroup10	SM_TrafficLights10_1	172.10	7.70	0	0	0	-90°
		SM_TrafficLights10_2	173.70	-7.50	0	0	0	-180°
11	TrafficLightGroup11	SM_TrafficLights11_3	-189.80	-118.45	0	0	0	90°
		SM_TrafficLights11_4	-191.05	-104.55	0	0	0	0
12	TrafficLightGroup12	SM_TrafficLights12_3	-117.60	-117.60	0	0	0	90°
		SM_TrafficLights12_4	-120.50	-105.40	0	0	0	0
13	TrafficLightGroup13	SM_TrafficLights13_1	-12.80	-102.50	0	0	0	-90°

Intersection	Unreal Engine Editor Name		Location					
	Traffic Light Group	Traffic Light	X	Y	Z	Roll	Pitch	Yaw
		SM_TrafficLights13_4	-30.50	-105.30	0	0	0	0
14	TrafficLightGroup14	SM_TrafficLights14_3	70.90	-118.70	0	0	0	90°
		SM_TrafficLights14_4	69.30	-105.30	0	0	0	0
15	TrafficLightGroup15	SM_TrafficLights15_1	171.40	-105.20	0	0	0	-90°
		SM_TrafficLights15_4	158.40	-107.20	0	0	0	0

Explore US City Block Scene

Explore the 3D US City Block scene and inspect its dimensions by using a corresponding 2D top-view image of the scene.

You can use this image to inspect the scene before simulation and choose starting coordinates for vehicles. For details on using these images to select waypoints for path-following applications, see the "Select Waypoints for Unreal Engine Simulation" example.

Load the 2D spatial referencing object that corresponds to the scene. This `imref2d` (Image Processing Toolbox) object describes the relationship between the pixels in the image and the world coordinates of the scene.

```
data = load('sim3d_SpatialReferences.mat');
spatialRef = data.spatialReference.USCityBlock
```

```
spatialRef = 
  imref2d with properties:

           XWorldLimits: [-243.0500 200.2500]
           YWorldLimits: [-215.6500 227.6500]
              ImageSize: [4275 4275]
    PixelExtentInWorldX: 0.1037
    PixelExtentInWorldY: 0.1037
    ImageExtentInWorldX: 443.3000
    ImageExtentInWorldY: 443.3000
```

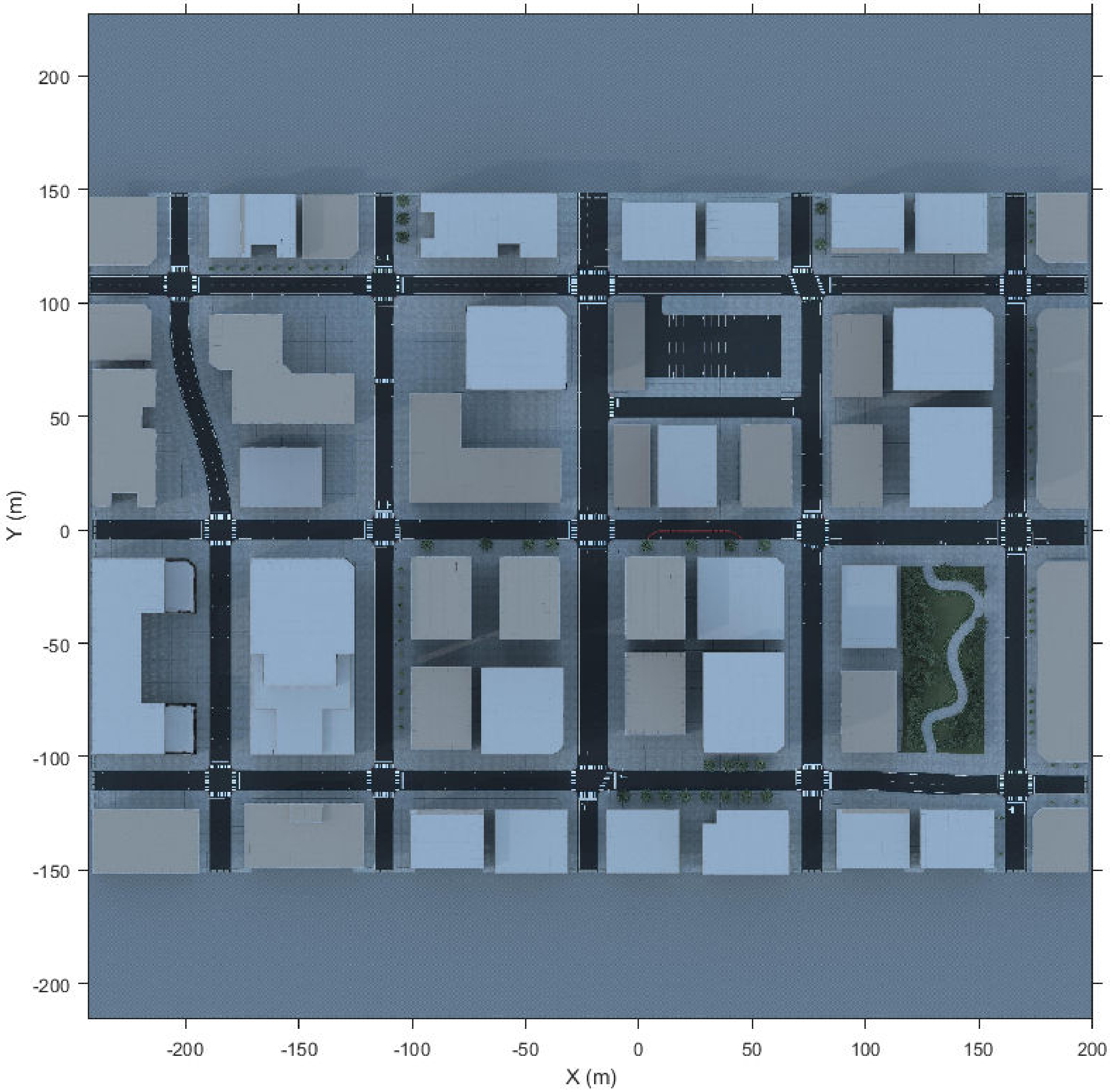
```
      XIntrinsicLimits: [0.5000 4.2755e+03]
      YIntrinsicLimits: [0.5000 4.2755e+03]
```

Display the image corresponding to the scene. Use the spatial referencing object to display the axes in the world coordinates of the scene. Units are in meters.

By default, the `imshow` function displays *Y*-axis values that increase from top to bottom. To align with the Automated Driving Toolbox™ world coordinate system, set the *Y*-direction to `'normal'` so that *Y*-axis values increase from bottom to top.

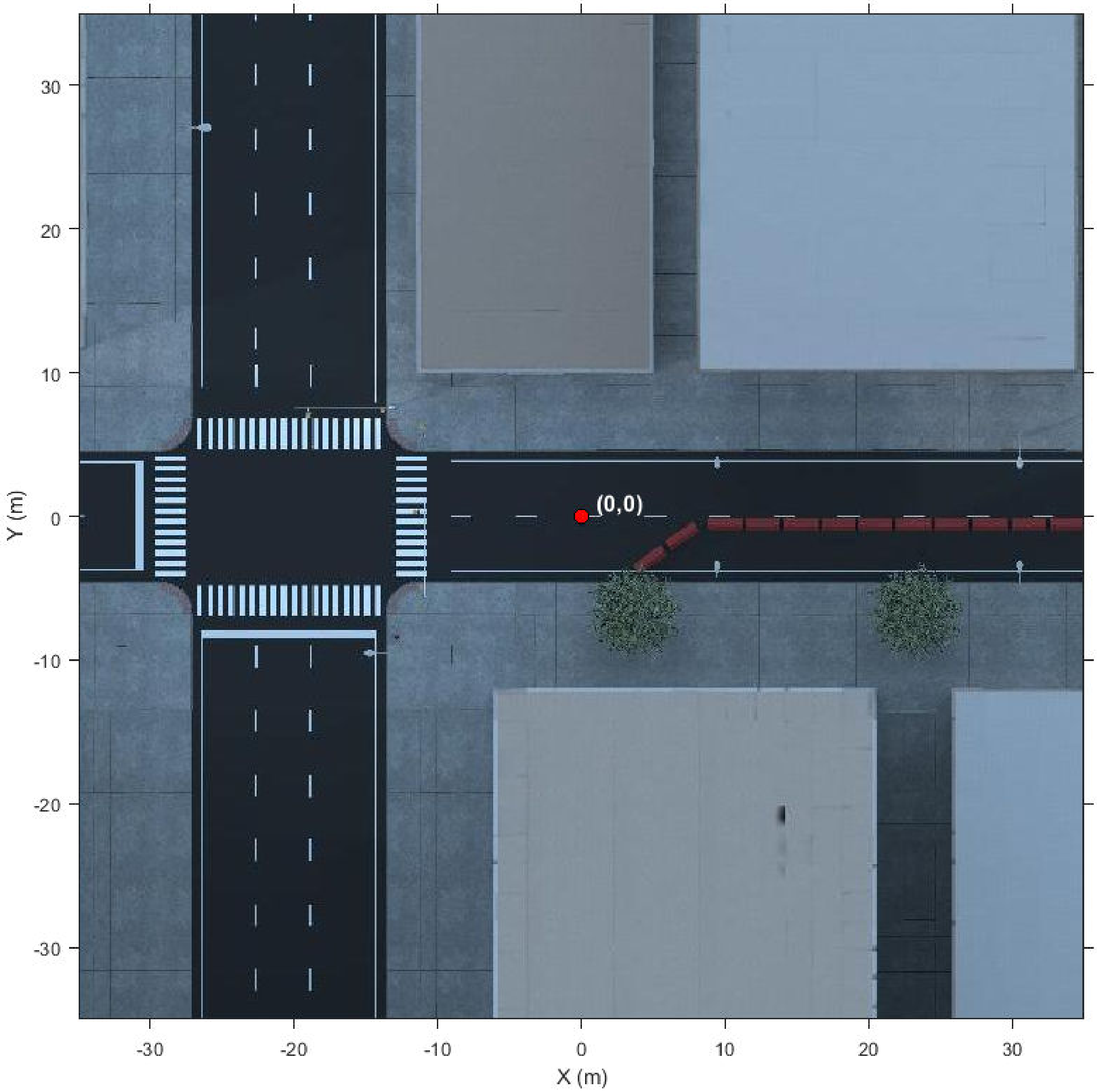
The image displays only the area of the scene containing the city block. The full scene has a length and width of 2040 meters.

```
figure
fileName = 'sim3d_USCityBlock.jpg';
I = imshow(fileName,spatialRef);
set(gca,'YDir','normal')
xlabel('X (m)')
ylabel('Y (m)')
```

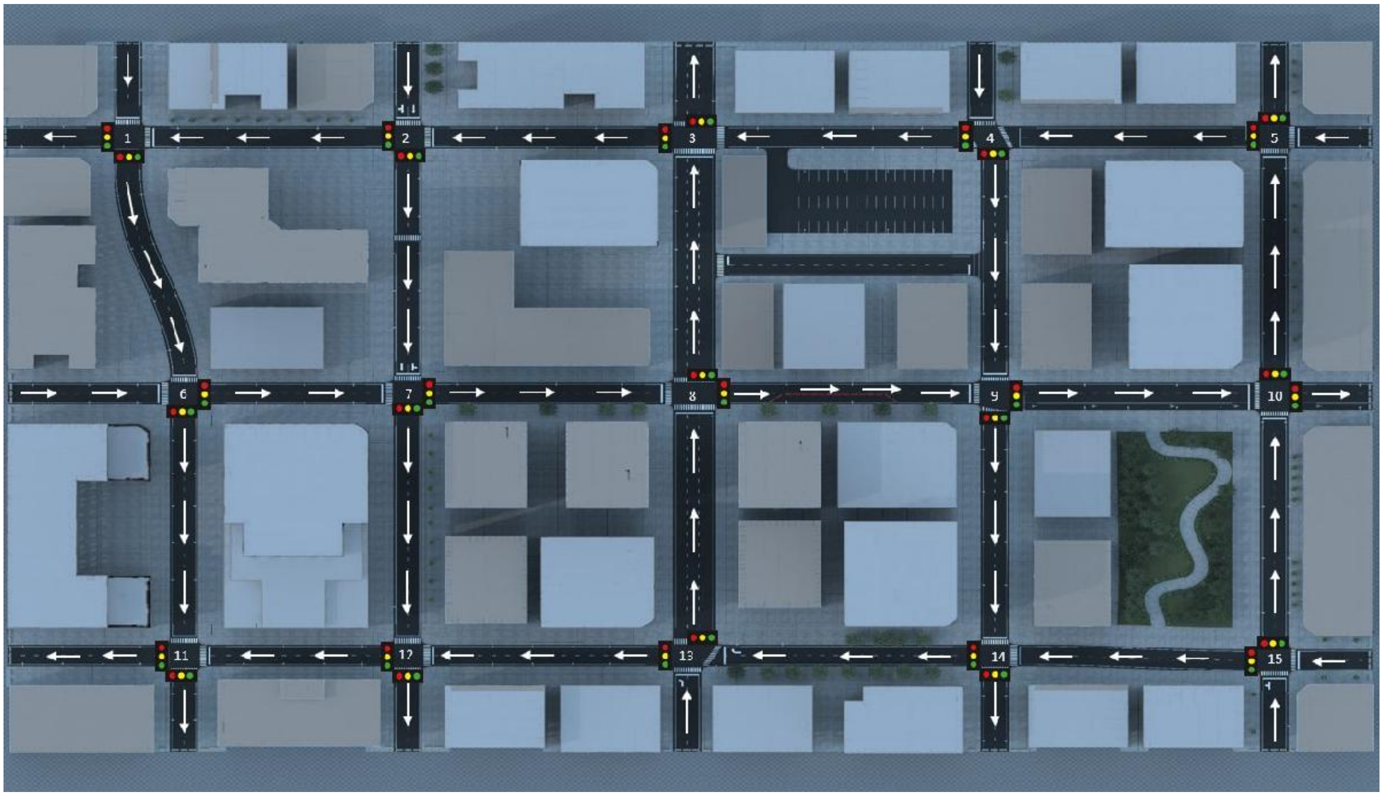


Zoom in on the origin of the scene. Place a marker at the origin.

```
xlim([-35 35])
ylim([-35 35])

hold on
plot(0,0,'o','MarkerFaceColor','r','MarkerEdgeColor','k','MarkerSize',8)
offset = 1; % px
text(offset,offset,'(0,0)','Color','w','FontWeight','bold','FontSize',12)
hold off
```



Each intersection in the scene contains two traffic light groups. These traffic lights change color based on common US traffic light patterns. All roads in the scene are one-way and follow the direction of traffic shown here.



Tips

- If you have the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, then you can modify this scene. In the Unreal Engine project file that comes with the support package, this scene is named `USCityBlock`.

  For more details on customizing scenes, see “Customize Unreal Engine Scenes for Automated Driving”.

See Also

Curved Road | Double Lane Change | Large Parking Lot | Open Surface | Parking Lot | Straight Road | US Highway | Virtual Mcity

Topics

“Unreal Engine Simulation for Automated Driving”
“Unreal Engine Simulation Environment Requirements and Limitations”
“Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox”
“Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer”

US Highway

US highway 3D environment

Description

The **US Highway** scene is a 3D environment of a US highway that contains barriers, cones, and traffic signs. The scene is rendered using the Unreal Engine from Epic Games.

To simulate a driving algorithm in this scene:

1. Add a Simulation 3D Scene Configuration block to your Simulink model.
2. In this block, set the **Scene source** parameter to `Default Scenes`.
3. Set the enabled **Scene name** parameter to `US highway`.

Explore US Highway Scene

Explore the 3D US Highway scene and inspect its dimensions by using a corresponding 2D top-view image of the scene.

You can use this image to inspect the scene before simulation and choose starting coordinates for vehicles. For details on using these images to select waypoints for path-following applications, see the "Select Waypoints for Unreal Engine Simulation" example.

Load the 2D spatial referencing object that corresponds to the scene. This `imref2d` (Image Processing Toolbox) object describes the relationship between the pixels in the image and the world coordinates of the scene.

```
data = load('sim3d_SpatialReferences.mat');
spatialRef = data.spatialReference.USHighway
```

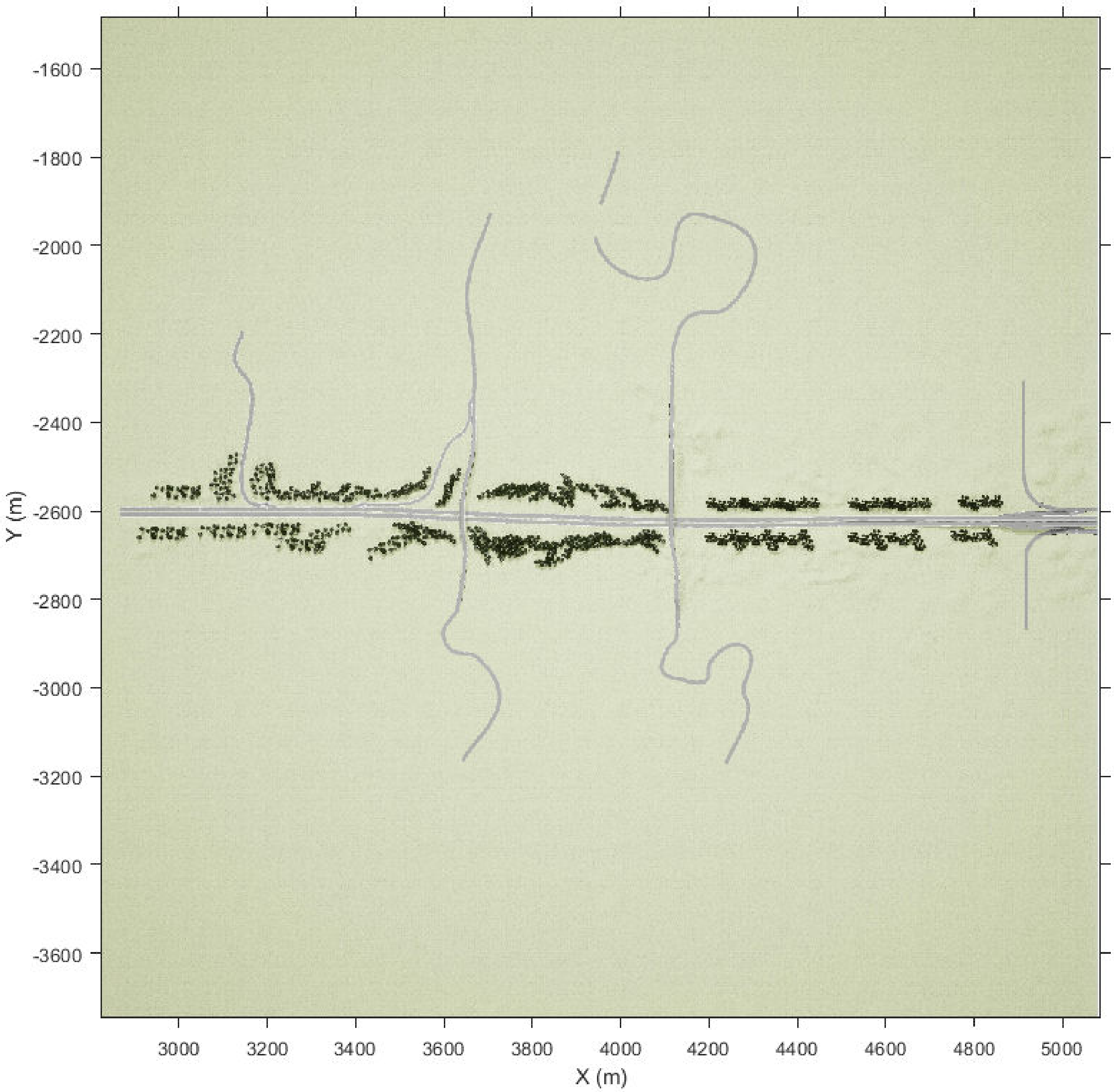
```
spatialRef = 
  imref2d with properties:

           XWorldLimits: [2.8218e+03 5.0868e+03]
           YWorldLimits: [-3.7469e+03 -1.4820e+03]
              ImageSize: [5585 5585]
    PixelExtentInWorldX: 0.4055
    PixelExtentInWorldY: 0.4055
    ImageExtentInWorldX: 2.2649e+03
    ImageExtentInWorldY: 2.2649e+03
       XIntrinsicLimits: [0.5000 5.5855e+03]
       YIntrinsicLimits: [0.5000 5.5855e+03]
```

Display the image corresponding to the scene. Use the spatial referencing object to display the axes in the world coordinates of the scene. Units are in meters.

By default, the `imshow` function displays *Y*-axis values that increase from top to bottom. To align with the Automated Driving Toolbox™ world coordinate system, set the *Y*-direction to `'normal'` so that *Y*-axis values increase from bottom to top.

The image displays only the area of the scene containing the highway. The full scene has a length and width of 10,160 meters. The origin of the scene is outside the range of the displayed image.

```
figure
fileName = 'sim3d_USHighway.jpg';
I = imshow(fileName,spatialRef);
set(gca,'YDir','normal')
xlabel('X (m)')
ylabel('Y (m)')
```



Tips

- If you have the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, then you can modify this scene. In the Unreal Engine project file that comes with the support package, this scene is named `USHighway`.

  For more details on customizing scenes, see “Customize Unreal Engine Scenes for Automated Driving”.

See Also

Curved Road | Double Lane Change | Large Parking Lot | Open Surface | Parking Lot | Straight Road | US City Block | Virtual Mcity

Topics

"Unreal Engine Simulation for Automated Driving"
"Unreal Engine Simulation Environment Requirements and Limitations"
"Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox"
"Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer"

Virtual Mcity

Virtual Mcity 3D environment

Description

The **Virtual Mcity** scene is a 3D environment containing a virtual representation of Mcity®, which is a testing ground belonging to the University of Michigan. For more details, see Mcity Test Facility.

The scene is rendered using the Unreal Engine from Epic Games.

To simulate a driving algorithm in this scene:

1. Add a Simulation 3D Scene Configuration block to your Simulink model.
2. In this block, set the **Scene source** parameter to `Default Scenes`.
3. Set the enabled **Scene name** parameter to `Virtual Mcity`.

Explore Virtual Mcity Scene

Explore the 3D Virtual Mcity scene and inspect its dimensions by using a corresponding 2D top-view image of the scene.

You can use this image to inspect the scene before simulation and choose starting coordinates for vehicles. For details on using these images to select waypoints for path-following applications, see the "Select Waypoints for Unreal Engine Simulation" example.

Load the 2D spatial referencing object that corresponds to the scene. This `imref2d` (Image Processing Toolbox) object describes the relationship between the pixels in the image and the world coordinates of the scene.

```
data = load('sim3d_SpatialReferences.mat');
spatialRef = data.spatialReference.VirtualMCity
```

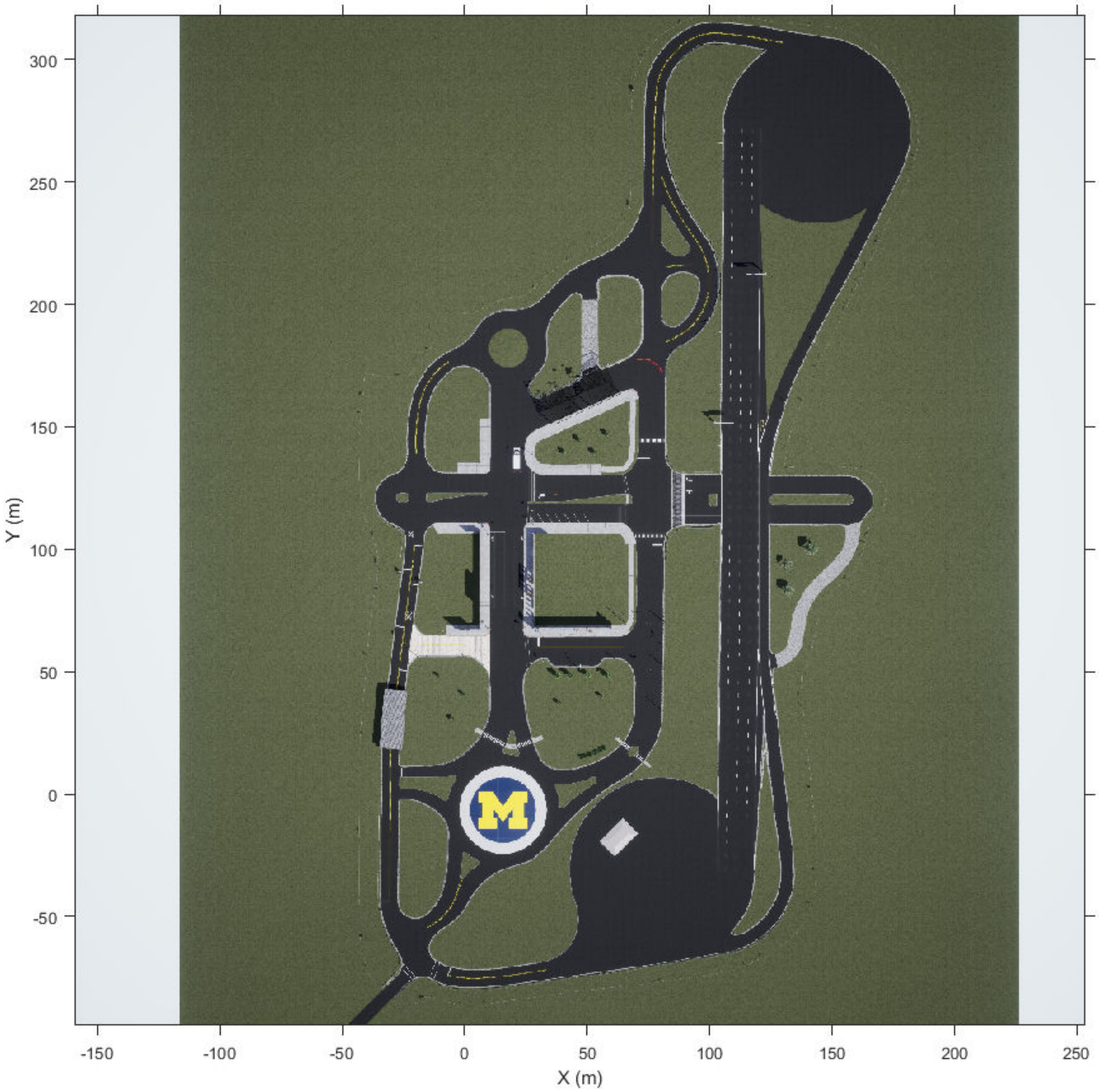
```
spatialRef = 
  imref2d with properties:

           XWorldLimits: [-159.3500 253.3500]
           YWorldLimits: [-94.4500 318.2500]
              ImageSize: [4845 4845]
    PixelExtentInWorldX: 0.0852
    PixelExtentInWorldY: 0.0852
    ImageExtentInWorldX: 412.7000
    ImageExtentInWorldY: 412.7000
       XIntrinsicLimits: [0.5000 4.8455e+03]
       YIntrinsicLimits: [0.5000 4.8455e+03]
```

Display the image corresponding to the scene. Use the spatial referencing object to display the axes in the world coordinates of the scene. Units are in meters.

By default, the `imshow` function displays *Y*-axis values that increase from top to bottom. To align with the Automated Driving Toolbox™ world coordinate system, set the *Y*-direction to `'normal'` so that *Y*-axis values increase from bottom to top.

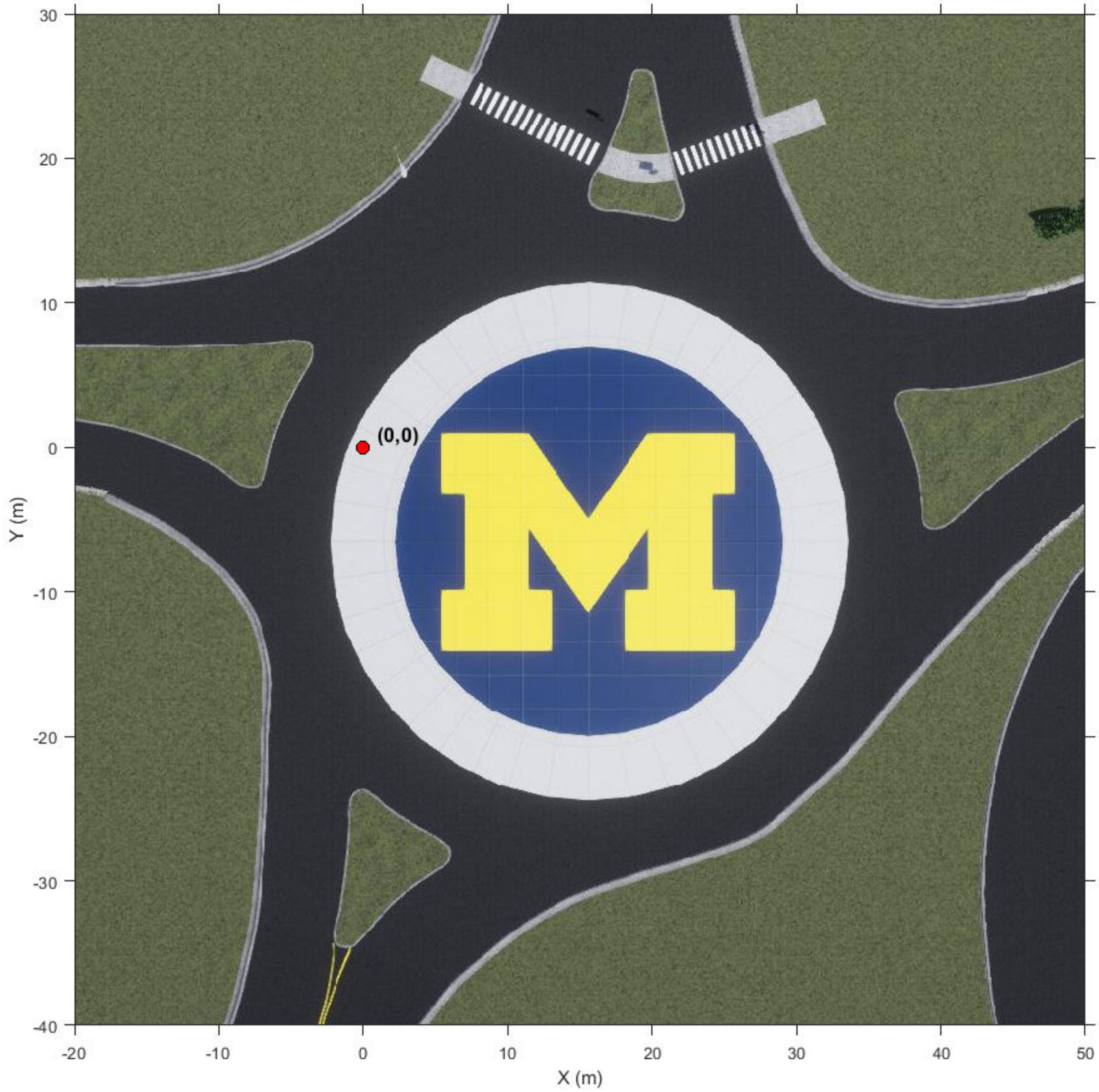
The image displays only the area of the scene containing the city. The full scene has a length of 541.44 meters and a width of 342.98 meters.

```
figure
fileName = 'sim3d_VirtualMCity.jpg';
I = imshow(fileName,spatialRef);
set(gca,'YDir','normal')
xlabel('X (m)')
ylabel('Y (m)')
```



Zoom in on the origin of the scene. Place a marker at the origin.

```
xlim([-20 50])
ylim([-40 30])

hold on
plot(0,0,'o','MarkerFaceColor','r','MarkerEdgeColor','k','MarkerSize',8)
offset = 1; % px
text(offset,offset,'(0,0)','Color','k','FontWeight','bold','FontSize',12)
hold off
```



Limitations

- In the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, this scene is not available for customization.

  For details on which scenes you can customize, see “Customize Unreal Engine Scenes for Automated Driving”.

See Also

Curved Road | Double Lane Change | Large Parking Lot | Open Surface | Parking Lot | Straight Road | US City Block | US Highway

Topics

"Unreal Engine Simulation for Automated Driving"
"Unreal Engine Simulation Environment Requirements and Limitations"
"Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox"

External Websites

Mcity Test Facility

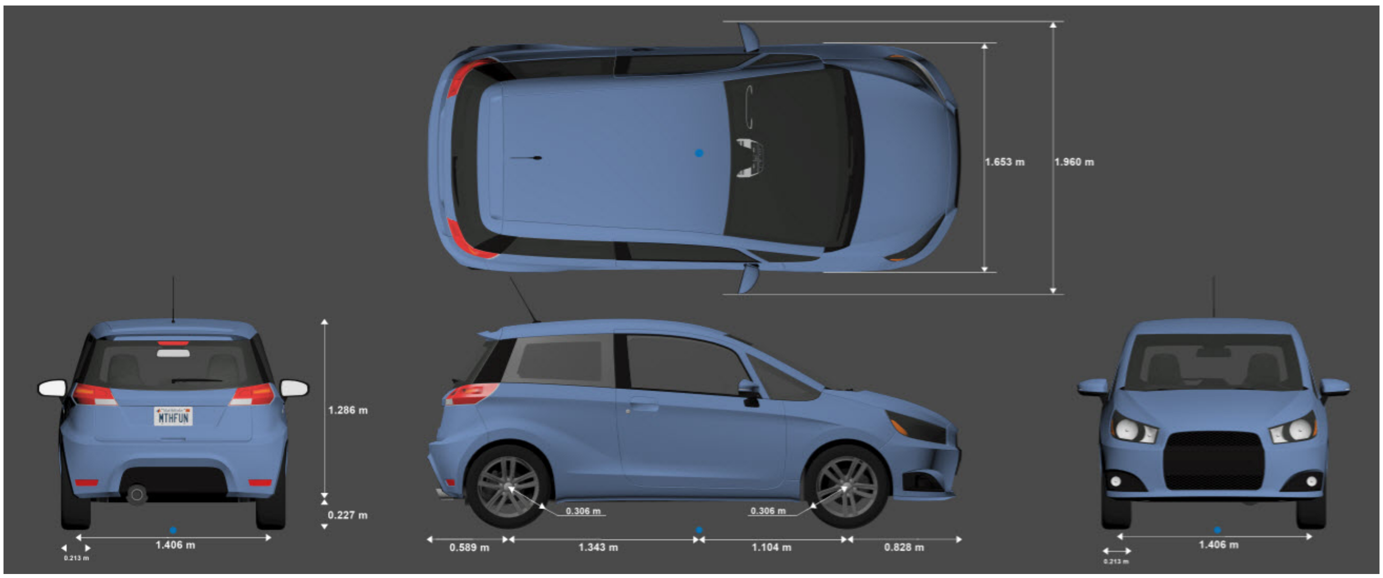
Vehicle Dimensions

Hatchback

Hatchback vehicle dimensions

Description

Hatchback is one of the vehicles that you can use within the 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The diagram provides the dimensions of this vehicle. The height dimensions are with respect to the vertical ground plane. The length and width dimensions are with respect to the origin of the vehicle in the vehicle coordinate system. The origin is on the ground, at the geometric center of the vehicle. For more detailed views of these diagrams, see the **Dimensions** section.

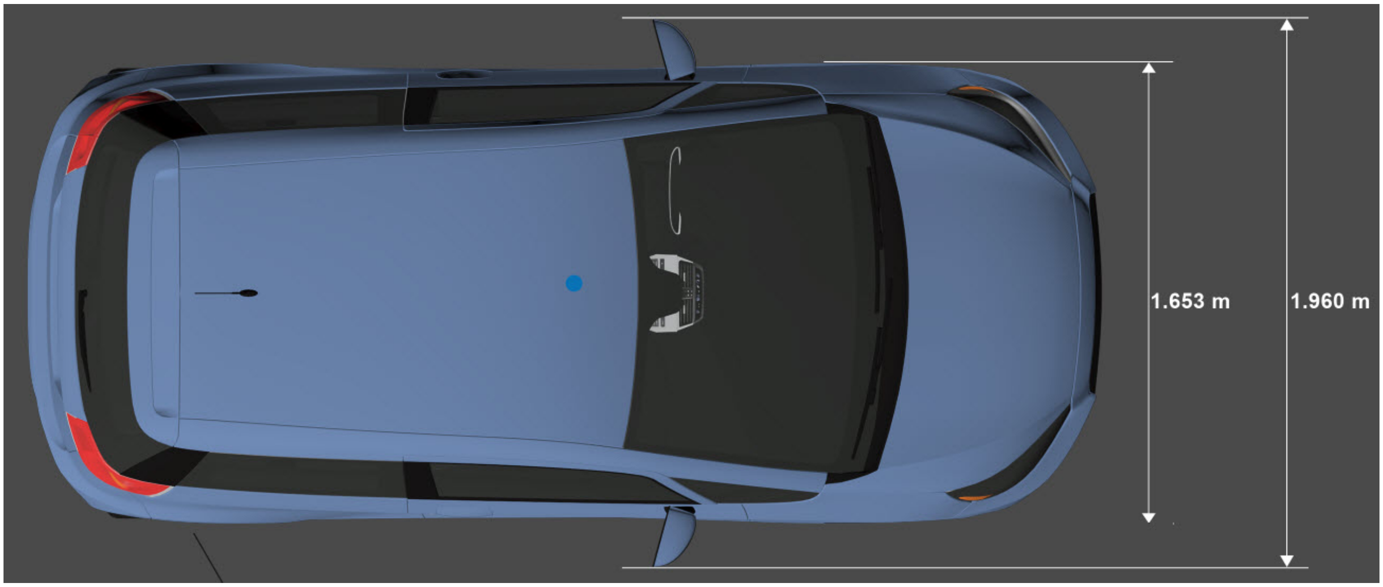


To add this type of vehicle to the 3D simulation environment:

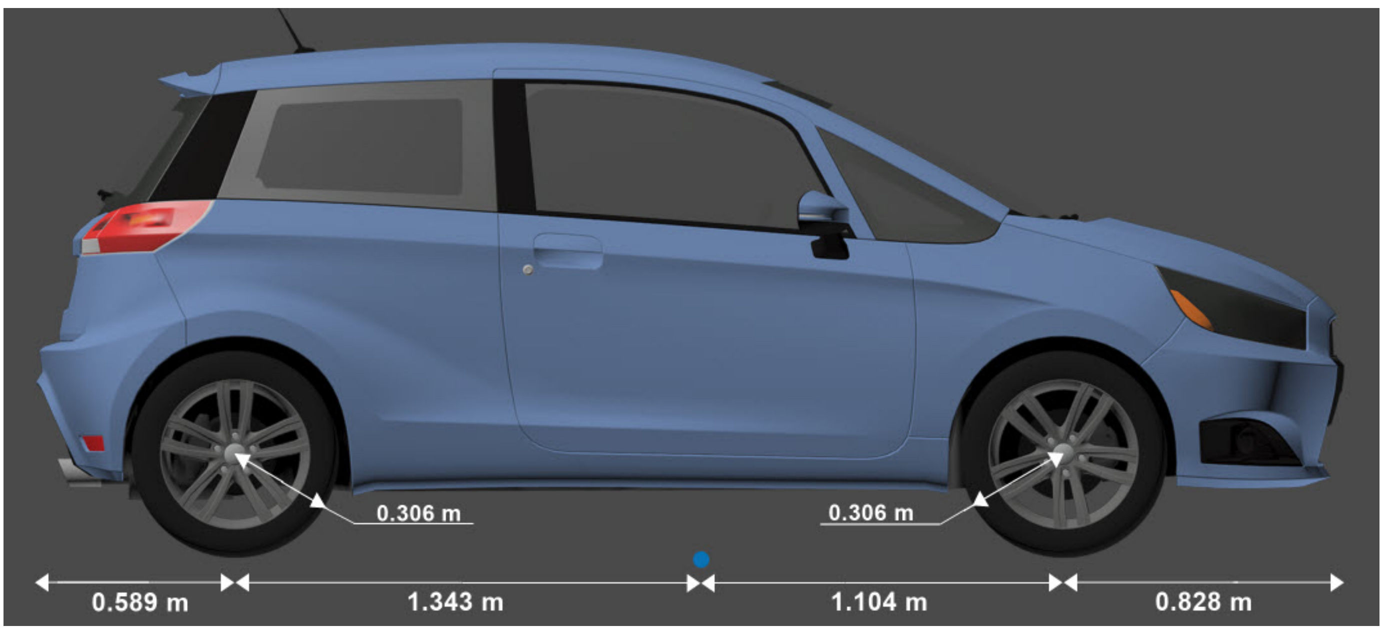
1. Add a Simulation 3D Vehicle with Ground Following block to your Simulink model.
2. In the block, set the **Type** parameter to `Hatchback`.

Dimensions

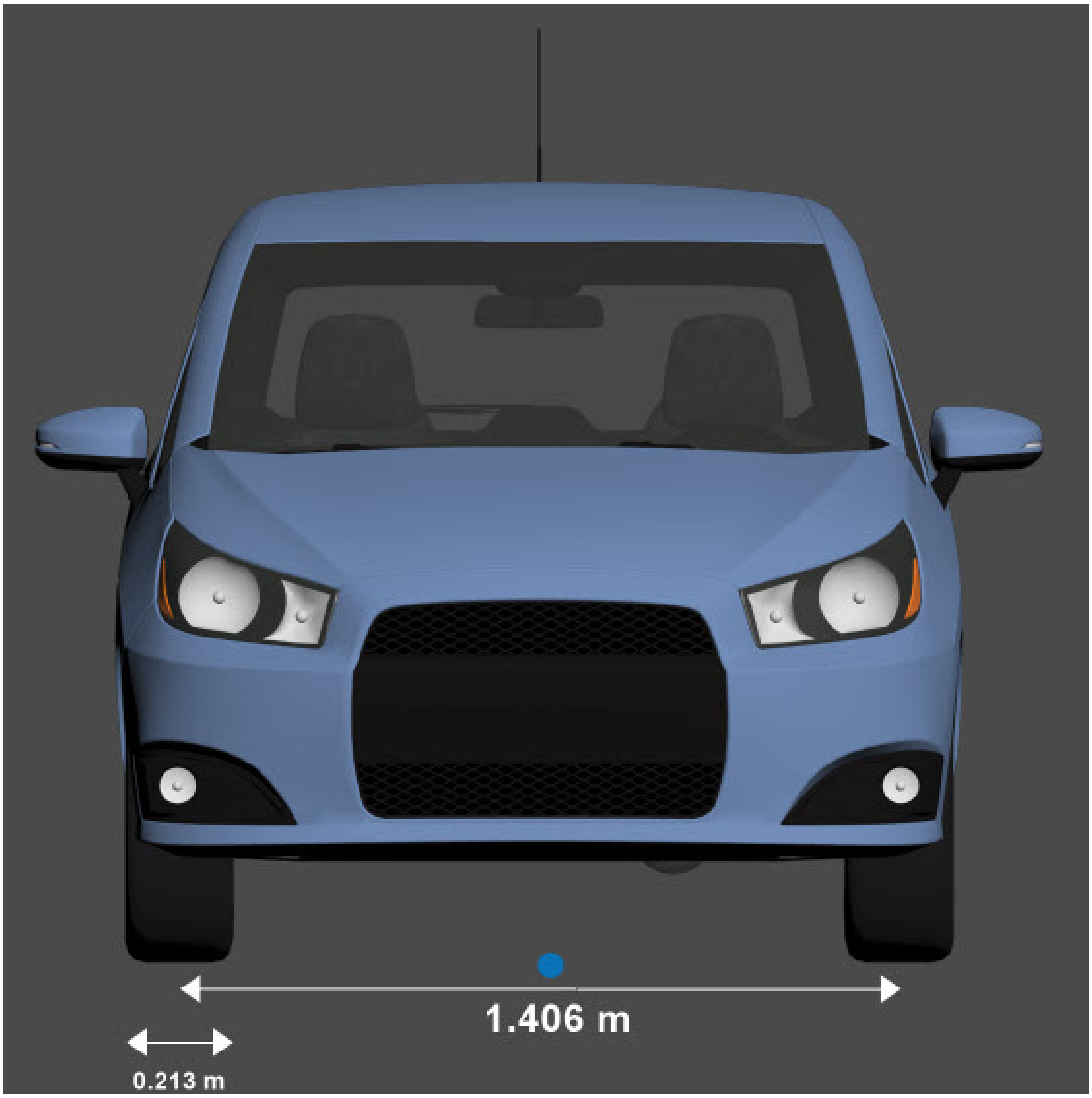
`Top-down view` — Vehicle width dimensions
diagram



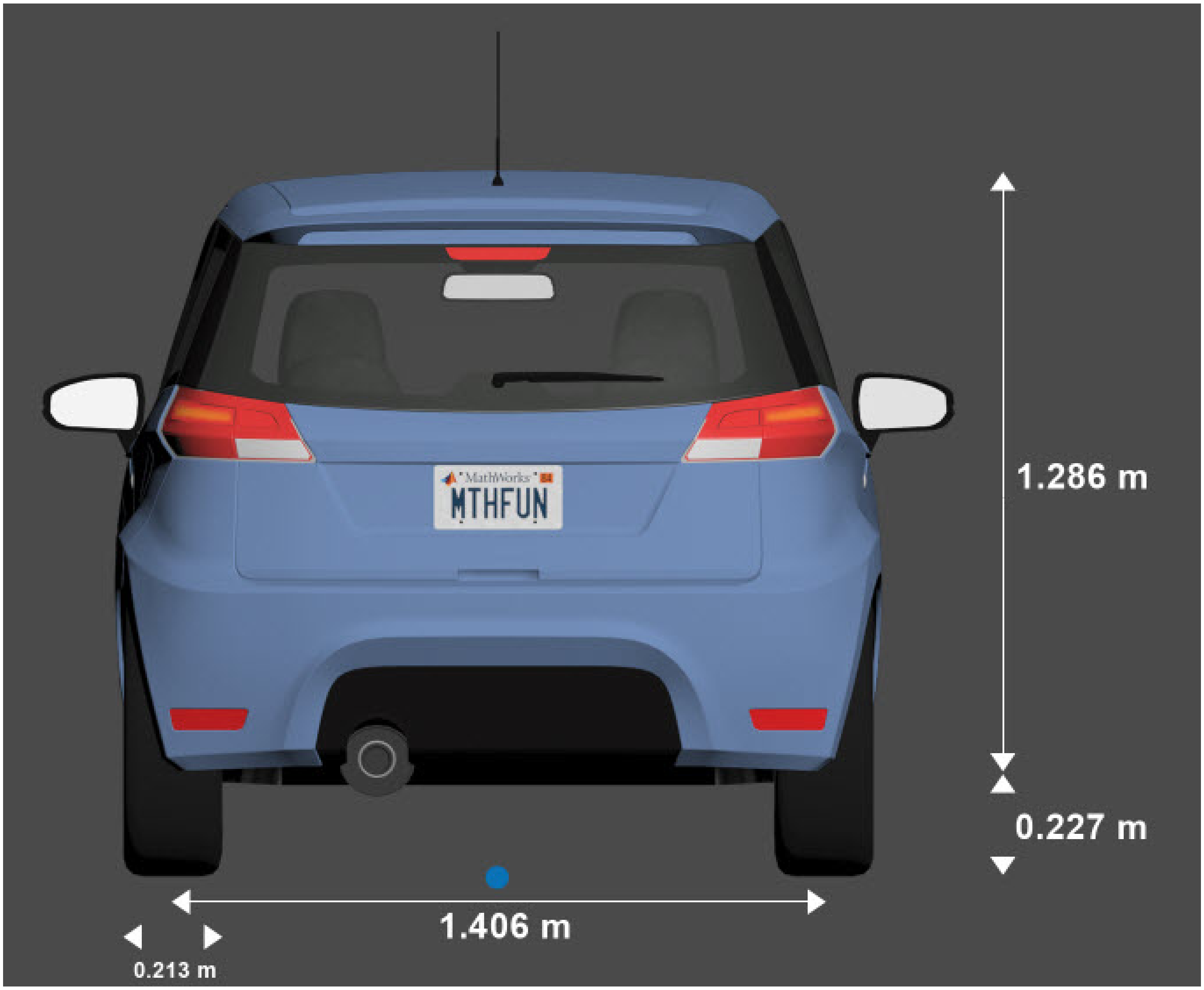
`Side view` — Vehicle length, front overhang, and rear overhang dimensions
diagram



`Front view` — Tire width and front axle dimensions
diagram



`Rear view` — Vehicle height and rear axle dimensions
diagram



Sensor Mounting Locations

In the 3D simulation sensor blocks, use the **Mounting location** parameter to mount sensors at predefined locations on the vehicle. The table shows the *X*, *Y*, and *Z* positions of the mounting locations relative to the vehicle origin. These locations are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when facing forward.
- The *Z*-axis points up from the ground.

Hatchback — Sensor Locations Relative to Vehicle Origin

Mounting Location	X (m)	Y (m)	Z (m)
`Front bumper`	1.93	0	0.51
`Rear bumper`	–1.93	0	0.51
`Right mirror`	0.43	–0.84	1.01
`Left mirror`	0.43	0.84	1.01
`Rearview mirror`	0.32	0	1.27
`Hood center`	1.44	0	1.01
`Roof center`	0	0	1.57

Specify Hatchback Vehicle Dimensions

When simulating a path planner in the 3D environment, the path planner must use a vehicle whose dimensions are consistent with one used in the 3D environment. To make these dimensions consistent, you can use a `vehicleDimensions` object.

Specify the dimensions of a Hatchback vehicle in a `vehicleDimensions` object. Units are in meters. For an example that uses this object in a path planner, see “Visualize Automated Parking Valet Using Unreal Engine Simulation”.

```
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)
```

```
hatchbackDims = 
  vehicleDimensions with properties:

           Length: 3.8640
            Width: 1.6530
           Height: 1.5130
        Wheelbase: 2.4470
     RearOverhang: 0.5890
    FrontOverhang: 0.8280
       WorldUnits: 'meters'
```

See Also

Box Truck | Muscle Car | Sedan | Small Pickup Truck | Sport Utility Vehicle

Topics

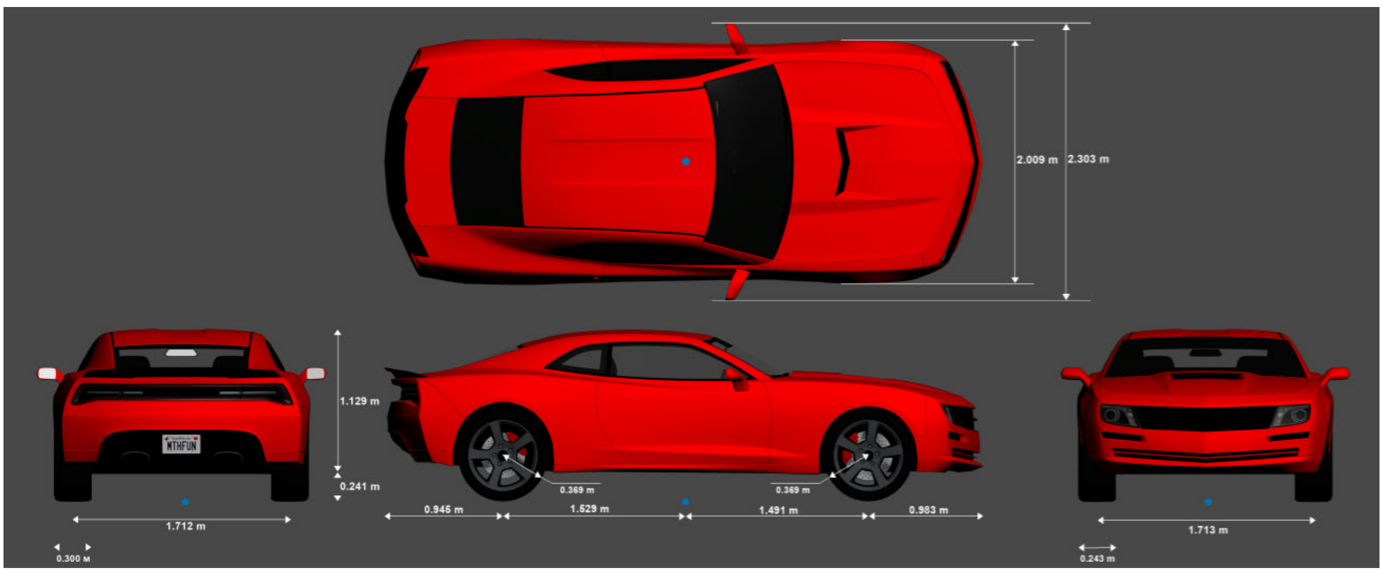
“Unreal Engine Simulation for Automated Driving”
“Coordinate Systems in Automated Driving Toolbox”

Muscle Car

Muscle car vehicle dimensions

Description

Muscle Car is one of the vehicles that you can use within the 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The following diagram provides the dimensions of this vehicle. The height dimensions are with respect to the vertical ground plane. The length and width dimensions are with respect to the origin of the vehicle in the vehicle coordinate system. The origin is on the ground, at the geometric center of the vehicle. For more detailed views of these diagrams, see the **Dimensions** section.



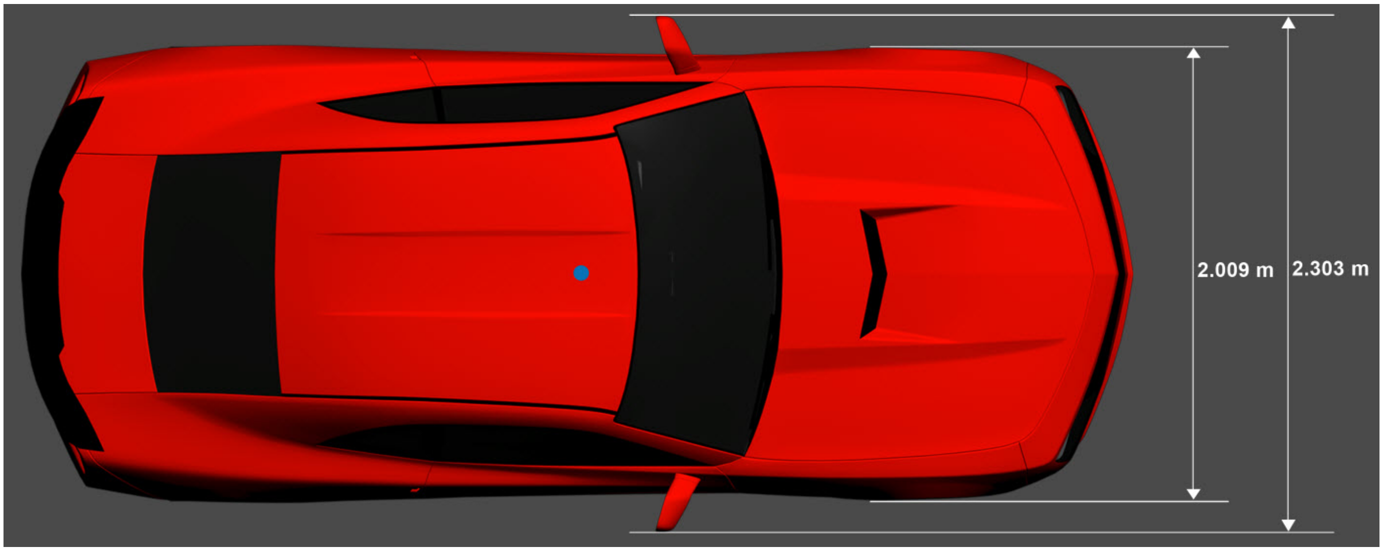
To add this type of vehicle to the 3D simulation environment:

1. Add a Simulation 3D Vehicle with Ground Following block to your Simulink model.
2. In the block, set the **Type** parameter to `Muscle car`.

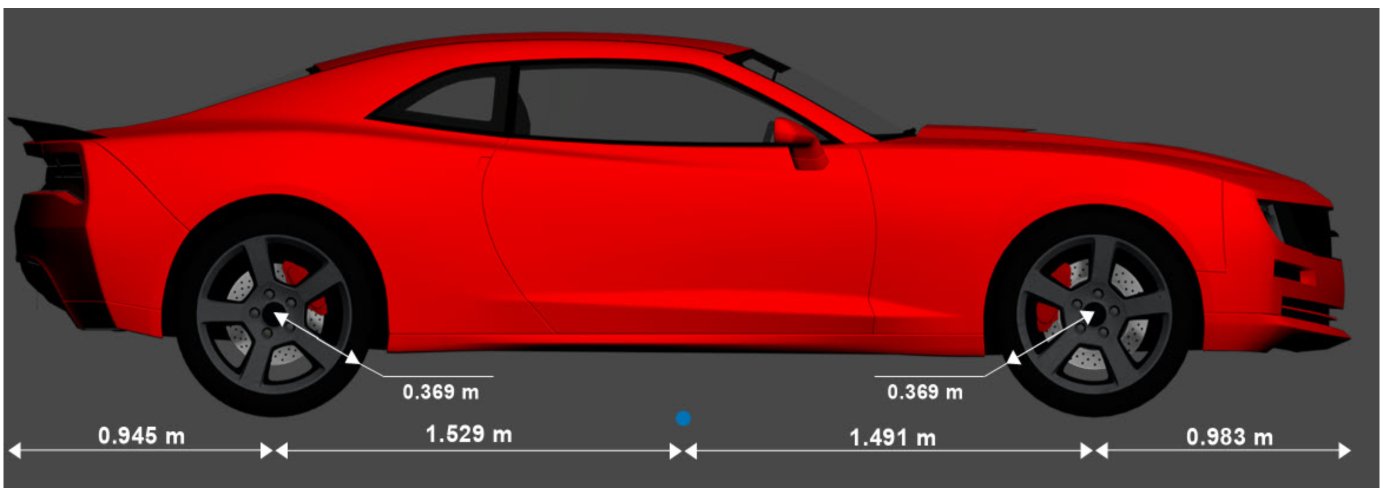
Dimensions

`Top-down view` — Vehicle width dimensions

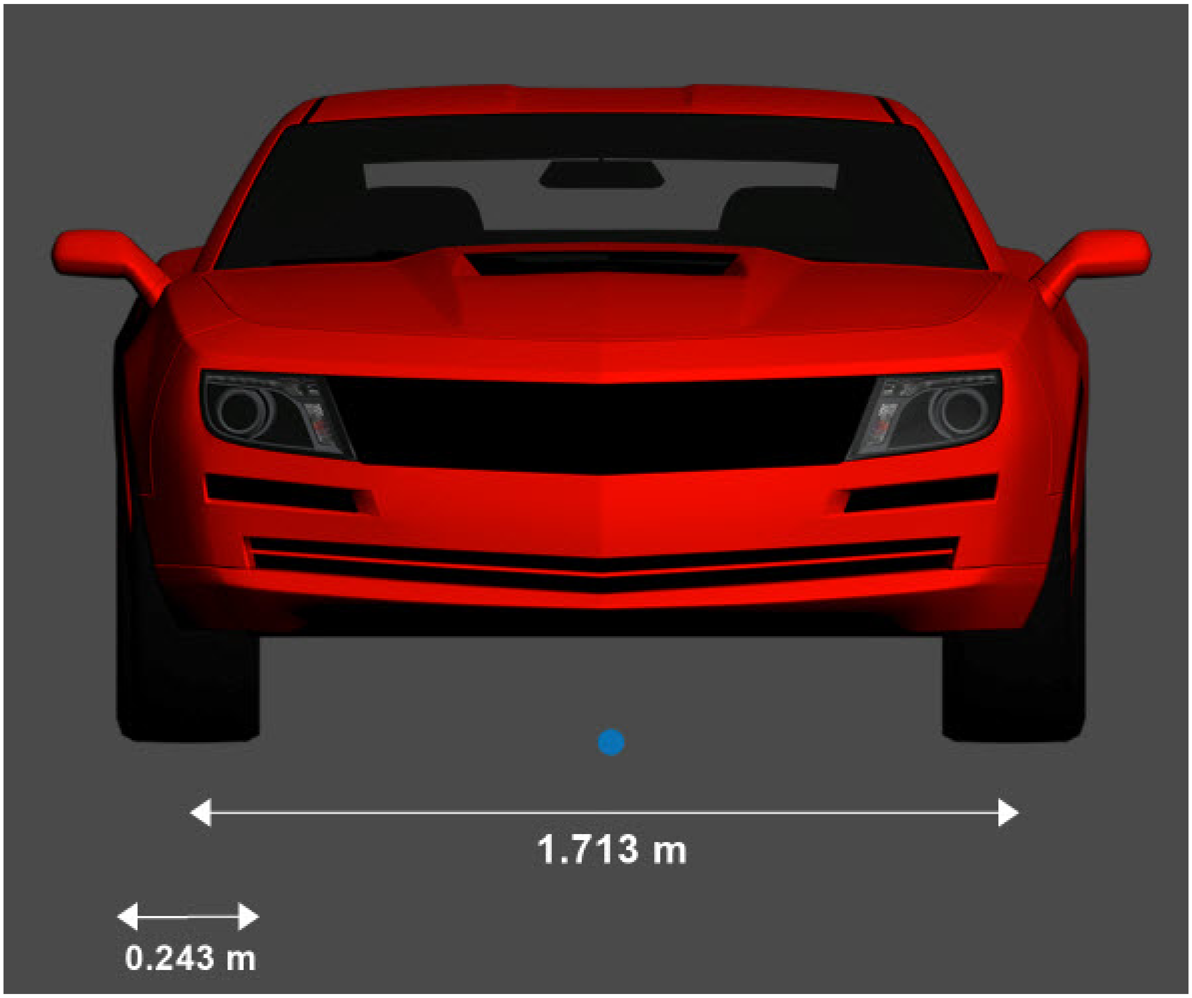
diagram



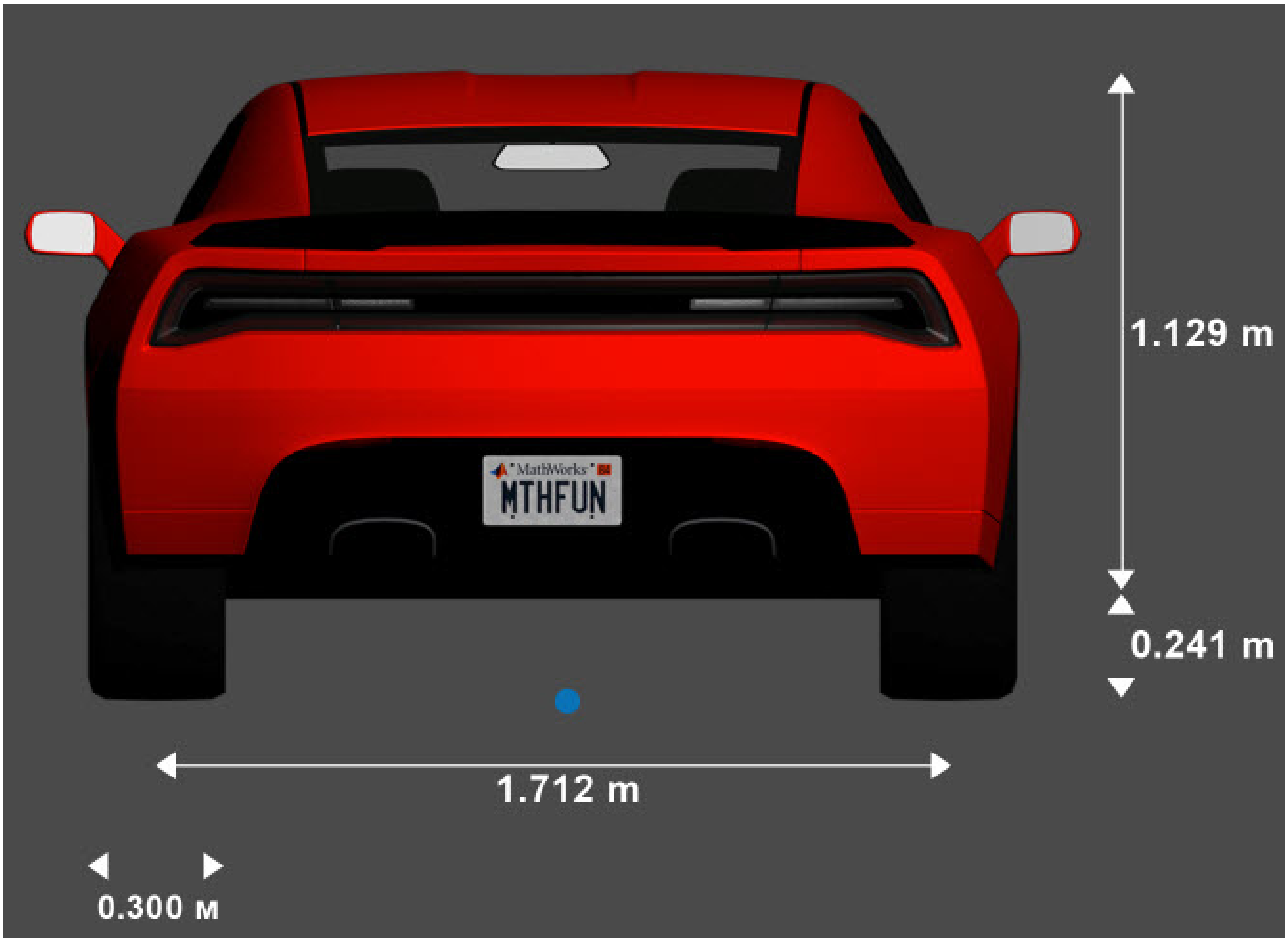
Side view — Vehicle length, front overhang, and rear overhang dimensions
diagram



Front view — Tire width and front axle dimensions
diagram



`Rear view` — Vehicle height and rear axle dimensions
diagram



Sensor Mounting Locations

In the 3D simulation sensor blocks, use the **Mounting location** parameter to mount sensors at predefined locations on the vehicle. The table shows the *X*, *Y*, and *Z* positions of the mounting locations relative to the vehicle origin. These locations are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when facing forward.
- The *Z*-axis points up from the ground.

Muscle Car — Sensor Locations Relative to Vehicle Origin

Mounting Location	X (m)	Y (m)	Z (m)
Front bumper	2.47	0	0.45
Rear bumper	–2.47	0	0.45
Right mirror	0.43	–1.08	1.01
Left mirror	0.43	1.08	1.01
Rearview mirror	0.32	0	1.20
Hood center	1.28	0	1.14
Roof center	–0.25	0	1.58

Specify Muscle Car Vehicle Dimensions

When simulating a path planner in the 3D environment, the path planner must use a vehicle whose dimensions are consistent with one used in the 3D environment. To make these dimensions consistent, you can use a `vehicleDimensions` object.

Specify the dimensions of a Muscle Car vehicle in a `vehicleDimensions` object. Units are in meters. For an example that uses this object in a path planner, see “Visualize Automated Parking Valet Using Unreal Engine Simulation”.

```
centerToFront = 1.491;
centerToRear  = 1.529;
frontOverhang = 0.983;
rearOverhang  = 0.945;
vehicleWidth  = 2.009;
vehicleHeight = 1.370;
vehicleLength = centerToFront + centerToRear + frontOverhang + rearOverhang;

muscleCarDims = vehicleDimensions(vehicleLength,vehicleWidth,vehicleHeight, ...
    'FrontOverhang',frontOverhang,'RearOverhang',rearOverhang)
```

```
muscleCarDims = 
  vehicleDimensions with properties:

           Length: 4.9480
            Width: 2.0090
           Height: 1.3700
        Wheelbase: 3.0200
     RearOverhang: 0.9450
    FrontOverhang: 0.9830
       WorldUnits: 'meters'
```

See Also
Box Truck | **Hatchback** | **Sedan** | **Small Pickup Truck** | **Sport Utility Vehicle**

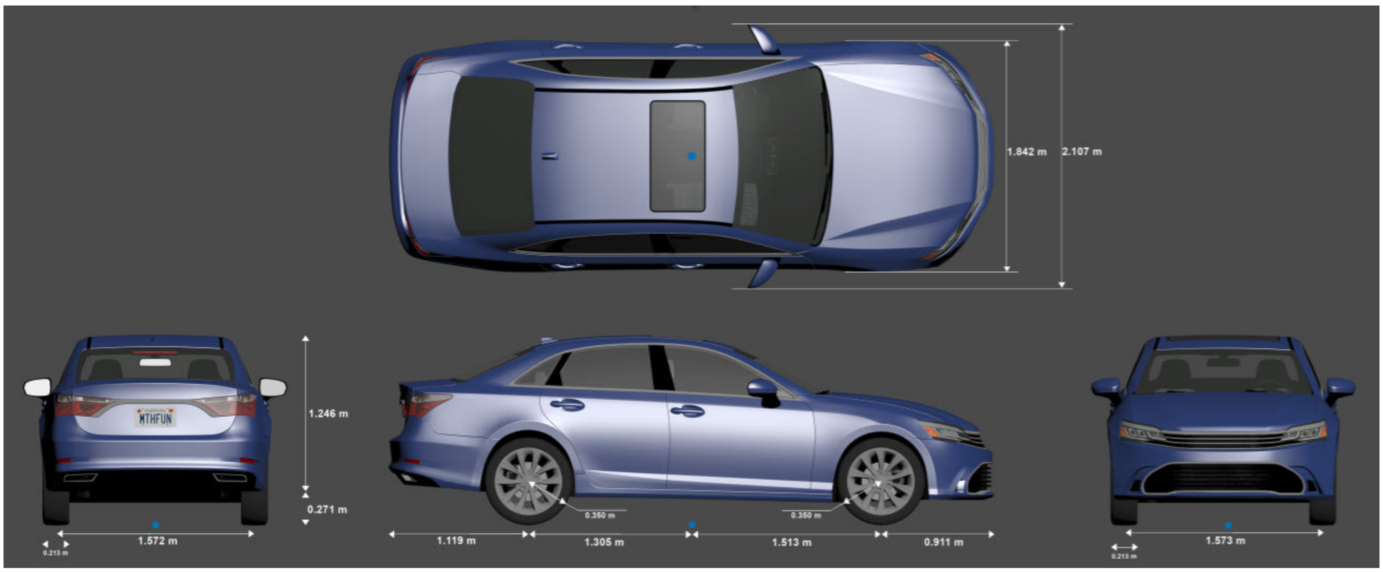
Topics
“Unreal Engine Simulation for Automated Driving”
“Coordinate Systems for Unreal Engine Simulation in Automated Driving Toolbox”

Sedan

Sedan vehicle dimensions

Description

Sedan is one of the vehicles that you can use within the 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The diagram provides the dimensions of this vehicle. The height dimensions are with respect to the vertical ground plane. The length and width dimensions are with respect to the origin of the vehicle in the vehicle coordinate system. The origin is on the ground, at the geometric center of the vehicle. For more detailed views of these diagrams, see the **Dimensions** section.

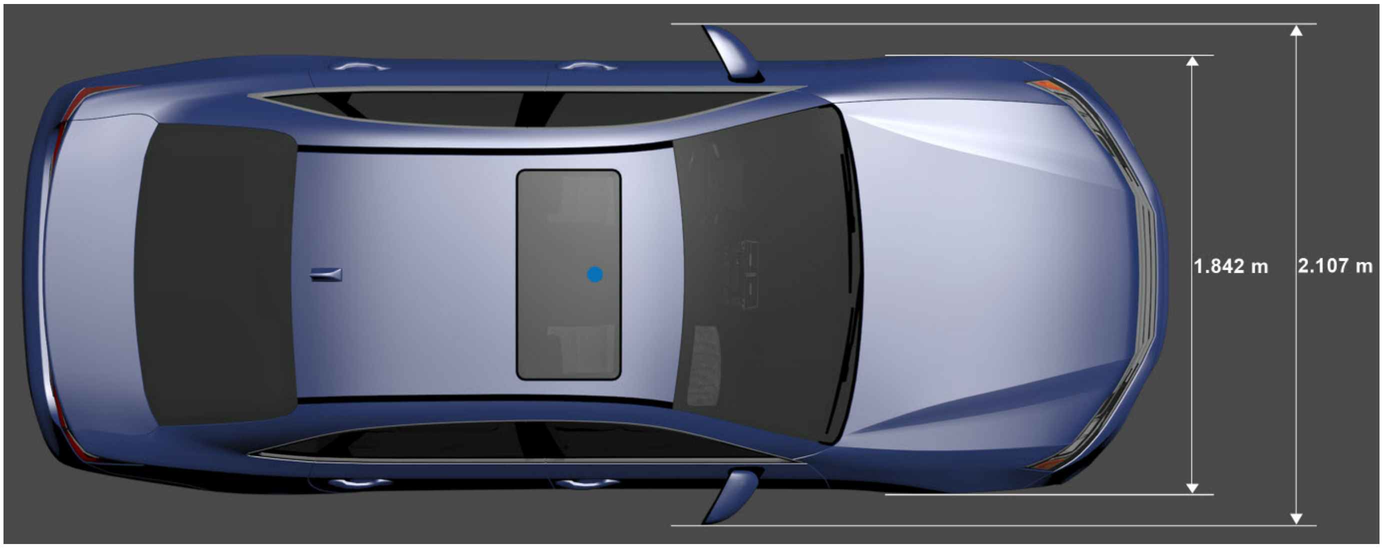


To add this type of vehicle to the 3D simulation environment:

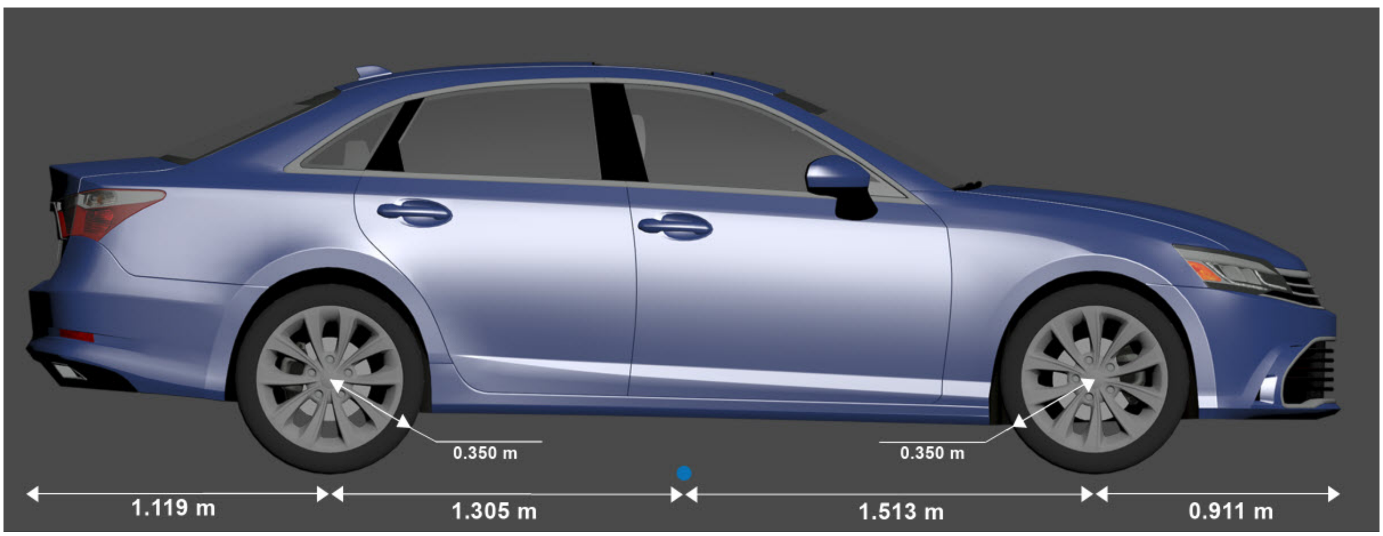
1. Add a Simulation 3D Vehicle with Ground Following block to your Simulink model.
2. In the block, set the **Type** parameter to `Sedan`.

Dimensions

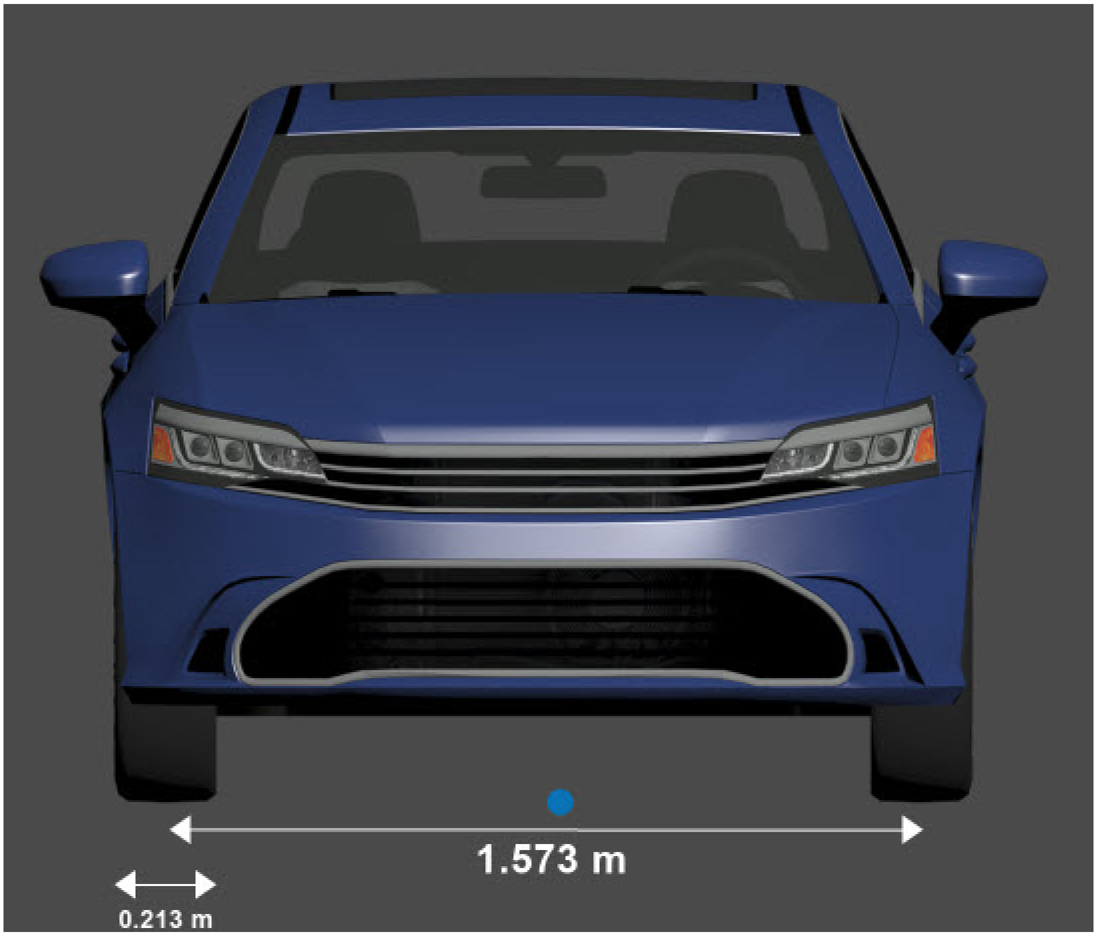
`Top-down view` — Vehicle width dimensions
diagram



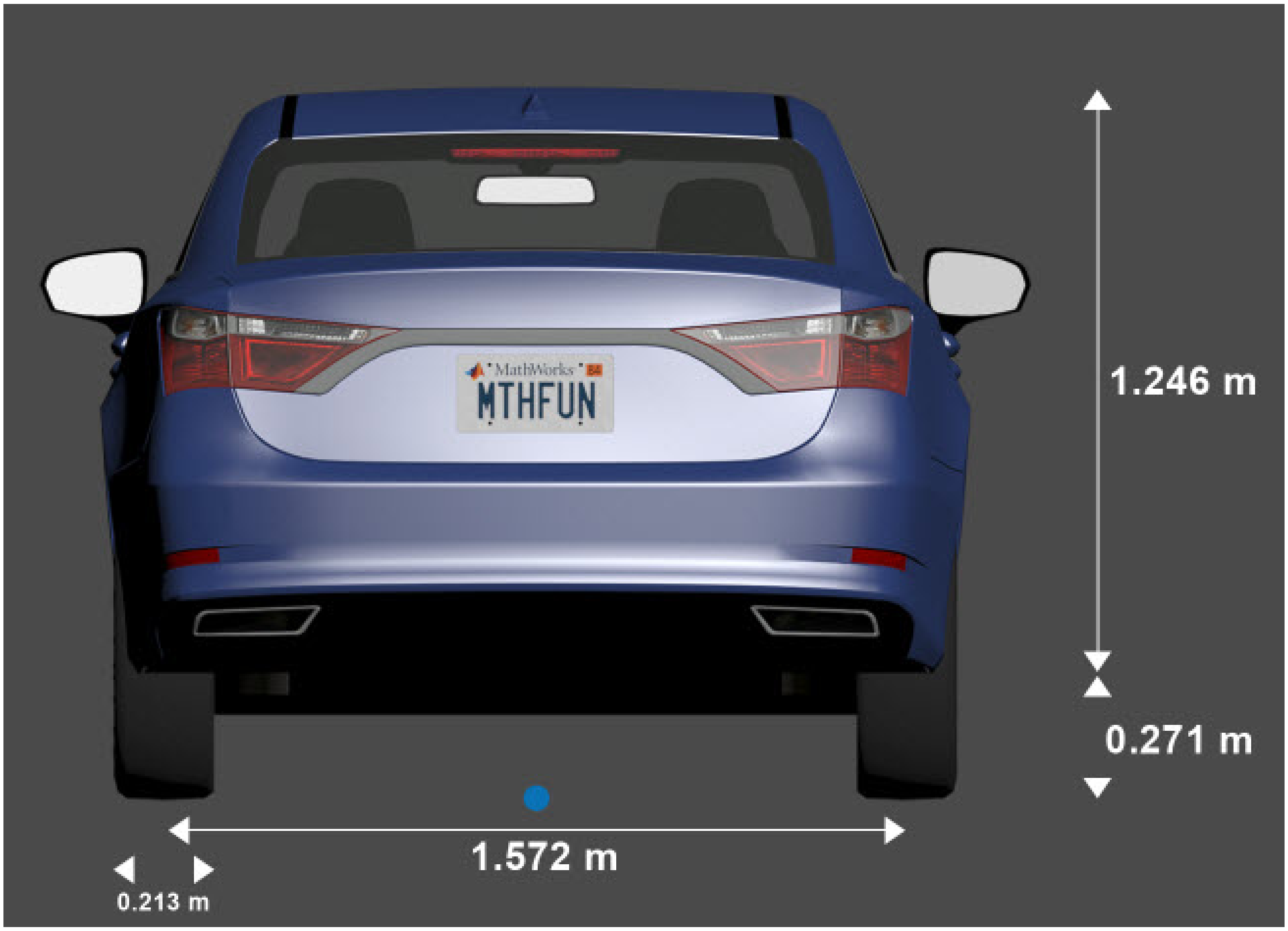
Side view — Vehicle length, front overhang, and rear overhang dimensions
diagram



Front view — Tire width and front axle dimensions
diagram



Rear view — Vehicle height and rear axle dimensions
diagram



Sensor Mounting Locations

In the 3D simulation sensor blocks, use the **Mounting location** parameter to mount sensors at predefined locations on the vehicle. The table shows the *X*, *Y*, and *Z* positions of the mounting locations relative to the vehicle origin. These locations are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when facing forward.
- The *Z*-axis points up from the ground.

Sedan — Sensor Locations Relative to Vehicle Origin

Mounting Location	X (m)	Y (m)	Z (m)
Front bumper	2.42	0	0.51
Rear bumper	–2.42	0	0.51
Right mirror	0.59	–0.94	1.09
Left mirror	0.59	0.94	1.09
Rearview mirror	0.43	0	1.31
Hood center	1.46	0	1.11
Roof center	–0.45	0	1.69

Specify Sedan Vehicle Dimensions

When simulating a path planner in the 3D environment, the path planner must use a vehicle whose dimensions are consistent with one used in the 3D environment. To make these dimensions consistent, you can use a `vehicleDimensions` object.

Specify the dimensions of a Sedan vehicle in a `vehicleDimensions` object. Units are in meters. For an example that uses this object in a path planner, see “Visualize Automated Parking Valet Using Unreal Engine Simulation”.

```
centerToFront = 1.513;
centerToRear  = 1.305;
frontOverhang = 0.911;
rearOverhang  = 1.119;
vehicleWidth  = 1.842;
vehicleHeight = 1.517;
vehicleLength = centerToFront + centerToRear + frontOverhang + rearOverhang;

sedanDims = vehicleDimensions(vehicleLength,vehicleWidth,vehicleHeight, ...
    'FrontOverhang',frontOverhang,'RearOverhang',rearOverhang)
```

```
sedanDims = 
  vehicleDimensions with properties:

           Length: 4.8480
            Width: 1.8420
           Height: 1.5170
        Wheelbase: 2.8180
     RearOverhang: 1.1190
    FrontOverhang: 0.9110
       WorldUnits: 'meters'
```

See Also

Box Truck | Hatchback | Muscle Car | Small Pickup Truck | Sport Utility Vehicle

Topics

“Unreal Engine Simulation for Automated Driving”
“Coordinate Systems in Automated Driving Toolbox”

Sport Utility Vehicle

Sport utility vehicle dimensions

Description

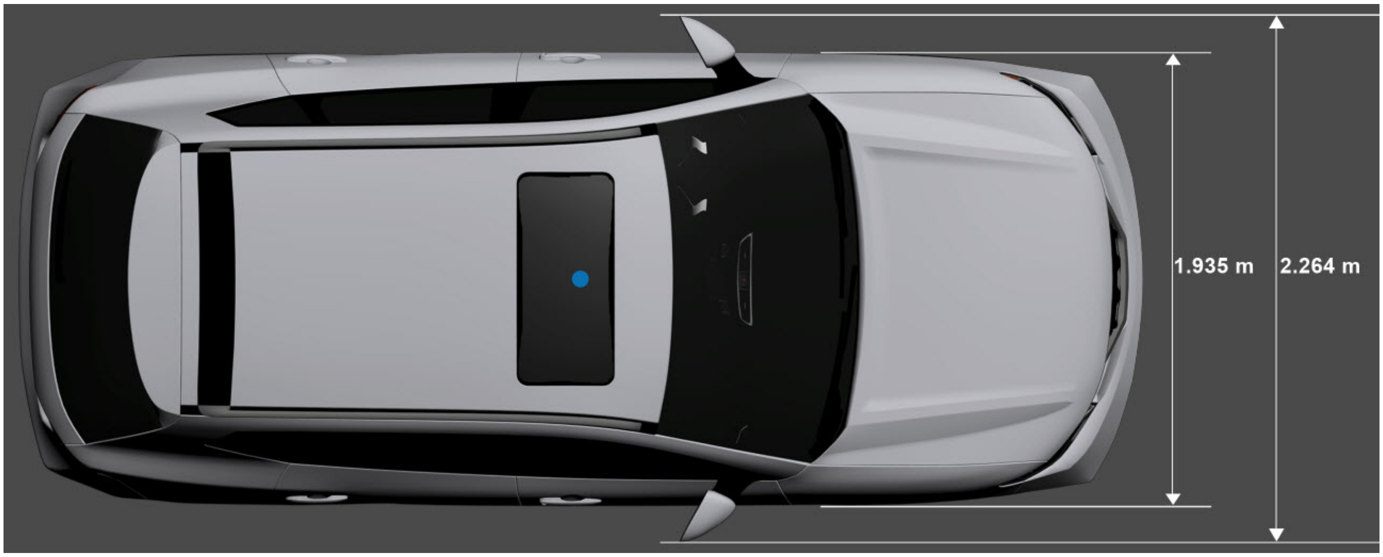
Sport Utility Vehicle is one of the vehicles that you can use within the 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The following diagram provides the dimensions of this vehicle. The height dimensions are with respect to the vertical ground plane. The length and width dimensions are with respect to the origin of the vehicle in the vehicle coordinate system. The origin is on the ground, at the geometric center of the vehicle. For more detailed views of these diagrams, see the **Dimensions** section.

To add this type of vehicle to the 3D simulation environment:

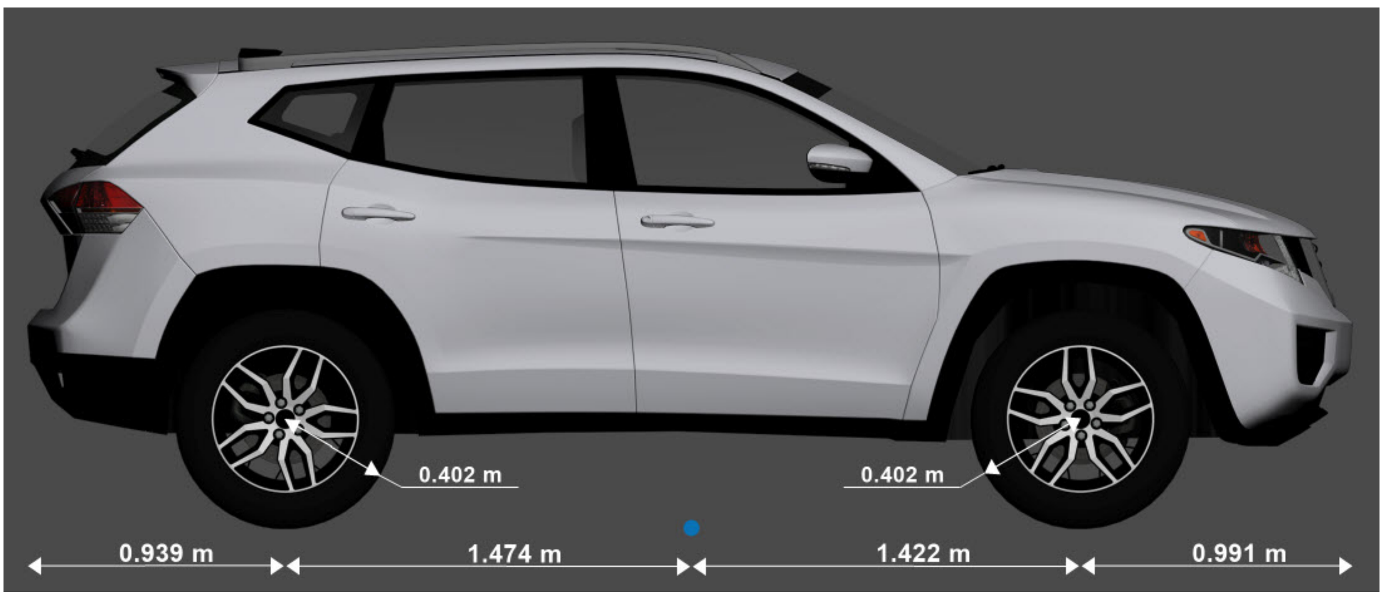
1. Add a Simulation 3D Vehicle with Ground Following block to your Simulink model.
2. In the block, set the **Type** parameter to `Sport utility vehicle`.

Dimensions

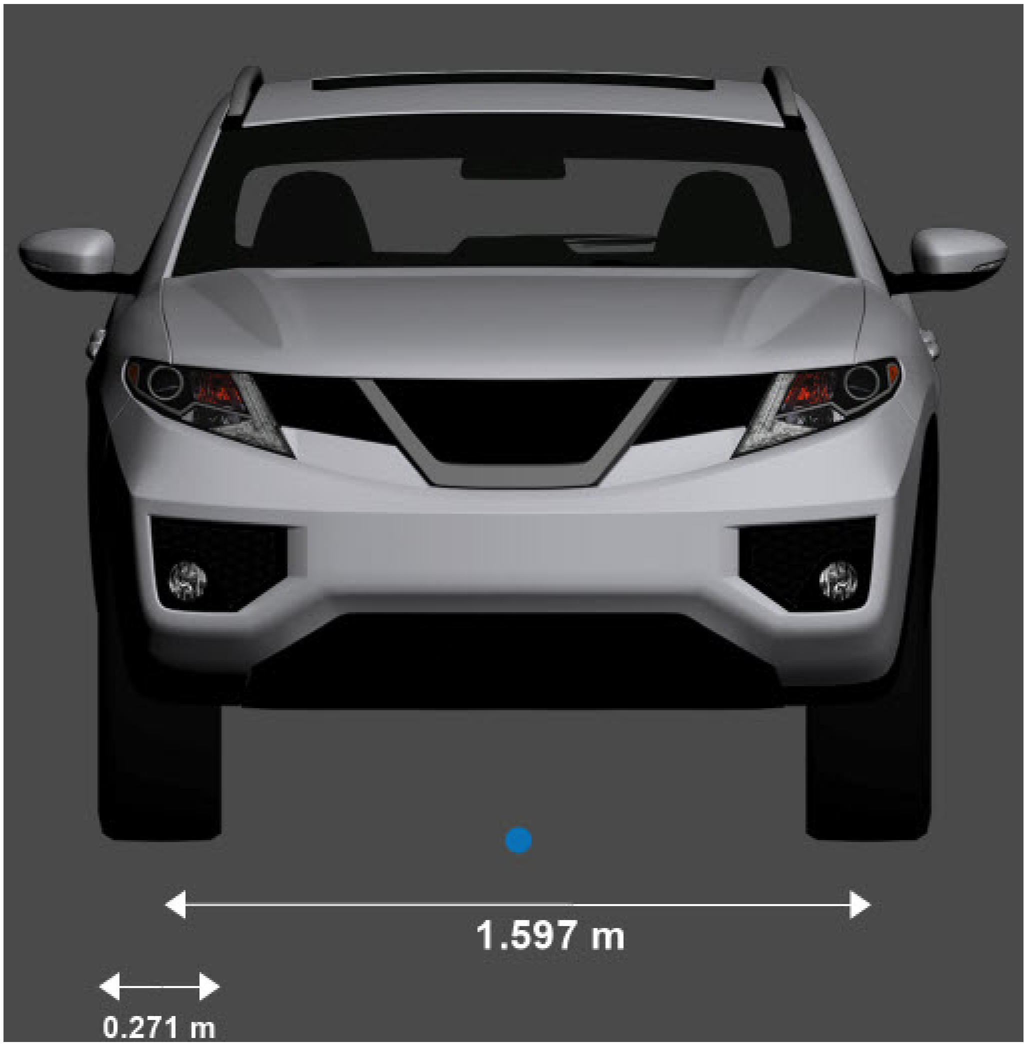
`Top-down view` — Vehicle width dimensions
diagram



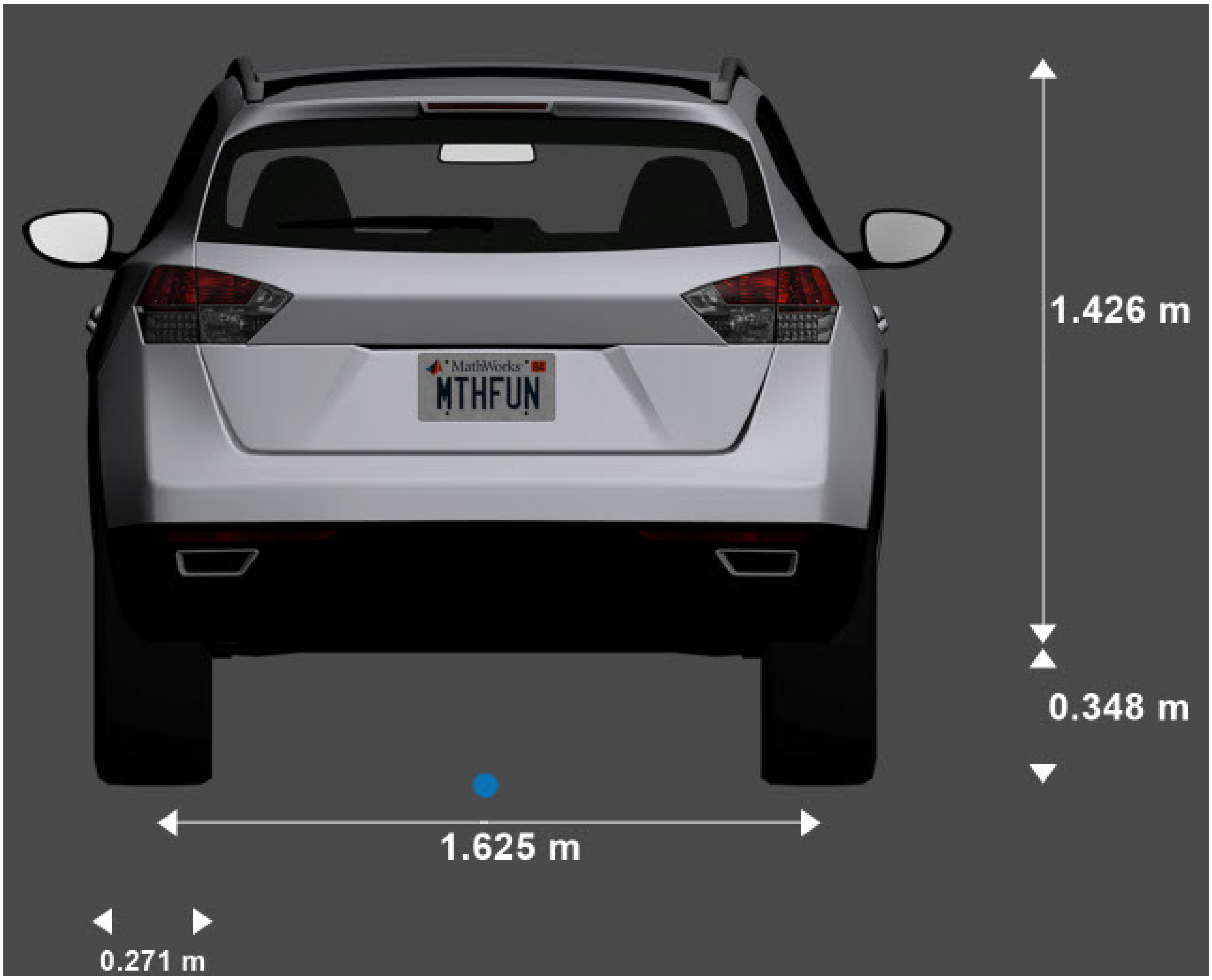
`Side view` — Vehicle length, front overhang, and rear overhang dimensions
diagram



`Front view` — Tire width and front axle dimensions
diagram



Rear view — Vehicle height and rear axle dimensions
diagram



Sensor Mounting Locations

In the 3D simulation sensor blocks, use the **Mounting location** parameter to mount sensors at predefined locations on the vehicle. The table shows the *X*, *Y*, and *Z* positions of the mounting locations relative to the vehicle origin. These locations are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when facing forward.
- The *Z*-axis points up from the ground.

Sport Utility Vehicle — Sensor Locations Relative to Vehicle Origin

Mounting Location	X (m)	Y (m)	Z (m)
Front bumper	2.42	0	0.51
Rear bumper	–2.42	0	0.51
Right mirror	0.60	–1	1.35
Left mirror	0.60	1	1.35
Rearview mirror	0.39	0	1.55
Hood center	1.58	0	1.39
Roof center	–0.56	0	2

Specify Sport Utility Vehicle Dimensions

When simulating a path planner in the 3D environment, the path planner must use a vehicle whose dimensions are consistent with one used in the 3D environment. To make these dimensions consistent, you can use a `vehicleDimensions` object.

Specify the dimensions of a Sport Utility Vehicle in a `vehicleDimensions` object. Units are in meters. For an example that uses this object in a path planner, see “Visualize Automated Parking Valet Using Unreal Engine Simulation”.

```
centerToFront = 1.422;
centerToRear  = 1.474;
frontOverhang = 0.991;
rearOverhang  = 0.939;
vehicleWidth  = 1.935;
vehicleHeight = 1.774;
vehicleLength = centerToFront + centerToRear + frontOverhang + rearOverhang;

suvDims = vehicleDimensions(vehicleLength,vehicleWidth,vehicleHeight, ...
    'FrontOverhang',frontOverhang,'RearOverhang',rearOverhang)
```

```
suvDims = 
  vehicleDimensions with properties:

           Length: 4.8260
            Width: 1.9350
           Height: 1.7740
        Wheelbase: 2.8960
     RearOverhang: 0.9390
    FrontOverhang: 0.9910
       WorldUnits: 'meters'
```

See Also
Box Truck | Hatchback | Muscle Car | Sedan | Small Pickup Truck

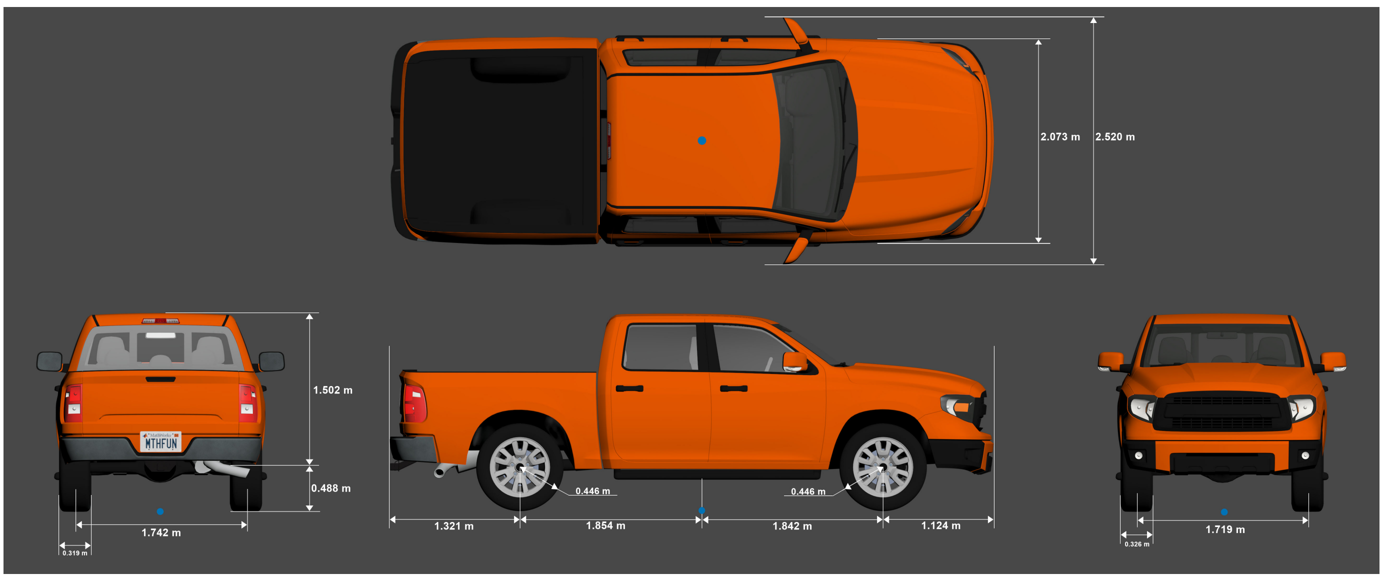
Topics
“Unreal Engine Simulation for Automated Driving”
“Coordinate Systems in Automated Driving Toolbox”

Small Pickup Truck

Small pickup truck vehicle dimensions

Description

Small Pickup Truck is one of the vehicles that you can use within the 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The following diagram provides the dimensions of this vehicle. The height dimensions are with respect to the vertical ground plane. The length and width dimensions are with respect to the origin of the vehicle in the vehicle coordinate system. The origin is on the ground, at the geometric center of the vehicle. For more detailed views of these diagrams, see the **Dimensions** section.

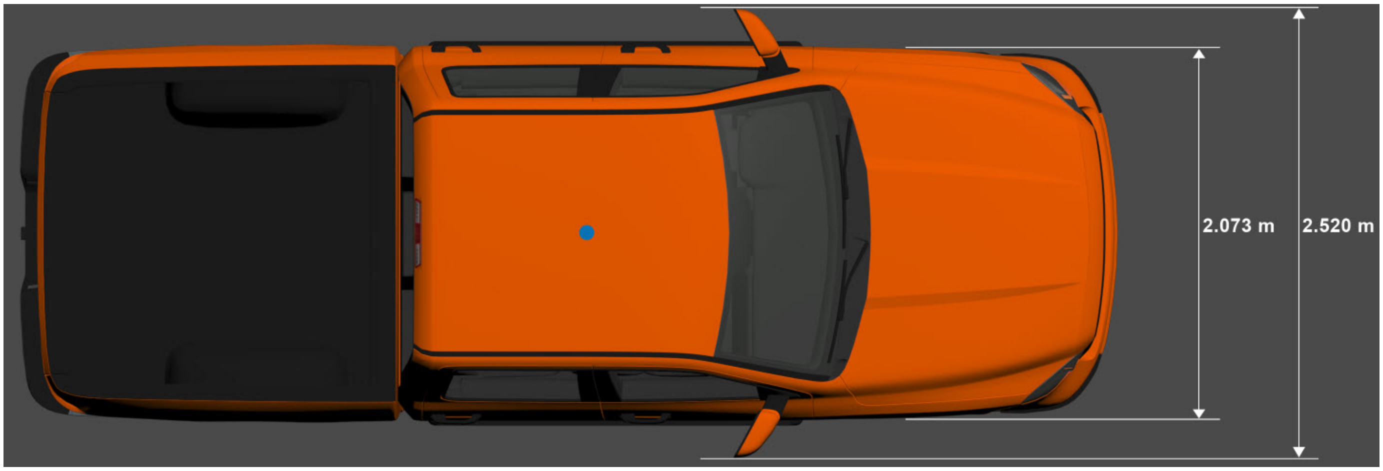


To add this type of vehicle to the 3D simulation environment:

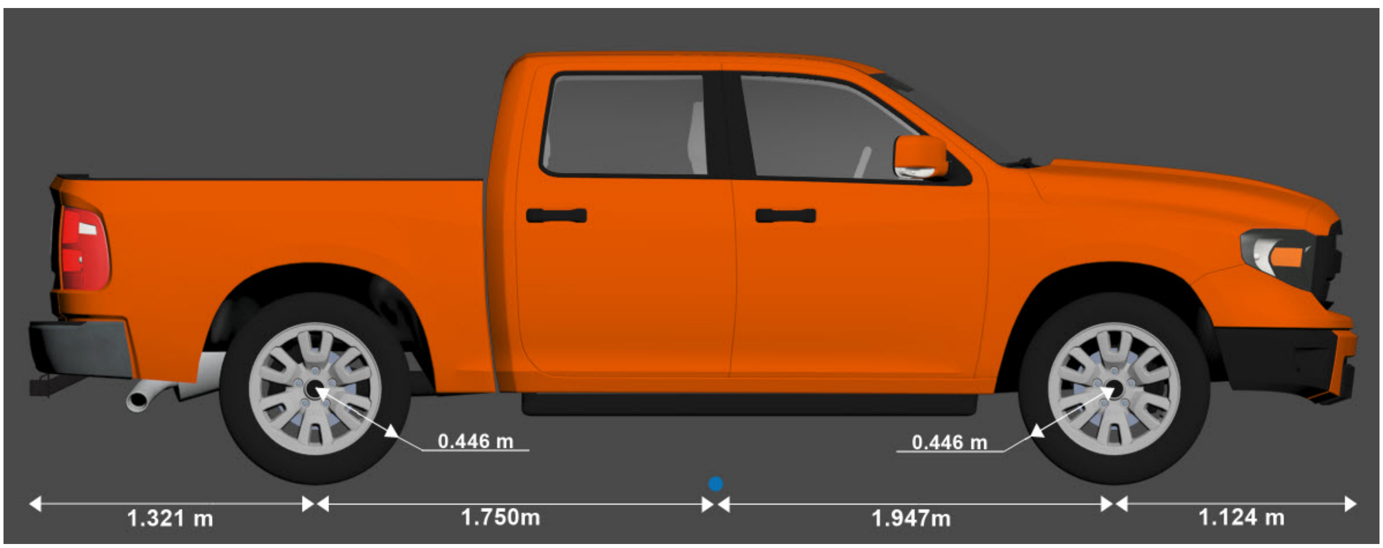
1. Add a Simulation 3D Vehicle with Ground Following block to your Simulink model.
2. In the block, set the **Type** parameter to `Small pickup truck`.

Dimensions

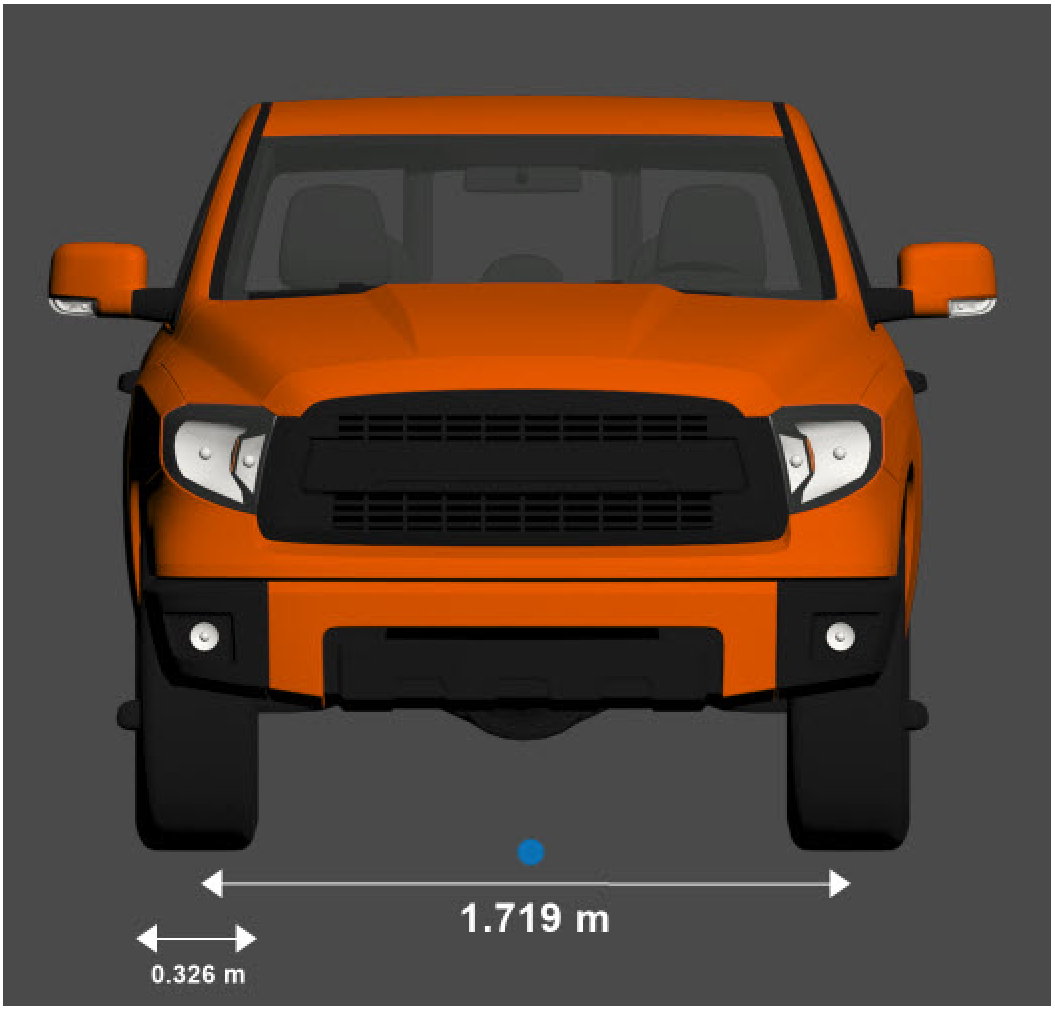
`Top-down view` — Vehicle width dimensions
diagram



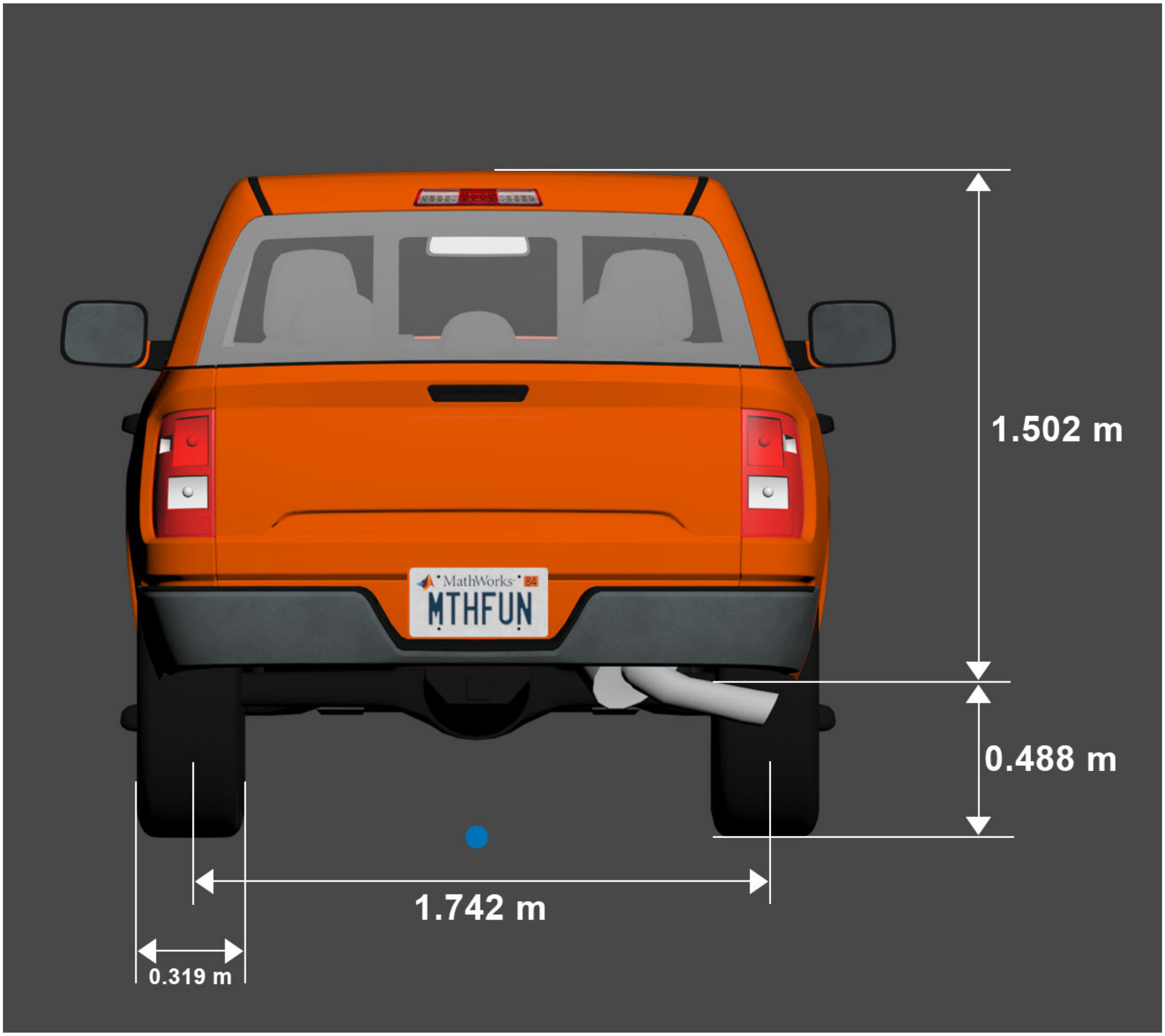
`Side view` — Vehicle length, front overhang, and rear overhang dimensions
diagram



`Front view` — Tire width and front axle dimensions
diagram



Rear view — Vehicle height and rear axle dimensions
diagram



Sensor Mounting Locations

In the 3D simulation sensor blocks, use the **Mounting location** parameter to mount sensors at predefined locations on the vehicle. The table shows the *X*, *Y*, and *Z* positions of the mounting locations relative to the vehicle origin. These locations are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when facing forward.
- The *Z*-axis points up from the ground.

Small Pickup Truck — Sensor Locations Relative to Vehicle Origin

Mounting Location	X (m)	Y (m)	Z (m)
`Front bumper`	3.07	0	0.51
`Rear bumper`	–3.07	0	0.51
`Right mirror`	1.10	–1.13	1.52
`Left mirror`	1.10	1.13	1.52
`Rearview mirror`	0.85	0	1.77
`Hood center`	2.22	0	1.59
`Roof center`	0	0	2.27

Specify Small Pickup Truck Vehicle Dimensions

When simulating a path planner in the 3D environment, the path planner must use a vehicle whose dimensions are consistent with one used in the 3D environment. To make these dimensions consistent, you can use a `vehicleDimensions` object.

Specify the dimensions of a Small Pickup Truck vehicle in a `vehicleDimensions` object. Units are in meters. For an example that uses this object in a path planner, see “Visualize Automated Parking Valet Using Unreal Engine Simulation”.

```
centerToFront = 1.947;
centerToRear  = 1.750;
frontOverhang = 1.124;
rearOverhang  = 1.321;
vehicleWidth  = 2.073;
vehicleHeight = 1.990;
vehicleLength = centerToFront + centerToRear + frontOverhang + rearOverhang;

smallPickupTruckDims = vehicleDimensions(vehicleLength,vehicleWidth,vehicleHeight, ...
    'FrontOverhang',frontOverhang,'RearOverhang',rearOverhang)
```

```
smallPickupTruckDims = 
  vehicleDimensions with properties:

           Length: 6.1420
            Width: 2.0730
           Height: 1.9900
        Wheelbase: 3.6970
     RearOverhang: 1.3210
    FrontOverhang: 1.1240
       WorldUnits: 'meters'
```

See Also

Box Truck | **Hatchback** | **Muscle Car** | **Sedan** | **Sport Utility Vehicle**

Topics

“Unreal Engine Simulation for Automated Driving”
“Coordinate Systems in Automated Driving Toolbox”

Box Truck

Box truck vehicle dimensions

Description

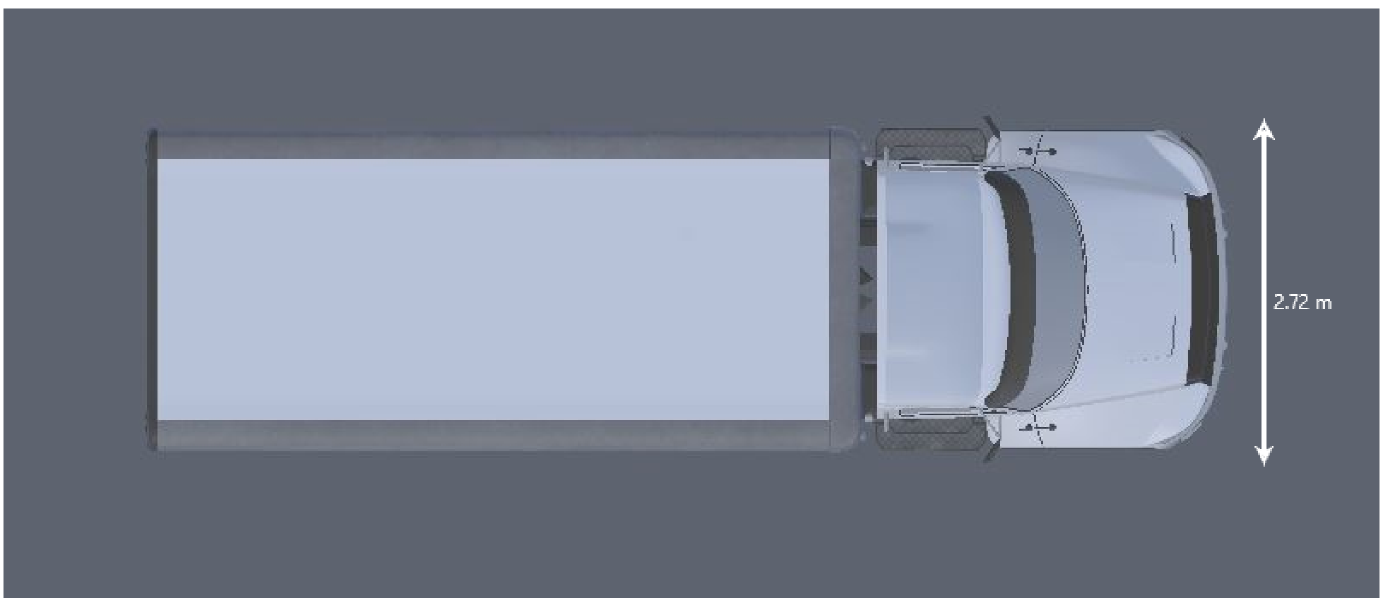
Box truck is one of the vehicles that you can use within the 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The following diagram provides the dimensions of this vehicle. The height dimensions are with respect to the vertical ground plane. The length and width dimensions are with respect to the origin of the vehicle in the vehicle coordinate system. The origin is on the ground, at the geometric center of the vehicle. For more detailed views of these diagrams, see the **Dimensions** section.

To add this type of vehicle to the 3D simulation environment:

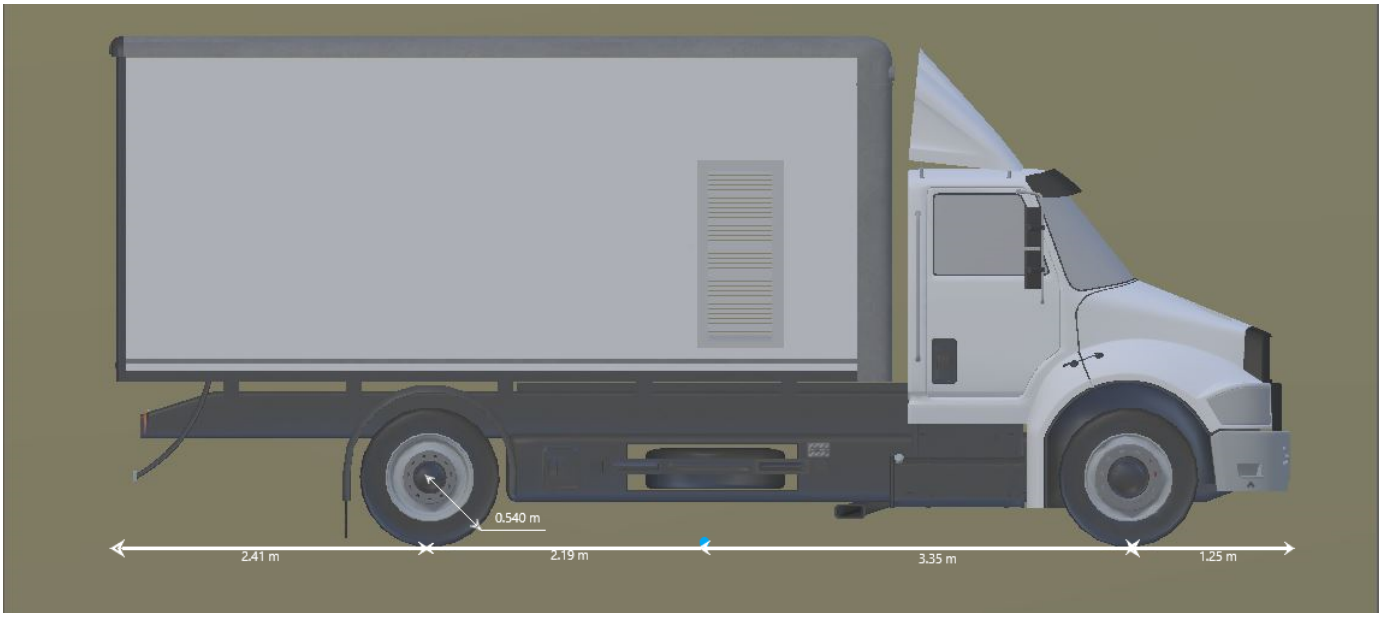
1. Add a Simulation 3D Vehicle with Ground Following block to your Simulink model.
2. In this block, set the **Type** parameter to `Box truck`.

Dimensions

`Top-down view` — Vehicle width dimensions
diagram



`Side view` — Vehicle length, front overhang, and rear overhang dimensions
diagram



`Front view` — Tire width and front axle dimensions
diagram

`Rear view` — Vehicle height and rear axle dimensions
diagram

Sensor Mounting Locations

In the 3D simulation sensor blocks, use the **Mounting location** parameter to mount sensors at predefined locations on the vehicle. The table shows the *X*, *Y*, and *Z* positions of the mounting locations relative to the vehicle origin. These locations are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when facing forward.
- The *Z*-axis points up from the ground.

Box Truck — Sensor Locations Relative to Vehicle Origin

Mounting Location	X (m)	Y (m)	Z (m)
Front bumper	5.10	0	0.60
Rear bumper	-5	0	0.60
Right mirror	2.90	1.60	2.10
Left mirror	2.90	-1.60	2.10
Rearview mirror	2.60	0.20	2.60
Hood center	3.80	0	2.10
Roof center	1.30	0	4.20

Specify Box Truck Vehicle Dimensions

When simulating a path planner in the 3D environment, the path planner must use a vehicle whose dimensions are consistent with the one used in the environment. To make these dimensions consistent, you can use a `vehicleDimensions` object.

Specify the dimensions of a Box Truck vehicle in a `vehicleDimensions` object. Units are in meters. For an example that uses this object in a path planner, see “Visualize Automated Parking Valet Using Unreal Engine Simulation”.

```
centerToFront = 3.35;
centerToRear  = 2.19;
frontOverhang = 1.25;
rearOverhang  = 2.41;
vehicleWidth  = 2.72;
vehicleHeight = 4.01;
vehicleLength = centerToFront + centerToRear + frontOverhang + rearOverhang;

boxTruckDims = vehicleDimensions(vehicleLength,vehicleWidth,vehicleHeight, ...
    'FrontOverhang',frontOverhang,'RearOverhang',rearOverhang)
```

```
boxTruckDims = 
  vehicleDimensions with properties:

           Length: 9.2000
            Width: 2.7200
           Height: 4.0100
        Wheelbase: 5.5400
     RearOverhang: 2.4100
    FrontOverhang: 1.2500
       WorldUnits: 'meters'
```

See Also

Hatchback | Muscle Car | Sedan | Small Pickup Truck | Sport Utility Vehicle

Topics

“Unreal Engine Simulation for Automated Driving”

“Coordinate Systems in Automated Driving Toolbox”