Automated Driving System Toolbox™

Reference

MATLAB®

R2018**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.
3 Apple Hill Drive
Natick, MA 01760-2098

Automated Driving System 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)

Contents

1

Apps in Automated Driving System Toolbox

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:

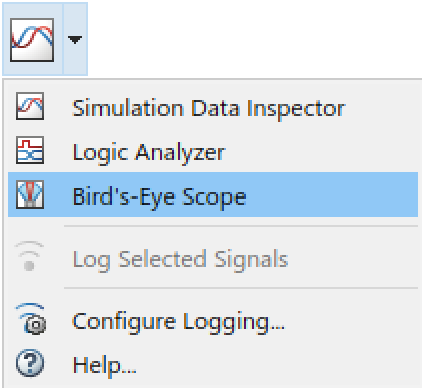
- Inspect the coverage areas of radar and vision sensors.
- Analyze the sensor detections of actors, road boundaries, and lane boundaries.
- Analyze the tracking results of moving actors within the scenario.

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

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



Your most recent choice for data visualization is saved across Simulink sessions.

Examples

- "Visualize Sensor Data and Tracks in Bird's-Eye Scope"
- "Sensor Fusion Using Synthetic Radar and Vision Data in Simulink"
- "Lane Keeping Assist with Lane Detection"
- "Adaptive Cruise Control with Sensor Fusion"
- "Lateral Control Tutorial"
- "Automatic Emergency Braking with Sensor Fusion"

Parameters

Global Settings

To access the global settings of the **Bird's-Eye Scope**, from the scope toolstrip, click **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

`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, from the scope toolstrip, click **Properties**.

`Alpha` — Transparency of coverage area

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

Transparency of the coverage area, specified as 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.

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) | scalar in the range [0, 20]

Scale factor for the magnitude length of the velocity vectors, specified as a 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

- 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, from the model toolstrip, click the Disable Fast Restart button .
- Actors buses are supported only as outputs of the Scenario Reader block, such as the one used in the “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” example.

Definitions

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, lane markings, and actors in the scenario, including the ego vehicle
	
	You cannot modify this group or any of the signals within it.
		• Scenario Reader block (such as the one used in the “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” example)
		• Vision Detection Generator and Radar Detection Generator blocks (for actor profile information only, such as the length, width, and height of actors)
		    • If actor profile information is not set or is inconsistent between blocks, the scope sets the actor profiles to the block defaults.
		    • The profile of the ego vehicle is always set to the block defaults.

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

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

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.

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.

See Also

Detection Concatenation | Multi Object Tracker | Radar Detection Generator | Vision Detection Generator

Topics

"Visualize Sensor Data and Tracks in Bird's-Eye Scope"
"Sensor Fusion Using Synthetic Radar and Vision Data in Simulink"
"Lane Keeping Assist with Lane Detection"
"Adaptive Cruise Control with Sensor Fusion"
"Lateral Control Tutorial"
"Automatic Emergency Braking with Sensor Fusion"

Introduced in R2018b

Driving Scenario Designer

Design driving scenarios, configure sensors, and generate synthetic object detections

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 and radar sensors mounted on the ego car, and use these sensors to simulate detections of actors and lane boundaries in the 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].
- Export sensor detections to MATLAB®, or generate MATLAB code of the scenario that produced the detections.

You can use synthetic detections generated from a scenario to test your sensor fusion or control algorithms. To learn more about using the app, see Driving Scenario Designer.

Open the Driving Scenario Designer App

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

Examples

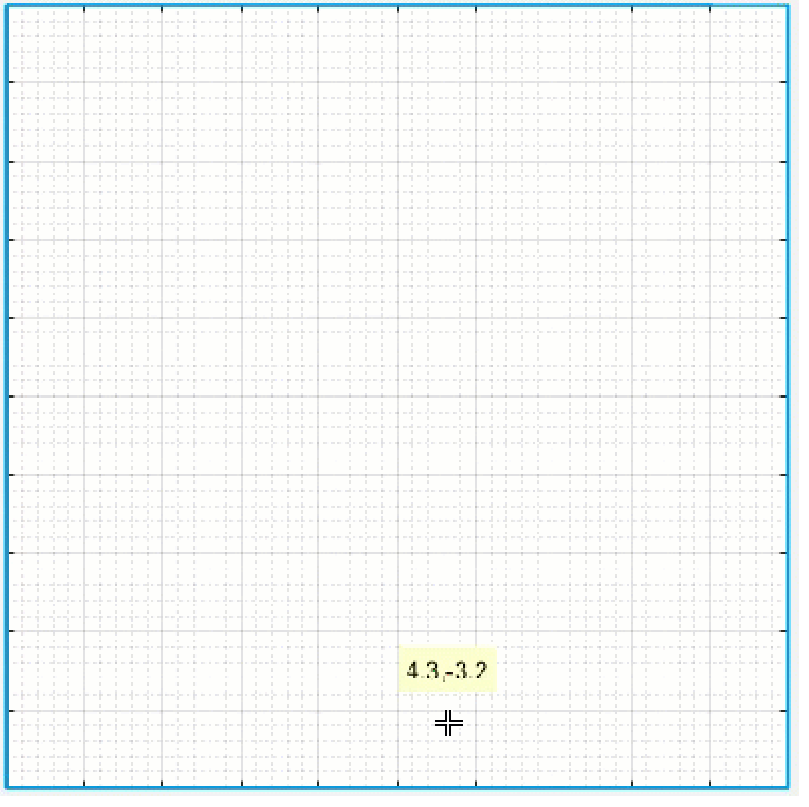
Build a Driving Scenario

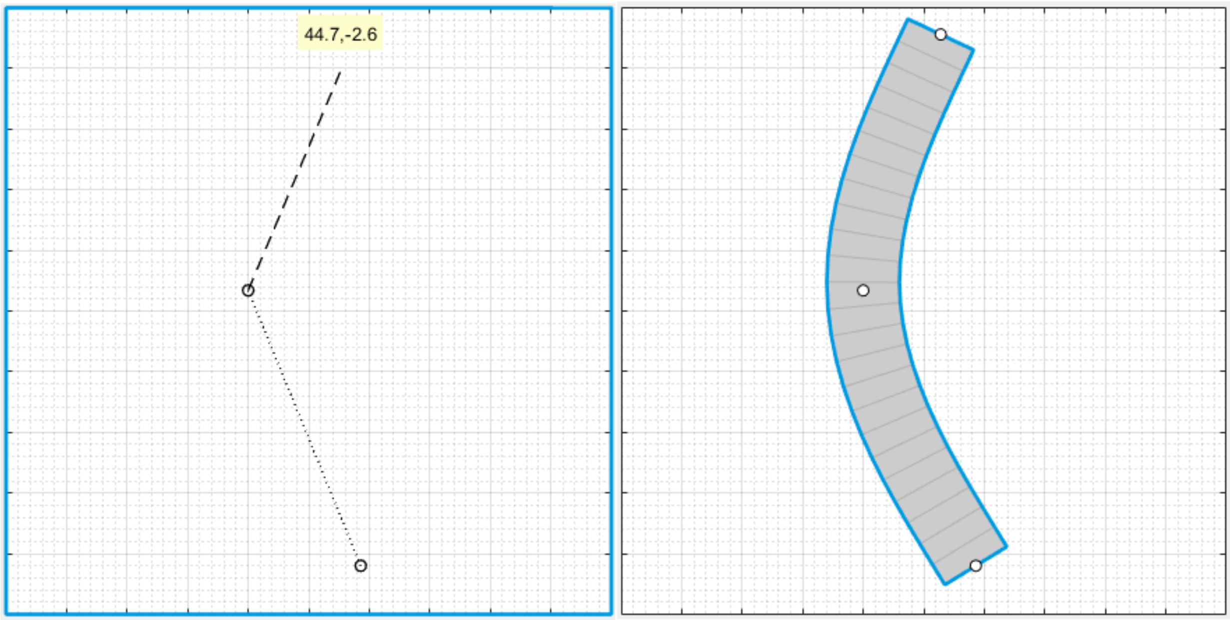
Build a driving scenario of a vehicle driving down a curved road, and export the road and vehicle models to the MATLAB workspace. For a more detailed example of building a driving scenario, see “Build a Driving Scenario and Generate Synthetic Detections”.

Open the **Driving Scenario Designer** app.

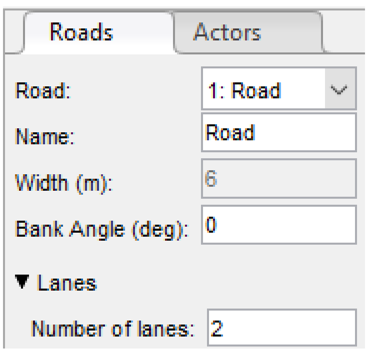
```
drivingScenarioDesigner
```

Create a curved road. From 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.

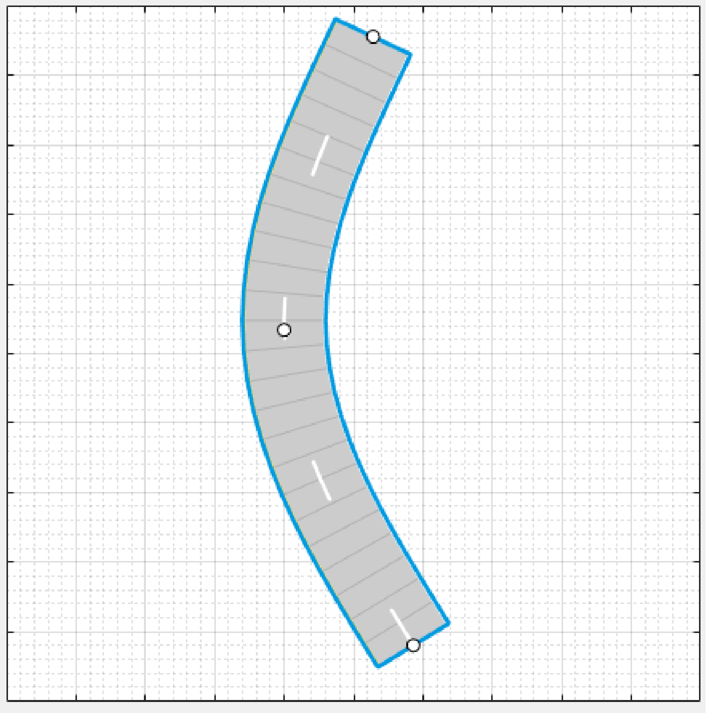




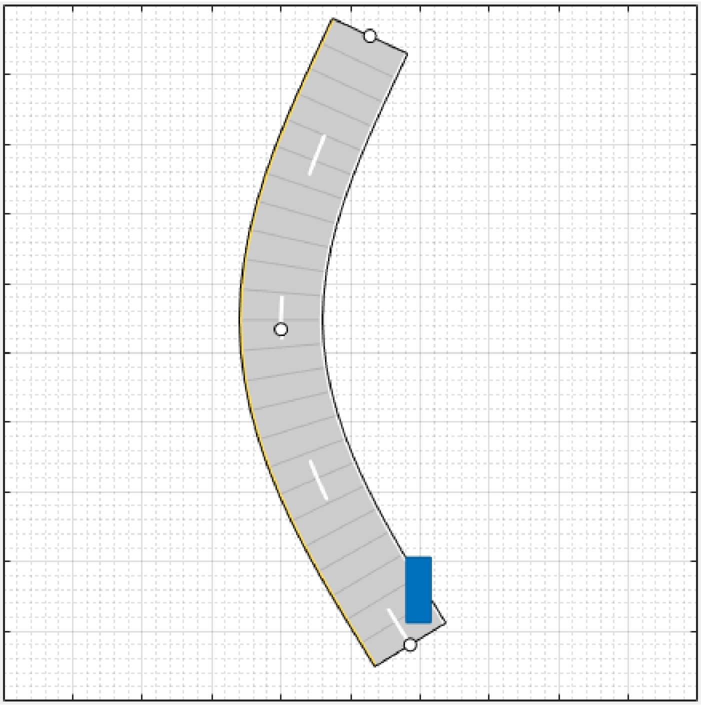
Add lanes to the road. In the left pane, on the **Roads** tab, expand the **Lanes** section. Set the **Number of lanes** to 2.



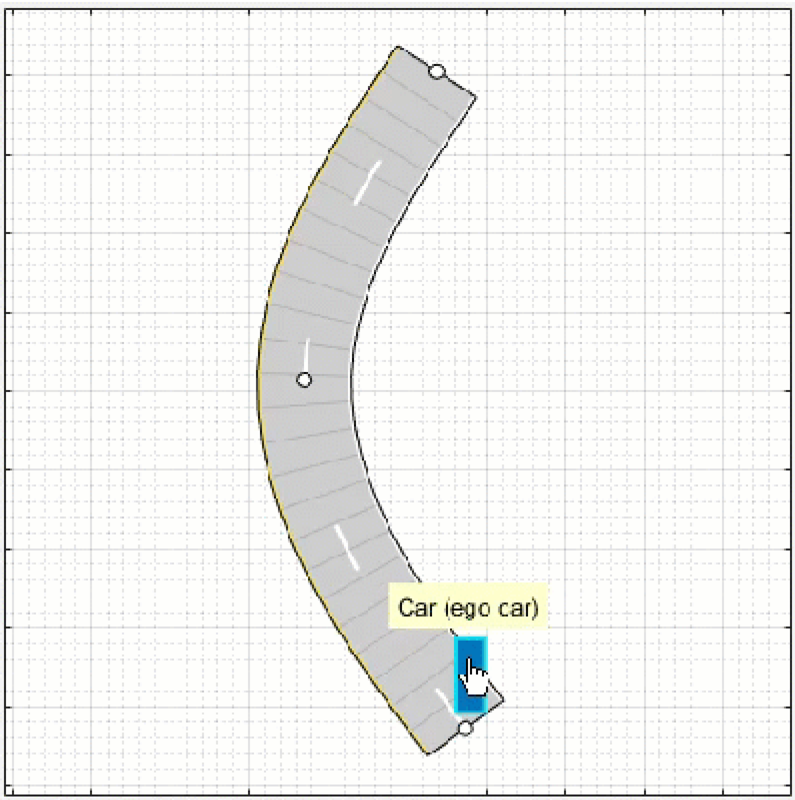
By default, the road is one-way and has solid lane markings on either side to indicate the shoulder.

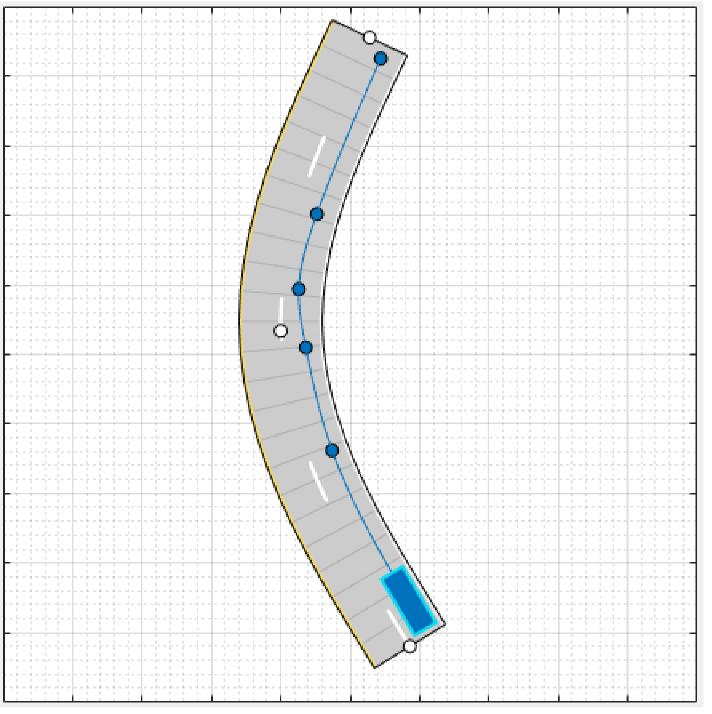


Add a vehicle at one end of the road. From the app toolstrip, select **Add Actor > Car**. Then click the road to set the initial position of the car.

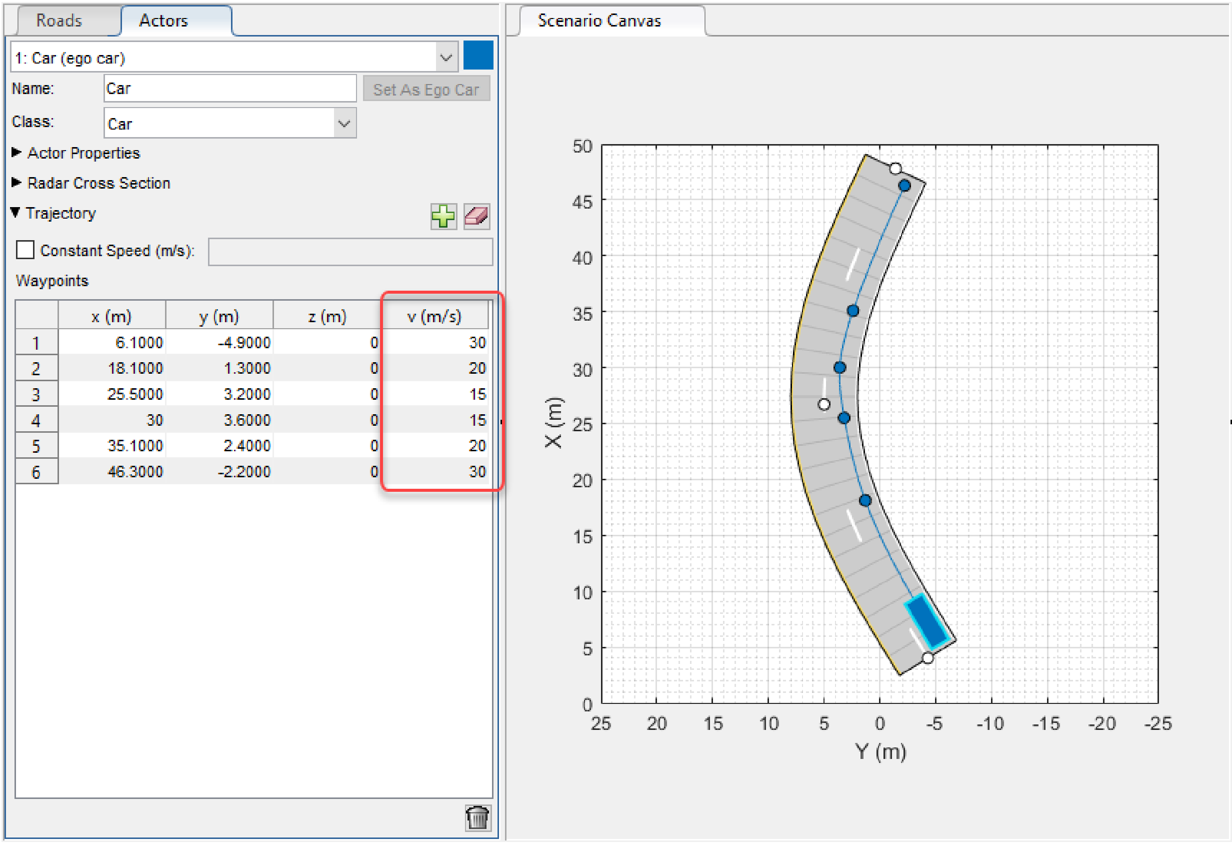


Set the driving path of the car. Right-click the car, select **Add Waypoints**, and add waypoints for the car to pass through. After you add the last waypoint, press **Enter**. The car autorotates in the direction of the first waypoint.


Car (ego car)



Adjust the speed of the car as it passes between waypoints. In the left pane, on the **Actors** tab, in the **Path** section, clear the **Constant Speed** check box. Then, in the **Waypoints** table, set the velocity, **v (m/s)**, of the car in m/s as it enters each waypoint segment. To model more realistic conditions, increase the speed of the car for the straight segments and decrease its speed for the curved segments. For example:



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

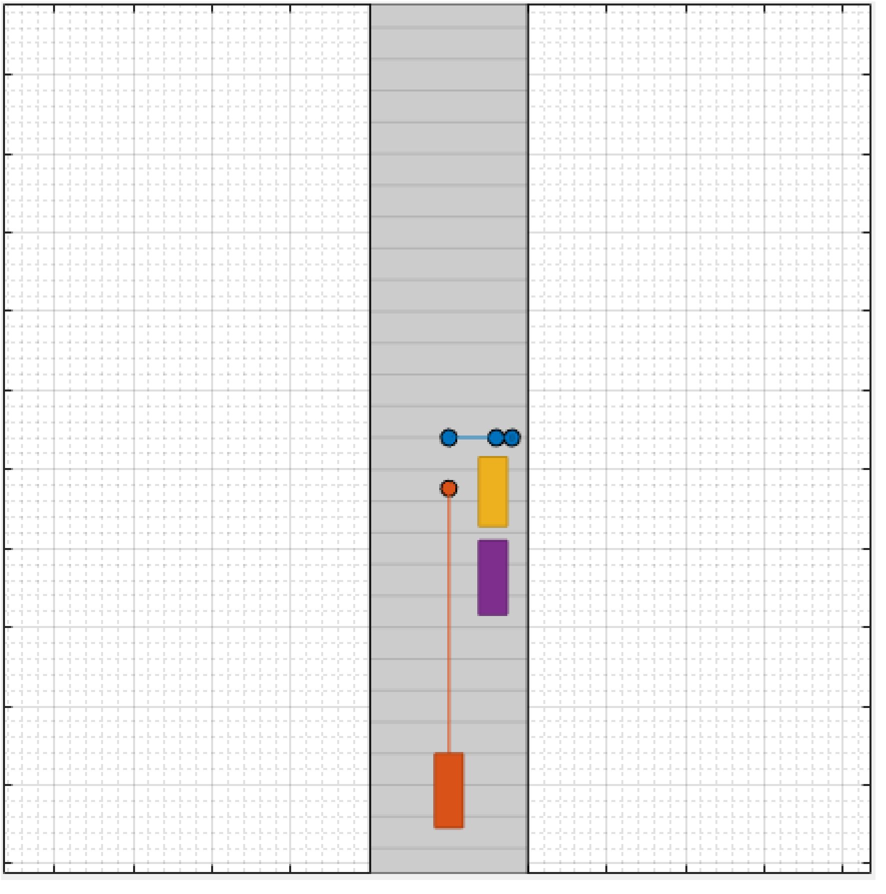
Generate Detections from Prebuilt Scenario

Generate vision sensor detections from a prebuilt driving scenario of a Euro NCAP test protocol.

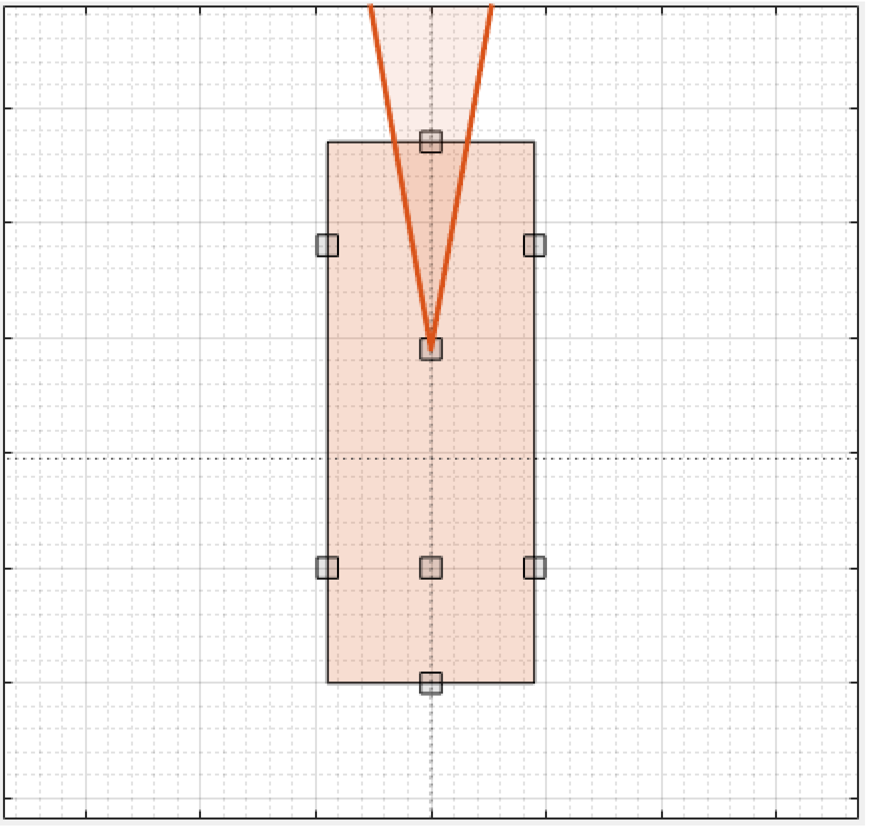
- For more details on prebuilt scenarios available from the app, see “Generate Synthetic Detections from a Prebuilt Driving Scenario”.
- For more details on available Euro NCAP scenarios, see “Generate Synthetic Detections from a Euro NCAP Scenario”.

Load a Euro NCAP automatic 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.

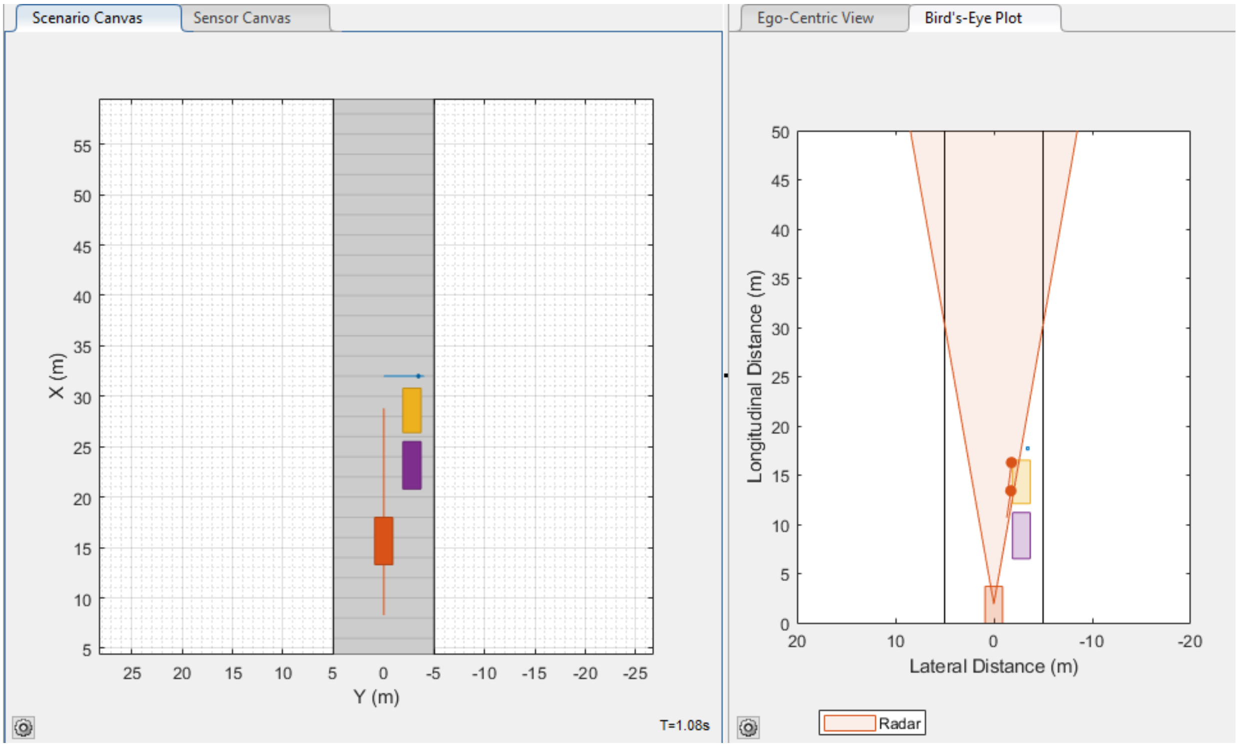
```
Path = fullfile(matlabroot,'toolbox','driving','drivingdata', ...
    'PrebuiltScenarios','EuroNCAP');
addpath(genpath(Path)) % Add folder to path
drivingScenarioDesigner('AEB_PedestrianChild_Nearside_50width.mat')
rmpath(path) % Remove folder from path
```



Add a front-facing radar sensor to the ego car. First click **Add Radar**. Then, on the **Sensor Canvas**, click the predefined sensor location at the front window of the car. By default, the radar is long-range.



Run the scenario. While the scenario simulation runs, inspect different aspects of the simulation 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**.



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

Add OpenDRIVE Road to Scenario

Import an OpenDRIVE road network into the **Driving Scenario Designer** app. For a more detailed example, see "Add OpenDRIVE Roads to Driving Scenario".

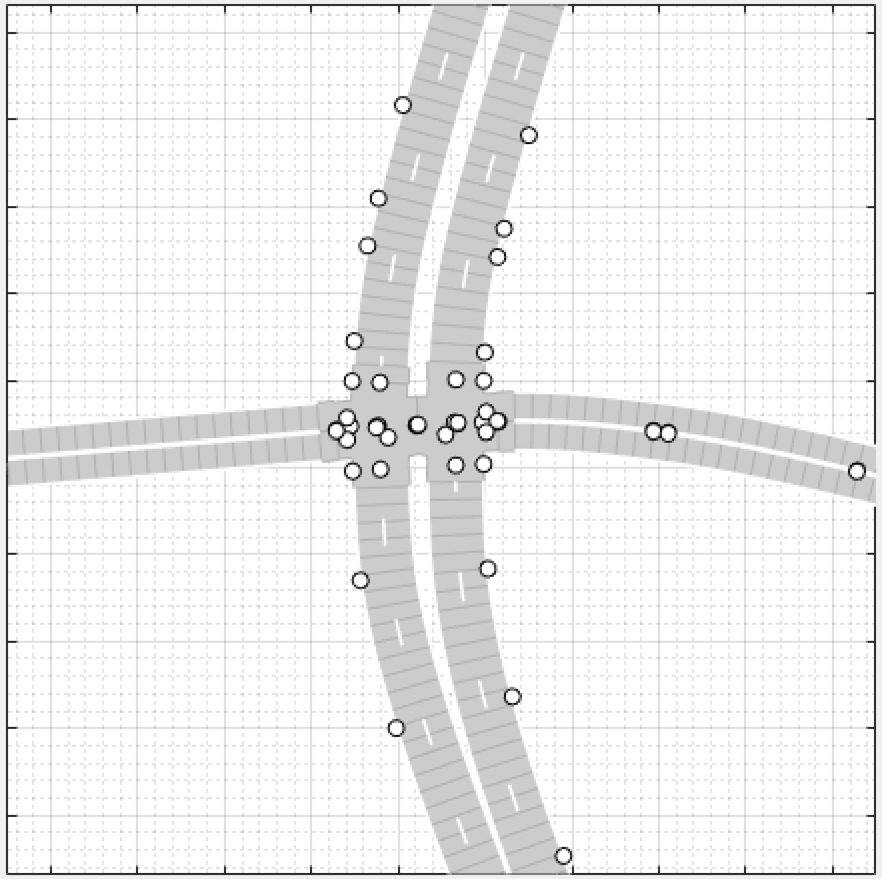
Open the **Driving Scenario Designer** app.

```
drivingScenarioDesigner
```

From the app toolstrip, select **Open** > **OpenDRIVE Road Network**. Then, from your MATLAB root folder, navigate to and open this file:

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

Inspect the road network by zooming in on the scenario.



- “Build a Driving Scenario and Generate Synthetic Detections”
- “Generate Synthetic Detections from a Prebuilt Driving Scenario”
- “Generate Synthetic Detections from a Euro NCAP Scenario”
- “Add OpenDRIVE Roads to Driving Scenario”
- “Automatic Emergency Braking with Sensor Fusion”

Programmatic Use

`drivingScenarioDesigner` opens a blank session of the **Driving Scenario Designer** app.

`drivingScenarioDesigner(sessionFileName)` opens the app and loads the specified MAT-file into the app. This file must be a saved **Driving Scenario Designer** app session. If the 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 driving scenario MAT-files. Before loading a prebuilt scenario, add the folder containing the scenario to the MATLAB path. For an example, see "Generate Detections from Prebuilt Scenario" on page 1-18.

Limitations

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.

OpenDRIVE Limitations

- You can import only lanes and roads. The import of road objects and traffic signals is not supported.
- OpenDRIVE files containing large road networks can take up to several minutes to load. In addition, these road networks can cause slow interactions on the app canvas. Examples of large road networks include ones that model the roads of a city or ones with roads that are thousands of meters long.
- Lanes with variable widths are not supported. The width is set to the highest width found within that lane. For example, if a lane has a width that varies from 2 meters to 4 meters, the app sets the lane width to 4 meters throughout.
- Roads with multiple lane marking styles are not supported. The app applies the first found marking style to all lanes in the road. For example, if a road has `Dashed` and `Solid` lane markings, the app applies `Dashed` lane markings throughout.

- Lane marking styles `Bott Dots`, `Curbs`, and `Grass` are not supported. If imported roads have these lane marking styles, the app sets their lane markings to the default style, as determined by the number of lanes in the road.

Definitions

Road Elevation and Banking Angle

The **Roads** tab provides options for controlling the elevation and banking angle of a road.

When working with roads containing nondefault elevations or banking angles, keep these tips in mind:

- When you add a road center to an elevated road, the default *z*-dimension of the road center is 0. To adjust the elevation of the road center to match the elevation of surrounding road centers, first select the road. Then, on the **Roads** tab, in the **Road Centers** section, adjust the **z (m)** parameter of the road center.
- When you add an actor to a road, you do not have to change the actor position to match changes in elevation angle or banking angle. The actor follows the elevation and banking angle of the road automatically.
- 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 onto the junction, the app might be unable to compute the precise elevation of the actor. Therefore, the app cannot place the actor on that junction.

  To address this issue, 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.

Lane Specifications

The **Roads** tab provides options for changing the number of lanes in a road and specifying its lane markings. You can specify the **Number of lanes** parameter as a:

- Positive integer scalar, *M* — Create an *M*-lane road whose default lane markings indicate that the road is one-way.

- Two-element vector of positive integers, [*M N*] — Create an (*M*+*N*)-lane road whose default lane markings indicate that the road is two-way. The first *M* lanes travel in one direction. The next *N* lanes travel in the opposite direction.

If you change the **Number of lanes** parameter from a scalar to a vector, the default lane markings also change. If the change creates an impossible road configuration, the app resets the **Lane Width (m)** parameter for all lanes to the default of `3.6`. This resetting can occur when the updated road contains lanes with very small widths. For example, if a lane has a width that is less than the width of one of its lane markings, then all lanes are reset to a width of 3.6 meters.

Sample Time

Under **Settings**, the **Sample Time (ms)** parameter controls how frequently the simulation updates. 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, it might take less than 10 seconds for the 10 seconds of simulation time to elapse. Any apparent synchronization between the sample time and actual time is coincidental.

Tips

- 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.
- During simulation, the default camera and radar sensors update every 100 ms (**Update Interval (ms)** = `100`). To ensure that the app samples and displays the detections found at these intervals, the update interval must be an integer multiple of the app sample time. By default, the app samples the simulation every 10 ms (**Sample Time (ms)** = `10`). For more details on the app sample time, see “Sample Time” on page 1-25.

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

Classes
`drivingScenario`

System Objects
`radarDetectionGenerator` | `visionDetectionGenerator`

Topics

"Build a Driving Scenario and Generate Synthetic Detections"
"Generate Synthetic Detections from a Prebuilt Driving Scenario"
"Generate Synthetic Detections from a Euro NCAP Scenario"
"Add OpenDRIVE Roads to Driving Scenario"
"Automatic Emergency Braking with Sensor Fusion"

External Websites

Euro NCAP Safety Assist Protocols
opendrive.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 a video, in an image sequence, or from a custom data source reader. Using the app, you can:

- Define rectangular regions of interest (ROI) labels, polyline ROI labels, pixel ROI labels, and scene labels, and use these labels to interactively label your ground truth data.
- Use built-in detection or tracking algorithms to label your ground truth data.
- Write, import, and use your own custom automation algorithm to automatically label ground truth. See “Create Automation Algorithm for Labeling” (Computer Vision System Toolbox).
- Evaluate the performance of your label automation algorithms using a visual summary. See “View Summary of Ground Truth Labels” (Computer Vision System Toolbox).
- Export the labeled ground truth as a `groundTruth` object. You can use this object for system verification or for training an object detector or semantic segmentation network. See “Train Object Detector or Semantic Segmentation Network from Ground Truth Data” (Computer Vision System Toolbox).
- Display time-synchronized signals, such as lidar or CAN bus data, using the `driving.connector.Connector` API.

To learn more about the app, see Ground Truth Labeler App.

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. `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.

`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(gtSource)` opens the app and loads the `groundTruthDataSource` object, `gtSource`. The object contains a custom data source and corresponding timestamps. See “Use Custom Data Source Reader for Ground Truth Labeling” (Computer Vision System Toolbox).

`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.

`groundTruthLabeler( ___ ,'ConnectorTargetHandle','connector')` opens the app with a custom connector. `'connector'` is a handle to a `driving.connector.Connector` class. The handle implements a custom analysis or visualization tool that is time-synchronized with the **Ground Truth Labeler** app. For example, to associate a connector target defined in class `MyConnectorClass`, specify `@MyConnectorClass`.

For example, open the app, load a 10-second video into it, and open a lidar visualization tool that is time-synchronized to the video.

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

Limitations

- The built-in automation algorithms support the automation of rectangular ROI labels only. When you select a built-in algorithm and click **Automate**, scene labels, pixel labels, polyline labels, sublabels, and attributes are not imported into the automation session. To automate the labeling of these features, create a custom automation algorithm. See “Create Automation Algorithm for Labeling” (Computer Vision System Toolbox).
- Pixel ROI labels do not support sublabels or attributes.
- The Label Summary window does not support sublabels or attributes

Tips

- To avoid having to relabel ground truth with new labels, organize the labeling scheme you want to use before marking your ground truth.

Algorithms

The **Ground Truth Labeler** app provides built-in algorithms that you can use to automate labeling. From the app toolstrip, click **Select Algorithm**, and then select an automation algorithm.

Built-In Automation Algorithm	Description
ACF People Detector	Detect and label people using a pretrained detector based on aggregate channel features (ACF). With this algorithm, you do not need to draw any ROI labels.
Point Tracker	Track and label one or more rectangular ROI labels over short intervals using the Kanade-Lucas-Tomasi (KLT) algorithm.
Temporal Interpolator	Estimate ROIs in intermediate frames using the interpolation of rectangular ROIs in key frames. Draw ROIs on a minimum of two frames (at the beginning and at the end of the interval). The interpolation algorithm estimates the ROIs between the frames.
ACF Vehicle Detector	Detect and label vehicles using a pretrained detector based on ACF. With this algorithm, you do not need to draw any ROI labels.

See Also

Apps
Image Labeler | **Video Labeler**

Functions
`objectDetectorTrainingData` | `pixelLabelTrainingData`

Objects
`groundTruth` | `groundTruthDataSource` | `labelDefinitionCreator`

Topics
"Get Started with the Ground Truth Labeler"

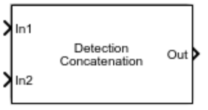
“Automate Ground Truth Labeling of Lane Boundaries”
“Automate Ground Truth Labeling for Semantic Segmentation”
“Automate Attributes of Labeled Objects”
“Evaluate Lane Boundary Detections Against Ground Truth Data”
“Evaluate and Visualize Lane Boundary Detections Against Ground Truth”
“Choose a Labeling App” (Computer Vision System Toolbox)
“Use Custom Data Source Reader for Ground Truth Labeling” (Computer Vision System Toolbox)
“Use Sublabels and Attributes to Label Ground Truth Data” (Computer Vision System Toolbox)
“Label Pixels for Semantic Segmentation” (Computer Vision System Toolbox)
“Create Automation Algorithm for Labeling” (Computer Vision System Toolbox)
“Share and Store Labeled Ground Truth Data” (Computer Vision System Toolbox)
“Train Object Detector or Semantic Segmentation Network from Ground Truth Data” (Computer Vision System Toolbox)

Introduced in R2017a

Blocks in Automated Driving System Toolbox — Alphabetical List

Detection Concatenation

Combine detection reports from different sensors
Library: Automated Driving System Toolbox



Description

The Detection Concatenation block combines detection reports from multiple sensor blocks onto a single output bus. Sensor blocks include the Radar Detection Generator and the Vision Detection Generator blocks. Concatenation is useful when detections from multiple sensor blocks are passed into a Multiobject 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 — Sensor detections via first input port
structure input via Simulink bus

Detection list, specified as a structure input via a Simulink bus. See “Getting Started with 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.

In2 — Sensor detections via second input port
structure input via Simulink bus

Detection list, specified as a structure input via a Simulink bus. See “Getting Started with 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.

`InN` — Sensor detections via N^{th} input port
structure input via Simulink bus

Detection list, specified as a structure input via a Simulink bus. See “Getting Started with 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.

Output

`Out` — Concatenated sensor detections
structure output via Simulink bus

Concatenated sensor detections from all input buses, output as a structure via a Simulink bus. See “Getting Started with 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 detection ports, specified as a positive integer. Each input port is labelled **In1**, **In2**, ... **InN** where *N* is the value set by this parameter.

Example: 5

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 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

character string

Name of output bus, specified as a character string.

Example: `visionbus`

Dependencies

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

`Simulate using` — Block simulation method

`Interpreted Execution` (default) | `Code Generation`

Block simulation, specified as `Interpreted Execution` or `Code Generation`. If you want your block to use the MATLAB interpreter, choose `Interpreted Execution`. If you want your block to run as compiled code, choose `Code Generation`. Compiled code requires time to compile but usually runs faster.

Interpreted execution is useful when you are developing and tuning a model. The block runs the underlying System object™ in MATLAB. You can change and execute your model quickly. When you are satisfied with your results, you can then run the block using `Code Generation`. Long simulations run faster than in interpreted execution. You can run repeated executions without recompiling. However, if you change any block parameters, then the block automatically recompiles before execution.

When setting this parameter, you must take into account the overall model simulation mode. The table shows how the **Simulate using** parameter interacts with the overall simulation mode.

When the Simulink model is in `Accelerator` mode, the block mode specified using **Simulate using** overrides the simulation mode.

Acceleration Modes

Block Simulation	Simulation Behavior		
	`Normal`	`Accelerator`	`Rapid Accelerator`
`Interpreted Execution`	The block executes using the MATLAB interpreter.	The block executes using the MATLAB interpreter.	Creates a standalone executable from the model.
`Code Generation`	The block is compiled.	All blocks in the model are compiled.	

For more information, see “Choosing a Simulation Mode” (Simulink) from the Simulink documentation.

See Also

Bird's-Eye Scope | Multiobject Tracker | Radar Detection Generator | Vision Detection Generator

Topics

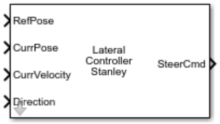
“Getting Started with Buses” (Simulink)

Introduced in R2017b

Lateral Controller Stanley

Compute steering angle command for path following using Stanley method

Library: Automated Driving System Toolbox / Vehicle Controller



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 a kinematic bicycle model. Use this controller for path following in low-speed environments, where inertial effects are minimal.

Ports

Input

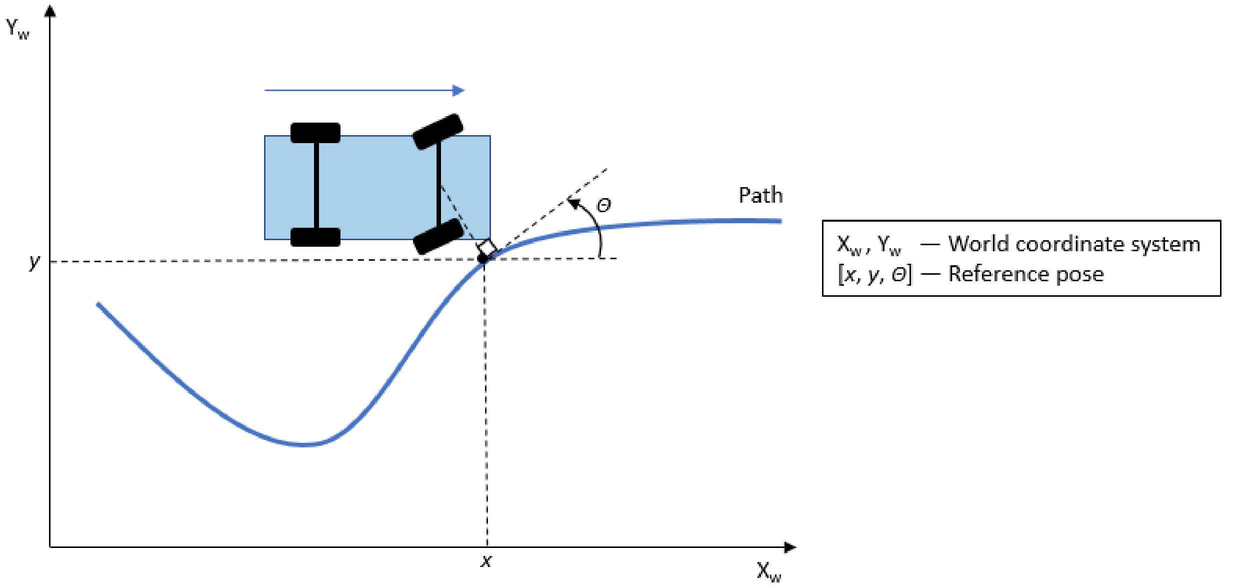
`RefPose` — Reference pose

[x, y, Θ] vector

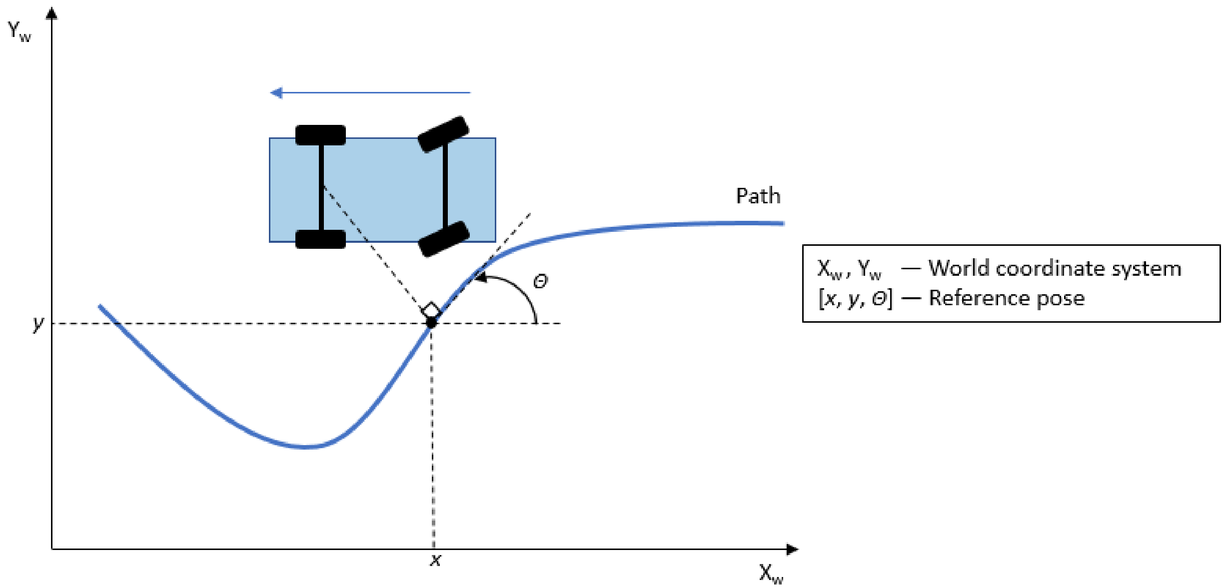
Reference pose, specified as an [x, y, Θ] vector. x and y are in meters, and Θ is in degrees.

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.



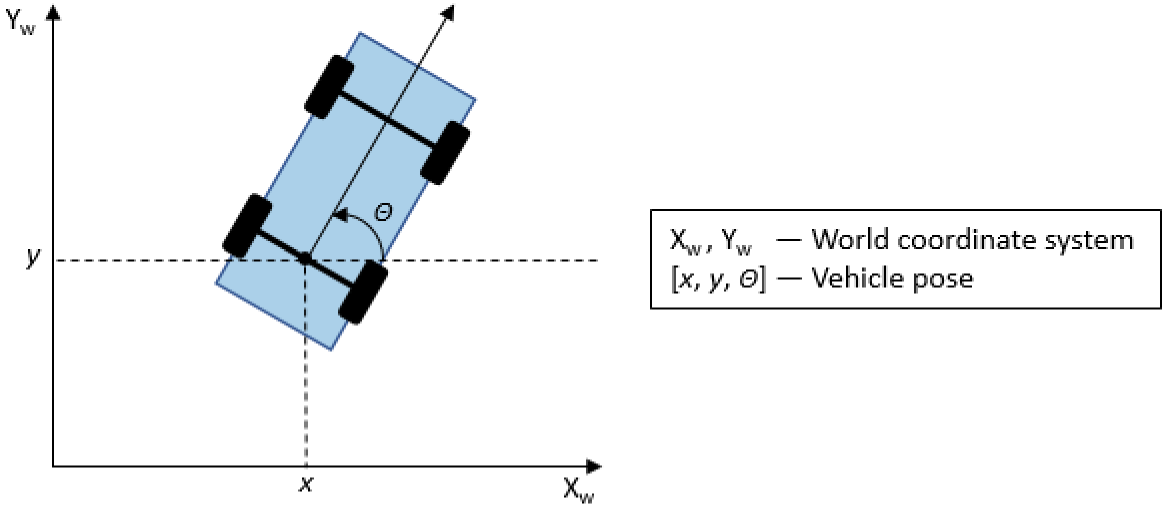
Data Types: `single` | `double`

`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 System Toolbox”.

Data Types: `single` | `double`

`CurrVelocity` — Current longitudinal velocity
scalar

Current longitudinal velocity of the vehicle, specified as a 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`

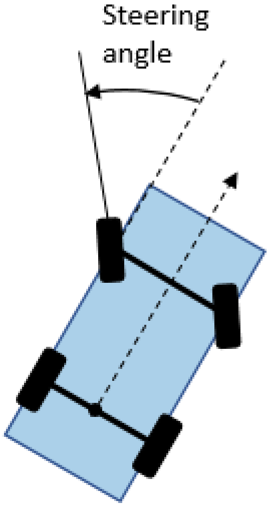
`Direction` — Driving direction of vehicle
`1` (forward motion) | `-1` (reverse motion)

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 2-11.

Output

`SteerCmd` — Steering angle command
scalar

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



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

Parameters

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

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

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

`Maximum steering angle (deg)` — Maximum allowed steering angle
`35` (default) | scalar in the range (0, 180)

Maximum allowed steering angle of the vehicle, in degrees, specified as a 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*.

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, 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

See Also

Functions
`lateralControllerStanley`

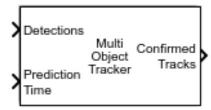
Objects
`pathPlannerRRT`

Topics
"Coordinate Systems in Automated Driving System Toolbox"

Introduced in R2018b

Multi Object Tracker

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



Description

The Multi Object Tracker block creates and manages the tracks of moving objects. The block initializes, confirms, predicts, corrects, and deletes tracks. Inputs to the tracker are detection reports generated by the Radar Detection Generator and Vision Detection Generator blocks. The tracker accepts detections from multiple sensors. Detections are assigned to tracks using a global nearest neighbor (GNN) criterion. A detection is assigned to only one track and when no assignment is possible, the tracker creates a new track.

A new track usually starts in a 'Tentative' state. If enough detections are assigned to the track, its status shifts to 'Confirmed'. When a track is confirmed, you have confidence that it represents a real object. If detections are not added to the track within a specifiable number of updates, the track can be deleted. The tracker also optimally estimates the state vector and state vector covariance matrix for each track using a Kalman filter.

Ports

Input

`Detections` — Detection list
structure input via Simulink bus

Detection list, specified as a structure input via a Simulink bus. See "Getting Started with 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
Detection structures		array of object detection structures. The first `NumDetections` of these 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 and greater than the update time specified in the previous invocation of the block.

`Prediction Time` — Track update time

scalar

Track update time, specified as a scalar. The 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 and greater than the update time in the previous invocation of the block.

Example: `6.5`

Dependencies

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

`Cost Matrix` — Generic input port

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 `alltracks` output argument of the previous call to `updateTracks`. For the first call to `updateTracks` or if there are no previous tracks, assign the cost matrix a size of `[0,Nd]`. Note that the cost must be calculated so that lower costs indicate a higher likelihood of assigning a detection to a track. You can use `Inf` to prevent some detections being assigned to certain tracks.

Dependencies

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

Output

`Confirmed Tracks` — Confirmed tracks

structure output via Simulink bus

Confirmed tracks, output as structure via a Simulink bus (see “Getting Started with Buses” (Simulink)). The fields of the structure are:

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

The track structure is defined as:

Field	Definition
`TrackID`	Unique track identifier.
`Time`	Time at which the track is updated. Units are in seconds.
`Age`	Number of updates since track initialization.
`State`	Updated state vector. The state vector is specific to each type of Kalman filter.

Field	Definition
StateCovariance	Updated state covariance matrix. The covariance matrix is specific to each type of Kalman filter.
IsConfirmed	Confirmation status. Set to true if the track is confirmed to be a real target.
IsCoasted	Coasting status – true if the track has been updated without a new detection.
ObjectClassID	Integer value representing the object classification. The value 0 represents a classification of unknown. Nonzero classifications apply only to confirmed tracks.
ObjectAttributes	Cell array of object attributes reported by the sensor making the detection.

A track is confirmed if:

- The track passes the *M*-out-of-*N* test specified by 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
structure output via Simulink bus

Tentative tracks, output as a structure via Simulink bus (see "Getting Started with Buses" (Simulink)). A track is tentative before it is confirmed.

This structure is the same as defined in the **Confirmed Tracks** port.

Dependencies

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

All Tracks — All tracks
structure output via Simulink bus

Combined list of confirmed and tentative tracks, output as a structure via Simulink bus (see "Getting Started with Buses" (Simulink)).

This structure is the same as defined in the **Confirmed Tracks** port.

Dependencies

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

Parameters

Tracker Management

`Filter initialization function name` — Function to initialize tracking filter
`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 scalar

Detection assignment threshold, specified as a positive 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 they should be 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) | 2-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`.

As a guide to setting `N`, consider the number of times you want the tracker to update before a confirmation decision must be made. For example, if a tracker updates every *.05* seconds, and you allow *.5* seconds to make a confirmation decision, set `N = 10`. To set `M`, take into account the probability that the sensors will detect objects. The probability of detection depends on factors such as occlusion or clutter. You can reduce the value of `M` when tracks fail to be confirmed or increase `M` when too many false detections get formed into tracks.

Example: `[3,5]`

`Number of times a confirmed track is coasted` — Coasting threshold for track deletion
`5` (default) | positive integer

Coasting threshold for track deletion, specified as a positive integer. A track *coasts* when no detections are assigned to the track after one or more `predict` steps. If the number of coasting steps exceeds this threshold, the track is deleted.

Example: `12`

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

Maximum number of tracks 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 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.

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 allow update time input using the `Prediction time` input port. Otherwise, the update time is automatically determined by the simulation clock managed by Simulink.

Example: `Auto`

`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

character string

Name of output bus, specified as a character string.

Example: `tracksbus`

Dependencies

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

`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 using the **Cost matrix** input port.

`Enable tentative tracks output` — Enable output port for tentative tracks

off (default) | on

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

`Simulate using` — Block simulation method

`Interpreted Execution` (default) | `Code Generation`

Block simulation, specified as `Interpreted Execution` or `Code Generation`. If you want your block to use the MATLAB interpreter, choose `Interpreted Execution`. If you want your block to run as compiled code, choose `Code Generation`. Compiled code requires time to compile but usually runs faster.

Interpreted execution is useful when you are developing and tuning a model. The block runs the underlying System object in MATLAB. You can change and execute your model quickly. When you are satisfied with your results, you can then run the block using `Code Generation`. Long simulations run faster than in interpreted execution. You can run repeated executions without recompiling. However, if you change any block parameters, then the block automatically recompiles before execution.

When setting this parameter, you must take into account the overall model simulation mode. The table shows how the **Simulate using** parameter interacts with the overall simulation mode.

When the Simulink model is in `Accelerator` mode, the block mode specified using **Simulate using** overrides the simulation mode.

Acceleration Modes

Block Simulation	Simulation Behavior		
	`Normal`	`Accelerator`	`Rapid Accelerator`
`Interpreted Execution`	The block executes using the MATLAB interpreter.	The block executes using the MATLAB interpreter.	Creates a standalone executable from the model.
`Code Generation`	The block is compiled.	All blocks in the model are compiled.	

For more information, see “Choosing a Simulation Mode” (Simulink) from the Simulink documentation.

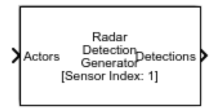
See Also

Bird's-Eye Scope | `multiObjectTracker`

Introduced in R2017b

Radar Detection Generator

Create detection objects from radar measurements
Library: Automated Driving System Toolbox



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. All 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 Multiobject Tracker block.

Ports

Input

`Actors` — Scenario actor poses
structure input via Simulink bus

Scenario actor poses, specified as a structure input via Simulink bus.

The structure has the form:

Field	Description	Type
`NumActors`	Number of actors	integer

Field	Description	Type
`Time`	False when updates are requested at times between block invocation intervals.	double scalar
Actor poses structures		Array length `NumActors` of actor poses structures

The actor poses structure is defined as:

Field	Description
`ActorID`	Unique actor identifier, specified as a scalar positive integer.
`Position`	Actor position vector, specified as real-valued 1-by-3 vector. Units are in meters.
`Velocity`	Actor velocity vector, specified as real-valued 1-by-3 vector. If velocity is not specified, the default value is `[0 0 0]`. Units are in meters per second.
`Speed`	Speed of actor, specified as a real scalar. When specified, the actor velocity is aligned with the *x*-axis of the actor in the ego actor coordinate system. You cannot specify both `Speed` and `Velocity`. The default value is `0`. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. If roll is not specified, the default value is `0`. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. If pitch is not specified, the default value is `0`. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. If yaw is not specified, the default value is `0`. Units are in degrees.

- You cannot specify both `Velocity` and `Speed` simultaneously.

- The values of `Position`, `Velocity`, `Speed`, `Roll`, `Pitch`, and `Yaw` are defined with respect to the ego coordinate system.
- See `Actor` and `Vehicle` for more precise definitions of the structure fields.

You can also specify this structure manually. You can omit many fields but you must include `ActorID` and `Position`. All others will take default values.

Output

`Detections` — Detection list

structure output via Simulink bus

Radar sensor detections, output as structure via a Simulink bus. See "Getting Started with 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
Detection structures		array of object detection structures of length set by the **Maximum number of reported detections** parameter. Only `NumDetections` of these 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

Property	Definition
`ObjectClassID`	Object classification
`MeasurementParameters`	Parameters used by initialization functions of nonlinear Kalman tracking filters
`ObjectAttributes`	Additional information passed to tracker

- 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'
'Sensor Cartesian'</td><td>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 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 multi-sensor system.

Example: 5

`Required interval between sensor updates (s)` — Required time interval

`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.

Parameters - 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 scalar

Radar sensor height above the ground plane, specified as a positive 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) | scalar

Yaw angle of radar sensor, specified as a 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) | scalar

Pitch angle of sensor, specified as a 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) | scalar

Roll angle of the radar sensor, specified as a 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.

Parameters - 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
character string

Name of output bus, specified as a character string.

Example: `radarbus`

Dependencies

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

Parameters - 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.

Example: `Sensor spherical`

`Simulate using` — Block simulation method

`Interpreted Execution` (default) | `Code Generation`

Block simulation, specified as `Interpreted Execution` or `Code Generation`. If you want your block to use the MATLAB interpreter, choose `Interpreted Execution`. If you want your block to run as compiled code, choose `Code Generation`. Compiled code requires time to compile but usually runs faster.

Interpreted execution is useful when you are developing and tuning a model. The block runs the underlying System object in MATLAB. You can change and execute your model quickly. When you are satisfied with your results, you can then run the block using `Code Generation`. Long simulations run faster than in interpreted execution. You can run repeated executions without recompiling. However, if you change any block parameters, then the block automatically recompiles before execution.

When setting this parameter, you must take into account the overall model simulation mode. The table shows how the **Simulate using** parameter interacts with the overall simulation mode.

When the Simulink model is in `Accelerator` mode, the block mode specified using **Simulate using** overrides the simulation mode.

Acceleration Modes

Block Simulation	Simulation Behavior		
	`Normal`	`Accelerator`	`Rapid Accelerator`
`Interpreted Execution`	The block executes using the MATLAB interpreter.	The block executes using the MATLAB interpreter.	Creates a standalone executable from the model.
`Code Generation`	The block is compiled.	All blocks in the model are compiled.	

For more information, see “Choosing a Simulation Mode” (Simulink) from the Simulink documentation.

Measurements - Accuracy Settings

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

Azimuth resolution of the radar, specified as a positive 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 scalar

Elevation resolution of the radar, specified as a positive 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 scalar

Range resolution of the radar, specified as a positive 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 scalar

Range rate resolution of the radar, specified as a positive 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.

Measurements - Bias Settings

`Fractional azimuthal bias component of radar` — Azimuth bias fraction

`0.1` (default) | nonnegative scalar

Azimuth bias fraction of the radar, specified as a nonnegative 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 scalar

Elevation bias fraction of the radar, specified as a nonnegative 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 scalar

Range bias fraction of the radar, specified as a nonnegative 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 scalar

Range rate bias fraction of the radar, specified as a nonnegative 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.

Measurements - 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 scalar

Maximum detection range, specified as a positive 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 scalar less than or equal to 1

Probability of detecting a target, specified as a positive 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 scalar

False alarm rate within a radar resolution cell, specified as a positive 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 scalar

Reference range for a given probability of detection, specified as a positive 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 scalar

Reference radar cross-section (RCS) for given probability of detection, specified as a nonnegative 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`

Measurements - 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` | `Nonrepeatable`

Method to set the random number generator seed, specified as `Repeatable`, `Specify seed`, or `Nonrepeatable`. When set to `Specify seed`, the value set in the `InitialSeed` parameter is used. When set to `Repeatable`, a random initial seed is generated for the first simulation and then reused for all subsequent simulations. You can, however, change the seed by issuing a `clear all` command. When set to `Nonrepeatable`, a new initial seed is generated each time the simulation runs.

Example: `Specify seed`

`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

MATLAB expression for actor profiles, specified as a MATLAB structure or MATLAB structure array.

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 via 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 scalar | length-*L* vector of positive values

Length of cuboid, specified as a positive 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 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 scalar | length-*L* vector of positive values

Width of cuboid, specified as a positive 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 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 scalar | length-*L* vector of positive values

Height of cuboid, specified as a positive 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 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`.

See Also

Bird's-Eye Scope | Detection Concatenation | Multiobject Tracker | Vision Detection Generator | `radarDetectionGenerator`

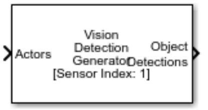
Topics

"Getting Started with Buses" (Simulink)

Introduced in R2017b

Vision Detection Generator

Detect objects and lanes from visual measurements
Library: Automated Driving System Toolbox



Description

The Vision Detection Generator block generates detections from camera measurements taken by a vision sensor mounted on an ego vehicle. Detections are derived from simulated actor poses and are generated at intervals equal to the sensor update interval. All 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 positive 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 Vision Detection Generator to create input to a Multiobject Tracker block.

Ports

Input

`Actors` — Scenario actor poses
structure input via Simulink bus

Scenario actor poses, specified as a structure input via Simulink bus. You can also create this structure manually.

The structure has the form:

Field	Description	Type
`NumActors`	Number of actors	integer

Field	Description	Type
`Time`	False when updates are requested at times between block invocation intervals.	double scalar
Actor pose structures		Array length `NumActors` of actor pose structures

The actor pose structure is defined as:

Field	Description
`ActorID`	Unique actor identifier, specified as a scalar positive integer.
`Position`	Actor position vector, specified as real-valued 1-by-3 vector. Units are in meters.
`Velocity`	Actor velocity vector, specified as real-valued 1-by-3 vector. If velocity is not specified, the default value is `[0 0 0]`. Units are in meters per second.
`Speed`	Speed of actor, specified as a real scalar. When specified, the actor velocity is aligned with the *x*-axis of the actor in the ego actor coordinate system. You cannot specify both `Speed` and `Velocity`. The default value is `0`. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a real-valued scalar. If roll is not specified, the default value is `0`. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a real-valued scalar. If pitch is not specified, the default value is `0`. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a real-valued scalar. If yaw is not specified, the default value is `0`. Units are in degrees.

- You cannot specify both `Velocity` and `Speed` simultaneously.

- The values of `Position`, `Velocity`, `Speed`, `Roll`, `Pitch`, and `Yaw` are defined with respect to the ego coordinate system.
- See `Actor` and `Vehicle` for more precise definitions of the structure fields.

You can also specify this structure manually. You can omit many fields but you must include `ActorID` and `Position`. All other fields have default values.

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

array of lane boundary structures

Lane boundaries, specified as an array of lane boundary structures defined in the table:

Lane Boundary Structure Fields

Field	Description
`Coordinates`	Lane boundary coordinates, specified as a real-valued *N*-by-3 matrix. Lane boundary coordinates define the position of points on the boundary at distances specified by `XDistance`. In addition, a set of boundary coordinates are inserted into the matrix at zero distance. 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 rows in the `Coordinates` matrix. Units are in degrees/m.
`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 rows in the `Coordinates` matrix. Units are in degrees/m. Units are in degrees/m^2.
`HeadingAngle`	Initial lane boundary heading, specified as a scalar. The heading angle of the lane boundary is relative to the ego car heading. Units are in degrees.
`LateralOffset`	Distance of the lane boundary from the ego vehicle position, specified as a scalar. An offset to a lane boundary to the left of the ego 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 the following: • '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	Strength of the lane boundary marking, specified as a scalar from 0 through 1. A value of 0 corresponds to a marking that is not visible and a value of 1 corresponds to a marking that is completely visible. Values in between are partially visible.
Width	Lane boundary width, specified as a positive scalar. In a double-line lane marker, the same width is used for both lines and the space between lines. Units are in meters.
Length	Length of dash in dashed lines, specified as a positive 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 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` — Detection list
structure output via Simulink bus

Vision sensor detections, output as structure via a Simulink bus. See “Getting Started with 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
Detection structures		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

Property	Definition
`MeasurementParameters`	Parameters used by initialization functions of nonlinear Kalman tracking filters
`ObjectAttributes`	Additional information passed to tracker

- 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. `MeasurementParameters` are reported in sensor Cartesian coordinates.

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. `Frame` is always set to `'rectangular'`, because the Vision Detection Generator reports detections in Cartesian coordinates.
`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)** properties specified in the Vision Detection Generator.
`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 Vision Detection Generator.
`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.
`SNR`	Signal-to-noise ratio of the detection. Units are in dB.

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

array of lane boundary detection structures

Lane boundary detections, returned as an array of lane boundary detection 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.

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

Main Tab

`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 multi-sensor system.

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 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.

`Required interval between lane detections updates (s)` — Time interval between lane detection updates
0.1 (default) | positive scalar

Required time interval between lane detection updates, specified as a positive 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.

Parameters - 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 a sedan dashboard. Units are in meters.

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

Vision sensor height above the ground plane, specified as a positive 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) | scalar

Yaw angle of vision sensor, specified as a 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) | scalar

Pitch angle of sensor, specified as a 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) | scalar

Roll angle of the vision sensor, specified as a 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.

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

Example: `Property`

`Source of output lane bus name` — Source of object 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
character string

Object bus name, specified as a character string.

Example: `visionbus`

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 name
character string

output lane bus name, specified as a character string.

Example: `lanebus`

Dependencies

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

Parameters - 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 detections
`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` | `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.

Example: `Sensor Cartesian`

`Simulate using` — Block simulation method
`Interpreted Execution` (default) | `Code Generation`

Block simulation, specified as `Interpreted Execution` or `Code Generation`. If you want your block to use the MATLAB interpreter, choose `Interpreted Execution`. If you want your block to run as compiled code, choose `Code Generation`. Compiled code requires time to compile but usually runs faster.

Interpreted execution is useful when you are developing and tuning a model. The block runs the underlying System object in MATLAB. You can change and execute your model quickly. When you are satisfied with your results, you can then run the block using `Code Generation`. Long simulations run faster than in interpreted execution. You can run repeated executions without recompiling. However, if you change any block parameters, then the block automatically recompiles before execution.

When setting this parameter, you must take into account the overall model simulation mode. The table shows how the **Simulate using** parameter interacts with the overall simulation mode.

When the Simulink model is in `Accelerator` mode, the block mode specified using **Simulate using** overrides the simulation mode.

Acceleration Modes

<table>
<tr><th rowspan="2">Block Simulation</th><th colspan="3">Simulation Behavior</th></tr>
<tr><th><code>Normal</code></th><th><code>Accelerator</code></th><th><code>Rapid Accelerator</code></th></tr>
<tr><td><code>Interpreted Execution</code></td><td>The block executes using the MATLAB interpreter.</td><td>The block executes using the MATLAB interpreter.</td><td rowspan="2">Creates a standalone executable from the model.</td></tr>
<tr><td><code>Code Generation</code></td><td>The block is compiled.</td><td>All blocks in the model are compiled.</td></tr>
</table>

For more information, see “Choosing a Simulation Mode” (Simulink) from the Simulink documentation.

Measurements - Settings

`Maximum detection range (m)` — Maximum detection range

`150` (default) | positive scalar

Maximum detection range, specified as a positive scalar. The vision sensor cannot detect objects beyond this range. Units are in meters.

Example: `250`

Measurements - Object Detector Settings

`Bounding box accuracy (pixels)` — Bounding box accuracy

`5` (default) | positive scalar

Bounding box accuracy, specified as a positive 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/s²)` — Noise intensity used for filtering position and velocity measurements
`5` (default) | positive scalar

Noise intensity used for filtering position and velocity measurements, specified as a positive 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`

`Maximum detectable object speed (m/s)` — Maximum detectable object speed
`50` (default) | positive scalar

Maximum detectable object speed, specified as a non-negative scalar. Units are in meters per second.

Example: `20`

`Maximum allowed occlusion for detector` — Maximum detectable object speed
`0.5` (default) | scalar in the range [0 1)

Maximum allowed occlusion of an object, specified as a 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`

`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 scalar less than or equal to 1

Probability of detecting a target, specified as a positive 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 the vision sensor per image
`0.1` (default) | nonnegative scalar

Number of false detections generated by the vision sensor per image, specified as a nonnegative scalar.

Example: `1.0`

Measurements - Lane Detector Settings

`Minimum lane size in image (pixels)` — Maximum size of lane
`[20 5]` (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 scalar

Accuracy of lane boundaries, specified as a positive 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.

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 check box 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` | `Nonrepeatable`

Method to set the random number generator seed, specified as `Repeatable`, `Specify seed`, or `Nonrepeatable`. When set to `Specify seed`, the value set in the `InitialSeed` parameter is used. When set to `Repeatable`, a random initial seed is generated for the first simulation and then reused for all subsequent simulations. You can, however, change the seed by issuing a `clear all` command. When set to `Nonrepeatable`, a new initial seed is generated each time the simulation runs.

Example: `Specify seed`

`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

MATLAB expression for actor profiles, specified as a MATLAB structure or MATLAB structure array.

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 via 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 scalar | length-*L* vector of positive values

Length of cuboid, specified as a positive 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 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 scalar | length-*L* vector of positive values

Width of cuboid, specified as a positive 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 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 scalar | length-*L* vector of positive values

Height of cuboid, specified as a positive 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 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) | real-valued 1-by-2 vector of positive integers

Camera focal length, specified as a real-valued 1-by-2 vector of positive integers. Units are in pixels. See `cameraIntrinsics`.

Example: `[480,320]`

`Optical center of the camera (pixels)` — Optical center of the camera

`[320,240]` (default) | real-valued 1-by-2 vector of positive integers

Optical center of the camera, specified as a real-valued 1-by-2 vector of positive integers. Units are in pixels. See `cameraIntrinsics`.

Example: `[480,320]`

`Image size produced by the camera (pixels)` — Image size produced by the camera
`[480,640]` (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. Units are in pixels. See `cameraIntrinsics`.

Example: `[240,320]`

`Radial distortion coefficients` — Radial distortion coefficients
`[0,0]` (default) | real-valued 1-by-2 matrix of nonnegative values

Radial distortion coefficients, specified as a real-valued 1-by-2 matrix of nonnegative values. See `cameraIntrinsics`.

Example: `[1,1]`

`Tangential distortion coefficients` — Tangential distortion coefficients
`[0,0]` (default) | real-valued 1-by-2 matrix of nonnegative values

Tangential distortion coefficients, specified as a real-valued 1-by-2 matrix of nonnegative values. See `cameraIntrinsics`.

Example: `[1,1]`

`Skew of the camera axes` — Skew of the camera axes
`0` (default) | nonnegative scalar

Skew of the camera axes, specified as a nonnegative scalar. See `cameraIntrinsics`

Example: `0.1`

See Also

Bird's-Eye Scope | Detection Concatenation | Multiobject Tracker | Radar Detection Generator | `cameraIntrinsics` | `visionDetectionGenerator`

Topics

"Getting Started with Buses" (Simulink)

Introduced in R2017b

3

Functions in Automated Driving System Toolbox

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 frame
`'rectangular'` (default) | `'spherical'`

Measurement frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of the *x*, *y*, and *z* Cartesian coordinates of the tracked object. When specified as `'spherical'`, a measurement consists of the azimuth, elevation, range, and range rate of the tracked object.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the global coordinate system, 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 global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters/second.

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 global coordinate system.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure

Measurement parameters, specified as a structure. The fields of the structure are:

measurementParameters struct

Parameter	Definition	Default
`OriginPosition`	Sensor position with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters.	`[0;0;0]`
`OriginVelocity`	Sensor velocity with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in m/s.	`[0;0;0]`
`Orientation`	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 global coordinate system.	`eye(3)`
`HasVelocity`	Indicates whether measurements contain velocity or range rate components, specified as `true` or `false`.	`false` when `frame` argument is `'rectangular'` and `true` when `frame` argument is `'spherical'`
`HasElevation`	Indicates whether measurements contain elevation components, specified as `true` or `false`.	`true`

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 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>[az;r]</td><td>[az;el;r]</td></tr>
<tr><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>
</table>

<table>
<tr><th>frame</th><th>measurement</th></tr>
<tr><td>'rectangular</td><td>Specifies the Cartesian position and velocity coordinates of the tracked object with respect to the ego coordinate system.

Rectangular measurements
<table><tr><td rowspan="2">HasVelocity</td><td>false</td><td>[x;y;y]</td></tr><tr><td>true</td><td>[x;vx;y,vy;z;vz]</td></tr></table>
Position units are in meters and velocity units are in m/s.</td></tr>
</table>

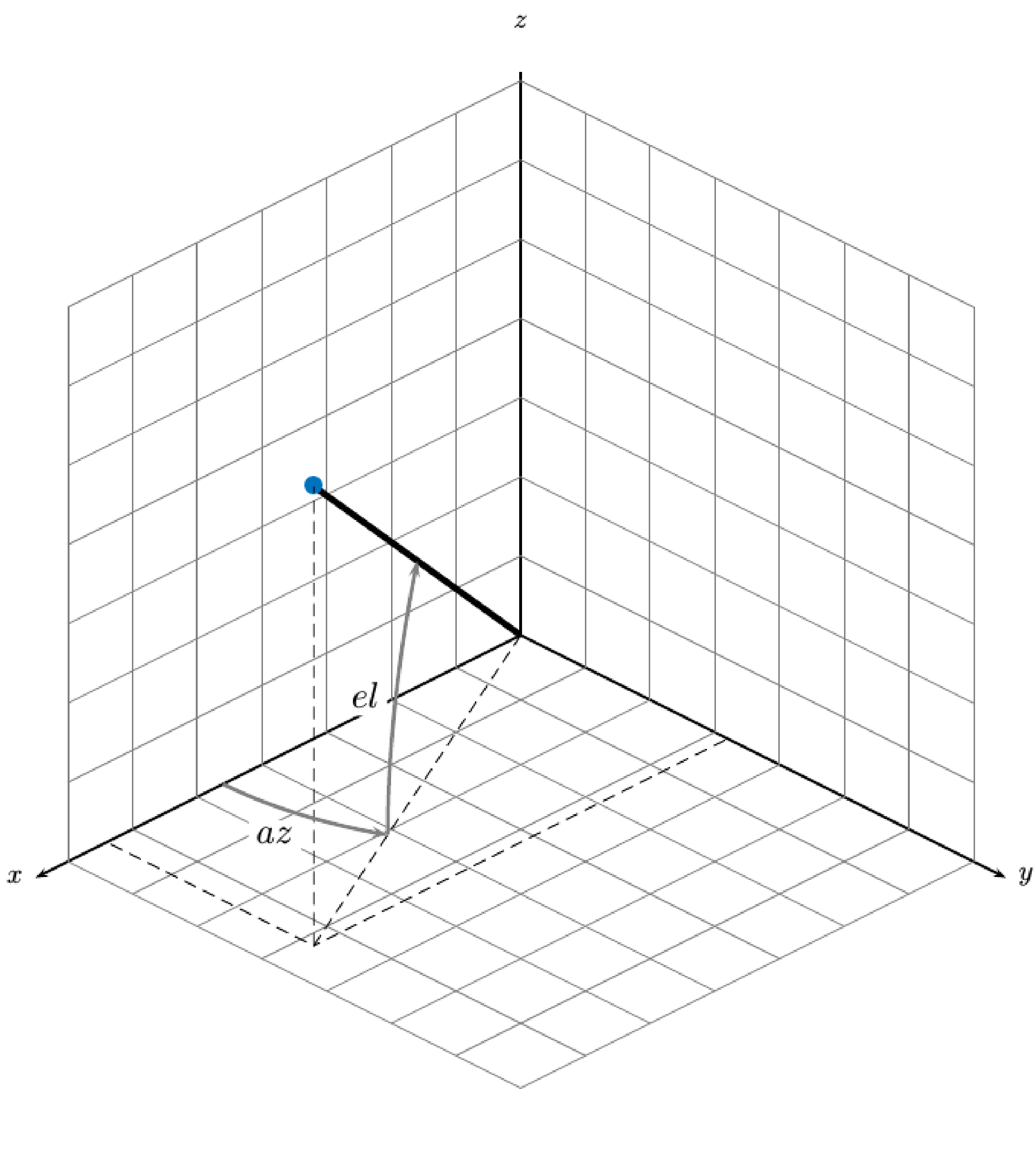
Data Types: double

Definitions

Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Automated Driving System 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.


z
el
az
x
y

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

Classes
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 frame

`'rectangular'` (default) | `'spherical'`

Measurement frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of the *x*, *y*, and *z* Cartesian coordinates of the tracked object. When specified as `'spherical'`, a measurement consists of the azimuth, elevation, range, and range rate of the tracked object.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the global coordinate system, 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 global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters/second.

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 global coordinate system.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure

Measurement parameters, specified as a structure. The fields of the structure are:

measurementParameters struct

Parameter	Definition	Default
`OriginPosition`	Sensor position with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters.	`[0;0;0]`
`OriginVelocity`	Sensor velocity with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in m/s.	`[0;0;0]`
`Orientation`	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 global coordinate system.	`eye(3)`
`HasVelocity`	Indicates whether measurements contain velocity or range rate components, specified as `true` or `false`.	`false` when `frame` argument is `'rectangular'` and `true` when `frame` argument is `'spherical'`
`HasElevation`	Indicates whether measurements contain elevation components, specified as `true` or `false`.	`true`

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.

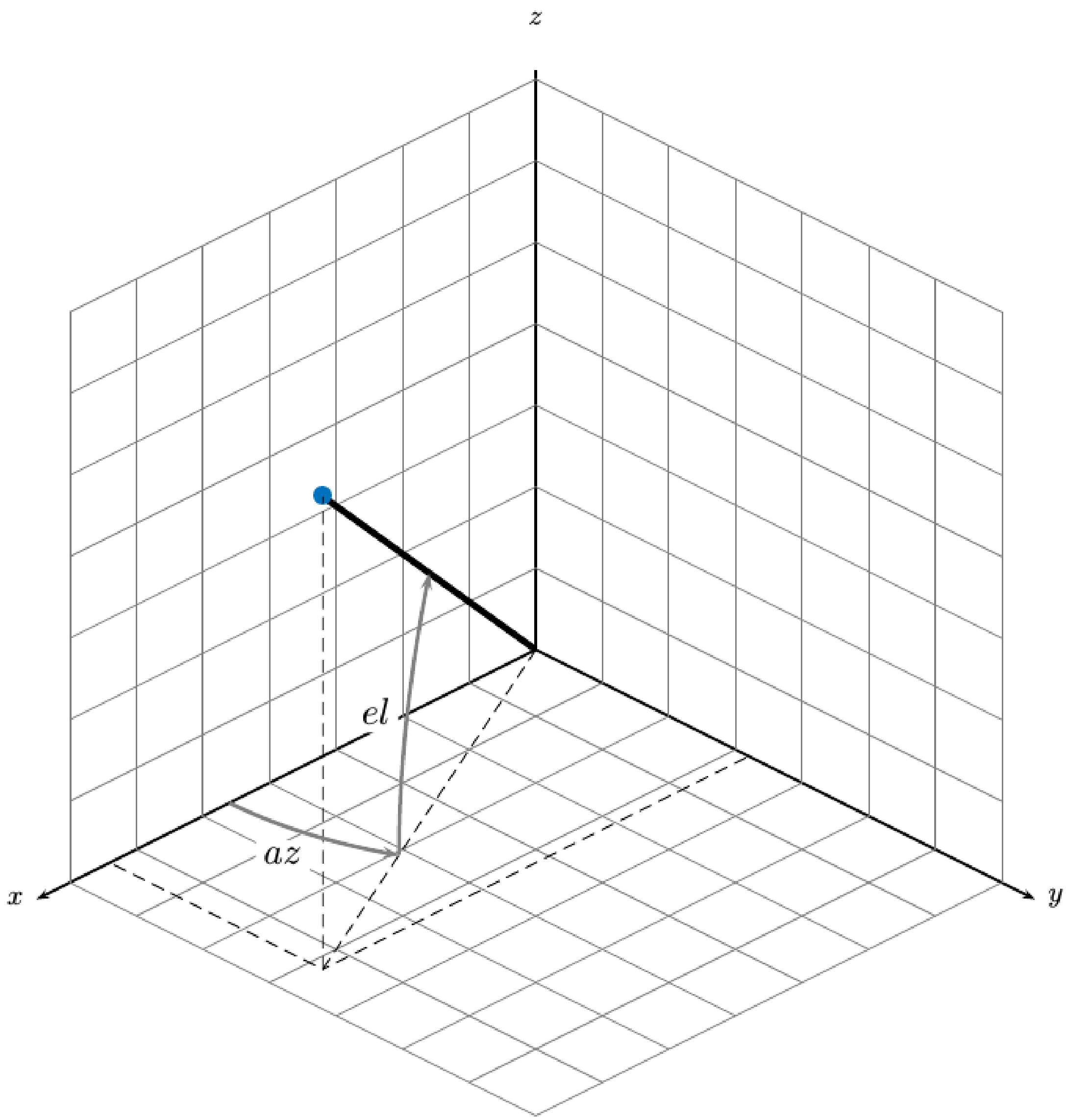
Frame	Measurement Jacobian
`'rectangular'`	Jacobian of the measurements `[x;y;z]` with respect to the state vector. The measurement vector is with respect to the local coordinate system. Coordinates are in meters.
`'spherical'`	Jacobian of the measurement vector `[az;el;r;rr]` with respect to the state 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.

Definitions

Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Automated Driving System 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.


z
el
az
x
y

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`

Classes
`trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

checkPathValidity

Check validity of planned vehicle path

Syntax

```
isValid = checkPathValidity(refPath,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`.

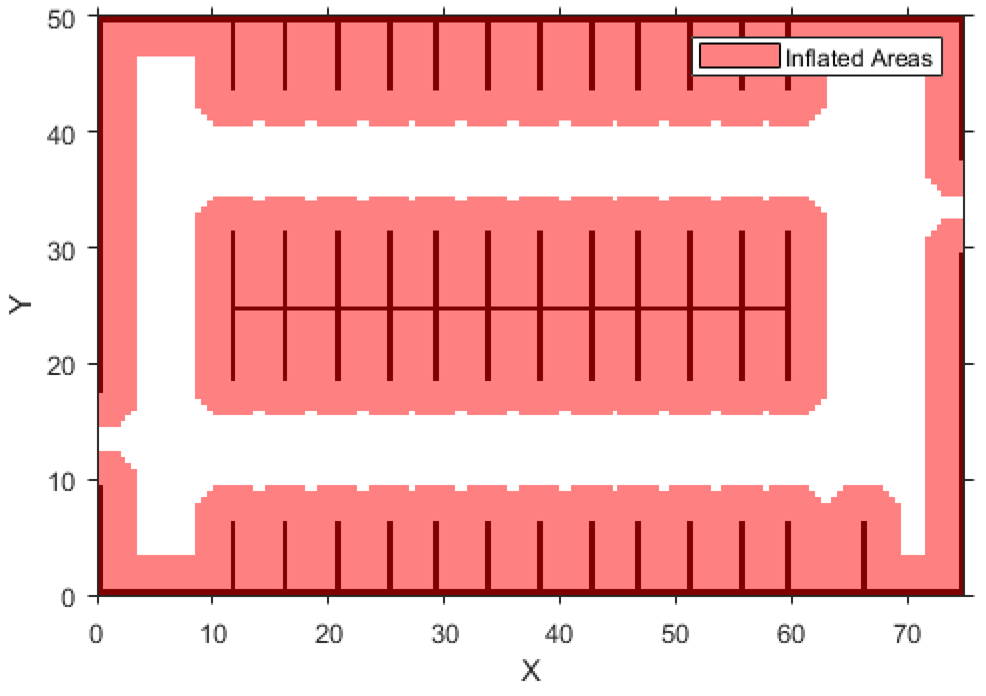
Examples

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)
```

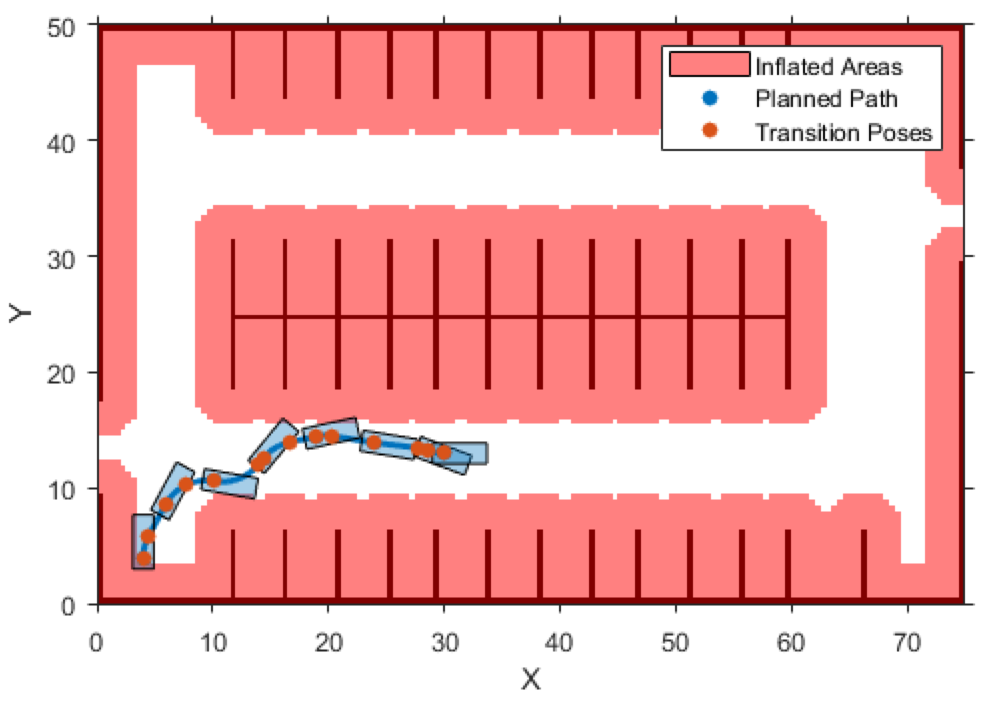
```
isPathValid = logical
   1
```

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.

Output Arguments

`isValid` — Indicates validity of planed vehicle path
`1` | `0`

Indicates validity of planed vehicle path, `refPath`, returned as a logical value of `1` or `0`.

A path is valid (`1`) if the following conditions are true:

- The path has at least one pose.
- The path is collision-free and within the limits of `costmap`.

Algorithms

To check if a vehicle path is valid, the `checkPathValidity` function discretizes the path and then 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.

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 an ACF (aggregate channel features), Faster R-CNN (regions with convolutional neural networks), or Fast R-CNN object 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.

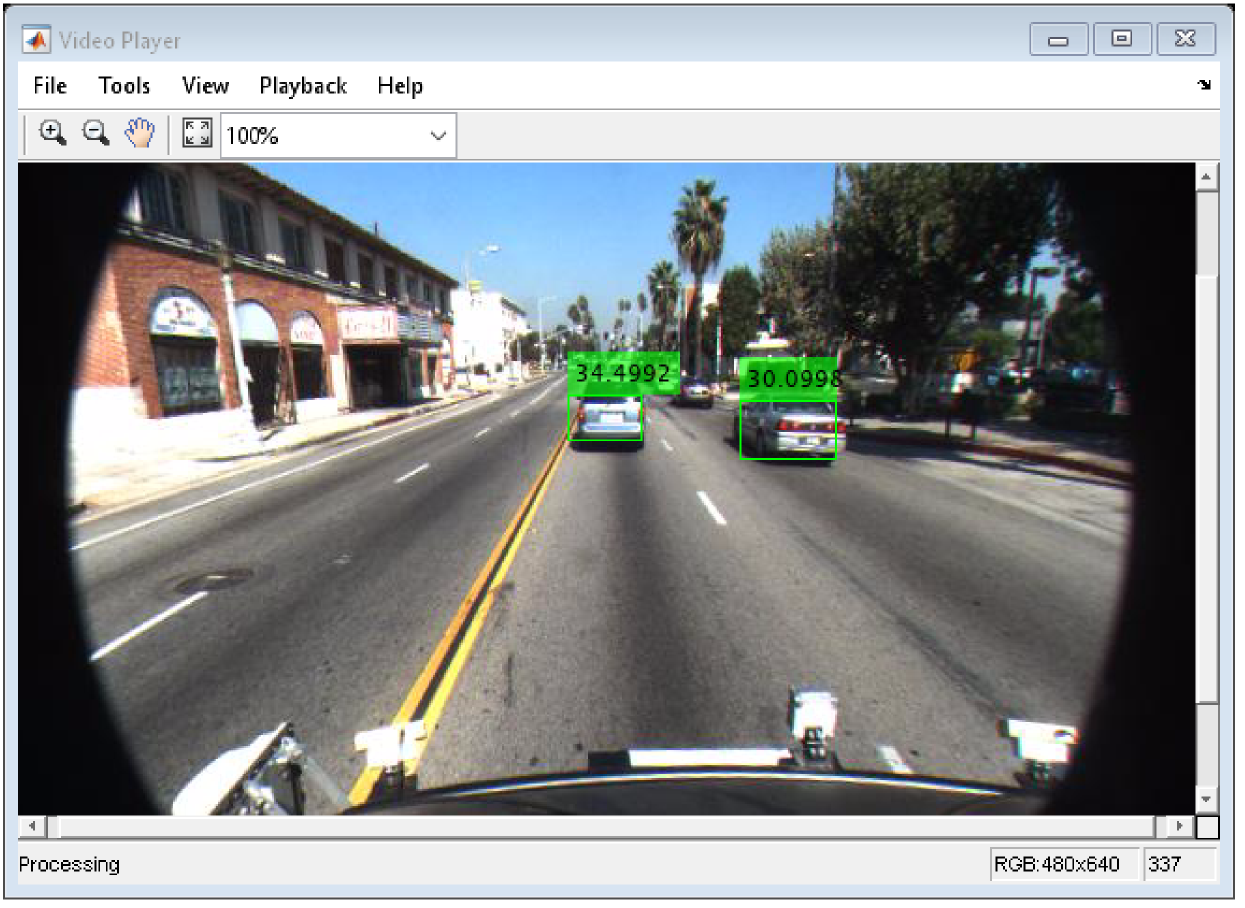
```
vehicleWidth = [1.5 2.5];
detectorMonoCam = configureDetectorMonoCamera(detector,monCam,vehicleWidth);
```

Load a video captured from the camera, and create a video reader and player.

```
videoFile = fullfile(toolboxdir('driving'),'drivingdata','caltech_washington1.avi');
reader = vision.VideoFileReader(videoFile,'VideoOutputDataType','uint8');
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 = ~isDone(reader);
while cont
   I = 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 = ~isDone(reader) && isOpen(videoPlayer);
end
```



Input Arguments

detector — Object detector to configure
`acfObjectDetector` object | `fastRCNNObjectDetector` object | `fasterRCNNObjectDetector` object

Object detector to configure, specified as one of these object detector objects:

- `acfObjectDetector`

- `fastRCNNObjectDetector`
- `fasterRCNNObjectDetector`

Train the object detector before configuring them by using:

- `trainACFObjectDetector`
- `trainFastRCNNObjectDetector`
- `trainFasterRCNNObjectDetector`

`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

Configured object detector, returned as one of these object detector objects:

- `acfObjectDetectorMonoCamera`
- `fastRCNNObjectDetectorMonoCamera`
- `fasterRCNNObjectDetectorMonoCamera`

See Also

`acfObjectDetector` | `acfObjectDetectorMonoCamera` | `fastRCNNObjectDetector` | `fastRCNNObjectDetectorMonoCamera` | `fasterRCNNObjectDetector` | `fasterRCNNObjectDetectorMonoCamera` | `monoCamera`

Introduced in R2017a

constacc

Constant-acceleration motion model

Syntax

```
updatedstate = constacc(state)
updatedstate = constacc(state,dt)
```

Description

`updatedstate = constacc(state)` returns the updated state, `state`, of a constant velocity Kalman filter motion model for a step time of one second.

`updatedstate = constacc(state,dt)` specifies the time step, `dt`.

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`

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`

Classes

`trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

constaccjac

Jacobian for constant-acceleration motion

Syntax

```
jacobian = constaccjac(state)
jacobian = constaccjac(state,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`.

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`

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.

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`

Classes
`trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

constturn

Constant turn-rate motion model

Syntax

```
updatedstate = constturn(state)
updatedstate = constturn(state,dt)
updatedstate = constturn(state,dt,w)
```

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,dt,w)` 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`

Classes
`trackingEKF` | `trackingUKF`

Introduced in R2017a

constturnjac

Jacobian for constant turn-rate motion

Syntax

```
jacobian = constturnjac(state)
jacobian = constturnjac(state,dt)
[jacobian,noisejacobian] = constturnjac(state,dt,w)
```

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,dt,w)` 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`

Classes
`trackingEKF`

Introduced in R2017a

constvel

Constant velocity state update

Syntax

```
updatedstate = constvel(state)
updatedstate = constvel(state,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`.

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 *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. 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`

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`

Classes
`trackingEKF` | `trackingKF` | `trackingUKF`

Introduced in R2017a

constveljac

Jacobian for constant-velocity motion

Syntax

```
jacobian = constveljac(state)
jacobian = constveljac(state,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`.

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. 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`

Output Arguments

`jacobian` — Constant-velocity motion Jacobian
real-valued *2N*-by-*2N* matrix

Constant-velocity motion Jacobian, returned as a real-valued *2N*-by-*2N* matrix. *N* is the number of spatial degrees of motion.

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

Classes

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 frame
`'rectangular'` (default) | `'spherical'`

Measurement frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of the *x*, *y*, and *z* Cartesian coordinates of the tracked object. When specified as `'spherical'`, a measurement consists of the azimuth, elevation, range, and range rate of the tracked object.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the global coordinate system, 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 global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters/second.

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 global coordinate system.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure

Measurement parameters, specified as a structure. The fields of the structure are:

measurementParameters struct

Parameter	Definition	Default
`OriginPosition`	Sensor position with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters.	`[0;0;0]`
`OriginVelocity`	Sensor velocity with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in m/s.	`[0;0;0]`
`Orientation`	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 global coordinate system.	`eye(3)`
`HasVelocity`	Indicates whether measurements contain velocity or range rate components, specified as `true` or `false`.	`false` when `frame` argument is `'rectangular'` and `true` when `frame` argument is `'spherical'`
`HasElevation`	Indicates whether measurements contain elevation components, specified as `true` or `false`.	`true`

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 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>'rectangular</td><td>Specifies the Cartesian position and velocity coordinates of the tracked object with respect to the ego coordinate system.

Rectangular measurements
<table>
<tr><td rowspan="2">HasVelocity</td><td>false</td><td>[x;y;y]</td></tr>
<tr><td>true</td><td>[x;vx;y,vy;z;vz]</td></tr>
</table>

Position units are in meters and velocity units are in m/s.</td></tr>
</table>

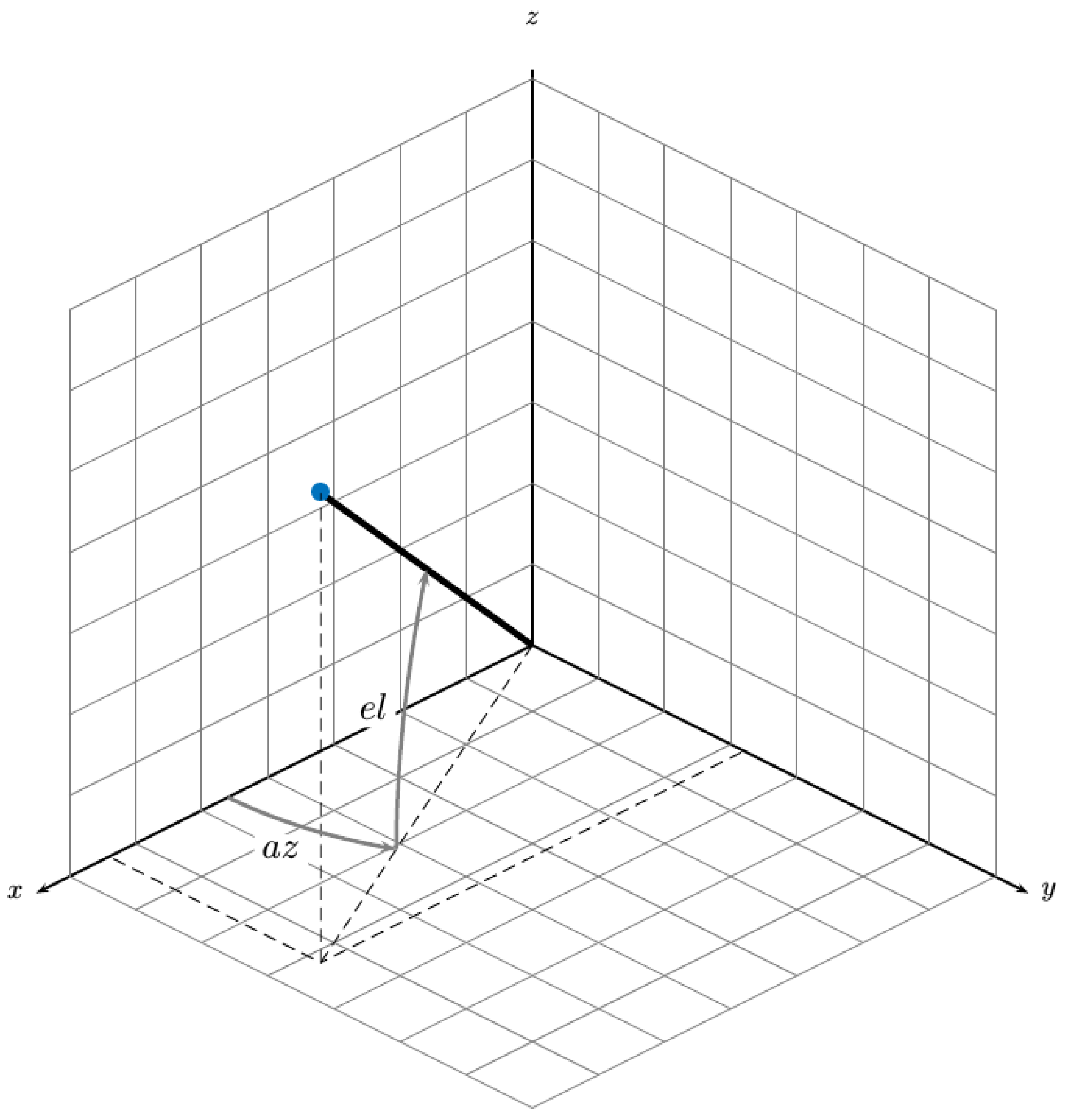
Data Types: double

Definitions

Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Automated Driving System 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.


z
el
az
x
y

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`

Classes

`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 frame
`'rectangular'` (default) | `'spherical'`

Measurement frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of the *x*, *y*, and *z* Cartesian coordinates of the tracked object. When specified as `'spherical'`, a measurement consists of the azimuth, elevation, range, and range rate of the tracked object.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the global coordinate system, 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 global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters/second.

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 global coordinate system.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure

Measurement parameters, specified as a structure. The fields of the structure are:

measurementParameters struct

Parameter	Definition	Default
`OriginPosition`	Sensor position with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters.	`[0;0;0]`
`OriginVelocity`	Sensor velocity with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in m/s.	`[0;0;0]`
`Orientation`	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 global coordinate system.	`eye(3)`
`HasVelocity`	Indicates whether measurements contain velocity or range rate components, specified as `true` or `false`.	`false` when `frame` argument is `'rectangular'` and `true` when `frame` argument is `'spherical'`
`HasElevation`	Indicates whether measurements contain elevation components, specified as `true` or `false`.	`true`

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.

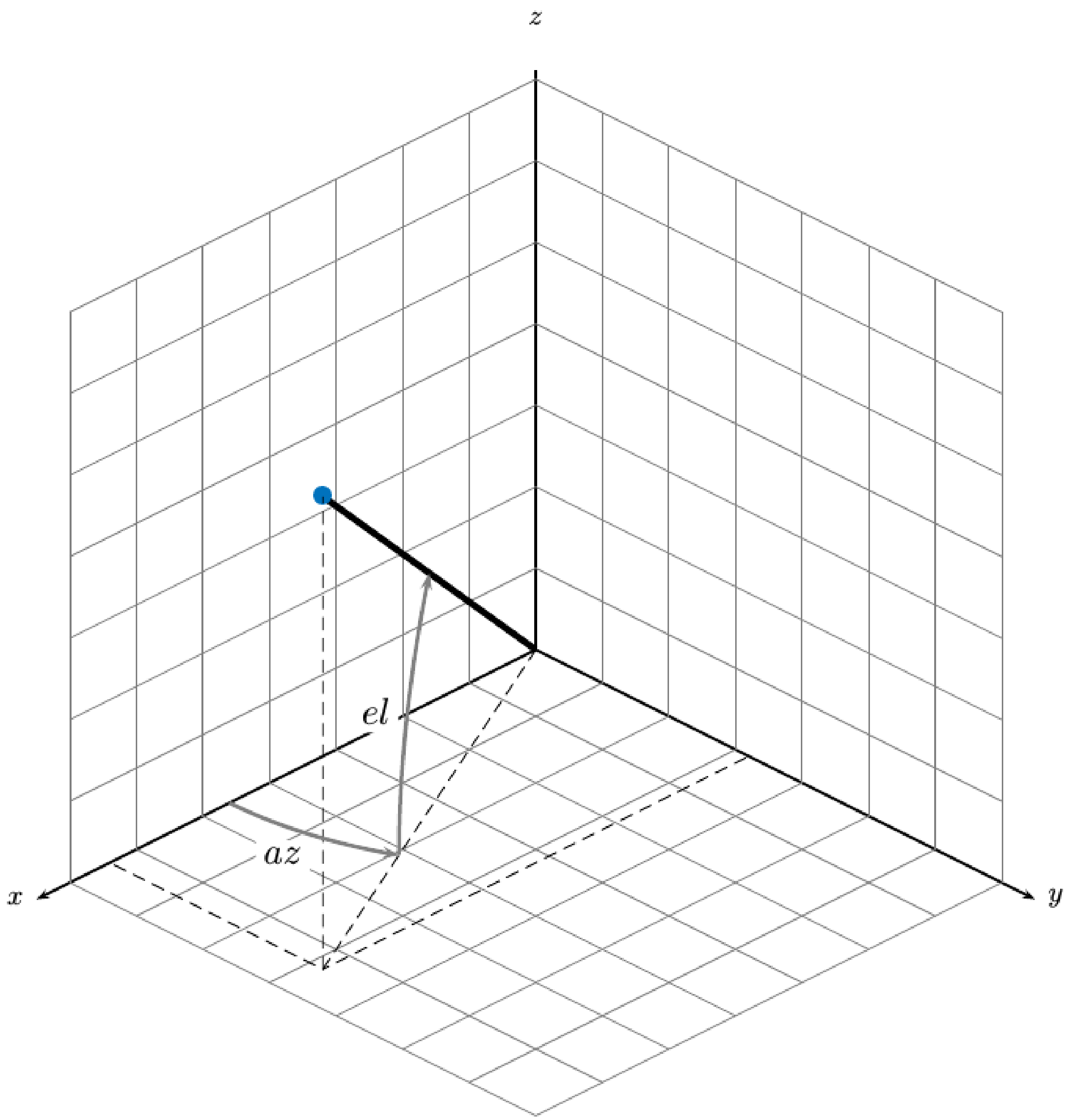
Frame	Measurement Jacobian
`'rectangular'`	Jacobian of the measurements `[x;y;z]` with respect to the state vector. The measurement vector is with respect to the local coordinate system. Coordinates are in meters.
`'spherical'`	Jacobian of the measurement vector `[az;el;r;rr]` with respect to the state 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.

Definitions

Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Automated Driving System 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.



z
el
az
x
y

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`

Classes

`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 *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. 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 frame
`'rectangular'` (default) | `'spherical'`

Measurement frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of the *x*, *y*, and *z* Cartesian coordinates of the tracked object. When specified as `'spherical'`, a measurement consists of the azimuth, elevation, range, and range rate of the tracked object.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the global coordinate system, 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 global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters/second.

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 global coordinate system.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure

Measurement parameters, specified as a structure. The fields of the structure are:

measurementParameters struct

Parameter	Definition	Default
`OriginPosition`	Sensor position with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters.	`[0;0;0]`
`OriginVelocity`	Sensor velocity with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in m/s.	`[0;0;0]`
`Orientation`	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 global coordinate system.	`eye(3)`
`HasVelocity`	Indicates whether measurements contain velocity or range rate components, specified as `true` or `false`.	`false` when `frame` argument is `'rectangular'` and `true` when `frame` argument is `'spherical'`
`HasElevation`	Indicates whether measurements contain elevation components, specified as `true` or `false`.	`true`

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 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>[az;r]</td><td>[az;el;r]</td></tr>
<tr><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>
</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 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;vx;y,vy;z;vz]</code></td></tr>
</table>
Position units are in meters and velocity units are in m/s.</td></tr>
</table>

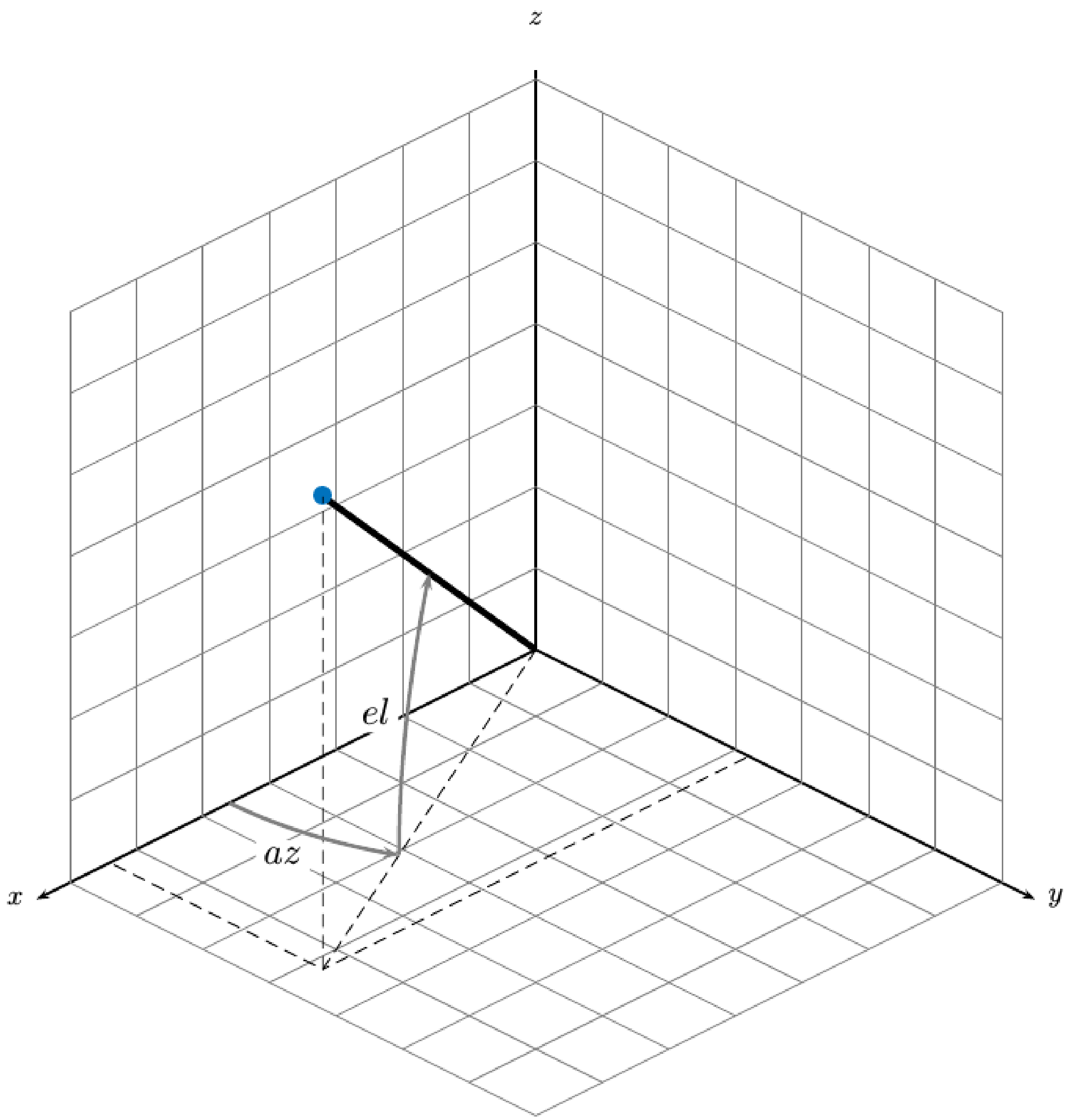
Data Types: `double`

Definitions

Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Automated Driving System 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.


z
el
az
x
y

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`

Classes
`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 *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. 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 frame

`'rectangular'` (default) | `'spherical'`

Measurement frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of the *x*, *y*, and *z* Cartesian coordinates of the tracked object. When specified as `'spherical'`, a measurement consists of the azimuth, elevation, range, and range rate of the tracked object.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the global coordinate system, 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 global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters/second.

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 global coordinate system.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure

Measurement parameters, specified as a structure. The fields of the structure are:

measurementParameters struct

Parameter	Definition	Default
`OriginPosition`	Sensor position with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters.	`[0;0;0]`
`OriginVelocity`	Sensor velocity with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in m/s.	`[0;0;0]`
`Orientation`	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 global coordinate system.	`eye(3)`
`HasVelocity`	Indicates whether measurements contain velocity or range rate components, specified as `true` or `false`.	`false` when `frame` argument is `'rectangular'` and `true` when `frame` argument is `'spherical'`
`HasElevation`	Indicates whether measurements contain elevation components, specified as `true` or `false`.	`true`

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.

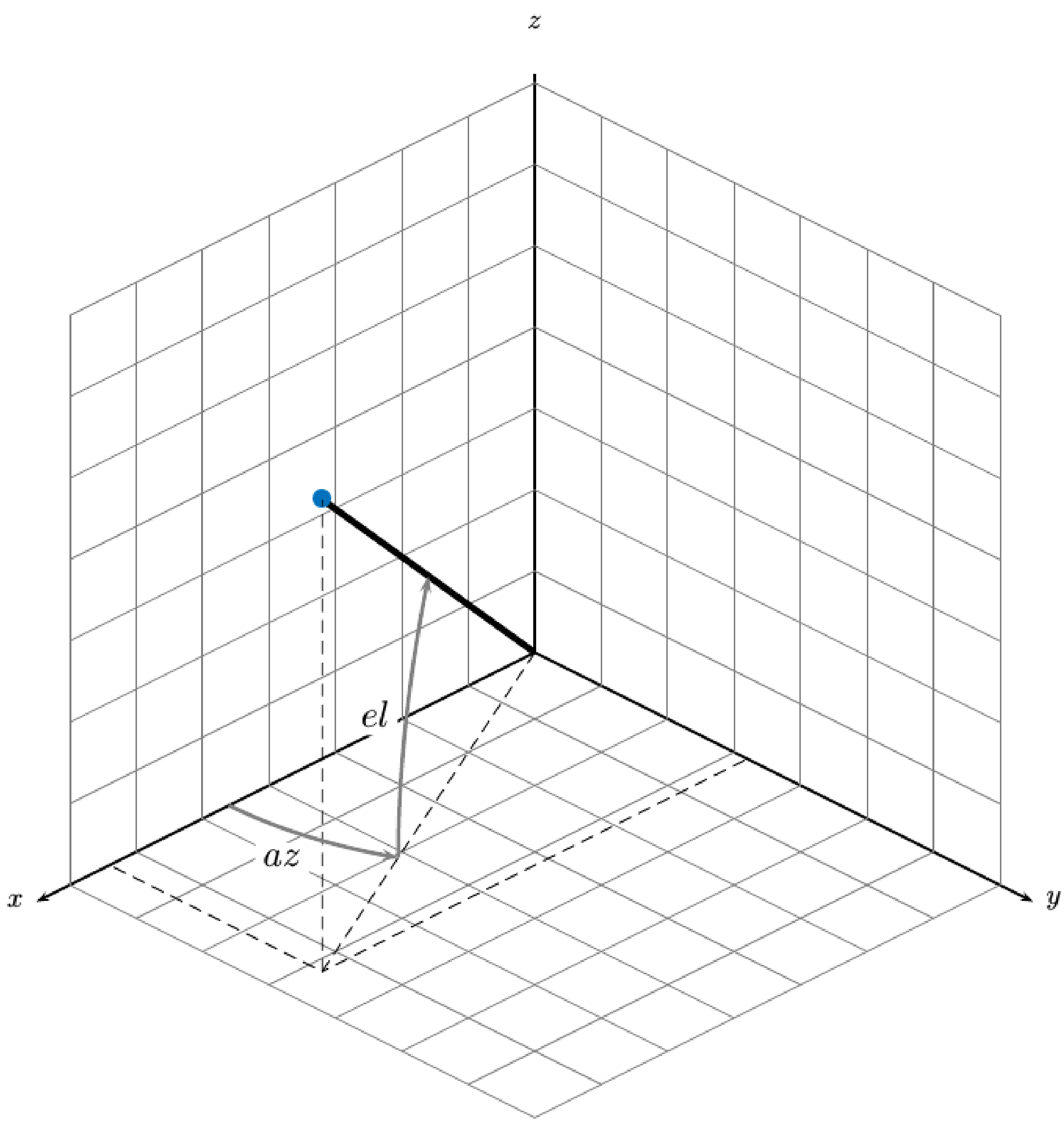
Frame	Measurement Jacobian
'rectangular'	Jacobian of the measurements `[x;y;z]` with respect to the state vector. The measurement vector is with respect to the local coordinate system. Coordinates are in meters.
'spherical'	Jacobian of the measurement vector `[az;el;r;rr]` with respect to the state vector. 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.

Definitions

Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Automated Driving System 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.


z
el
az
x
y

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`

Classes

`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 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 System Toolbox) and vehicle coordinate system on page 3-103 axes. Also defined is the height of the camera above the ground. Specify the intrinsic parameters, the image and world coordinates of corner points in the checkerboard pattern, 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

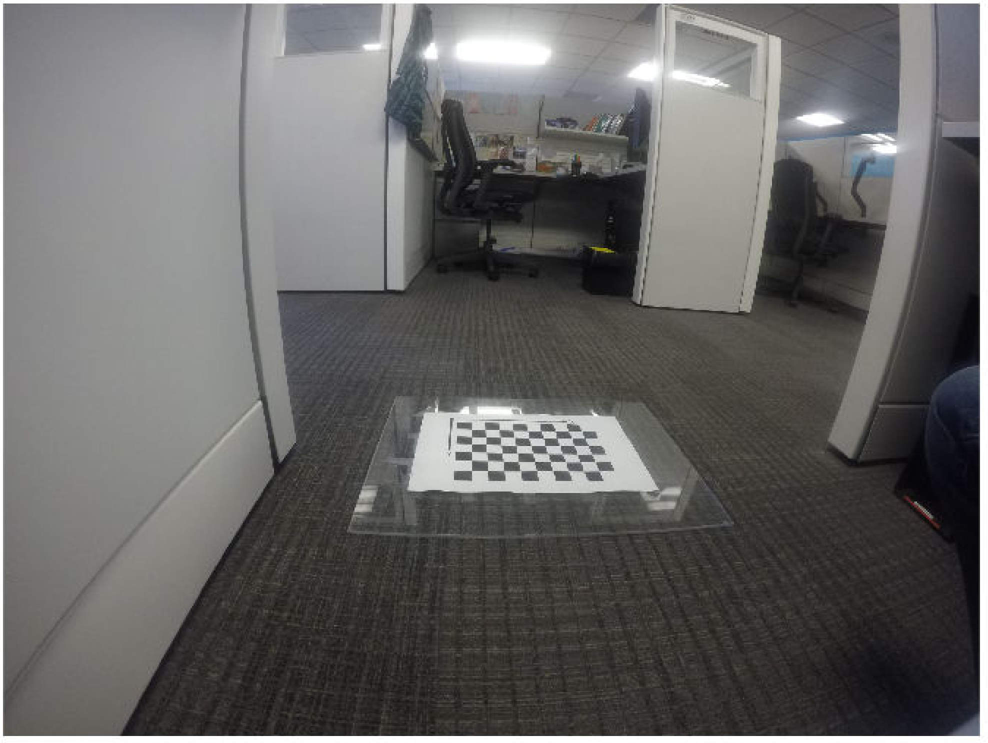
Configure a monocular fisheye camera by removing lens distortion and then estimating the camera's extrinsic parameters. Use an image of a checkerboard as the calibration pattern. For a more detailed look at how to configure a monocular camera that has a fisheye lens, see the “Configure Monocular Fisheye Camera” example.

Load the intrinsic parameters of a monocular camera that has a fisheye lens. `intrinsics` is a `fisheyeIntrinsics` 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)
```



```
Warning: Image is too big to fit on screen; displaying at 33%
```

Detect the coordinates of the checkerboard corners in the image.

```
[imagePoints,boardSize] = detectCheckerboardPoints(I);
```

Generate the corresponding world coordinates of the corners.

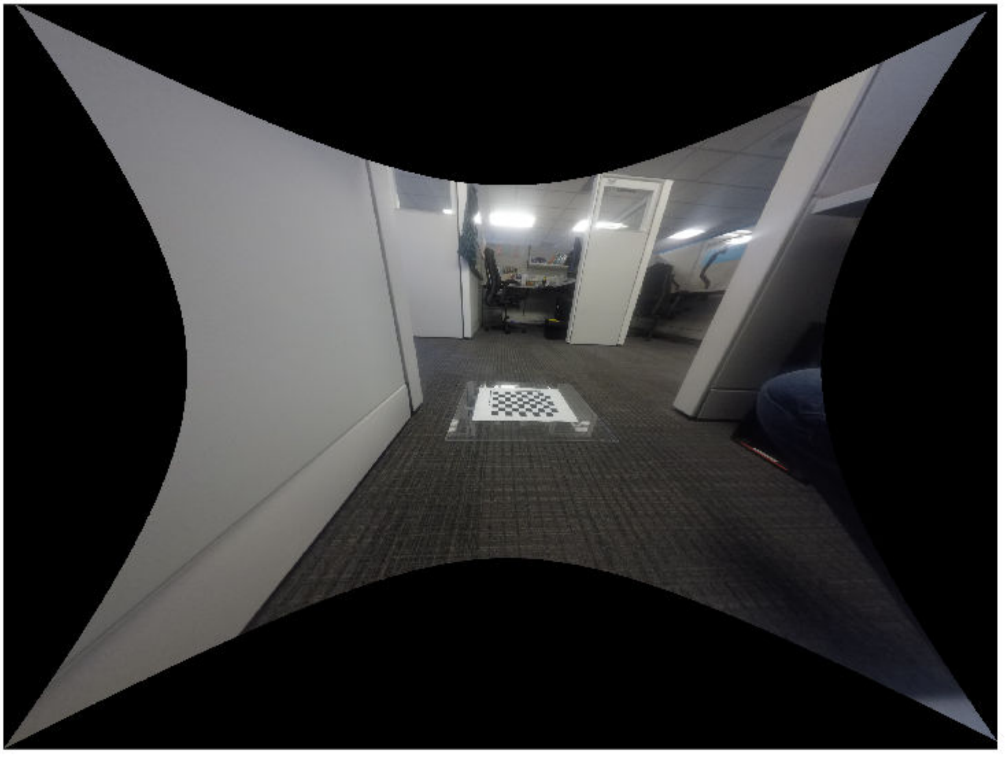
```
squareSize = 0.029; % Square size in meters
worldPoints = generateCheckerboardPoints(boardSize,squareSize);
```

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` object. Display the image to confirm distortion is removed.

```
[undistortedI,camIntrinsics] = undistortFisheyeImage(I,intrinsics,'Output','full');
imshow(undistortedI)
```



```
Warning: Image is too big to fit on screen; displaying at 17%
```

Configure the monocular camera using the estimated parameters.

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

```
monoCam = 
  monoCamera with properties:

        Intrinsics: [1×1 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`.

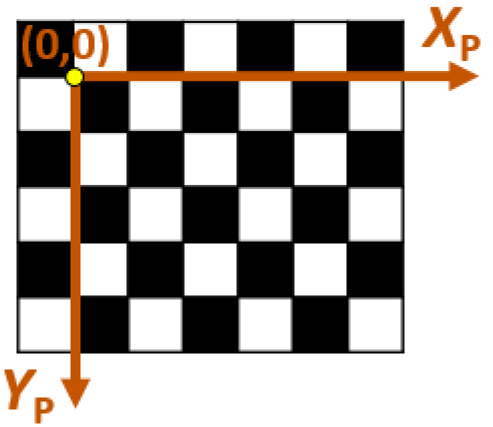
Data Types: `single` | `double`

`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`.

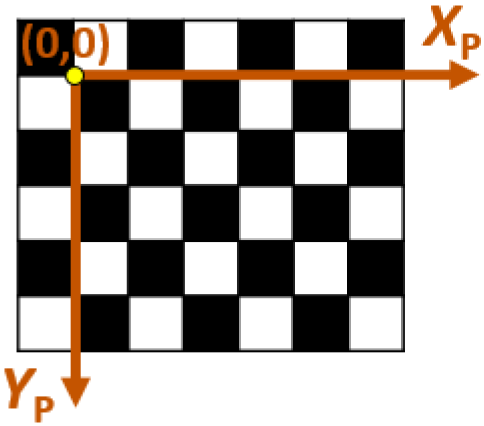
Point (0,0) corresponds to the bottom-right corner of the top-left square of the checkerboard.



Data Types: `single` | `double`

`patternOriginHeight` — Height of checkerboard pattern's origin
nonnegative scalar

Height of the checkerboard pattern's origin above the ground, specified as a nonnegative scalar. The origin is the bottom-right corner of the top-left square of the checkerboard. If the pattern is on the ground, 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'`

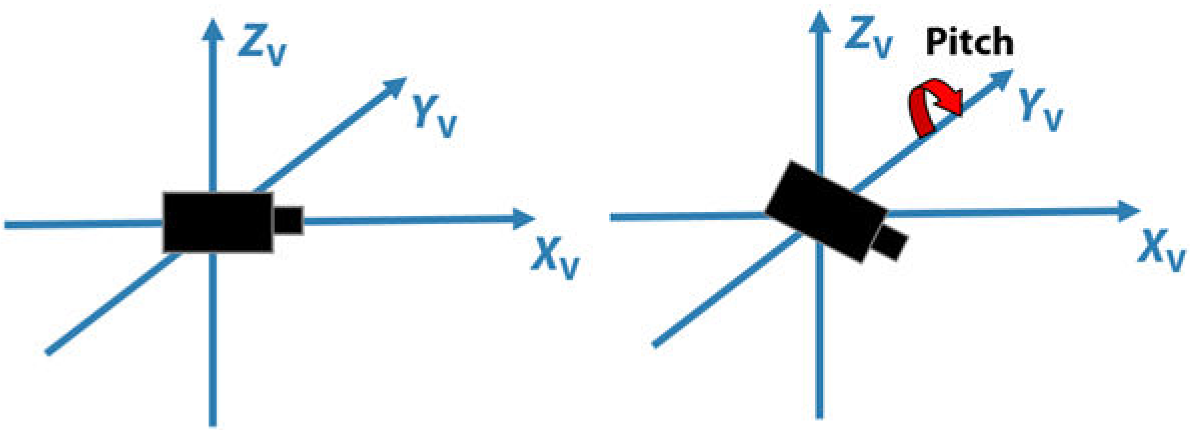
Position of the checkerboard pattern relative to the ground, specified as the comma-separated pair consisting of `'PatternPosition'` and one of the following:

- `'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
scalar

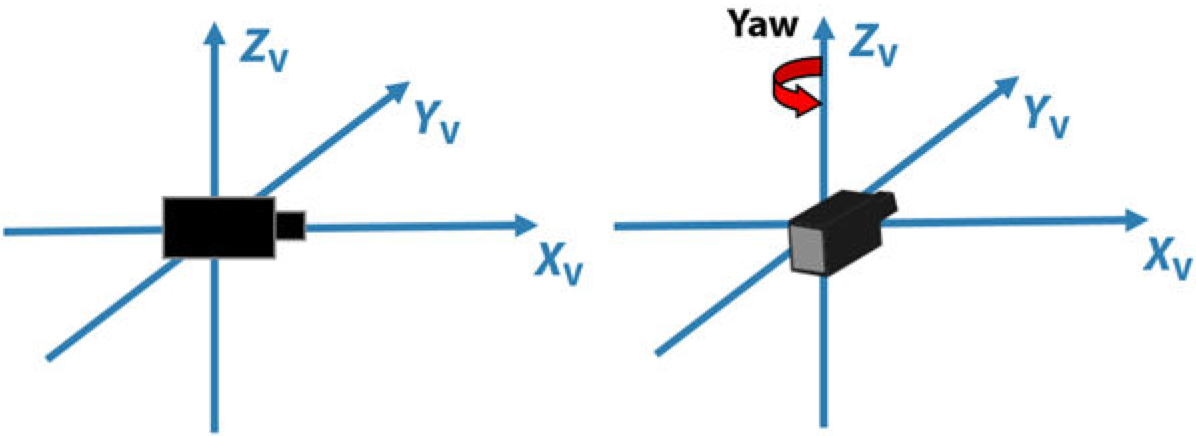
Pitch angle between the horizontal plane of the vehicle and the optical axis of the camera, returned as a scalar in degrees. `pitch` uses the ISO convention for rotation, with a clockwise positive angle direction when looking in the positive direction of the vehicle's Y_V-axis.



For more details, see “Angle Directions” on page 3-104.

`yaw` — Yaw angle
scalar

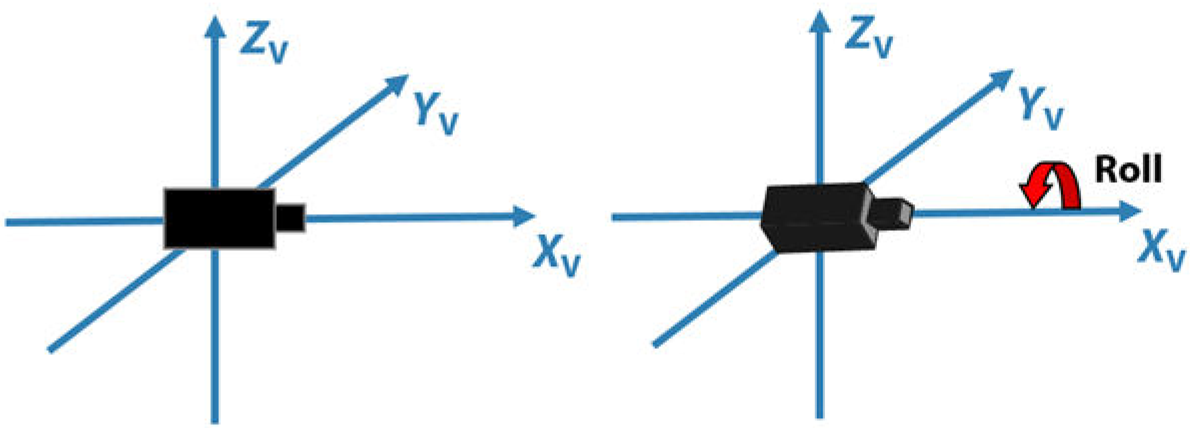
Yaw angle between the X_V-axis of the vehicle and the optical axis of the camera, returned as a scalar in degrees. `yaw` uses the ISO convention for rotation, with a clockwise positive angle direction when looking in the positive direction of the vehicle's Z_V-axis.



For more details, see “Angle Directions” on page 3-104.

`roll` — Roll angle
scalar

Roll angle of the camera around its optical axis, returned as a 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 3-104.

`height` — Perpendicular height from ground to camera
nonnegative scalar

Perpendicular height from the ground to the focal point of the camera, returned as a nonnegative scalar in world units, such as meters.

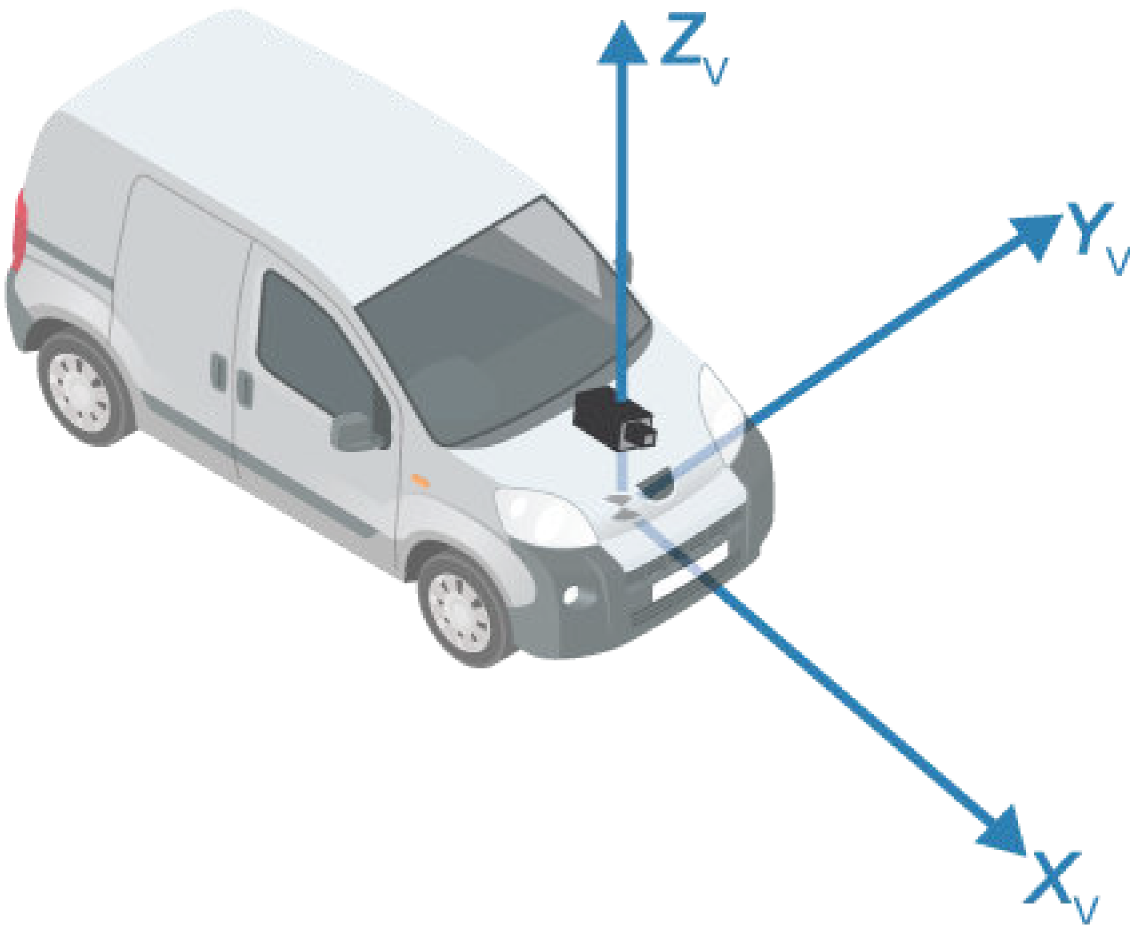
Definitions

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.

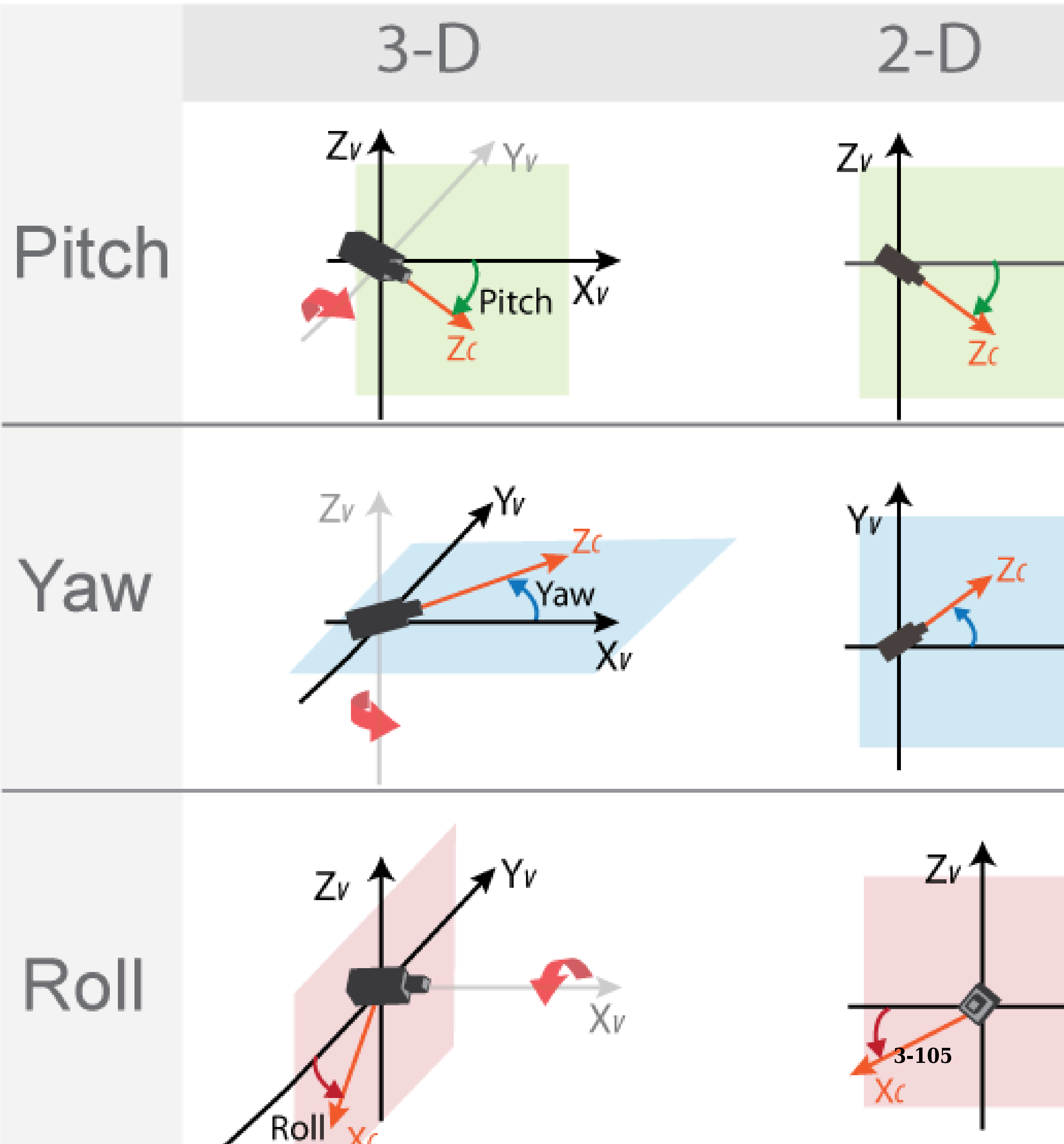
By default, the origin of this coordinate system is on the road surface, directly below the camera center (focal point of camera).



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
Pitch
Xv
Zc
Zv
Zc
Yaw
Zv
Yv
Zc
Yaw
Xv
Yv
Zc
Roll
Zv
Yv
Xv
Roll
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 System 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
numeric 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 numeric 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
vector of numeric scalars

x-axis locations of boundary, specified as a vector of numeric scalars. 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)
numeric scalar

Number of matches (true positives), returned as a numeric scalar.

`numMissed` — Number of misses (false negatives)
numeric scalar

Number of misses (false negatives), returned as a numeric scalar.

`numFalsePositives` — Number of false positives
numeric scalar

Number of false positives, returned as a numeric scalar.

`assignments` — Assignment indices for ground truth boundaries
cell array of numeric arrays

Assignment indices for ground truth boundaries, returned as a cell array of numeric 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

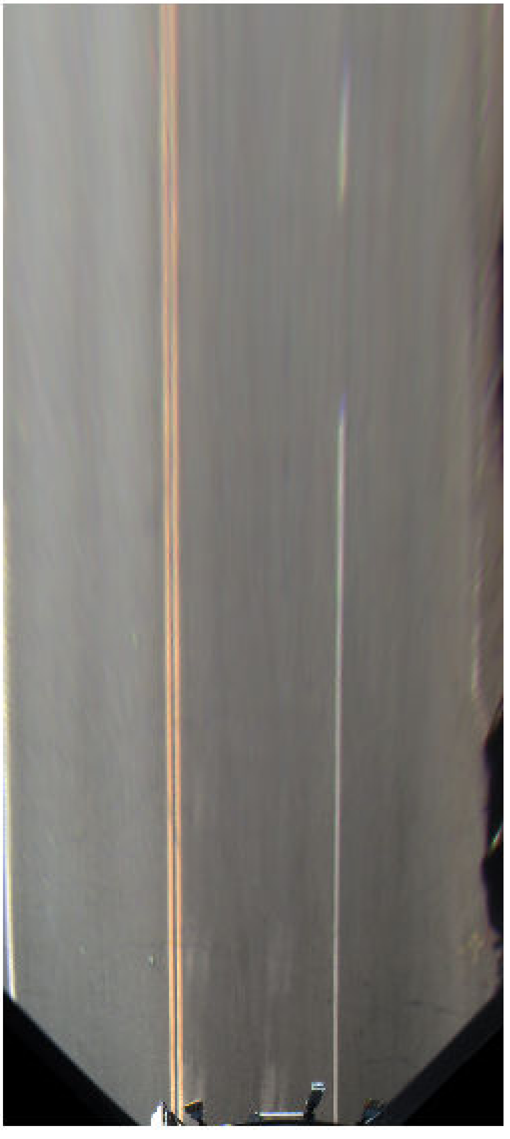
Find lanes in an image by using cubic lane boundary models. Overlay the identified lanes on the original image and on a bird's-eye-view transformation of the image.

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)
```

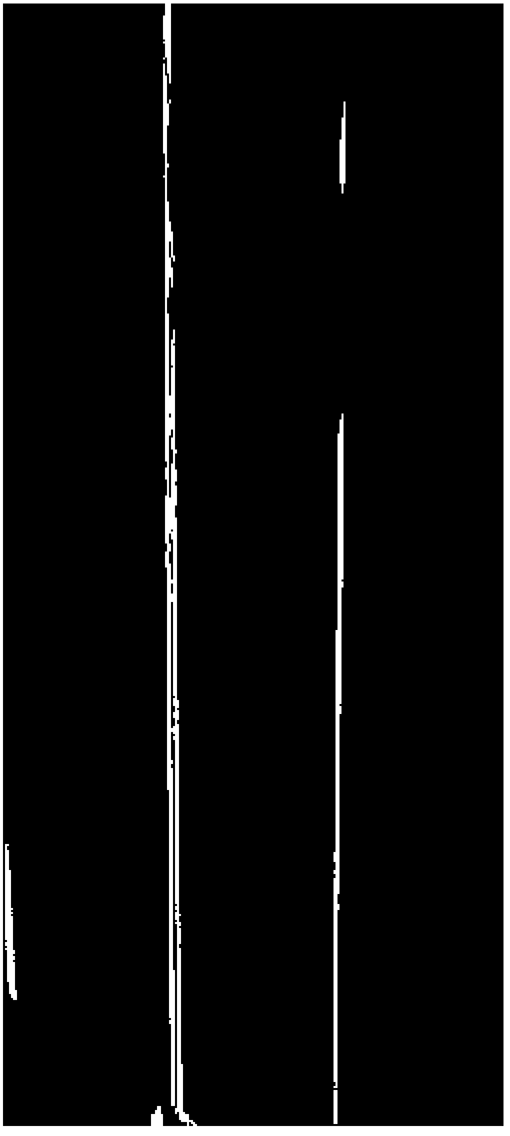


Set the approximate lane marker width in world units (meters).

```
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.

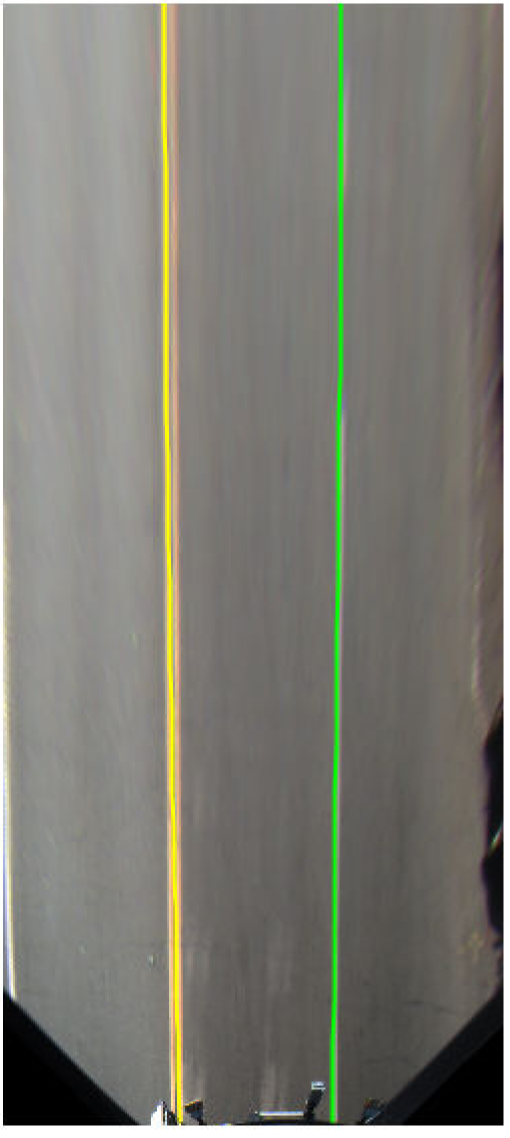
```
boundaries = findCubicLaneBoundaries(xyBoundaryPoints,approxBoundaryWidth);
```

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
scalar

Approximate boundary width, specified as a 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.

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

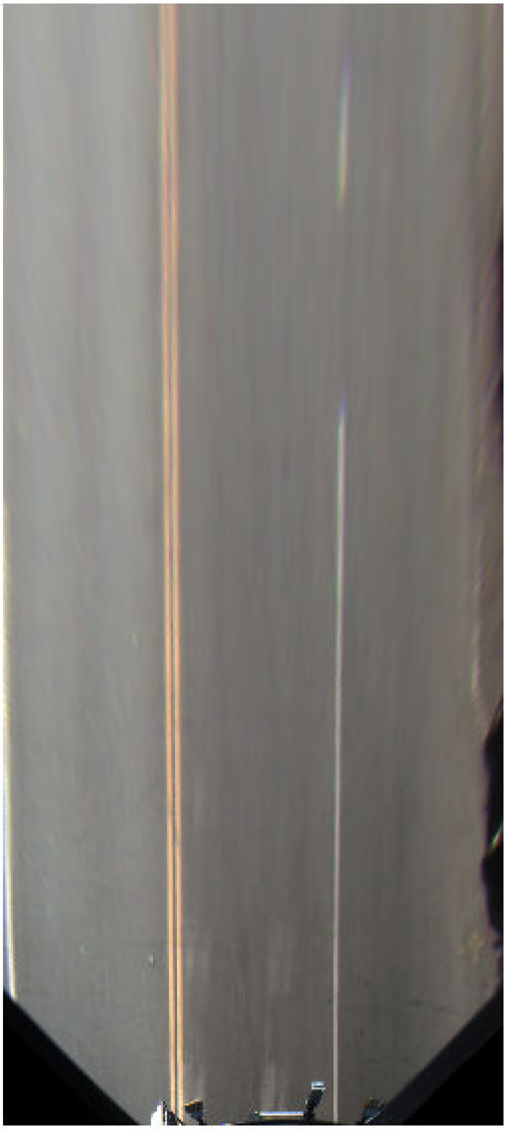
Find lanes in an image by using parabolic lane boundary models. Overlay the identified lanes on the original image and on a bird's-eye-view transformation of the image.

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)
```

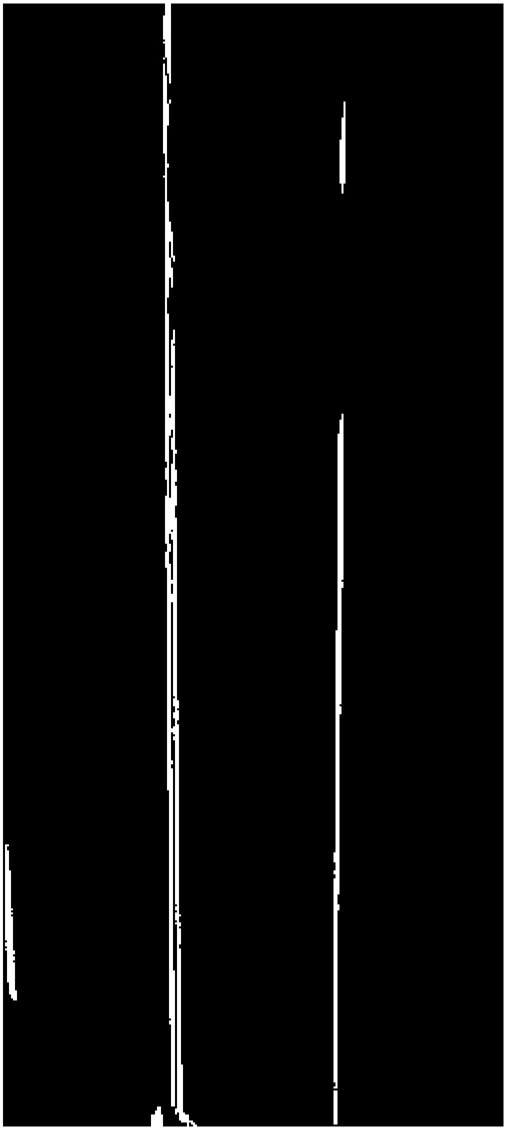


Set the approximate lane marker width in world units (meters).

```
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
scalar

Approximate boundary width, specified as a 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.

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 = multiObjectTracker('FilterInitializationFcn',@initcaekf);
```

Update the tracker with a single detection and get the tracks output.

```
detection = objectDetection(0,[10;-20;4],'ObjectClassID',3);
tracks = updateTracks(tracker,detection,0)
```

```
tracks = struct with fields:
             TrackID: 1
                Time: 0
                 Age: 1
               State: [9x1 double]
```

```
     StateCovariance: [9x9 double]
         IsConfirmed: 1
           IsCoasted: 0
       ObjectClassID: 3
    ObjectAttributes: {}
```

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   -20     4
```

Find Position and Covariance of 3-D Constant-Velocity Object

Create an extended Kalman filter tracker for 3-D constant-velocity motion.

```
tracker = multiObjectTracker('FilterInitializationFcn',@initcvekf);
```

Update the tracker with a single detection and get the tracks output.

```
detection = objectDetection(0,[10;3;-7],'ObjectClassID',3);
tracks = updateTracks(tracker,detection,0)
```

```
tracks = struct with fields:
             TrackID: 1
                Time: 0
                 Age: 1
               State: [6x1 double]
     StateCovariance: [6x6 double]
         IsConfirmed: 1
           IsCoasted: 0
       ObjectClassID: 3
    ObjectAttributes: {}
```

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     3    -7


positionCovariance = 3×3

     1     0     0
     0     1     0
     0     0     1
```

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 System 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.

Definitions

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`

Classes
`objectDetection`

System Objects
`multiObjectTracker`

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 = multiObjectTracker('FilterInitializationFcn',@initcaekf);
```

Initialize the tracker with a one detection.

```
detection = objectDetection(0,[10;-20;4],'ObjectClassID',3);
tracks = updateTracks(tracker,detection,0);
```

Add a second detection at a later time and translated position.

```
detection = objectDetection(0.1,[10.3;-20.2;4],'ObjectClassID',3);
tracks = updateTracks(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 = multiObjectTracker('FilterInitializationFcn',@initcaekf);
```

Initialize the tracker with a one detection.

```
detection = objectDetection(0,[10;-20;4],'ObjectClassID',3);
tracks = updateTracks(tracker,detection,0);
```

Add a second detection at a later time and translated position.

```
detection = objectDetection(0.1,[10.3;-20.2;4.3],'ObjectClassID',3);
tracks = updateTracks(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 System 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.

Definitions

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

Classes
objectDetection

System Objects
multiObjectTracker

Introduced in R2017a

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  100.0000         0         0         0         0         0         0
         0         0  100.0000         0         0         0         0         0
         0         0         0    2.1000         0         0         0         0
         0         0         0         0  100.0000         0         0         0
         0         0         0         0         0  100.0000         0         0
         0         0         0         0         0         0    2.1000         0
         0         0         0         0         0         0         0  100.0000
         0         0         0         0         0         0         0         0  100.0
```

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`

Classes
`objectDetection` | `trackingEKF` | `trackingKF` | `trackingUKF`

System Objects
`multiObjectTracker`

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`

Classes
`objectDetection` | `trackingEKF` | `trackingKF` | `trackingUKF`

System Objects
`multiObjectTracker`

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     0     0     0     0     0     0     0     0
     0   100     0     0     0     0     0     0     0
     0     0   100     0     0     0     0     0     0
     0     0     0     2     0     0     0     0     0
     0     0     0     0   100     0     0     0     0
     0     0     0     0     0   100     0     0     0
     0     0     0     0     0     0     2     0     0
     0     0     0     0     0     0     0   100     0
```

```
     0     0     0     0     0     0     0     0   100
```

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

Classes
objectDetection | trackingEKF | trackingKF | trackingUKF

System Objects
multiObjectTracker

Introduced in R2017a

initctekf

Create constant turn-rate extended Kalman filter from detection report

Syntax

```
filter = initcaekf(detection)
```

Description

`filter = initcaekf(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     0     0     0     0     0     0
     0   100     0     0     0     0     0
     0     0     2     0     0     0     0
     0     0     0   100     0     0     0
     0     0     0     0   100     0     0
     0     0     0     0     0     2     0
     0     0     0     0     0     0   100
```

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`

Classes

`objectDetection` | `trackingEKF` | `trackingKF` | `trackingUKF`

System Objects

`multiObjectTracker`

Introduced in R2017a

initctukf

Create constant turn-rate unscented Kalman filter from detection report

Syntax

```
filter = initcaukf(detection)
```

Description

`filter = initcaukf(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     0     0     0     0     0     0
     0   100     0     0     0     0     0
     0     0     2     0     0     0     0
     0     0     0   100     0     0     0
     0     0     0     0   100     0     0
     0     0     0     0     0     2     0
     0     0     0     0     0     0   100
```

Create 2-D Constant Turnrate UKF 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 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 = initctekf(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

Classes
objectDetection | trackingEKF | trackingKF | trackingUKF

System Objects
multiObjectTracker

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

Classes
objectDetection | trackingEKF | trackingKF | trackingUKF

System Objects
multiObjectTracker

Introduced in R2017a

initcvkf

Create constant-velocity linear Kalman filter from detection report

Syntax

```
filter = initcakf(detection)
```

Description

`filter = initcakf(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

Classes
objectDetection | objectDetection | trackingEKF | trackingKF | trackingUKF

System Objects
multiObjectTracker

Introduced in R2017a

initcvukf

Create constant-velocity unscented Kalman filter from detection report

Syntax

```
filter = initcaukf(detection)
```

Description

`filter = initcaukf(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

Classes
objectDetection | trackingEKF | trackingKF | trackingUKF

System Objects
multiObjectTracker

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

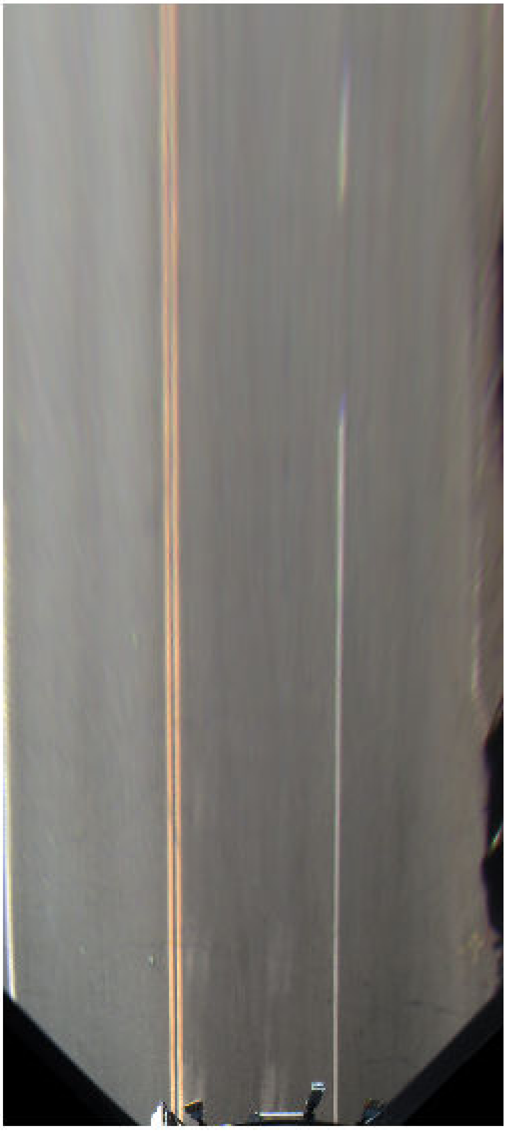
Find lanes in an image by using parabolic lane boundary models. Overlay the identified lanes on the original image and on a bird's-eye-view transformation of the image.

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)
```

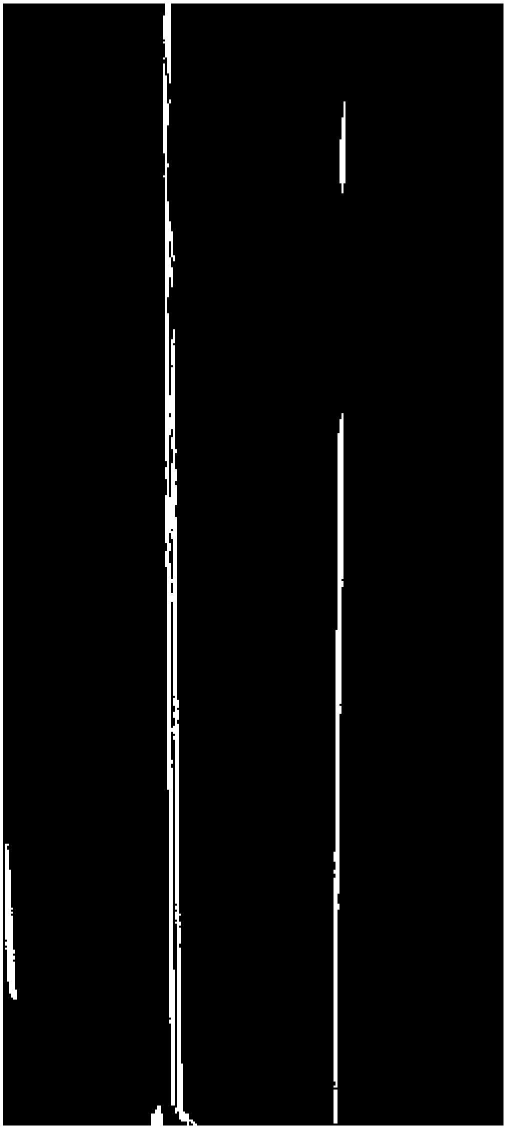


Set the approximate lane marker width in world units (meters).

```
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.

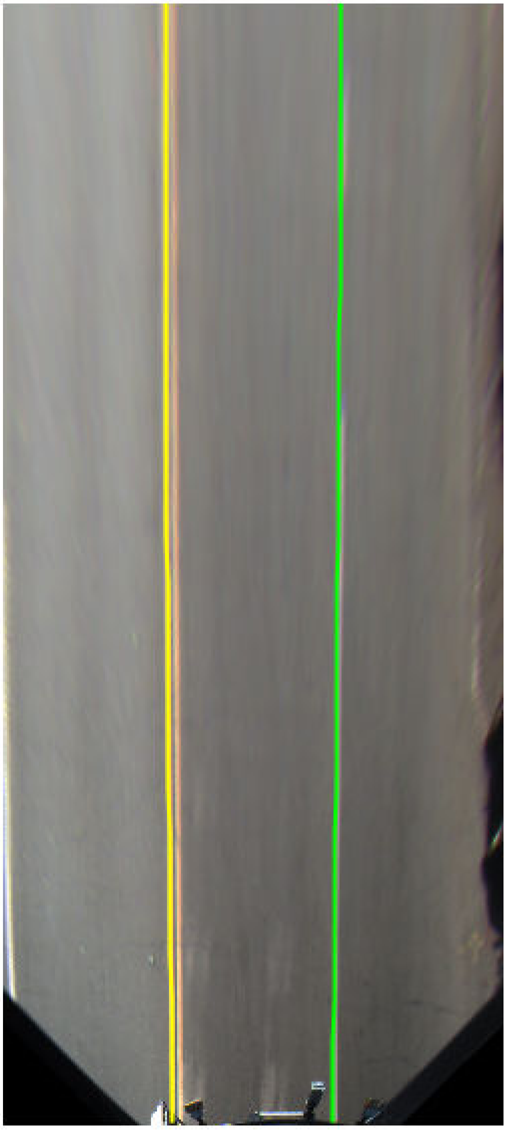
```
boundaries = findParabolicLaneBoundaries(xyBoundaryPoints,approxBoundaryWidth);
```

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
vector of numeric scalars

x-axis locations at which to display the lane boundaries, specified as a vector of numeric scalars 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.

See Also

birdsEyeView | cubicLaneBoundary | fitPolynomialRANSAC | monoCamera | parabolicLaneBoundary

Introduced in R2017a

lateralControllerStanley

Compute steering angle command for path following 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
```

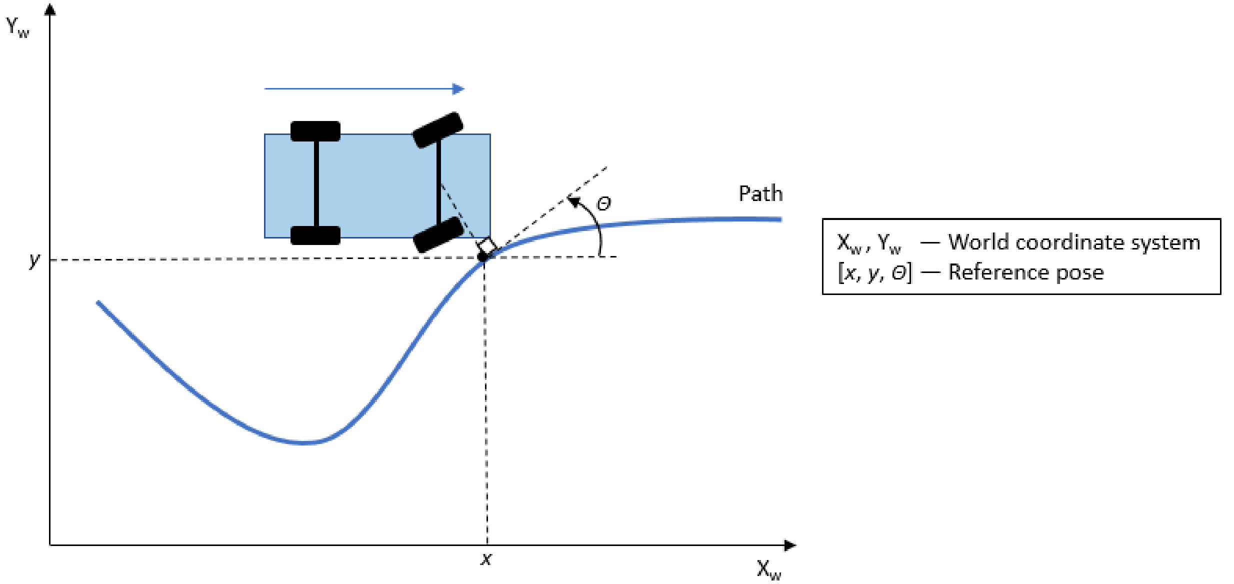
Input Arguments

refPose — Reference pose
[*x*, *y*, *Θ*] vector

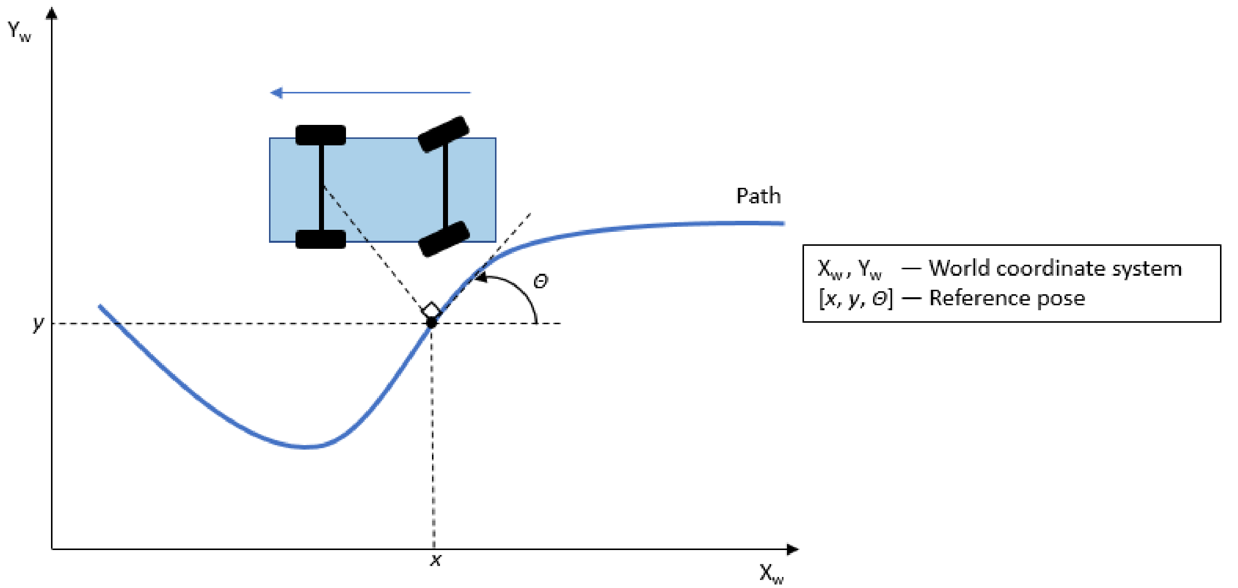
Reference pose, specified as an [*x*, *y*, *Θ*] vector. *x* and *y* are in meters, and *Θ* is in degrees.

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.



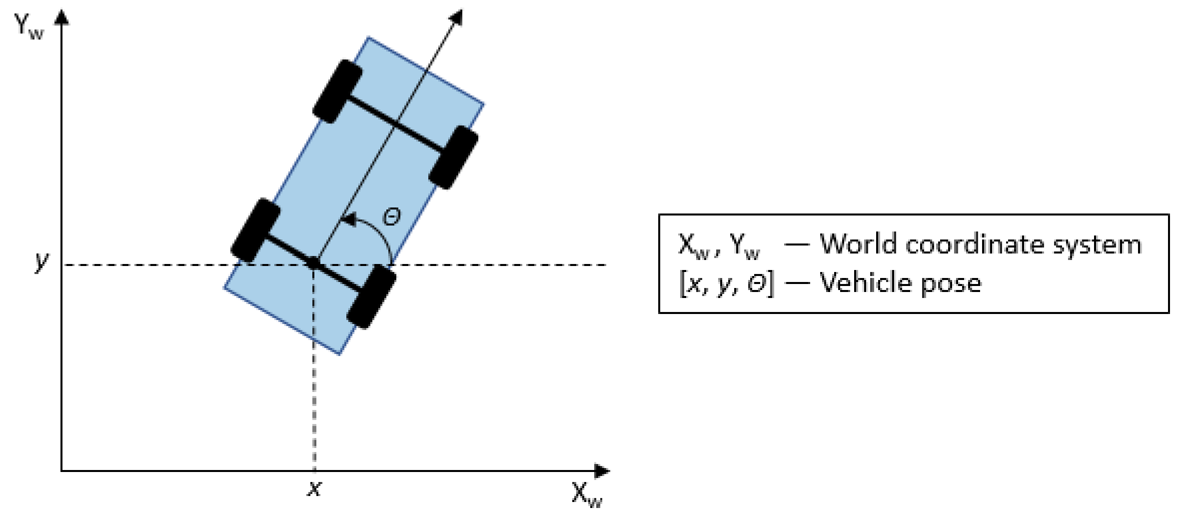
Data Types: `single` | `double`

`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 System Toolbox”.

Data Types: `single` | `double`

`currVelocity` — Current longitudinal velocity
scalar

Current longitudinal velocity of the vehicle, specified as a 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 3-200.

PositionGain — Position gain
2.5 (default) | positive scalar

Position gain of the vehicle, specified as the comma-separated pair consisting of 'PositionGain' and 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.

Wheelbase — Distance between front and rear axles of vehicle
2.8 (default) | scalar

Distance between the front and rear axles of the vehicle, in meters, specified as the comma-separated pair consisting of 'Wheelbase' and a scalar. This value applies only when the vehicle is in forward motion.

MaxSteeringAngle — Maximum allowed steering angle
35 (default) | 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 scalar in the range (0, 180).

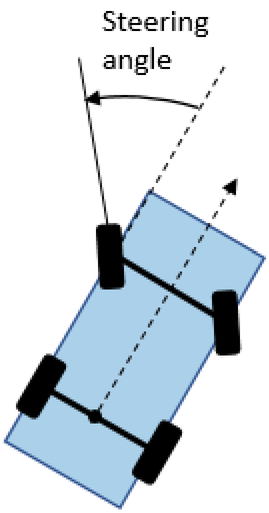
The steerCmd value is saturated to the range [-MaxSteeringAngle, MaxSteeringAngle].

- Values below -MaxSteeringAngle are set to -MaxSteeringAngle.
- Values above MaxSteeringAngle are set to MaxSteeringAngle.

Output Arguments

steerCmd — Steering angle command
scalar

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



For more details, see “Coordinate Systems in Automated Driving System 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

Objects
`pathPlannerRRT`

Topics

“Automated Parking Valet”
“Coordinate Systems in Automated Driving System Toolbox”

Introduced in R2018b

plotCoverageArea

Plot bird’s-eye view coverage area

Syntax

```
plotCoverageArea(caPlotter,position,range,orientation,fieldOfView)
```

Description

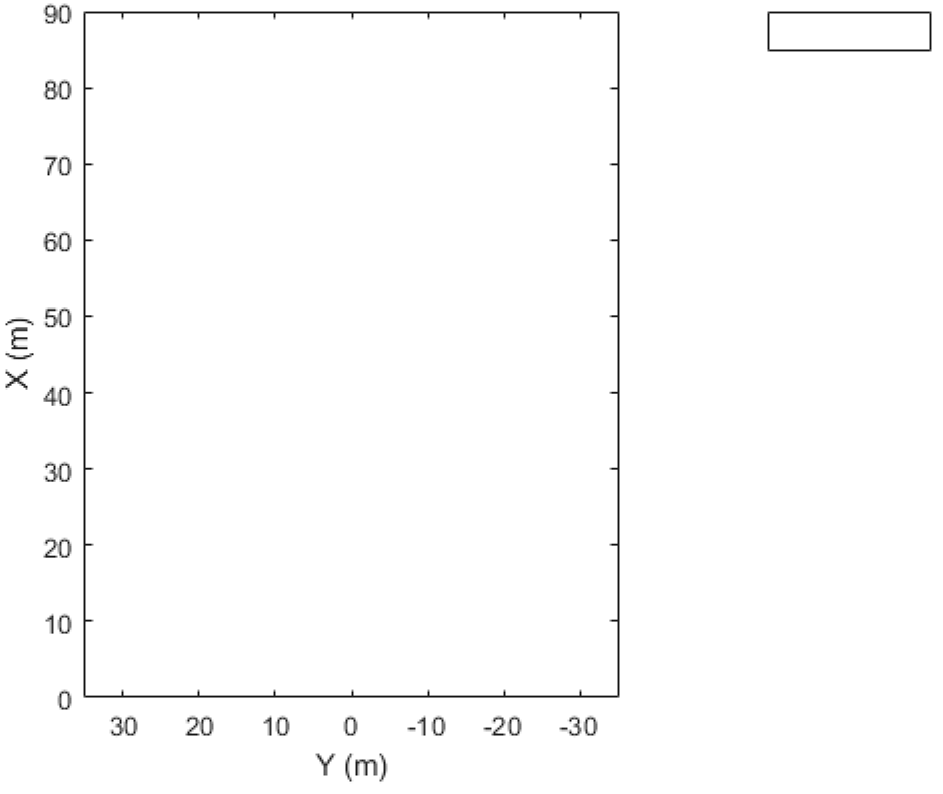
`plotCoverageArea(caPlotter,position,range,orientation,fieldOfView)` returns a plot of a bird’s-eye view coverage area. Use `coverageAreaPlotter` to obtain the `caPlotter` figure.

Examples

Create Coverage Area for Front-Facing Center-Mounted Radar Sensor

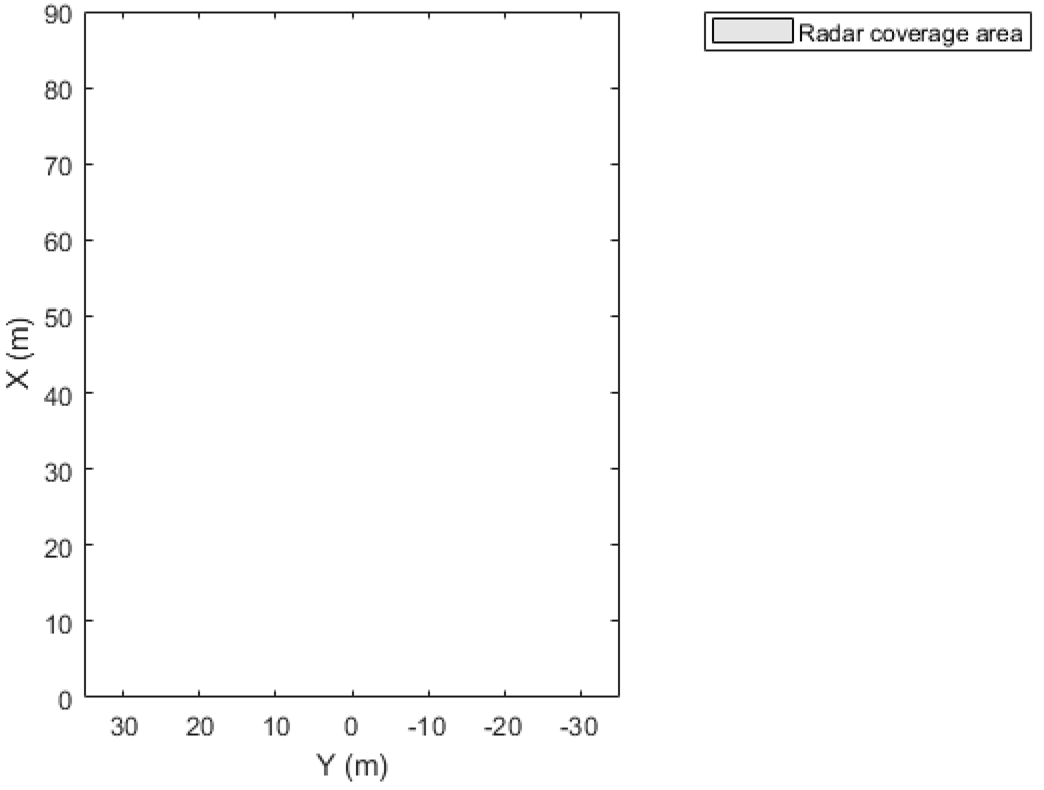
Create a bird's-eye plot.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
```



Create a coverage plotter for the bird's-eye plot.

```
caPlotter = coverageAreaPlotter(bep,'DisplayName','Radar coverage area');
```

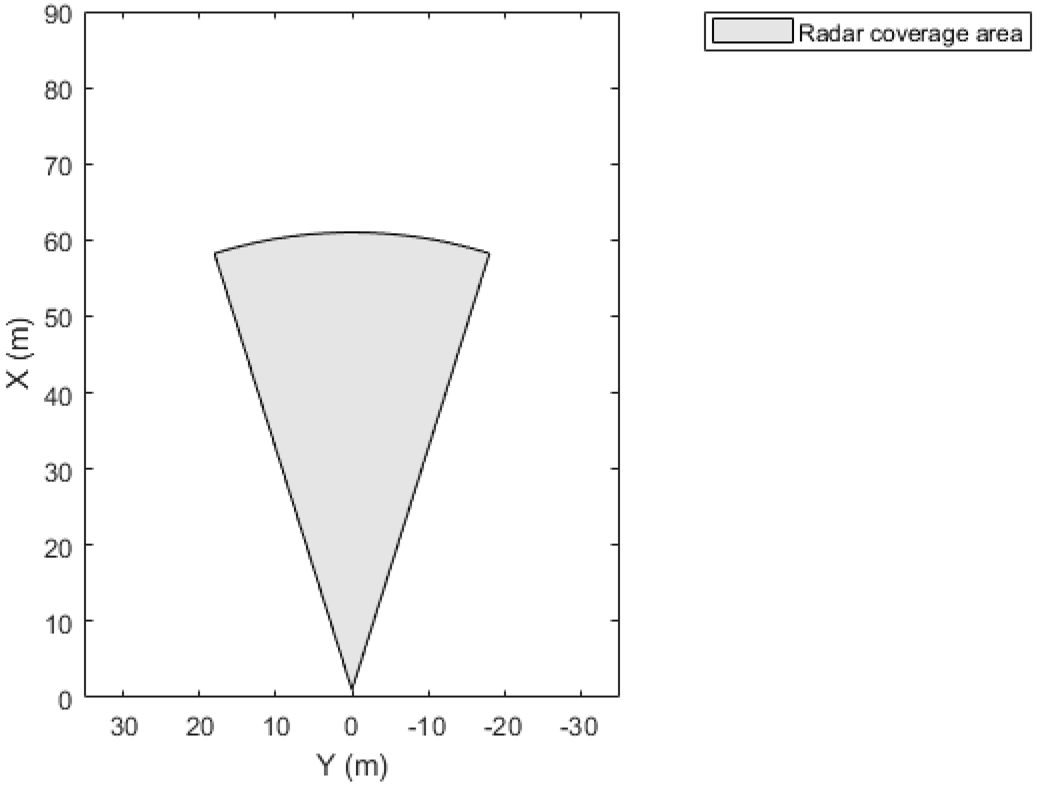


Update the plot with a field of view of 35 degrees and a range of 60 meters.

```
mountPosition = [1 0];
range = 60;
orientation = 0;
fieldOfView = 35;
```

Plot the coverage area.

```
plotCoverageArea(caPlotter,mountPosition,range,orientation,fieldOfView);
```

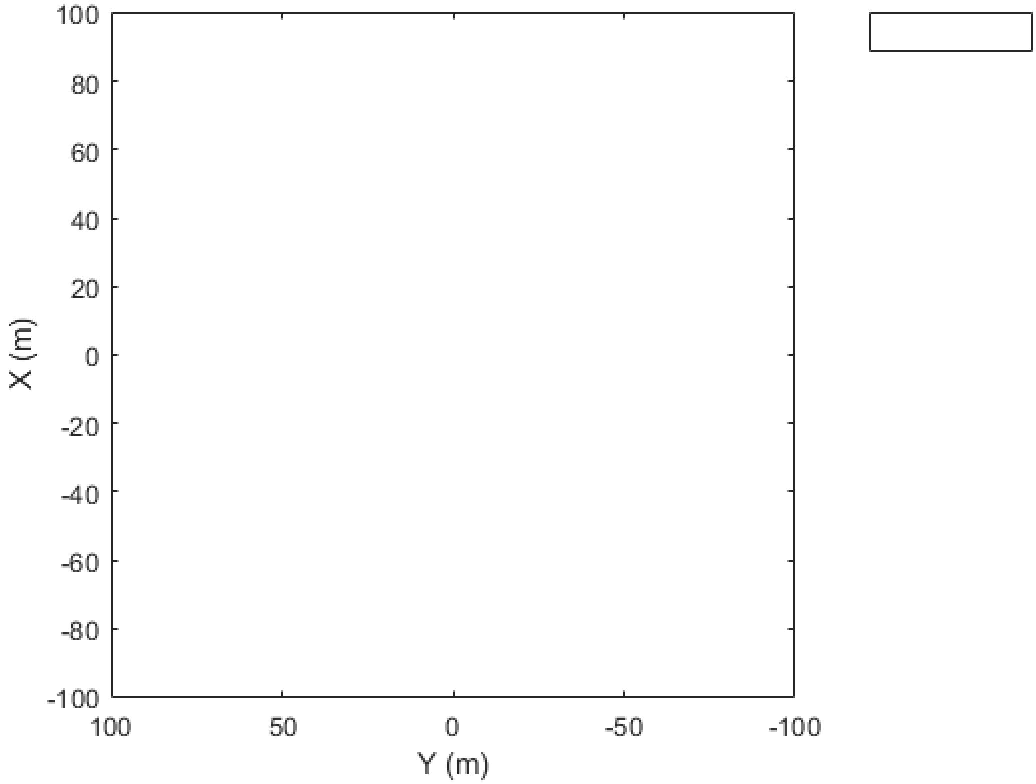


Plot Radar Coverage Areas at Four Corners of Vehicle

Create radar coverage areas at the four corners of a vehicle. The sensors have a maximum range of 90 meters and a field of view of 30 degrees.

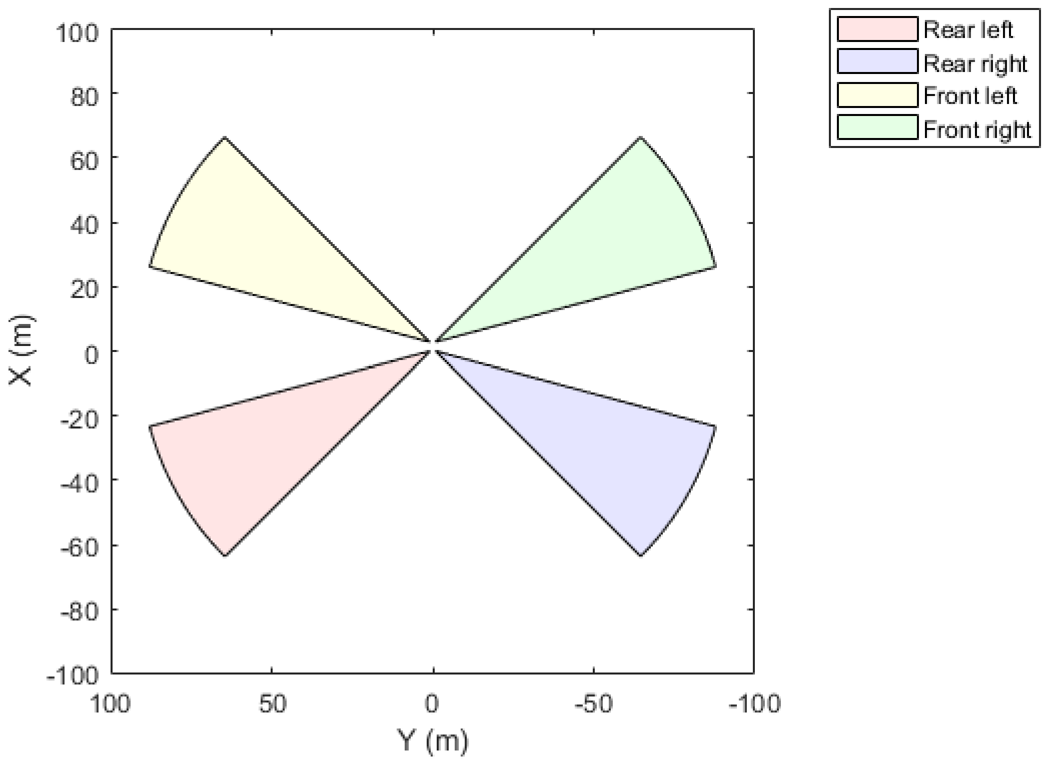
Create a bird's-eye plot.

```
bep = birdsEyePlot('XLim',[-100, 100],'YLim',[-100, 100]);
```



Set the positions, range, orientation, and field of view for the sensors. Plot the coverage areas.

```
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

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);
```



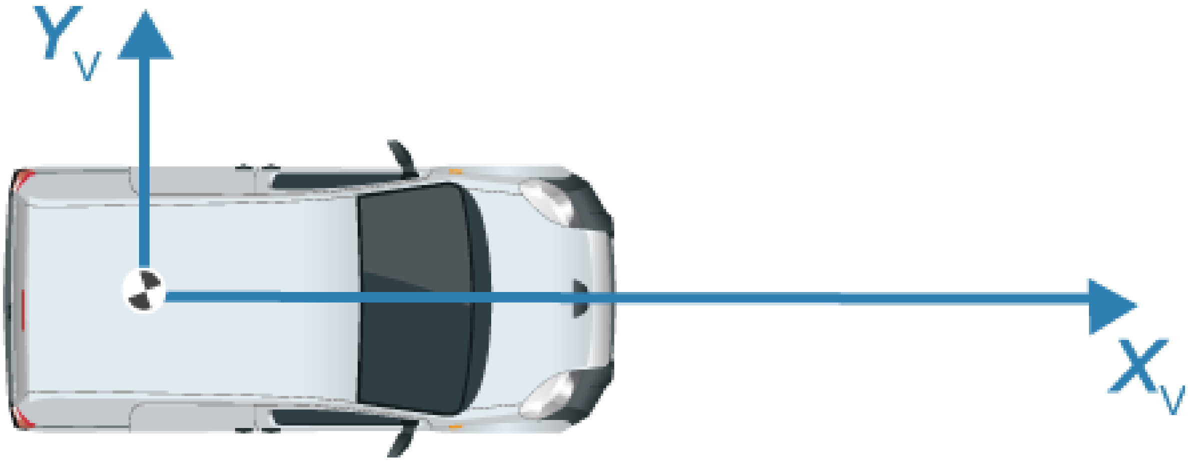
Input Arguments

`caPlotter` — Bird’s-eye plot of coverage area
figure

Bird’s-eye plot of coverage area, specified as a figure plot.

`position` — Position of sensor on vehicle
[*xorigin* *yorigin*] row vector

Position of sensor on vehicle, specified as a [*xorigin yorigin*] row vector. *xorigin* corresponds to the distance in front of the center of the vehicle. *yorigin* corresponds to the distance to the left of the origin of the vehicle, which is the center of the rear axle.



Vehicle Coordinate System

`range` — Sensor coverage distance
scalar in meters

Sensor coverage distance, specified as a scalar in meters.

`orientation` — Heading angle of coverage area
degrees

Heading angle of coverage area, specified in degrees, from the *X*-axis. The orientation is measured in a positive counterclockwise direction (to the left.)

`fieldOfView` — Sensor coverage angle
degrees

Sensor coverage angle, specified in degrees.

See Also

Functions
`birdsEyePlot` | `coverageAreaPlotter`

Introduced in R2017a

plotDetection

Plot a set of object detections

Syntax

```
plotDetection(detPlotter,positions)
plotDetection(detPlotter,positions,velocities)
plotDetection(detPlotter,positions, ___ ,labels)
```

Description

`plotDetection(detPlotter,positions)` returns a plot of object detections. Use `detectionPlotter` to obtain the `detPlotter` figure.

To remove all detections associated with this plotter, call `clearData` with a handle to the detection plotter as its argument.

`plotDetection(detPlotter,positions,velocities)` additionally specifies the detection velocities.

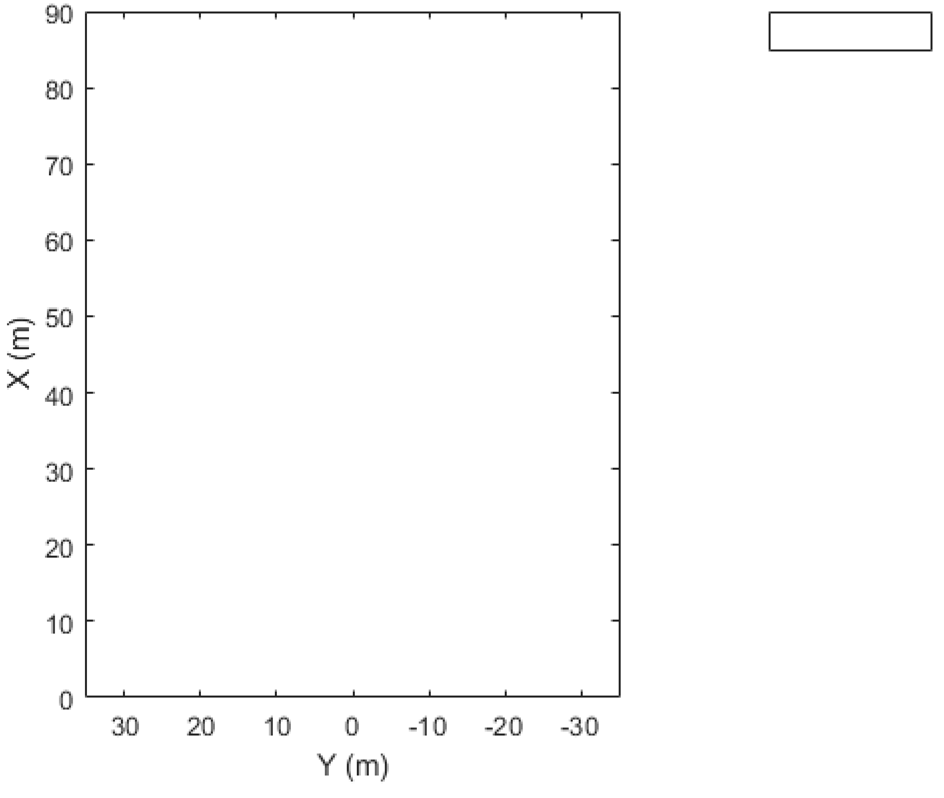
`plotDetection(detPlotter,positions, ___ ,labels)` additionally specifies labels for the detections.

Examples

Create and Display a Bird's-Eye Plot

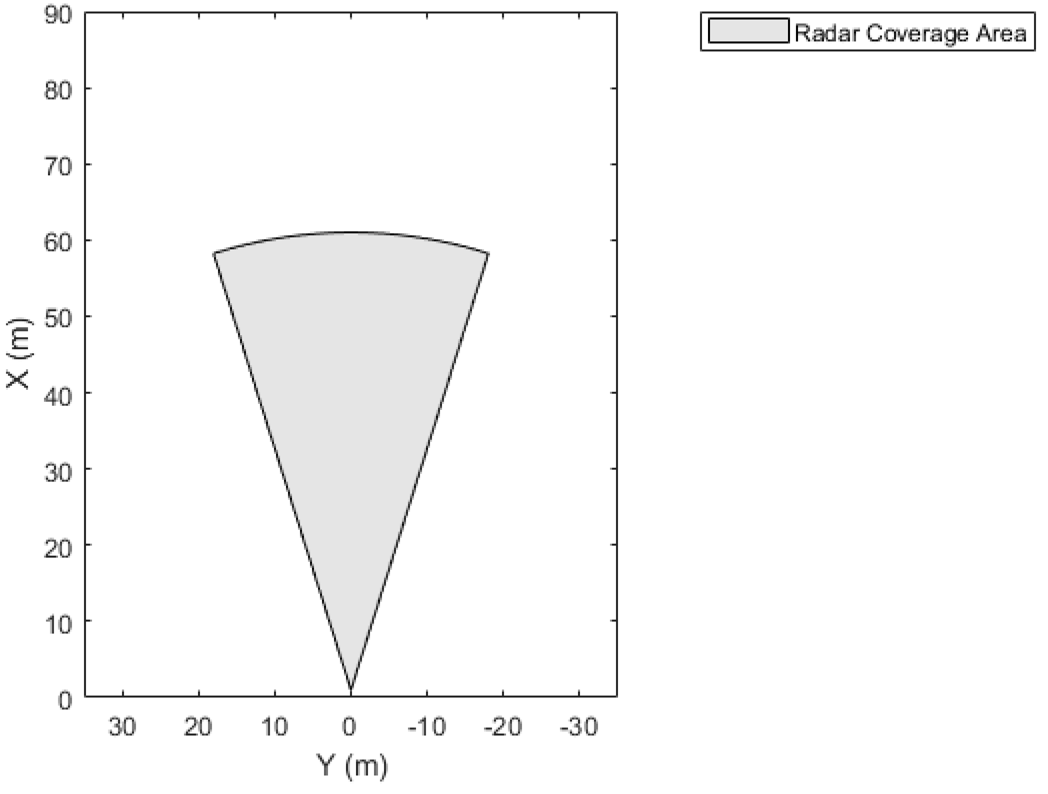
Create the bird's-eye plot.

```
bep = birdsEyePlot('XLim',[0,90],'YLim',[-35,35]);
```



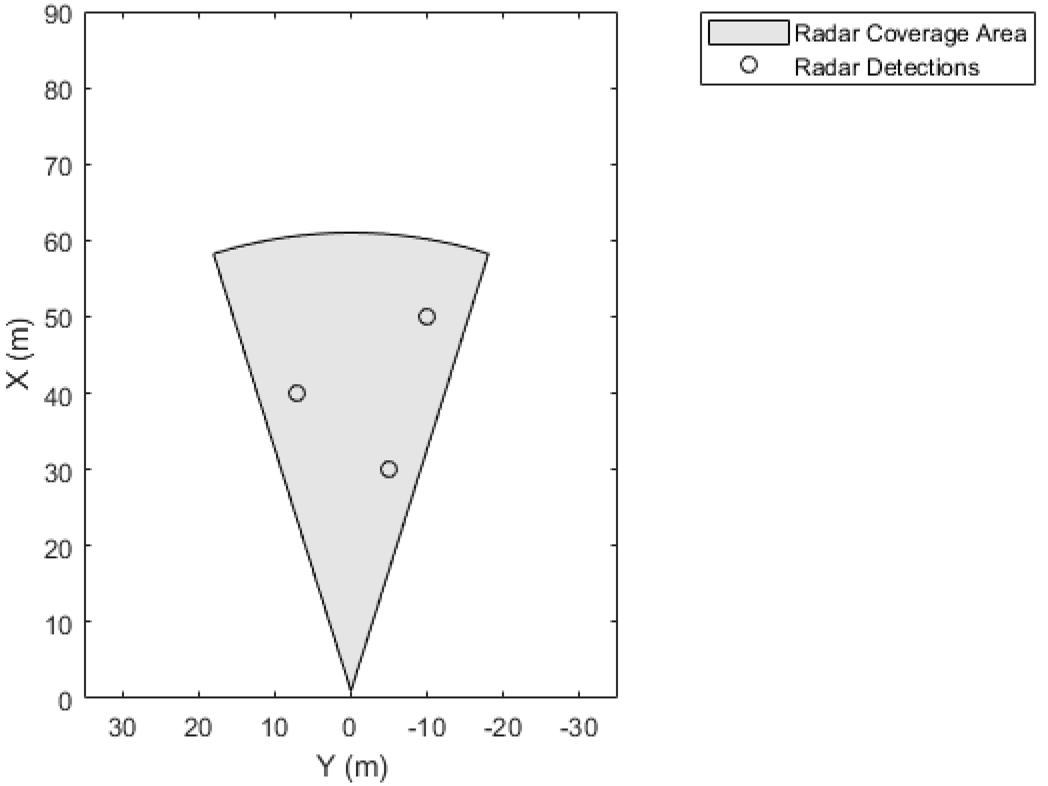
Display a coverage area with a field of view of 35 degrees and a range of 60 meters

```
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]);
```



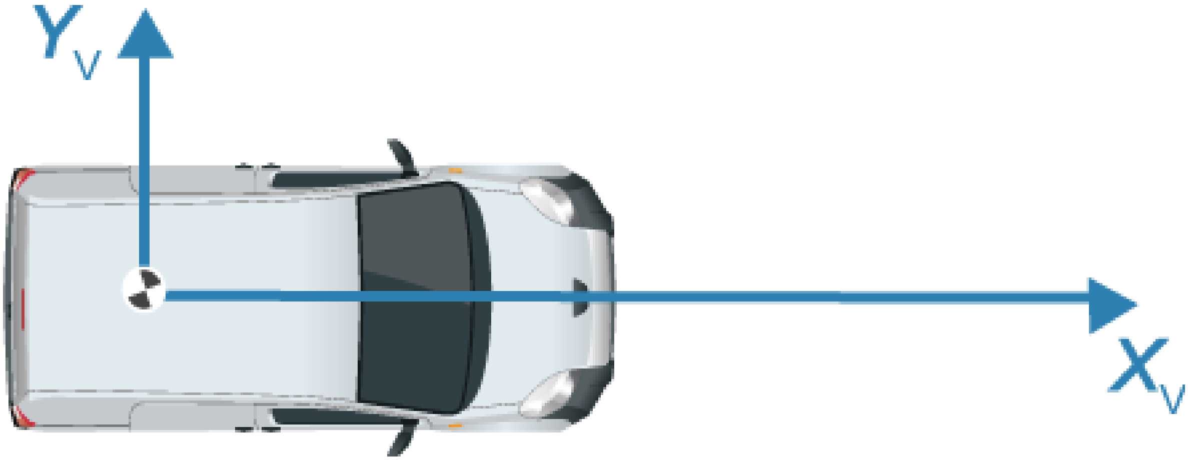
Input Arguments

`detPlotter` — Detection plotter to use for bird's-eye view display
figure

Detection plotter to use for bird's-eye view display, specified as a figure.

`positions` — Positions of detected objects
M-by-2 matrix

Positions of detected objects, specified as an *M*-by-2 matrix of (*x*, *y*) positions. 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.



Vehicle Coordinate System

`velocities` — Velocity of detections
M-by-2 matrix

Velocity of detections, specified as an*M*-by-2 matrix.

`labels` — Detection labels
cell vector

Detection labels, specified as a cell vector of length *M*. The labels correspond to the locations in the positions matrix. If you do not specify labels, they are omitted. You can use the `clearData` function to remove all annotations and labels associated with the detection plotter.

```
clearData(detPlotter)
```

See Also

Functions
`birdsEyePlot` | `detectionPlotter`

Introduced in R2017a

plotLaneBoundary

Plot lane boundary for bird’s-eye plot

Syntax

```
plotLaneBoundary(lbPlotter,boundaryCoordList)
plotLaneBoundary(lbPlotter,laneBoundary)
```

Description

`plotLaneBoundary(lbPlotter,boundaryCoordList)` displays lane boundaries from a boundary coordinate list in a bird’s-eye plot. Use `laneBoundaryPlotter` to obtain the `lbPlotter` figure.

To remove all lane boundaries associated with this plotter, call `clearData` with a handle to the lane boundary plotter as its argument.

`plotLaneBoundary(lbPlotter,laneBoundary)` displays lane boundaries from an object or vector of lane boundary objects.

Examples

Create and Plot Road Boundaries

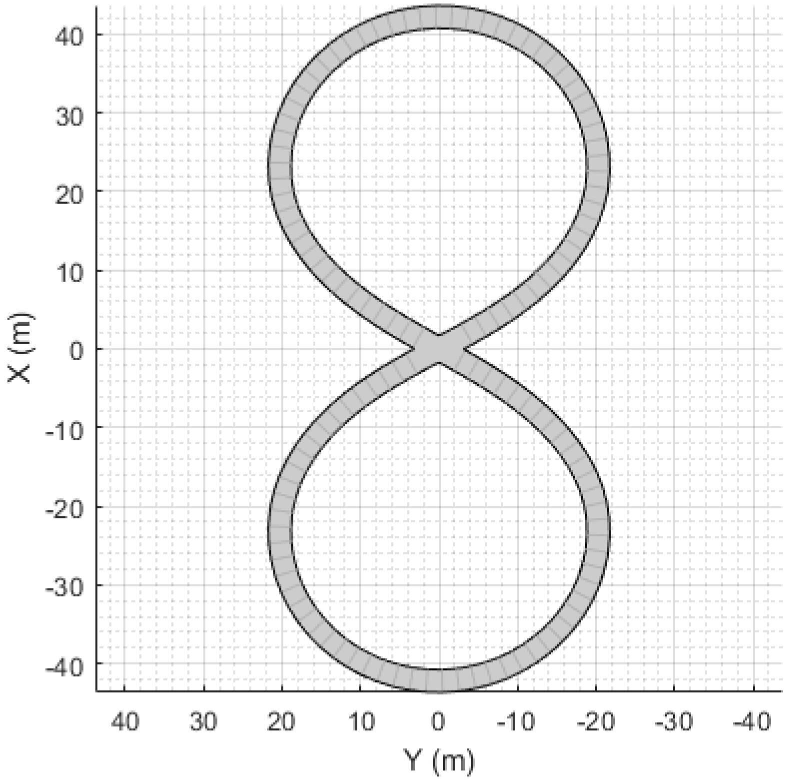
Create a driving scenario containing a figure-8 road specified in scenario coordinates. Convert the coordinates to an actor's ego coordinate system.

```
s = drivingScenario;
```

Add the figure-8 road to the scenario.

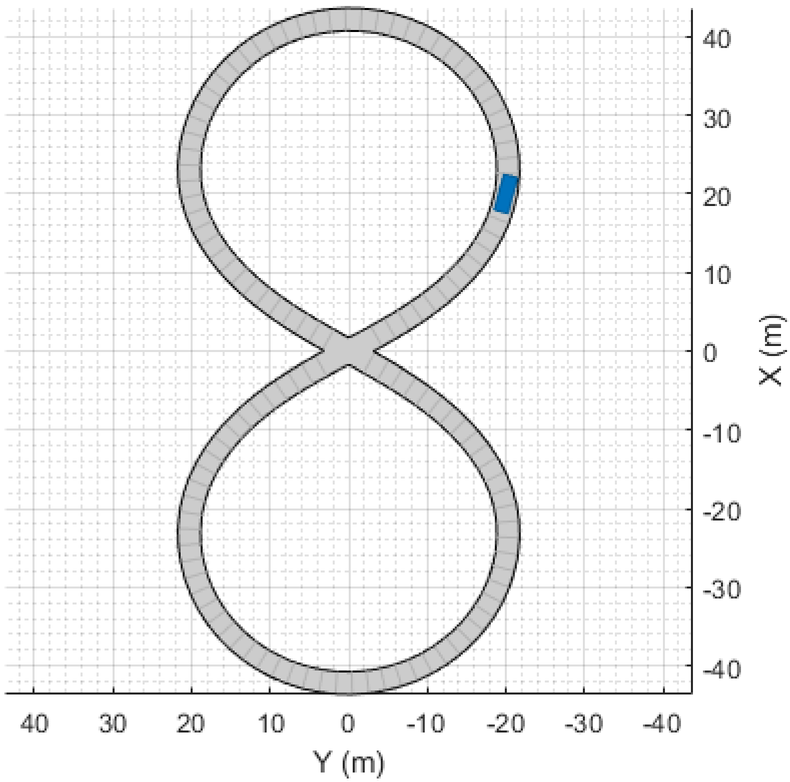
```
roadCenters = [ 0  0  1
    20 -20  1
    20  20  1
```

```
    -20 -20  1
    -20  20  1
    0   0  1];

roadWidth = 3;
bankAngle = [0 15 15 -15 -15 0];
road(s,roadCenters,roadWidth,bankAngle);
plot(s)
```



Add the ego actor at coordinates (20,-20), oriented at 30 degrees yaw angle with respect to scenario coordinates.

```
ego = actor(s,'Position',[20 -20 0],'Yaw',-15);
```



Obtain the road boundaries in scenario coordinates using the `roadBoundaries` method with the scenario specified as the input argument.

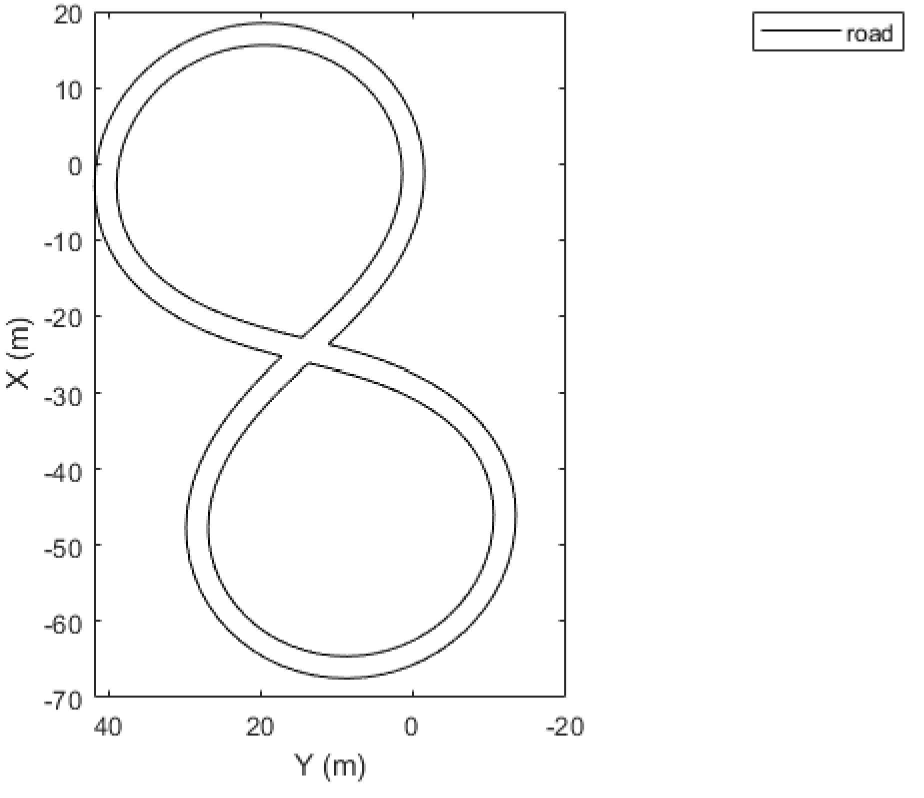
```
rbScenario = roadBoundaries(s);
```

Obtain the road boundaries in ego actor coordinates using the `roadBoundaries` method with the ego actor specified 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)
```

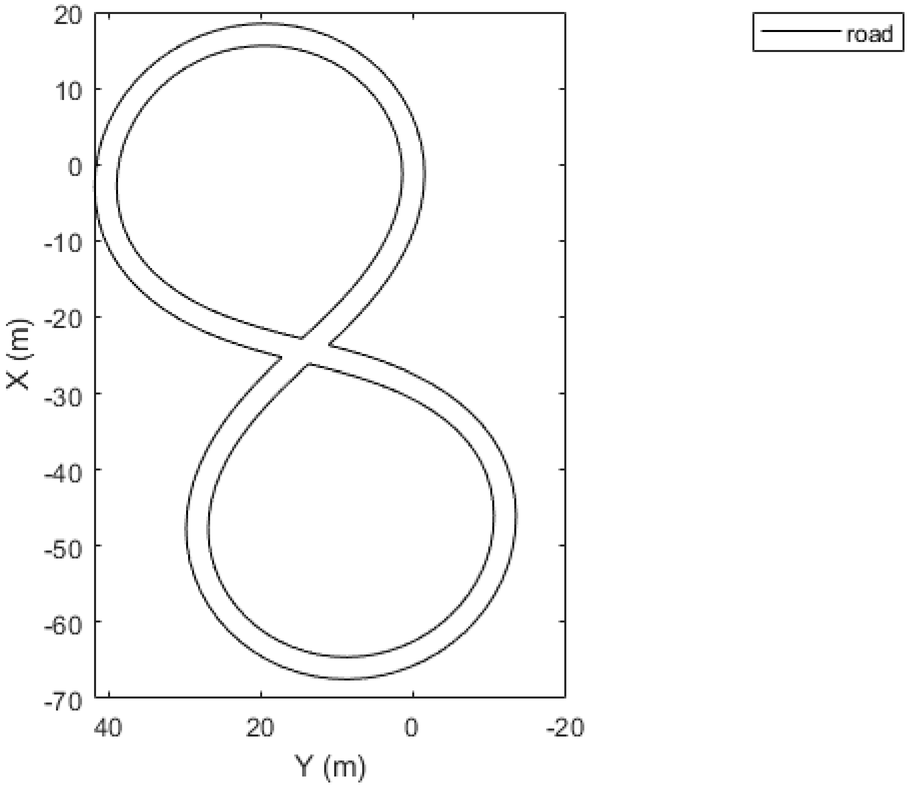


Obtain the road boundaries in ego actor coordinates using the `roadBoundariesToEgo` method.

```
rbEgo2 = driving.scenario.roadBoundariesToEgo(rbScenario,ego);
```

Display the result on a bird's-eye plot.

```
bep = birdsEyePlot;
lbp = laneBoundaryPlotter(bep,'DisplayName','road');
plotLaneBoundary(lbp, {rbEgo2})
```



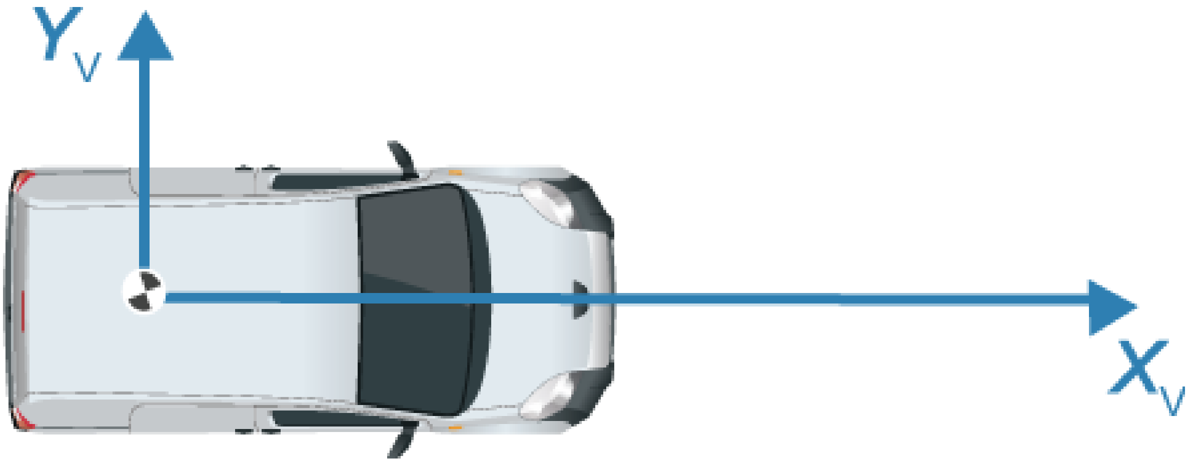
Input Arguments

`lbPlotter` — Lane boundary plotter
figure

Lane boundary plotter, specified as a figure.

`boundaryCoordList` — Coordinates for a boundary lane
cell array of *M*-by-2 matrices

Coordinates for a boundary lane, specified as a cell array of *M*-by-2 matrices. The first and second column of each matrix 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.



Vehicle Coordinate System

`laneBoundary` — Land boundary data
cell array of vectors | `landBoundary` objects

Land boundary data, specified as a cell array of vectors or as a vector of `landBoundary` objects. Each element of the cell array contains a vector. Each vector contains an *N*-by-2 matrix of (*x*,*y*) coordinates in two columns. You can provide an *N*-by-3 matrix, but `birdsEyePlot` ignores the third column, which represents height.

See Also

Functions
`birdsEyePlot` | `laneBoundaryPlotter`

Introduced in R2017a

plotLaneMarking

Plot lane markings on bird's-eye plot

Syntax

```
plotLaneMarking(lmPlotter,lmv,lmf)
```

Description

`plotLaneMarking(lmPlotter,lmv,lmf)` plots lane markings on a bird's-eye plot using the plotter, `lmPlotter`, the lane marking vertices, `lmv`, and the lane marking faces, `lmf`. Use `laneMarkingPlotter` to obtain the `lmPlotter` object. You can use `laneMarkingVertices` to generate lane marking vertices and faces.

To remove all lane marking vertices and faces associated with this plotter, call `clearData` with `lmPlotter` as its argument.

Examples

Plot Lane Markings in Car and Pedestrian Scenario

Construct a driving scenario containing a car and pedestrian on a straight road. Then, create and display lane markings in a bird's-eye plot.

Create an empty driving scenario.

```
sc = drivingScenario;
```

Construct a straight road segment 25 m in length with two travel lanes in one direction.

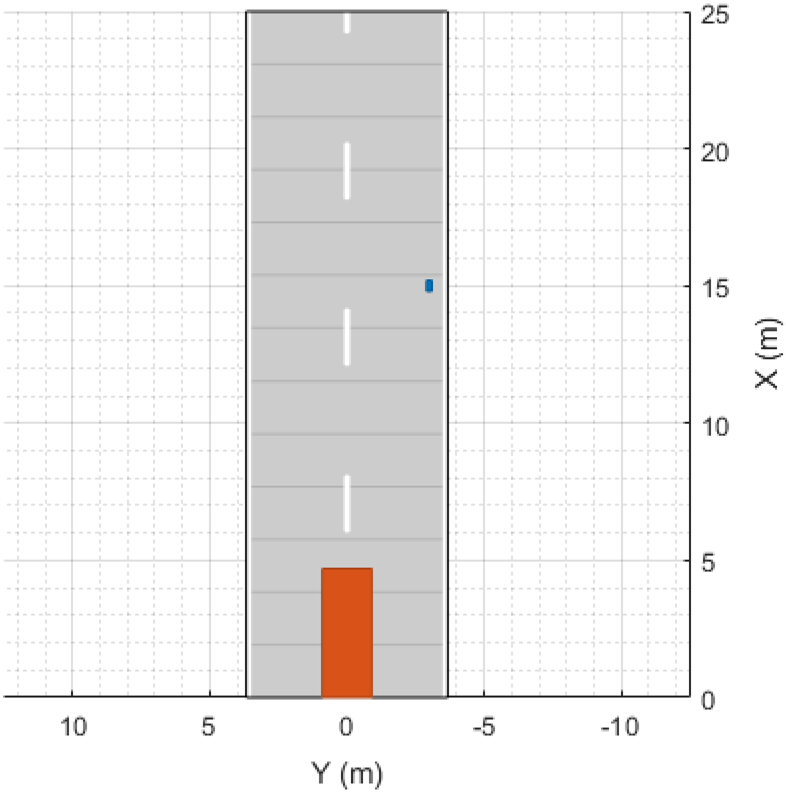
```
lm = [laneMarking('Solid')
    laneMarking('Dashed','Length',2,'Space',4)
    laneMarking('Solid')];
l = lanespec(2,'Marking',lm);
road(sc, [0 0 0; 25 0 0],'Lanes',l);
```

Add a pedestrian crossing the road at 1 m/s and a car following the road at 10 m/s.

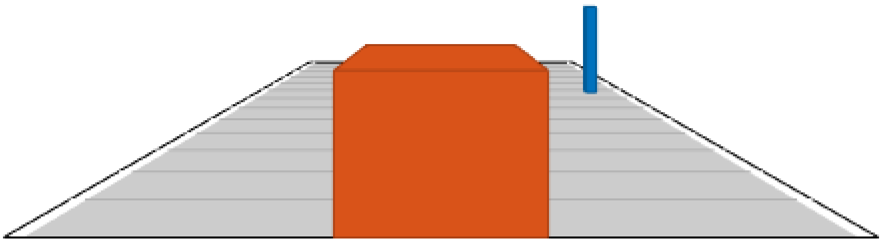
```
ped = actor(sc, 'Length', 0.2, 'Width', 0.4, 'Height', 1.7);
car = vehicle(sc);
trajectory(ped,[15 -3 0; 15 3 0], 1);
trajectory(car,[car.RearOverhang 0 0; 25-car.Length+car.RearOverhang 0 0], 10);
```

Display the scenario and corresponding chase plot.

```
plot(sc)
```




```
chasePlot(car)
```

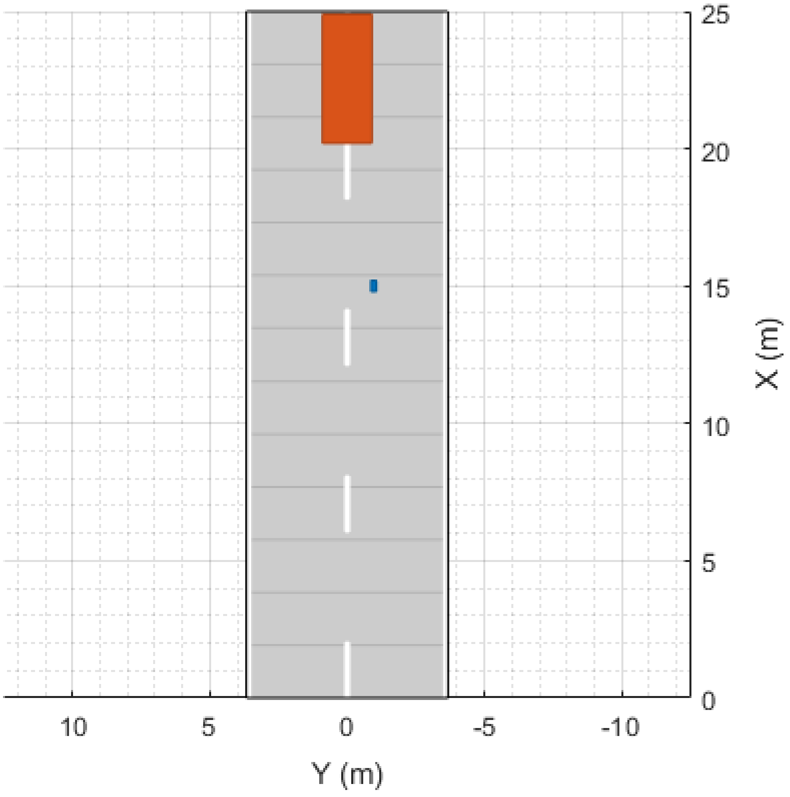


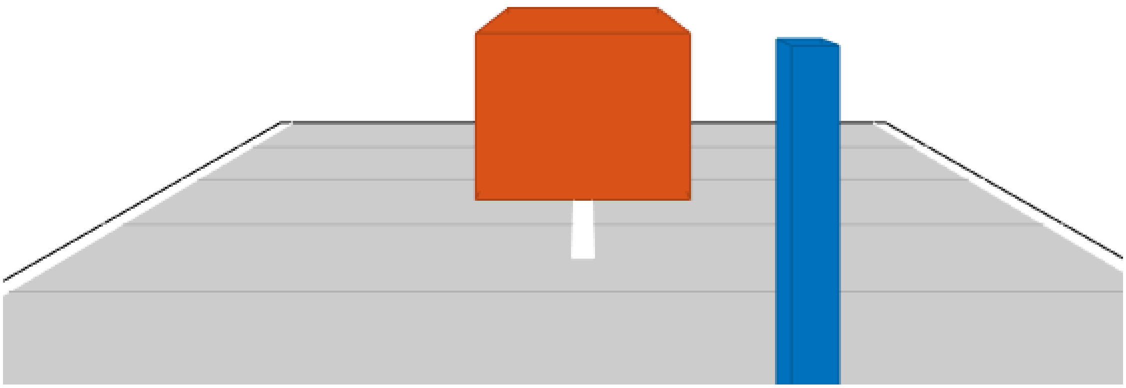
Run the simulation.

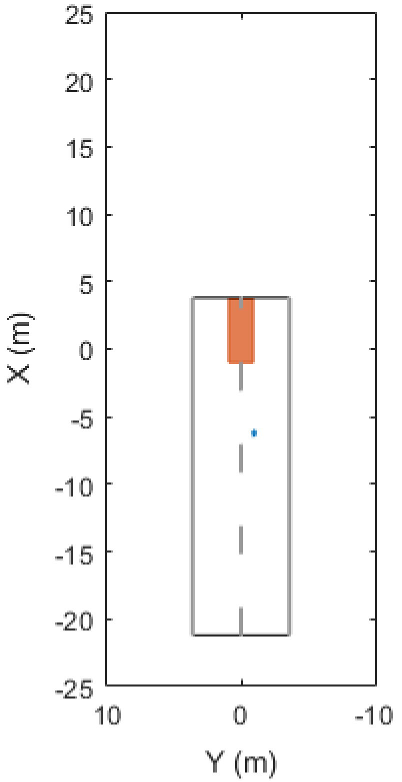
- Create the bird's eye plot and add an outline plotter, a lane boundary plotter and lane marking plotter.
- Get the road boundaries and target outlines.
- Get lane marking vertices and faces.
- Plot the boundaries and lane markers.
- Run the simulation loop.

```
bep = birdsEyePlot('XLim',[-25 25],'YLim',[-10 10]);
olPlotter = outlinePlotter(bep);
lbPlotter = laneBoundaryPlotter(bep);
```

```
lmPlotter = laneMarkingPlotter(bep,'DisplayName','Lanes');
legend('off');
while advance(sc)
    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
```







Input Arguments

`lmPlotter` — Lane marking plotter
`laneMarkingPlotter` object

Lane marking plotter, specified as a `laneMarkingPlotter` object.

`lmv` — Lane marking vertices
real-valued *L*-by-3 matrix

Lane marking vertices, specified as a real-valued *L*-by-3 matrix. Each row of the lane marking matrix represents the *x*, *y*, and *z* coordinates of a vertex. The plotter only uses the *x* and *y* coordinates.

`lmf` — Lane marking faces
real-valued matrix

Lane marking faces, specified as a real-valued matrix. Each row of the matrix is a face that defines the connection between vertices for one lane marking.

See Also

Functions
`birdsEyePlot` | `laneMarkingPlotter` | `laneMarkingVertices`

Introduced in R2018a

plotOutline

Plot object outlines

Syntax

```
plotOutline(olPlotter,positions,yaw,length,width)
plotOutline( ___ ,Name,Value)
```

Description

`plotOutline(olPlotter,positions,yaw,length,width)` plots rectangular outlines of the objects stored in a bird's-eye-view plotter. Specify the position of each rectangle, the angle of rotation (`yaw`), and the length and width of each rectangle. To obtain the `olPlotter` input, use `outlinePlotter`.

To remove all outlines associated with this plotter, call `clearData` with a handle to the outline plotter as its argument.

From a given driving scenario, use `targetOutlines` to get the dimensions for all actors in the scene. 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.

`plotOutline( ___ ,Name,Value)` specifies additional options using one or more `Name,Value` pair arguments.

Examples

Plot Outlines of Targets in Bird's-Eye Plot

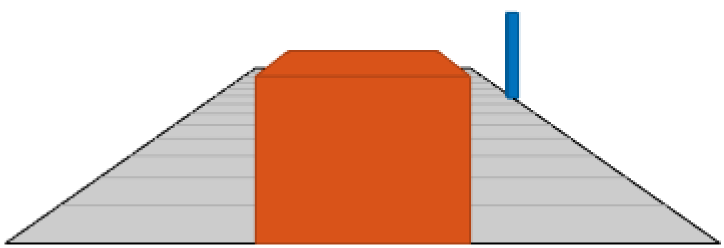
Create a driving scenario. Construct a 25 m road segment, add a pedestrian and a vehicle, and specify their trajectories to follow. The pedestrian crosses the road at 1 m/s. The vehicle drives along the road at 10 m/s.

```
s = drivingScenario;
```

```
road(s, [0 0 0; 25 0 0]);

p = actor(s,'Length',0.2,'Width',0.4,'Height',1.7);
v = vehicle(s);

trajectory(p,[15 -3 0; 15 3 0], 1);
trajectory(v,[v.RearOverhang 0 0; 25-v.Length+v.RearOverhang 0 0], 10);
```

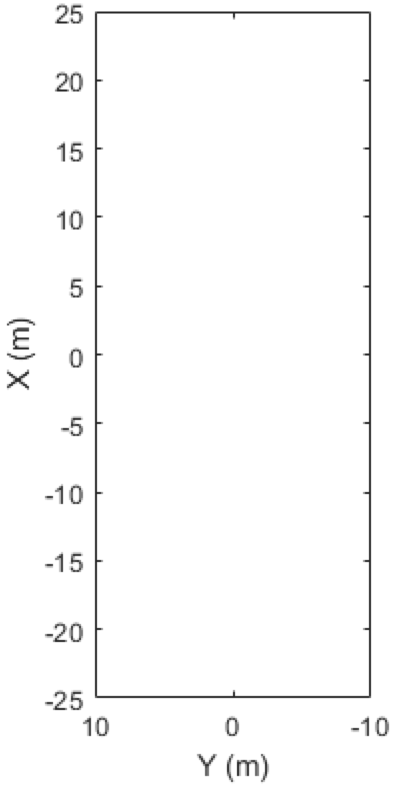
Add an egocentric plot for the vehicle

```
chasePlot(v,'Centerline','on')
```



Create a bird's-eye plot.

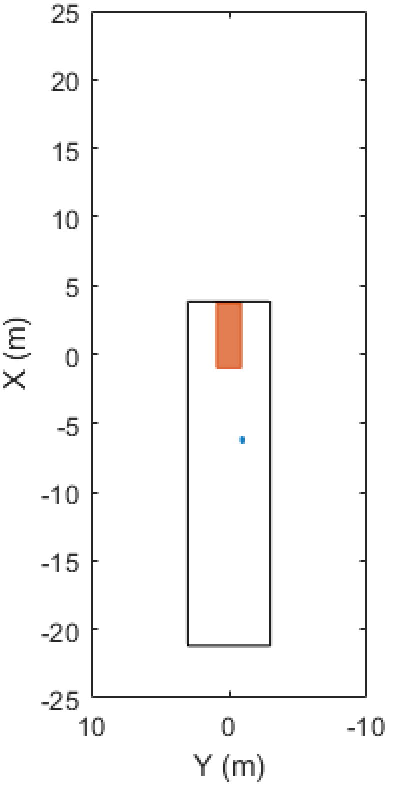
```
bep = birdsEyePlot('XLim',[-25 25],'YLim',[-10 10]);
olPlotter = outlinePlotter(bep);
lbPlotter = laneBoundaryPlotter(bep);
legend('off')
```

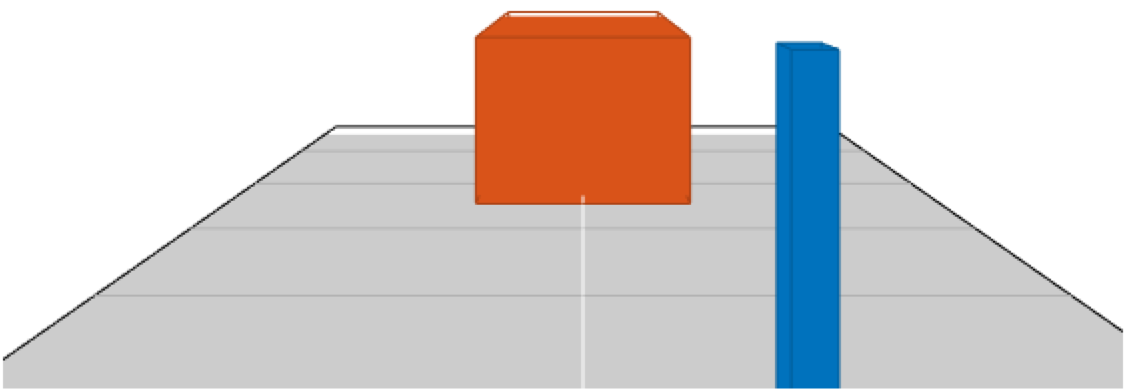


Start the simulation loop. Update the plotter with outlines for the targets.

```
while advance(s)
   % get 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
```





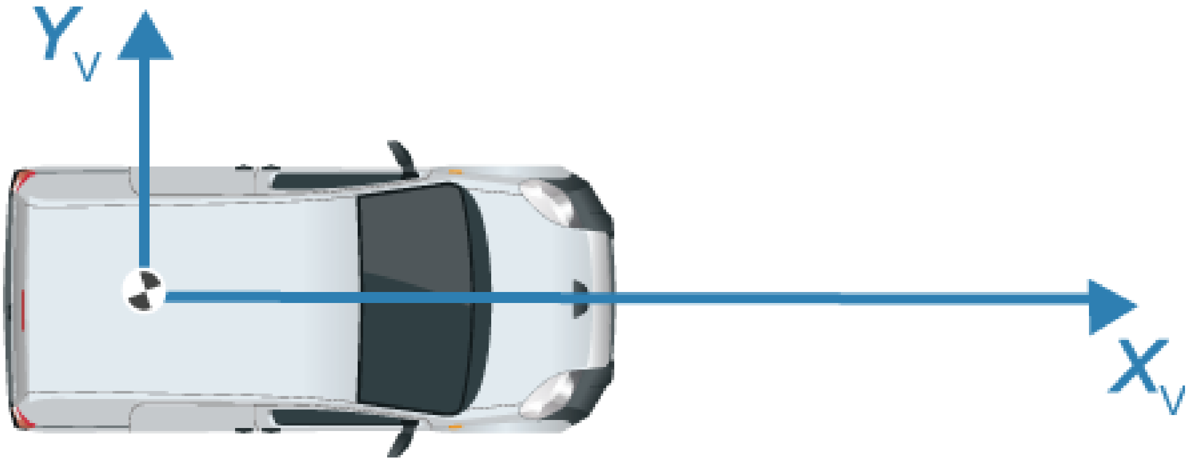
Input Arguments

`olPlotter` — Outline plotter
plotter object

Outline plotter to use for the bird's-eye plot, returned as a plotter object. To create the object, use `outlinePlotter`.

`positions` — Positions of detected objects
M-by-2 matrix

Positions of detected objects, specified as an *M*-by-2 matrix of (*x*, *y*) positions, where *M* is the number of 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.



Vehicle Coordinate System

`yaw` — Angles of rotation
M-element vector

Angles of rotation for each outline, specified as an *M*-element vector, where *M* is the number of objects.

`length` — Lengths of outlines
M-element vector

Length of outlines, specified as an *M*-element vector, where *M* is the number of objects.

`width` — Widths of outlines
M-element vector

Widths of outlines, specified as an *M*-element 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 vector

Rotational center of rectangles relative to origin, specified as the comma-separated pair consisting of `'OriginOffset'` and an *M*-by-2 vector, where *M* is the number of objects. Each row corresponds to the rotational center about which to rotate a rectangle, specified as an *xy*-displacement from the geometrical center of the rectangle.

`Color` — Outline color
RGB triplet

Outline color, specified as the comma-separated pair consisting of `'Color'` and an RGB triplet. If this argument is not specified, the function uses the default colormap.

Example: `'Color',[0 0.5 1]`

See Also

Functions
`birdsEyePlot` | `outlinePlotter`

Introduced in R2017b

plotPath

Plot lane boundary for bird's-eye plot

Syntax

`plotPath(pPlotter,pathCoordList)`

Description

`plotPath(pPlotter,pathCoordList)` returns lane boundaries to display from a boundary coordinate list in a bird's-eye plot. Use `pathPlotter` to obtain the `lbPlotter` figure.

To remove all paths associated with this plotter, call `clearData` with a handle to the path plotter as its argument.

Examples

Plot Path of Ego Vehicle

Create a 3-meter-wide lane.

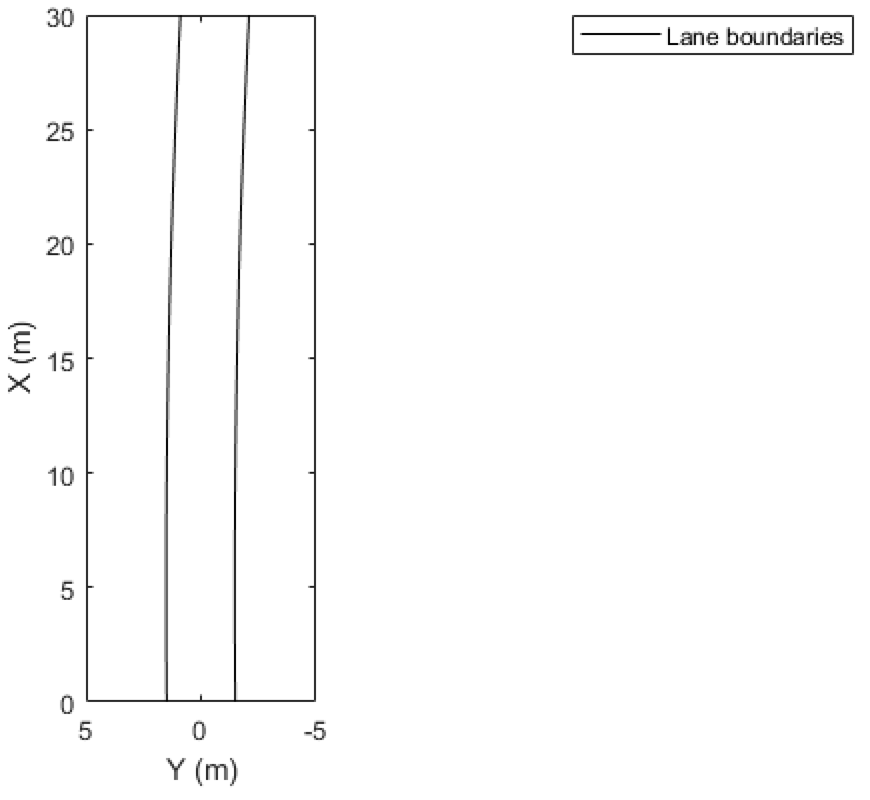
```
lb = parabolicLaneBoundary([-0.001,0.01,1.5]);
rb = parabolicLaneBoundary([-0.001,0.01,-1.5]);
```

Compute the model manually up to 30 meters ahead in the lane.

```
xWorld = (0:30)';
yLeft = computeBoundaryModel(lb,xWorld);
yRight = computeBoundaryModel(rb,xWorld);
```

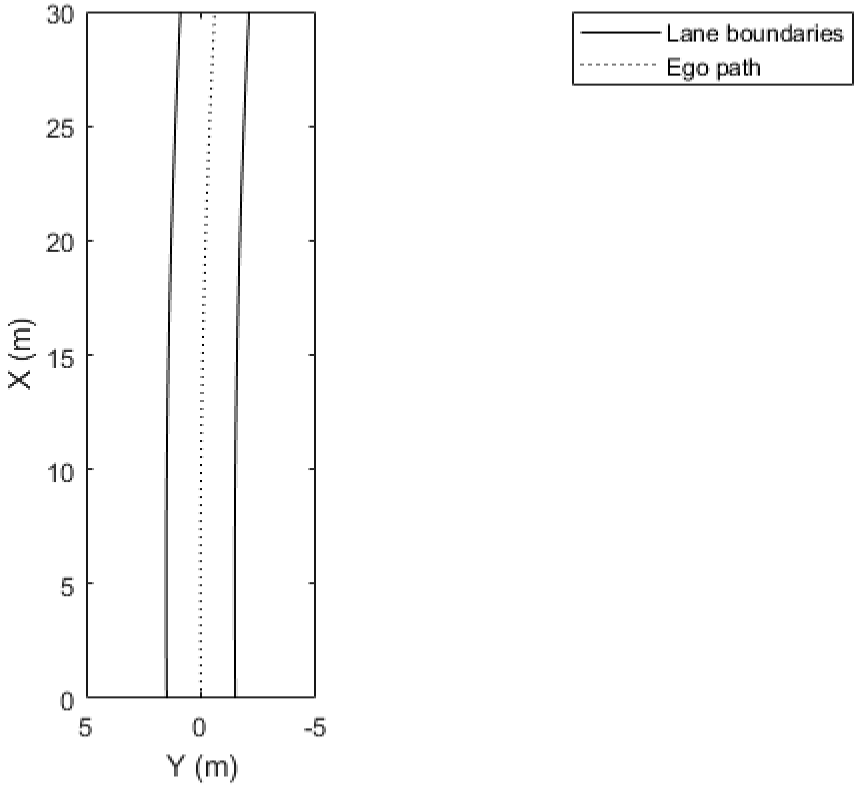
Create a bird's-eye plot and plot the lane information.

```
bep = birdsEyePlot('XLimits',[0 30],'YLimits',[-5 5]);
lanePlotter = laneBoundaryPlotter(bep,'DisplayName','Lane boundaries');
plotLaneBoundary(lanePlotter,{[xWorld,yLeft],[xWorld,yRight]});
```



Plot the path of an ego vehicle that travels through the center of the lane.

```
yCenter = (yLeft + yRight)/2;
egoPathPlotter = pathPlotter(bep,'DisplayName','Ego path');
plotPath(egoPathPlotter,{[xWorld,yCenter]});
```



Input Arguments

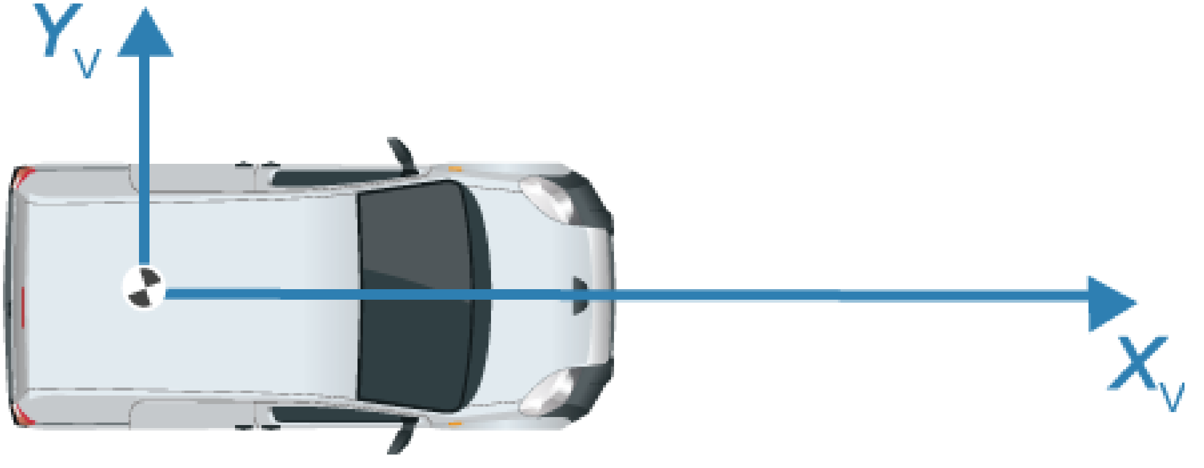
`pPlotter` — Path plotter
figure

Lane boundary plotter, specified as a figure.

`pathCoordList` — Coordinates for paths
cell array of *M*-by-2 matrices

Coordinates for paths, specified as a cell array of *M*-by-2 matrices. The first and second column of each matrix represents the (*x*, *y*) positions of a curve that represent the path.

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.



Vehicle Coordinate System

See Also

Functions
`birdsEyePlot` | `pathPlotter`

Introduced in R2017a

plotTrack

Plot a set of detection tracks

Syntax

```
plotTrack(tPlotter,positions)
plotTrack(tPlotter,positions,velocities)
plotTrack(tPlotter,positions, ___ ,labels)
plotTrack(tPlotter,positions, ___ ,labels,covariances)
```

Description

`plotTrack(tPlotter,positions)` returns a plot of object detection tracks. Use `trackPlotter` to obtain the `tPlotter` figure.

To remove all tracks associated with this plotter, call `clearData` with a handle to the track plotter as its argument.

`plotTrack(tPlotter,positions,velocities)` additionally specifies the detection velocities.

`plotTrack(tPlotter,positions, ___ ,labels)` additionally specifies labels for the detections.

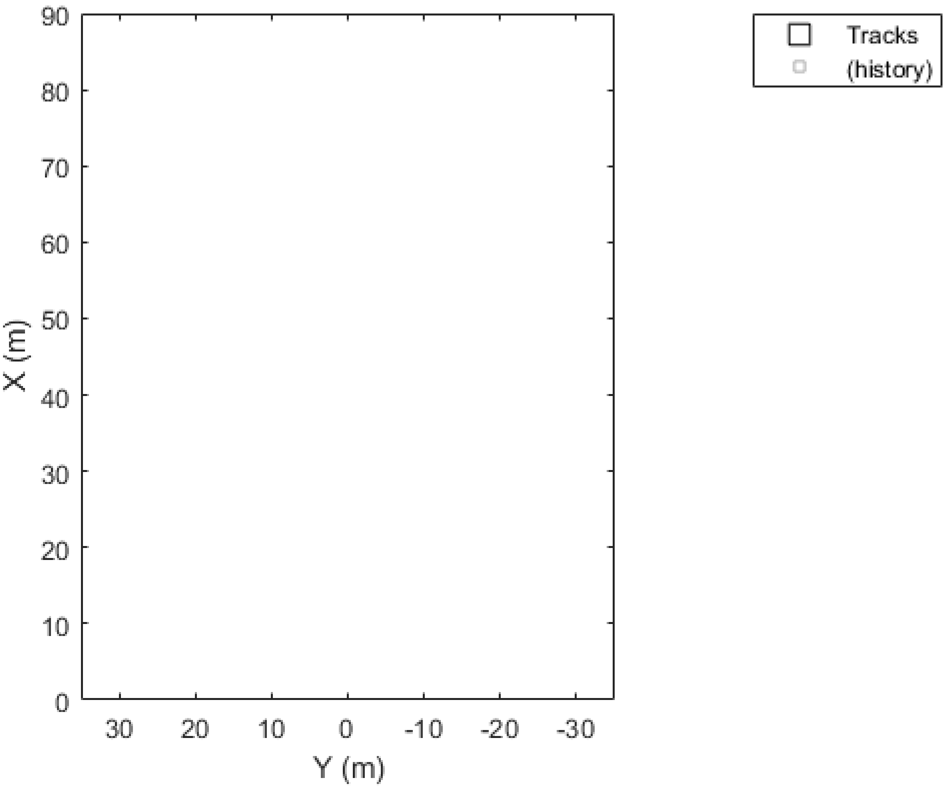
`plotTrack(tPlotter,positions, ___ ,labels,covariances)` additionally specifies covariances of track uncertainties.

Examples

Create Bird's-Eye Plot with Labeled Tracks

Create a bird's-eye plot and a track plotter. Set the plotter to display up to seven history values for each track.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
tPlotter = trackPlotter(bep,'DisplayName','Tracks','HistoryDepth',7);
```

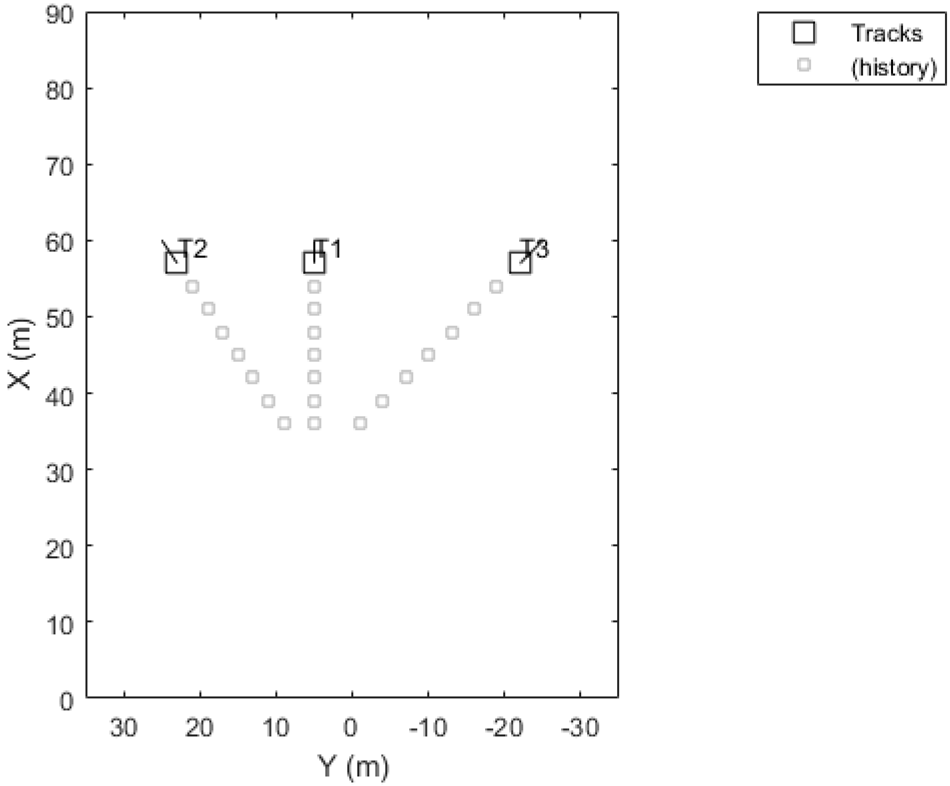


Set the positions, velocities, and labels of each track.

```
positions = [30, 5; 30, 5; 30, 5];
velocities = [3, 0; 3, 2; 3, -3];
labels = {'T1','T2','T3'};
```

Update the tracks for 10 trials, showing the seven history values specified previously.

```
for i=1:10
   plotTrack(tPlotter,positions,velocities,labels);
   positions = positions + velocities;
end
```



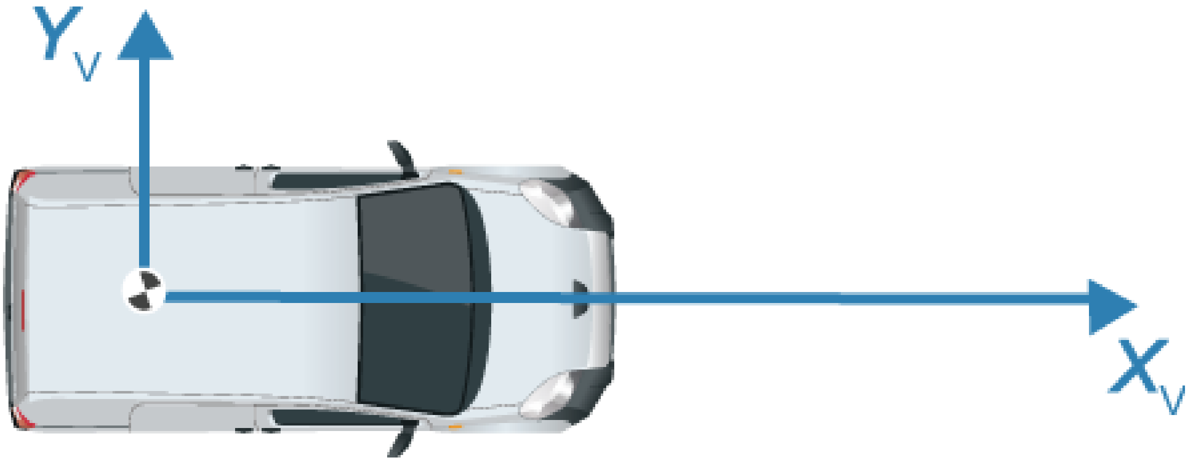
Input Arguments

`tPlotter` — Detection plotter to use for bird’s-eye view display
figure

Detection plotter to use for bird’s-eye view display, specified as a figure.

`positions` — Positions of detected objects
M-by-2 matrix

Positions of detected objects, specified as an *M*-by-2 matrix of (*x*, *y*) positions. 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.



Vehicle Coordinate System

`velocities` — Velocity of detections
M-by-2 matrix

Velocity of detections, specified as an*M*-by-2 matrix.

`labels` — Detection labels
cell vector

Detection labels, specified as a cell vector of length *M*. The labels correspond to the locations in the positions matrix. If you do not specify labels, they are omitted. You can use the `clearData` function to remove all annotations and labels associated with the detection plotter.

```
clearData(tPlotter)
```

`covariances` — Covariances of track uncertainties
2-by-2-by-*M* matrix

Covariances of track uncertainties centered at the track positions, specified as a 2-by-2-by-*M* matrix. The uncertainties are plotted as an ellipse.

See Also

Functions
birdsEyePlot | trackPlotter

Introduced in R2017a

driving.scenario.roadBoundariesToEgo

Convert road boundaries to ego coordinates

Syntax

```
egoRoadboundaries = driving.scenario.roadBoundariesToEgo(
scenarioRoadboundaries,egoActor)
```

Description

`egoRoadboundaries = driving.scenario.roadBoundariesToEgo(scenarioRoadboundaries,egoActor)` converts road boundaries, `scenarioRoadboundaries`, in scenario coordinates to road boundaries, `egoRoadboundaries`, in the coordinate system of the ego actor, `egoActor`.

Examples

Create and Plot Road Boundaries

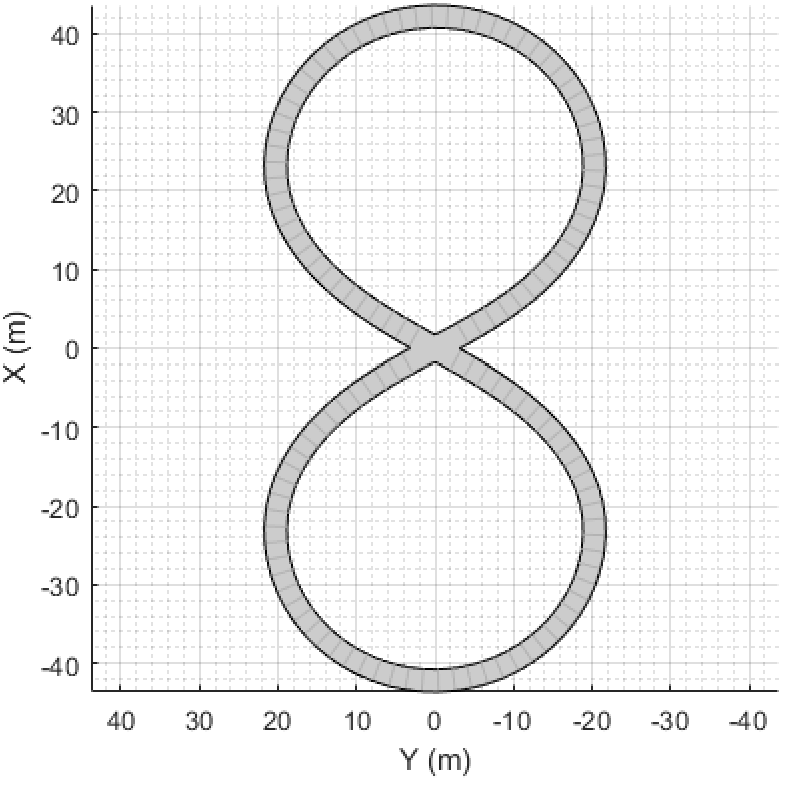
Create a driving scenario containing a figure-8 road specified in scenario coordinates. Convert the coordinates to an actor's ego coordinate system.

```
s = drivingScenario;
```

Add the figure-8 road to the scenario.

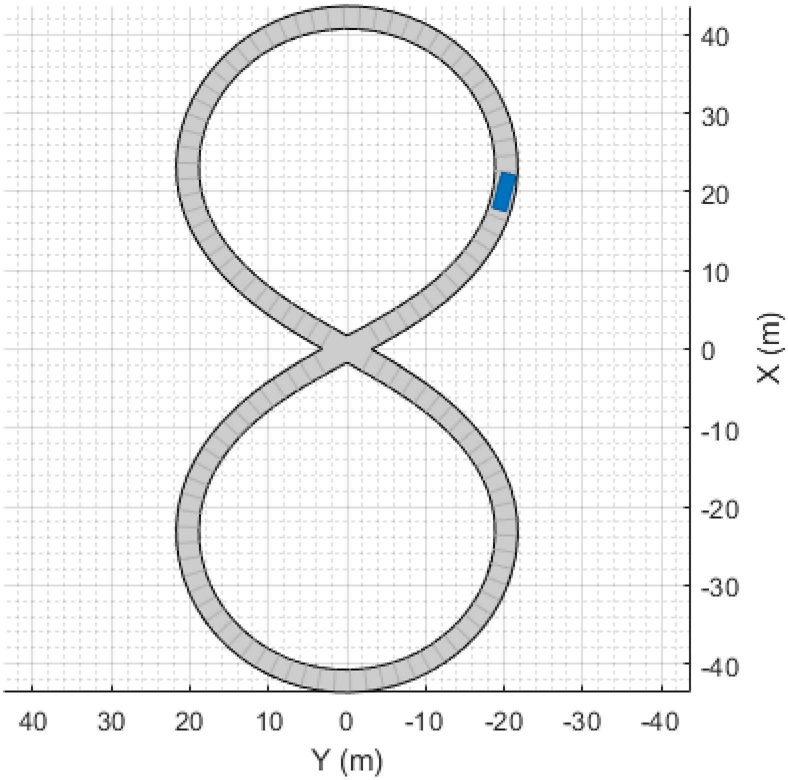
```
roadCenters = [ 0  0  1
    20 -20  1
    20  20  1
    -20 -20  1
    -20  20  1
    0   0  1];

roadWidth = 3;
```

```
bankAngle = [0 15 15 -15 -15 0];
road(s,roadCenters,roadWidth,bankAngle);
plot(s)
```



Add the ego actor at coordinates (20,-20), oriented at 30 degrees yaw angle with respect to scenario coordinates.

```
ego = actor(s,'Position',[20 -20 0],'Yaw',-15);
```



Obtain the road boundaries in scenario coordinates using the `roadBoundaries` method with the scenario specified as the input argument.

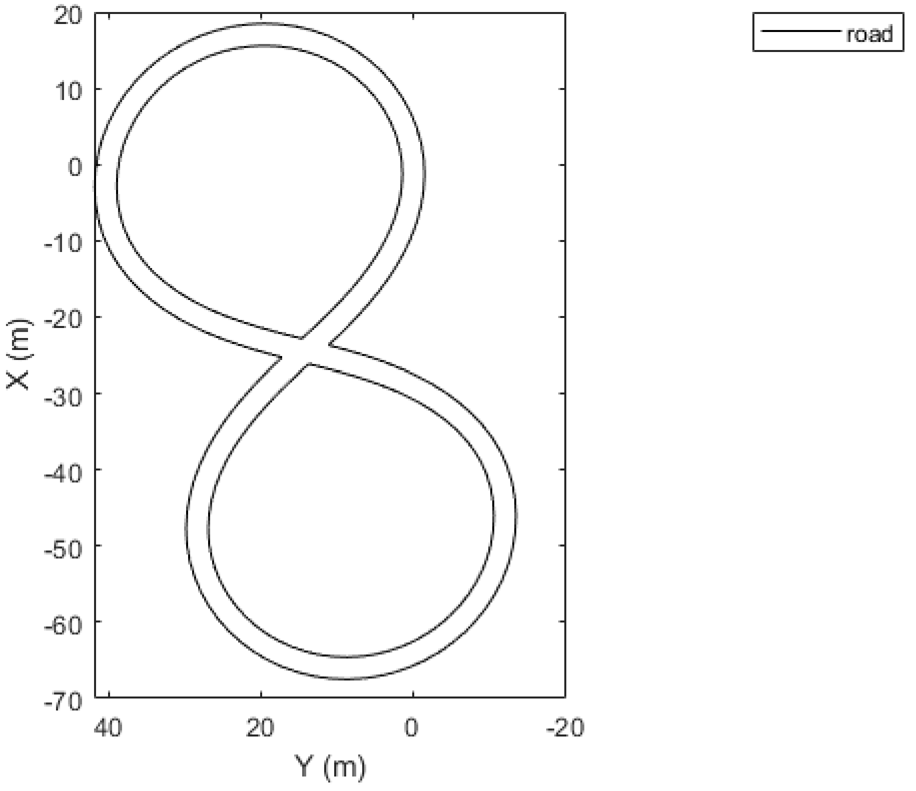
```
rbScenario = roadBoundaries(s);
```

Obtain the road boundaries in ego actor coordinates using the `roadBoundaries` method with the ego actor specified 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)
```

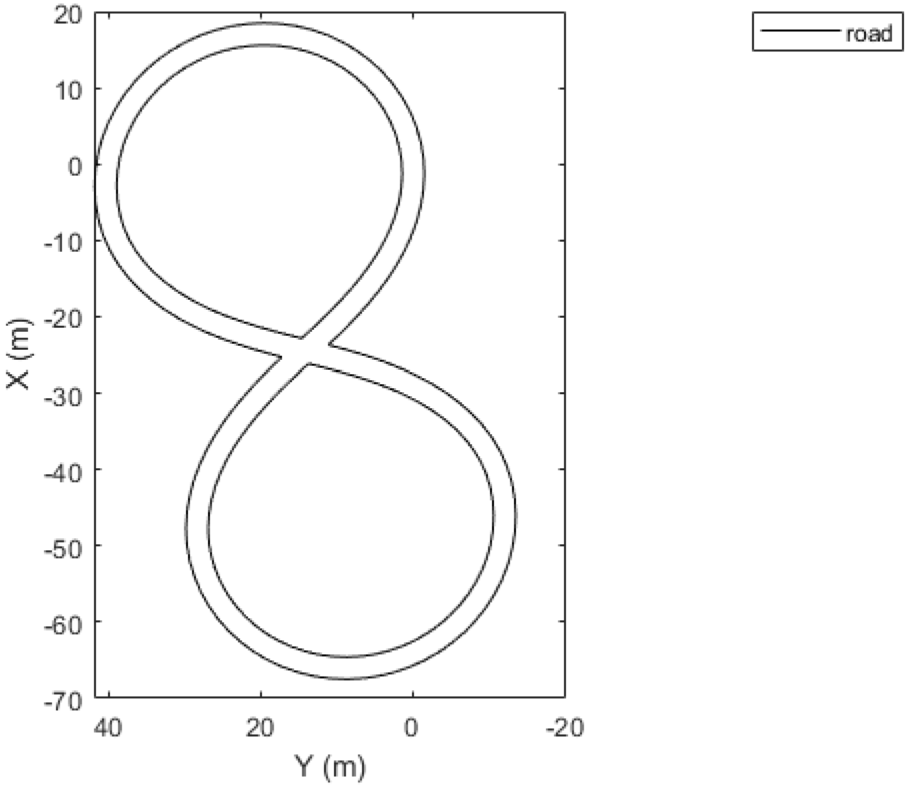


Obtain the road boundaries in ego actor coordinates using the `roadBoundariesToEgo` method.

```
rbEgo2 = driving.scenario.roadBoundariesToEgo(rbScenario,ego);
```

Display the result on a bird's-eye plot.

```
bep = birdsEyePlot;
lbp = laneBoundaryPlotter(bep,'DisplayName','road');
plotLaneBoundary(lbp, {rbEgo2})
```



Input Arguments

`scenarioRoadboundaries` — Road boundaries in scenario coordinates

1-by-*N* cell array

Road boundaries in scenario 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 real-valued matrix. *P* can vary from cell to cell. Units are in meters.

Data Types: `double`

egoActor — ego actor pose
structure

Ego actor pose, specified as a structure. Pose is defined with respect to scenario coordinates. The structure fields:

Field	Description
ActorID	Scenario-defined actor identifier
Position	Position of actor, specified as a real-valued 1-by-3 vector. Units are in meters.
Velocity	Velocity of actor, specified as a real-valued 1-by-3 vector. Units are in meters per second.
Roll	Roll angle of actor, specified as a scalar. Units are in degrees.
Pitch	Pitch angle of actor, specified as a scalar. Units are in degrees.
Yaw	Yaw angle of actor, specified as a scalar. Units are in degrees.
AngularVelocity	Angular velocity of actor, specified as a real-valued 1-by-3 vector. Units are in degrees per second.

Output Arguments

egoRoadboundaries — Road boundaries in ego actor coordinates
real-valued Q-by-3 matrix

Road boundaries in ego actor coordinates, returned as a real-valued Q-by-3 matrix. Q is the number of road boundary point coordinates, 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 had 3 road boundaries of length P_1, P_2, and P_3, then $Q = P_1 + P_2 + P_3 + 2$. Units are in meters.

Data Types: `double`

See Also

`drivingScenario.actor` | `drivingScenario.actorPoses` | `drivingScenario.vehicle` | `targetPoses`

Introduced in R2017a

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')
```

Original Image

Create a bird's-eye-view image.

```
birdsEyeImage = transformImage(birdsEyeConfig,I);
imshow(birdsEyeImage)
```

Convert bird's-eye-view image to grayscale.

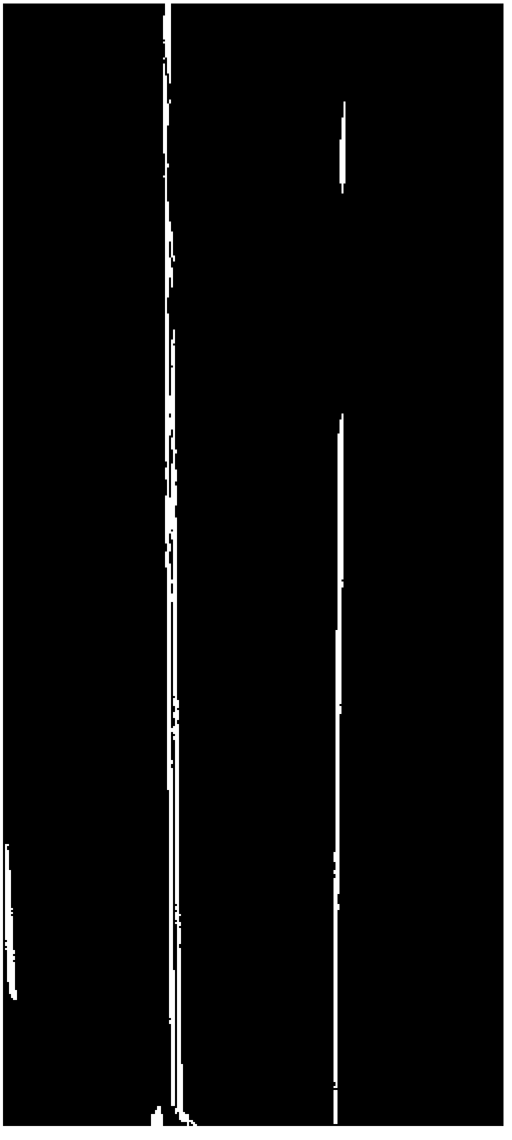
```
birdsEyeImage = rgb2gray(birdsEyeImage);
```

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

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) | nonnegative scalar in the range [0,1]

Sensitivity factor, specified as the comma-separated pair consisting of `'Sensitivity'` and a nonnegative 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.

Definitions

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 System 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.

See Also

`birdsEyeView`

Introduced in R2017a

driving.scenario.TargetsToEgo

Convert actor poses to ego coordinate system

Syntax

```
targetPoses = driving.scenario.TargetsToEgo(actorPoses,egoActor)
```

Description

`targetPoses = driving.scenario.TargetsToEgo(actorPoses,egoActor)` transforms target actor poses, `actorPoses`, from scenario coordinates to the ego-centric coordinate system of the actor, `egoActor`, and returns the transformed poses in `targetPoses` (see “Ego and target actors” on page 3-264).

Examples

Obtain Target Poses in Ego Coordinates

Create a driving scenario containing three vehicles. Find the target poses of two of the vehicles as viewed by the third vehicle. Target poses are returned in the egocentric coordinate system of the third vehicle.

First, create a driving scenario.

```
s = drivingScenario;
```

Then, create the target actors.

```
actor(s,'Position',[10 20 30], ...
    'Velocity',[12 113 14], ...
    'Yaw', 54, ...
    'Pitch', 25, ...
    'Roll', 22, ...
    'AngularVelocity',[24 42 27]);
```

```
actor(s,'Position', [17 22 12], ...
    'Velocity', [19 13 15], ...
    'Yaw', 45, ...
    'Pitch', 52, ...
    'Roll', 2, ...
    'AngularVelocity',[42 24 29]);
```

Add the ego actor.

```
ego = actor(s,'Position', [1 2 3], ...
    'Velocity', [1.2 1.3 1.4], ...
    'Yaw', 4, ...
    'Pitch', 5, ...
    'Roll', 2, ...
    'AngularVelocity', [4 2 7]);
```

Use `actorPoses` to return the poses of all the actors. Pose quantities (position, velocity, and orientation) are defined with respect to scenario coordinates.

```
allposes = actorPoses(s);
```

Use `targetsToEgo` to convert just the target poses to the egocentric coordinates of the ego actor. Examine the pose of the first actor.

```
targetposes1 = driving.scenario.targetsToEgo(allposes(1:2),ego);
disp(targetposes1(1))
```

```
            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: [20 40 20]
```

Alternatively, use `targetPoses` to obtain all non-ego actor poses in ego actor coordinates. Compare this result to the previous calculation of poses.

```
targetposes2 = targetPoses(ego);
disp(targetposes2(1))
```

```
            ActorID: 1
            ClassID: 0
           Position: [7.8415 18.2876 27.1675]
```

```
          Velocity: [18.6826 112.0403 9.2960]
              Roll: 16.4327
             Pitch: 23.2186
               Yaw: 47.8114
   AngularVelocity: [20 40 20]
```

Input Arguments

actorPoses — Actor poses in scenario coordinates
structure | array of structures

Actor poses in scenario coordinates, specified as a structure or array of structures. Each pose structure has the fields:

Field	Description
ActorID	Scenario-defined actor identifier
Position	Position of actor, specified as a real-valued 1-by-3 vector. Units are in meters.
Velocity	Velocity of actor, specified as a real-valued 1-by-3 vector. Units are in meters per second.
Roll	Roll angle of actor, specified as a scalar. Units are in degrees.
Pitch	Pitch angle of actor, specified as a scalar. Units are in degrees.
Yaw	Yaw angle of actor, specified as a scalar. Units are in degrees.
AngularVelocity	Angular velocity of actor, specified as a real-valued 1-by-3 vector. Units are in degrees per second.

See `Actor` and `Vehicle` for full definitions of the structure fields.

egoActor — Ego actor pose in scenario coordinates
structure

Ego actor pose in scenario coordinates, specified as a structure. The structure fields are:

Field	Description
ActorID	Scenario-defined actor identifier
Position	Position of actor, specified as a real-valued 1-by-3 vector. Units are in meters.
Velocity	Velocity of actor, specified as a real-valued 1-by-3 vector. Units are in meters per second.
Roll	Roll angle of actor, specified as a scalar. Units are in degrees.
Pitch	Pitch angle of actor, specified as a scalar. Units are in degrees.
Yaw	Yaw angle of actor, specified as a scalar. Units are in degrees.
AngularVelocity	Angular velocity of actor, specified as a real-valued 1-by-3 vector. Units are in degrees per second.

See `Actor` and `Vehicle` for full definitions of the structure fields.

Output Arguments

`targetPoses` — Target poses in ego coordinates
structure | array of structures

Target poses in ego coordinates, specified as a structure or array of structures. Each structure has the fields:

Field	Description
ActorID	Scenario-defined actor identifier
Position	Position of actor, specified as a real-valued 1-by-3 vector. Units are in meters.
Velocity	Velocity of actor, specified as a real-valued 1-by-3 vector. Units are in meters per second.

Field	Description
`Roll`	Roll angle of actor, specified as a scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a scalar. Units are in degrees.
`AngularVelocity`	Angular velocity of actor, specified as a real-valued 1-by-3 vector. Units are in degrees per second.

See `Actor` and `Vehicle` for full definitions of the structure fields.

Definitions

Ego and target actors

In a driving scenario, you can specify one actor as the observer of all other actors, much as the driver of a car observes all other cars. The observer actor is called the *ego actor*. From the perspective of the ego actor, all other actors are the observed actors and are called *target actors* or *targets*. Ego coordinates are coordinates centered and oriented with reference to the ego actor. Driving scenario coordinates are world or global coordinates.

See Also

`driving.scenario.roadBoundariesToEgo` | `drivingScenario.actor` | `drivingScenario.actorPoses` | `drivingScenario.vehicle` | `roadBoundaries` | `targetPoses`

Introduced in R2017a

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

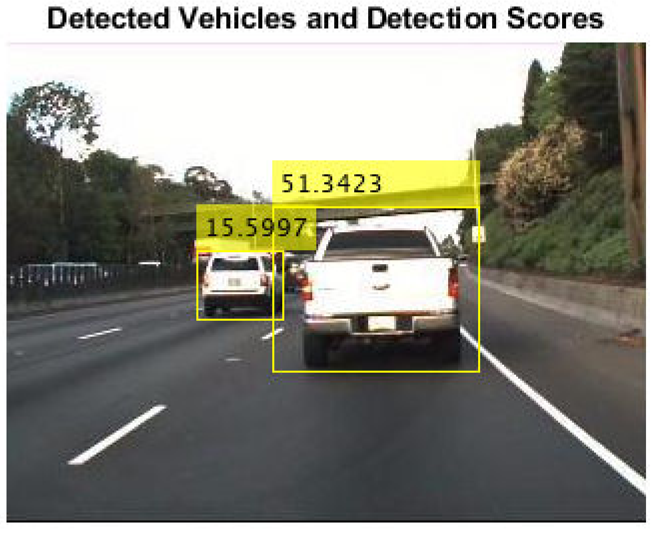
```
detector = vehicleDetectorACF('front-rear-view');
```

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
detector = vehicleDetectorFasterRCNN(modelName)
```

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 function trains the detector 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 CIFAR-10 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`.

`detector = vehicleDetectorFasterRCNN(modelName)` returns a pretrained vehicle detector based on the model name specified in `modelName`. The default `'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 of only the front and rear sides of the vehicles.

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.

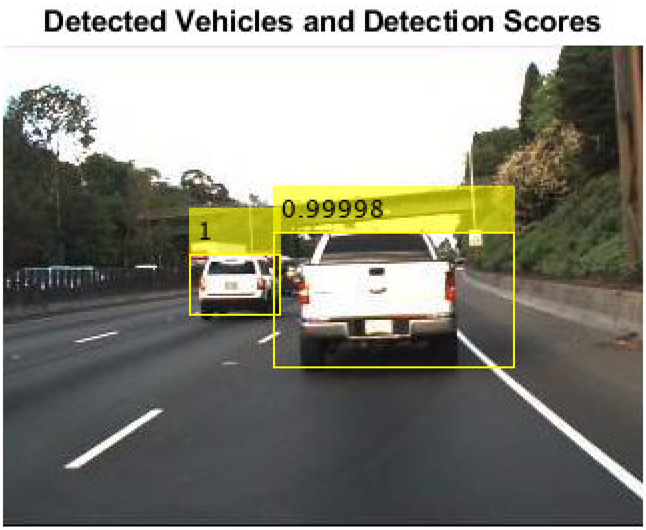
```
fasterRCNN = vehicleDetectorFasterRCNN('full-view');
```

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')
```



Input Arguments

modelName — Type of vehicle detector model
'full-view' (default) | 'front-rear-view'

Type of vehicle detector model, specified as either 'full-view' or 'front-rear-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 of only the front and rear sides of the vehicles.

Data Types: char | string

Output Arguments

detector — Trained Faster R-CNN-based object detector
fasterRCNNObjectDetector object

Trained Faster R-CNN-based object detector, returned as an fasterRCNNObjectDetector object.

See Also

fasterRCNNObjectDetector | trainFasterRCNNObjectDetector | vehicleDetectorACF

Introduced in R2017a

4

Objects in Automated Driving System Toolbox

driving.connector.Connector class

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 the **Ground Truth Labeler** app.

Construction

The `Connector` class that inherits from the `Connector` interface, is called a client.

The client can:

- Sync an external tool to each frame change event within 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 for it in the app.
- Export custom labeled data from an external tool via the app.

1. Define a client class that inherits from `driving.connector.Connector`. You can use a `ConnectorClass` template to define the 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, add the folder into which you saved the file to the MATLAB path. To add a folder to the path, use the `addpath` function.

Properties

`VideoStartTime` — Start time of source video file
scalar in seconds

This property is read-only.

Start time of source video file, specified as a scalar in seconds.

`VideoEndTime` — End time of source video file
scalar in seconds

This property is read-only.

End time of source video file, specified as a scalar in seconds.

`StartTime` — Start time of video interval in app
scalar in seconds

This property is read-only.

Start time of video interval in app, specified as a scalar in seconds. To set the start time, use the start flag interval in the app.

`CurrentTime` — Time of video frame currently displaying in app
scalar in seconds

This property is read-only.

Time of video frame currently displaying in app, specified as a scalar in seconds.

`EndTime` — End time of video in app
scalar in seconds

This property is read-only.

End time of video in app, specified as a scalar in seconds. To set the end time, use the end flag interval in the app.

`TimeVector` — Time stamps for the loaded video
array

This property is read-only.

Timestamps for the loaded video, specified in an array.

LabelData — Label data imported from external tool
two-column table

This property is read-only.

Label data imported from external tool, specified as a two-column table. The first column contains timestamps and the second column contains the label information that you specify for the corresponding timestamp.

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`.

LabelDescription — Descriptions of labels
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`.

Methods

The client class must implement the following methods:

frameChangeListener	Update external tool when a new frame is detected

The client class can optionally implement the following methods:

close	Close external tool
labelDefinitionLoadListener	Update new label definitions from external tool
labelLoadListener	Update new label data from external tool

The client class can call the following methods:

addLabelData	Add custom label data at current time
dataSourceChangeListener	Update external tool when you add data source to app
disconnect	Disconnect external tool from app
queryLabelData	Query for custom label data at current time
updateLabelerCurrentTime	Update current time for app

Examples

Connect Lidar Display to Ground Truth Labeler

Connect a lidar data visualization tool to the Ground Truth Labeler app. Use the app and tool to display synchronized lidar and video data. To use another set of data, modify the MATLAB code in this example.

Specify the video name to display in the Ground Truth Labeler.

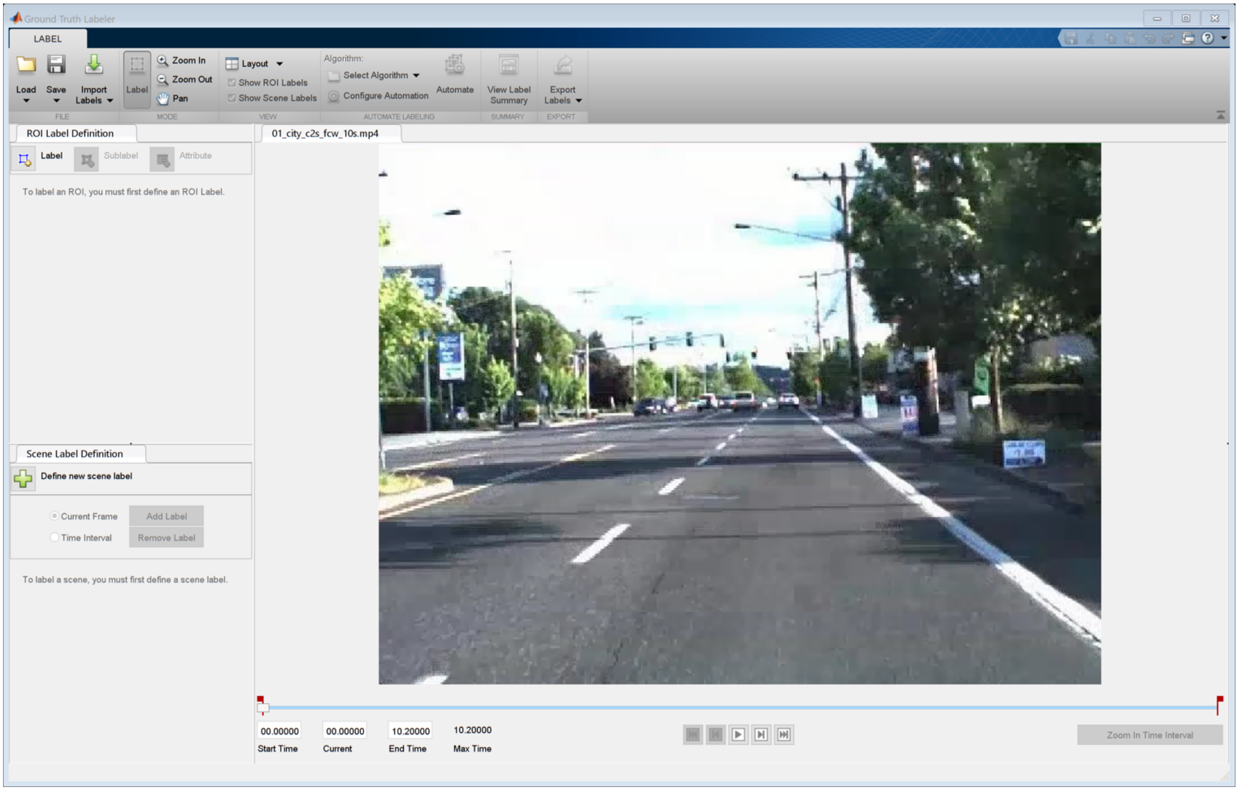
```
videoName = '01_city_c2s_fcw_10s.mp4';
```

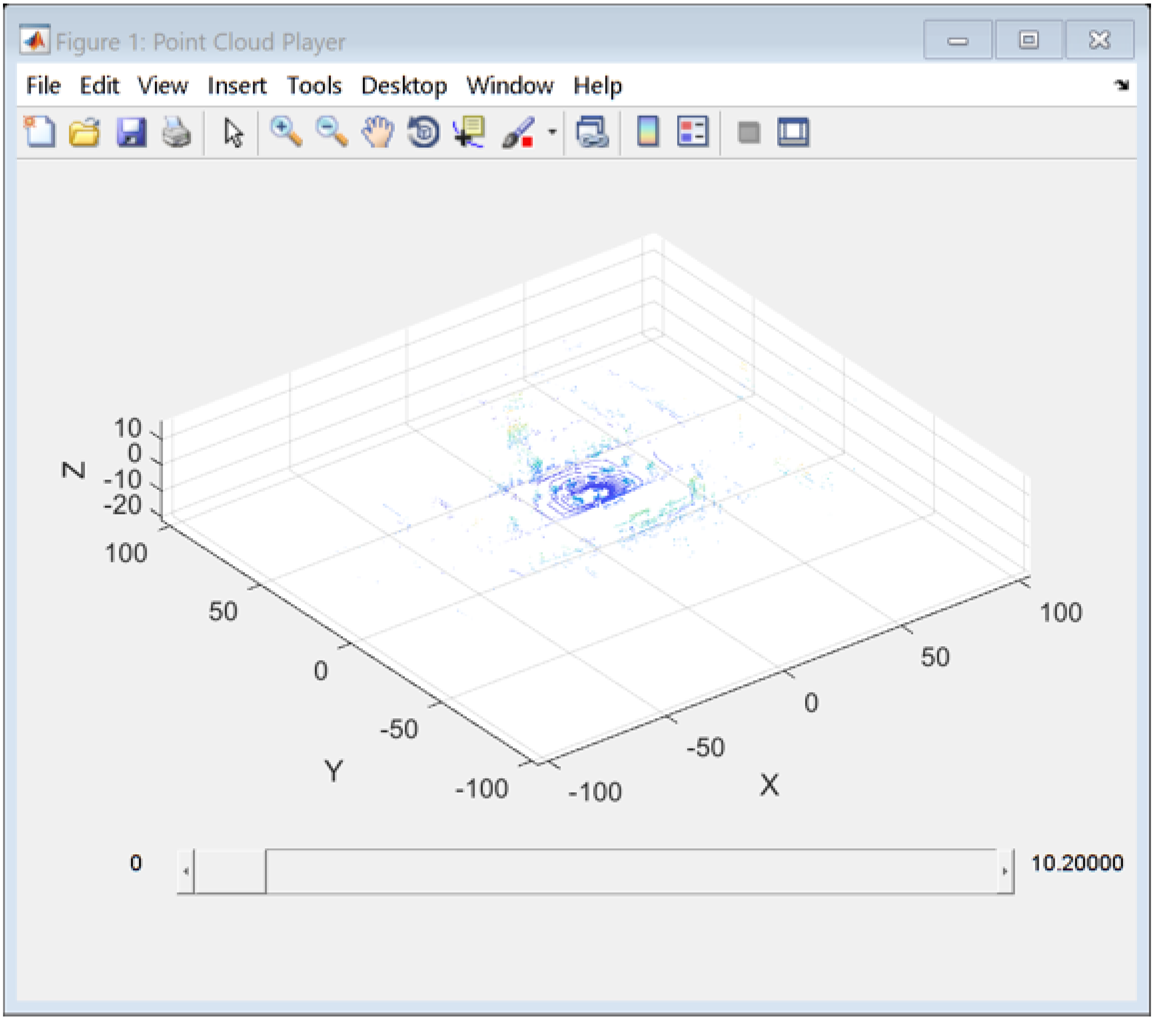
Add the path to the lidar display data.

```
addpath(fullfile(matlabroot,'toolbox','driving','drivingdemos'));
```

Connect the lidar display to the Ground Truth Labeler.

```
groundTruthLabeler(videoName,'ConnectorTargetHandle',@LidarDisplay);
```


Ground Truth Labeler
LABEL
Load
Save
Import Labels
Label
Zoom In
Zoom Out
Pan
Layout
Show ROI Labels
Show Scene Labels
Algorithm:
Select Algorithm
Configure Automation
Automate
View Label Summary
Export Labels
ROI Label Definition
Label
Sublabel
Attribute
To label an ROI, you must first define an ROI Label.
01_city_c2s_fcw_10s.mp4
Scene Label Definition
Define new scene label
Current Frame
Time Interval
Add Label
Remove Label
To label a scene, you must first define a scene label.
00.00000
Start Time
00.00000
Current
10.20000
End Time
10.20000
Max Time
Zoom In Time Interval



See Also

Apps
Ground Truth Labeler

Introduced in R2017a

addLabelData

Class: driving.connector.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.

The label data added using this method is incorporated into the `groundTruth` object, which is exported by the **Ground Truth Labeler** app. The label data is added as a custom label, with its name specified by the `LabelName` property.

Input Arguments

`connectorObj` — Connector object
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`.

See Also

Apps
Ground Truth Labeler

Functions
`driving.connector.Connector` | `groundTruth`

Introduced in R2017a

close

Class: driving.connector.Connector

Close external tool

Syntax

`close(connectorObj)`

Description

`close(connectorObj)` provides the option to close the external tool when the **Ground Truth Labeler** closes. The app calls this method using the `connectorObj` object.

Input Arguments

`connectorObj` — Connector object
object

Connector object, specified as a `driving.connector.Connector` object.

See Also

Apps
Ground Truth Labeler

Functions
`driving.connector.Connector`

Introduced in R2017a

dataSourceChangeListener

Class: driving.connector.Connector

Update external tool when you add data source to app

Syntax

`dataSourceChangeListener(connectorObj)`

Description

`dataSourceChangeListener(connectorObj)` provides an 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 a new data source being connected to the app.

A new data source can be a video, image sequence, or custom reader. You can load a new data source while loading a new session.

Input Arguments

`connectorObj` — Connector object
object

Connector object, specified as a `driving.connector.Connector` object.

See Also

Apps
Ground Truth Labeler

Functions
`driving.connector.Connector`

Introduced in R2017a

disconnect

Class: driving.connector.Connector

Disconnect external tool from app

Syntax

`disconnect(connectorObj)`

Description

`disconnect(connectorObj)` disconnects the external tool from the **Ground Truth Labeler** app. After the external tool is disconnected, the **Ground Truth Labeler** app no longer calls the `frameChangeListener` method in the client class. The client calls this method using the `connectorObj` object.

Input Arguments

`connectorObj` — Connector object
object

Connector object, specified as a `driving.connector.Connector` object.

See Also

Apps
Ground Truth Labeler

Functions
`driving.connector.Connector`

Introduced in R2017a

frameChangeListener

Class: driving.connector.Connector

Update external tool when a new frame is detected

Syntax

`frameChangeListener(connectorObj)`

Description

`frameChangeListener(connectorObj)` provides an option to synchronize the external tool with frame changes in the **Ground Truth Labeler** app. The app calls this method whenever a new frame is displayed in the app and must be implemented by the client class.

Input Arguments

`connectorObj` — Connector object
object

Connector object, specified as a `driving.connector.Connector` object.

See Also

Apps
Ground Truth Labeler

Functions
`driving.connector.Connector`

Introduced in R2017a

labelDefinitionLoadListener

Class: driving.connector.Connector

Update new label definitions from external tool

Syntax

`labelDefinitionLoadListener(connectorObj)`

Description

`labelDefinitionLoadListener(connectorObj)` provides an option to update the external tool when a new set of label definitions is imported into the **Ground Truth Labeler** app. The app calls this method using the `conectorObj` object. You can optionally use this method to react to a new data source being connected to the app.

Input Arguments

`connectorObj` — Connector object
object

Connector object, specified as a `driving.connector.Connector` object.

See Also

Apps
Ground Truth Labeler

Functions
`driving.connector.Connector`

Introduced in R2017a

labelLoadListener

Class: driving.connector.Connector

Update new label data from external tool

Syntax

`labelLoadListener(connectorObj)`

Description

`labelLoadListener(connectorObj)` provides the option to update the external tool when a new set of label data or new session with label data is imported into the **Ground Truth Labeler** app. The app calls this method using the `connectorObj` object. Use this method to react to new label data being loaded into the app.

Input Arguments

`connectorObj` — Connector object
object

Connector object, specified as a `driving.connector.Connector` object.

See Also

Apps
Ground Truth Labeler

Functions
`driving.connector.Connector`

Introduced in R2017a

queryLabelData

Class: driving.connector.Connector

Query for custom label data at current time

Syntax

`queryLabelData(connectorObj)`

Description

`queryLabelData(connectorObj)` queries label data related to the current time in the **Ground Truth Labeler** app. The client calls this method using the `connectorObj`.

Input Arguments

`connectorObj` — Connector object
object

Connector object, specified as a `driving.connector.Connector` object.

See Also

Apps
Ground Truth Labeler

Functions
`driving.connector.Connector`

Introduced in R2017a

updateLabelerCurrentTime

Class: driving.connector.Connector

Update current time for app

Syntax

`updateLabelerCurrentTime(connectorObj,newTime)`

Description

`updateLabelerCurrentTime(connectorObj,newTime)` updates the current time in the **Ground Truth Labeler** app to the specified new time. The client calls this method using the `connectorObj` object.

Input Arguments

`connectorObj` — Connector object
object

Connector object, specified as a `driving.connector.Connector` object.

`newTime` — Current time for app
scalar in seconds

Current time for app, specified as a scalar in seconds. The `newTime` value sets the current time in the **Ground Truth Labeler** app.

See Also

Apps
Ground Truth Labeler

Functions
`driving.connector.Connector`

Introduced in R2017a

geoplayer

Visualize streaming geographic map data

Description

The `geoplayer` object displays a stream of geographic coordinates on a map.

Creation

Use the `geoplayer` function to create a player for streaming geographic coordinates.

Syntax

```
player = geoplayer(latCenter,lonCenter)
player = geoplayer(latCenter,lonCenter,zoomLevel)
player = geoplayer( ___ ,Name,Value)
```

Description

`player = geoplayer(latCenter,lonCenter)` creates a streaming geographic player, centered at latitude coordinate `latCenter` and longitude coordinate `lonCenter`.

`player = geoplayer(latCenter,lonCenter,zoomLevel)` creates a streaming geographic player with a map magnification specified by `zoomLevel`.

`player = geoplayer( ___ ,Name,Value)` sets properties of the geoplayer by using name-value pair arguments. `Name` is the property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For example, `geoplayer(45,0,'HistoryDepth',5)` creates a geoplayer centered at the (lat,lon) coordinate (45,0), and sets the `HistoryDepth` property to display the five previous geographic coordinates.

Input Arguments

`latCenter` — Latitude coordinate
numeric scalar in the range (-90, 90)

Latitude coordinate at which the geoplayer is centered, specified as a numeric scalar in the range (-90, 90).

Data Types: `single` | `double`

`lonCenter` — Longitude coordinate
numeric scalar in the range [-180, 180]

Longitude coordinate at which the geoplayer is centered, specified as a numeric scalar in the range [-180, 180].

Data Types: `single` | `double`

`zoomLevel` — Magnification
`15` | scalar integer in the range [0, 25]

Magnification of the geoplayer, specified as a scalar integer in the range [0, 25]. The 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) | scalar integer | `Inf`

Number of previous geographic coordinates to display, specified as a scalar integer or `Inf`. A value of `0` displays only the current geographic coordinates. A value of `Inf` displays all geographic coordinates previously plotted using `plotPosition`.

Example: 7

`HistoryStyle` — Style of displayed geographic coordinates
`'point'` (default) | `'line'`

Style of displayed geographic coordinates, specified as one of the following:

- `'point'` — Display the track as discrete, unconnected points.
- `'line'` — Display the track as a single connected line.

`Parent` — Player axes handle
figure graphics object | panel graphics object

Player axes handle, specified as a `figure` or `uipanel` graphics object. If you do not specify '`Parent`', then `geoplayer` creates the player in a new figure.

Object Functions

plotPosition	Display current position in geoplayer
plotRoute	Display continuous route in geoplayer
reset	Remove all existing plots from geoplayer
show	Make geoplayer figure visible
hide	Make geoplayer figure invisible
isOpen	Return true if geoplayer is visible

Examples

Animate Sequence of Latitude and Longitude Coordinates

Load latitude and longitude coordinates.

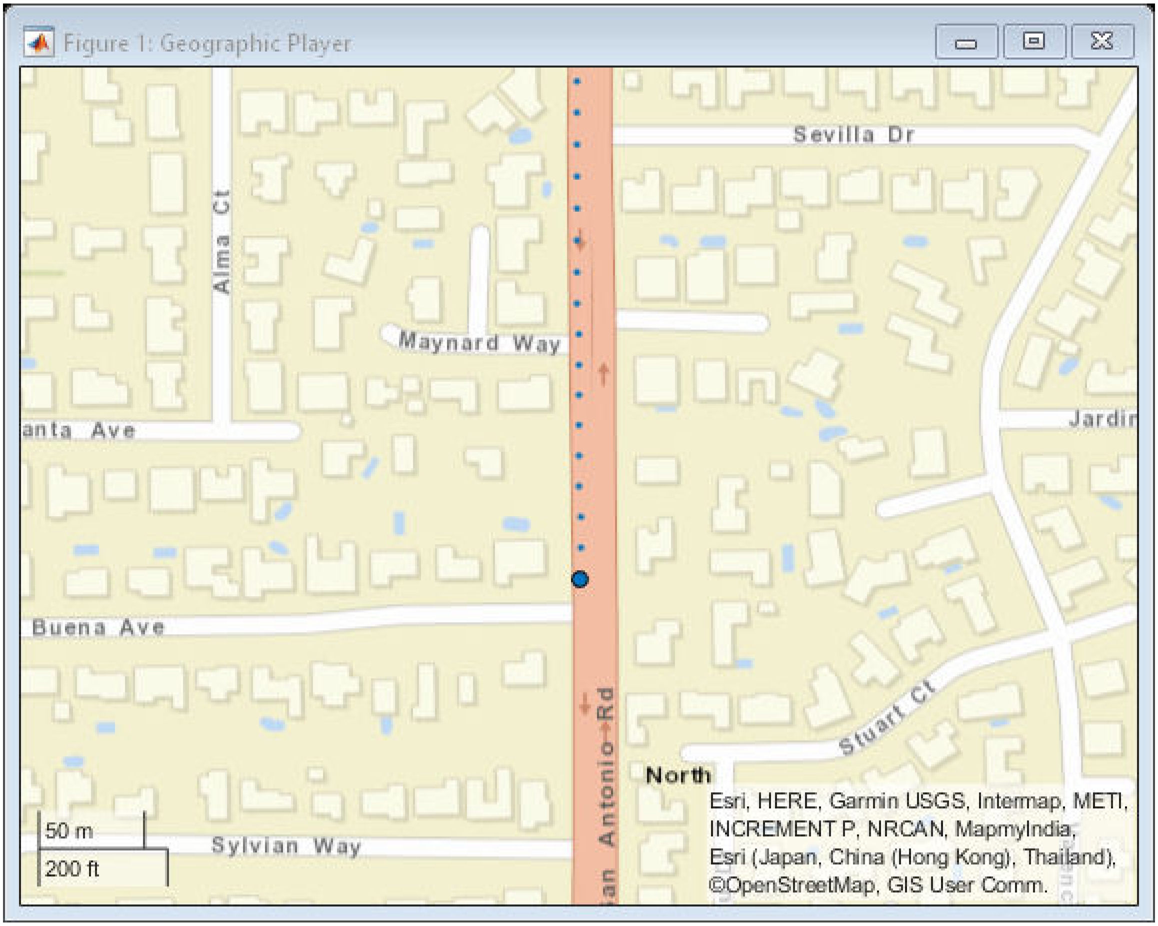
```
data = load('geoSequence.mat');
```

Create the geoplayer and configure it to display all points in the history.

```
player = geoplayer(data.latitude(1),data.longitude(1),17,'HistoryDepth',Inf);
```

Display the coordinates in a sequence.

```
for i = 1:length(data.latitude)
    plotPosition(player,data.latitude(i),data.longitude(i));
    pause(0.01)
end
```



View Position of a Vehicle Along a Route

Load a sequence of latitude and longitude coordinates.

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

Create the geoplayer and set the zoom level to 12. The map is zoomed out by a factor of 8 compared to the default zoom level.

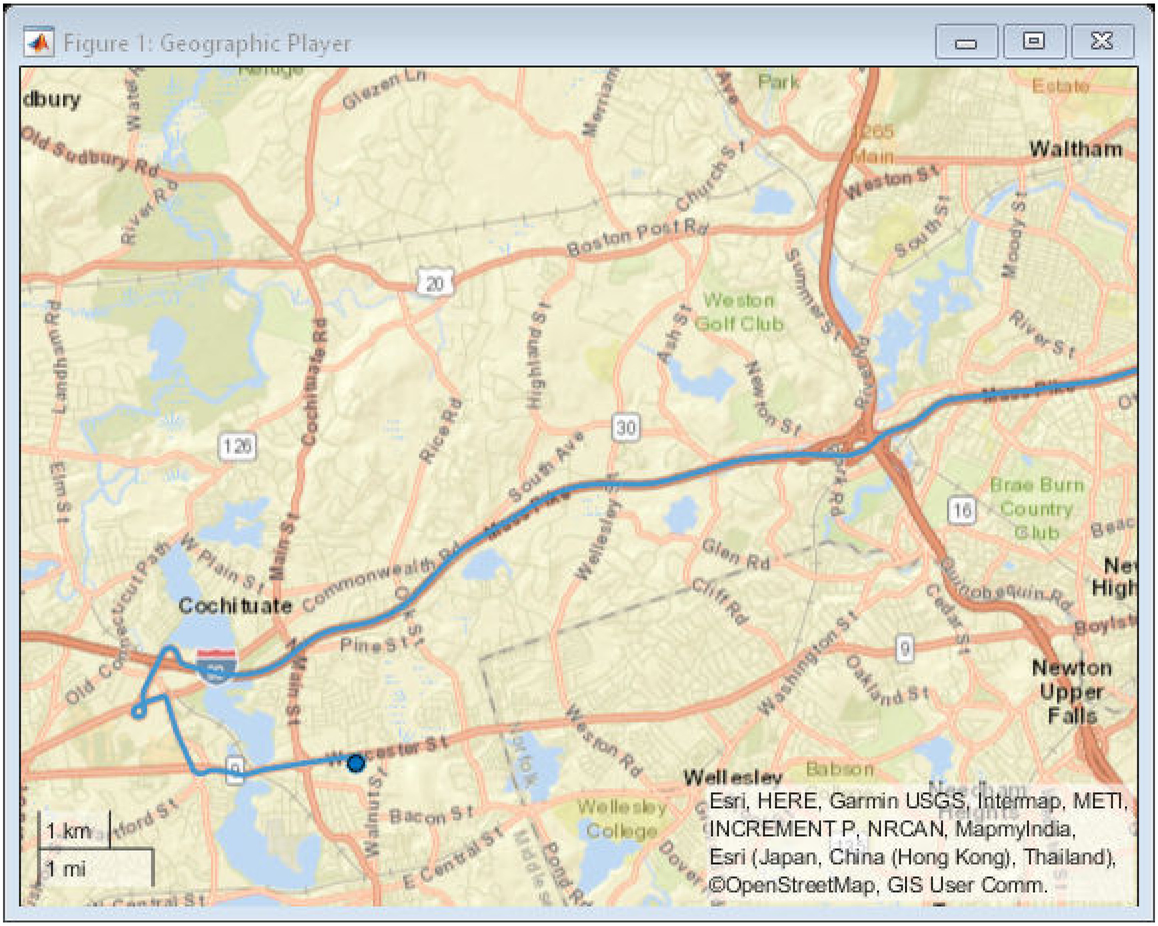
```
player = geoplayer(data.latitude(1),data.longitude(1),12);
```

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
```



Limitations

- Geographic map tiles are not available for all locations.

Tips

- `geoplayer` displays geographic map tiles using the World Street Map provided by Esri®. This basemap requires access to an internet connection to fetch map tiles. For more information about the map, see World Street Map on the Esri ArcGIS website.
- The `geoplayer` automatically scrolls the map whenever it plots a position that is outside the current view of the map.

See Also

`geobubble`

Introduced in R2018a

plotPosition

Display current position in geoplayer

Syntax

```
plotPosition(player,latitude,longitude)
plotPosition(player,latitude,longitude,Name,Value)
```

Description

`plotPosition(player,latitude,longitude)` plots a point with `latitude` and `longitude` coordinates in a geoplayer.

`plotPosition(player,latitude,longitude,Name,Value)` uses `Name,Value` pair arguments to modify the visual style of the plotted points.

For example, `plotPosition(player,45,0,'Color','w','Marker','*')` plots a point in the geoplayer as a white star.

Examples

View Position of a Vehicle Along a Route

Load a sequence of latitude and longitude coordinates.

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

Create the geoplayer and set the zoom level to `12`. The map is zoomed out by a factor of 8 compared to the default zoom level.

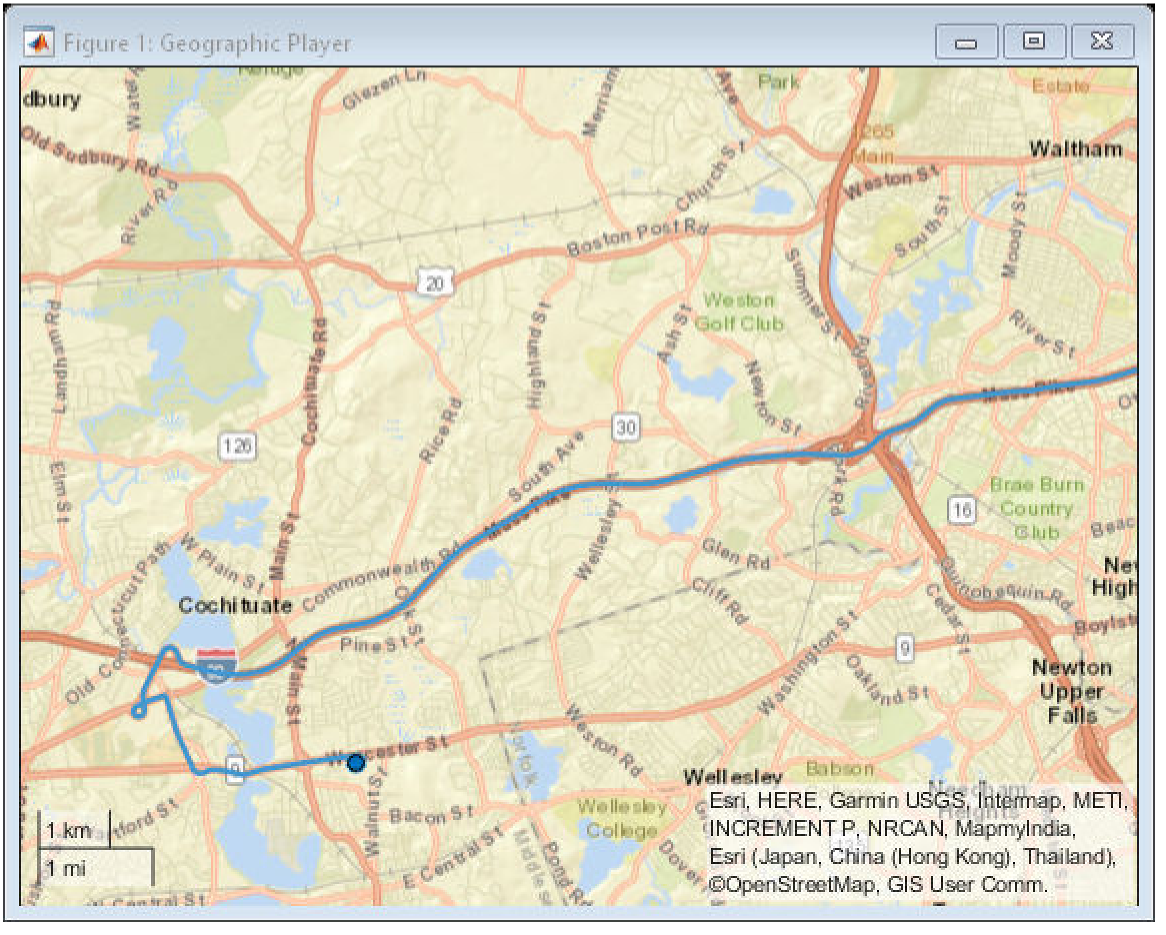
```
player = geoplayer(data.latitude(1),data.longitude(1),12);
```

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
```



Input Arguments

`player` — Streaming geographic player
`geoplayer` object

Streaming geographic player, specified as a `geoplayer` object.

`latitude` — Latitude coordinate
numeric scalar in the range [-90, 90]

Latitude coordinate of the point to display in the geoplayer, specified as a numeric scalar in the range [-90, 90].

Data Types: `single` | `double`

`longitude` — Longitude coordinate
numeric scalar in the range [-180, 180]

Longitude coordinate of the point to display in the geoplayer, specified as a numeric 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
`ColorSpec`

Marker color, specified as the comma-separated pair consisting of `'Color'` and a `ColorSpec`, such as an RGB triplet or one of the MATLAB predefined names. Color is used only for filled marker symbols. By default, the marker color is selected automatically.

Example: `'Color',[1 0 1]`

Example: `'Color','m'`

Example: `'Color','magenta'`

`Marker` — Marker symbol
`'o'` (default) | character

Marker symbol, specified as the comma-separated pair consisting of `'Marker'` and one of these characters.

Value	Description
`'.'`	Point
`'x'`	Cross
`'+'`	Plus sign
`'*'`	Asterisk
`'o'`	Circle (default)
`'s'`	Square
`'d'`	Diamond
`'p'`	Five-pointed star (pentagram)
`'h'`	Six-pointed star (hexagram)
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'<'`	Left-pointing triangle
`'>'`	Right-pointing triangle

`MarkerSize` — Diameter of marker
6 (default) | positive scalar

Approximate diameter of marker in points, specified as the comma-separated pair consisting of '`MarkerSize`' and a positive scalar. 1 point = 1/72 inch. A marker size larger than 6 can reduce the rendering performance.

See Also
`geoplayer` | `plotRoute` | `reset`

Introduced in R2018a

plotRoute

Display continuous route in geoplayer

Syntax

```
plotRoute(player,latitude,longitude)
plotRoute(player,latitude,longitude,Name,Value)
```

Description

`plotRoute(player,latitude,longitude)` displays a series of points specified by `latitude` and `longitude` coordinates as a route in a geoplayer. The route appears as a continuous line on a map.

`plotRoute(player,latitude,longitude,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 as a black line.

Examples

View Position of a Vehicle Along a Route

Load a sequence of latitude and longitude coordinates.

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

Create the geoplayer and set the zoom level to `12`. The map is zoomed out by a factor of 8 compared to the default zoom level.

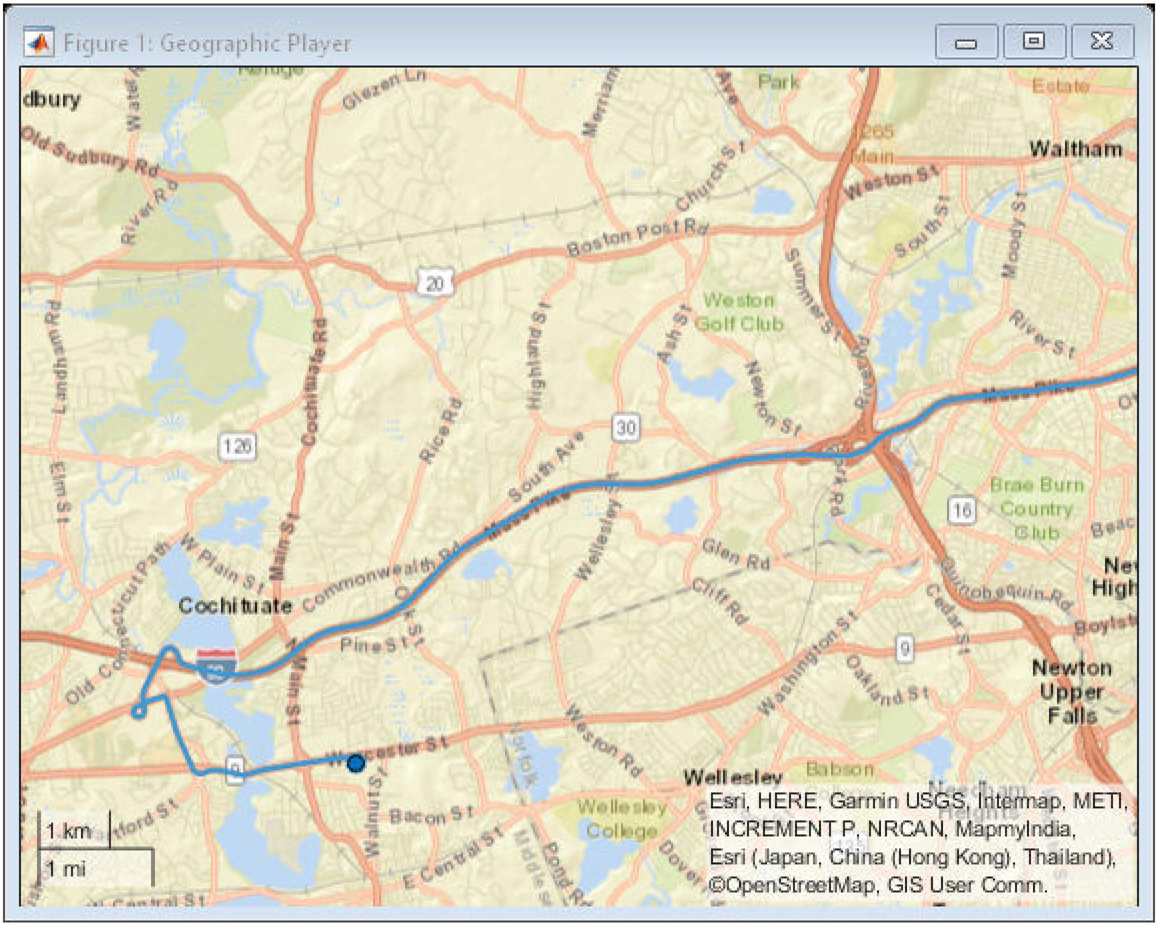
```
player = geoplayer(data.latitude(1),data.longitude(1),12);
```

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
```



Input Arguments

`player` — Streaming geographic player
`geoplayer` object

Streaming geographic player, specified as a `geoplayer` object.

`latitude` — Latitude coordinates
numeric vector

Latitude coordinates of points along the route, specified as a numeric vector with elements in the range [-90, 90].

Data Types: `single` | `double`

`longitude` — Longitude coordinates
numeric vector

Longitude coordinates of points along the route, specified as a numeric 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
`ColorSpec`

Line color, specified as the comma-separated pair consisting of `'Color'` and a `ColorSpec`, such as an RGB triplet or one of the MATLAB predefined names. By default, the line color is selected automatically.

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 | plotPosition | reset

Introduced in R2018a

reset

Remove all existing plots from geoplayer

Syntax

`reset(player)`

Description

`reset(player)` removes all previously plotted points and routes from the geoplayer.

Examples

Reset Geoplayer Figure

Load a sequence of latitude and longitude coordinates.

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

Create a geoplayer with a zoom level of 12. Configure the geoplayer to display all points in the history.

```
player = geoplayer(data.latitude(1),data.longitude(1),12,'HistoryDepth',Inf);
```

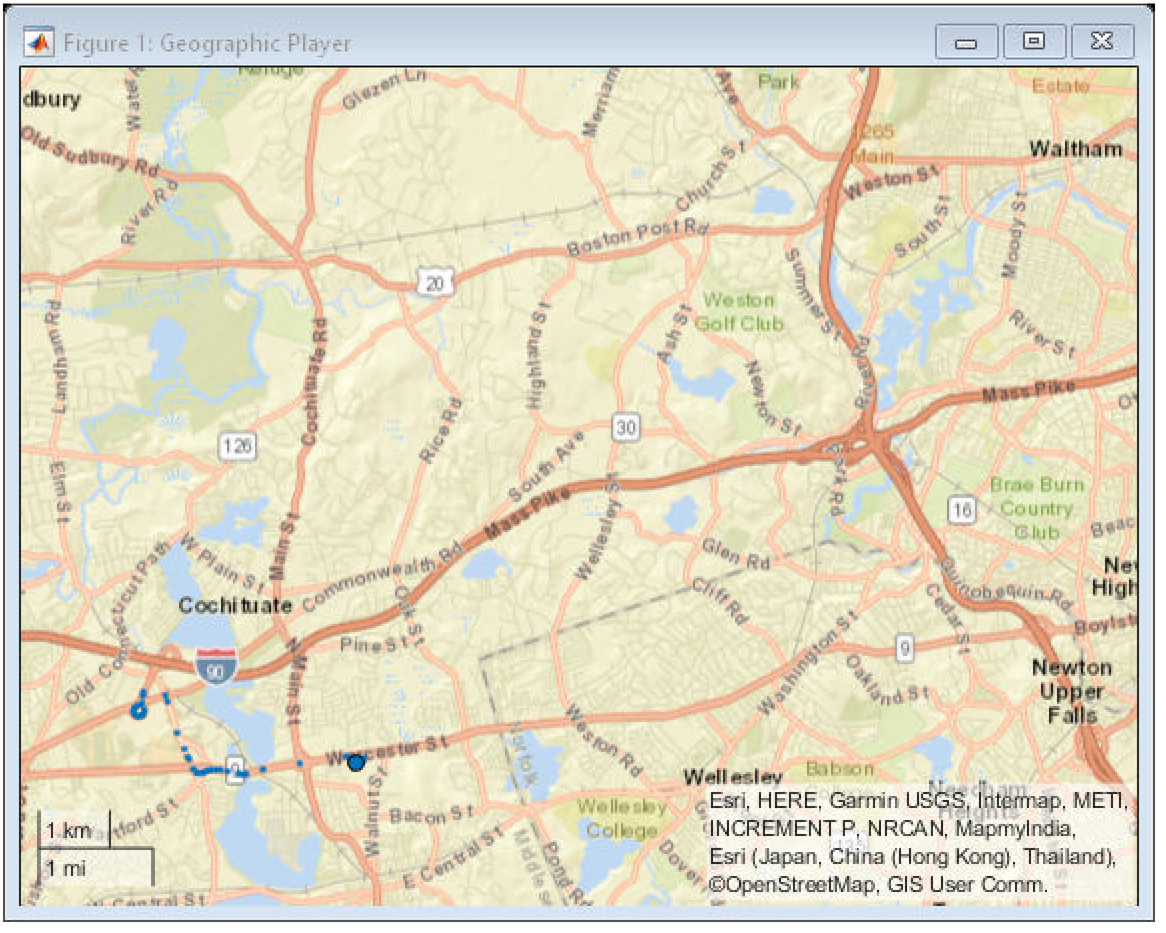
Display the full route.

```
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 geoplayer. 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.

See Also

plotPosition | plotRoute

Introduced in R2018a

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 Geoplayer Figure

Load a sequence of latitude and longitude coordinates.

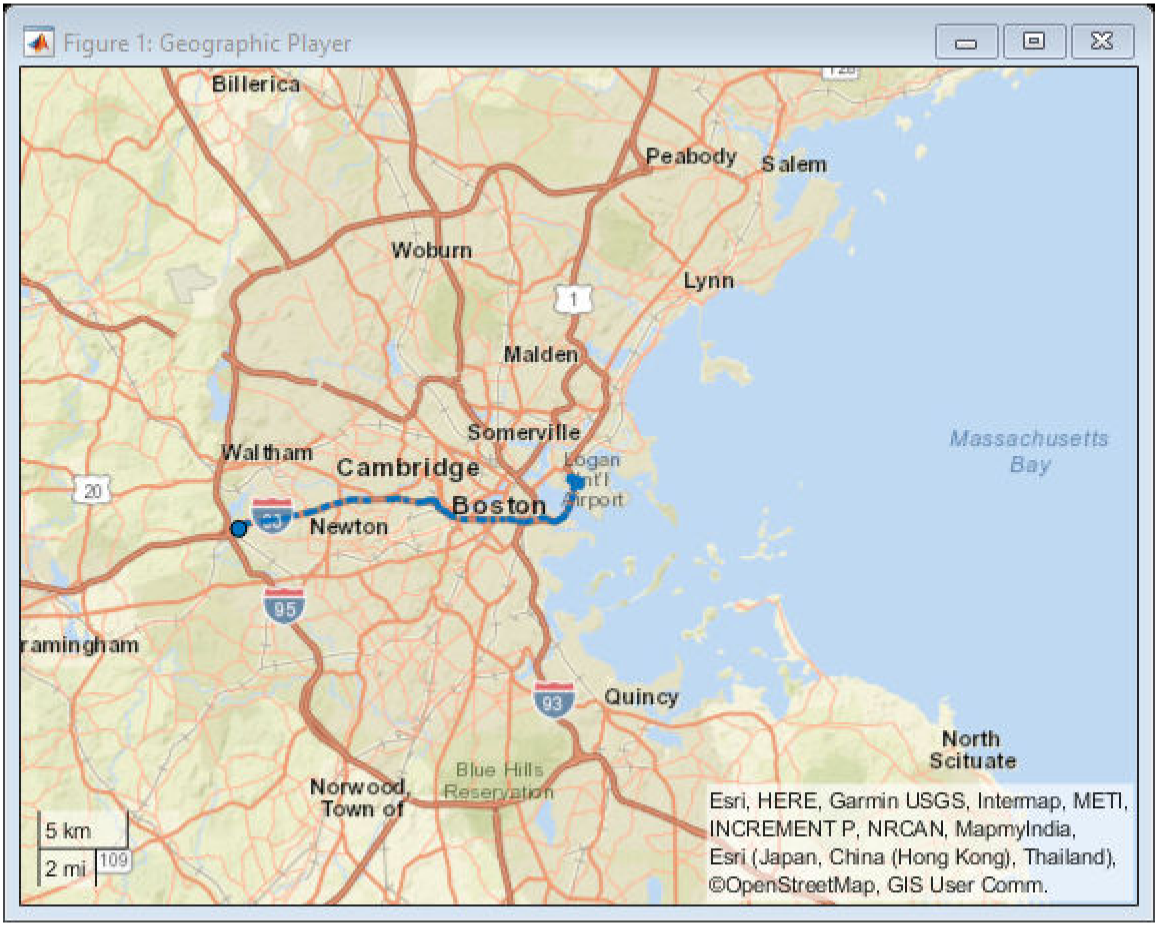
```
data = load('geoRoute.mat');
```

Create a geoplayer with a zoom level of 10. Configure it to show the 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 geoplayer and confirm that it is no longer visible.

```
hide(player)
isOpen(player)
```

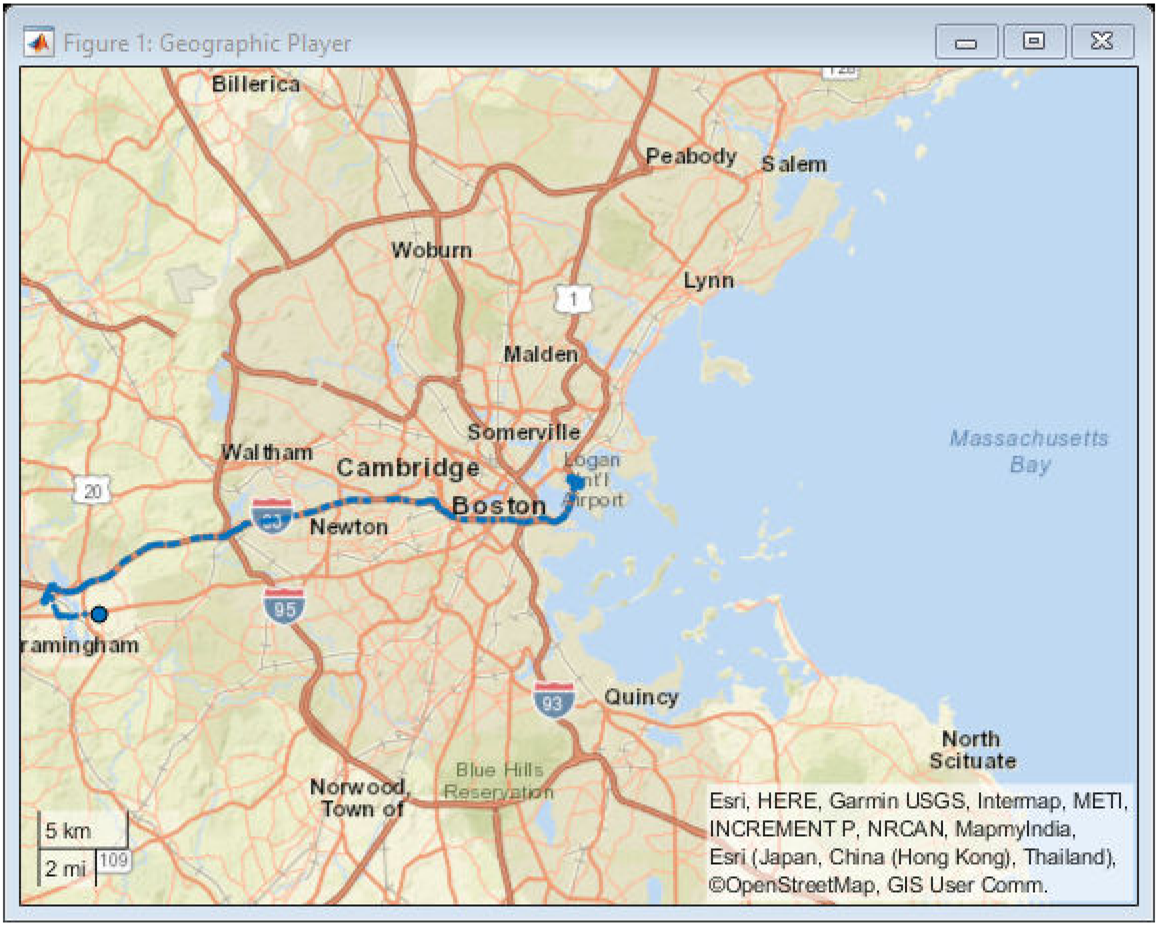
```
ans = logical
   0
```

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 geoplayer. The geoplayer 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.

See Also

`hide` | `isOpen`

Introduced in R2018a

hide

Make geoplayer figure invisible

Syntax

```
hide(player)
```

Description

`hide(player)` hides the geoplayer figure. To redisplay the geoplayer, use `show(player)`.

Examples

Hide and Show Geoplayer Figure

Load a sequence of latitude and longitude coordinates.

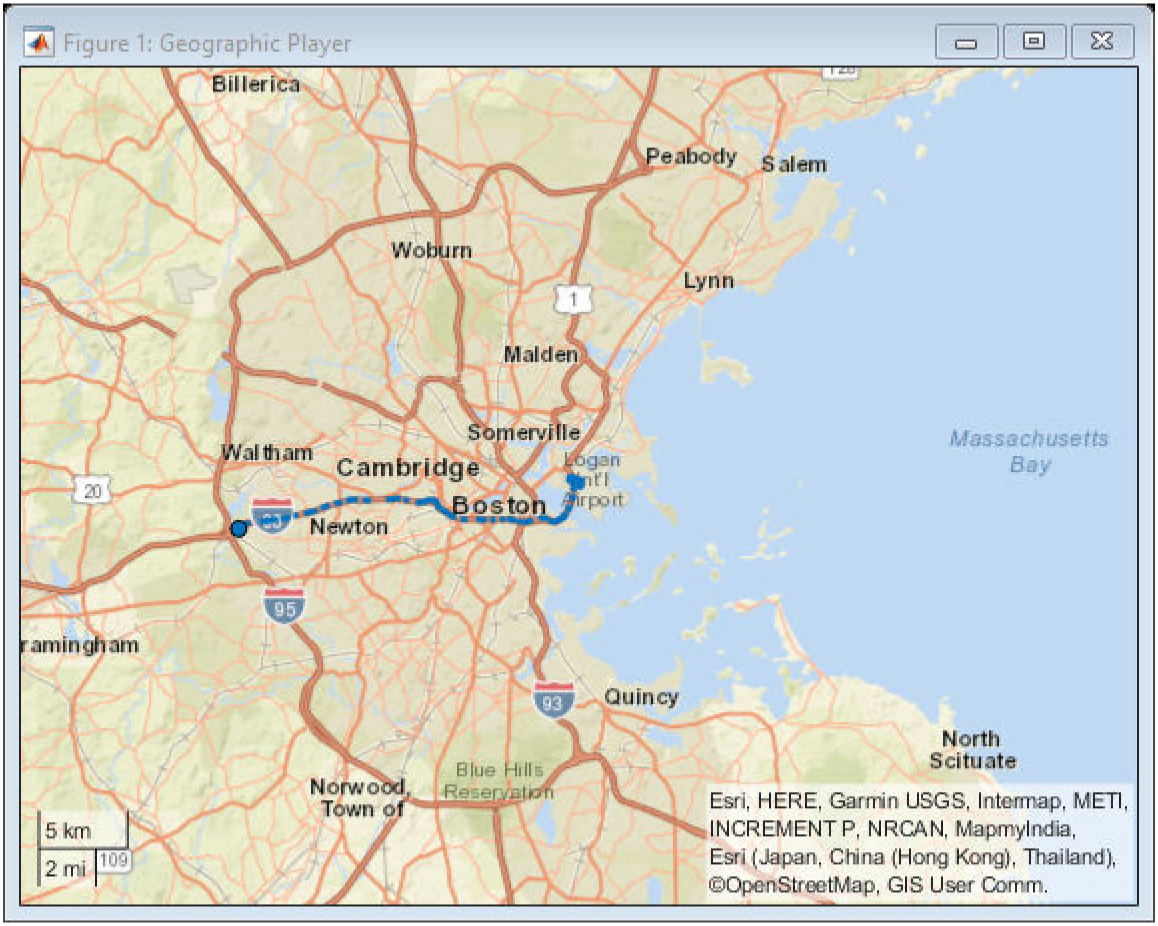
```
data = load('geoRoute.mat');
```

Create a geoplayer with a zoom level of 10. Configure it to show the 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 geoplayer and confirm that it is no longer visible.

```
hide(player)
isOpen(player)
```

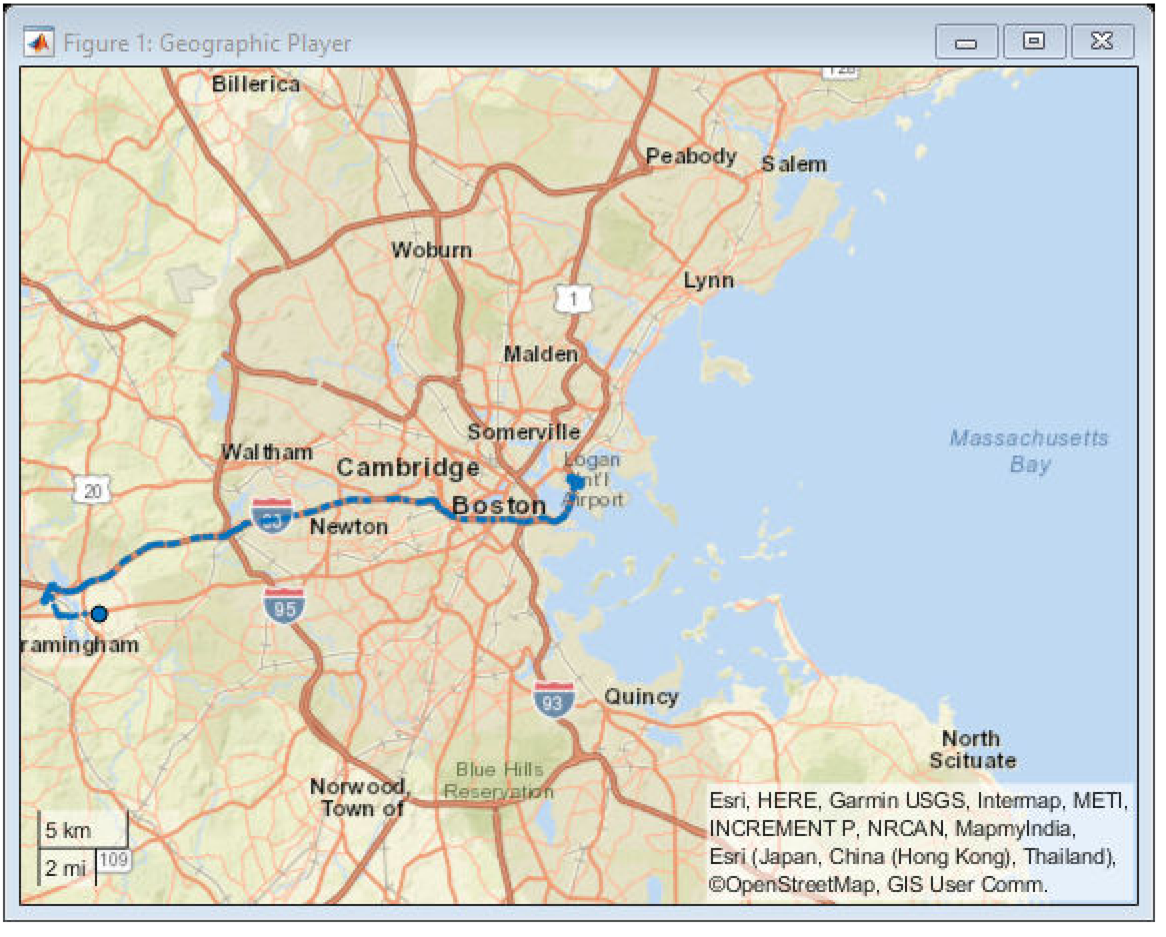
```
ans = logical
   0
```

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 geoplayer. The geoplayer 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.

See Also

`isOpen` | `show`

Introduced in R2018a

isOpen

Return true if geoplayer is visible

Syntax

```
tf = isOpen(player)
```

Description

`tf = isOpen(player)` returns `true` or `false` to indicate whether the geoplayer figure is visible.

Examples

Plot Points While Geoplayer Is Open

Load a sequence of latitude and longitude coordinates.

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

Create a geoplayer with a zoom level of 12. Configure the geoplayer to display all points in the history.

```
player = geoplayer(data.latitude(1),data.longitude(1),12,'HistoryDepth',Inf);
```

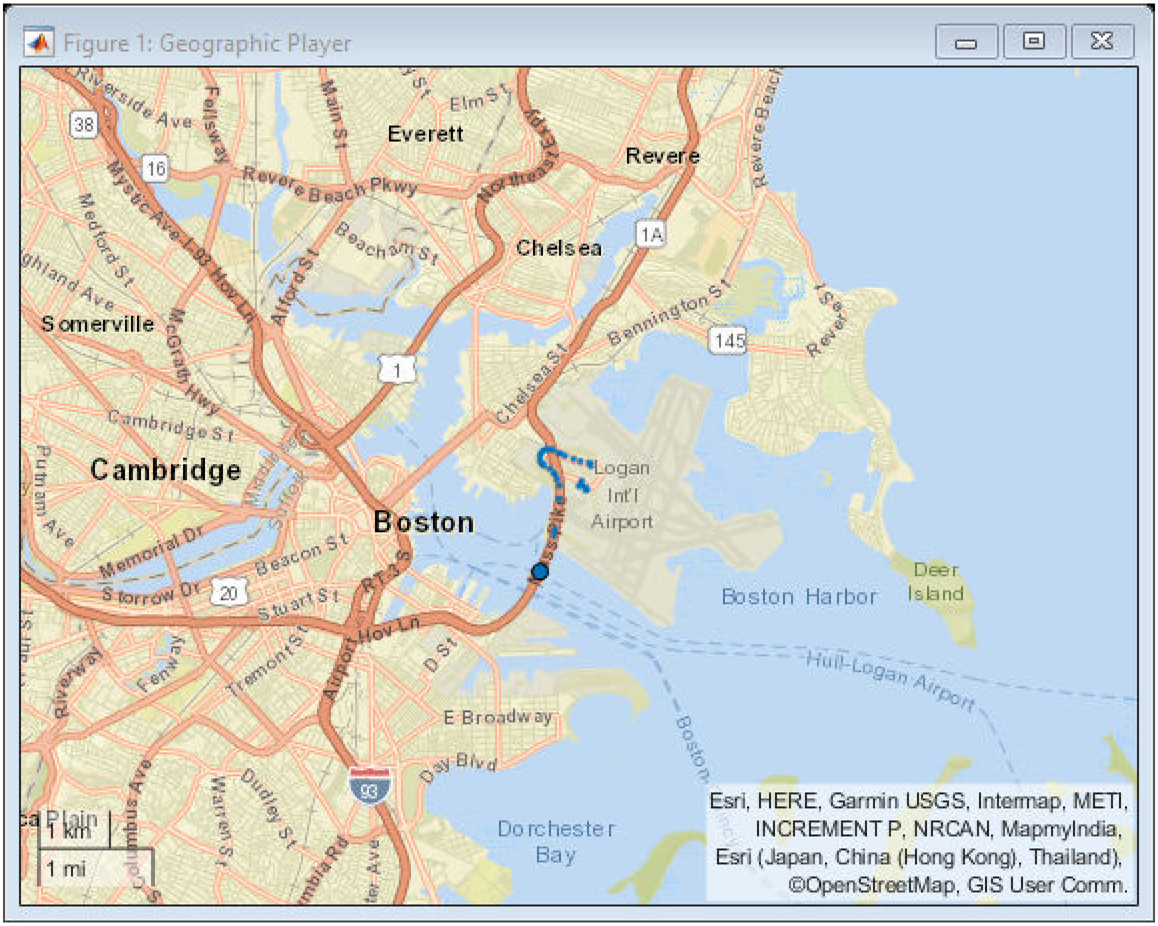
Display the geographic coordinates in a sequence by using the `plotPosition` function. Put the call to `plotPosition` inside a `while` loop, so that the geoplayer plots points only while the figure is open. You can exit the loop by closing the figure. If you do not close the figure, then the loop automatically 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
```

You can make the geoplayer figure visible again by using the `show` function.

```
show(player)
```



Input Arguments

`player` — Streaming geographic player
`geoplayer` object

Streaming geographic player, specified as a `geoplayer` object.

Output Arguments

`tf` — Geoplayer is visible
`true` | `false`

Geoplayer is visible, returned as `true` when the geoplayer figure is open, and `false` otherwise.

See Also

`hide` | `show`

Introduced in R2018a

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 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.

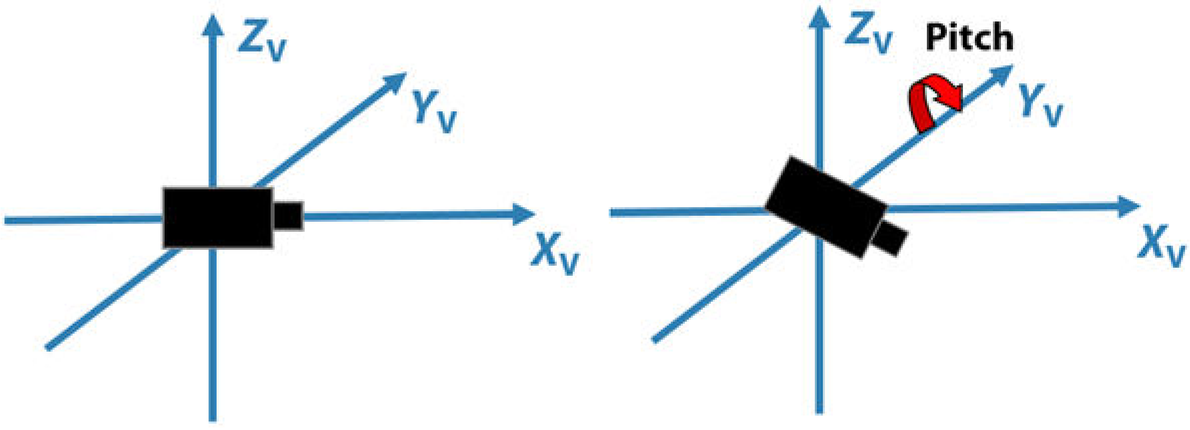
`Height` — Height from road surface to camera sensor
scalar

Height from the road surface to the camera sensor, specified as a 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
scalar

Pitch angle between the horizontal plane of the vehicle and the optical axis of the camera, specified as a 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.

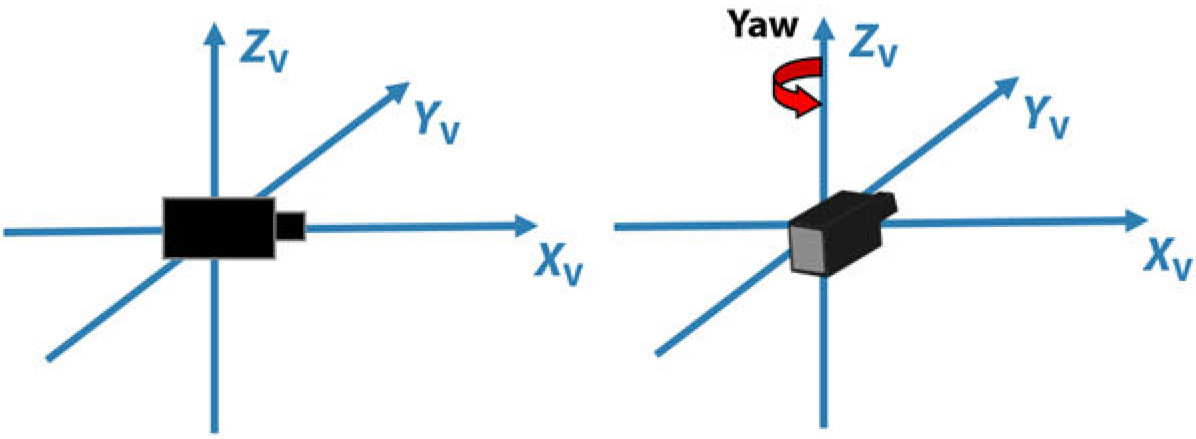


For more details, see "Angle Directions" on page 4-63.

Yaw — Yaw angle
scalar

Yaw angle between the X_V axis of the vehicle and the optical axis of the camera, specified as a 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.

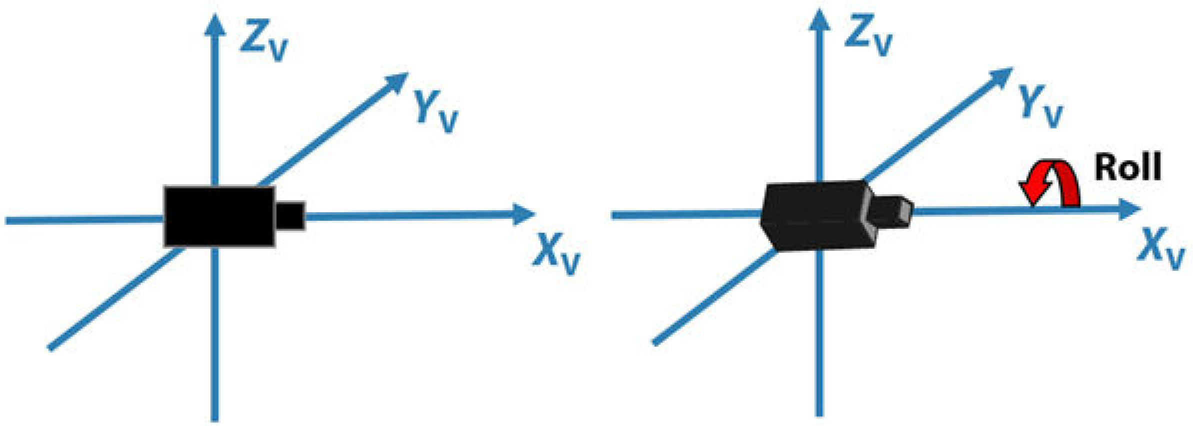


For more details, see “Angle Directions” on page 4-63.

Roll — Roll angle
scalar

Roll angle of the camera around its optical axis, returned as a 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.

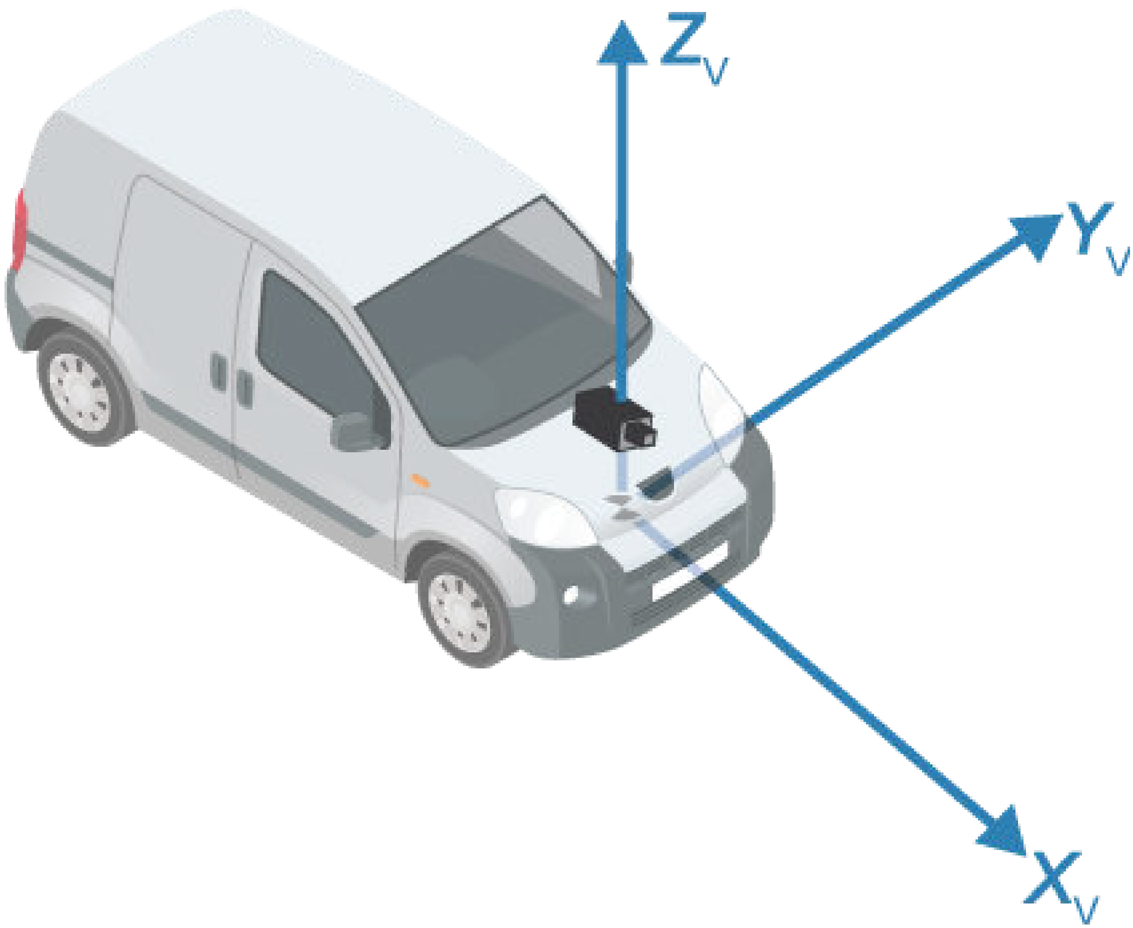


For more details, see “Angle Directions” on page 4-63.

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` 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')
```

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')
```

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 car 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.

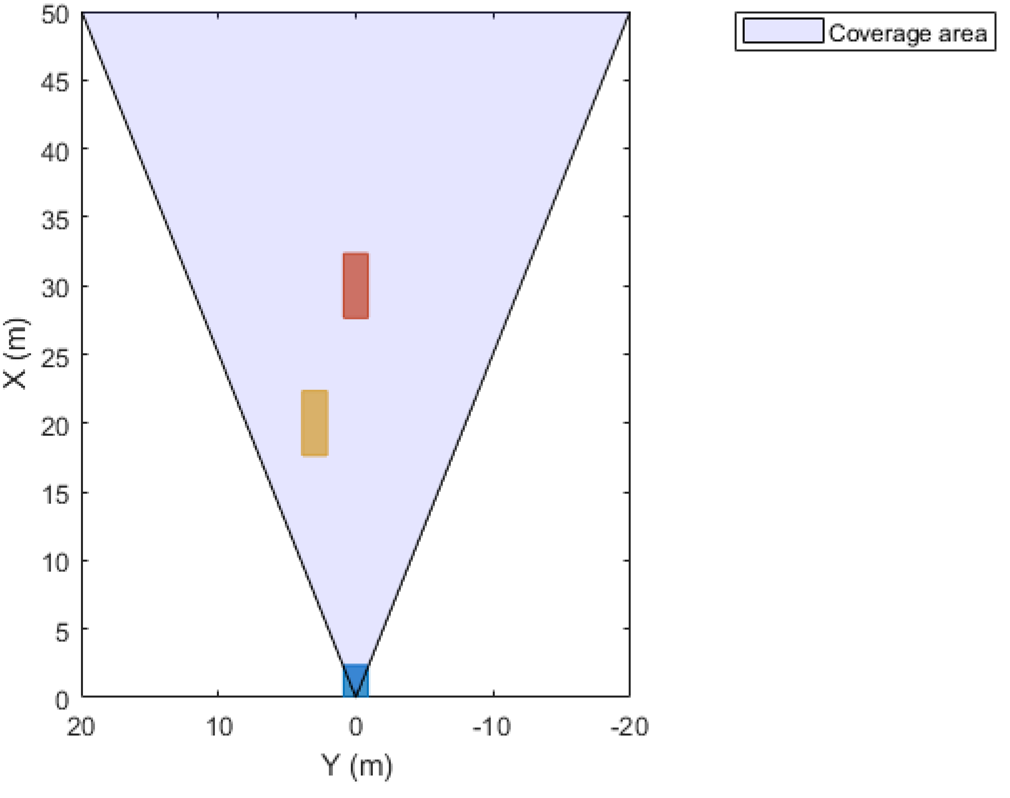
```
visionSensor = visionDetectionGenerator(monoCamConfig);
```

Generate a driving scenario with an ego car and two target cars. Position the first target car 30 meters directly in front of the ego car. Position the second target car 20 meters in front of the ego car but offset to the left by 3 meters.

```
scenario = drivingScenario;
egoCar = vehicle(scenario);
targetCar1 = vehicle(scenario,'Position',[30 0 0]);
targetCar2 = vehicle(scenario,'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(egoCar);
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 car. Use these poses to generate detections from the sensor.

```
poses = targetPoses(egoCar);
[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 = 2.77 meters
Detection 2: X = 27.81 meters, Y = 0.08 meters
```

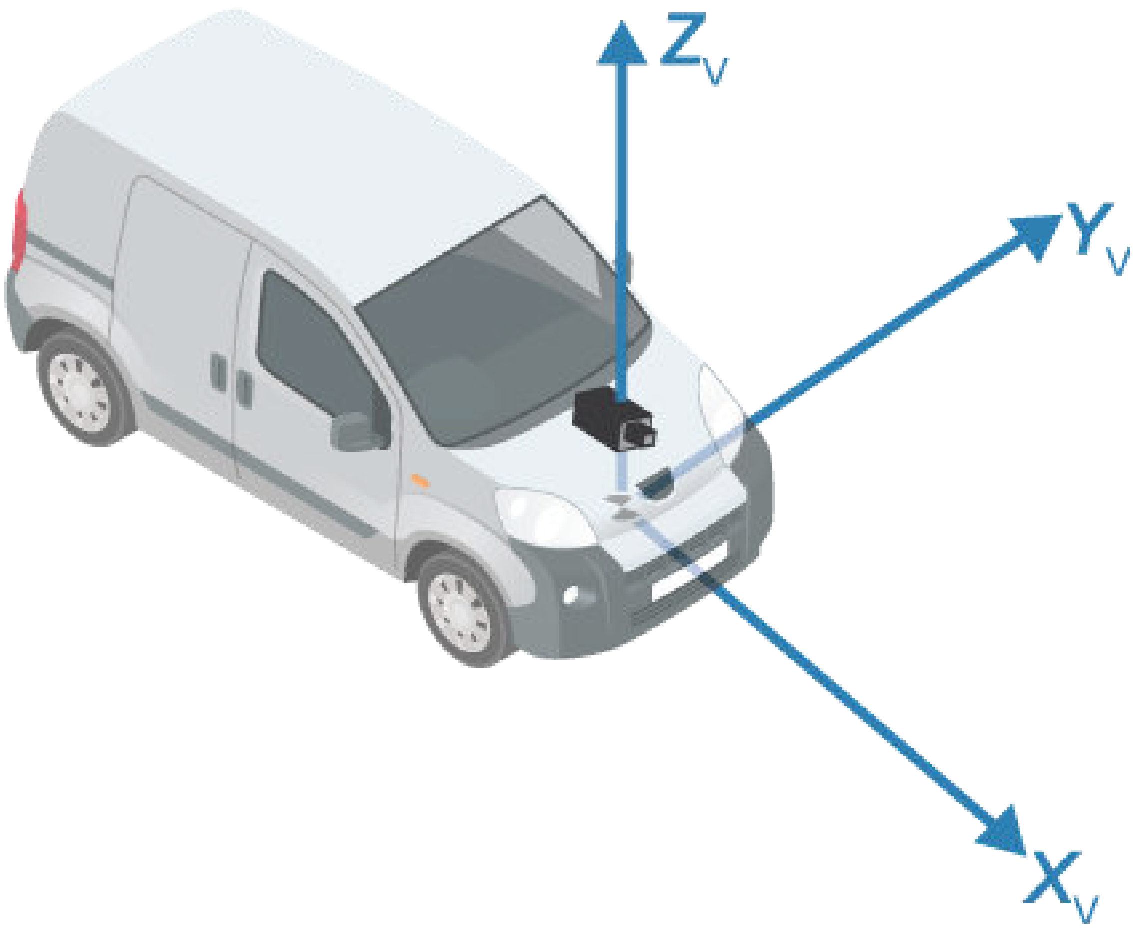
Definitions

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.

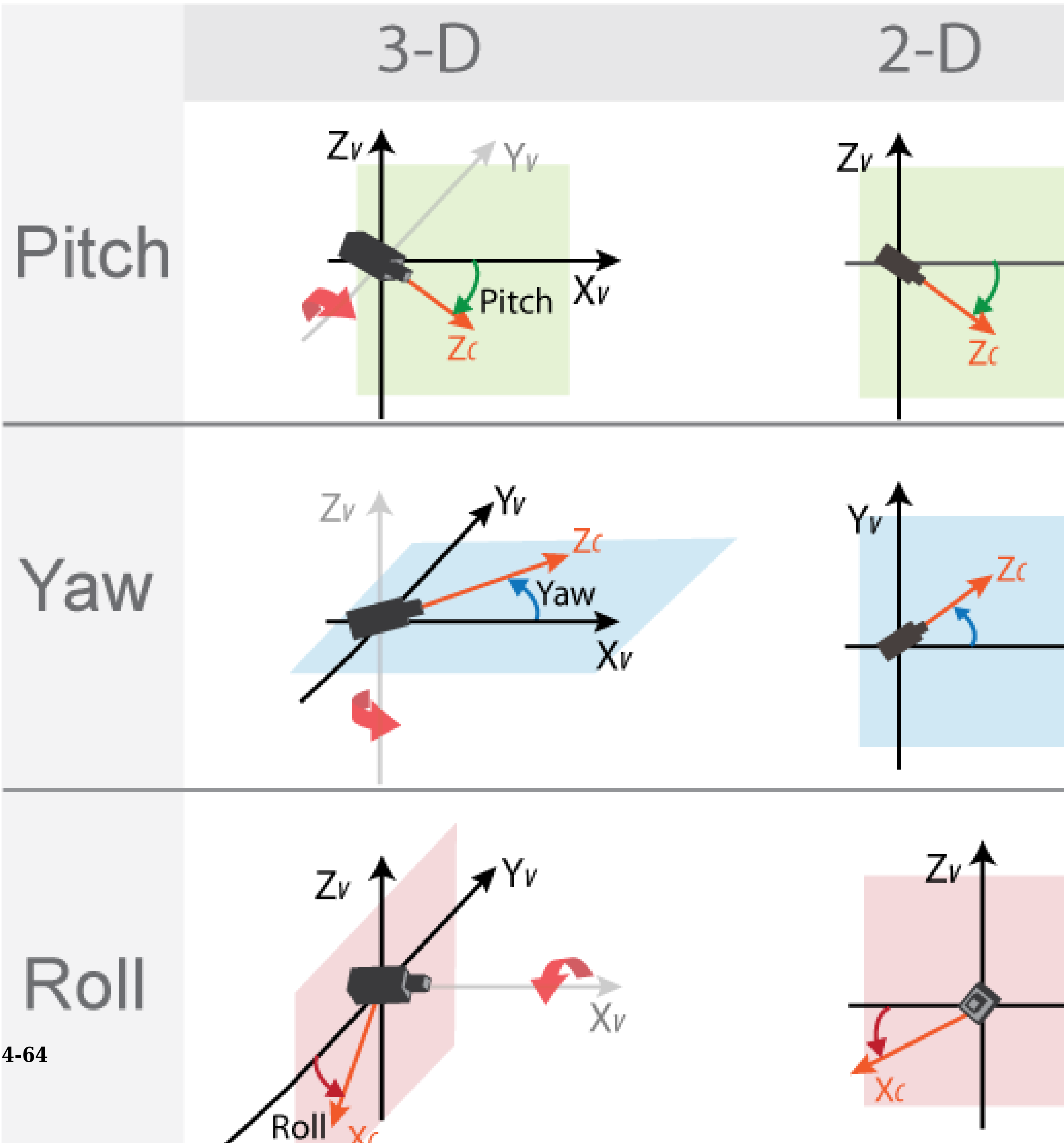


To change the placement of the origin within the vehicle coordinate system, update the `SensorLocation` property.

For more details about the vehicle coordinate system, see “Coordinate Systems in Automated Driving System 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
Pitch
Xv
Zc
Zv
Zc
Yaw
Zv
Yv
Zc
Yaw
Xv
Yv
Zc
Roll
Zv
Yv
Xv
Roll
Zv
Xc

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 System 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` 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')
```

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 System 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` 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')
```

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')
```

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 System Toolbox"

Introduced in R2017a

birdsEyePlot

Plot detections and object tracking results around vehicle

Description

The `birdsEyePlot` object displays a bird's-eye plot of a 2-D scene in the immediate vicinity of a vehicle. This type of plot can be used with sensors capable of detecting objects and lanes. For an example of how to use `birdsEyePlot`, see the “Visualize Sensor Coverage, Detections, and Tracks”.

Creation

`bep = birdsEyePlot` creates a bird’s-eye plot in a new figure.

`bep = birdsEyePlot(Name,Value)` creates a bird’s-eye plot in a new figure with optional input properties specified by one or more `Name,Value` pair arguments.

Properties

`Parent` — Axes on which to plot
axes handle

Axes on which to plot, specified as an axes handle. By default, `birdsEyePlot` uses the current axes handle, which is returned by the `gca` function.

`Plotters` — Plotters created for the bird's-eye plot
array

Plotters created for the bird's-eye plot, specified as an array.

`XLimits` — Limits of the *x*-axis
two-element row vector

Limits of the *x*-axis in vehicle coordinates, specified as a two-element row vector, [*x1*,*x2*]. The values *x1* and *x2* are the respective lower and upper limit ranges for the bird's-eye plot display. If you do not specify the limits, then the default values for the Parent axes are used. See "Coordinate Systems in Automated Driving System Toolbox" for coordinate system definitions.

`YLimits` — Limits of the *y*-axis
two-element row vector

Limits of the *y*-axis in vehicle coordinates, specified as a two-element row vector, [*y1*,*y2*]. The values *y1* and *y2* are the respective lower and upper limit ranges for the bird's-eye plot display. If you do not specify the limits, then the default values for the Parent axes are used. See "Coordinate Systems in Automated Driving System Toolbox" for coordinate system definitions.

Object Functions

Plotter Objects

clearData	Clear data from a specific plotter of bird's-eye plot
clearPlotterData	Clear data from bird's-eye plot
coverageAreaPlotter	Create bird's-eye-view coverage area plotter
detectionPlotter	Create bird's-eye-view detection plotter
findPlotter	Find plotters associated with bird's-eye plot
laneBoundaryPlotter	Create bird's-eye-view lane boundary plotter
laneMarkingPlotter	Bird's-eye plot lane marking plotter
outlinePlotter	Create bird's-eye-view outline plotter
pathPlotter	Create bird's-eye-view path plotter
trackPlotter	Create bird's-eye-view track plotter

Plotting Functions

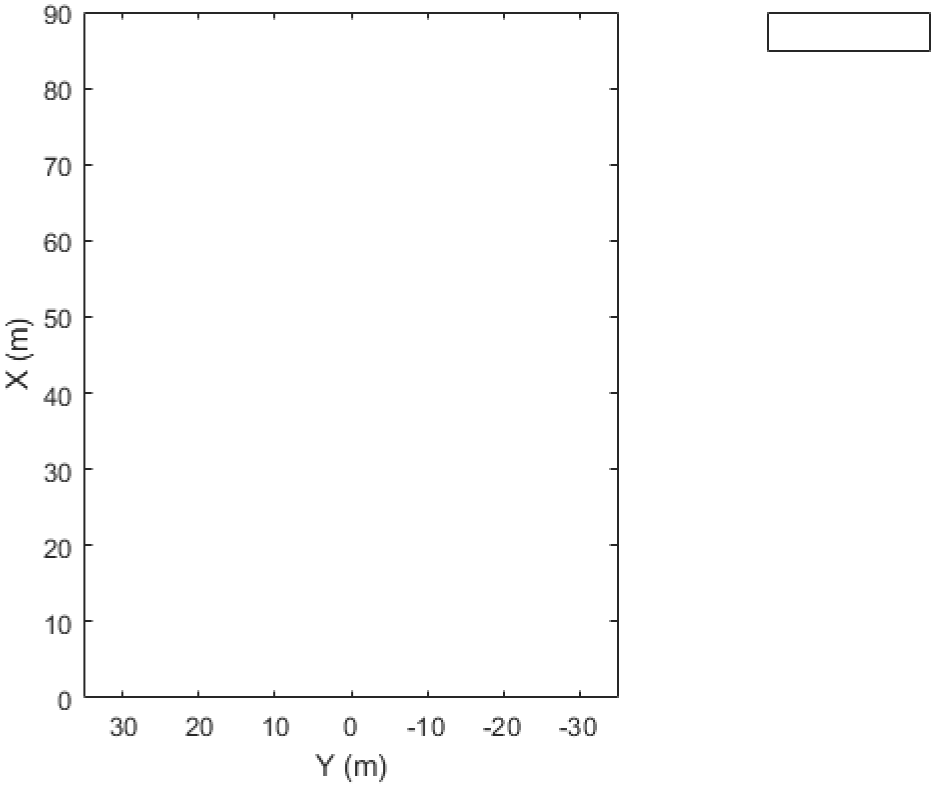
plotCoverageArea	Plot bird's-eye view coverage area
plotDetection	Plot a set of object detections
plotLaneBoundary	Plot lane boundary for bird's-eye plot
plotLaneMarking	Plot lane markings on bird's-eye plot
plotOutline	Plot object outlines
plotPath	Plot lane boundary for bird's-eye plot
plotTrack	Plot a set of detection tracks

Examples

Create and Display a Bird's-Eye Plot

Create the bird's-eye plot.

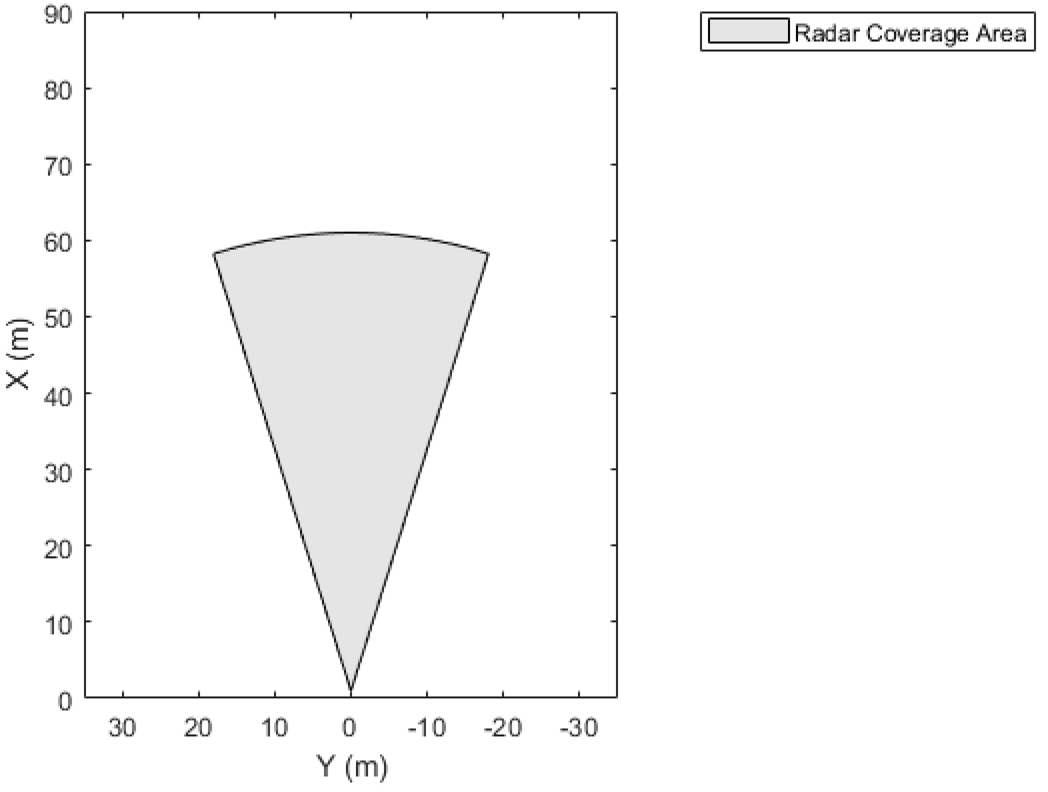
```
bep = birdsEyePlot('XLim',[0,90],'YLim',[-35,35]);
```



Display a coverage area with a field of view of 35 degrees and a range of 60 meters

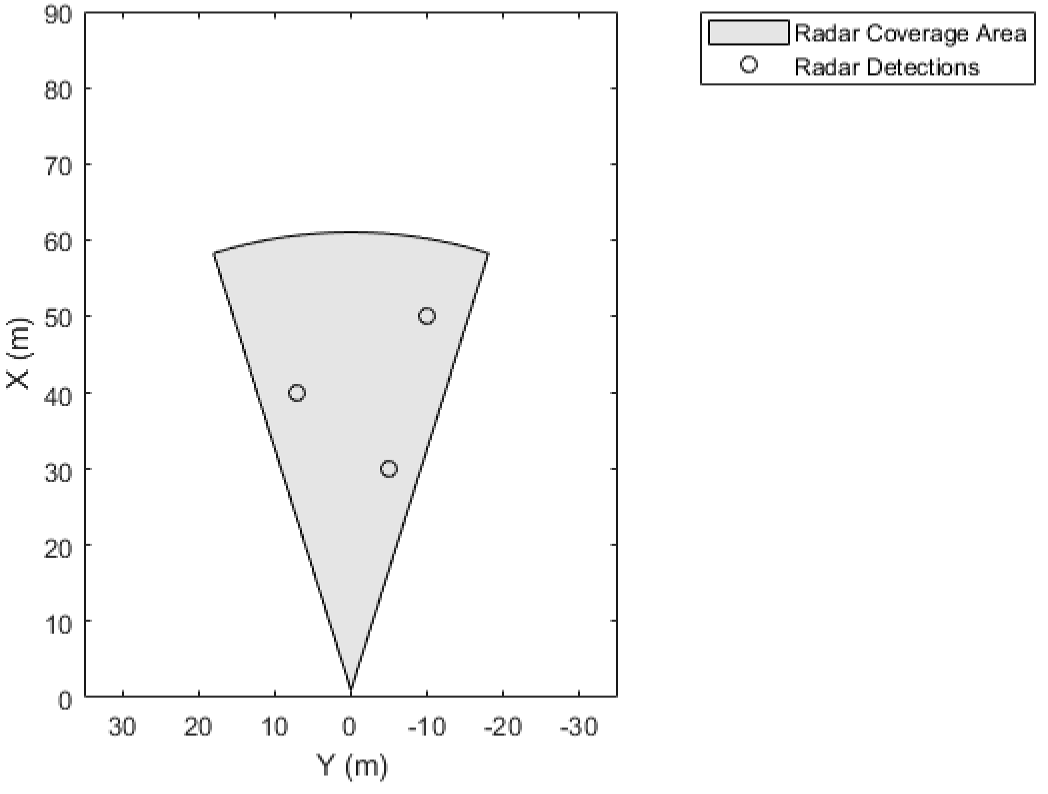
```
caPlotter = coverageAreaPlotter(bep,'DisplayName','Radar Coverage Area');
mountPosition = [1 0];
range = 60;
orientation = 0;
```

```
fieldOfView = 35;
plotCoverageArea(caPlotter,mountPosition,range,orientation,fieldOfView);
```



Display radar detections with coordinates at (30,-5),(50,-10), and (40,7).

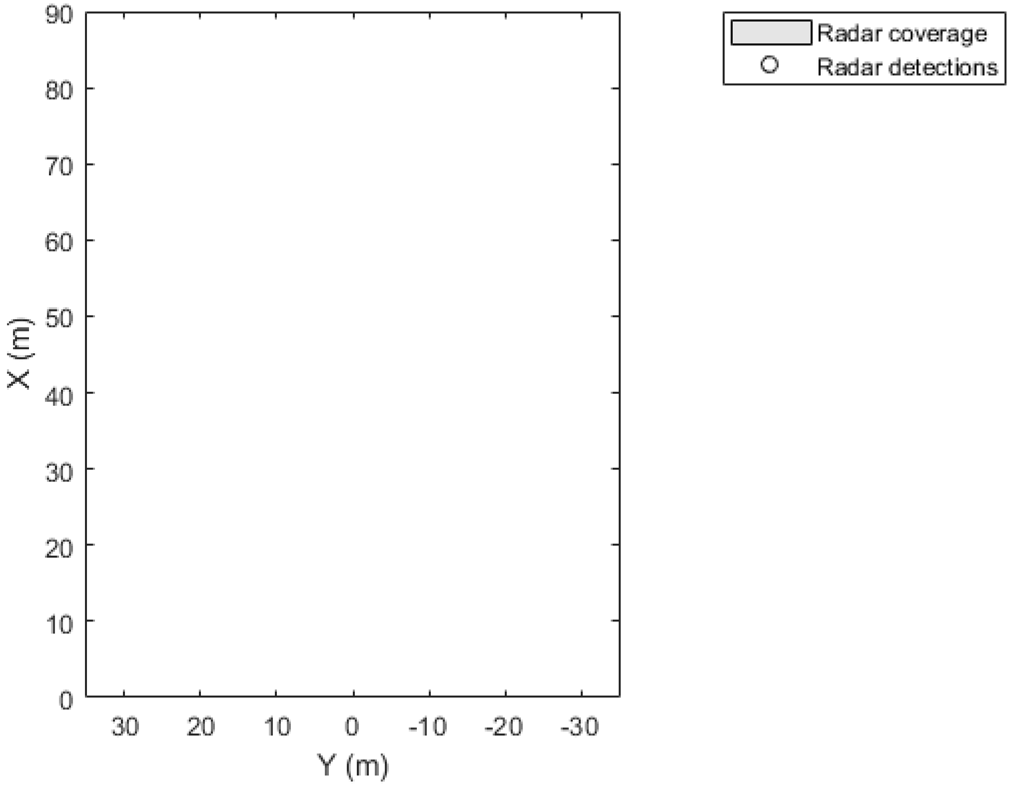
```
radarPlotter = detectionPlotter(bep,'DisplayName','Radar Detections');
plotDetection(radarPlotter, [30 -5;50 -10;40 7]);
```



Create Bird's-Eye Plot with Coverage Area and Detection Plotters

Create a bird's-eye plot with the plotters and set selected properties.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
coverageAreaPlotter(bep,'DisplayName','Radar coverage');
detectionPlotter(bep,'DisplayName','Radar detections');
```

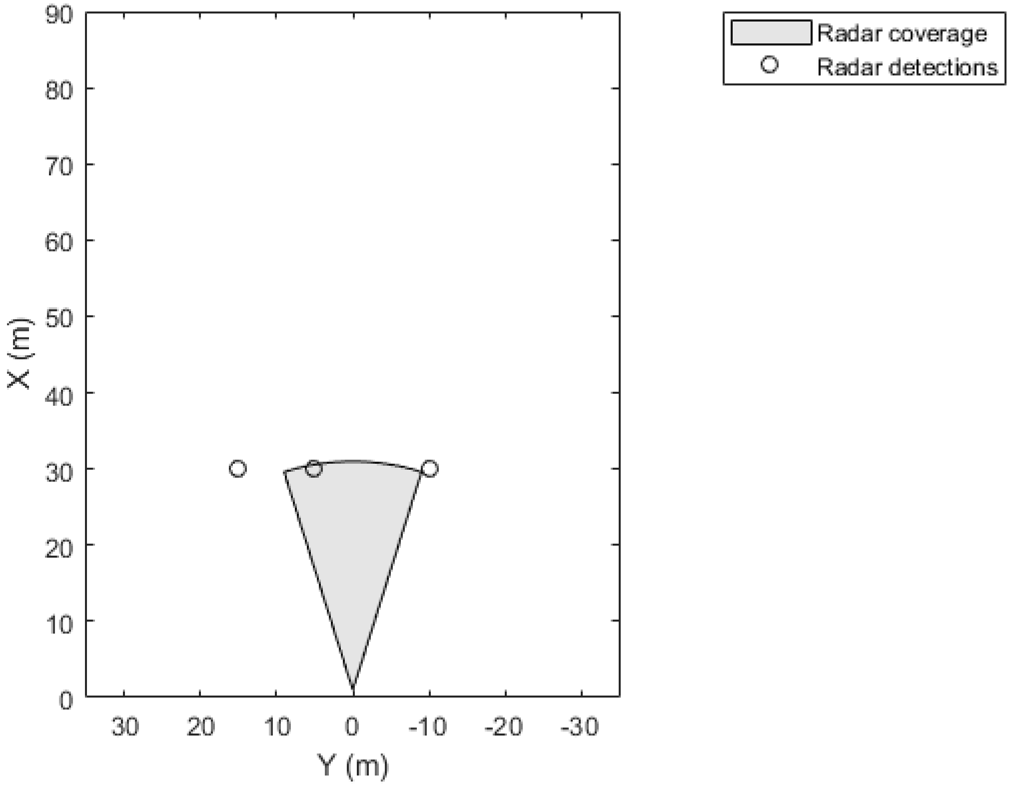


Use `findPlotter` to locate their plotters by display names.

```
caPlotter = findPlotter(bep,'DisplayName','Radar coverage');
radarPlotter = findPlotter(bep,'DisplayName','Radar detections');
```

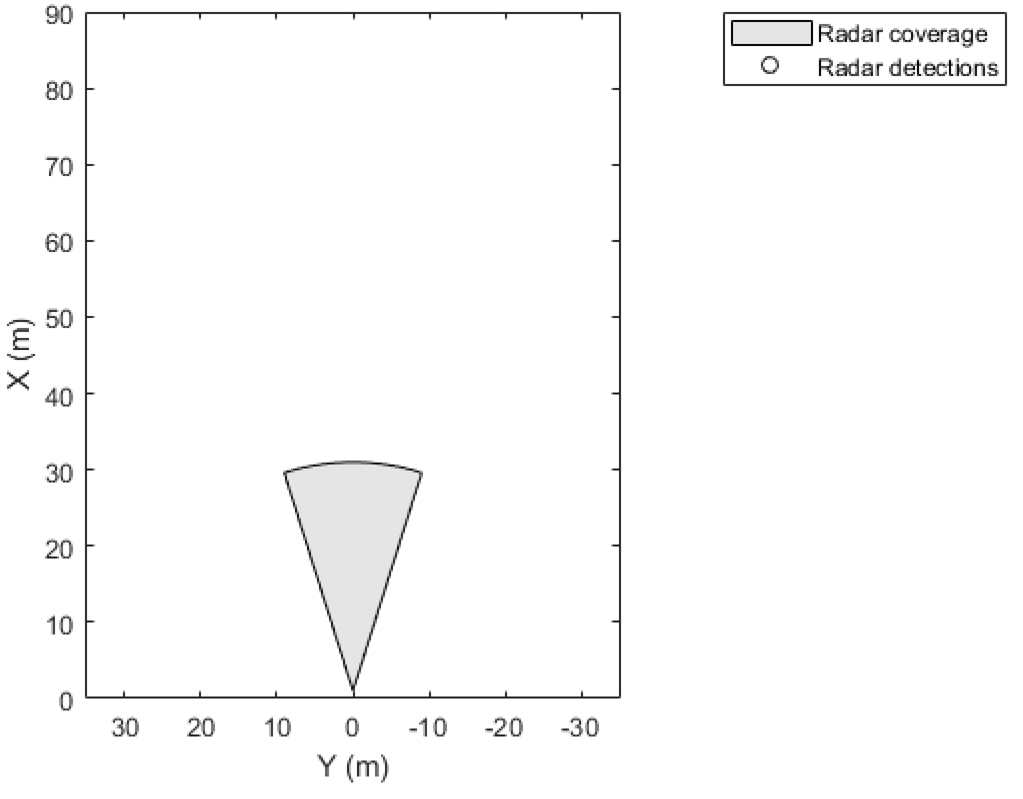
Plot the coverage area and detected objects.

```
plotCoverageArea(caPlotter, [1 0],30,0,35);
plotDetection(radarPlotter, [30,5;30,-10;30,15]);
```



Clear data from the plot.

```
clearPlotterData(bep);
```

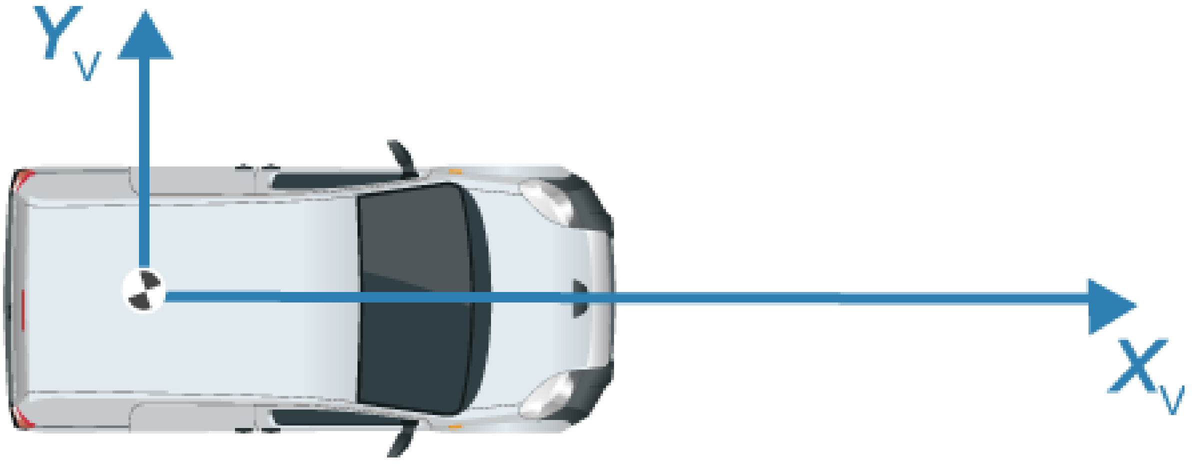


Limitations

You cannot use the rectangle-zoom feature in the `birdsEyePlot` figure.

Tips

- The vehicle coordinate system defined by `birdsEyePlot` uses the *X*-axis pointing forward from the vehicle and the *Y*-axis pointing to the left (as viewed when facing forward). The coordinate system origin is with respect to the vehicle's center of rotation, which is typically on the ground beneath the rear axle of the vehicle.



Vehicle Coordinate System

- To create and use a bird's-eye plot, follow these steps:

  1. Create a `birdsEyePlot`.
  2. Create desired plotters for coverage areas, detections, tracks, lane boundary markings, and paths using one of the `birdsEyePlot` methods.
  3. Use the plotters to update the plot with corresponding information and data.

See Also

`birdsEyeView`

Topics

"Visualize Sensor Coverage, Detections, and Tracks"
"Coordinate Systems in Automated Driving System Toolbox"

Introduced in R2017a

clearData

Clear data from a 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 method clears data from plotters created by the following plotter methods:

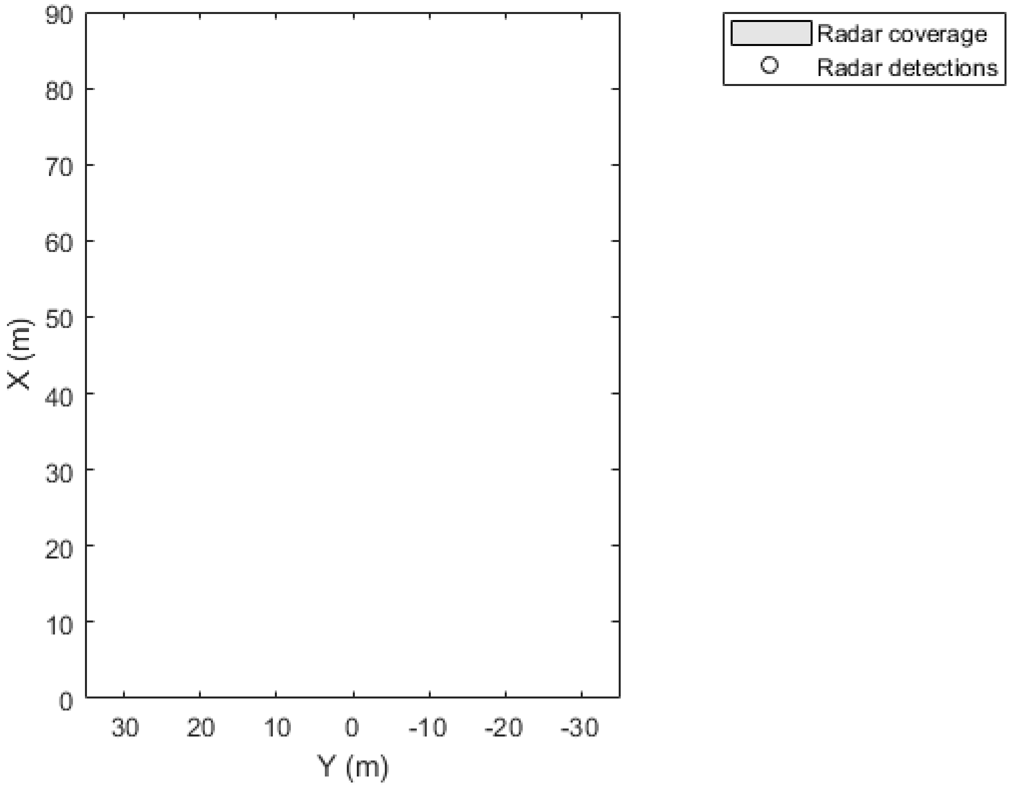
- `detectionPlotter`
- `laneBoundaryPlotter`
- `laneMarkingPlotter`
- `outlinePlotter`
- `pathPlotter`
- `trackPlotter`

Examples

Create Bird's-Eye Plot with Coverage Area and Detection Plotters

Create a bird's-eye plot with the plotters and set selected properties.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
coverageAreaPlotter(bep,'DisplayName','Radar coverage');
detectionPlotter(bep,'DisplayName','Radar detections');
```

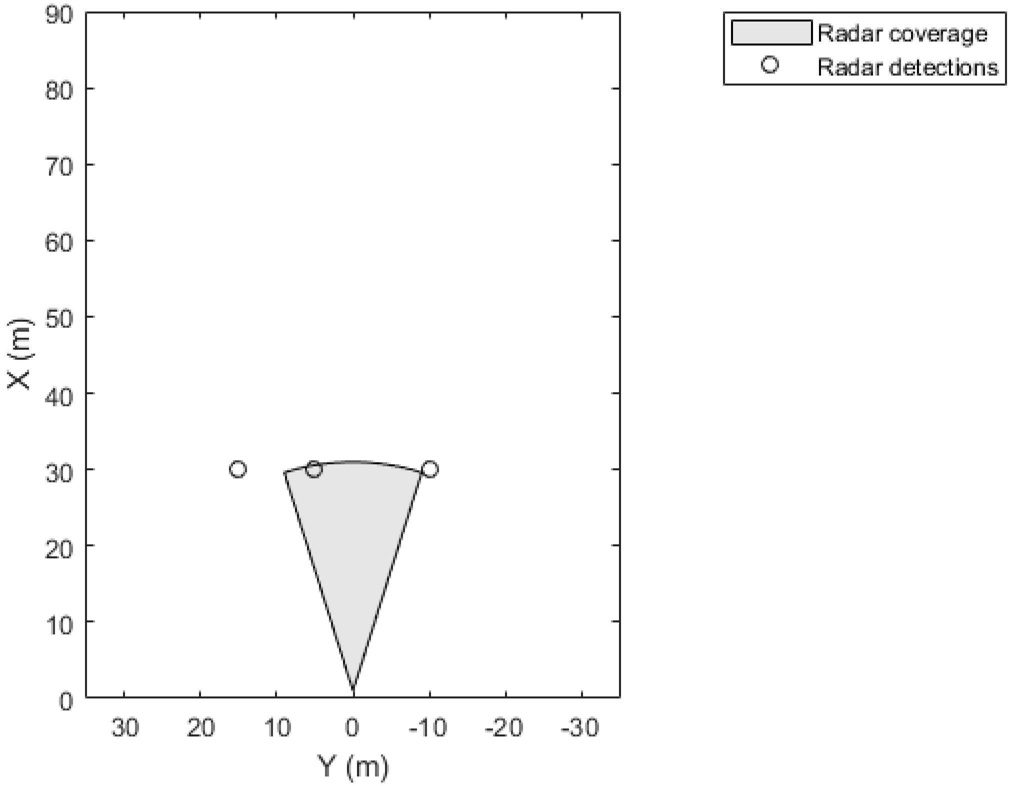


Use `findPlotter` to locate their plotters by display names.

```
caPlotter = findPlotter(bep,'DisplayName','Radar coverage');
radarPlotter = findPlotter(bep,'DisplayName','Radar detections');
```

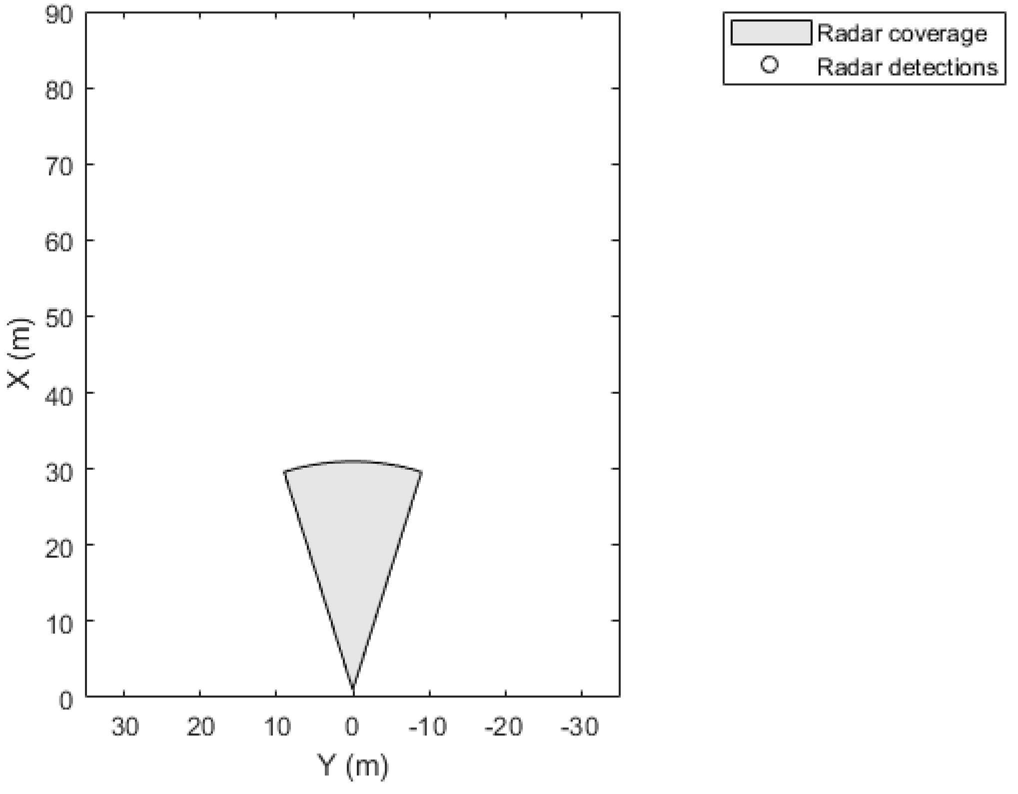
Plot the coverage area and detected objects.

```
plotCoverageArea(caPlotter, [1 0],30,0,35);
plotDetection(radarPlotter, [30,5;30,-10;30,15]);
```



Clear data from the plot.

```
clearPlotterData(bep);
```



Input Arguments

pl — Specific plotter belonging to a bird's-eye plot
`specific plotter of bird's-eye plot` handle

Specific plotter belonging to a bird's-eye plot, specified as a plotter handle of `birdsEyePlot`.

See Also

Objects
`birdsEyePlot` | `clearPlotterData`

Introduced in R2017a

clearPlotterData

Clear data from bird’s-eye plot

Syntax

`clearPlotterData(bep)`

Description

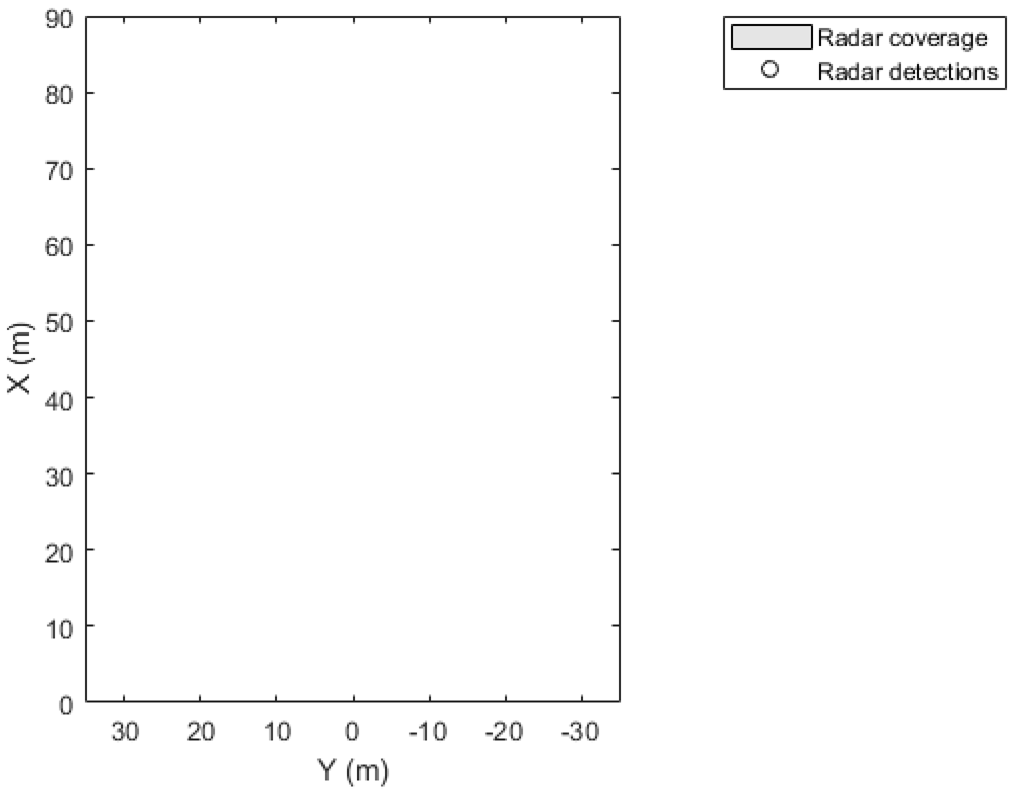
`clearPlotterData(bep)` clears data shown in the bird’s-eye plot from all the `bep` plotters. Legend entries and coverage areas are not cleared from the plot.

Examples

Create Bird's-Eye Plot with Coverage Area and Detection Plotters

Create a bird's-eye plot with the plotters and set selected properties.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
coverageAreaPlotter(bep,'DisplayName','Radar coverage');
detectionPlotter(bep,'DisplayName','Radar detections');
```

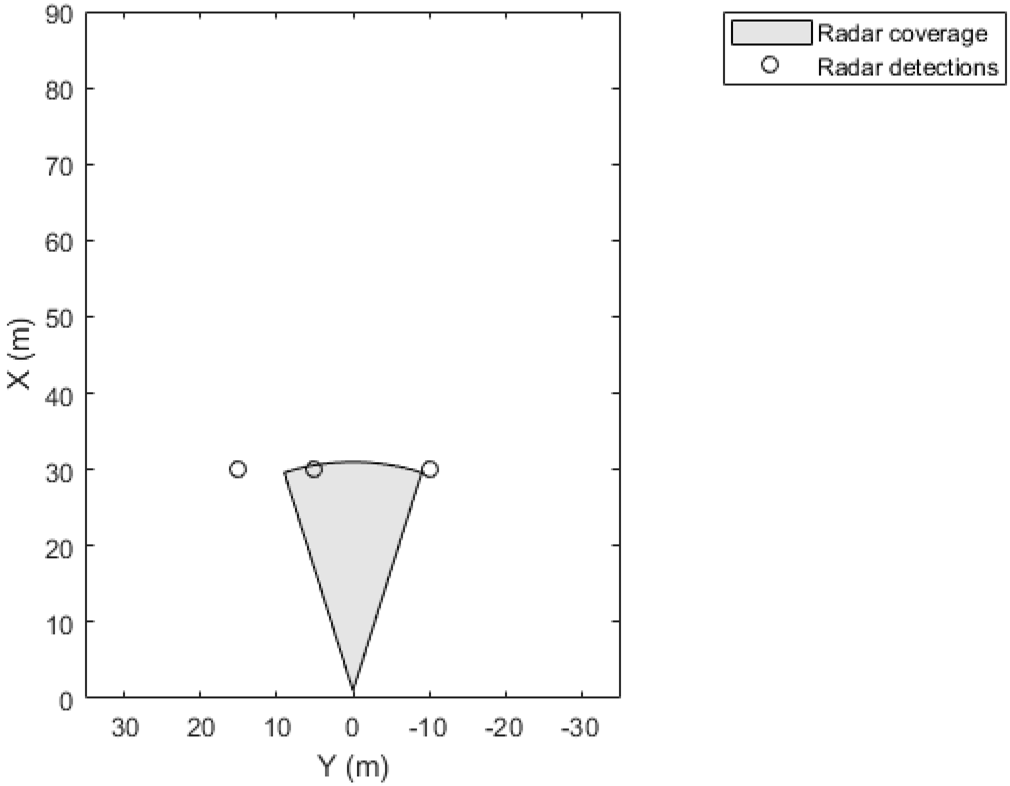


Use `findPlotter` to locate their plotters by display names.

```
caPlotter = findPlotter(bep,'DisplayName','Radar coverage');
radarPlotter = findPlotter(bep,'DisplayName','Radar detections');
```

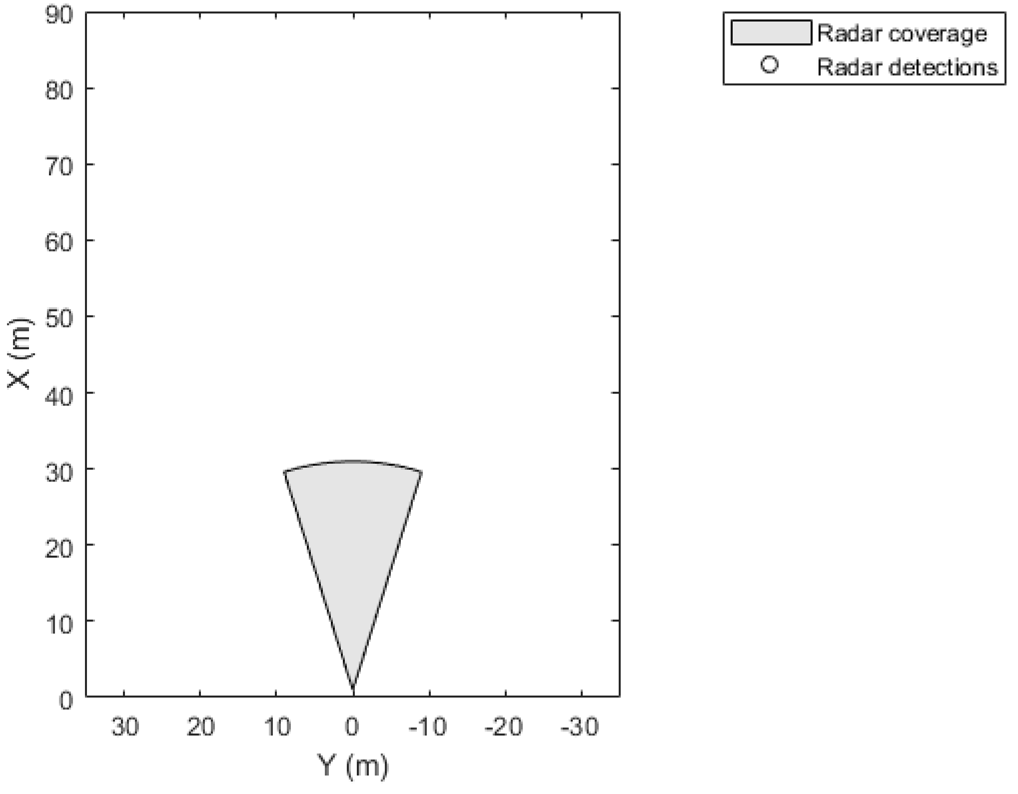
Plot the coverage area and detected objects.

```
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 — Unpopulated bird's-eye plot
`birdsEyePlot` handle

Unpopulated bird's-eye plot, specified as a `birdsEyePlot` handle that you can update with various plotters.

See Also

Functions
`birdsEyePlot`

Introduced in R2017a

coverageAreaPlotter

Create bird’s-eye-view coverage area plotter

Syntax

```
caPlotter = coverageAreaPlotter(bep)
caPlotter = coverageAreaPlotter(bep,Name,Value)
```

Description

`caPlotter = coverageAreaPlotter(bep)` returns a plotter for displaying the coverage area of a bird’s-eye plot.

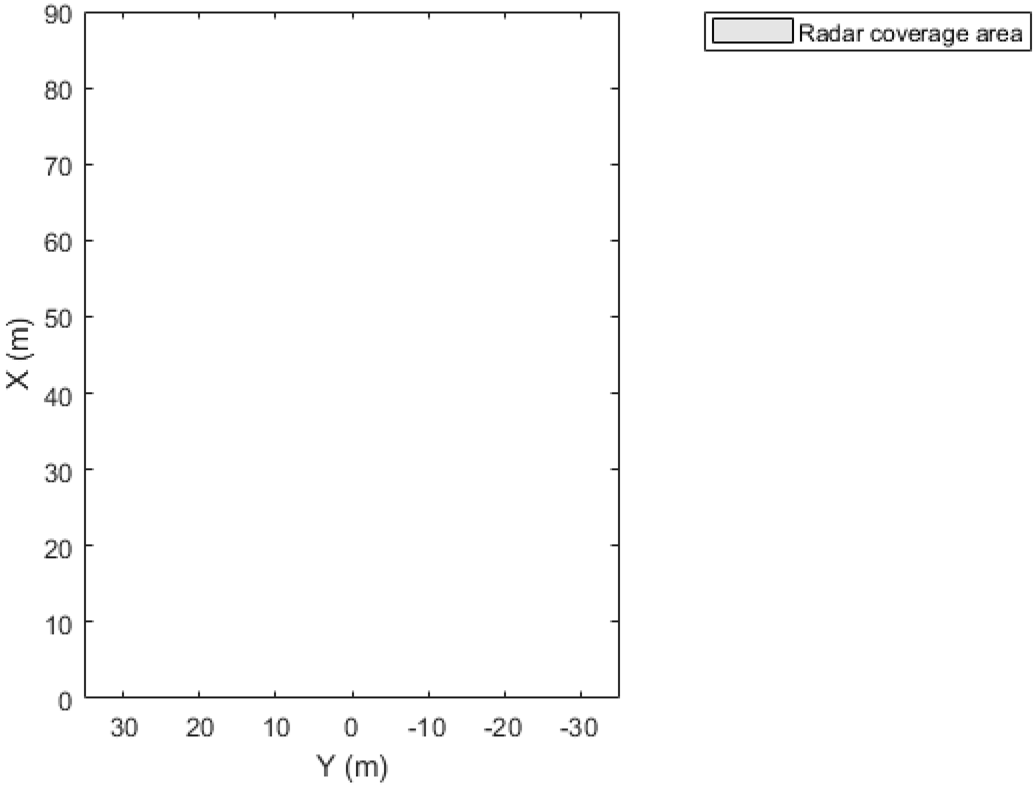
`caPlotter = coverageAreaPlotter(bep,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Create and Display Coverage Area Bird's-Eye Plot

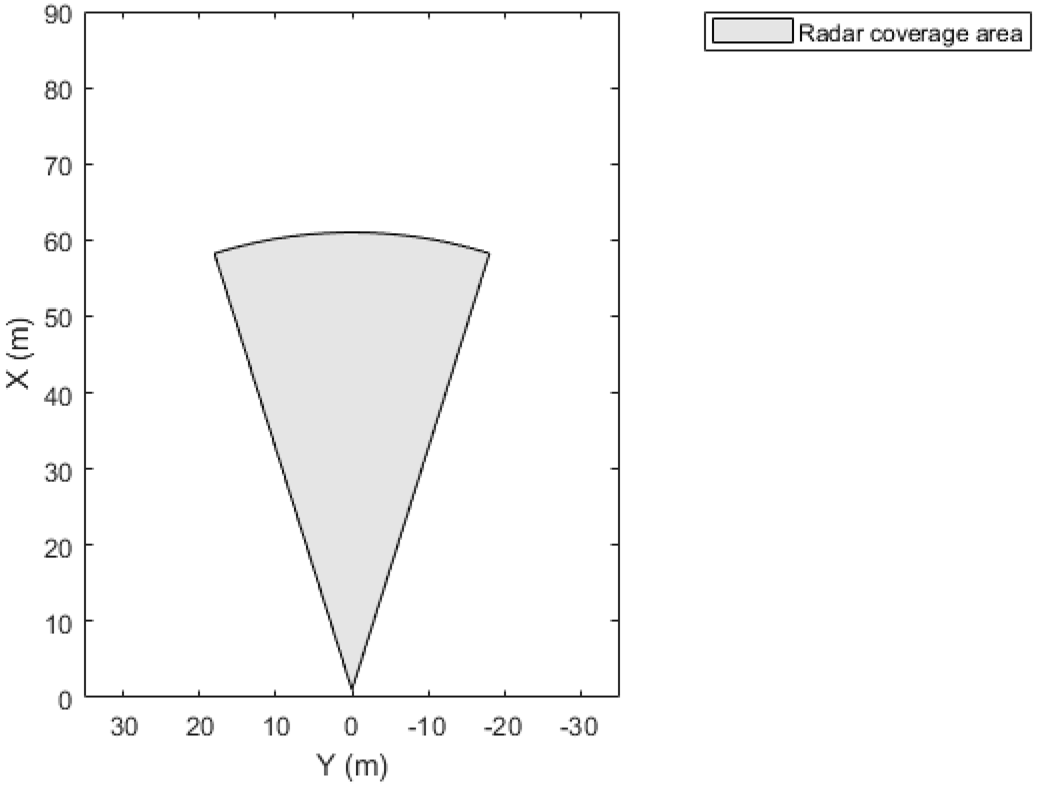
Create a bird's-eye plot and coverage area plotter.

```
bep = birdsEyePlot('XLim',[0, 90],'YLim',[-35, 35]);
caPlotter = coverageAreaPlotter(bep,'DisplayName','Radar coverage area');
```



Update the plotter with a 35-degree field of view and a 60-meter range.

```
mountPosition = [1 0];
range = 60;
orientation = 0;
fieldOfView = 35;
plotCoverageArea(caPlotter,mountPosition,range,orientation,fieldOfView);
```



Input Arguments

bep — Unpopulated bird's-eye plot
`birdsEyePlot` handle

Unpopulated bird's-eye plot, specified as a `birdsEyePlot` handle that you can update with various plotters.

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: `'FaceColor'`,`'black'`.

`DisplayName` — Plot name to display in legend
character vector | string scalar

Plot name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and a character vector or string scalar. If you do not specify a name, no entry is displayed.

`FaceColor` — Coverage area color
`'black'` (default) | character vector | string scalar | [RGB] vector

Coverage area color, specified as the comma-separated pair consisting of `'FaceColor'` and a character vector, string scalar, or an [RGB] vector.

`EdgeColor` — Coverage area border color
`'black'` (default) | character vector | string scalar | [RGB] vector

Coverage area border color, specified as the comma-separated pair consisting of `'EdgeColor'` and a character vector, string scalar, or an [RGB] vector.

`FaceAlpha` — Transparency of coverage area
`1` (default) | scalar in the range [0,1]

Transparency of coverage area, 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, and a value of `1` makes it fully opaque.

`Tag` — Tag to identify plot of coverage area
`'PlotterN'` (default) | character vector | string scalar

Tag used to identify the plot of the coverage area, specified as the comma-separated pair consisting of `'Tag'` and a character vector or string scalar. The default `'Tag'` used is,`'PlotterN'`, where *N* is an integer.

Output Arguments

`caPlotter` — Bird's-eye plot of coverage area
plotter object

Bird's-eye plot of coverage area, returned as a plotter object. To plot the coverage area, specify `caPlotter` as an input to `plotCoverageArea`.

See Also

Functions
`birdsEyePlot` | `plotCoverageArea`

Introduced in R2017a

detectionPlotter

Create bird’s-eye-view detection plotter

Syntax

```
detPlotter = detectionPlotter(bep)
detPlotter = detectionPlotter(bep,Name,Value)
```

Description

`detPlotter = detectionPlotter(bep)` returns a detection plotter for displaying detections in a bird’s-eye plot.

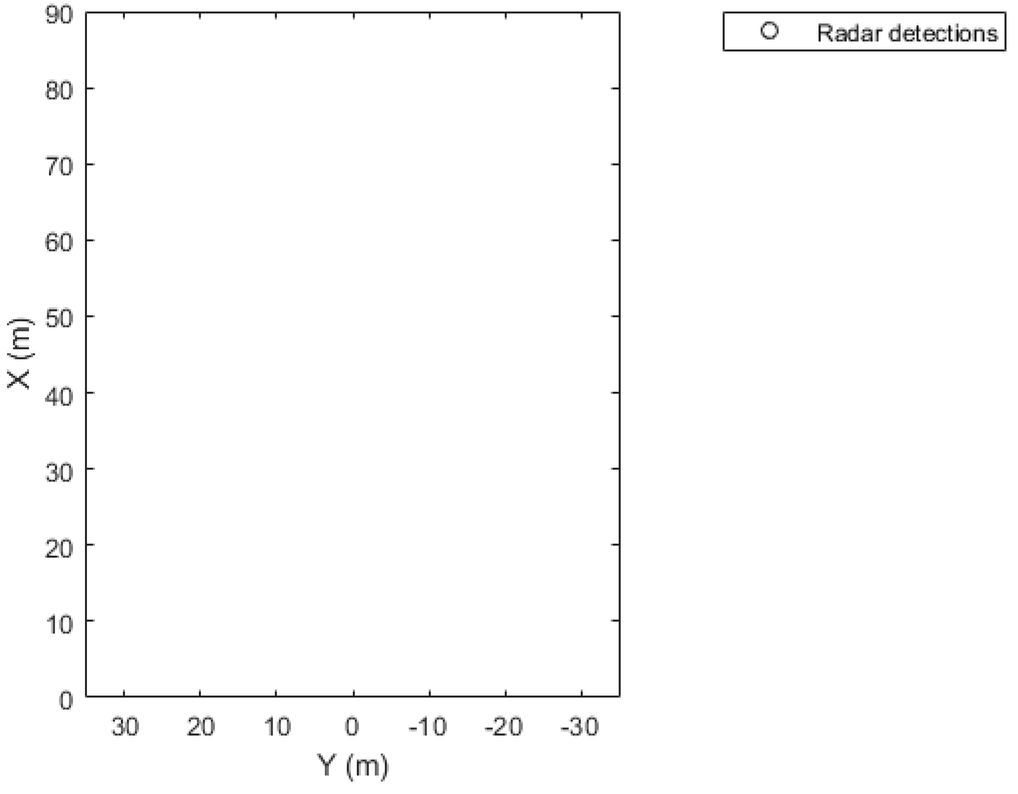
`detPlotter = detectionPlotter(bep,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Create Bird's-Eye Plot with Labeled Detections

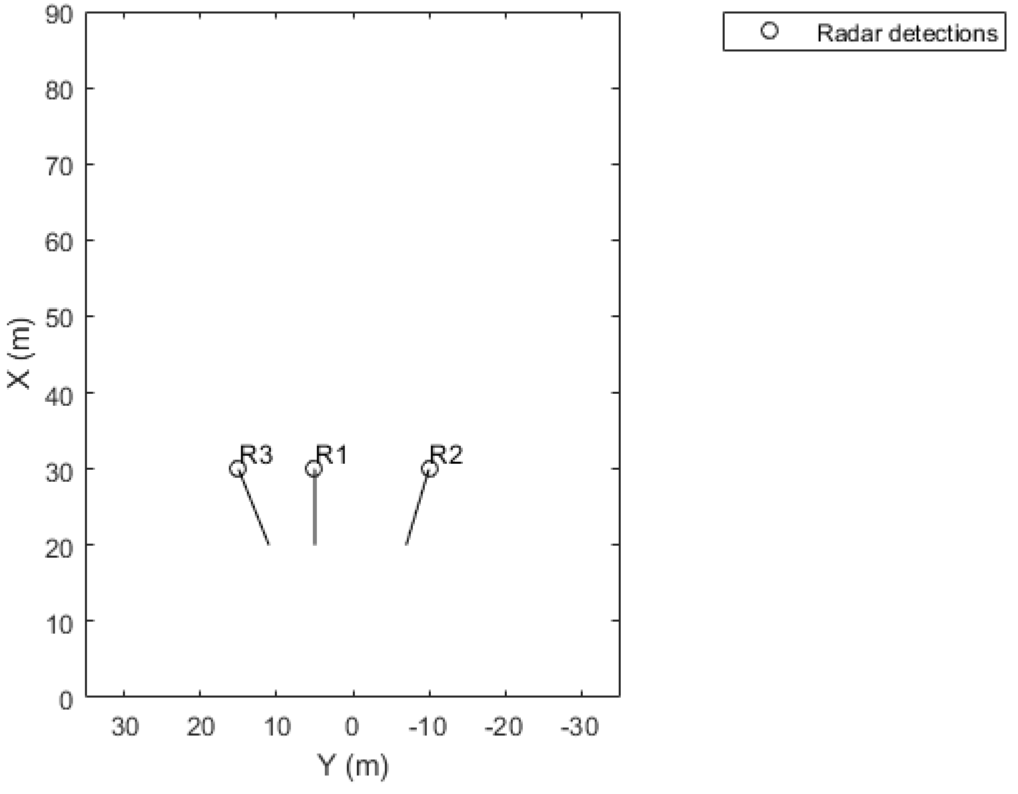
Create a bird's-eye plot and radar plotter.

```
bep = birdsEyePlot('XLim',[0,90],'YLim',[-35,35]);
radarPlotter = detectionPlotter(bep,'DisplayName','Radar detections');
```



Label the detections, positioned in meters, with corresponding velocities.

```
positions = [30,5;30,-10;30,15];
velocities = [-10,0;-10,3;-10,-4];
labels = {'R1','R2','R3'};
plotDetection(radarPlotter,positions,velocities,labels);
```



Input Arguments

bep — Unpopulated bird's-eye plot
`birdsEyePlot` handle

Unpopulated bird's-eye plot, specified as a `birdsEyePlot` handle that you can update with various plotters.

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'`.

`DisplayName` — Plot name to display in legend

character vector | string scalar

Plot name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and a character vector or string scalar. If you do not specify a name, no entry is displayed.

`Marker` — Marker symbol

`'o'` (default) | character

Marker symbol, specified as the comma-separated pair consisting of `'Marker'` and one of these symbols.

Value	Description
`'.'`	Point
`'x'`	Cross
`'+'`	Plus sign
`'*'`	Asterisk
`'o'`	Circle (default)
`'s'`	Square
`'d'`	Diamond
`'h'`	Six-pointed star (hexagram)
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'<'`	Left-pointing triangle
`'>'`	Right-pointing triangle

MarkerSize — Size of marker
positive integer

Size of marker, specified as the comma-separated pair consisting of 'MarkerSize' and a positive integer.

MarkerEdgeColor — Marker outline color
'black' (default) | character vector | string scalar | [RGB] vector

Marker outline color, specified as the comma-separated pair consisting of 'MarkerEdgeColor' and a character vector, string scalar, or an [RGB] vector.

MarkerFaceColor — Marker fill color
character vector | string scalar | [RGB] vector | 'none'

Marker outline color, specified as the comma-separated pair consisting of ''MarkerFaceColor' and a character vector, string scalar, [RGB] vector, or 'none'.

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 that represents font points.

LabelOffset — Gap between label and positional point
[0 0] (default) | two-element row vector

Gap between label and positional point, specified as the comma-separated pair consisting of 'LabelOffset' and a two-element row vector. You must specify the [*x* *y*] offset in meters.

VelocityScaling — Scale factor for magnitude length of velocity vectors
1 (default) | positive scalar

Scale factor for magnitude length of velocity vectors, specified as the comma-separated pair consisting of 'VelocityScaling' and a positive scalar. The plot renders the magnitude vector value as (*magnitude of velocity*) × VelocityScaling.

Tag — Tag to identify plot of coverage area
'Plotter*N*' (default) | character vector | string scalar

Tag to identify plot of coverage area, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. The default 'Tag' used is,'Plotter*N*', where *N* is an integer.

Output Arguments

detPlotter — Detection plotter to use for bird's-eye plot
plotter object

Detection plotter to use for bird's-eye plot, returned as a plotter object.

See Also

Functions
birdsEyePlot

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.

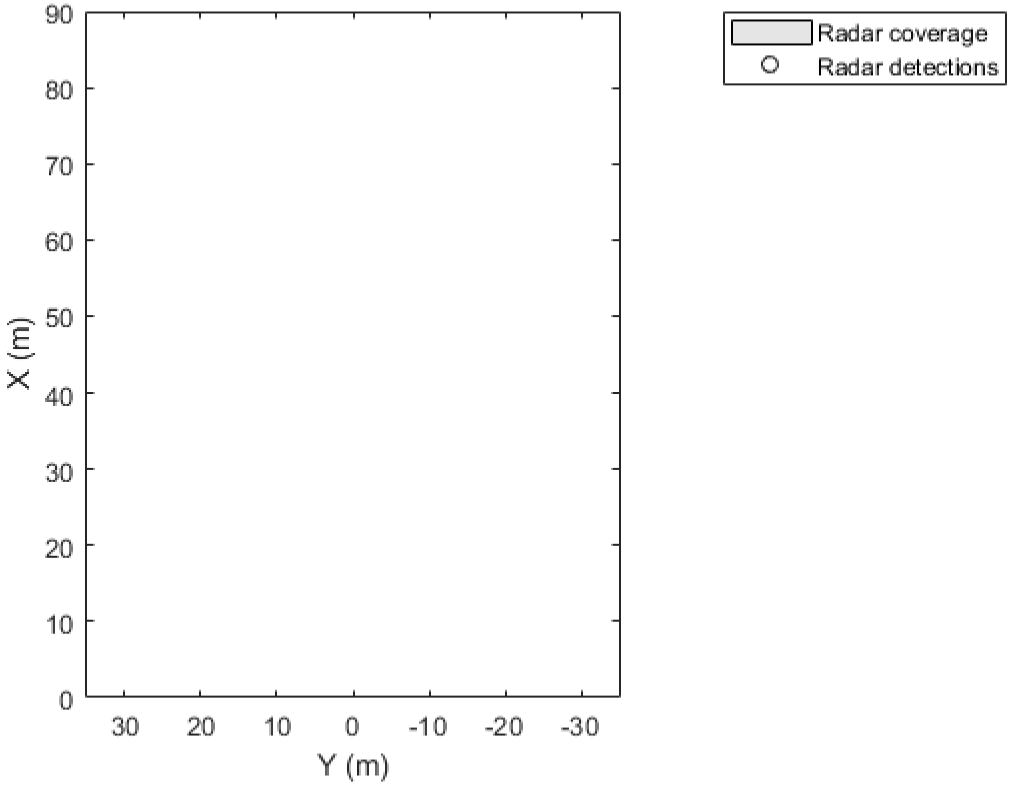
`p = findPlotter(bep,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Create Bird's-Eye Plot with Coverage Area and Detection Plotters

Create a bird's-eye plot with the plotters and set selected properties.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
coverageAreaPlotter(bep,'DisplayName','Radar coverage');
detectionPlotter(bep,'DisplayName','Radar detections');
```

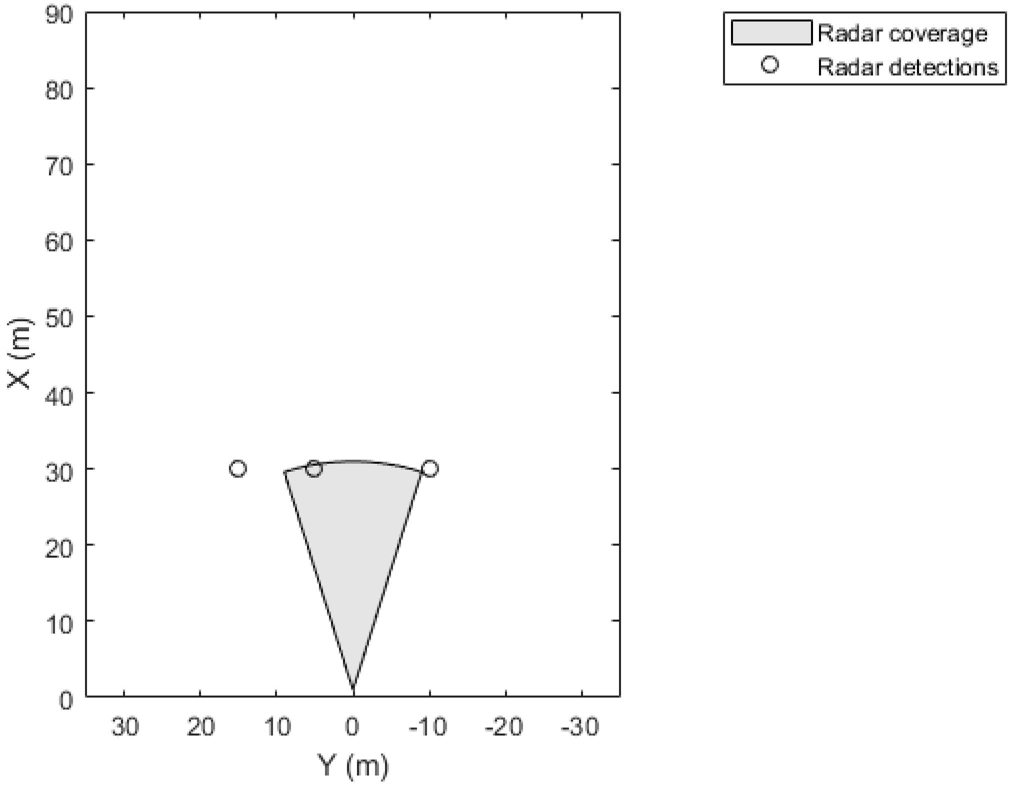


Use `findPlotter` to locate their plotters by display names.

```
caPlotter = findPlotter(bep,'DisplayName','Radar coverage');
radarPlotter = findPlotter(bep,'DisplayName','Radar detections');
```

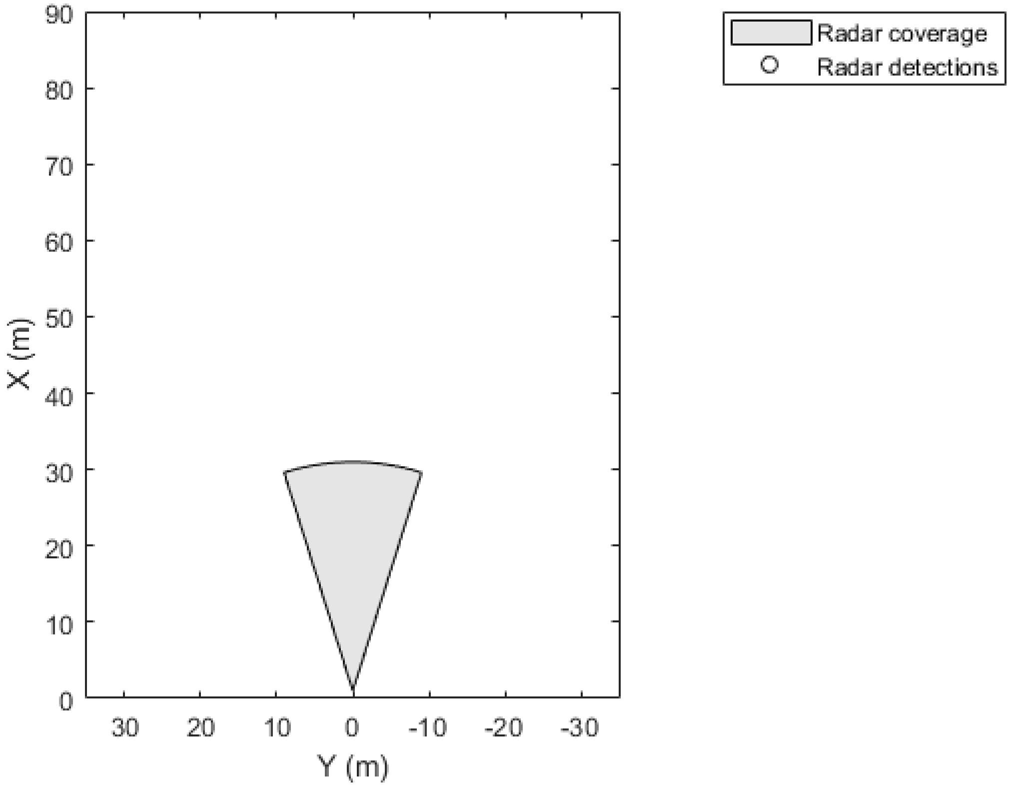
Plot the coverage area and detected objects.

```
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 — Unpopulated bird's-eye plot
`birdsEyePlot` handle

Unpopulated bird's-eye plot, specified as a `birdsEyePlot` handle that you can update with various plotters.

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` — Plot name to display in legend
character vector | string scalar

Plot name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and a character vector or string scalar. If you do not specify a name, no entry is displayed.

Tag — Tag to identify plot of coverage area
`'PlotterN'` (default) | character vector | string scalar

Tag used to identify the plot of the coverage area, specified as the comma-separated pair consisting of `'Tag'` and a character vector or string scalar. The default `'Tag'` used is `'PlotterN'`, where *N* is an integer.

Output Arguments

p — Plotters associated with bird's-eye plot
array of plotters

Plotters associated with a bird's-eye plot, returned as an array of plotters.

See Also

Functions
`birdsEyePlot`

Introduced in R2017a

laneBoundaryPlotter

Create bird’s-eye-view lane boundary plotter

Syntax

```
lbPlotter = laneBoundaryPlotter(bep)
lbPlotter = laneBoundaryPlotter(bep,Name,Value)
```

Description

`lbPlotter = laneBoundaryPlotter(bep)` returns a lane boundary plotter for displaying lane boundaries in a bird’s-eye plot.

`lbPlotter = laneBoundaryPlotter(bep,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

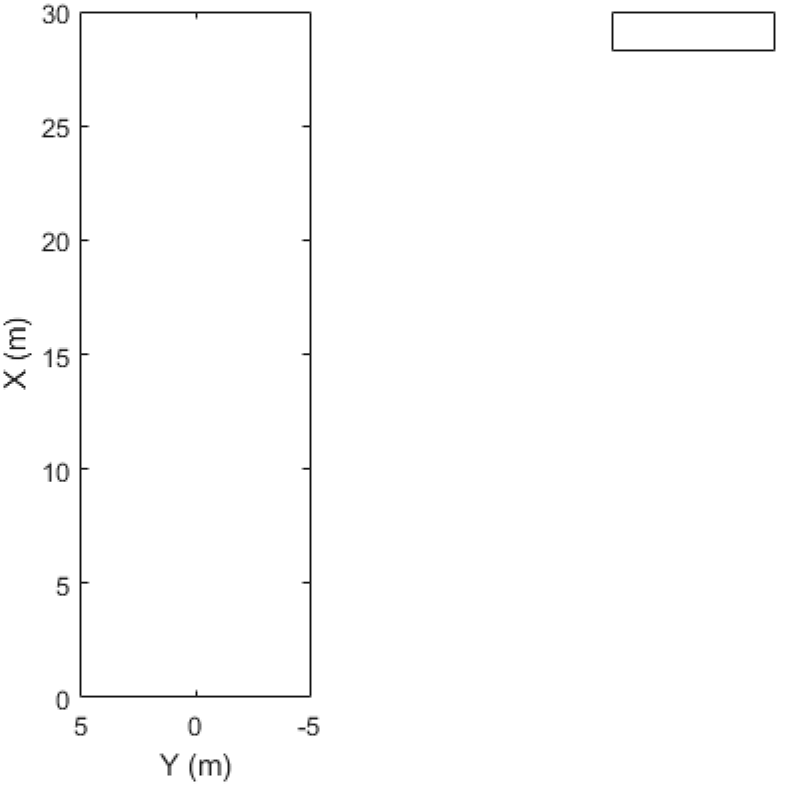
Create Bird's-Eye Plot Containing Two Lane Boundaries

Create the left-lane and right-lane boundaries.

```
lb = parabolicLaneBoundary([-0.001,0.01, 0.5]);
rb = parabolicLaneBoundary([-0.001,0.01,-0.5]);
```

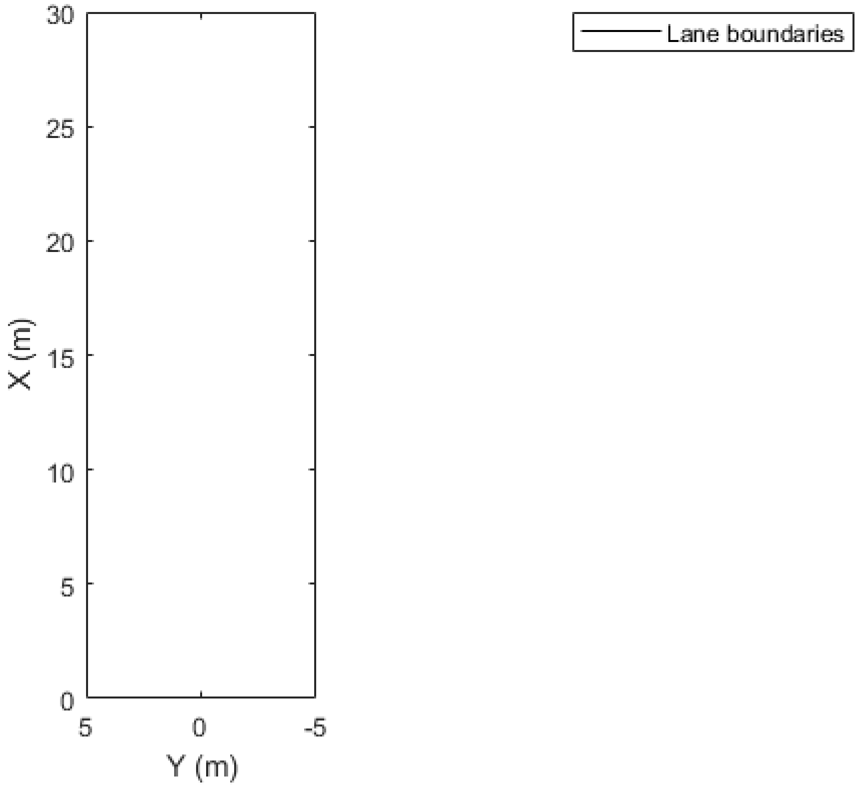
Create the bird's-eye plot.

```
bep = birdsEyePlot('XLimits',[0 30],'YLimits',[-5 5]);
```



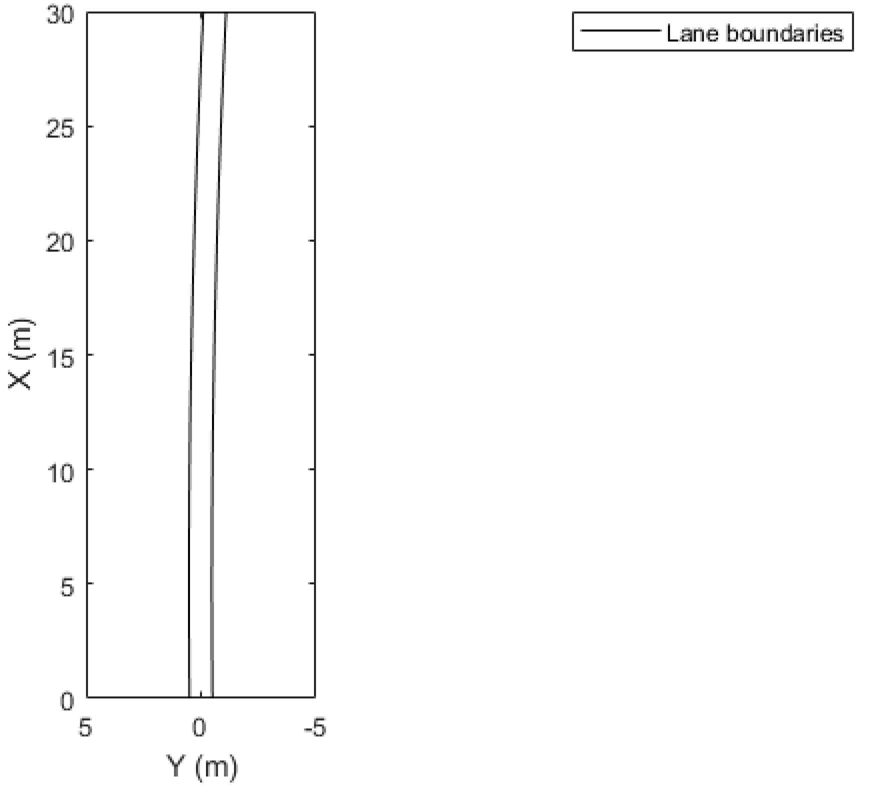
Create the lane boundary plotter.

```
lbPlotter = laneBoundaryPlotter(bep,'DisplayName','Lane boundaries');
```



Plot the lane boundaries.

```
plotLaneBoundary(lbPlotter,[lb,rb]);
```



Input Arguments

bep — Unpopulated bird's-eye plot
`birdsEyePlot` handle

Unpopulated bird's-eye plot, specified as a `birdsEyePlot` handle that you can update with various plotters.

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'`,`'black'`.

`DisplayName` — Plot name to display in legend

character vector | string scalar

Plot name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and a character vector or string scalar. If you do not specify a name, no entry is displayed.

`Color` — Boundary color

`'black'` (default) | character vector | string scalar | [RGB] vector

Boundary color, specified as the comma-separated pair consisting of `'FaceColor'` and a character vector, string scalar, or an [RGB] vector.

`LineStyle` — Boundary line style

`'-'` (default) | `'--'` | `':'` | `'-.'`

Boundary line style, specified as the comma-separated pair consisting of `'LineStyle'` and one of these styles.

Marker Symbol	Type
`'-'`	Solid line (default)
`'--'`	Dashed line
`':'`	Dotted line
`'-.'`	Dashed-dotted line

`Tag` — Tag to identify plot of coverage area

`'Plotter`*`N`*`'` (default) | character vector | string scalar

Tag used to identify the plot of the coverage area, specified as the comma-separated pair consisting of `'Tag'` and a character vector or string scalar. The default `'Tag'` used is,`'Plotter`*`N`*`'`, where *N* is an integer.

Output Arguments

`lbPlotter` — Lane boundary plotter
plotter object

Lane boundary plotter to use for bird's-eye plot, returned as a plotter object.

See Also

Functions
`birdsEyePlot`

Introduced in R2017a

laneMarkingPlotter

Bird’s-eye plot lane marking plotter

Syntax

```
lmPlotter = laneMarkingPlotter(bep)
lmPlotter = laneMarkingPlotter(bep,Name,Value)
```

Description

`lmPlotter = laneMarkingPlotter(bep)` returns a lane boundary plotter for displaying lane markings in a bird’s-eye plot.

`lmPlotter = laneMarkingPlotter(bep,Name,Value)` also enables you to specify additional options using one or more `Name,Value` pair arguments. `Name` can also be a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Examples

Generate Object and Lane Boundary Detections

Create a driving scenario containing an ego car and a target vehicle traveling along a three-lane road. Detect the lane boundaries using a vision sensor.

```
sc = drivingScenario;
```

Create a three-lane road using lane specifications.

```
roadCenters = [0 0 0; 60 0 0; 120 30 0];
lspc = lanespec(3);
road(sc,roadCenters,'Lanes',lspc);
```

The ego car follows the center lane at 30 m/s.

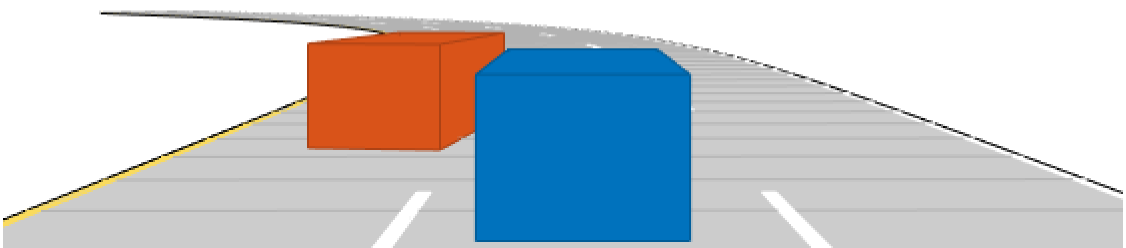
```
egocar = vehicle(sc);
egopath = [1.5 0 0; 60 0 0; 111 25 0];
egospeed = 30;
trajectory(egocar,egopath,egospeed);
```

The target vehicle travels ahead at 40 m/s and changes lanes close to the ego vehicle.

```
targetcar = vehicle(sc,'ClassID',2);
targetpath = [8 2; 60 -3.2; 120 33];
targetspeed = 40;
trajectory(targetcar,targetpath,targetspeed);
```

Display a chase plot showing a 3-D view from behind the ego vehicle.

```
chasePlot(egocar)
```



Create a vision detection generator that detects lanes and objects. The pitch of the sensor points one degree downward.

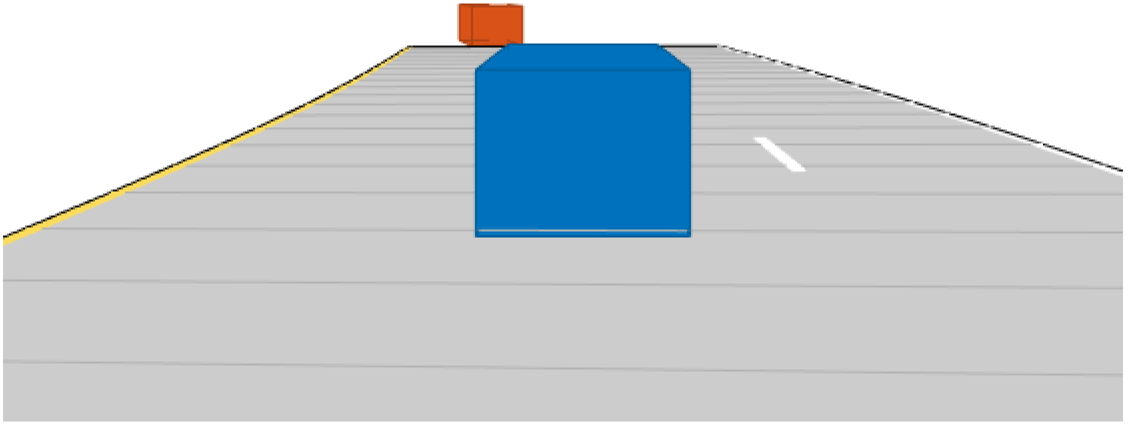
```
visionSensor = visionDetectionGenerator('Pitch',1.0);
visionSensor.DetectorOutput = 'Lanes and objects';
visionSensor.ActorProfiles = actorProfiles(sc);
```

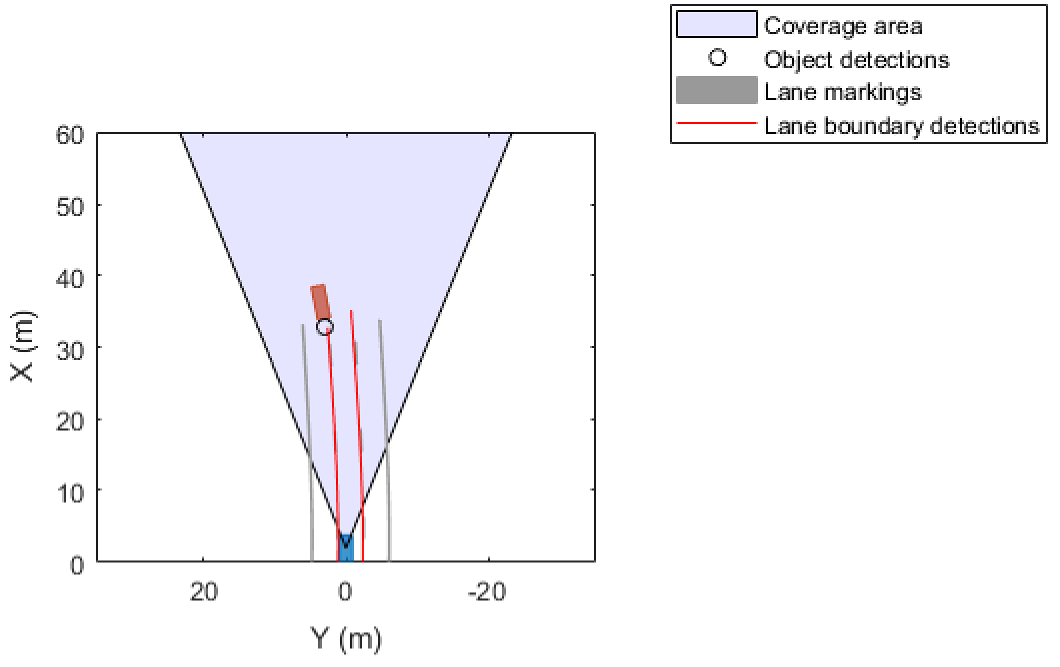
Run the simulation.

- Create a bird's eye plot and the associated plotters.
- Plot the sensor coverage area.
- Display lane markings.
- Obtain ground truth poses of targets on the road.
- Obtain ideal lane boundary points up to 60 m ahead.
- Generate detections from the ideal target poses and lane boundaries.
- Plot outline of target.
- Plot object detections when the object detection is valid.
- Plot 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(sc)
    [lmv,lmf] = laneMarkingVertices(egocar);
    plotLaneMarking(lmPlotter,lmv,lmf)
    tgtpose = targetPoses(egocar);
    lookaheadDistance = 0:0.5:60;
    lb = laneBoundaries(egocar,'XDistance',lookaheadDistance,'LocationType','inner');
    [obdets,nobdets,obValid,lb_dets,nlb_dets,lbValid] = ...
        visionSensor(tgtpose,lb,sc.SimulationTime);
    [objposition,objyaw,objlength,objwidth,objriginOffset,color] = targetOutlines(egoca
    plotOutline(olPlotter,objposition,objyaw,objlength,objwidth, ...
        'OriginOffset',objriginOffset,'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 — Empty bird's-eye plot
`birdsEyePlot` object

Empty bird's-eye plot, specified as a `birdsEyePlot` object to which you can add different plotters.

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'`,`'black'`

`DisplayName` — Name to show in legend
character vector | string scalar

Name to show in legend, specified as the comma-separated pair consisting of `'DisplayName'` and a character vector or string scalar. If you do not specify a name, the legend is empty.

`FaceColor` — Face color of lane marking patches
`'black'` (default) | MATLAB color string | `[r g b]` vector

Face color of lane marking patches, specified as the comma-separated pair consisting of `'FaceColor'` and a MATLAB color string or an `[r g b]` vector.

Tag — Plotter identification tag
`'PlotterN'` (default) | character vector | string scalar

Tag used to identify the plot of the coverage area, specified as the comma-separated pair consisting of `'Tag'` and a character vector or string scalar. Tags provide a way to identify plotter objects, for example, when searching for plotters using `findPlotter`.

By default, when no tags are assigned, `'Tag'` is constructed using `'PlotterN'`. *N* is an integer assigned sequentially as each plotter is created.

Output Arguments

`lmPlotter` — Lane marking plotter
`laneMarkingPlotter` object

Lane marking plotter to add to a bird’s-eye plot, returned as a `laneMarkingPlotter` object.

See Also

`birdsEyePlot` | `plotLaneMarking`

Introduced in R2018a

pathPlotter

Create bird’s-eye-view path plotter

Syntax

```
pPlotter = pathPlotter(bep)
pPlotter = pathPlotter(bep,Name,Value)
```

Description

`pPlotter = pathPlotter(bep)` returns a path plotter for displaying paths in a bird’s-eye plot.

`pPlotter = pathPlotter(bep,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Plot Path of Ego Vehicle

Create a 3-meter-wide lane.

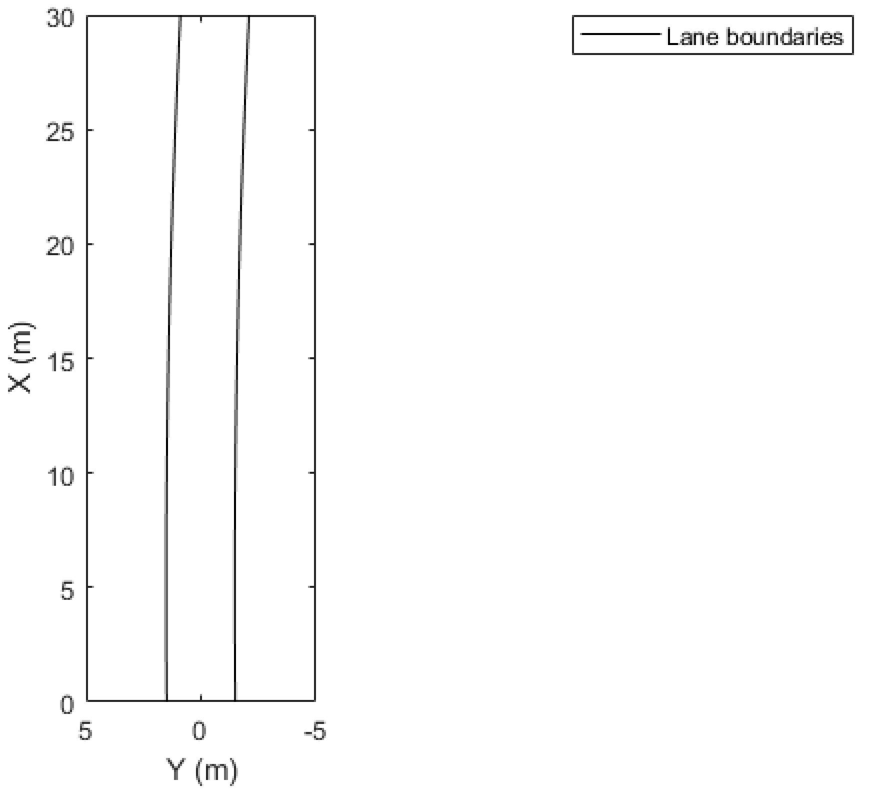
```
lb = parabolicLaneBoundary([-0.001,0.01,1.5]);
rb = parabolicLaneBoundary([-0.001,0.01,-1.5]);
```

Compute the model manually up to 30 meters ahead in the lane.

```
xWorld = (0:30)';
yLeft = computeBoundaryModel(lb,xWorld);
yRight = computeBoundaryModel(rb,xWorld);
```

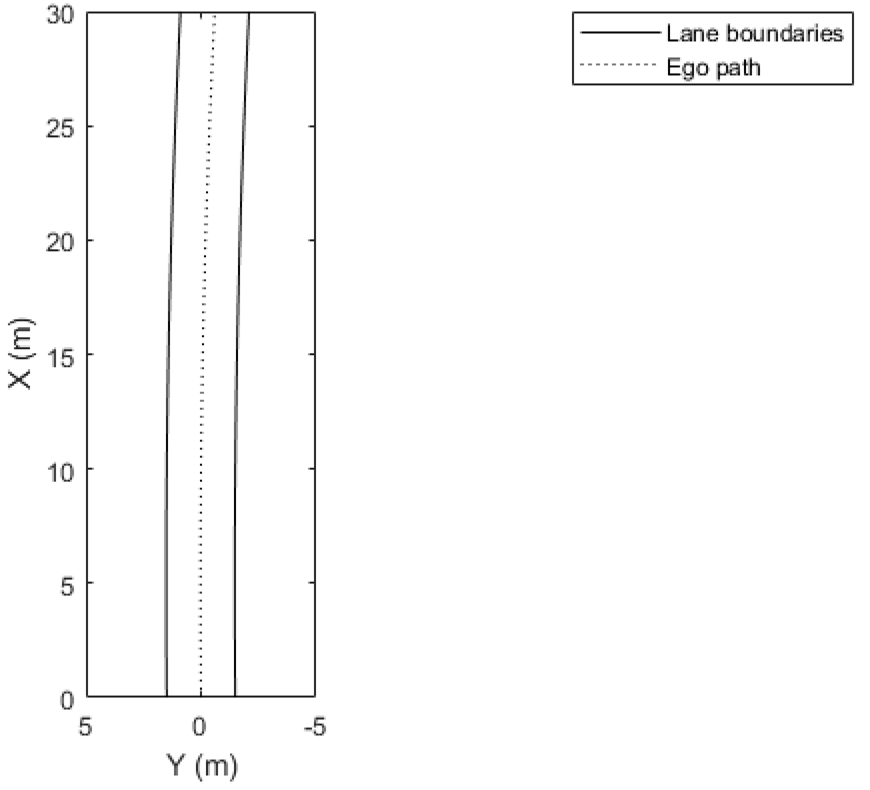
Create a bird's-eye plot and plot the lane information.

```
bep = birdsEyePlot('XLimits',[0 30],'YLimits',[-5 5]);
lanePlotter = laneBoundaryPlotter(bep,'DisplayName','Lane boundaries');
plotLaneBoundary(lanePlotter,{[xWorld,yLeft],[xWorld,yRight]});
```



Plot the path of an ego vehicle that travels through the center of the lane.

```
yCenter = (yLeft + yRight)/2;
egoPathPlotter = pathPlotter(bep,'DisplayName','Ego path');
plotPath(egoPathPlotter,{[xWorld,yCenter]});
```



Input Arguments

bep — Unpopulated bird's-eye plot
`birdsEyePlot` handle

Unpopulated bird's-eye plot, specified as a `birdsEyePlot` handle that you can update with various plotters.

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'`,`'black'`.

`DisplayName` — Plot name to display in legend
character vector | string scalar

Plot name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and a character vector or string scalar. If you do not specify a name, no entry is displayed.

`Color` — Boundary color
`'black'` (default) | character vector | string scalar | [RGB] vector

Boundary color, specified as the comma-separated pair consisting of `'FaceColor'` and a character vector, string scalar, or an [RGB] vector.

`LineStyle` — Boundary line style
`':'` (default) | `'-'` | `'--'` | `'-.'`

Boundary line style, specified as the comma-separated pair consisting of `'LineStyle'` and one of these styles.

Marker Symbol	Type
`'-'`	Solid line
`'--'`	Dashed line
`':'`	Dotted line (default)
`'-.'`	Dashed-dotted line

`Tag` — Tag to identify plot of coverage area
`'Plotter`*`N`*`'` (default) | character vector | string scalar

Tag used to identify the plot of the coverage area, specified as the comma-separated pair consisting of `'Tag'` and a character vector or string scalar. The default `'Tag'` used is,`'Plotter`*`N`*`'`, where *N* is an integer.

Output Arguments

`pPlotter` — Path plotter
plotter object

Path plotter to use for bird's-eye plot, returned as a plotter object.

See Also

Functions
`birdsEyePlot`

Introduced in R2017a

trackPlotter

Create bird’s-eye-view track plotter

Syntax

```
tPlotter = trackPlotter(bep)
tPlotter = trackPlotter(bep,Name,Value)
```

Description

`tPlotter = trackPlotter(bep)` returns a track plotter for displaying tracks in a bird’s-eye plot.

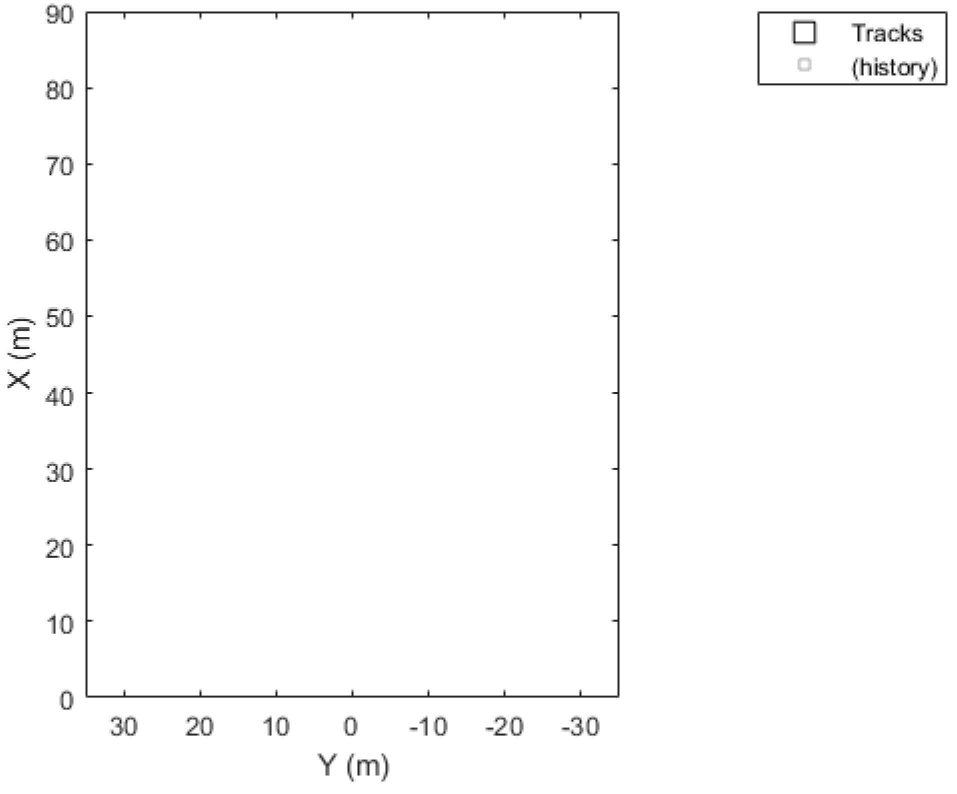
`tPlotter = trackPlotter(bep,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Create Bird's-Eye Plot with Labeled Tracks

Create a bird's-eye plot and a track plotter. Set the plotter to display up to seven history values for each track.

```
bep = birdsEyePlot('XLim',[0 90],'YLim',[-35 35]);
tPlotter = trackPlotter(bep,'DisplayName','Tracks','HistoryDepth',7);
```

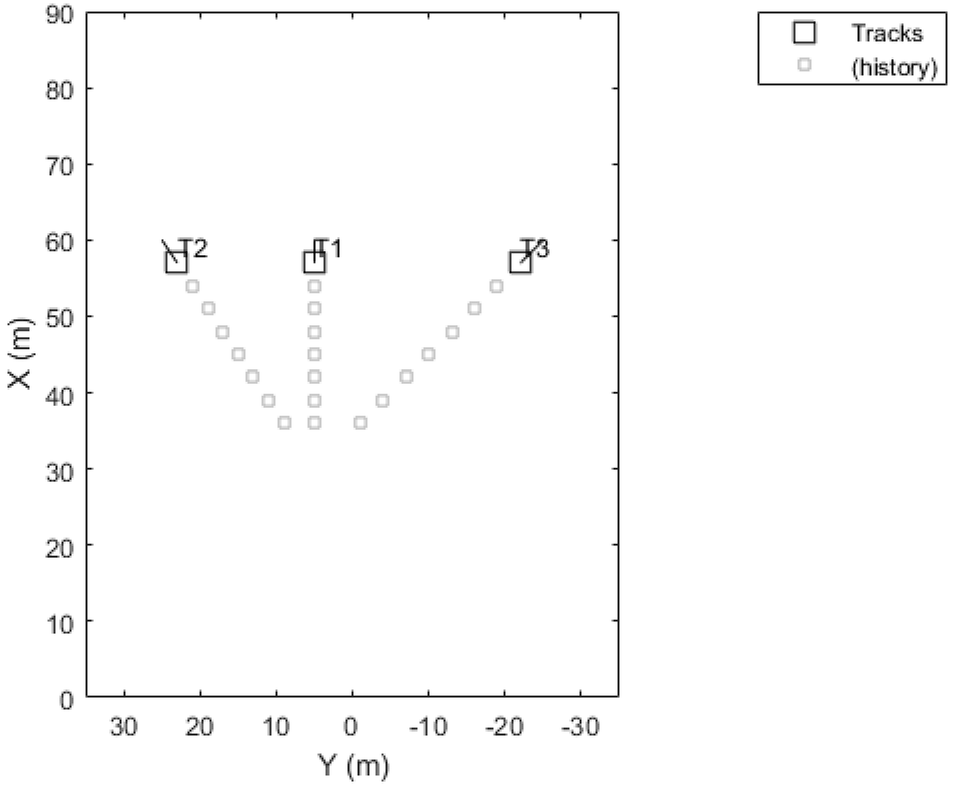


Set the positions, velocities, and labels of each track.

```
positions = [30, 5; 30, 5; 30, 5];
velocities = [3, 0; 3, 2; 3, -3];
labels = {'T1','T2','T3'};
```

Update the tracks for 10 trials, showing the seven history values specified previously.

```
for i=1:10
   plotTrack(tPlotter,positions,velocities,labels);
   positions = positions + velocities;
end
```



Input Arguments

bep — Unpopulated bird's-eye plot
`birdsEyePlot` handle

Unpopulated bird's-eye plot, specified as a `birdsEyePlot` handle that you can update with various plotters.

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','s'`.

`DisplayName` — Plot name to display in legend

character vector | string scalar

Plot name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and a character vector or string scalar. If you do not specify a name, no entry is displayed.

`HistoryDepth` — Number of previous track updates to display

`0` | value in the range [0,100]

Number of previous track updates to display, specified as the comma-separated pair consisting of '`HistoryDepth`' and a value in the range [0,100]. When you set this value to `0`, no previous updates are displayed.

`Marker` — Marker symbol

`'o'` (default) | character

Marker symbol, specified as the comma-separated pair consisting of `'Marker'` and one of these symbols.

Value	Description
`'.'`	Point
`'x'`	Cross
`'+'`	Plus sign
`'*'`	Asterisk
`'o'`	Circle (default)
`'s'`	Square
`'d'`	Diamond
`'h'`	Six-pointed star (hexagram)

Value	Description
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'<'	Left-pointing triangle
'>'	Right-pointing triangle

MarkerSize — Size of marker
positive integer

Size of marker, specified as the comma-separated pair consisting of 'MarkerSize' and a positive integer.

MarkerEdgeColor — Marker outline color
'black' (default) | character vector | string scalar | [RGB] vector

Marker outline color, specified as the comma-separated pair consisting of 'MarkerEdgeColor' and a character vector, string scalar, or an [RGB] vector.

MarkerFaceColor — Marker fill color
character vector | string scalar | [RGB] vector | 'none'

Marker outline color, specified as the comma-separated pair consisting of ''MarkerFaceColor' and a character vector, string scalar, [RGB] vector, or 'none'.

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 that represents font points.

LabelOffset — Gap between label and positional point
[0 0] (default) | two-element row vector

Gap between label and positional point, specified as the comma-separated pair consisting of 'LabelOffset' and a two-element row vector. You must specify the [*x* *y*] offset in meters.

VelocityScaling — Scale factor for magnitude length of velocity vectors
1 (default) | positive scalar

Scale factor for magnitude length of velocity vectors, specified as the comma-separated pair consisting of 'VelocityScaling' and a positive scalar. The plot renders the magnitude vector value as (*magnitude of velocity*) × VelocityScaling.

Tag — Tag to identify plot of coverage area
'Plotter*N*' (default) | character vector | string scalar

Tag to identify plot of coverage area, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. The default 'Tag' used is,'Plotter*N*', where *N* is an integer.

Output Arguments

tPlotter — Track plotter
plotter object

Track plotter to use for bird's-eye plot, returned as a plotter object.

See Also

Functions
birdsEyePlot

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.

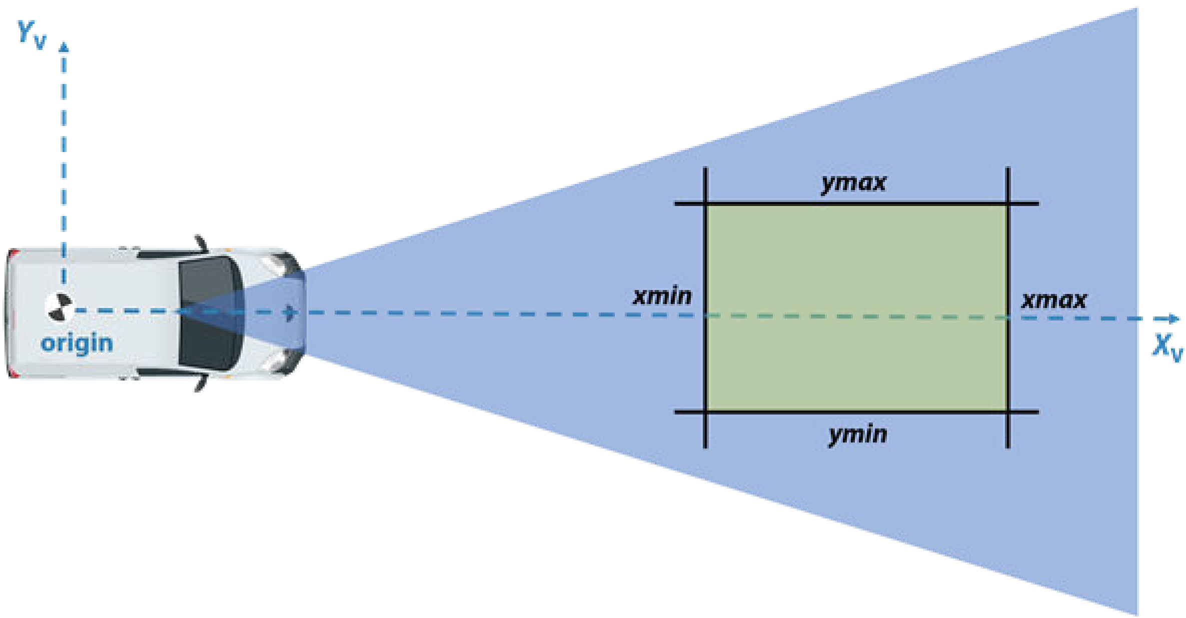
Properties

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 4-146.

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.

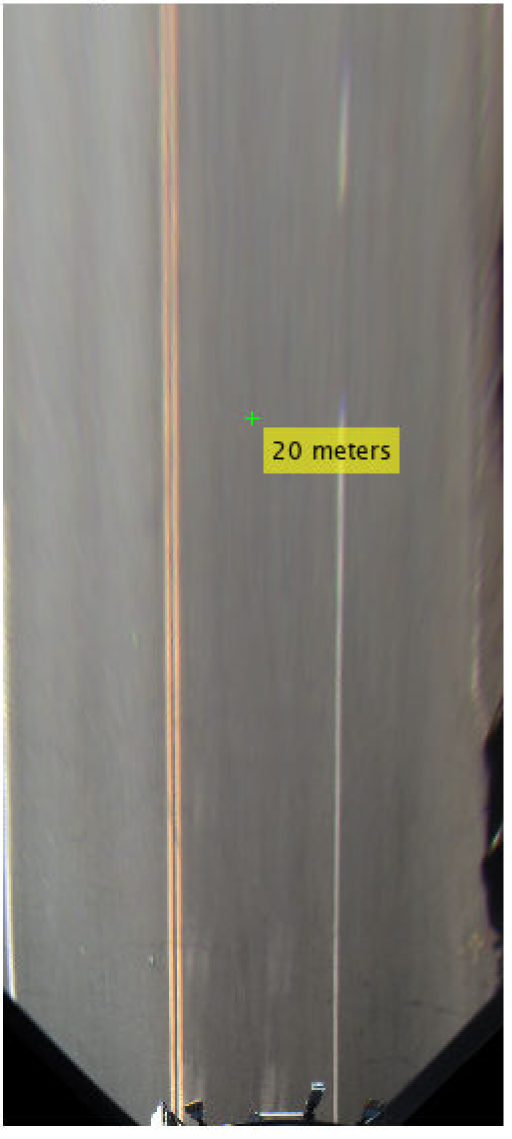
```
BEV = transformImage(birdsEye,I);
```

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')
```

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')
```


Bird's-Eye-View Image: imageToVehicle
10.77 meters

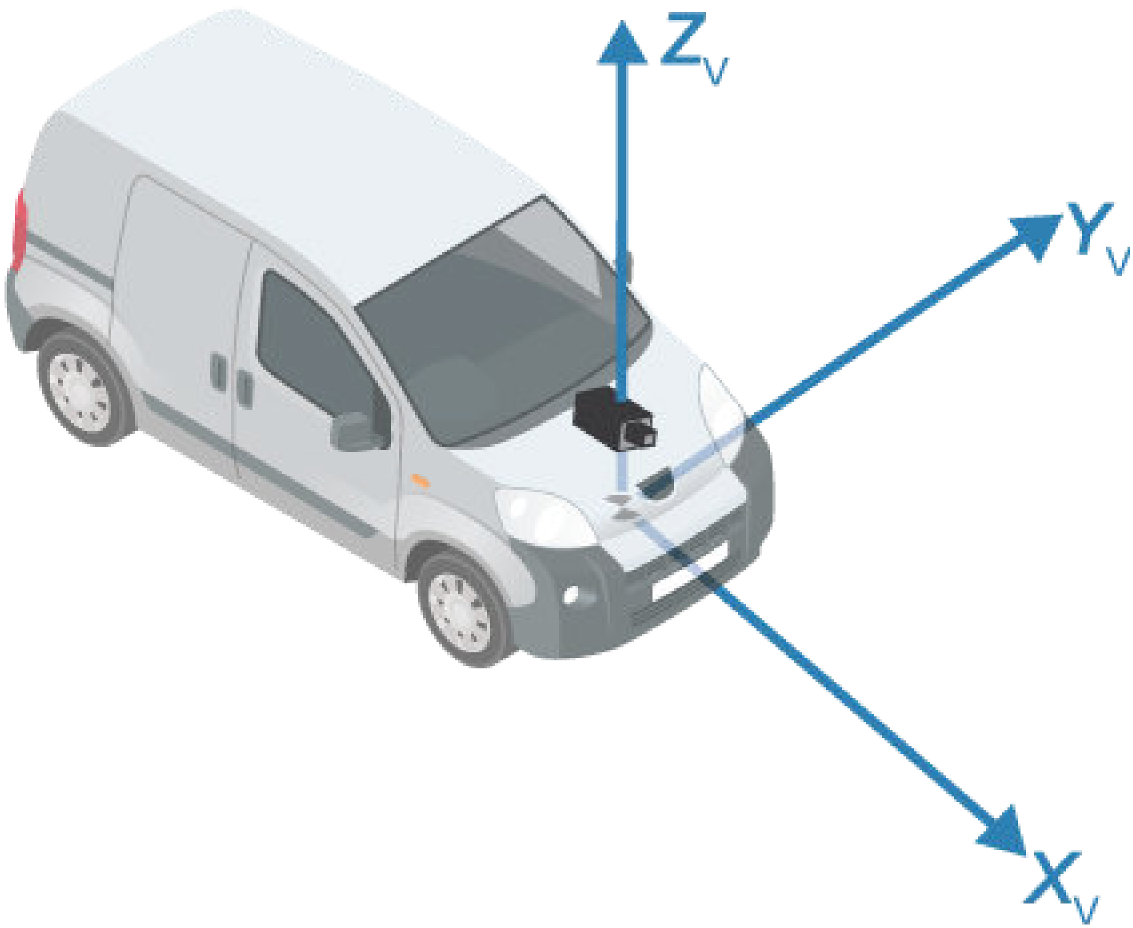
Definitions

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 System Toolbox”.

See Also

Functions
`monoCamera`

Topics

“Coordinate Systems in Automated Driving System Toolbox”

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')
```

Original Image

Transform the input image into a bird's-eye-view image.

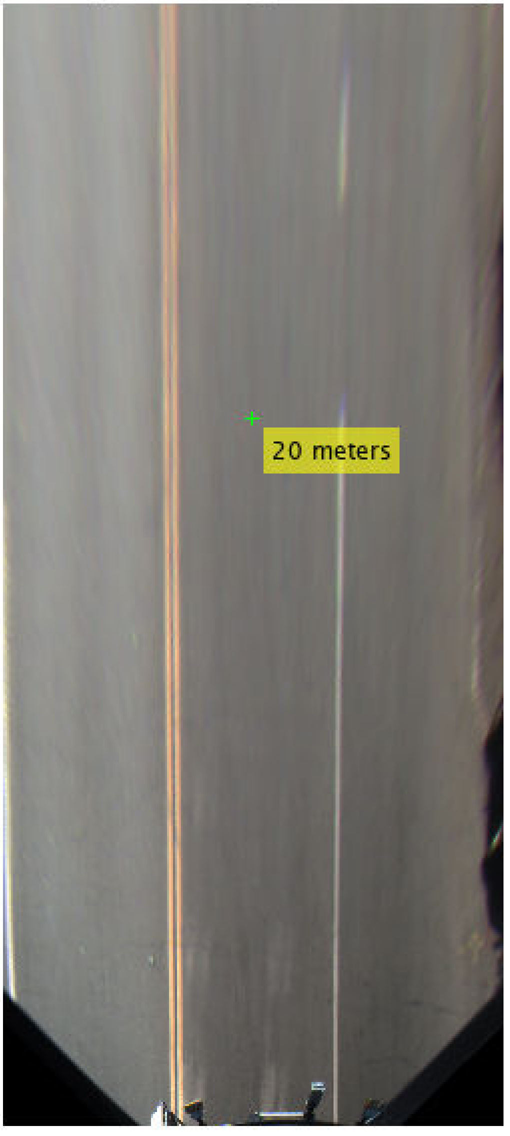
```
BEV = transformImage(birdsEye,I);
```

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')
```

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')
```


Bird's-Eye-View Image: imageToVehicle
10.77 meters

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.

See Also

Objects
`birdsEyeView`

Functions
`imageToVehicle`

Topics
"Coordinate Systems in Automated Driving System 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')
```

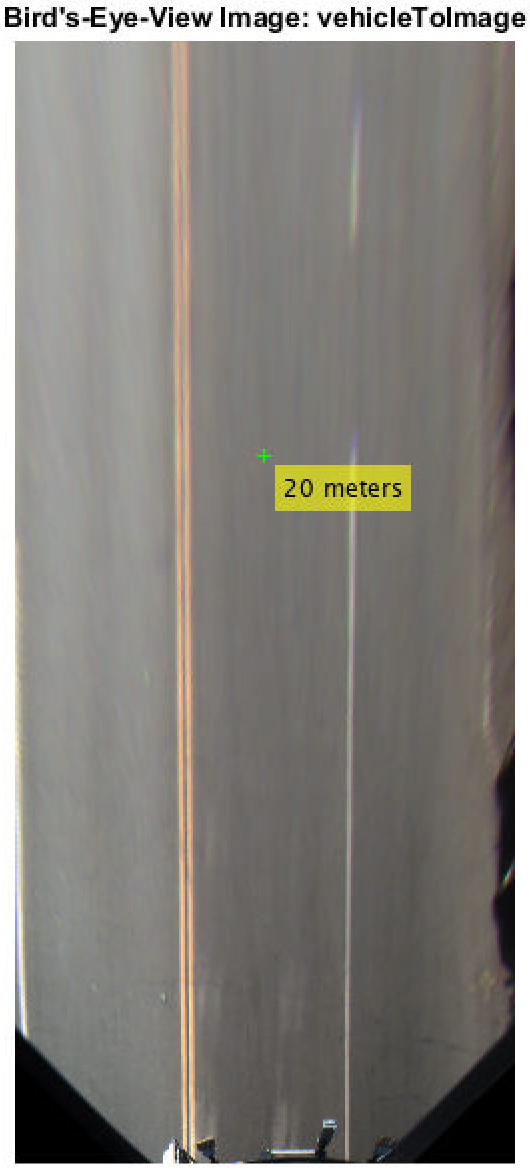
Transform the input image into a bird's-eye-view image.

```
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')
```


Bird's-Eye-View Image: vehicleToImage
20 meters

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')
```


Bird's-Eye-View Image: imageToVehicle
10.77 meters

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.

See Also

Objects
`birdsEyeView`

Functions
`vehicleToImage`

Topics
"Coordinate Systems in Automated Driving System 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

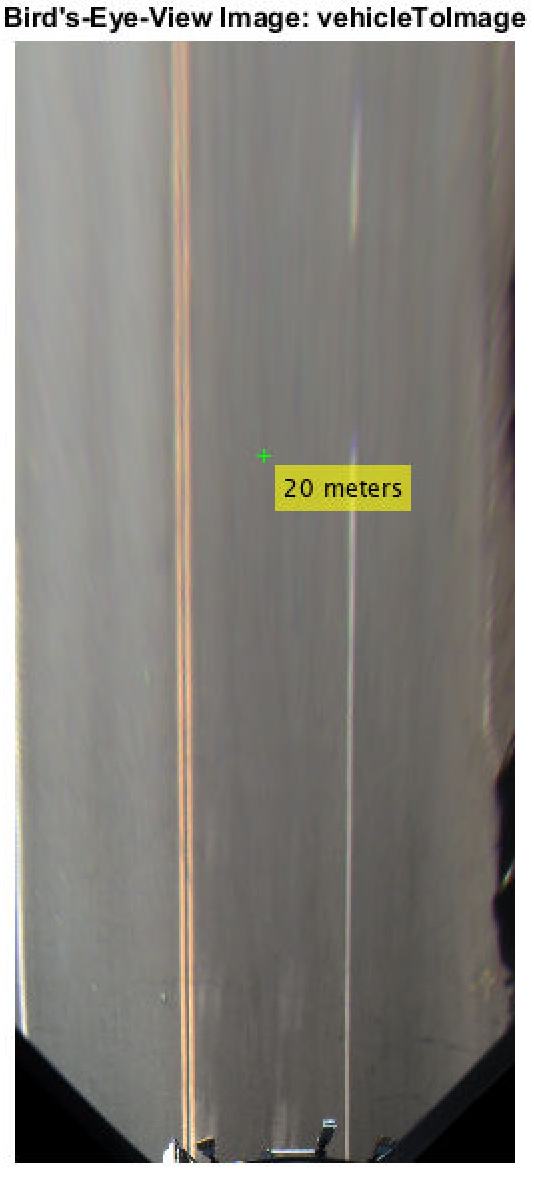
Transform the input image into a bird's-eye-view image.

```
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
```

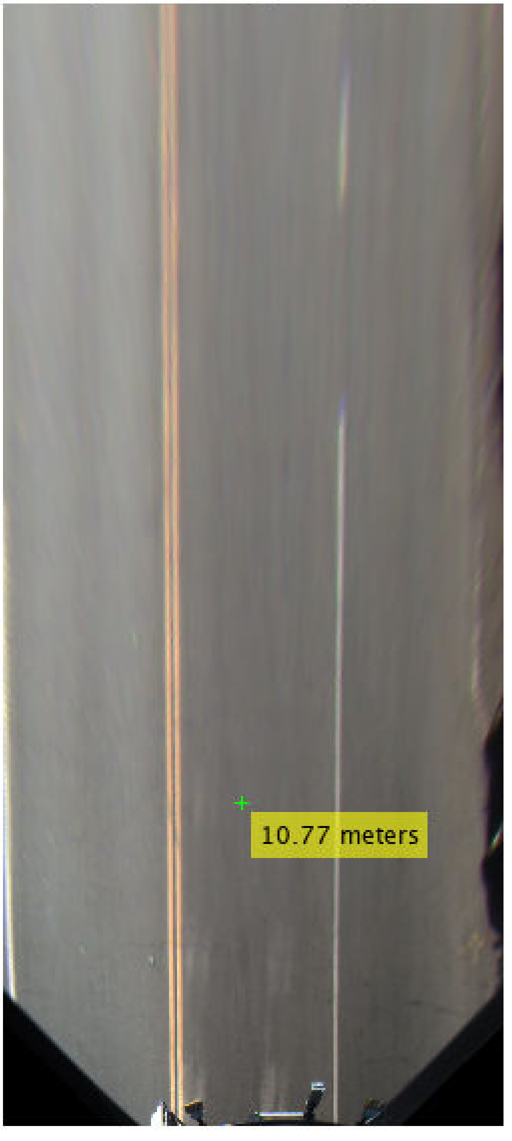
```
imshow(annotatedBEV)
title('Bird''s-Eye-View Image: vehicleToImage')
```


Bird's-Eye-View Image: vehicleToImage
20 meters

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')
```

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.

See Also

Objects
birdsEyeView

Functions
imageToVehicle | vehicleToImage

Introduced in R2017a

trackingKF class

Linear Kalman filter

Description

The trackingKF class creates a discrete-time linear Kalman filter used for tracking positions and velocities of objects which can be encountered in an automated driving scenario, such as 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. Both the process and the measurements can have additive noise. The filter also allows for optional controls or forces to act on the vehicle. 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 two ways:

- The first way is to specify explicitly the motion model. Set the motion model property, MotionModel, to Custom and then use the StateTransitionModel property to set the state transition matrix.
- The second way is to 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'

Construction

`filter = trackingKF` returns 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 `MotionModel` property is set to `'2D Constant Velocity'`.

`filter = trackingKF(F,H)` specifies the state transition model, `F`, and the measurement model, `H`. The `MotionModel` property is set to `'Custom'`.

`filter = trackingKF(F,H,G)` also specifies the control model, `G`. The `MotionModel` property is set 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 using one or more `Name`,`Value` pair arguments. 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.

MotionModel	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]`

MotionModel	Form of State Vector	Form of State Transition Model
'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.

Example: `[1.0 0.05; 0.05 2]`

Data Types: `double`

`MeasurementModel` — Measurements model from state vector
`[1 0 0 0; 0 0 1 0]` (default) | real-valued *N*-by-*M* matrix

Measurement model, 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`

Methods

clone	Create Linear Kalman filter object with identical property values
correct	Correct Kalman state vector and state covariance matrix
distance	Distance from measurements to predicted measurement
predict	Predict linear Kalman filter state
initialize	Initialize Kalman filter
likelihood	Measurement likelihood
residual	Measurement residual and residual covariance

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.

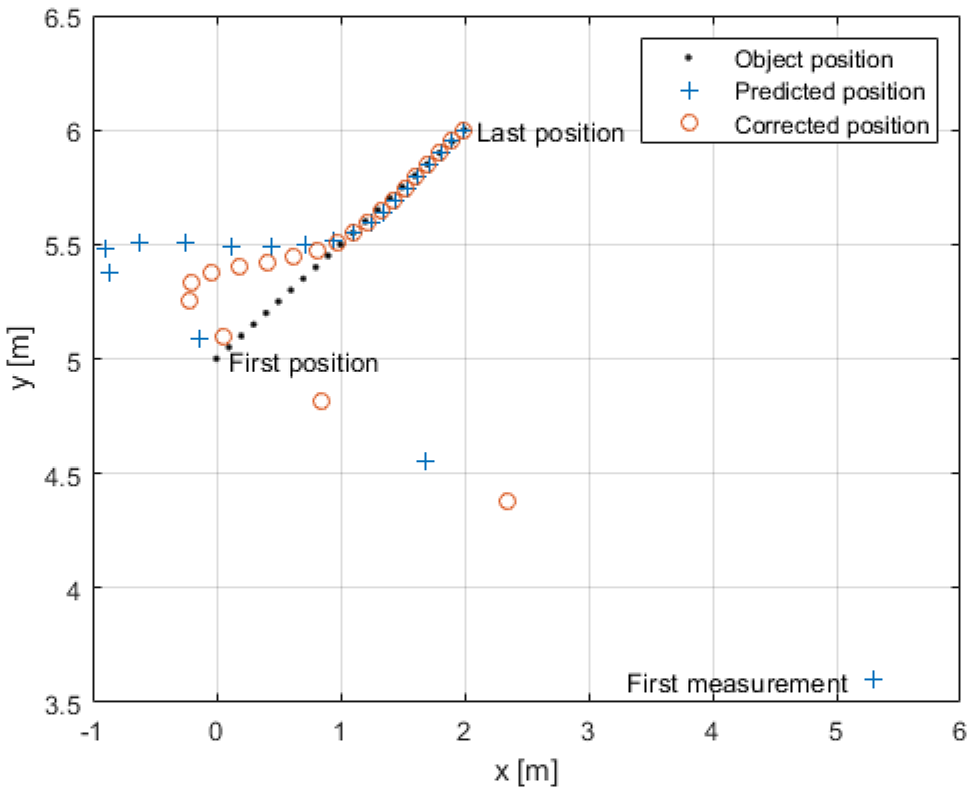
```
vx = 0.2;
vy = 0.1;
T  = 0.5;
pos = [0:vx*T:2;5:vy*T:6]';
```

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')
```



Definitions

Filter Parameters

This table relates the filter model parameters to the object properties. *M* is the size of the state vector and *N* is the size of the measurement vector. *L* is the size of the control model.

Model Parameter	Meaning	Specified in 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*
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*-

Model Parameter	Meaning	Specified in Property	Size
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.

You can read a brief description of the linear Kalman filter algorithm in “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 a value other than `Custom` for the `MotionModel` value, 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 but does 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`

Classes
`trackingEKF` | `trackingUKF`

System Objects
`multiObjectTracker`

Topics

"Linear Kalman Filters"

Introduced in R2017a

clone

Class: trackingKF

Create Linear Kalman filter object with identical property values

Syntax

`filter2 = clone(filter)`

Description

`filter2 = clone(filter)` creates another instance of the object, `filter`, having identical property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

Input Arguments

`filter` — Linear Kalman filter
`trackingKF` object

Linear Kalman filter, specified as a `trackingKF` object.

Example: `filter = trackingKF`

Output Arguments

`filter2` — Linear Kalman filter
`trackingKF` object

Linear Kalman filter, returned as a `trackingKF` object.

Introduced in R2017a

correct

Class: trackingKF

Correct Kalman state vector and state covariance matrix

Syntax

```
[xcorr,Pcorr] = correct(filter,z)
[xcorr,Pcorr] = correct(filter,z,zcov)
```

Description

`[xcorr,Pcorr] = correct(filter,z)` returns the corrected state vector, `xcorr`, and the corrected state error covariance matrix, `Pcorr`, of the tracking filter, `filter`, based on the current measurement, `z`. The internal state and covariance of the Kalman filter are overwritten by the corrected values.

`[xcorr,Pcorr] = correct(filter,z,zcov)` also specifies the measurement error covariance matrix, `zcov`. When specified, `zcov` is used as the measurement noise. Otherwise, measurement noise will have the value of the `MeasurementNoise` property.

The corrected state and covariance replaces the internal values of the Kalman filter.

Input Arguments

`filter` — Kalman filter

`trackingKF` object

Kalman filter, specified as a `trackingKF` object.

Example: `filter = trackingKF`

`z` — Object measurement

real-valued *N*-element vector

Object measurement, specified as a real-valued *N*-element vector.

Example: `[2;1]`

Data Types: `double`

`zcov` — Error covariance matrix of measurements
positive-definite real-valued *N*-by-*N* matrix

Error covariance matrix of measurements, specified as a positive-definite real-valued *N*-by-*N* matrix.

Example: `[2,1;1,20]`

Data Types: `double`

Output Arguments

`xcorr` — Corrected state
real-valued *M*-element vector

Corrected state, returned as a real-valued *M*-element vector. The corrected state represents the *a posteriori* estimate of the state vector, taking into account the current measurement.

`Pcorr` — Corrected state error covariance matrix
positive-definite real-valued *M*-by-*M* matrix

Corrected state error covariance matrix, returned as a positive-definite real-valued *M*-by-*M* matrix. The corrected covariance matrix represents the *a posteriori* estimate of the state error covariance matrix, taking into account the current measurement.

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.

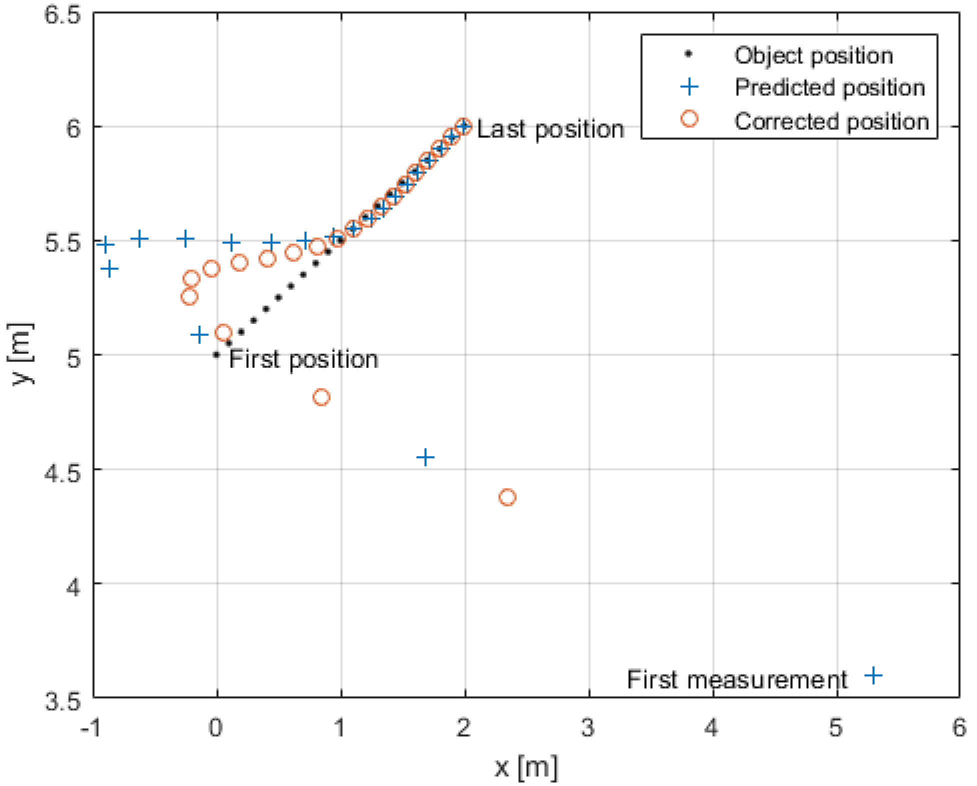
```
vx = 0.2;
vy = 0.1;
T  = 0.5;
pos = [0:vx*T:2;5:vy*T:6]';
```

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')
```



Introduced in R2017a

distance

Class: trackingKF

Distance from measurements to predicted measurement

Syntax

```
dist = distance(filter,zmat)
```

Description

`dist = distance(filter,zmat)` computes the Mahalanobis distances, `dist`, between multiple candidate measurements, `zmat`, of an object and the measurement predicted from the state of the tracking filter, `filter`. The `distance` method is useful for associating measurements to tracks.

The distance computation uses the covariance of the predicted state and the covariance of the process noise. You can call the `distance` method only after calling the `predict` method.

Input Arguments

`filter` — Linear Kalman filter
`trackingKF` object

Linear Kalman filter, specified as a `trackingKF` object.

Example: `filter = trackingKF`

`zmat` — Object measurements
real-valued *K*-by-*N* matrix

Object measurements, specified as a real-valued *K*-by-*N* matrix. *N* is the number of rows in the `MeasurementModel` property. *K* is the number of candidate measurement vectors. Each row forms a single measurement vector.

Example: `[2,1;3,0]`

Data Types: `double`

Output Arguments

`dist` — Mahalanobis distances
positive real-valued *K*-element vector

Mahalanobis distances between candidate measurements and a predicted measurement, returned as a real-valued *K*-element vector. *K* is the number of candidate measurement vectors. The method computes one distance value for each measurement vector.

Introduced in R2017a

predict

Class: trackingKF

Predict linear Kalman filter state

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)
```

Description

`[xpred,Ppred] = predict(filter)` returns the predicted state vector and the predicted state error covariance matrix for the next time step based on the current time step. The predicted values overwrite the internal state vector and covariance matrix of the filter.

This syntax applies when you set the `ControlModel` to an empty matrix.

`[xpred,Ppred] = predict(filter,u)` also specifies a control input or force, `u`.

This syntax applies when you set the `ControlModel` to a non-empty matrix.

`[xpred,Ppred] = predict(filter,F)` also 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` to an empty matrix.

`[xpred,Ppred] = predict(filter,F,Q)` also specifies the process noise covariance, `Q`. Use this syntax to change the state transition model and the process noise covariance during a simulation.

This syntax applies when you set the `ControlModel` to an empty matrix.

`[xpred,Ppred] = predict(filter,u,F,G)` also specifies 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` to a non-empty 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` to a non-empty matrix.

`[xpred,Ppred] = predict(filter,dt)` returns the predicted state and state estimation error covariance after the time step, `dt`.

This syntax applies when the `MotionModel` property is not set to `'Custom'` and the `ControlModel` property is set to an empty matrix.

`[xpred,Ppred] = predict(filter,u,dt)` also specifies a control input, `u`.

This syntax applies when the `MotionModel` property is not set to `'Custom'` and the `ControlModel` property is set to a non-empty matrix.

Input Arguments

`filter` — Kalman filter
`trackingKF` object

Kalman filter, specified as `trackingKF` object.

Example: `filter = trackingKF`

u — Control vector
real-valued *L*-element vector

Control vector, real-valued *L*-element vector.

Data Types: `double`

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.

Data Types: `double`

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.

Data Types: `double`

G — Control model

real-valued *M*-by-*L* matrix

Control model, specified as a real-valued *M*-by-*L* matrix, where *M* is the size of the state vector and *L* is the number of independent controls.

dt — Time step

positive scalar

Time step, specified as a positive scalar. Units are in seconds.

Data Types: `double`

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.

Data Types: `double`

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.

Data Types: `double`

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]';
```

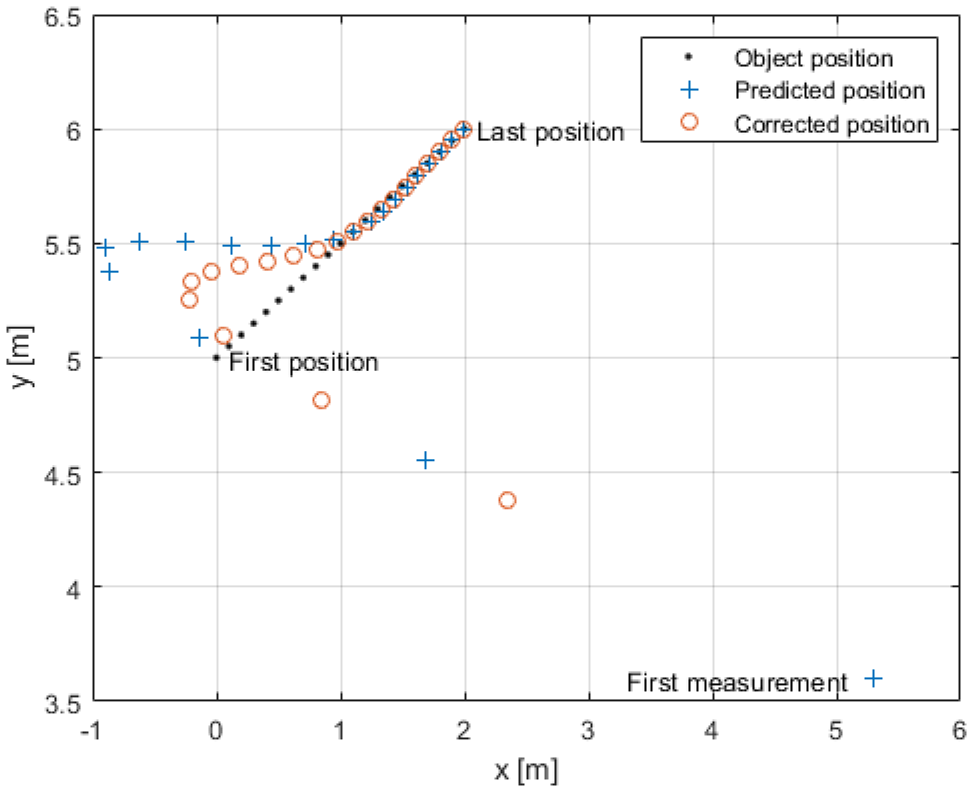
Predict and correct the state of the object.

```
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')
```



Introduced in R2017a

initialize

Class: trackingKF

Initialize Kalman filter

Syntax

```
initialize(filter,X,P)
initialize(filter,X,P,Name,Value)
```

Description

`initialize(filter,X,P)` initializes the Kalman filter, `filter`, using the state, `x`, and the state covariance, `P`.

`initialize(filter,X,P,Name,Value)` initializes the Kalman filter properties using one of more name-value pairs of the filter.

Note: you cannot change the size or type of properties you are initializing.

Input Arguments

`filter` — Kalman tracking filter
`Kalman filter object`

Kalman tracking filter, specified as a Kalman filter object.

X — Initial Kalman filter state
vector | matrix

Initial Kalman filter state, specified as a vector or matrix.

P — Initial Kalman filter state covariance
matrix

Initial Kalman filter state covariance, specified as a matrix.

Introduced in R2018a

likelihood

Class: trackingKF

Measurement likelihood

Syntax

```
measlikelihood = likelihood(filter,zmeas)
```

Description

`measlikelihood = likelihood(filter,zmeas)` returns the likelihood of the measurement, `zmeas`, of an object tracked by the Kalman filter, `filter`.

Input Arguments

`filter` — Kalman tracking filter
`Kalman filter object`

Kalman tracking filter, specified as a Kalman filter object.

`zmeas` — Measurement of tracked object
vector | matrix

Measurement of the tracked object, specified as a vector or matrix.

Output Arguments

`measlikelihood` — Likelihood of measurement
scalar

Likelihood of measurement, returned as a scalar.

See Also

Introduced in R2018a

residual

Class: trackingKF

Measurement residual and residual covariance

Syntax

`[zres,rescov] = residual(filter,zmeas)`

Description

`[zres,rescov] = residual(filter,zmeas)` computes the residual, `zres`, between a measurement, `zmeas`, and a predicted measurement derived from the state of the Kalman filter, `filter`. The function also returns the covariance of the residual, `rescov`.

Input Arguments

`filter` — Linear Kalman tracking filter

`Linear Kalman filter object`

Linear Kalman tracking filter, specified as a Kalman filter object.

`zmeas` — Measurement of tracked object

vector | matrix

Measurement of the tracked object, specified as a vector or matrix.

Output Arguments

`zres` — Residual between measurement and predicted measurement

matrix

Residual between measurement and predicted measurement, returned as a matrix.

`rescov` — Covariance of residuals
matrix

Covariance of the residuals, returned as a matrix.

Algorithms

- The residual is the difference between a measurement and the value predicted by the filter. The residual d is defined as $d = z - Hx$. H is the measurement model set by the `MeasurementModel` property, x is the current filter state, and z is the current measurement.
- The covariance of the residual, S, is defined as $S = HPH' + R$ where P is the state covariance matrix, R is the measurement noise matrix set by the `MeasurementNoise` property.

Introduced in R2018a

trackingEKF class

Extended Kalman filter

Description

The `trackingEKF` class creates a discrete-time extended Kalman filter used for tracking positions and velocities of objects which are encountered in an automated driving scenario, such as 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 that follows a nonlinear motion model, or when the measurements are nonlinear functions of the state, or both. The filter also allows for optional controls or forces to act on the object. 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 the measurements can have Gaussian noise which can be included in two ways:

- Noise can be added 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.
- Noises can be included in the state transition function, the measurement model function, or both. In these cases, the corresponding noise sizes are not restricted.

Construction

`filter = trackingEKF` creates an extended Kalman filter object for a discrete-time system 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 using one or more `Name,Value` pair arguments. 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.

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.

- If `HasAdditiveProcessNoise` is `true`, specify the function using one of these syntaxes:

  ```
  x(k) = transitionfcn(x(k-1))
  ```

  ```
  x(k) = transitionfcn(x(k-1),parameters)
  ```

  where `x(k)` is the state at time `k`. The `parameters` term stands for all additional arguments required by the state transition function.
- If `HasAdditiveProcessNoise` is `false`, specify the function using one of these syntaxes:

```
x(k) = transitionfcn(x(k-1),w(k-1))
```

```
x(k) = transitionfcn(x(k-1),w(k-1),parameters)
```

where `x(k)` is the state at time `k` and `w(k)` is a value for the process noise at time `k`. The `parameters` argument stands for all additional arguments required by the state transition function.

Example: `@constacc`

Data Types: `function_handle`

`StateTransitionJacobianFcn` — State transition function Jacobian
function handle

The Jacobian of the state transition function, specified as a function handle. This function has the same input arguments as the state transition function.

- If `HasAdditiveProcessNoise` is `true`, specify the Jacobian function using one of these syntaxes:

  ```
  Jx(k) = statejacobianfcn(x(k))
  ```

  ```
  Jx(k) = statejacobianfcn(x(k),parameters)
  ```

  where `x(k)` is the state at time `k`. The `parameters` argument stands for all additional arguments required by the state transition function.

  `Jx(k)` denotes the Jacobian of the predicted state with respect to the previous state. The 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.
- If `HasAdditiveProcessNoise` is `false`, specify the Jacobian function using one of these syntaxes:

  ```
  [Jx(k),Jw(k)] = statejacobianfcn(x(k),w(k))
  ```

  ```
  [Jx(k),Jw(k)] = statejacobianfcn(x(k),w(k),parameters)
  ```

  where `x(k)` is the state at time `k` and `w(k)` is a sample *Q*-element vector of the process noise at time `k`. *Q* is the size of the process noise covariance. Unlike the case of additive process noise, the process noise vector in the non-additive noise case need not 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.

If not specified, the Jacobians are computed by numerical differencing at each call of the `predict` method. This computation can increase the processing time and numerical inaccuracy.

Example: `@constaccjac`

Data Types: `function_handle`

`ProcessNoise` — Process noise covariance
1 (default) | positive real-valued scalar | positive-definite real-valued matrix

Process noise covariance:

- When `HasAdditiveProcessNoise` is `true`, specify the process noise covariance as a 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 an *Q*-by-*Q* matrix. *Q* is the size of the process noise vector.

  You must specify `ProcessNoise` before any call to the `predict` method. 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 processes 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)
  ```

  where `x(k)` is the state at time `k` and `z(k)` is the predicted measurement at time `k`. The `parameters` term 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)
  ```

  where `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)
  ```

where `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)
  ```

  where `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` method. 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` method. 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.

Methods

clone	Create extended Kalman filter object with identical property values
correct	Correct Kalman state vector and state error covariance matrix
distance	Distance from measurements to predicted measurement
predict	Predict extended Kalman state vector and state error covariance matrix
initialize	Initialize extended Kalman filter
likelihood	Measurement likelihood
residual	Measurement residual and residual covariance

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` methods 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
```

Definitions

Filter Parameters

This table relates the filter model parameters to the object properties. In this table, *M* is the size of the state vector and *N* is the size of the measurement vector.

Filter Parameter	Meaning	Specified in 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

Filter Parameter	Meaning	Specified in Property	Size
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`.
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`

Classes
`trackingKF` | `trackingUKF`

System Objects
`multiObjectTracker`

Topics
"Extended Kalman Filters"

Introduced in R2017a

clone

Class: trackingEKF

Create extended Kalman filter object with identical property values

Syntax

`filter2 = clone(filter)`

Description

`filter2 = clone(filter)` creates another instance of the object, `trackingEKF`, having identical property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

Input Arguments

`filter` — Extended Kalman filter
`trackingEKF` object

Extended Kalman filter, specified as a `trackingEKF` object.

Example: `filter = trackingEKF`

Output Arguments

`filter2` — Extended Kalman filter
`trackingEKF` object

Extended Kalman filter, returned as a `trackingEKF` object.

Introduced in R2017a

correct

Class: trackingEKF

Correct Kalman state vector and state error covariance matrix

Syntax

```
[xcorr,Pcorr] = correct(filter,z)
[xcorr,Pcorr] = correct(filter,z,varargin)
```

Description

`[xcorr,Pcorr] = correct(filter,z)` returns the corrected state vector, `xcorr`, and the corrected state error covariance matrix, `Pcorr`, for the extended Kalman filter defined in `filter`, based on the current measurement, `z`. The internal state and covariance of the Kalman filter are overwritten by the corrected values.

`[xcorr,Pcorr] = correct(filter,z,varargin)` also specifies any input arguments to the measurement function. These arguments are used as input to the measurement function specified in the `MeasurementFcn` property.

Input Arguments

`filter` — Extended Kalman filter

`trackingEKF` object

Extended Kalman filter, specified as a `trackingEKF` object.

Example: `filter = trackingEKF`

`z` — Object measurement

real-valued *N*-element vector

Object measurement, specified as a real-valued *N*-element vector.

Example: `[2;1]`

`varargin` — Measurement function arguments
comma-separated list

Measurement function arguments, specified as a comma-separated list. These arguments are the same ones that are passed into the measurement function specified by the `MeasurementFcn` property. For example, if you set `MeasurementFcn` to `@cameas`, and then call

```
[xcorr,Pcorr] = correct(filter,frame,sensorpos,sensorvel)
```

the `correct` method will internally call

```
meas = cameas(state,frame,sensorpos,sensorvel)
```

.

Output Arguments

`xcorr` — Corrected state
real-valued *M*-element vector

Corrected state, returned as a real-valued *M*-element vector. The corrected state represents the *a posteriori* estimate of the state vector, taking into account the current measurement.

`Pcorr` — Corrected state error covariance matrix
positive-definite real-valued *M*-by-*M* matrix

Corrected state error covariance matrix, returned as a positive-definite real-valued *M*-by-*M* matrix. The corrected state covariance matrix represents the *a posteriori* estimate of the state covariance matrix, taking into account the current measurement.

Introduced in R2017a

distance

Class: trackingEKF

Distance from measurements to predicted measurement

Syntax

```
dist = distance(filter,zmat)
dist = distance(filter,zmat,measurementParams)
```

Description

`dist = distance(filter,zmat)` computes the Mahalanobis distances between multiple candidate measurements of an object, `zmat`, and the predicted measurement computed by the `trackingEKF` object. The distance method is used to assign measurements to tracks.

This distance computation takes into account the covariance of the predicted state and the covariance of the process noise. You can call the `distance` method only after calling the `predict` method.

`dist = distance(filter,zmat,measurementParams)` also specifies the parameters used by the measurement function set in the `MeasurementFcn` property.

Input Arguments

`filter` — Extended Kalman filter
`trackingEKF` object

Extended Kalman filter, specified as a `trackingEKF` object.

Example: `filter = trackingEKF`

`zmat` — Object measurements
real-valued *K*-by-*N* matrix

Measurements, specified as a real-valued *K*-by-*N* matrix. *K* is the number of candidate measurement vectors. Each row corresponds to a candidate measurement vector.*N* is the number of rows in the output of the function specified by the `MeasurementFcn` property.

Example: `[2,1;3,0]`

Data Types: `double`

`measurementParams` — Measurement function parameters
`{}` (default) | cell array

Measurement function parameters, specified as a cell array containing arguments to the measurement function specified by the `MeasurementFcn` property. Suppose you set `MeasurementFcn` to `@cameas`, and then set these values:

```
measurementParams = {frame,sensorpos,sensorpos)
```

The `distance` method internally calls the following:

```
cameas(state,frame,sensorpos,sensorvel)
```

Data Types: `cell`

Output Arguments

`dist` — Mahalanobis distances
real-valued *K*-element vector of positive values

Mahalanobis distances between candidate measurements and the predicted measurement, returned as a real-valued *K*-element vector of positive values. There is one distance value per measurement vector.

Data Types: `double`

Introduced in R2017a

predict

Class: trackingEKF

Predict extended Kalman state vector and state error covariance matrix

Syntax

```
[xpred,Ppred] = predict(filter)
[xpred,Ppred] = predict(filter,varargin)
[xpred,Ppred] = predict( ___ ,dt)
```

Description

`[xpred,Ppred] = predict(filter)` returns the predicted state vector, `xpred`, and state error covariance matrix, `Ppred`, at the next time step based on the current time step. The predicted values overwrite the internal state vector and state error covariance matrix of the extended Kalman filter.

`[xpred,Ppred] = predict(filter,varargin)` specifies input arguments, `varargin`, for the state transition function set in the `StateTransitionFcn` property.

`[xpred,Ppred] = predict( ___ ,dt)` also specifies the time step, `dt`.

Input Arguments

`filter` — Extended Kalman filter
`trackingEKF` object

Extended Kalman filter, specified as a `trackingEKF` object.

Example: `filter = trackingEKF`

`varargin` — State transition function arguments
comma-separated list

State transition function arguments, specified as a comma-separated list. These arguments are the same ones that are passed into the state transition function specified by the `StateTransitionFcn` property. For example, if you set the `StateTransitionFcn` property to `@constacc`, and then call

```
[xpred,Ppred] = predict(filter,dt)
```

the `predict` method will internally call

```
state = constacc(state,dt)
```

dt — Time step
positive scalar

Time step, specified as a positive scalar. Units are in seconds.

Data Types: `double`

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 *a priori* estimate of the state vector propagated from the previous state. The prediction uses the state transition function specified in the `StateTransitionFcn` property.

Data Types: `double`

Ppred — Predicted state error covariance matrix
real-valued *M*-by-*M* matrix

Predicted state error covariance matrix, returned as a real-valued *M*-by-*M* matrix. This predicted error is the *a priori* estimate of the state error covariance matrix. `predict` uses the state transition function Jacobian specified in the `StateTransitionJacobianFcn` property.

Data Types: `double`

Introduced in R2017a

initialize

Class: trackingEKF

Initialize extended Kalman filter

Syntax

```
initialize(filterobj,X,P)
initialize(filterobj,X,P,Name,Value)
```

Description

`initialize(filterobj,X,P)` initializes the extended Kalman filter, `filterobj`, using the state, `x`, and the state covariance, `P`.

`initialize(filterobj,X,P,Name,Value)` initializes Kalman filter properties using name-value pairs.

Note: you cannot change the size or type of properties you are initializing.

Input Arguments

`filterobj` — Extended Kalman tracking filter
`Extended Kalman filter object`

Kalman tracking filter, specified as a Kalman filter object.

X — Initial extended Kalman filter state
vector | matrix

Initial extended Kalman filter state, specified as a vector or matrix.

P — Initial extended Kalman filter state covariance
matrix

Initial extended Kalman filter state covariance, specified as a matrix.

Introduced in R2018a

likelihood

Class: trackingEKF

Measurement likelihood

Syntax

```
measlikelihood = likelihood(filterobj,zmeas)
measlikelihood = likelihood(filterobj,zmeas,measparams)
```

Description

`measlikelihood = likelihood(filterobj,zmeas)` returns the likelihood of the measurement, `zmeas`, of an object tracked by the extended Kalman filter, `filterobj`.

`measlikelihood = likelihood(filterobj,zmeas,measparams)` also specifies measurement parameters, `measparams`.

Input Arguments

`filterobj` — Extended Kalman tracking filter
`Kalman filter object`

Extended Kalman tracking filter, specified as an extended Kalman filter object.

`zmeas` — Measurement of tracked object
vector | matrix

Measurement of the 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 defined in the `MeasurementFcn` property of the Extended Kalman filter, `filterobj`.

Output Arguments

`measlikelihood` — Likelihood of measurement
scalar

Likelihood of measurement, returned as a scalar.

See Also

Introduced in R2018a

residual

Class: trackingEKF

Measurement residual and residual covariance

Syntax

```
[zres,rescov] = residual(filterobj,zmeas)
[zres,rescov] = residual(filterobj,zmeas,measparams)
```

Description

`[zres,rescov] = residual(filterobj,zmeas)`computes the residual, `zres`, between a measurement, `zmeas`, and a predicted measurement produced by the Kalman filter, `filterobj`. The function also returns the covariance of the residual, `zres`.

`[zres,rescov] = residual(filterobj,zmeas,measparams)` also specifies measurement parameters, `measparams`.

Input Arguments

`filterobj` — Kalman tracking filter
`Kalman filter object`

Kalman tracking filter, specified as a Kalman filter object.

`zmeas` — Measurement of tracked object
vector | matrix

Measurement of the 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 defined in the `MeasurementFcn` property of the `filterobj`

Output Arguments

`zres` — Residual between measurement and predicted measurement
matrix

Residual between measurement and predicted measurement, returned as a matrix.

`rescov` — Covariance of residuals
matrix

Covariance of the residuals, returned as a matrix.

Algorithms

- The residual is the difference between a measurement and the value predicted by the filter. The residual d is defined as $d = z - h(x)$. h is the measurement function set by the `MeasurementFcn` property, x is the current filter state, and z is the current measurement.
- The covariance of the residual, S, is defined as $S = HPH' + R$ where P is the state covariance matrix, R is the measurement noise matrix set by the `MeasurementNoise` property.

Introduced in R2018a

trackingUKF class

Unscented Kalman filter

Description

The `trackingUKF` class creates a discrete-time unscented Kalman filter used for tracking positions and velocities of objects which may be encountered in an automated driving scenario, such as 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. In addition, the process and the measurements can have noise. Use an unscented Kalman filter when the current state is a nonlinear function of the previous state or when the measurements are nonlinear functions of the state or when both conditions apply. The unscented Kalman filter estimates the uncertainty about the state, and its propagation through the nonlinear state and measurement equations, using a fixed number of sigma points. Sigma points are chosen using the unscented transformation as parameterized by the `Alpha`, `Beta`, and `Kappa` properties.

Construction

`filter = trackingUKF` creates an unscented Kalman filter object for a discrete-time system 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. 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.

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.

- If `HasAdditiveProcessNoise` is `true`, specify the function using one of these syntaxes:

  ```
  x(k) = transitionfcn(x(k-1))
  ```

  ```
  x(k) = transitionfcn(x(k-1),parameters)
  ```

  where `x(k)` is the state at time `k`. The `parameters` term stands for all additional arguments required by the state transition function.
- If `HasAdditiveProcessNoise` is `false`, specify the function using one of these syntaxes:

  ```
  x(k) = transitionfcn(x(k-1),w(k-1))
  ```

  ```
  x(k) = transitionfcn(x(k-1),w(k-1),parameters)
  ```

where `x(k)` is the state at time `k` and `w(k)` is a value for the process noise at time `k`. The `parameters` argument 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-valued scalar | positive-definite real-valued matrix

Process noise covariance:

- When `HasAdditiveProcessNoise` is `true`, specify the process noise covariance as a 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 an *Q*-by-*Q* matrix. *Q* is the size of the process noise vector.

  You must specify `ProcessNoise` before any call to the `predict` method. 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 processes 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)
  ```

  where `x(k)` is the state at time `k` and `z(k)` is the predicted measurement at time `k`. The `parameters` term 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)
  ```

  where `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` method. 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 zero and less than or equal to one.

`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.

Methods

clone	Create unscented Kalman filter object with identical property values
correct	Correct Kalman state vector and state error covariance matrix
distance	Distance from measurements to predicted measurement
predict	Predict unscented Kalman state vector and state error covariance matrix
initialize	Initialize unscented Kalman filter
likelihood	Measurement likelihood
residual	Measurement residual and residual covariance

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` methods 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
```

Definitions

Filter parameters and dimensions

This table relates the filter model parameters to the object properties. *M* is the size of the state vector and *N* is the size of the measurement vector.

Filter Parameter	Meaning	Specified in 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*
P_k	State error covariance matrix representing the uncertainty in the values of the state	`StateCovariance`	*M*-by-*M*

Filter Parameter	Meaning	Specified in Property	Size
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`

Classes
`trackingEKF` | `trackingKF`

System Objects
`multiObjectTracker`

Introduced in R2017a

clone

Class: trackingUKF

Create unscented Kalman filter object with identical property values

Syntax

`filter2 = clone(filter)`

Description

`filter2 = clone(filter)` creates another instance of the object, `trackingUKF`, having identical property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

Input Arguments

`filter` — Unscented Kalman filter

`trackingUKF` object

Unscented Kalman filter, specified as a `trackingUKF` object.

Example: `filter = trackingEKF`

Output Arguments

`filter2` — Unscented Kalman filter

`trackingUKF` object

Unscented Kalman filter, returned as a `trackingUKF` object.

Introduced in R2017a

correct

Class: trackingUKF

Correct Kalman state vector and state error covariance matrix

Syntax

```
[xcorr,Pcorr] = correct(filter,z)
[xcorr,Pcorr] = correct(filter,z,varargin)
```

Description

`[xcorr,Pcorr] = correct(filter,z)` returns the corrected state vector, `xcorr`, and the corrected state error covariance matrix, `Pcorr`, for the unscented Kalman filter defined in `filter`, based on the current measurement, `z`. The internal state and covariance of the Kalman filter are overwritten by the corrected values.

`[xcorr,Pcorr] = correct(filter,z,varargin)` also specifies any input arguments to the measurement function. These arguments are used as input to the measurement function specified in the `MeasurementFcn` property.

Input Arguments

`filter` — Unscented Kalman filter
`trackingUKF` object

Unscented Kalman filter, specified as a `trackingUKF` object.

Example: `filter = trackingUKF`

`z` — Object measurement
real-valued *N*-element vector

Object measurement, specified as a real-valued *N*-element vector.

Example: `[2;1]`

`varargin` — Measurement function arguments
comma-separated list

Measurement function arguments, specified as a comma-separated list. These arguments are the same ones that are passed into the measurement function specified by the `MeasurementFcn` property. For example, if you set `MeasurementFcn` to `@cameas`, and then call

```
[xcorr,Pcorr] = correct(filter,frame,sensorpos,sensorvel)
```

the `correct` method will internally call

```
meas = cameas(state,frame,sensorpos,sensorvel)
```

.

Output Arguments

`xcorr` — Corrected state
real-valued *M*-element vector

Corrected state, returned as a real-valued *M*-element vector. The corrected state represents the *a posteriori* estimate of the state vector, taking into account the current measurement.

`Pcorr` — Corrected state error covariance matrix
positive-definite real-valued *M*-by-*M* matrix

Corrected state error covariance matrix, returned as a positive-definite real-valued *M*-by-*M* matrix. The corrected state covariance matrix represents the *a posteriori* estimate of the state covariance matrix, taking into account the current measurement.

Introduced in R2017a

distance

Class: trackingUKF

Distance from measurements to predicted measurement

Syntax

```
dist = distance(filter,zmat)
dist = distance(filter,zmat,measurementParams)
```

Description

`dist = distance(filter,zmat)` computes the Mahalanobis distances between multiple candidate measurements of an object, `zmat`, and the predicted measurement computed by the `trackingUKF` object. The distance method is used to assign measurements to tracks.

This distance computation takes into account the covariance of the predicted state and the covariance of the process noise. You can call the `distance` method only after calling the `predict` method.

`dist = distance(filter,zmat,measurementParams)` also specifies the parameters used by the measurement function set in the `MeasurementFcn` property.

Input Arguments

`filter` — Unscented Kalman filter
`trackingUKF`object

Unscented Kalman filter, specified as a `trackingUKF` object.

Example: `filter = trackingUKF`

`zmat` — Object measurements
real-valued *K*-by-*N* matrix

Measurements, specified as a real-valued *K*-by-*N* matrix. *K* is the number of candidate measurement vectors. Each row corresponds to a candidate measurement vector.*N* is the number of rows in the output of the function specified by the `MeasurementFcn` property.

Example: `[2,1;3,0]`

Data Types: `double`

`measurementParams` — Measurement function parameters
`{}` (default) | cell array

Measurement function parameters, specified as a cell array containing arguments to the measurement function specified by the `MeasurementFcn` property. Suppose you set `MeasurementFcn` to `@cameas`, and then set these values:

```
measurementParams = {frame,sensorpos,sensorpos)
```

The `distance` method internally calls the following:

```
cameas(state,frame,sensorpos,sensorvel)
```

Data Types: `cell`

Output Arguments

`dist` — Mahalanobis distances
real-valued *K*-element vector of positive values

Mahalanobis distances between candidate measurements and the predicted measurement, returned as a real-valued *K*-element vector of positive values. There is one distance value per measurement vector.

Data Types: `double`

Introduced in R2017a

predict

Class: trackingUKF

Predict unscented Kalman state vector and state error covariance matrix

Syntax

```
[xpred,Ppred] = predict(filter)
[xpred,Ppred] = predict(filter,varargin)
[xpred,Ppred] = predict(___,dt)
```

Description

`[xpred,Ppred] = predict(filter)` returns the predicted state vector, `xpred`, and state error covariance matrix, `Ppred`, at the next time step based on the current time step. The predicted values overwrite the internal state vector and state error covariance matrix of the unscented Kalman filter.

`[xpred,Ppred] = predict(filter,varargin)` specifies in `varargin` input arguments of the state transition function set in the `StateTransitionFcn` property.

`[xpred,Ppred] = predict(___,dt)` also specifies the time step, `dt`.

Input Arguments

`filter` — Unscented Kalman filter
`trackingUKF` object

Unscented Kalman filter, specified as a `trackingUKF` object.

Example: `filter = trackingUKF`

`varargin` — State transition function arguments
comma-separated list

State transition function arguments, specified as a comma-separated list. These arguments are the same ones that are passed into the state transition function specified by the `StateTransitionFcn` property. For example, if you set the `StateTransitionFcn` property to `@constacc`, and then call

```
[xpred,Ppred] = predict(filter,dt)
```

the `predict` method will internally call

```
state = constacc(state,dt)
```

dt — Time step
positive scalar

Time step, specified as a positive scalar. Units are in seconds.

Data Types: `double`

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 *a priori* estimate of the state vector propagated from the previous state. The prediction uses the state transition function specified in the `StateTransitionFcn` property.

Data Types: `double`

Ppred — Predicted state error covariance matrix
real-valued *M*-by-*M* matrix

Predicted state error covariance matrix, returned as a real-valued *M*-by-*M* matrix. This predicted error is the *a priori* estimate of the state error covariance matrix. `predict` uses the state transition function Jacobian specified in the `StateTransitionJacobianFcn` property.

Data Types: `double`

Introduced in R2017a

initialize

Class: trackingUKF

Initialize unscented Kalman filter

Syntax

```
initialize(filter,X,P)
initialize(filter,X,P,Name,Value)
```

Description

`initialize(filter,X,P)` initializes the unscented Kalman filter, `filter`, using the state, `X`, and the state covariance, `P`.

`initialize(filter,X,P,Name,Value)` initializes the Kalman filter properties using name-value pairs.

Note: you cannot change the size or type of properties you are initializing.

Input Arguments

`filter` — Unscented Kalman tracking filter
`Unscented Kalman filter object`

Kalman tracking filter, specified as an unscented Kalman filter object.

X — Initial unscented Kalman filter state
vector | matrix

Initial unscented Kalman filter state, specified as a vector or matrix.

P — Initial unscented Kalman filter state covariance
matrix

Initial unscented Kalman filter state covariance, specified as a matrix.

Introduced in R2018b

likelihood

Class: trackingUKF

Measurement likelihood

Syntax

```
measlikelihood = likelihood(filter,zmeas)
measlikelihood = likelihood(filter,zmeas,measparams)
```

Description

`measlikelihood = likelihood(filter,zmeas)` returns the likelihood of the measurement, `zmeas`, of an object tracked by the unscented Kalman filter, `filter`.

`measlikelihood = likelihood(filter,zmeas,measparams)` also specifies measurement parameters, `measparams`.

Input Arguments

`filter` — Unscented Kalman tracking filter
`Unscented Kalman filter object`

Unscented Kalman tracking filter, specified as an unscented Kalman filter object.

`zmeas` — Measurement of tracked object
vector | matrix

Measurement of the 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 defined in the `MeasurementFcn` property of the unscented Kalman filter, `filter`.

Output Arguments

`measlikelihood` — Likelihood of measurement
scalar

Likelihood of measurement, returned as a scalar.

Introduced in R2018a

residual

Class: trackingUKF

Measurement residual and residual covariance

Syntax

```
[zres,rescov] = residual(filterobj,zmeas)
```

Description

`[zres,rescov] = residual(filterobj,zmeas)` computes the residual, `zres`, between a measurement, `zmeas`, and a predicted measurement produced by the Kalman filter, `filterobj`. The function also returns the covariance of the residual, `zres`.

Input Arguments

`filterobj` — Unscented Kalman tracking filter
`Kalman filter object`

Unscented Kalman tracking filter, specified as a Kalman filter object.

`zmeas` — Measurement of tracked object
vector | matrix

Measurement of the tracked object, specified as a vector or matrix.

Output Arguments

`zres` — Residual between measurement and predicted measurement
matrix

Residual between measurement and predicted measurement, returned as a matrix.

`rescov` — Covariance of residuals
matrix

Covariance of the residuals, returned as a matrix.

Algorithms

- The residual is the difference between a measurement and the value predicted by the filter. The residual d is defined as $d = z - h(x)$. h is the measurement function set by the `MeasurementFcn` property, x is the current filter state, and z is the current measurement.
- The covariance of the residual, S, is computed as $S = R + R_p$. R_p is the state covariance matrix projected onto the measurement space and R is the measurement noise matrix set by the `MeasurementNoise` property.

Introduced in R2018a

objectDetection class

Create object detection report

Description

The `objectDetection` class creates and reports detections of objects in a driving scenario. Each report contains information obtained by a sensor for a single object. You can use the `objectDetection` output as the input to a tracker such as `multiObjectTracker`.

Construction

`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. The dimension *N* is determined by the type of measurement. For example, a measurement of the Cartesian coordinates implies that $N = 3$. A measurement of spherical coordinates and range rate implies that $N = 4$. This argument sets the Measurement property.

Output Arguments

`detection` — Detection report
`objectDetection` class object

Detection report, returned as an `objectDetection` class object. An `objectDetection` class 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
`MeasurementParameters`	Parameters used by initialization functions of nonlinear Kalman tracking filters
`ObjectAttributes`	Additional information passed to tracker

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.

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.

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) | cell array

Measurement function parameters, specified as a cell array. The cell array contains all the arguments used by the measurement function specified by the `MeasurementFcn` property of a nonlinear tracking filter such as `trackingEKF` or `trackingUKF`. Each cell contains a single argument.

Example: `{[1;0;0],'rectangular'}`

`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 time stamp 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

Classes

trackingEKF | trackingKF | trackingUKF

System Objects

multiObjectTracker | radarDetectionGenerator | visionDetectionGenerator

Introduced in R2017a

multiObjectTracker System object

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 4-271.

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? (MATLAB).

Creation

Syntax

```
tracker = multiObjectTracker
tracker = multiObjectTracker(Name,Value)
```

Description

`tracker = multiObjectTracker` creates a `multiObjectTracker` System objectwith default property values.

`tracker = multiObjectTracker(Name,Value)` sets properties 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 (MATLAB).

`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 System 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 an object belonging to one of the Kalman filter classes: `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` (default) | positive scalar

Detection assignment threshold, specified as a positive 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, increase the threshold. If some detections are assigned to incorrect tracks, decrease the threshold.

Data Types: `double`

`ConfirmationParameters` — Confirmation parameters for track creation
`[2 3]` (default) | two-element vector of positive increasing integers

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

- 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`

`NumCoastingUpdates` — Coasting threshold for track deletion
`5` (default) | positive integer

Coasting threshold for track deletion, specified as a positive integer. A track coasts when no detections are assigned to that confirmed track after one or more prediction steps. If the number of coasting steps exceeds this coasting threshold, the object deletes the track.

Data Types: `double`

`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

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

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)
```

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` structure array corresponds to a single track.

`[confirmedTracks,tentativeTracks] = tracker(detections,time)` also returns a structure array containing details about the tentative tracks.

`[confirmedTracks,tentativeTracks,allTracks] = tracker(detections, time)` also returns a structure array containing details about all the confirmed and tentative tracks, `allTracks`. 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`.

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

scalar

Time of update, specified as a 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`

Output Arguments

`confirmedTracks` — Confirmed tracks

structure array

Confirmed tracks, returned as a structure array with these fields.

Field	Definition
`TrackID`	Unique track identifier.
`Time`	Time at which the track is updated. Units are in seconds.
`Age`	Number of updates since track initialization.
`State`	Updated state vector. The state vector is specific to each type of Kalman filter.
`StateCovariance`	Updated state covariance matrix. The covariance matrix is specific to each type of Kalman filter.
`IsConfirmed`	Confirmation status. This field is `true` if the track is confirmed to be a real target.
`IsCoasted`	Coasting status. This field is `true` if the track is updated without a new detection.
`ObjectClassID`	Integer value representing the object classification. The value `0` represents an unknown classification. Nonzero classifications apply only to confirmed tracks.
`ObjectAttributes`	Cell array of object attributes reported by the sensor making the detection.

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 N]`, use the `ConfirmationParameters` property of the multi-object tracker.
- The `objectDetection` object initiating the track has an `ObjectClassID` greater than zero.

`tentativeTracks` — Tentative tracks

structure array

Tentative tracks, returned as a structure array with these fields.

Field	Definition
TrackID	Unique track identifier.
Time	Time at which the track is updated. Units are in seconds.
Age	Number of updates since track initialization.
State	Updated state vector. The state vector is specific to each type of Kalman filter.
StateCovariance	Updated state covariance matrix. The covariance matrix is specific to each type of Kalman filter.
IsConfirmed	Confirmation status. This field is `true` if the track is confirmed to be a real target.
IsCoasted	Coasting status. This field is `true` if the track is updated without a new detection.
ObjectClassID	Integer value representing the object classification. The value `0` represents an unknown classification. Nonzero classifications apply only to confirmed tracks.
ObjectAttributes	Cell array of object attributes reported by the sensor making the detection.

A track is tentative before it is confirmed.

allTracks — All confirmed and tentative tracks
structure array

All confirmed and tentative tracks, returned as a structure array with these fields.

Field	Definition
TrackID	Unique track identifier.
Time	Time at which the track is updated. Units are in seconds.

Field	Definition
`Age`	Number of updates since track initialization.
`State`	Updated state vector. The state vector is specific to each type of Kalman filter.
`StateCovariance`	Updated state covariance matrix. The covariance matrix is specific to each type of Kalman filter.
`IsConfirmed`	Confirmation status. This field is `true` if the track is confirmed to be a real target.
`IsCoasted`	Coasting status. This field is `true` if the track is updated without a new detection.
`ObjectClassID`	Integer value representing the object classification. The value `0` represents an unknown classification. Nonzero classifications apply only to confirmed tracks.
`ObjectAttributes`	Cell array of object attributes reported by the sensor making the detection.

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`

isLocked	Determine if System object is in use
getTrackFilterProperties	Obtain filter properties of track from multi-object tracker
setTrackFilterProperties	Set filter properties of track from multi-object tracker
updateTracks	Update multi-object tracker with new detections

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

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('ConfirmationParameters',[4 5], ...
    'NumCoastingUpdates',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, ...
    'ConfirmationParameters',[3 4],'NumCoastingUpdates',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, ...
    'ConfirmationParameters',[3 4],'NumCoastingUpdates',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`.

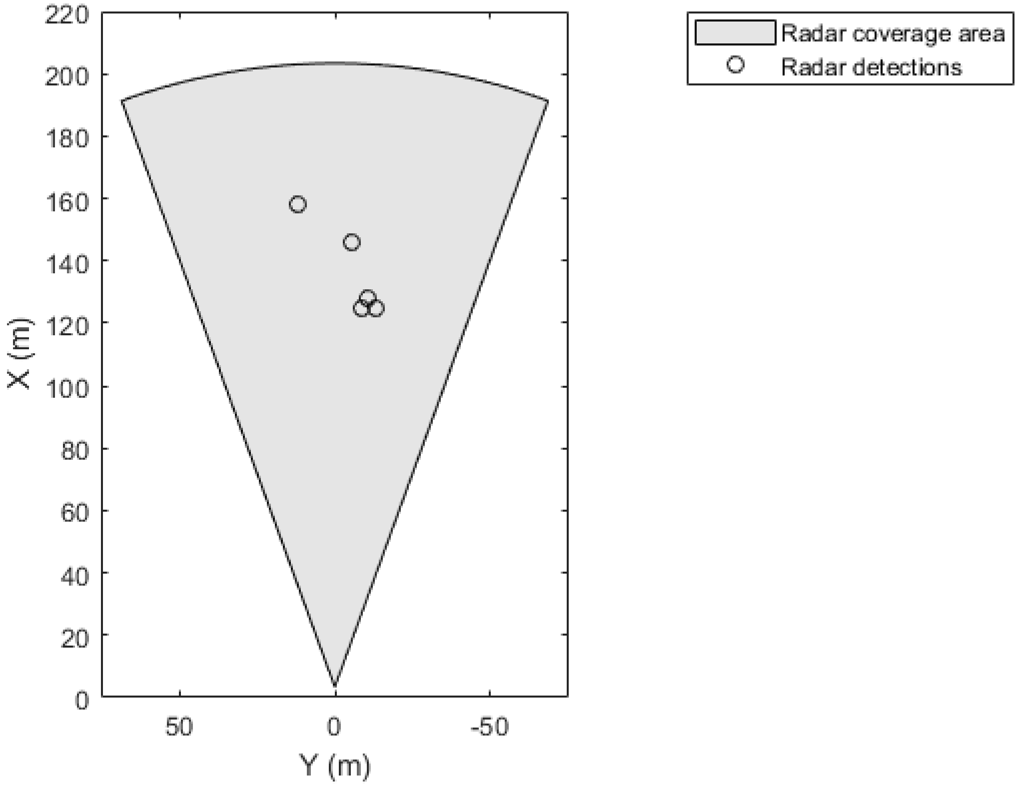
```
simTime = 0;
nsteps = 10;
for k = 1:nsteps
    dets = radar([car1 car2 car3],simTime);
    [confirmedTracks,tentativeTracks,allTracks] = tracker(dets,simTime);
```

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, using the `assignDetectionsToTracks` function.
- Creates new tracks from unassigned detections.
- Updates already assigned tracks and possibly confirms them, based on the `ConfirmationParameters` property of the multi-object tracker.

- Deletes tracks that have no assigned detections within the last `NumCoastingUpdates` updates.

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`

Classes
`drivingScenario` | `objectDetection` | `trackingEKF` | `trackingKF` | `trackingUKF`

System Objects
`radarDetectionGenerator` | `visionDetectionGenerator`

Topics

“Multiple Object Tracking Tutorial”

“Track Multiple Vehicles Using a Camera”
“Track Pedestrians from a Moving Car”

Introduced in R2017a

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],'NumCoastingUpdates',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 = 2x1 struct array with fields:
    TrackID
    Time
    Age
    State
    StateCovariance
    IsConfirmed
    IsCoasted
    ObjectClassID
    ObjectAttributes
```

Get filter property values for the first track. Display the process noise values.

```
values = getTrackFilterProperties(tracker,1,'MeasurementNoise','ProcessNoise','MotionMo
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

System Objects
multiObjectTracker

Classes
trackingEKF | trackingKF | trackingUKF

Functions
setTrackFilterProperties | updateTracks

Introduced in R2017a

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],'NumCoastingUpdates',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 = 2x1 struct array with fields:
    TrackID
    Time
    Age
    State
    StateCovariance
    IsConfirmed
    IsCoasted
    ObjectClassID
    ObjectAttributes
```

Get filter property values for the first track. Display the process noise values.

```
values = getTrackFilterProperties(tracker,1,'MeasurementNoise','ProcessNoise','MotionMo
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

System Objects
`multiObjectTracker`

Classes
`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)
```

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` structure array corresponds to a single track.

`[confirmedTracks,tentativeTracks] = updateTracks(tracker,detections, time)` also returns a structure array containing details about the tentative tracks.

`[confirmedTracks,tentativeTracks,allTracks] = updateTracks(tracker, detections,time)` also returns a structure array containing details about all confirmed and tentative tracks, `allTracks`. 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`.

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, ...
    'ConfirmationParameters',[3 4],'NumCoastingUpdates',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`.

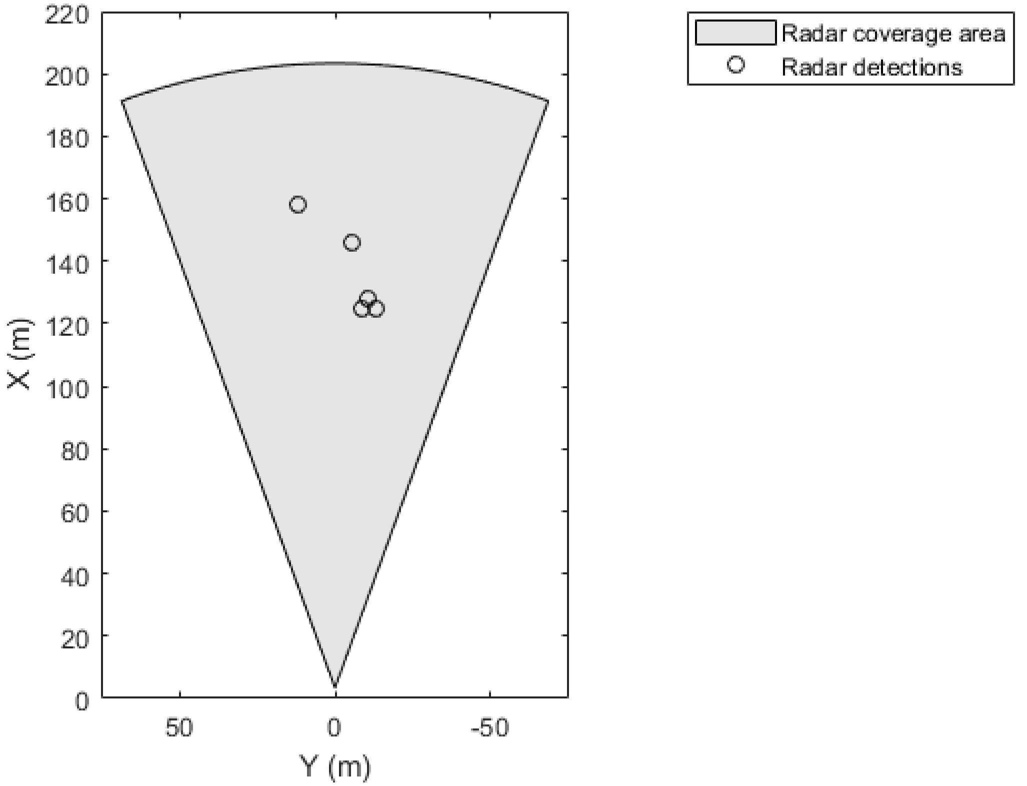
```
simTime = 0;
nsteps = 10;
for k = 1:nsteps
    dets = radar([car1 car2 car3],simTime);
    [confirmedTracks,tentativeTracks,allTracks] = updateTracks(tracker,dets,simTime);
```

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
scalar

Time of update, specified as a 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`

Output Arguments

`confirmedTracks` — Confirmed tracks
structure array

Confirmed tracks, returned as a structure array with these fields.

Field	Definition
TrackID	Unique track identifier.
Time	Time at which the track is updated. Units are in seconds.
Age	Number of updates since track initialization.
State	Updated state vector. The state vector is specific to each type of Kalman filter.
StateCovariance	Updated state covariance matrix. The covariance matrix is specific to each type of Kalman filter.
IsConfirmed	Confirmation status. This field is `true` if the track is confirmed to be a real target.
IsCoasted	Coasting status. This field is `true` if the track is updated without a new detection.
ObjectClassID	Integer value representing the object classification. The value `0` represents an unknown classification. Nonzero classifications apply only to confirmed tracks.
ObjectAttributes	Cell array of object attributes reported by the sensor making the detection.

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 N]`, use the `ConfirmationParameters` property of the multi-object tracker.
- The `objectDetection` object initiating the track has an `ObjectClassID` greater than zero.

`tentativeTracks` — Tentative tracks
structure array

Tentative tracks, returned as a structure array with these fields.

Field	Definition
TrackID	Unique track identifier.
Time	Time at which the track is updated. Units are in seconds.
Age	Number of updates since track initialization.
State	Updated state vector. The state vector is specific to each type of Kalman filter.
StateCovariance	Updated state covariance matrix. The covariance matrix is specific to each type of Kalman filter.
IsConfirmed	Confirmation status. This field is true if the track is confirmed to be a real target.
IsCoasted	Coasting status. This field is true if the track is updated without a new detection.
ObjectClassID	Integer value representing the object classification. The value 0 represents an unknown classification. Nonzero classifications apply only to confirmed tracks.
ObjectAttributes	Cell array of object attributes reported by the sensor making the detection.

A track is tentative before it is confirmed.

allTracks — All confirmed and tentative tracks
structure array

All confirmed and tentative tracks, returned as a structure array with these fields.

Field	Definition
TrackID	Unique track identifier.
Time	Time at which the track is updated. Units are in seconds.

Field	Definition
`Age`	Number of updates since track initialization.
`State`	Updated state vector. The state vector is specific to each type of Kalman filter.
`StateCovariance`	Updated state covariance matrix. The covariance matrix is specific to each type of Kalman filter.
`IsConfirmed`	Confirmation status. This field is `true` if the track is confirmed to be a real target.
`IsCoasted`	Coasting status. This field is `true` if the track is updated without a new detection.
`ObjectClassID`	Integer value representing the object classification. The value `0` represents an unknown classification. Nonzero classifications apply only to confirmed tracks.
`ObjectAttributes`	Cell array of object attributes reported by the sensor making the detection.

Algorithms

When you pass detections into `updateTracks`, the function:

- Attempts to assign the input detections to existing tracks, using the `assignDetectionsToTracks` function.
- Creates new tracks from unassigned detections.
- Updates already assigned tracks and possibly confirms them, based on the `ConfirmationParameters` property of the multi-object tracker.
- Deletes tracks that have no assigned detections within the last `NumCoastingUpdates` updates.

See Also

Classes
`objectDetection`

System Objects
`multiObjectTracker`

Functions
`getTrackFilterProperties` | `setTrackFilterProperties`

Introduced in R2017a

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]` numeric 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]` numeric 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]` numeric vector

Coefficients for a parabolic model of the form $y = Ax^2 + Bx + C$, specified as an `[A B C]` numeric 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

numeric scalar

Strength of the boundary model, specified as a numeric 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]` numeric vector

Length of the boundary along the *x*-axis, specified as a `[minX maxX]` numeric vector that describes the minimum and maximum *x*-axis locations.

Object Functions

computeBoundaryModel   Obtain y-coordinates of lane boundaries given x-coordinates

Examples

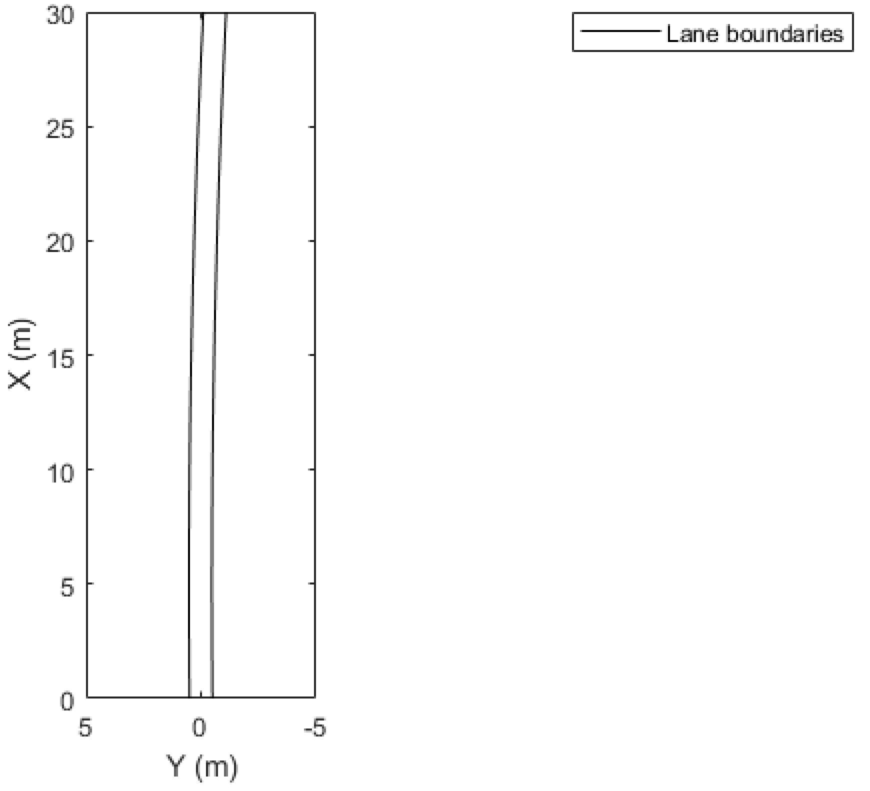
Create Parabolic Lane Boundaries

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

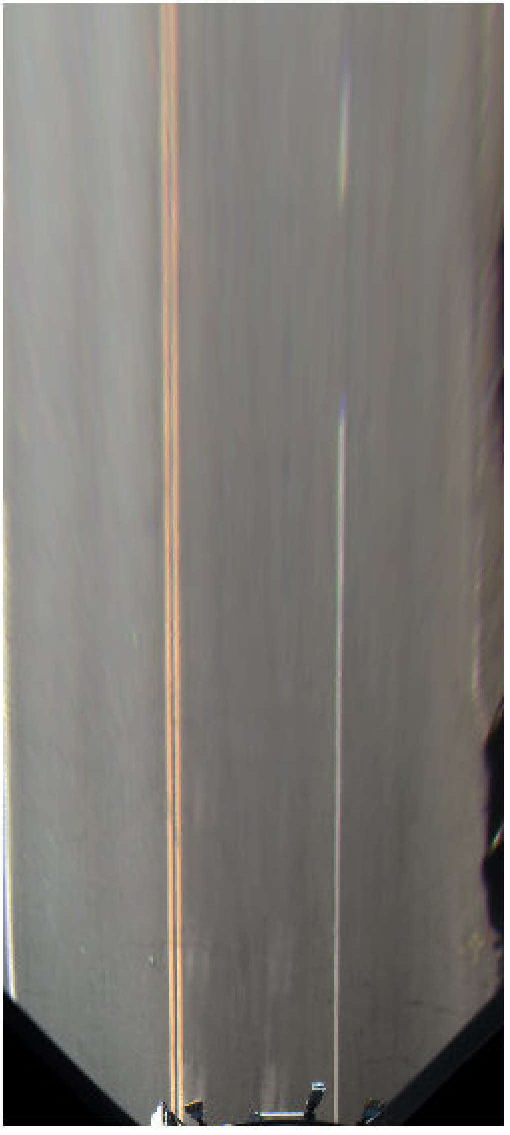
Find lanes in an image by using parabolic lane boundary models. Overlay the identified lanes on the original image and on a bird's-eye-view transformation of the image.

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)
```

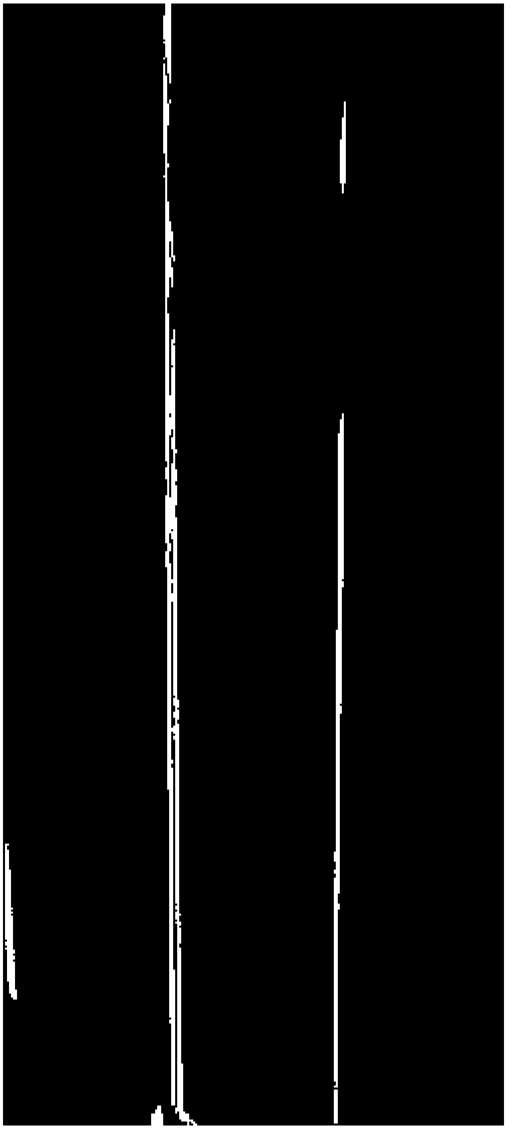


Set the approximate lane marker width in world units (meters).

```
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.

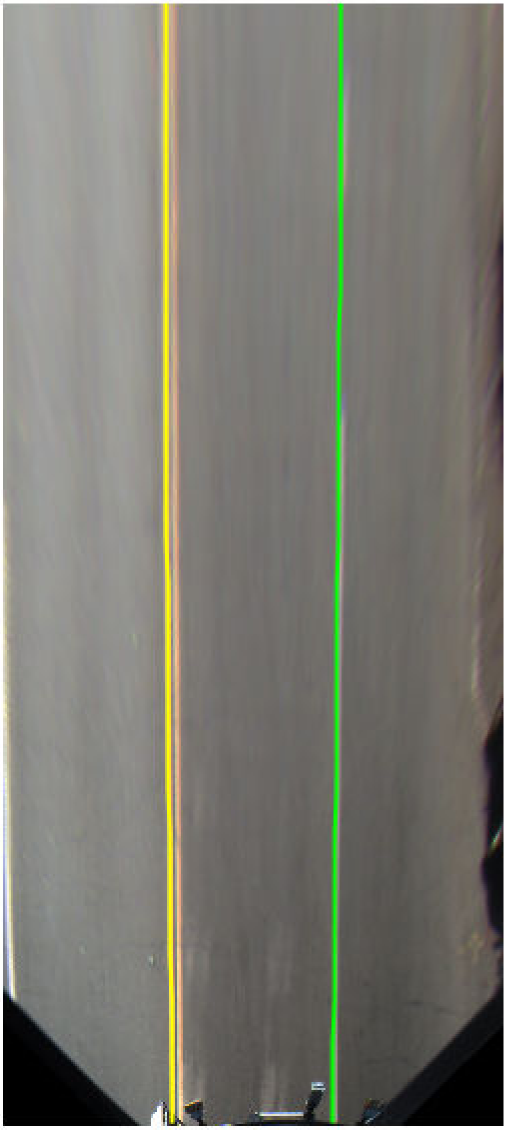
```
boundaries = findParabolicLaneBoundaries(xyBoundaryPoints,approxBoundaryWidth);
```

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)
```



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]` numeric 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]` numeric 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]` numeric vector

Coefficients for a cubic model of the form $y = Ax^3 + Bx^2 + Cx + D$, specified as an `[A B C D]` numeric 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
numeric scalar

Strength of the boundary model, specified as a numeric 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]` numeric vector

Length of the boundary along the *x*-axis, specified as a `[minX maxX]` numeric vector that describes the minimum and maximum *x*-axis locations.

Object Functions

computeBoundaryModel   Obtain y-coordinates of lane boundaries given x-coordinates

Examples

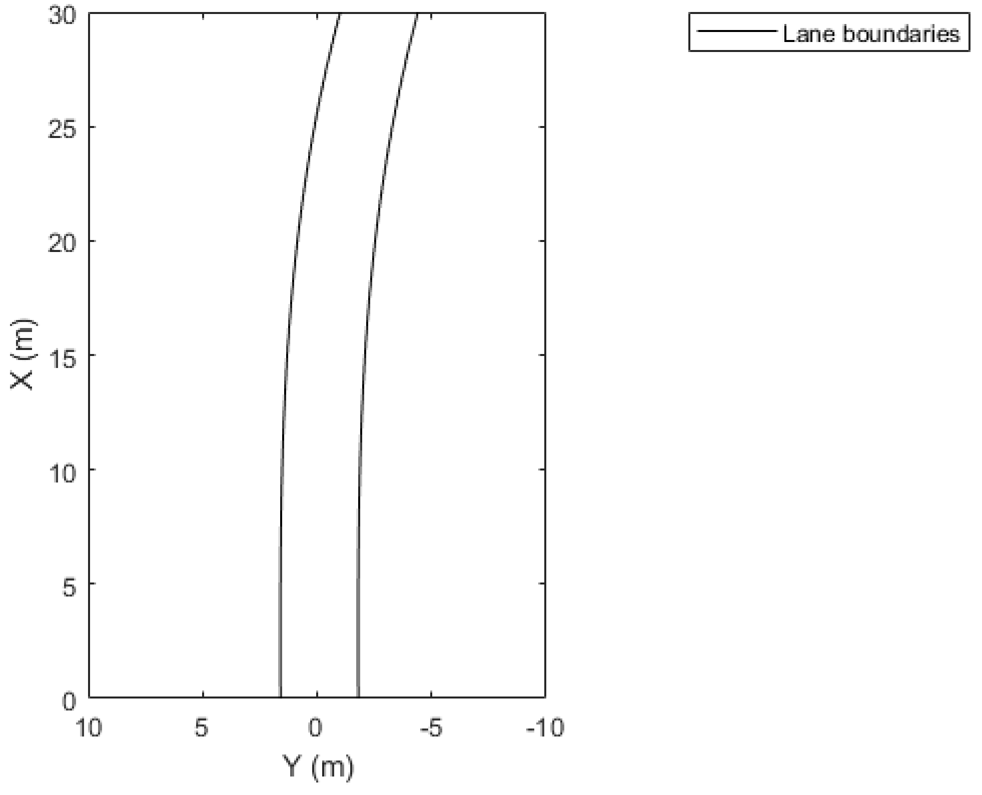
Create Cubic Lane Boundaries

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

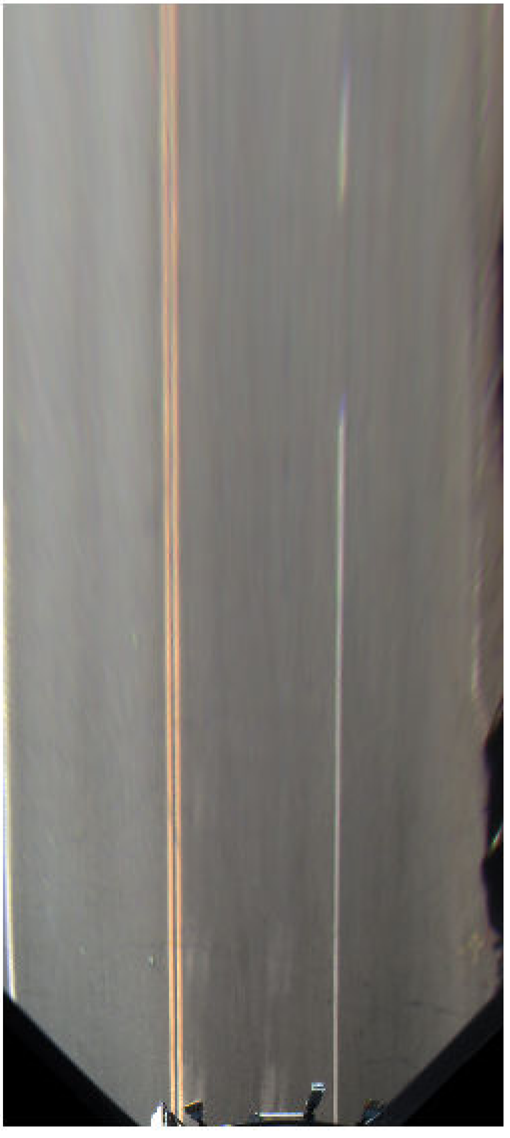
Find lanes in an image by using cubic lane boundary models. Overlay the identified lanes on the original image and on a bird's-eye-view transformation of the image.

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)
```

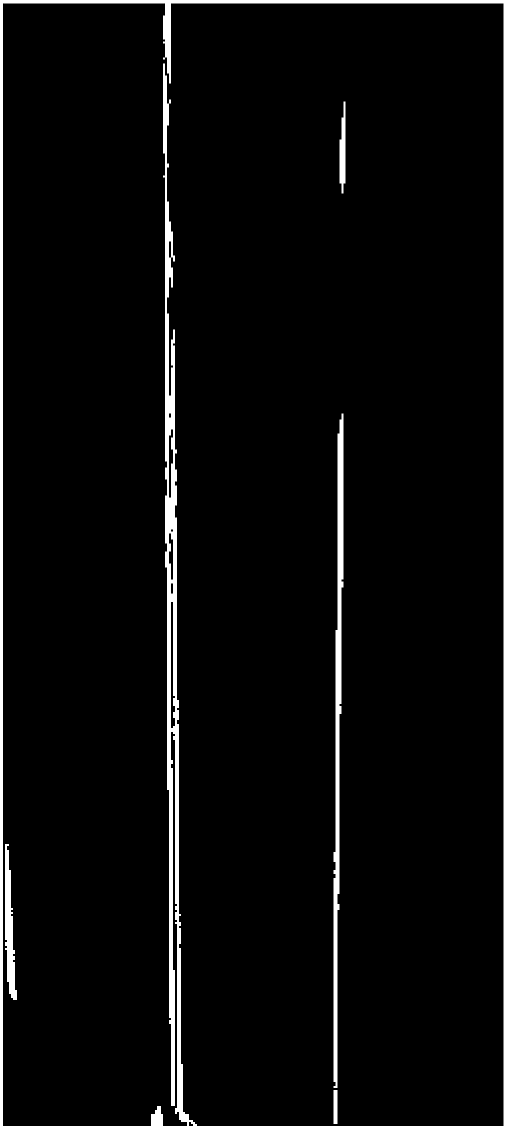


Set the approximate lane marker width in world units (meters).

```
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.

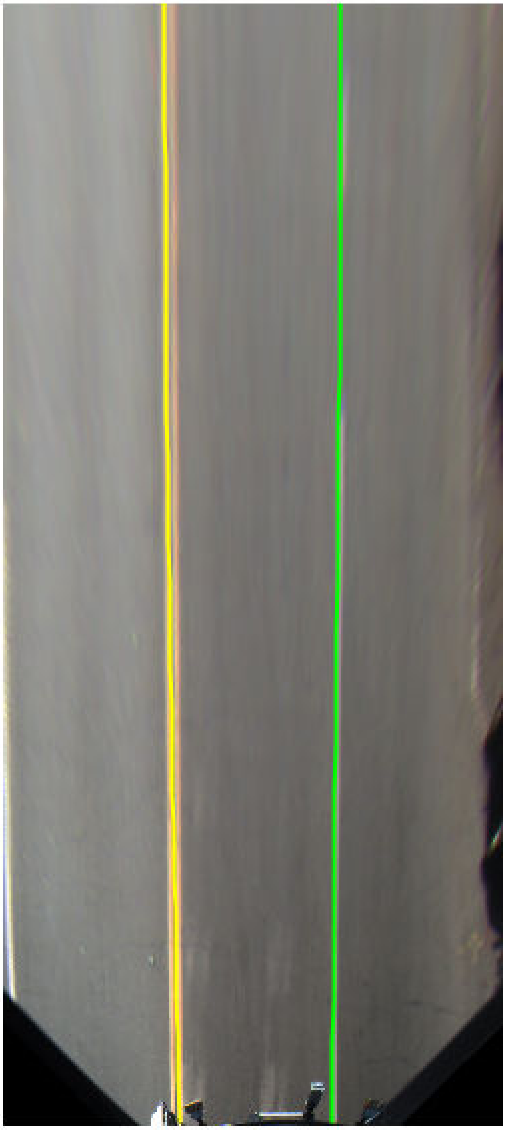
```
boundaries = findCubicLaneBoundaries(xyBoundaryPoints,approxBoundaryWidth);
```

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)
```



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.

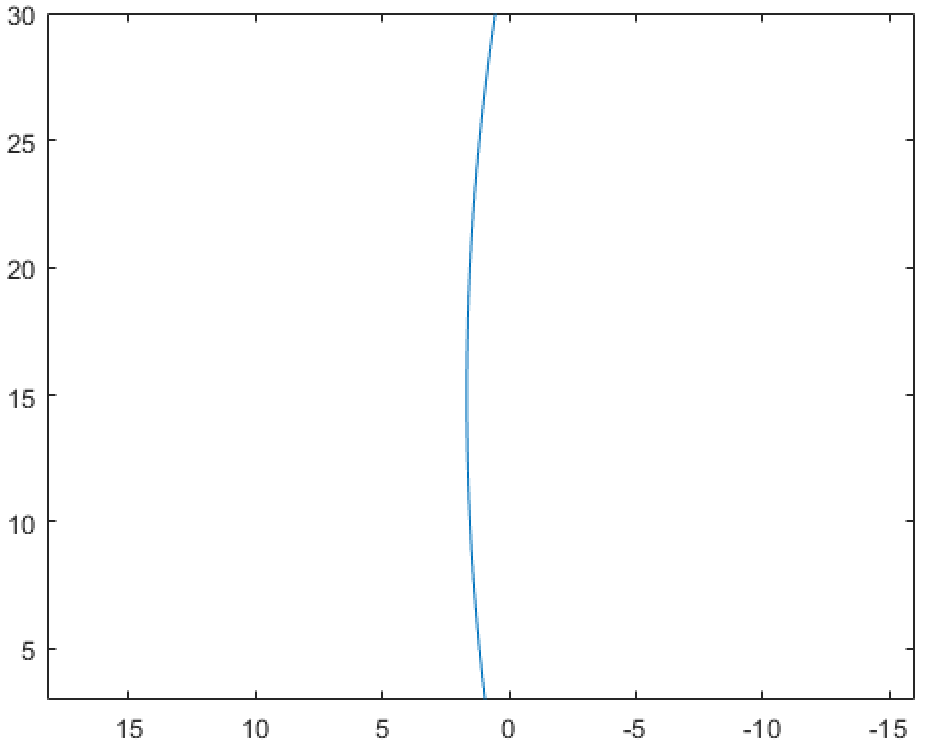
```
parabolicParams = [-0.005 0.15 0.55];
lb = parabolicLaneBoundary(parabolicParams);
```

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.

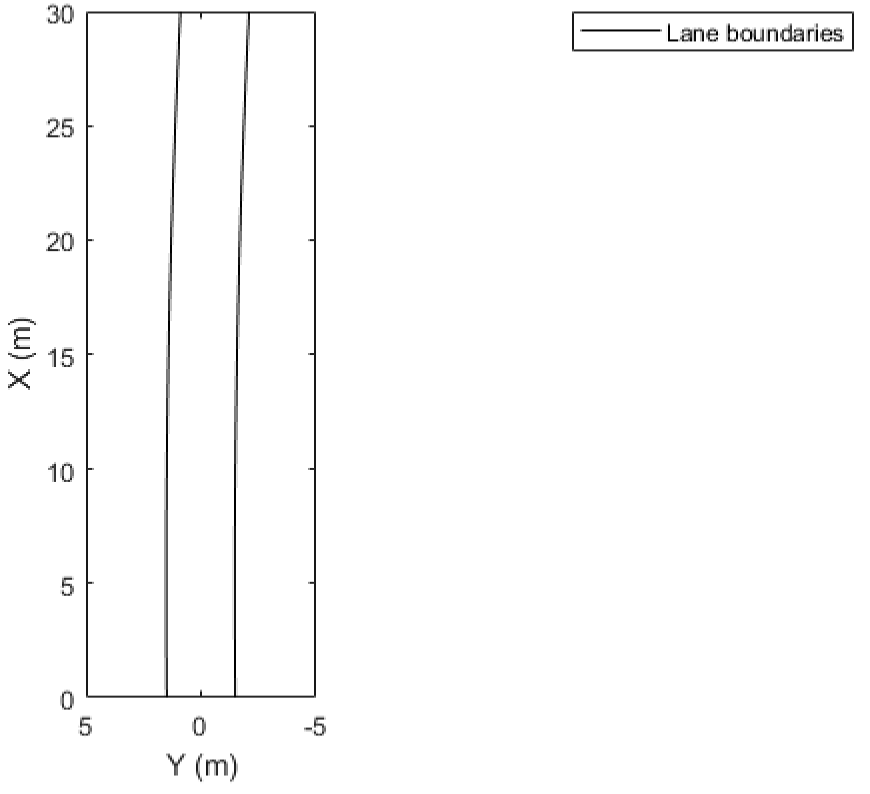
```
lb = parabolicLaneBoundary([-0.001,0.01,1.5]);
rb = parabolicLaneBoundary([-0.001,0.01,-1.5]);
```

Compute the model manually up to 30 meters ahead in the lane.

```
xWorld = (0:30)';
yLeft = computeBoundaryModel(lb,xWorld);
yRight = computeBoundaryModel(rb,xWorld);
```

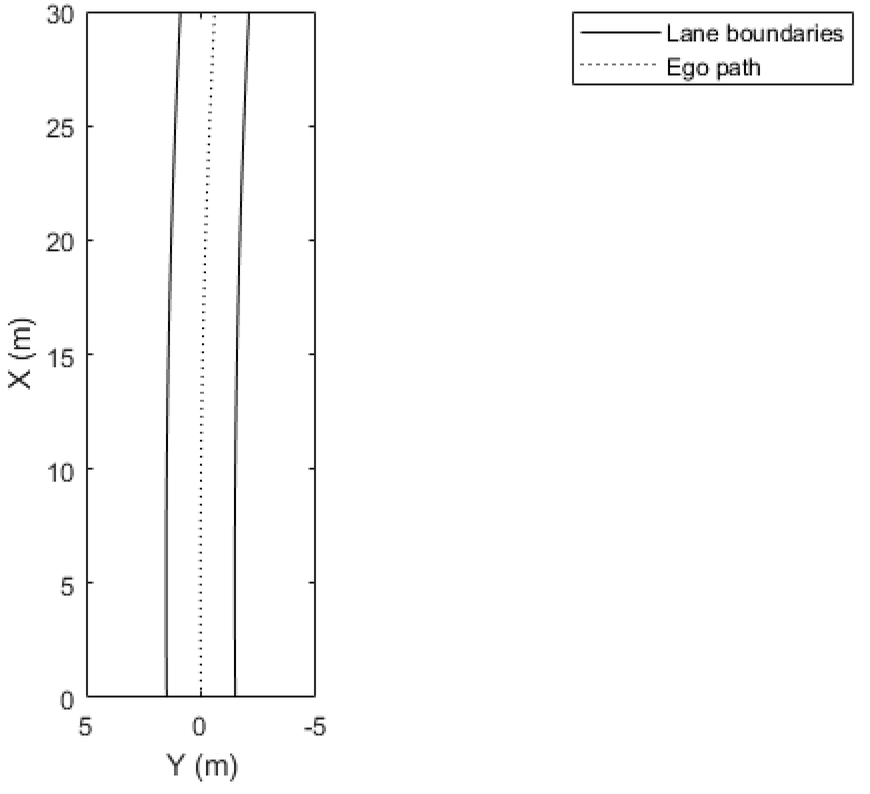
Create a bird's-eye plot and plot the lane information.

```
bep = birdsEyePlot('XLimits',[0 30],'YLimits',[-5 5]);
lanePlotter = laneBoundaryPlotter(bep,'DisplayName','Lane boundaries');
plotLaneBoundary(lanePlotter,{[xWorld,yLeft],[xWorld,yRight]});
```



Plot the path of an ego vehicle that travels through the center of the lane.

```
yCenter = (yLeft + yRight)/2;
egoPathPlotter = pathPlotter(bep,'DisplayName','Ego 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.

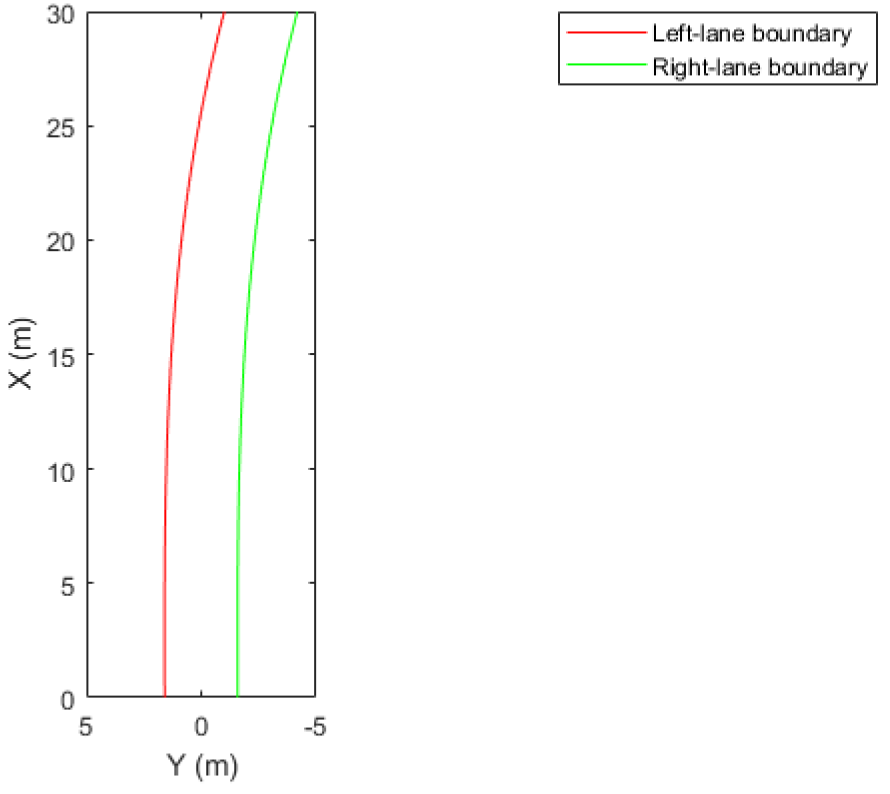
```
xWorld = 0; % meters
yWorld = computeBoundaryModel(lbs,0);
```

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

numeric scalar | numeric vector

x-axis locations of the boundaries in world coordinates, specified as a numeric scalar or vector.

See Also

Objects
`cubicLaneBoundary` | `parabolicLaneBoundary`

Functions
`insertLaneBoundary`

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.

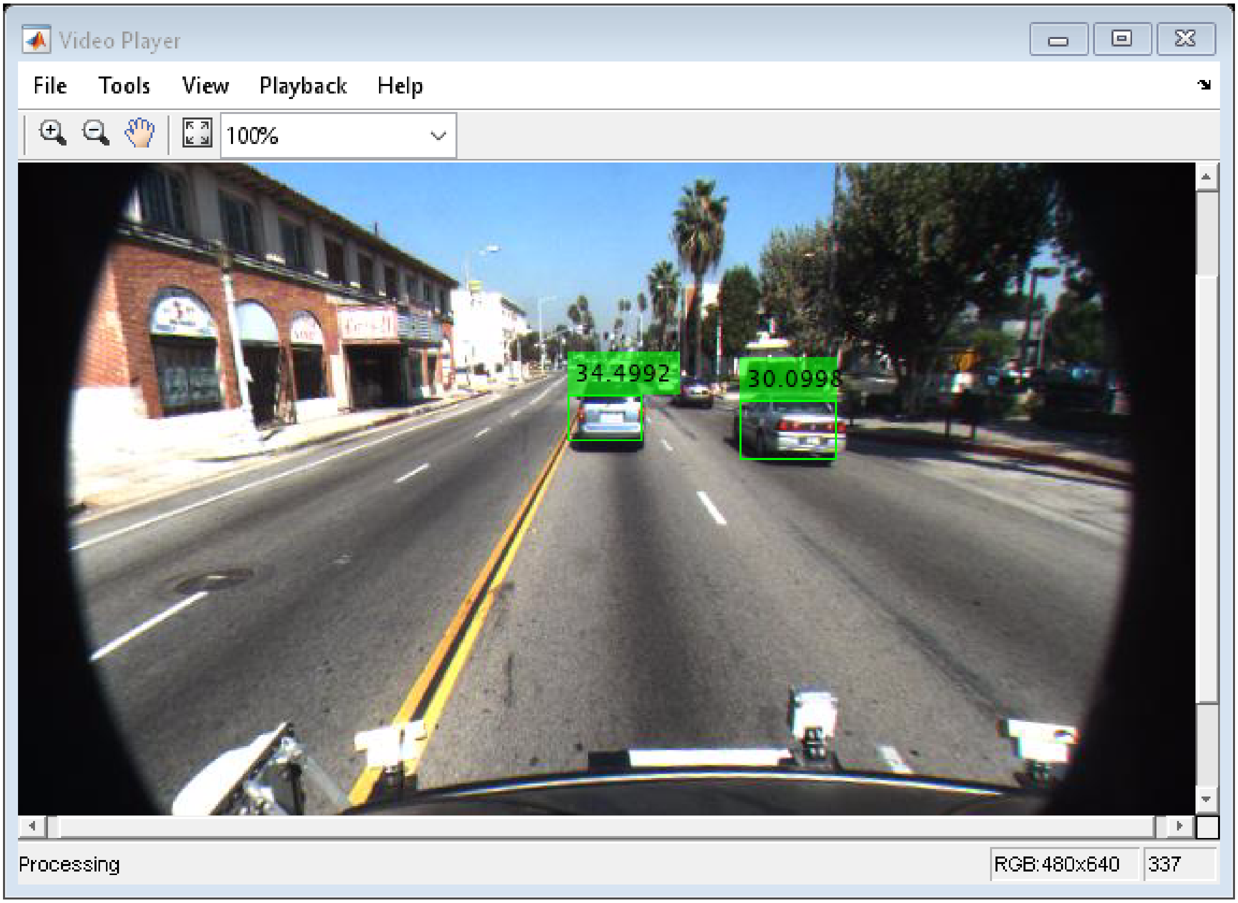
```
vehicleWidth = [1.5 2.5];
detectorMonoCam = configureDetectorMonoCamera(detector,monCam,vehicleWidth);
```

Load a video captured from the camera, and create a video reader and player.

```
videoFile = fullfile(toolboxdir('driving'),'drivingdata','caltech_washington1.avi');
reader = vision.VideoFileReader(videoFile,'VideoOutputDataType','uint8');
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 = ~isDone(reader);
while cont
   I = 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 = ~isDone(reader) && isOpen(videoPlayer);
end
```



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);
```

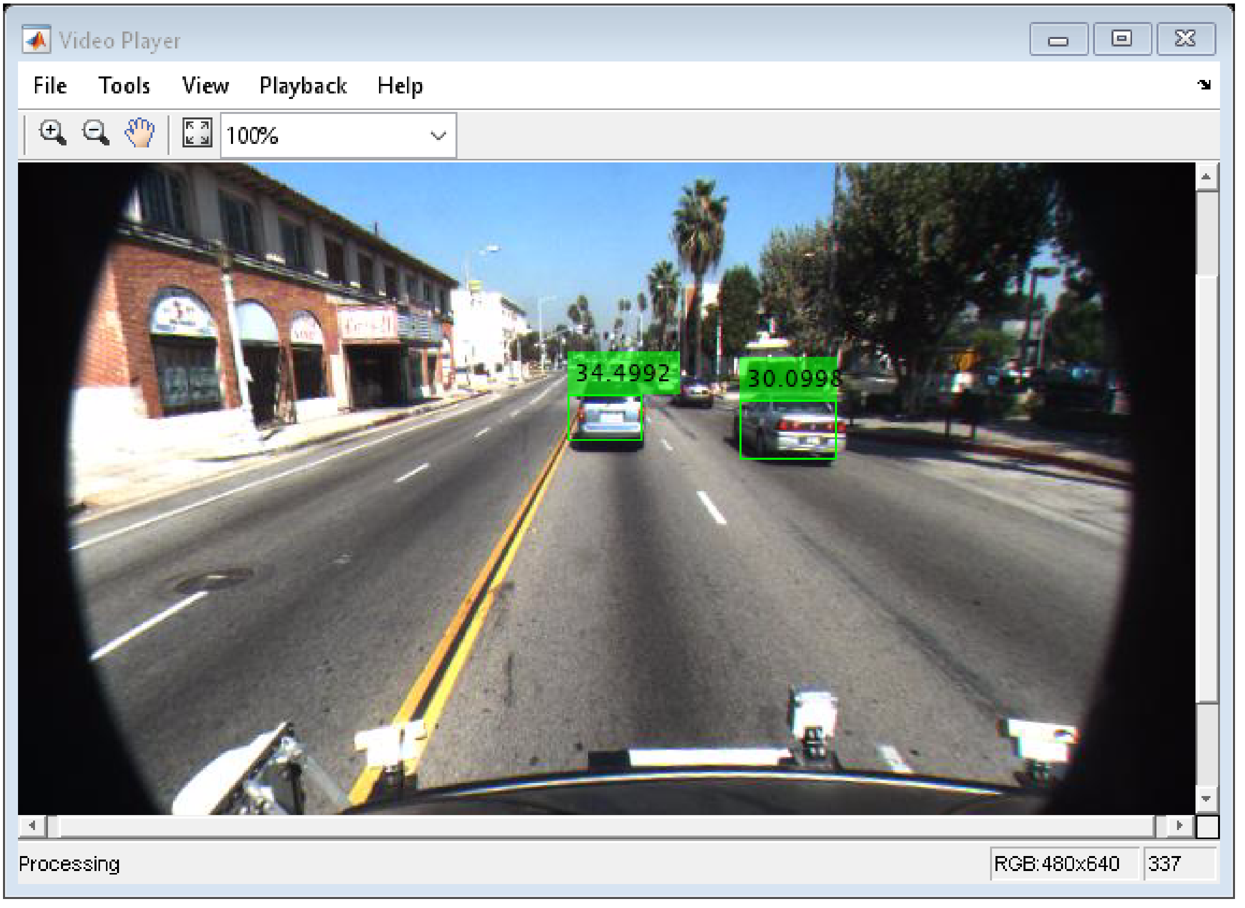
Load a video captured from the camera, and create a video reader and player.

```
videoFile = fullfile(toolboxdir('driving'),'drivingdata','caltech_washington1.avi');
reader = vision.VideoFileReader(videoFile,'VideoOutputDataType','uint8');
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 = ~isDone(reader);
while cont
   I = 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 = ~isDone(reader) && isOpen(videoPlayer);
end
```



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
"R-CNN, Fast R-CNN, and Faster R-CNN Basics" (Computer Vision System 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)
[___] = 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`.

`[___] = 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
grayscale image | RGB image

Input image, specified as a real, nonsparse, grayscale or RGB image.

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

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: 'NumStongestRegions',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.

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 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 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.

labels — Labels for bounding boxes
M-by-1 categorical array

Labels for bounding boxes, returned as an *M*-by-1 categorical array of *M* labels. You define the class names used to label the objects when you train the input detector.

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

"R-CNN, Fast R-CNN, and Faster R-CNN Basics" (Computer Vision System 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)
[___] = 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`.

`[___] = 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
grayscale image | RGB image

Input image, specified as a real, nonsparse, grayscale or RGB image.

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`

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: `'NumStongestRegions',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.

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 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 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.

`labels` — Labels for bounding boxes
M-by-1 categorical array

Labels for bounding boxes, returned as an *M*-by-1 categorical array of *M* labels. You define the class names used to label the objects when you train the input `detector`.

See Also

Apps
Ground Truth Labeler

Functions
`configureDetectorMonoCamera` | `selectStrongestBboxMulticlass` | `trainFasterRCNNObjectDetector`

Objects
`fasterRCNNObjectDetectorMonoCamera` | `monoCamera`

Introduced in R2017a

drivingScenario class

Create driving scenario

Description

The `drivingScenario` class creates a driving scenario object. A driving scenario is a 3-D arena containing roads and actors. Actors represent anything that moves, such as cars, pedestrians, bicycles, and other objects. Actors can also include stationary obstacles that can influence the motion of other actors. There are two classes of actors. The first class is a general-purpose actor belonging to the `Actor` class. All actors are modeled as cuboid, that is, box shapes. The second class is vehicles. Vehicles are a special type of actor with additional properties and belong to the `Vehicle` class. Except when noted, references to an actor includes vehicles as well. You can populate the scenario by using the `actor`, `vehicle`, and `road` methods.

Construction

`sc = drivingScenario` returns an empty driving scenario.

`sc = drivingScenario(Name,Value)` uses name-value pair arguments to specify the `SampleTime` and `StopTime` properties. Enclose each property name in quotes.

Properties

`SampleTime` — Time interval between scenario simulation steps
`0.01` (default) | positive scalar

Time interval between scenario simulation steps, specified as a positive scalar. Units are in seconds.

Example: `1.5`

Data Types: `double`

`StopTime` — End time of simulation
`Inf` (default) | positive scalar

End time of simulation, specified as a positive scalar. Units are in seconds.

Example: `60.0`

Data Types: `double`

`SimulationTime` — Current time of simulation
positive scalar

This property is read-only.

Current time of the simulation, specified as a positive scalar. To reset the time to zero and restart the simulation, call the `restartSimulation` method. Units are in seconds.

Data Types: `double`

`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`.

Data Types: `logical`

`Actors` — Actors and vehicles contained in scenario
heterogeneous array of actors

This property is read-only.

Actors and vehicles contained in the scenario, specified as a heterogeneous array. To add an actor to the scenario, use the `actor` or `vehicle` method.

Methods

actor	Create an actor within driving scenario
actorPoses	Positions, velocities, and orientations of actors in driving scenario
actorProfiles	Physical and radar properties of actors in driving scenario
advance	Advance driving scenario simulation by one time step
vehicle	Create a vehicle within driving scenario
plot	Create driving scenario plot
road	Add a road to driving scenario
roadNetwork	Add road network to driving scenario
record	Run driving scenario and record actor states
restart	Restart driving scenario simulation from beginning
updatePlots	Update driving scenario plots
laneMarkingVertices	Lane marking vertices and faces

chasePlot	Egocentric projective perspective plot
currentLane	Current lane of actor
laneBoundaries	Lane boundaries
roadBoundaries	Show road boundaries
targetOutlines	Outlines of targets viewed by actor
targetPoses	Target positions and orientations seen from an actor
trajectory	Create actor or vehicle trajectory in 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.

Set up the driving scenario object.

```
sc = 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 starting. 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(sc,roadcenters,roadwidth);
```

Add a two straight roads with the default width, using road center points at each end.

```
roadcenters = [700 0 0; 100 0 0];
road(sc,roadcenters)
roadcenters = [400 400 0; 0 0 0];
road(sc,roadcenters)
```

Get the road boundaries.

```
rbdry = roadBoundaries(sc);
```

Add a car and a bicycle to the scenario. Position the car at the beginning of the first straight road.

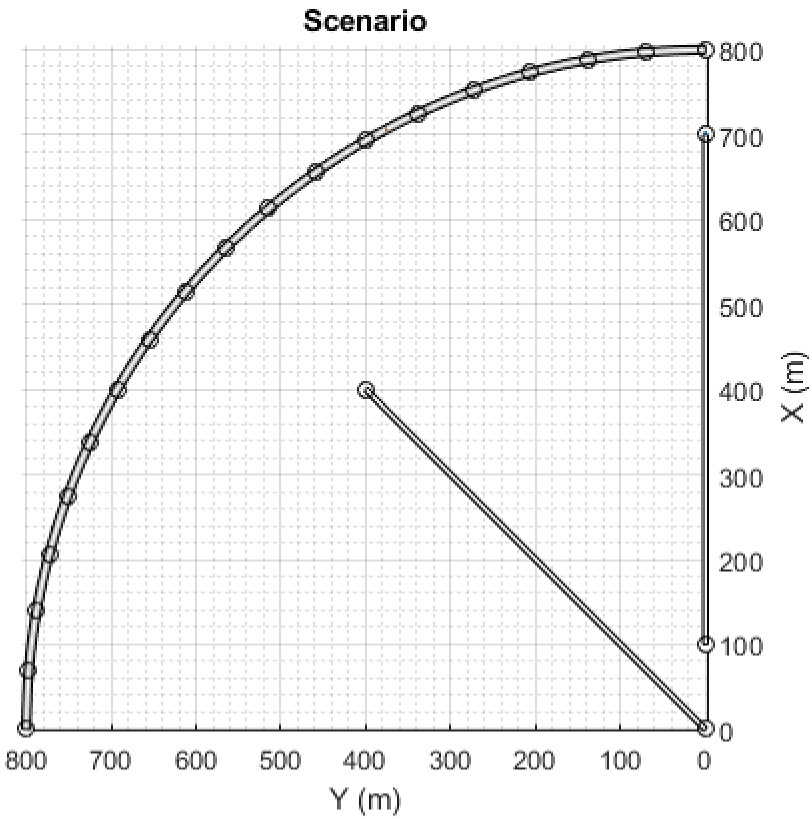
```
car = vehicle(sc,'Position',[700 0 0],'Length',3,'Width',2,'Height',1.6);
```

Position the bicycle farther down the road.

```
bicycle = actor(sc,'Position',[706 376 0]','Length',2,'Width',0.45,'Height',1.5);
```

Plot the scenario.

```
plot(sc,'Centerline','on','RoadCenters','on');
title('Scenario');
```



Display the actors poses and profiles.

```
poses = actorPoses(sc)
```

```
poses = 2x1 struct array with fields:
    ActorID
    Position
    Velocity
    Roll
    Pitch
    Yaw
    AngularVelocity
```

```
profiles = actorProfiles(sc)
```

```
profiles = 2x1 struct array with fields:
    ActorID
    ClassID
    Length
    Width
    Height
    OriginOffset
    RCSPattern
    RCSAzimuthAngles
    RCSElevationAngles
```

Show Target Outlines in Driving Scenario Simulation

Create a driving scenario and show how target outlines change as the simulation advances.

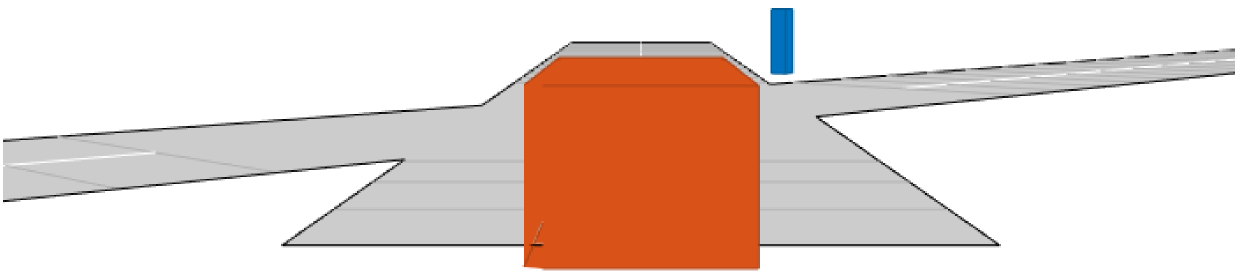
Set up a driving scenario with a vehicle and a pedestrian

Set up a driving scenario consisting of two intersecting straight roads. Construct one straight road segment to be 45 m long. The second straight road is 32 meters long and intersects the first road. A car travelling at 12.0 m/s along the first road approaches a running pedestrian crossing the intersection moving at 2.0 m/s.

```
s = drivingScenario('SampleTime',0.1,'StopTime',1);
road(s,[-10 0 0; 45 -20 0]);
road(s,[-10 -10 0; 35 10 0]);
ped = actor(s,'Length',0.4,'Width',0.6,'Height',1.7);
car = vehicle(s);
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 egocentric chase plot for the vehicle

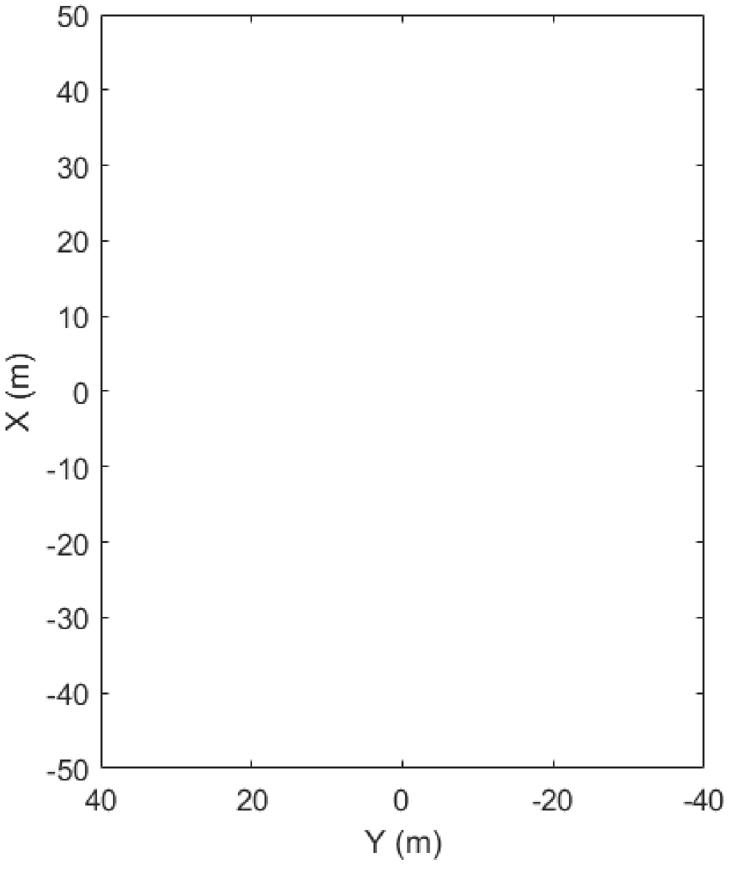
```
chasePlot(car,'Centerline','on')
```



Create a bird's-eye plot of road boundaries and actors

Create an empty bird's-eye plot and add an outline plotter and lane boundary plotter.

```
bepPlot = birdsEyePlot('XLim',[-50 50],'YLim',[-40 40]);
outlineplotter = outlinePlotter(bepPlot);
laneplotter = laneBoundaryPlotter(bepPlot);
legend('off')
```



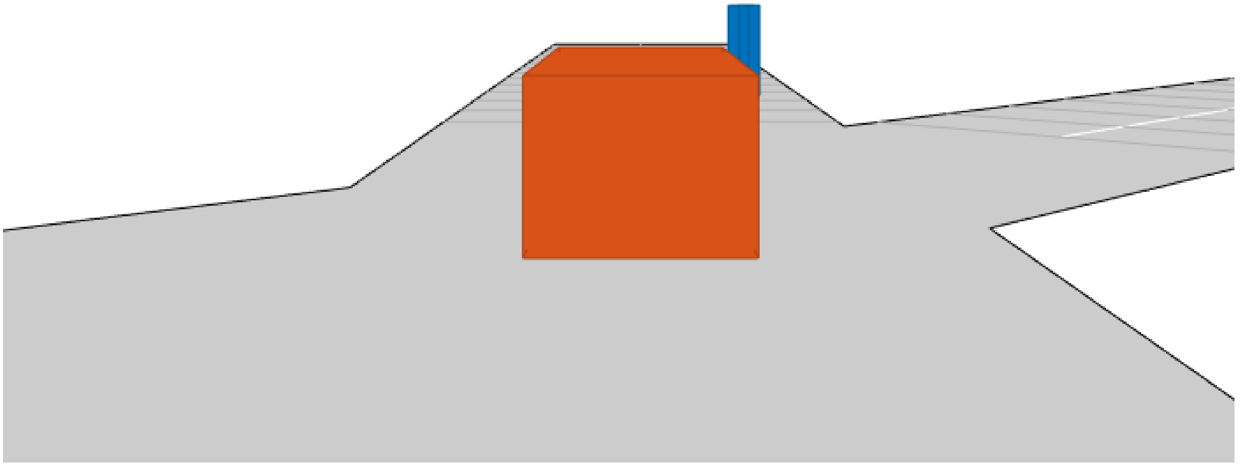
Run the simulation

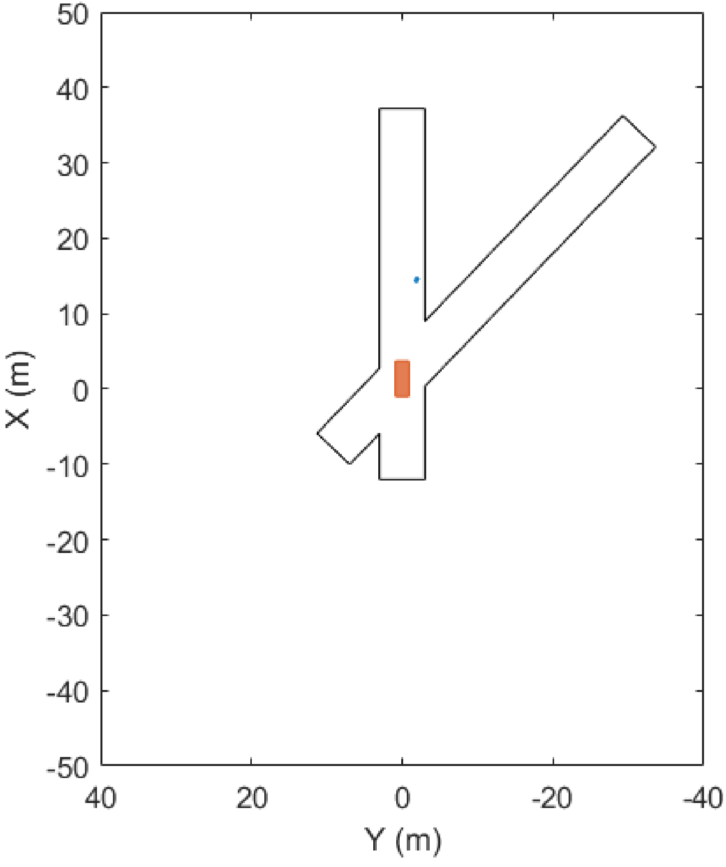
At each simulation step:

- Update and display the chase plot road boundaries and target outline.
- Update the bird's-eye plotter for the road boundary and target outline. The plot perspective is always with respect to the ego actor.

```
while advance(s)
    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
```





Algorithms

How to specify motion in a driving scenario

There are two ways that you can manage an actor's motion in a driving scenario.

- When an actor's motion is defined using the `trajectory` method, the actor pose parameters (position, velocity, yaw, pitch, roll, and angular velocity) are determined by the trajectory waypoints and speed arguments. Because the actor follows a trajectory, the motion is completely defined by speed, not velocity, because the direction of motion is determined by the trajectory. The actor moves along the trajectory each time the `advance` method is called. You can manually set any pose property at any time during a simulation, but these properties are overwritten with updated values at the next call to `advance`.

- When the actor's motion is not defined by a trajectory, you must manage the actor motion manually. Setting the velocity or angular velocity properties will not automatically move the actor in successive calls to `advance`. You must update the position, velocity and the other pose parameters at each simulation time step using your own motion model.

See Also

Apps
Driving Scenario Designer

System Objects
`multiObjectTracker` | `radarDetectionGenerator` | `visionDetectionGenerator`

Topics

"Define Road Layouts"
"Create Actor and Vehicle Trajectories"
"Sensor Fusion Using Synthetic Radar and Vision Data"
"Model Radar Sensor Detections"
"Model Vision Sensor Detections"
"Radar Signal Simulation and Processing for Automated Driving"
"Coordinate Systems in Automated Driving System Toolbox"

Introduced in R2017a

actor

Class: drivingScenario

Create an actor within driving scenario

Syntax

```
ac = actor(sc)
ac = actor(sc,Name,Value)
```

Description

`ac = actor(sc)` adds an `Actor` object, `ac`, to the driving scenario, `sc`. The method creates an actor with default property values. Actors are cuboid (box shaped) generic objects. Each actor is assigned a unique integer ID specified in the `ActorID` field of the `Actor` class.

`ac = actor(sc,Name,Value)` adds an actor with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Any unspecified properties take default values.

Input Arguments

`sc` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Example: `drivingScenario`

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`.

`Length` — Length of actor

`4.7` (default) | positive scalar

Length of actor, specified as a positive scalar. Units are in meters.

Example: `5.5`

Data Types: `double`

`Width` — Width of actor

`1.8` (default) | positive scalar

Width of actor, specified as a positive scalar. Units are in meters.

Example: `3.0`

Data Types: `double`

`Height` — Height of actor

`1.4` (default) | positive scalar

Height of actor, specified as a positive scalar. Units are in meters.

Example: `2.1`

Data Types: `double`

`Position` — Position of actor center

`[0 0 0]` (default) | real-valued three-element vector

Position of the center of an actor, specified as a real-valued three-element vector. The height, *H*, length, *L*, and width, *W*, determine the dimensions of the actor. The center of the actor is the midpoint of its length, *L*/2, and the midpoint of its width, *W*/2, on the bottom of the cuboid. The `Position` property specifies the position of this center. The `Velocity` property specifies the velocity of the center. Units are in meters.

Example: `[10;50;0]`

Data Types: `double`

`Velocity` — Velocity of actor

`[0 0 0]` (default) | real-valued three-element vector

Velocity of actor, specified as a real-valued three-element vector representing the (*x*,*y*,*z*) velocity components of the actor. The `Velocity` property specifies the velocity of the center specified by `Position`. Units are in meters per second.

Example: `[-4;7;10]`

Data Types: `double`

`Roll` — Roll angle of the actor
`0` (default) | scalar

Roll angle of actor, specified as a scalar. Roll is the clockwise angle of rotation of the actor around the *x*-axis. Units are in degrees.

Example: `-10`

Data Types: `double`

`Pitch` — Roll angle of the actor
`0` (default) | scalar

Pitch angle of actor, specified as a scalar. Pitch is the clockwise angle of rotation of the actor around the *y*-axis. Units are in degrees.

Example: `5.8`

Data Types: `double`

`Yaw` — Yaw angle of the actor
`0` (default) | scalar

Yaw angle of actor, specified as a scalar. Yaw is the clockwise angle of rotation of the actor around the *z*-axis. Units are in degrees.

Example: `-0.4`

Data Types: `double`

`AngularVelocity` — Angular rotation velocity of actor
`[0 0 0]` (default) | real-valued three-element row vector

Angular rotation velocity of actor, specified as a real-valued three-element row vector. The vector defines the components of the angular velocity vector in (*x*,*y*,*z*) scenario coordinates. Units are in degrees per second.

`RCSPattern` — Radar cross-section pattern of actor
`[10 10; 10 10]` (default) | real-valued *Q*-by-*P* matrix

Radar cross-section (RCS) pattern of actor, specified as a real-valued *Q*-by-*P* matrix. The radar cross-section pattern is a function of azimuth and elevation. *Q* is the number of elevation angles specified by the `RCSElevationAngles` property. *P* is the number of azimuth angles specified by the `RCSAzimuthAngles` property. Units are in dBsm.

Example: `[5.8 5.9 5.9]`

Data Types: `double`

`RCSAzimuthAngles` — Azimuth angles of radar cross-section pattern
`[-180 180]` (default) | real-valued *P*-element vector

Azimuth angles of the radar cross-section pattern, specified as a real-valued *P*-element vector. Each entry defines the azimuth angle of the corresponding column of the radar cross-section specified by the `RCSPattern` property. Units are in degrees. Azimuth angles lie in the range from –180° to 180°.

Example: `[-90:90]`

Data Types: `double`

`RCSElevationAngles` — Elevation angles of radar cross-section pattern
`[-90 90]` (default) | real-valued *Q*-element vector

Elevation angles of the radar cross-section pattern, specified as a real-valued *Q*-element vector. Each entry defines the elevation angle of the corresponding row of the radar cross-section specified by the `RCSPattern` property. Units are in degrees. Elevation angles lie in the range from –90° to 90°.

Example: `[0:90]`

Data Types: `double`

`ClassID` — Classification identifier
`0` (default) | nonnegative integer

Classification identifier specified as a nonnegative integer. You can define your own actor classification scheme and assign `ClassID`values to actors according to the scheme. The value of `0` is reserved for an object of unknown or unassigned class.

Example: `5`

Data Types: `double`

Output Arguments

ac — Scenario actor
`Actor` object

Scenario actor, returned as an `Actor` object.

Methods

path	(To be removed) Create actor or vehicle path in driving scenario
chasePlot	Egocentric projective perspective plot
roadBoundaries	Show road boundaries
targetOutlines	Outlines of targets viewed by actor
targetPoses	Target positions and orientations seen from an actor
trajectory	Create actor or vehicle trajectory in driving scenario

Introduced in R2017a

actorPoses

Class: drivingScenario

Positions, velocities, and orientations of actors in driving scenario

Syntax

`poses = actorPoses(sc)`

Description

`poses = actorPoses(sc)` returns the current poses (positions, velocities, and orientations) for all actors in the driving scenario, `sc`. Actors include `Actor` class objects and `Vehicle` class objects. Poses components are relative to scenario coordinates.

Input Arguments

sc — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Example: `drivingScenario`

Output Arguments

poses — Actor poses in scenario coordinates
structures | array of structures

Actor poses in scenario coordinates, returned as a structure or array of structures. Poses are the positions and orientation of actors and their rates of change. The pose structure contains these fields:

Field	Description
`ActorID`	Scenario-defined actor identifier
`Position`	Position of actor, specified as a real-valued 1-by-3 vector. Units are in meters.
`Velocity`	Velocity of actor, specified as a real-valued 1-by-3 vector. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a scalar. Units are in degrees.
`AngularVelocity`	Angular velocity of actor, specified as a real-valued 1-by-3 vector. Units are in degrees per second.

See `Actor` and `Vehicle` for full definitions of the structure fields.

Introduced in R2017a

actorProfiles

Class: drivingScenario

Physical and radar properties of actors in driving scenario

Syntax

```
profiles = actorProfiles(sc)
```

Description

`profiles = actorProfiles(sc)` returns the physical and radar properties, `profiles`, for all actors in a driving scenario, `sc`. Actors include `Actor` and `Vehicle` class objects.

Input Arguments

`sc` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Example: `drivingScenario`

Output Arguments

`profiles` — Actor profiles
array of structures

Actor profiles, returned as an array of structures. Each profile contains the physical and radar properties of an actor. The structure contains these fields.

Field	Description
`ActorID`	Scenario-defined actor identifier
`ClassID`	Classification identifier
`Length`	Length of actor
`Width`	Width of actor
`Height`	Height of actor
`OriginOffset`	Displacement from the bottom center of the actor that defines the rotational center of the actor. For vehicles, the center is the point on the ground beneath the center of the rear axle
`RCSPattern`	Radar cross-section pattern matrix.
`RCSAzimuthAngle`	Azimuth angles corresponding to rows of `RCSPattern`
`RCSElevationAngle`	Elevation angles corresponding to columns of `RCSPattern`

See `Actor` and `Vehicle` for full definitions of the structure fields.

Introduced in R2017a

advance

Class: drivingScenario

Advance driving scenario simulation by one time step

Syntax

`isrunning = advance(sc)`

Description

`isrunning = advance(sc)` advances the driving scenario simulation, `sc`, by one time step. To specify the step time, use the `SampleTime` property. The method returns the status, `isrunning`, of the simulation.

Input Arguments

`sc` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Example: `drivingScenario`

Output Arguments

`isrunning` — Run-state of simulation
`0` | `1`

The run-state of the simulation, returned as `0` or `1`. If `isrunning` is `1`, the simulation is running. If `isrunning` `0`, the simulation has stopped. 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 `sc`.
- Any actor or vehicle reaches the end of its assigned trajectory. The assigned trajectory is specified by the most recent call to the `trajectory` method.

The `advance` method 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.

Introduced in R2017a

vehicle

Class: drivingScenario

Create a vehicle within driving scenario

Syntax

```
vc = vehicle(sc)
vc = vehicle(sc,Name,Value)
```

Description

`vc = vehicle(sc)` adds a driving scenario vehicle `Vehicle` object, `vc`, to the driving scenario, `sc`. The method creates a vehicle with default property values. Vehicles are cuboid (box shaped) objects. A vehicle is an actor with additional properties.

`vc = vehicle(sc,Name,Value)` adds a vehicle with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Any unspecified properties take default values.

Input Arguments

`sc` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Example: `drivingScenario`

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`.

`Length` — Length of vehicle
`4.7` (default) | positive scalar

Length of vehicle, specified as a positive scalar. Units are in meters.

Example: `5.5`

Data Types: `double`

`Width` — Width of vehicle
`1.8` (default) | positive scalar

Width of vehicle, specified as a positive scalar. Units are in meters.

Example: `2.0`

Data Types: `double`

`Height` — Height of vehicle
`1.4` (default) | positive scalar

Height of vehicle, specified as a positive scalar. Units are in meters.

Example: `2.1`

Data Types: `double`

`FrontOverhang` — Front overhang of vehicle
`0.9` (default) | nonnegative scalar

Front overhang of vehicle, specified as a nonnegative scalar. The front overhang is the distance that the vehicle extends forward beyond the front axle. Units are in meters.

Data Types: `double`

`RearOverhang` — Rear overhang of vehicle
`1.0` (default) | nonnegative scalar

Rear overhang of vehicle, specified as a nonnegative scalar. The rear overhang is the distance that the vehicle extends rearward beyond the rear axle. Units are in meters.

Data Types: `double`

`Wheelbase` — Distance between axles
`2.8` (default) | positive scalar

The distance between axles, specified as a positive scalar. Units are in meters.

Data Types: `double`

`Position` — Position of vehicle center
`[0 0 0]` (default) | real-valued three-element vector

Position of the rotational center of a vehicle, specified as a real-valued three-element 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 a distance equal to the sum of the wheelbase and forward overhang. The `Position` property specifies the position of this center. The `Velocity` property specifies the velocity of the center. Units are in meters.

Example: `[10;50;0]`

Data Types: `double`

`Velocity` — Velocity of vehicle
`[0 0 0]` (default) | real-valued three-element vector

Velocity of vehicle, specified as a real-valued three-element vector representing the (*x*,*y*,*z*) velocity components of the vehicle. The `Velocity` property specifies the velocity of the center specified by `Position`. Units are in meters per second.

Example: `[-4;7;10]`

Data Types: `double`

`Roll` — Roll angle of = vehicle
`0` (default) | scalar

Roll angle of vehicle, specified as a scalar. Roll is the clockwise angle of rotation of the vehicle around the *x*-axis. Units are in degrees.

Example: `-1`

Data Types: `double`

`Pitch` — Pitch angle of vehicle
`0` (default) | scalar

Pitch angle of vehicle, specified as a scalar. Pitch is the clockwise angle of rotation of the vehicle around the *y*-axis. Units are in degrees.

Example: `5.8`

Data Types: `double`

`Yaw` — Yaw angle of vehicle
`0` (default) | scalar

Yaw angle of vehicle, specified as a scalar. Yaw is the clockwise angle of rotation of the vehicle around the *z*-axis. Units are in degrees.

Example: `-0.4`

Data Types: `double`

`AngularVelocity` — Angular rotation velocity of vehicle
`[0 0 0]` (default) | real-valued three-element row vector

Angular rotation velocity of vehicle, specified as a real-valued three-element row vector. The vector defines the components of the angular velocity vector in (*x*,*y*,*z*) scenario coordinates. Units are in degrees per second.

`RCSPattern` — Radar cross-section pattern of vehicle
`[10 10; 10 10]` (default) | real-valued *Q*-by-*P* matrix

Radar cross-section (RCS) pattern of vehicle, specified as a real-valued *Q*-by-*P* matrix. *Q* is the number of elevation angles specified by the `RCSElevationAngles` property. *P* is the number of azimuth angles specified by the `RCSAzimuthAngles` property. The radar cross-section pattern is a function of azimuth and elevation. Units are in dBsm.

Example: `[5.8 5.9 5.9]`

Data Types: `double`

`RCSAzimuthAngles` — Azimuth angles of radar cross-section pattern
`[-180 180]` (default) | real-valued *P*-length vector

Azimuth angles of radar cross-section pattern, specified as a real-valued *P*-element vector. Azimuth angles define the angle coordinates of the rows of the radar cross-section specified by the `RCSPattern` property. Units are in degrees. Azimuth angles lie from –180° to 180°.

Example: `[-90:90]`

Data Types: `double`

`RCSElevationAngles` — Elevation angles of radar cross-section pattern
`[-90 90]` (default) | real-valued *P*-element vector

Elevation angles of radar cross-section pattern, specified as a real-valued *P*-element vector. Elevation angles define the angle coordinates of the columns of the radar cross-section specified by the `RCSPattern` property. Units are in degrees. Elevation angles lie from –90° to 90°.

Example: `[0:90]`

Data Types: `double`

`ClassID` — Classification identifier
`0` (default) | nonnegative integer

Classification identifier, specified as a nonnegative integer. You can define your own actor classification scheme and assign `ClassID`values to actors according to the scheme. The value of `0` is reserved for an object of unknown or unassigned class.

Example: `5`

Data Types: `double`

Output Arguments

vc — Scenario vehicle
`Vehicle` object

Scenario vehicle, returned as a `Vehicle` object.

Methods

path	(To be removed) Create actor or vehicle path in driving scenario
chasePlot	Egocentric projective perspective plot
roadBoundaries	Show road boundaries
targetOutlines	Outlines of targets viewed by actor
targetPoses	Target positions and orientations seen from an actor
trajectory	Create actor or vehicle trajectory in driving scenario

Introduced in R2017a

plot

Class: drivingScenario

Create driving scenario plot

Syntax

```
plot(sc)
plot(sc,Name,Value)
```

Description

`plot(sc)` creates a 3-D plot with orthonormal perspective, as seen from immediately above the driving scenario, `sc`.

`plot(sc,Name,Value)` specifies one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Any unspecified properties take default values.

Tip To rotate any plot, in the figure window, select **View** > **Camera Toolbar**.

Input Arguments

`sc` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Example: `drivingScenario`

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`.

`Parent` — Axes object
axes object

Axes object in which to draw the plot. If you do not specify `Parent`, a new figure is created.

`Centerline` — Enable display of road center line
`'off'` (default) | `'on'`

Enable the display of the road center line, specified as `'off'` or `'on'`. A road center line follows the middle of the road segment.

Data Types: `char` | `string`

`RoadCenters` — Display road centers
`'off'` (default) | `'on'`

Display road centers, specified as `'off'` or `'on'`. If `'on'`, the road centers used to define the roads are shown in the plot.

Data Types: `char` | `string`

`Waypoints` — Display actor waypoints
`'off'` (default) | `'on'`

Display actor waypoints on plot, specified as `'off'` or `'on'`.

Data Types: `char` | `string`

Introduced in R2017a

road

Class: drivingScenario

Add a road to driving scenario

Syntax

```
road(sc,roadcenters)
road(sc,roadcenters,roadwidth)
road(sc,roadcenters,roadwidth,bankingangle)
road(sc,roadcenters,'Lanes',ls)
```

Description

`road(sc,roadcenters)` adds a road to the driving scenario, `sc`. You specify the road shape using a set of road centers, `roadcenters`, at discrete points.

`road(sc,roadcenters,roadwidth)` also specifies the width of the road, `roadwidth`.

`road(sc,roadcenters,roadwidth,bankingangle)` also specifies the banking angle of the road, `bankingangle`.

`road(sc,roadcenters,'Lanes',ls)` specifies the road using a `lanespec` object. Do not specify `roadwidth` when using this syntax. `bankingangle` is an optional argument.

Input Arguments

`sc` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Example: `sc = drivingScenario`

`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. When `roadcenters` is an *N*-by-3 matrix, each row specifies the *x*, *y*, and *z*-coordinates of a road center. If `roadcenters` is an *N*-by-2 matrix, the *z*-coordinate is zero. 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 scalar

Width of road, specified as a positive scalar. The width is constant along the entire road. Units are in meters.

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.

Data Types: `double`

`'Lanes'` — Lane specification
lane specification object

Lane specification, specified as a name,value pair consisting of `'Lanes'` and a lane specification object. For description of lane specifications, see `lanespec`. For a description of lane markings, see `laneMarking`.

Data Types: `double`

Algorithms

This method creates a road for an actor to follow in a 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 method 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 non-closed 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

`drivingScenario` | `laneMarking` | `lanespec`

Introduced in R2017a

roadNetwork

Class: drivingScenario

Add road network to driving scenario

Syntax

`roadNetwork(scenario,'OpenDRIVE',filePath)`

Description

`roadNetwork(scenario,'OpenDRIVE',filePath)` imports roads and lanes from an OpenDRIVE road network file into a driving scenario. This method supports OpenDRIVE format specification version 1.4H [1].

Input Arguments

`scenario` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object. `scenario` must contain no roads and no other OpenDRIVE road network.

`filePath` — Path to valid OpenDRIVE file
character vector | string scalar

Path to a valid OpenDRIVE file of type `.xml` or `.xodr`, specified as a character vector or string scalar.

Example: `'OpenDRIVE','C:\Desktop\myRoadNetwork.xodr'`

Examples

Import OpenDRIVE Road Network into Driving Scenario

Create an empty driving scenario.

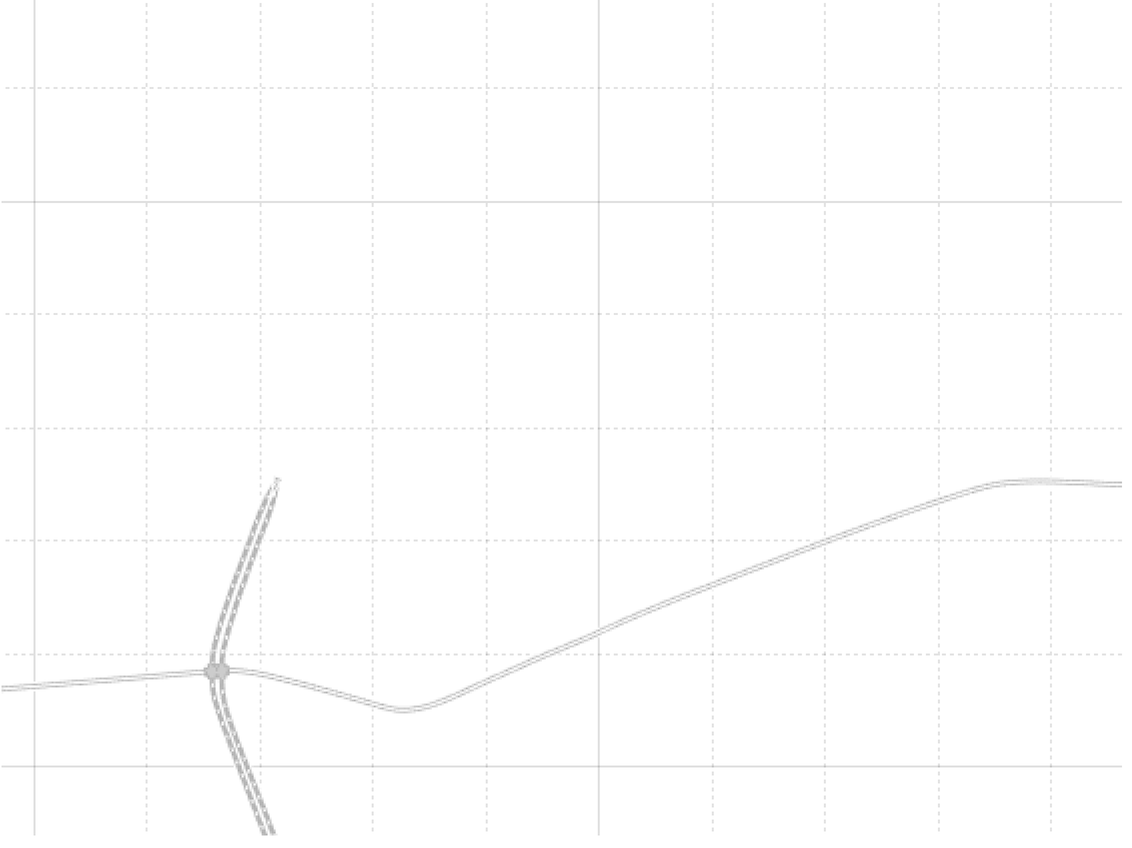
```
scenario = drivingScenario;
```

Import an OpenDRIVE road network into the scenario.

```
filePath = fullfile(matlabroot,'examples','driving','intersection.xodr');
roadNetwork(scenario,'OpenDRIVE',filePath);
```

Plot the scenario and zoom in on the road network.

```
plot(scenario)
zoom(5)
```



Limitations

- You can import only lanes and roads. The import of road objects and traffic signals is not supported.
- OpenDRIVE files containing large road networks can take up to several minutes to load. 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 method sets the lane width to 4 meters throughout.
- Roads with multiple lane marking styles are not supported. The method applies the first found marking style to all lanes in the road. For example, if a road has `Dashed` and `Solid` lane markings, the method applies `Dashed` lane markings throughout.
- Lane marking styles `Bott Dots`, `Curbs`, and `Grass` are not supported. If imported roads have these lane marking styles, the method sets their lane markings to the default style, as determined by the number of lanes in the road.

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

`actor` | `drivingScenario` | `trajectory` | `vehicle`

External Websites

opendrive.org

Introduced in R2018b

record

Class: drivingScenario

Run driving scenario and record actor states

Syntax

`rec = record(sc)`

Description

`rec = record(sc)` returns a record, `rec`, of the evolution of the simulation of the driving scenario, `sc`.

Input Arguments

`sc` — Driving scenario

`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Example: `drivingScenario`

Output Arguments

`rec` — Record of actor and vehicle states during simulation

M-by-1 vector of structures

A record of actor and vehicle states during the simulation, returned as an *M*-by-1 vector of structures. *M* is the number of time steps in the simulation. Each record corresponds to a simulation time step. The structure has these fields:

`SimulationTime`
`ActorPoses`

The `SimulationTime` field contains the simulation time of the record. `ActorPoses` is an *N*-by-1 vector of structures, where *N* is the number of actors, including vehicles. Each `ActorPoses` structure contains these fields.

Field	Meaning
`ActorID`	Scenario-defined actor identifier
`Position`	Position of actor in scenario coordinates
`Velocity`	Velocity of actor in scenario coordinates
`Roll`	Roll angle of actor
`Pitch`	Pitch angle of actor
`Yaw`	Yaw angle of actor
`AngularVelocity`	Angular velocity of actor

See `Actor` and `Vehicle` for full definitions of the structure fields for each actor.

Data Types: `struct`

Introduced in R2017a

restart

Class: drivingScenario

Restart driving scenario simulation from beginning

Syntax

```
restart(sc)
```

Description

`restart(sc)` restarts the simulation of the driving scenario, `sc`, from the beginning. The method sets the `SimulationTime` property of the driving scenario to zero.

Input Arguments

`sc` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Example: `drivingScenario`

Introduced in R2017a

updatePlots

Class: drivingScenario

Update driving scenario plots

Syntax

`updatePlots(sc)`

Description

`updatePlots(sc)` updates all existing plots for the driving scenario, `sc`. Use this method after you update any actor properties and want to refresh the plot. This method does not advance the simulation.

Input Arguments

`sc` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Example: `drivingScenario`

Introduced in R2017a

Actor class

Actor belonging in driving scenario

Description

The `Actor` class defines an actor object belonging to a driving scenario. Actors are cuboid (box-shaped) objects.

Properties

`ActorID` — Scenario-defined actor identifier

`1` (default) | positive integer

This property is read-only.

Scenario-defined actor identifier specified as a positive integer. The scenario automatically assigns `ActorID` values to actors, including vehicles.

Example: `1`

Data Types: `double`

`ClassID` — Classification identifier

`0` (default) | nonnegative integer

Classification identifier specified as a nonnegative integer. You can define your own actor classification scheme and assign `ClassID`values to actors according to the scheme. The value of `0` is reserved for an object of unknown or unassigned class.

Example: `5`

Data Types: `double`

`Position` — Position of actor center

`[0 0 0]` (default) | real-valued three-element vector

Position of the center of an actor, specified as a real-valued three-element vector. The height, *H*, length, *L*, and width, *W*, determine the dimensions of the actor. The center of the actor is the midpoint of its length, *L*/2, and the midpoint of its width, *W*/2, on the bottom of the cuboid. The `Position` property specifies the position of this center. The `Velocity` property specifies the velocity of the center. Units are in meters.

Example: `[10;50;0]`

Data Types: `double`

`Velocity` — Velocity of actor
`[0 0 0]` (default) | real-valued three-element vector

Velocity of actor, specified as a real-valued three-element vector representing the (*x*,*y*,*z*) velocity components of the actor. The `Velocity` property specifies the velocity of the actor center specified by `Position`. Units are in meters per second.

Example: `[-4;7;10]`

Data Types: `double`

`Yaw` — Yaw angle of the actor
`0` (default) | scalar

Yaw angle of actor, specified as a scalar. Yaw is the clockwise angle of rotation of the actor around the *z*-axis. Units are in degrees.

Example: `-0.4`

Data Types: `double`

`Pitch` — Roll angle of the actor
`0` (default) | scalar

Pitch angle of actor, specified as a scalar. Pitch is the clockwise angle of rotation of the actor around the *y*-axis. Units are in degrees.

Example: `5.8`

Data Types: `double`

`Roll` — Roll angle of the actor
`0` (default) | scalar

Roll angle of actor, specified as a scalar. Roll is the clockwise angle of rotation of the actor around the *x*-axis. Units are in degrees.

Example: `-10`

Data Types: double

AngularVelocity — Angular rotation velocity of actor
[0 0 0] (default) | real-valued three-element row vector

Angular rotation velocity of actor, specified as a real-valued three-element row vector. The vector defines the components of the angular velocity vector in (x,y,z) scenario coordinates. Units are in degrees per second.

Length — Length of actor
4.7 (default) | positive scalar

Length of actor, specified as a positive scalar. Units are in meters.

Example: 5.5

Data Types: double

Width — Width of actor
1.8 (default) | positive scalar

Width of actor, specified as a positive scalar. Units are in meters.

Example: 3.0

Data Types: double

Height — Height of actor
1.4 (default) | positive scalar

Height of actor, specified as a positive scalar. Units are meters.

Example: 2.1

Data Types: double

RCSPattern — Radar cross-section pattern of actor
[10 10; 10 10] (default) | real-valued *Q*-by-*P* matrix

Radar cross-section (RCS) pattern of actor, specified as a real-valued *Q*-by-*P* matrix. The radar cross-section pattern is a function of azimuth and elevation. *Q* is the number of elevation angles specified by the RCSElevationAngles property. *P* is the number of azimuth angles specified by the RCSAzimuthAngles property. Units are in dBsm.

Example: 5.8

Data Types: `double`

`RCSAzimuthAngles` — Azimuth angles of radar cross-section pattern
`[-180 180]` (default) | real-valued *P*-length vector

Azimuth angles of the radar cross-section pattern, specified as a real-valued *P*-element vector. Each entry defines the azimuth angle of the corresponding column of the radar cross-section specified by the `RCSPattern` property. Units are in degrees. Azimuth angles lie in the range from –180° to 180°.

Example: `[-90:90]`

Data Types: `double`

`RCSElevationAngles` — Elevation angles of radar cross-section pattern
`[-90 90]` (default) | real-valued *Q*-length vector

Elevation angles of the radar cross-section pattern, specified as a real-valued *Q*-element vector. Each entry defines the elevation angle of the corresponding row of the radar cross-section specified by the `RCSPattern` property. Units are in degrees. Elevation angles lie in the range from –90° to 90°.

Example: `[0:90]`

Data Types: `double`

Introduced in R2017a

Vehicle class

Vehicle class for use in a driving scenario

Description

The `Vehicle` class defines a vehicle object belonging to a driving scenario. Vehicles are cuboid (box-shaped) objects.

Properties

`ActorID` — Scenario-defined vehicle identifier
positive integer

This property is read-only.

Scenario-defined vehicle identifier, specified as a positive integer. The scenario automatically assigns `ActorID` values to vehicles.

Example: `1`

Data Types: `double`

`ClassID` — Classification identifier
`0` (default) | nonnegative integer

Classification identifier, specified as a nonnegative integer. You can define your own actor classification scheme and assign `ClassID`values to actors according to the scheme. The value of `0` is reserved for an object of unknown or unassigned class.

Example: `5`

Data Types: `double`

`Position` — Position of vehicle center
`[0 0 0]` (default) | real-valued three-element vector

Position of the rotational center of a vehicle, specified as a real-valued three-element 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 a distance equal to the sum of the wheelbase and forward overhang. The `Position` property specifies the position of this center. The `Velocity` property specifies the velocity of the center. Units are in meters.

Example: `[10;50;0]`

Data Types: `double`

`Velocity` — Velocity of vehicle

`[0 0 0]` (default) | real-valued three-element vector

Velocity of vehicle, specified as a real-valued three-element vector representing the (*x*,*y*,*z*) velocity components of the vehicle. The `Velocity` property specifies the velocity of the vehicle center specified by `Position`. Units are in meters per second.

Example: `[-4;7;10]`

Data Types: `double`

`Yaw` — Yaw angle of vehicle

`0` (default) | scalar

Yaw angle of vehicle, specified as a scalar. Yaw is the clockwise angle of rotation of the vehicle around the *z*-axis. Units are in degrees.

Example: `-0.4`

Data Types: `double`

`Pitch` — Pitch angle of vehicle

`0` (default) | scalar

Pitch angle of vehicle, specified as a scalar. Pitch is the clockwise angle of rotation of the vehicle around the *y*-axis. Units are in degrees.

Example: `5.8`

Data Types: `double`

`Roll` — Roll angle of = vehicle

`0` (default) | scalar

Roll angle of vehicle, specified as a scalar. Roll is the clockwise angle of rotation of the vehicle around the *x*-axis. Units are in degrees.

Example: `-1`

Data Types: `double`

`AngularVelocity` — Angular rotation velocity of vehicle
`[0 0 0]` (default) | real-valued three-element row vector

Angular rotation velocity of vehicle, specified as a real-valued three-element row vector. The vector defines the components of the angular velocity vector in (x,y,z) scenario coordinates. Units are in degrees per second.

`Length` — Length of vehicle
`4.7` (default) | positive scalar

Length of vehicle, specified as a positive scalar. Units are in meters.

Example: `5.5`

Data Types: `double`

`Width` — Width of vehicle
`1.8` (default) | positive scalar

Width of vehicle, specified as a positive scalar. Units are in meters.

Example: `2.0`

Data Types: `double`

`Height` — Height of vehicle
`1.4` (default) | positive scalar

Height of vehicle, specified as a positive scalar. Units are in meters.

Example: `2.1`

Data Types: `double`

`RCSPattern` — Radar cross-section pattern of vehicle
`[10 10; 10 10]` (default) | real-valued Q-by-P matrix

Radar cross-section (RCS) pattern of vehicle, specified as a real-valued Q-by-P matrix. Q is the number of elevation angles specified by the `RCSElevationAngles` property. P is the number of azimuth angles specified by the `RCSAzimuthAngles` property. The radar cross-section pattern is a function of azimuth and elevation. Units are in dBsm.

Example: [5.8 5.9 5.9]

Data Types: double

RCSAzimuthAngles — Azimuth angles of radar cross-section pattern
[-180 180] (default) | real-valued *P*-length vector

Azimuth angles of radar cross-section pattern, specified as a real-valued *P*-element vector. Azimuth angles define the angle coordinates of the rows of the radar cross-section specified by the RCSPattern property. Units are in degrees. Azimuth angles lie from –180° to 180°.

Example: [-90:90]

Data Types: double

RCSElevationAngles — Elevation angles of radar cross-section pattern
[-90 90] (default) | real-valued *Q*-element vector

Elevation angles of radar cross-section pattern, specified as a real-valued *Q*-element vector. Elevation angles define the angle coordinates of the columns of the radar cross-section specified by the RCSPattern property. Units are in degrees. Elevation angles lie from –90° to 90°.

Example: [0:90]

Data Types: double

FrontOverhang — Front overhang of vehicle
0.9 (default) | nonnegative scalar

The front overhang of a vehicle, specified as a nonnegative scalar. The front overhang is the distance that the vehicle extends beyond the front axle. Units are in meters.

Data Types: double

RearOverhang — Rear overhang of vehicle
1.0 (default) | nonnegative scalar

The rear overhang of a vehicle, specified as a nonnegative scalar. The rear overhang is the distance that the vehicle extends beyond the rear axle. Units are in meters.

Data Types: double

Wheelbase — Distance between axles
2.8 (default) | positive scalar

The distance between axles, specified as a positive scalar. Units are in meters.

Data Types: `double`

Introduced in R2017a

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 — Scenario actor

`Actor` object | `Vehicle` object

Scenario actor, specified as an `Actor` or `Vehicle` object. To create actors, use the `actor` or `vehicle` method.

`waypoints` — Path waypoints

real-valued *N*-by-2matrix | real-valued *N*-by-3 matrix

Path waypoints, specified as a real-valued *N*-by-2 or *N*-by-3 matrix. If you specify the waypoints as an *N*-by-3 matrix, each row of the matrix represents the (x,y,z) coordinates of a waypoint. If you specify the waypoints as an *N*-by-2 matrix, each row represents the (x,y) coordinates of a waypoint. The z-coordinates of the waypoints are zero. All coordinates belong to the scenario coordinate system. Units are in meters.

Example: `[1 0 0; 2 7 7]`

Data Types: `double`

speed — Actor speed
`30.0` | positive scalar | *N*-element vector of nonnegative values

Actor speed, specified as a positive 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, it specifies 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 meters per second.

Example: `[10,8,10,11]`

Algorithms

This method 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 method 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.

See Also

`trajectory`

Introduced in R2017a

trajectory

Create actor or vehicle trajectory in driving scenario

Syntax

`trajectory(ac,waypoints,speed)`

Description

`trajectory(ac,waypoints,speed)` creates a trajectory for an actor or vehicle, `ac`, from a set of waypoints. The actor follows the trajectory at the specified speed, `speed`.

Input Arguments

ac — Scenario actor
`Actor` object | `Vehicle` object

Scenario actor, specified as an `Actor` or `Vehicle` object. To create actors, use the `actor` or `vehicle` method.

`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. If you specify the waypoints as an *N*-by-3 matrix, each row of the matrix represents the (*x*,*y*,*z*) coordinates of a waypoint. If you specify the waypoints as an *N*-by-2 matrix, each row represents the (*x*,*y*) coordinates of a waypoint. The *z*-coordinates of the waypoints are zero. All coordinates belong to the scenario coordinate system. Units are in meters.

Example: `[1 0 0; 2 7 7; 3 8 8]`

Data Types: `double`

speed — Actor speed at waypoints
`30.0` | positive scalar | *N*-element vector of nonnegative values

Actor speed at waypoints, specified as a positive 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, it specifies 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: `[10,8,9]`

Algorithms

This method 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 method 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 non-closed 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.

Introduced in R2018a

chasePlot

Egocentric projective perspective plot

Syntax

```
chasePlot(ac)
chasePlot(ac,Name,Value)
```

Description

`chasePlot(ac)` adds an egocentric projective perspective plot to the scenario. The view is as seen from immediately behind the actor.

`chasePlot(ac,Name,Value)` adds a plot using one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Any unspecified arguments take default values.

Input Arguments

ac — Scenario actor

`Actor` object | `Vehicle` object

Scenario actor, specified as an `Actor` or `Vehicle` object. To create actors, use the `actor` or `vehicle` method.

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('Parent',ax,'Centerline','on','Waypoints','on')`

`Parent` — Axes object
axes object

Axes object in which to draw the plot. If you leave `Parent` unspecified, a new figure is created.

`Centerline` — Paint road center line
`'off'` (default) | `'on'`

Paint road center line on plot, specified as `'off'` or `'on'`. The display of the center line follows normal road conventions. Center lines are not displayed as continuous through an intersection or road split.

Data Types: `char` | `string`

`RoadCenters` — Display road centers
`'off'` (default) | `'on'`

Display road centers, specified as `'off'` or `'on'`. If `'on'`, the road centers used to define the roads are shown in the plot.

Data Types: `char` | `string`

`Waypoints` — Show actor waypoints
`'off'` (default) | `'on'`

Show actor waypoints on plot, specified as `'off'` or `'on'`.

Data Types: `char` | `string`

`ViewHeight` — Height of plot viewpoint
1.5 times actor height (default) | positive scalar

Height of plot viewpoint, specified as a positive scalar. Height is with respect to the bottom of the actor. Units are in meters.

Data Types: `double`

`ViewLocation` — Location of plot viewpoint
2.5 times actor length (default) | 1-by-2 real-valued vector

The location of the plot viewpoint, specified as a 1-by-2 real-valued vector. The viewpoint, `[x y]`, is with respect to the cuboid center in the cuboid coordinate system. The default location of the viewpoint is behind the cuboid center, [2.5*length,0]. Units are in meters.

Data Types: double

ViewRoll — Roll angle of view orientation
0 (default) | scalar

Roll angle of view orientation, specified as a scalar. Units are in degrees.

Data Types: double

ViewPitch — Pitch angle of view orientation
0 (default) | scalar

Pitch angle of view orientation, specified as a scalar. Units are in degrees.

Data Types: double

ViewYaw — Yaw angle of view orientation
0 (default) | scalar

Yaw angle of view orientation, specified as a scalar. Units are in degrees.

Data Types: double

Introduced in R2017a

roadBoundaries

Show road boundaries

Syntax

```
rbdry = roadBoundaries(sc)
rbdry = roadBoundaries(ac)
```

Description

`rbdry = roadBoundaries(sc)` returns the road boundaries, `rbdry`, in a driving scenario, `sc`.

`rbdry = roadBoundaries(ac)` returns the road boundaries followed by the actor, `ac`, in a driving scenario.

Input Arguments

`sc` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Example: `drivingScenario`

`ac` — Scenario actor
`Actor` object | `Vehicle` object

Scenario actor, specified as an `Actor` or `Vehicle` object. To create actors, use the `actor` or `vehicle` method.

Output Arguments

`rbdry` — Road boundaries
cell array

Road boundaries, returned as a cell array. Each cell of the array contains a real-valued *N*-by-3 matrix. Each row of the matrix corresponds to the (*x*,*y*,*z*) coordinates of a vertex of the road boundary.

When the input argument is a driving scenario, the road coordinates are with respect to the scenario coordinate system. When the input argument is an actor, the road coordinates are with respect to the actor coordinate system.

Data Types: `double`

Introduced in R2017a

targetPoses

Target positions and orientations seen from an actor

Syntax

```
poses = targetPoses(ac)
```

Description

`poses = targetPoses(ac)` returns the poses of all targets in a scenario with respect to the ego actor `ac` (see "Ego and target actors" on page 4-421). Targets include vehicles. Pose defines the position, velocity, and orientation of a target with respect to the ego coordinate system belonging to the actor. Pose also includes rates of change of position and orientation. The actor must be previously added to the driving scenario via an actor or vehicle method. A target is an actor located with respect to the coordinate system of another actor.

Input Arguments

ac — Scenario actor
`Actor` object | `Vehicle` object

Scenario actor, specified as an `Actor` or `Vehicle` object. To create actors, use the `actor` or `vehicle` method.

Output Arguments

poses — Scenario target poses
structure | array of structures

Scenario target poses, returned as a structure or an array of structures. The pose of the input ego actor, `ac`, is not included. Pose consists of the position, velocity, and orientation of a target and their rates of change. The returned structure has these fields:

Field	Description
`ActorID`	Scenario-defined actor identifier
`Position`	Position of actor, specified as a real-valued 1-by-3 vector. Units are in meters.
`Velocity`	Velocity of actor, specified as a real-valued 1-by-3 vector. Units are in meters per second.
`Roll`	Roll angle of actor, specified as a scalar. Units are in degrees.
`Pitch`	Pitch angle of actor, specified as a scalar. Units are in degrees.
`Yaw`	Yaw angle of actor, specified as a scalar. Units are in degrees.
`AngularVelocity`	Angular velocity of actor, specified as a real-valued 1-by-3 vector. Units are in degrees per second.

The values of the `Position`, `Velocity`, `Roll`, `Pitch`, `Yaw`, and `AngularVelocity` fields are with respect to the coordinate system of the input actor, `ac`. See `Actor` and `Vehicle` for full definitions of the structure fields.

Definitions

Ego and target actors

In a driving scenario, you can specify one actor as the observer of all other actors, much as the driver of a car observes all other cars. The observer actor is called the *ego actor*. From the perspective of the ego actor, all other actors are the observed actors and are called *target actors* or *targets*. Ego coordinates are coordinates centered and oriented with reference to the ego actor. Driving scenario coordinates are world or global coordinates.

See Also

`birdsEyePlot` | `targetOutlines`

Introduced in R2017a

targetOutlines

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 belonging to a driving scenario as viewed from a designated ego actor, `ac` (see “Ego and target actors” on page 4-430). A target outline is the projection of the target actor cuboid into the *x*-*y* plane of the local coordinate system of the ego actor. Target outline parameters are the `position`, `yaw`, `length`, `width`, `originOffset`, and `color` output arguments. All actors must have been previously added to the driving scenario using the `actor` or `vehicle` methods of the `drivingScenario` class.

You can use the returned outlines as input arguments to the outline plotter in `birdsEyePlot`. Then, call `outlinePlotter` to create a plotter object and use `plotOutline` 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.

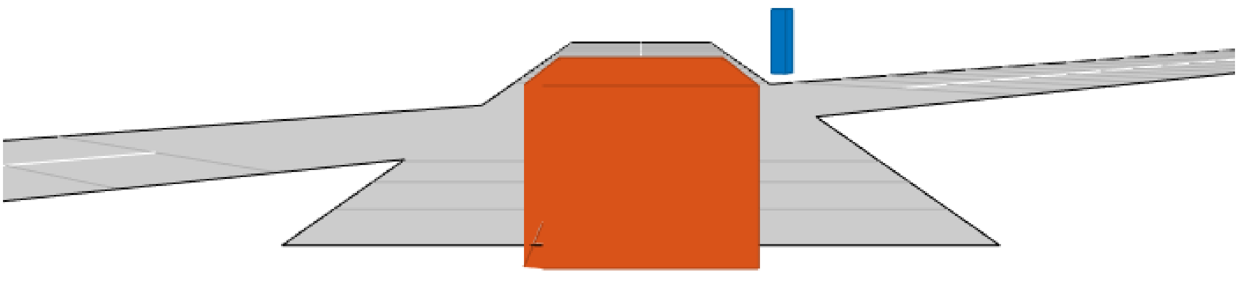
Set up a driving scenario with a vehicle and a pedestrian

Set up a driving scenario consisting of two intersecting straight roads. Construct one straight road segment to be 45 m long. The second straight road is 32 meters long and intersects the first road. A car travelling at 12.0 m/s along the first road approaches a running pedestrian crossing the intersection moving at 2.0 m/s.

```
s = drivingScenario('SampleTime',0.1,'StopTime',1);
road(s,[-10 0 0; 45 -20 0]);
road(s,[-10 -10 0; 35 10 0]);
ped = actor(s,'Length',0.4,'Width',0.6,'Height',1.7);
car = vehicle(s);
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 egocentric chase plot for the vehicle

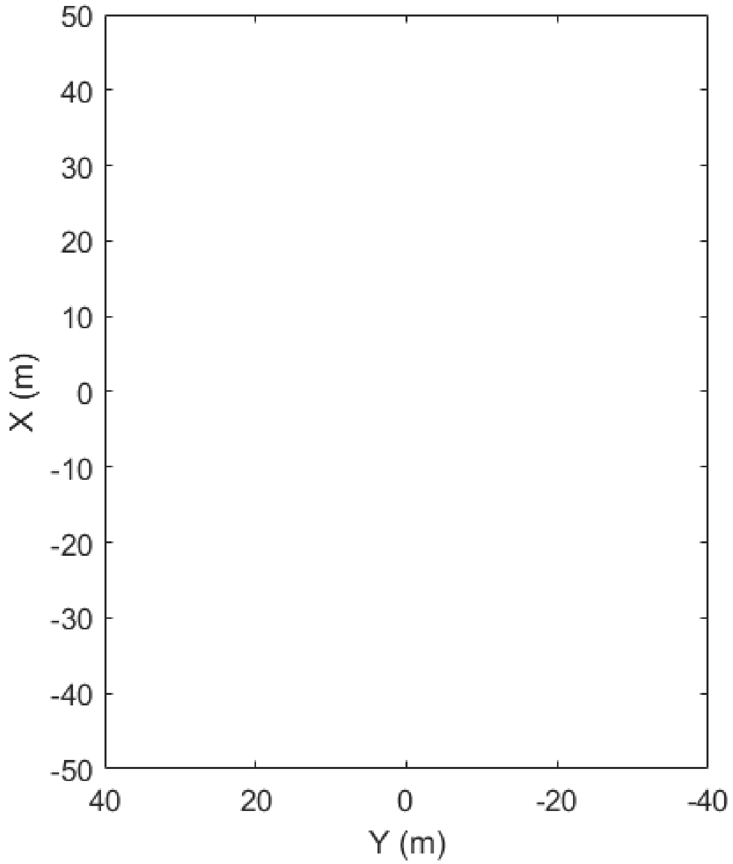
```
chasePlot(car,'Centerline','on')
```



Create a bird's-eye plot of road boundaries and actors

Create an empty bird's-eye plot and add an outline plotter and lane boundary plotter.

```
bepPlot = birdsEyePlot('XLim',[-50 50],'YLim',[-40 40]);
outlineplotter = outlinePlotter(bepPlot);
laneplotter = laneBoundaryPlotter(bepPlot);
legend('off')
```



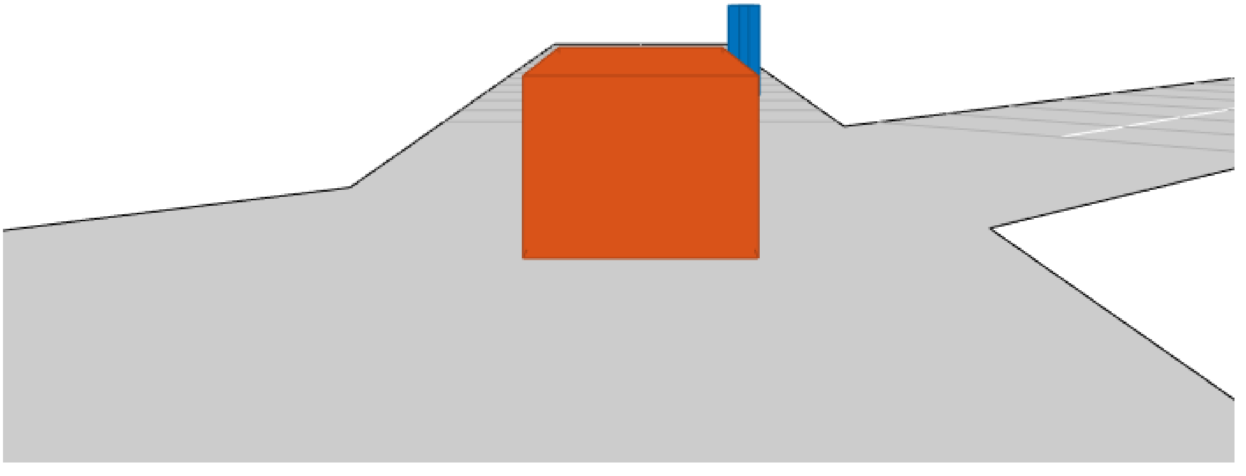
Run the simulation

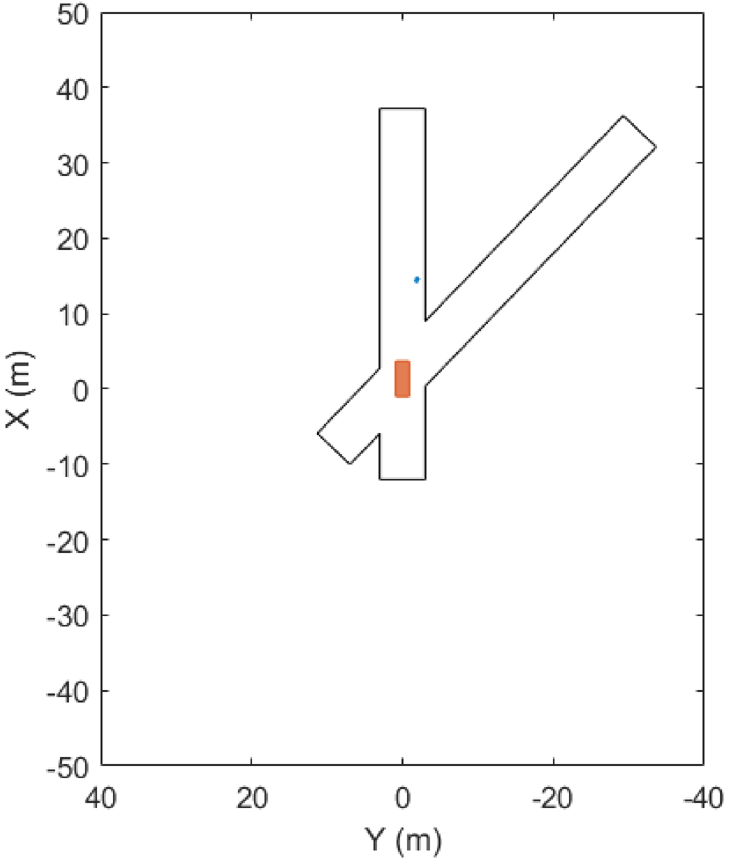
At each simulation step:

- Update and display the chase plot road boundaries and target outline.
- Update the bird's-eye plotter for the road boundary and target outline. The plot perspective is always with respect to the ego actor.

```
while advance(s)
    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
```





Input Arguments

ac — Scenario actor
`Actor` object | `Vehicle` object

Scenario actor, specified as an `Actor` or `Vehicle` object. To create actors, use the `actor` or `vehicle` method.

Output Arguments

Rotational center of rectangle

`position` — Rotational center of rectangle
real-valued *N*-by-2 matrix

Rotational center of rectangle, returned as a real-valued *N*-by-2 matrix. *N* is the number of target actors. Each row contains the *x* and *y* coordinates of the rotational center of the target outline. Units are in meters.

Data Types: `double`

`yaw` — Yaw angle of target
real-valued *N*-element vector

Yaw angle of target about the rotational center, returned as a real-valued *N*-element vector. *N* is the number of target actors. Each element contains the yaw angle of each target. Yaw angles are measured in the counterclockwise direction as seen from above. Units are in degrees.

Data Types: `double`

`length` — Length of rectangular outline of target
positive, real-valued *N*-element vector

Length of rectangular outline of target, returned as a real-valued *N*-element vector. *N* is the number of target actors. Units are in meters.

Data Types: `double`

`width` — Width of rectangular outline of target
positive, real-valued *N*-element vector

Width of rectangular outline of target, returned as a real-valued *N*-element vector. *N* is the number of target actors. Units are in meters.

Data Types: `double`

`originOffset` — Offset of rotational center from geometric center
real-valued *N*-by-2 matrix

Offset of target rotational center from geometric center, returned as a real-valued *N*-by-2 matrix. *N* is the number of target actors. Each row defines a 2D offset vector from the geometric center of the rectangle to the rotational center of the rectangle. Vehicles typically define this offset so that the rotational center rests directly beneath the rear axle of the vehicle. Units are in meters.

Data Types: `double`

`color` — RGB representation of target colors
positive, 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.

Data Types: `double`

Definitions

Ego and target actors

In a driving scenario, you can specify one actor as the observer of all other actors, much as the driver of a car observes all other cars. The observer actor is called the *ego actor*. From the perspective of the ego actor, all other actors are the observed actors and are called *target actors* or *targets*. Ego coordinates are coordinates centered and oriented with reference to the ego actor. Driving scenario coordinates are world or global coordinates.

See Also

`birdsEyePlot` | `targetOutlines` | `targetPoses`

Introduced in R2017a

radarDetectionGenerator System object

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`.

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? (MATLAB).

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 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 (MATLAB).

`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 scalar

Required time interval between sensor updates, specified as a positive 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 scalar

Radar sensor height above the ground plane, specified as a positive 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) | scalar

Yaw angle of radar sensor, specified as a 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) | scalar

Pitch angle of sensor, specified as a 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) | scalar

Roll angle of the radar sensor, specified as a 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 scalar

Maximum detection range, specified as a positive 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 scalar less than or equal to 1

Probability of detecting a target, specified as a positive 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 scalar

False alarm rate within a radar resolution cell, specified as a positive 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 scalar

Reference range for a given probability of detection, specified as a positive 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 scalar

Reference radar cross-section (RCS) for given probability of detection, specified as a 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
scalar

This property is read-only.

Radar loop gain, specified as a 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 scalar

Azimuth resolution of the radar, specified as a positive 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 scalar

Elevation resolution of the radar, specified as a positive 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 scalar

Range resolution of the radar, specified as a positive 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 scalar

Range rate resolution of the radar, specified as a positive 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 scalar

Azimuth bias fraction of the radar, specified as a nonnegative 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 scalar

Elevation bias fraction of the radar, specified as a nonnegative 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 scalar

Range bias fraction of the radar, specified as a nonnegative 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 scalar

Range rate bias fraction of the radar, specified as a nonnegative 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 — Physical characteristics of actors
structure | array of structures

Physical characteristics of actors, specified as structure or an array of structures. Each structure defines the physical characteristics, or profile, 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.

You can generate an array of structures for your driving scenario by using the actorProfiles method that acts on a drivingScenario object. This table shows the valid fields of the structure. When you do not specify a field, the fields are set to their default values.

Valid Actor Profile Fields	Description
ActorID	Scenario-defined actor identifier.
ClassID	User-defined classification identifier.
Length	Length of cuboid.
Width	Width of cuboid.
Height	Height of cuboid.
OriginOffset	Rotational center of the actor, defined as a displacement 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.
RCSPattern	Radar cross-section pattern matrix.
RCSAzimuthAngle	Azimuth angles corresponding to rows of RCSPattern.
RCSElevationAngle	Elevation angles corresponding to columns of RCSPattern.

For definitions of the structure fields and their default values, see the `Actor` and `Vehicle` classes.

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 this structure using the `targetPoses` method of an actor or vehicle. You can also create such a structure manually. The table shows the required fields of the structure:

Actor Fields	Description
ActorID	Unique actor identifier, specified as a scalar positive integer.
Position	Actor position vector, specified as real-valued 1-by-3 vector. Units are in meters.
Velocity	Actor velocity vector, specified as real-valued 1-by-3 vector. If velocity is not specified, the default value is [0 0 0]. Units are in meters per second.
Speed	Speed of actor, specified as a real scalar. When specified, the actor velocity is aligned with the *x*-axis of the actor in the ego actor coordinate system. You cannot specify both Speed and Velocity. Units are in meters per second.
Roll	Roll angle of actor, specified as a real-valued scalar. If roll is not specified, the default value is 0. Units are in degrees.
Pitch	Pitch angle of actor, specified as a real-valued scalar. If pitch is not specified, the default value is 0. Units are in degrees.
Yaw	Yaw angle of actor, specified as a real-valued scalar. If yaw is not specified, the default value is 0. Units are in degrees.

The values of the Position, Velocity, Speed, Roll, Pitch, and Yaw fields are defined with respect to the ego coordinate system. For definitions of the structure fields, see Actor and Vehicle.

time — Current simulation time

nonnegative scalar

Current simulation time, specified as a nonnegative 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
`MeasurementParameters`	Parameters used by initialization functions of nonlinear Kalman tracking filters
`ObjectAttributes`	Additional information passed to tracker

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'
'Sensor Cartesian'</td><td>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 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, ...
    'ConfirmationParameters',[3 4],'NumCoastingUpdates',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
```

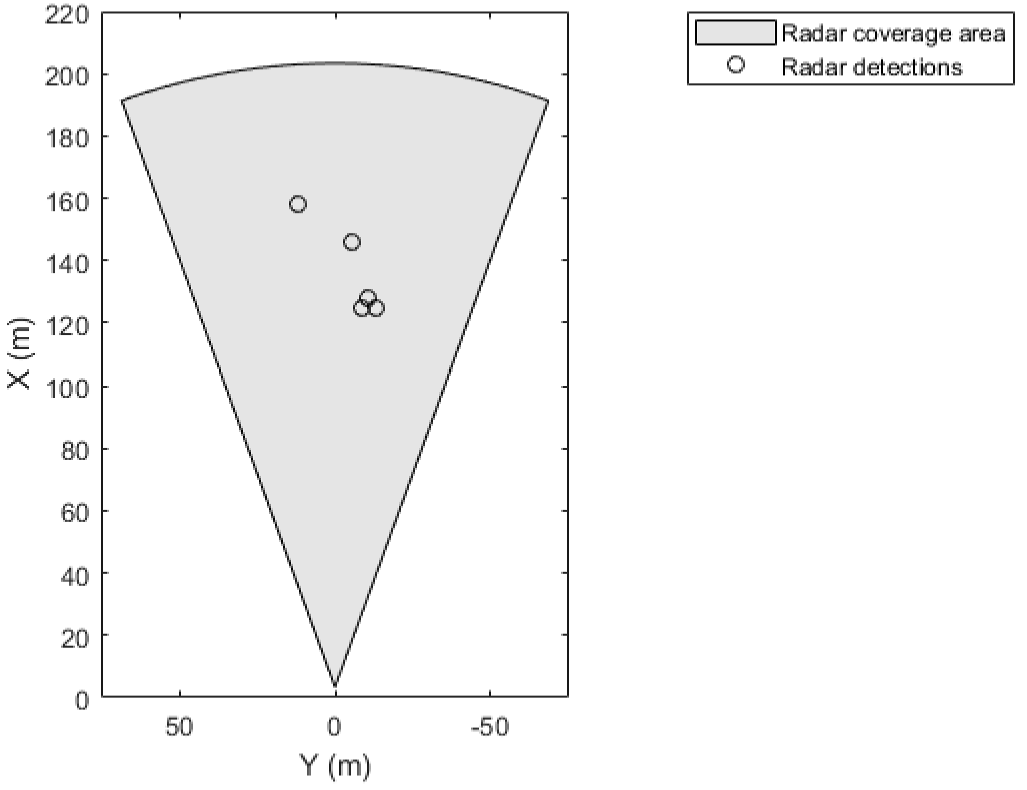
```
    dets = radar([car1 car2 car3],simTime);
    [confirmedTracks,tentativeTracks,allTracks] = updateTracks(tracker,dets,simTime);
```

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.

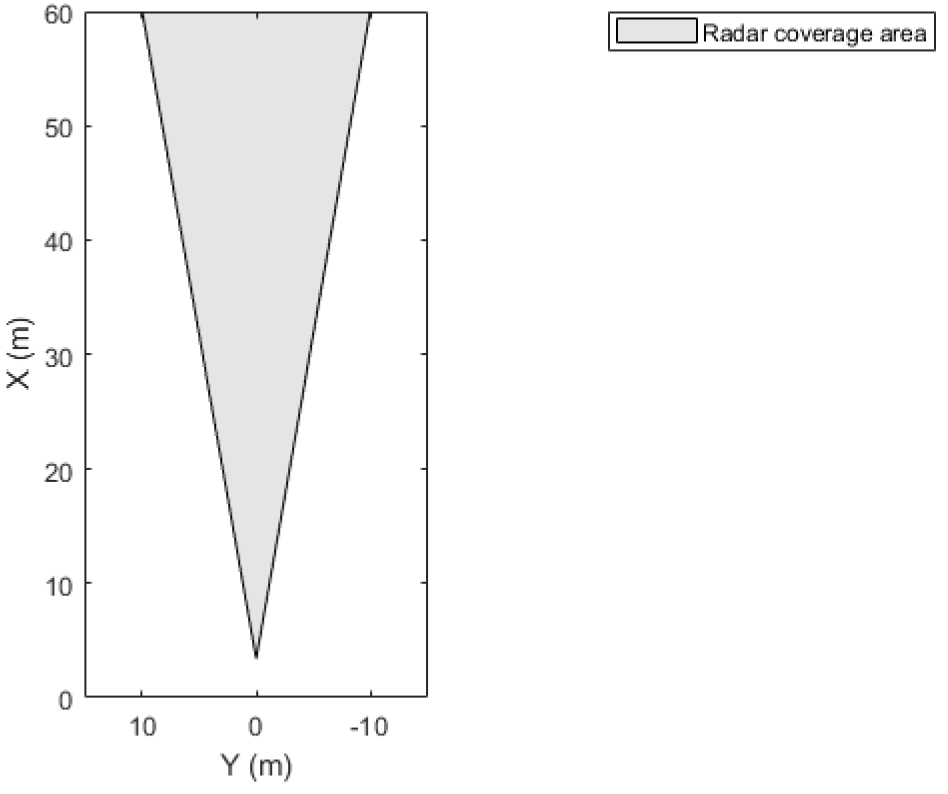
```
car1 = struct('ActorID',1,'Position',[40 0 0]);
car2 = struct('ActorID',2,'Position',[50 0 0]);
```

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);
```

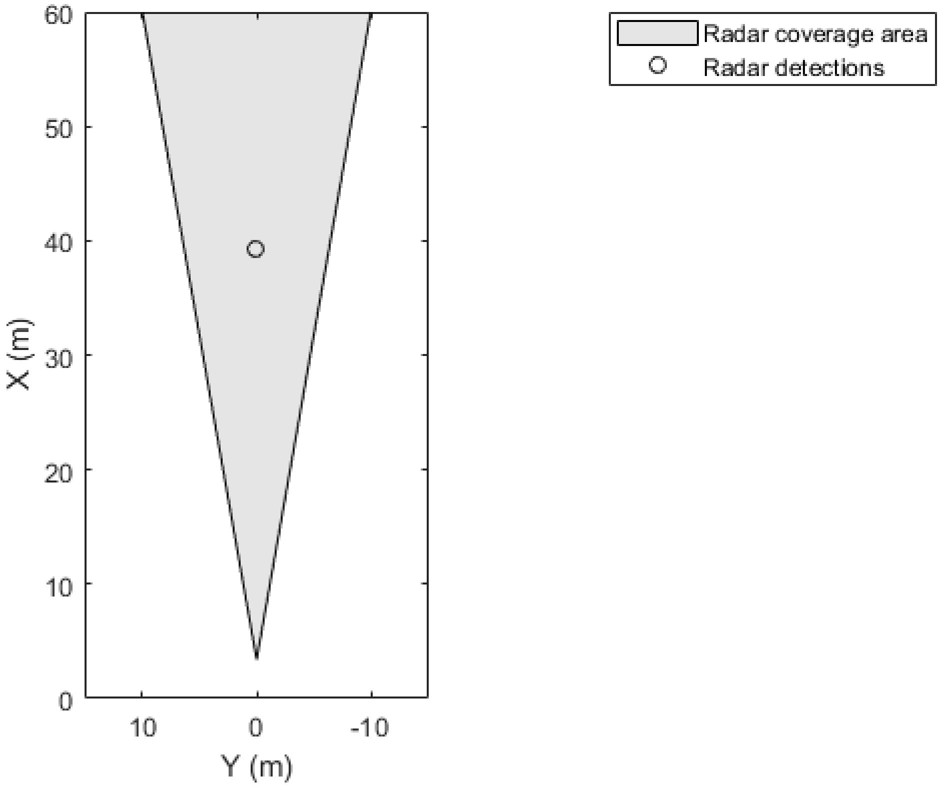
Display the coverage area of the radar detection generator on a bird's-eye plot.

```
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));
```



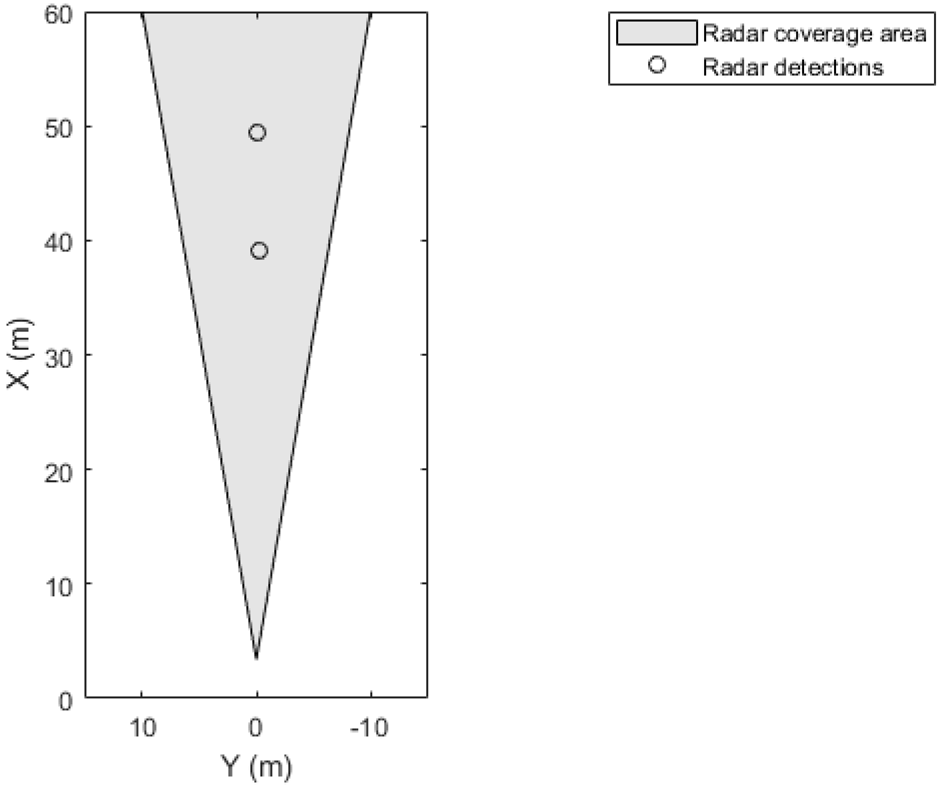
Extract the (*X*,*Y*) positions of the targets by converting the (*X*,*Y*) values of the `Measurement` field of the cell array into a MATLAB array. Then, display the detections.

```
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` | `objectDetection`

System Objects
`multiObjectTracker` | `visionDetectionGenerator`

Topics
“Model Radar Sensor Detections”
“Coordinate Systems in Automated Driving System Toolbox”

Introduced in R2017a

visionDetectionGenerator System object

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`.

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? (MATLAB).

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 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 (MATLAB).

`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 scalar

Required time interval between sensor updates, specified as a positive 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 scalar

Sensor height above the vehicle ground plane, specified as a positive 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 | scalar

Yaw angle of vision sensor, specified as a 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 | scalar

Pitch angle of vision sensor, specified as a 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 | scalar

Roll angle of the vision sensor, specified as a 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 scalar

Maximum detection range, specified as a positive scalar. The sensor cannot detect a target beyond this range. Units are in meters.

Example: `200`

Data Types: `double`

`MaxSpeed` — Maximum detectable object speed
`50` (default) | non-negative scalar

Maximum detectable object speed, specified as a non-negative scalar. Units are in meters per second.

Example: `10.0`

Data Types: `double`

`MaxAllowedOcclusion` — Maximum allowed occlusion of an object
`0.5` (default) | scalar in the range (0 1]

Maximum allowed occlusion of an object, specified as a 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 scalar less than or equal to 1

Probability of detecting a target, specified as a positive 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 scalar

Number of false detections that the vision sensor generates for each image, specified as a nonnegative 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 scalar

Bounding box accuracy, specified as a positive 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 scalar

Noise intensity used for filtering position and velocity measurements, specified as a positive 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 scalar

Required time interval between lane detection updates, specified as a positive 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 scalar

Accuracy of lane boundaries, specified as a positive 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` — Physical characteristics of actors

structure | structure array

Physical characteristics of actors, specified as structure or an array of structures. Each structure defines the physical characteristics, or profile, 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.

You can generate an array of structures for your driving scenario by using the `actorProfiles` method that acts on a `drivingScenario` object. This table shows the valid fields of the structure. When you do not specify a field, the fields are set to their default values.

Valid Actor Profile Fields	Description
`ActorID`	Scenario-defined actor identifier.
`ClassID`	User-defined classification identifier.
`Length`	Length of cuboid.
`Width`	Width of cuboid.
`Height`	Height of cuboid.

Valid Actor Profile Fields	Description
OriginOffset	Rotational center of the actor, defined as a displacement 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.
RCSPattern	Radar cross-section pattern matrix.
RCSAzimuthAngle	Azimuth angles corresponding to rows of RCSPattern.
RCSElevationAngle	Elevation angles corresponding to columns of RCSPattern.

For definitions of the structure fields and their default values, see the Actor and Vehicle classes.

Usage

Syntax

```
dets = sensor(actorposes,time)
lanedets = sensor(laneboundaries,time)
lanedets = sensor(actorposes,laneboundaries,time)
[ ___ ,numValidDets] = sensor( ___ )
[ ___ ,numValidDetsisValidTime] = sensor( ___ )
[dets,numValidDets,isValidTime,lanedets,numValidLaneDets,
isValidLaneTime] = sensor(actorposes,laneboundaries,time)
```

Description

`dets = sensor(actorposes,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(actorposes,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(actorposes,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

actorposes — 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 `targetPoses` method of an actor or vehicle. You can also create such a structure manually. The table shows the required fields of the structure:

Field	Description
ActorID	Unique actor identifier, specified as a scalar positive integer.
Position	Actor position vector, specified as real-valued 1-by-3 vector. Units are in meters.
Velocity	Actor velocity vector, specified as real-valued 1-by-3 vector. If velocity is not specified, the default value is [0 0 0]. Units are in meters per second.
Speed	Speed of actor, specified as a real scalar. When specified, the actor velocity is aligned with the *x*-axis of the actor in the ego actor coordinate system. You cannot specify both Speed and Velocity. Units are in meters per second.
Roll	Roll angle of actor, specified as a real-valued scalar. If roll is not specified, the default value is 0. Units are in degrees.
Pitch	Pitch angle of actor, specified as a real-valued scalar. If pitch is not specified, the default value is 0. Units are in degrees.
Yaw	Yaw angle of actor, specified as a real-valued scalar. If yaw is not specified, the default value is 0. Units are in degrees.

The values of the Position, Velocity, Speed, Roll, Pitch, and Yaw fields are defined with respect to the ego coordinate system. For definitions of the structure fields, see Actor and Vehicle.

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 defined in the table:

Lane Boundary Structure Fields

Field	Description
`Coordinates`	Lane boundary coordinates, specified as a real-valued *N*-by-3 matrix. Lane boundary coordinates define the position of points on the boundary at distances specified by `XDistance`. In addition, a set of boundary coordinates are inserted into the matrix at zero distance. 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 rows in the `Coordinates` matrix. Units are in degrees/m.
`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 rows in the `Coordinates` matrix. Units are in degrees/m. Units are in degrees/m^2.
`HeadingAngle`	Initial lane boundary heading, specified as a scalar. The heading angle of the lane boundary is relative to the ego car heading. Units are in degrees.
`LateralOffset`	Distance of the lane boundary from the ego vehicle position, specified as a scalar. An offset to a lane boundary to the left of the ego 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 the following: • `'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`	Strength of the lane boundary marking, specified as a scalar from 0 through 1. A value of 0 corresponds to a marking that is not visible and a value of 1 corresponds to a marking that is completely visible. Values in between are partially visible.
`Width`	Lane boundary width, specified as a positive scalar. In a double-line lane marker, the same width is used for both lines and the space between lines. Units are in meters.
`Length`	Length of dash in dashed lines, specified as a positive 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 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 scalar

Current simulation time, specified as a positive 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
`MeasurementParameters`	Parameters used by initialization functions of nonlinear Kalman tracking filters
`ObjectAttributes`	Additional information passed to tracker

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..

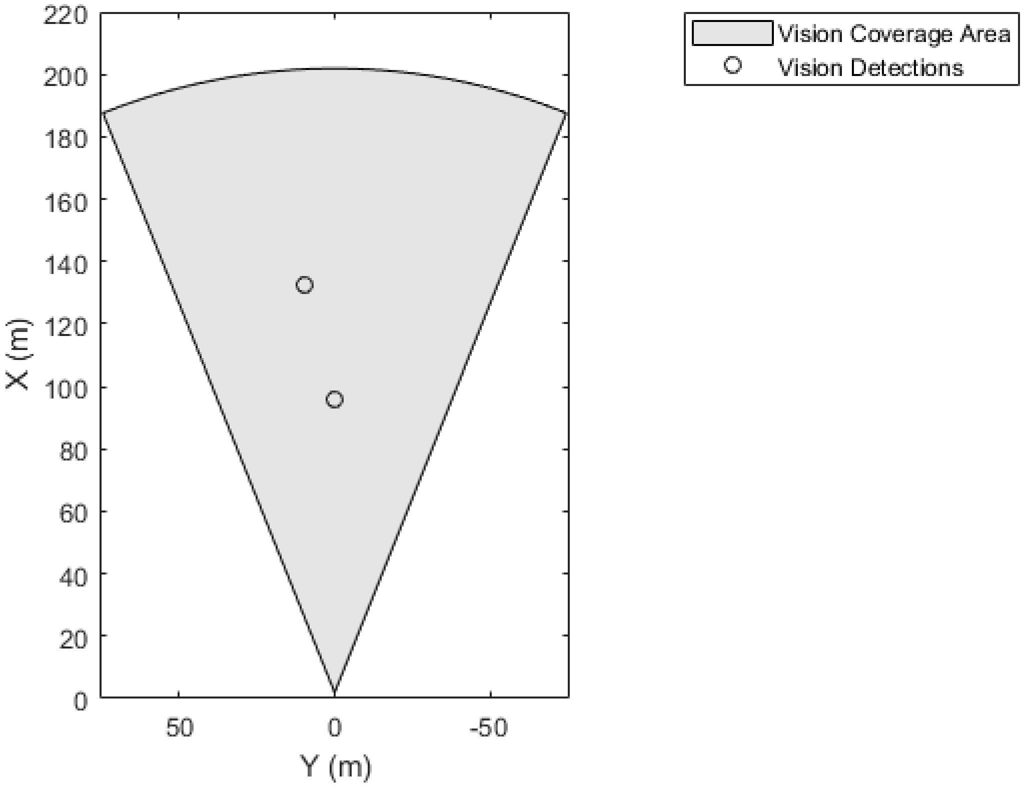
```
sensor = visionDetectionGenerator('DetectionProbability',1, ...
    'MinObjectImageSize',[5 5],'MaxRange',200,'DetectionCoordinates','Sensor Cartesian
tracker = multiObjectTracker('FilterInitializationFcn',@initcvkf, ...
    'ConfirmationParameters',[3 4],'NumCoastingUpdates',6);
```

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` methods.

```
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 car 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.

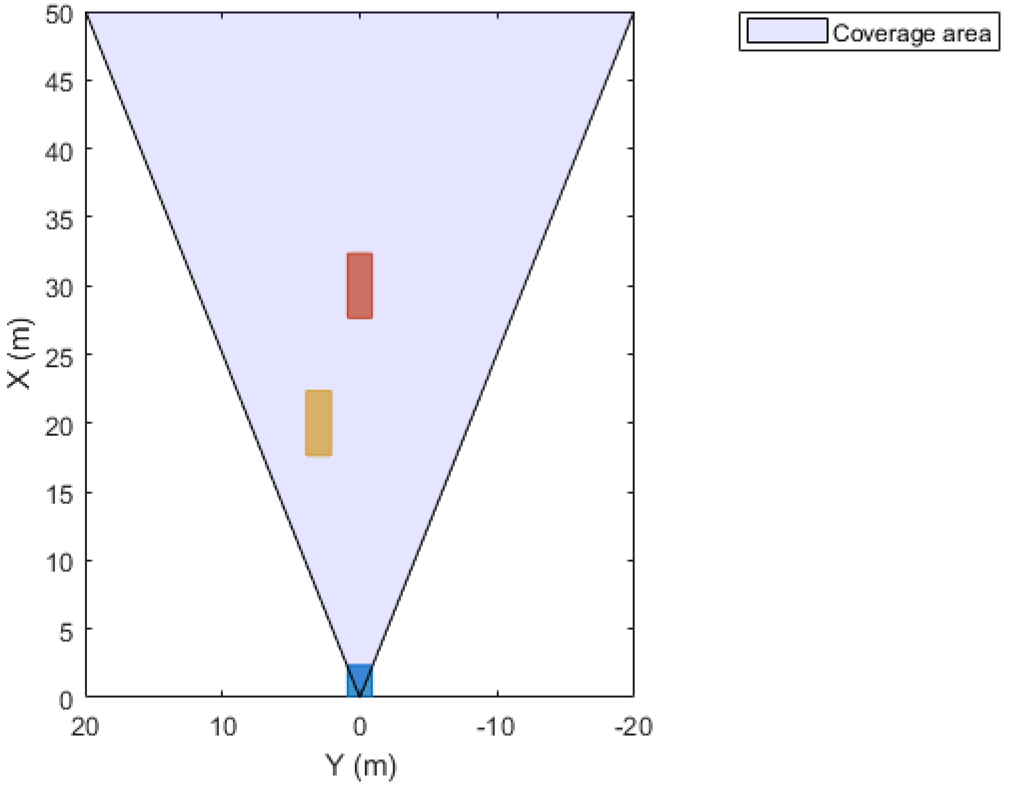
```
visionSensor = visionDetectionGenerator(monoCamConfig);
```

Generate a driving scenario with an ego car and two target cars. Position the first target car 30 meters directly in front of the ego car. Position the second target car 20 meters in front of the ego car but offset to the left by 3 meters.

```
scenario = drivingScenario;
egoCar = vehicle(scenario);
targetCar1 = vehicle(scenario,'Position',[30 0 0]);
targetCar2 = vehicle(scenario,'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(egoCar);
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 car. Use these poses to generate detections from the sensor.

```
poses = targetPoses(egoCar);
[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 = 2.77 meters
Detection 2: X = 27.81 meters, Y = 0.08 meters
```

Generate Object and Lane Boundary Detections

Create a driving scenario containing an ego car and a target vehicle traveling along a three-lane road. Detect the lane boundaries using a vision sensor.

```
sc = drivingScenario;
```

Create a three-lane road using lane specifications.

```
roadCenters = [0 0 0; 60 0 0; 120 30 0];
lspc = lanespec(3);
road(sc,roadCenters,'Lanes',lspc);
```

The ego car follows the center lane at 30 m/s.

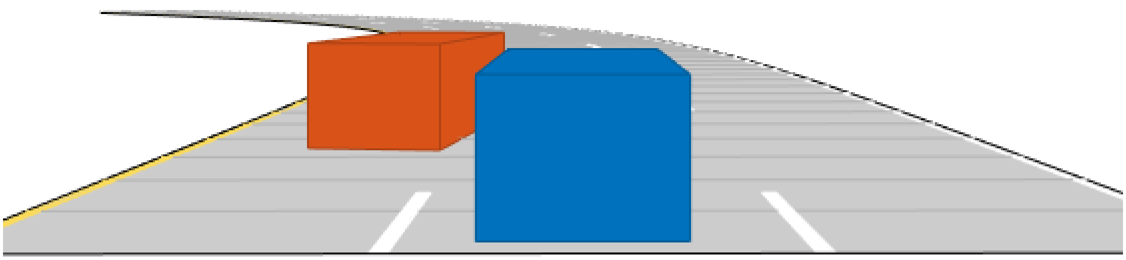
```
egocar = vehicle(sc);
egopath = [1.5 0 0; 60 0 0; 111 25 0];
egospeed = 30;
trajectory(egocar,egopath,egospeed);
```

The target vehicle travels ahead at 40 m/s and changes lanes close to the ego vehicle.

```
targetcar = vehicle(sc,'ClassID',2);
targetpath = [8 2; 60 -3.2; 120 33];
targetspeed = 40;
trajectory(targetcar,targetpath,targetspeed);
```

Display a chase plot showing a 3-D view from behind the ego vehicle.

```
chasePlot(egocar)
```



Create a vision detection generator that detects lanes and objects. The pitch of the sensor points one degree downward.

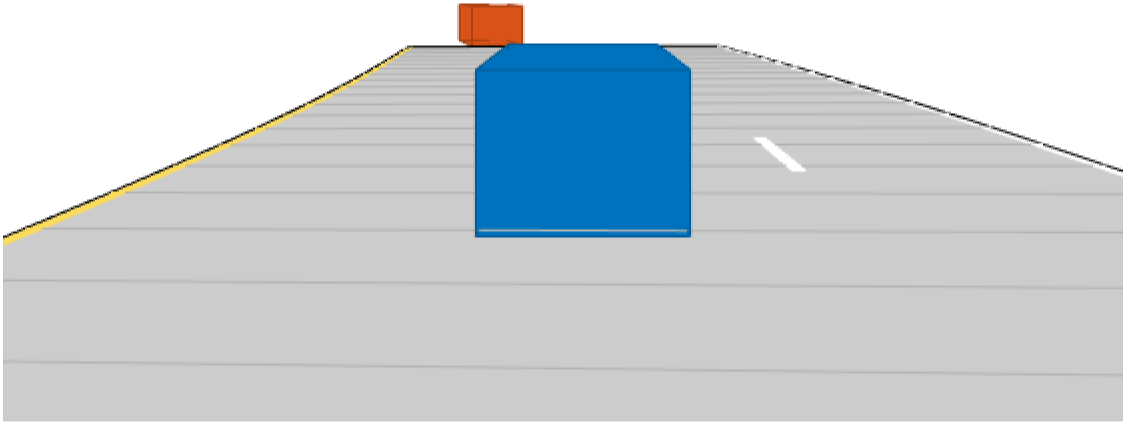
```
visionSensor = visionDetectionGenerator('Pitch',1.0);
visionSensor.DetectorOutput = 'Lanes and objects';
visionSensor.ActorProfiles = actorProfiles(sc);
```

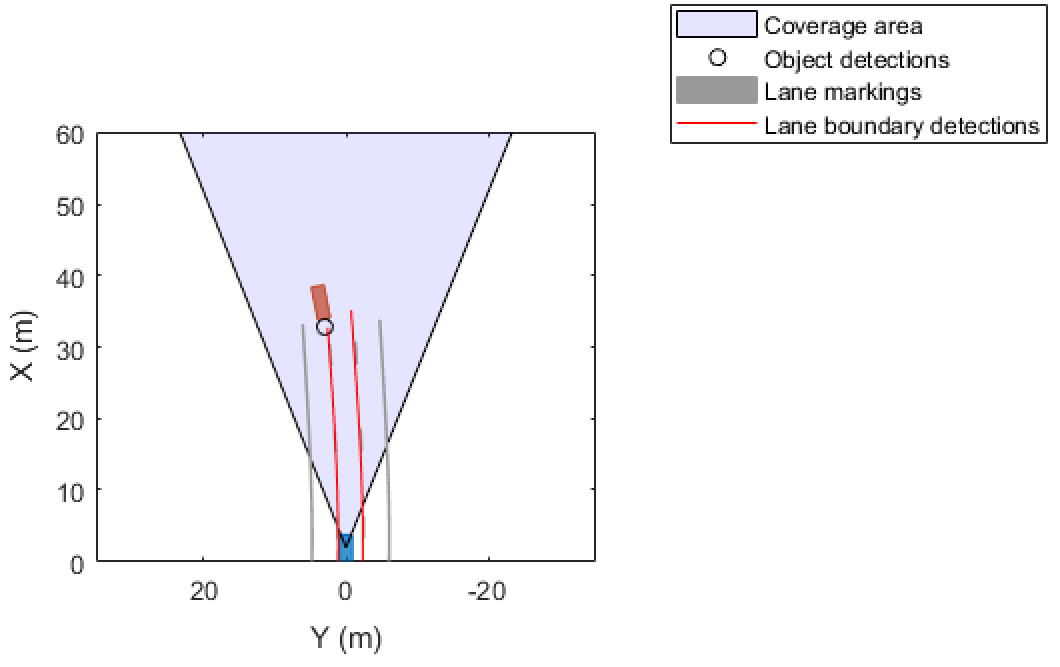
Run the simulation.

- Create a bird's eye plot and the associated plotters.
- Plot the sensor coverage area.
- Display lane markings.

- Obtain ground truth poses of targets on the road.
- Obtain ideal lane boundary points up to 60 m ahead.
- Generate detections from the ideal target poses and lane boundaries.
- Plot outline of target.
- Plot object detections when the object detection is valid.
- Plot 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(sc)
    [lmv,lmf] = laneMarkingVertices(egocar);
    plotLaneMarking(lmPlotter,lmv,lmf)
    tgtpose = targetPoses(egocar);
    lookaheadDistance = 0:0.5:60;
    lb = laneBoundaries(egocar,'XDistance',lookaheadDistance,'LocationType','inner');
    [obdets,nobdets,obValid,lb_dets,nlb_dets,lbValid] = ...
        visionSensor(tgtpose,lb,sc.SimulationTime);
    [objposition,objyaw,objlength,objwidth,objriginOffset,color] = targetOutlines(egoca
    plotOutline(olPlotter,objposition,objyaw,objlength,objwidth, ...
        'OriginOffset',objriginOffset,'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.

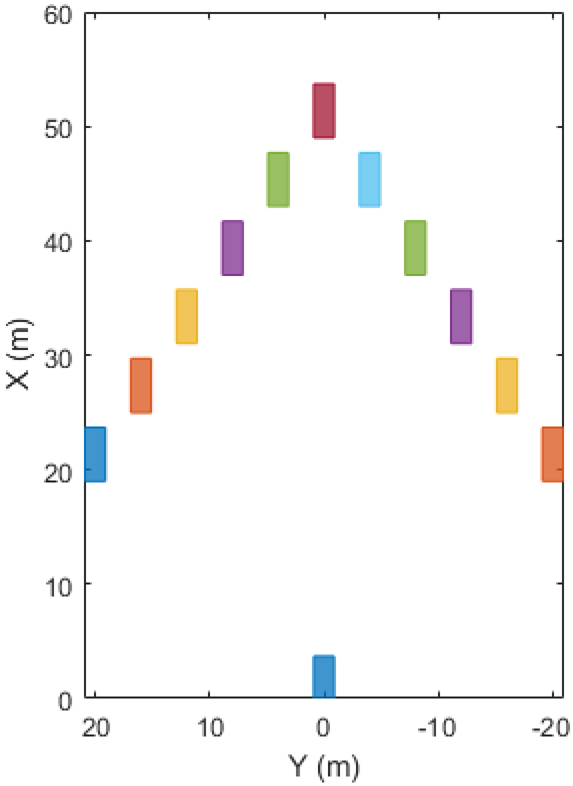
Create a Driving Scenario

Create a driving scenario in which the ego car 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;
egoCar = vehicle(scenario);

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,'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(egoCar);
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.

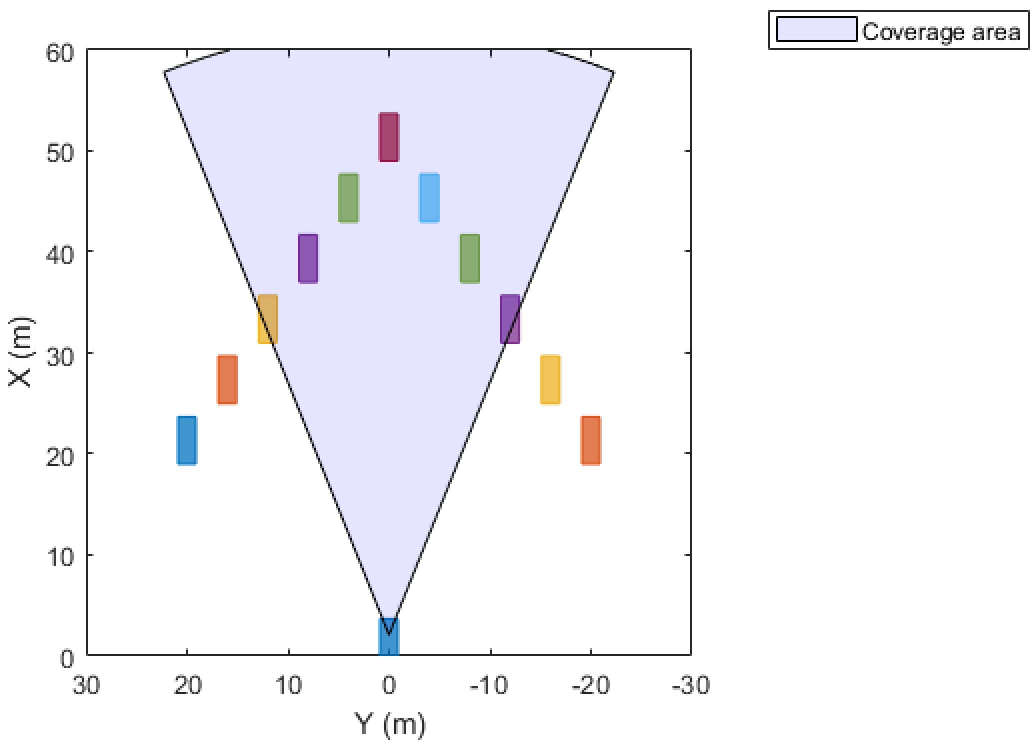
```
idealSensor = visionDetectionGenerator( ...
    'SensorIndex',1, ...
    'UpdateInterval',scenario.SampleTime, ...
    'SensorLocation',[0.75*egoCar.Wheelbase 0], ...
    'Height',1.1, ...
    'Pitch',0, ...
    'Intrinsics',cameraIntrinsics(800,[320 240],[480 640]), ...
    'BoundingBoxAccuracy',50, ... % 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: 50
       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(egoCar);
```

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
```

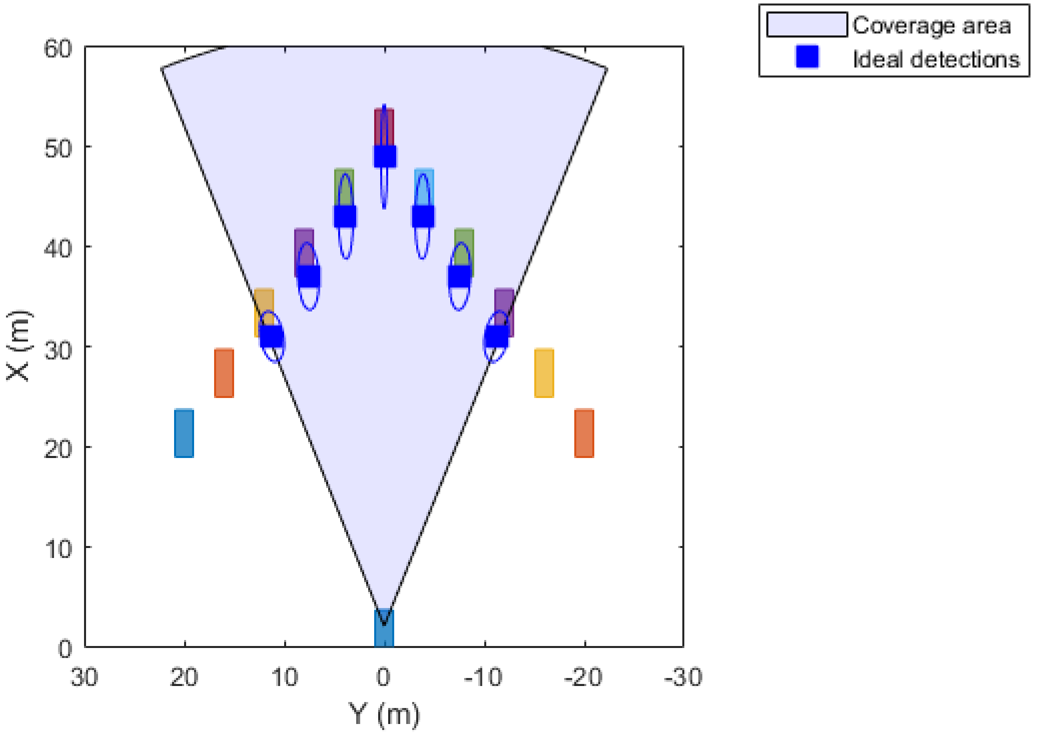
```
dets{1}.MeasurementNoise
```

```
ans = 6×6

    1.5903   -0.2174         0         0         0         0
   -0.2174    0.3744         0         0         0         0
         0         0  100.0000         0         0         0
         0         0         0    0.5808   -0.0405         0
         0         0         0   -0.0405    0.3544         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.

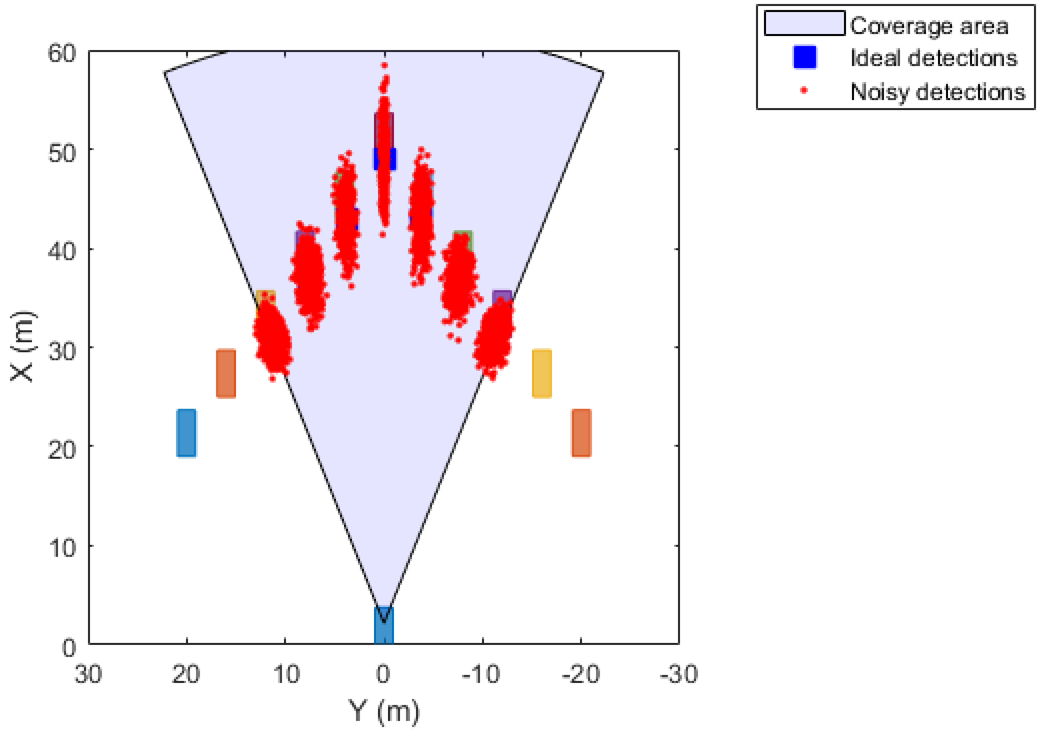
```
restart(scenario)
```

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 | monoCamera | objectDetection

System Objects
multiObjectTracker | radarDetectionGenerator

Functions
laneBoundaries | road

Topics
"Model Vision Sensor Detections"
"Coordinate Systems in Automated Driving System Toolbox"

Introduced in R2017a

outlinePlotter

Create bird's-eye-view outline plotter

Syntax

```
olPlotter = outlinePlotter(bep)
olPlotter = outlinePlotter(bep,Name,Value)
```

Description

`olPlotter = outlinePlotter(bep)` returns an object outline plotter for displaying outlines in a bird's-eye plot (`bep`). To plot the outlines in a bird's-eye-plot, use `plotOutline`.

From a given driving scenario, use `targetOutlines` to get the dimensions for all actors in the scene. 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.

`olPlotter = outlinePlotter(bep,Name,Value)` specifies additional options using one or more `Name,Value` pair arguments.

Examples

Plot Outlines of Targets in Bird's-Eye Plot

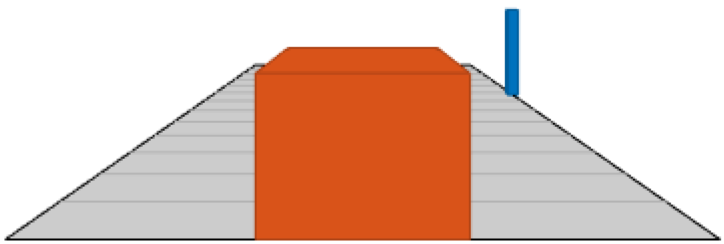
Create a driving scenario. Construct a 25 m road segment, add a pedestrian and a vehicle, and specify their trajectories to follow. The pedestrian crosses the road at 1 m/s. The vehicle drives along the road at 10 m/s.

```
s = drivingScenario;

road(s, [0 0 0; 25 0 0]);

p = actor(s,'Length',0.2,'Width',0.4,'Height',1.7);
```

```
v = vehicle(s);

trajectory(p,[15 -3 0; 15 3 0], 1);
trajectory(v,[v.RearOverhang 0 0; 25-v.Length+v.RearOverhang 0 0], 10);
```

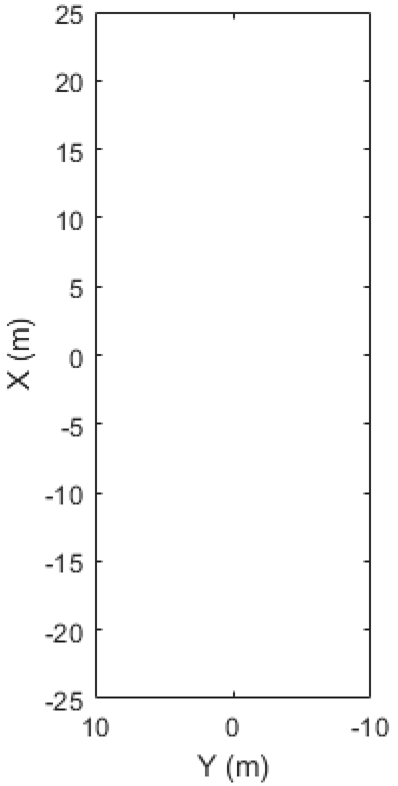
Add an egocentric plot for the vehicle

```
chasePlot(v,'Centerline','on')
```



Create a bird's-eye plot.

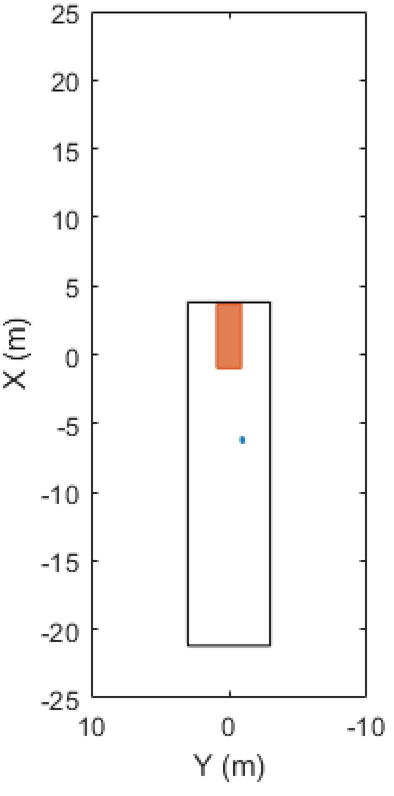
```
bep = birdsEyePlot('XLim',[-25 25],'YLim',[-10 10]);
olPlotter = outlinePlotter(bep);
lbPlotter = laneBoundaryPlotter(bep);
legend('off')
```

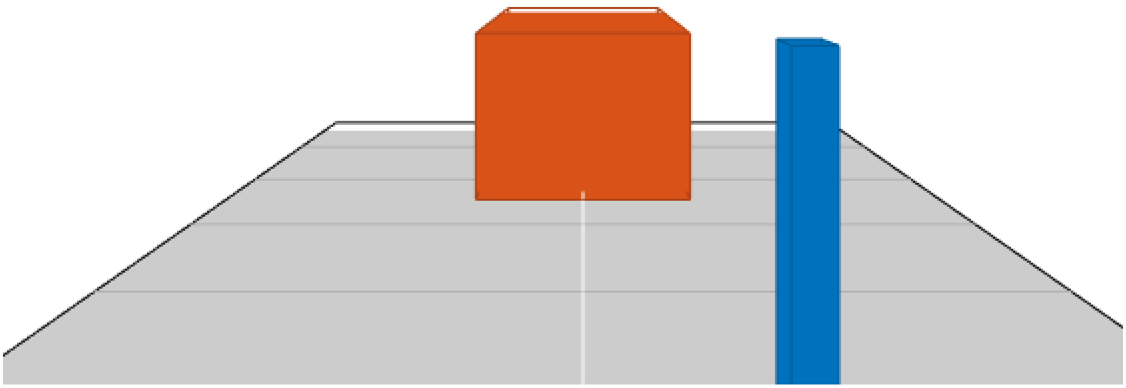


Start the simulation loop. Update the plotter with outlines for the targets.

```
while advance(s)
   % get 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
```





Input Arguments

bep — Unpopulated bird's-eye plot
birdsEyePlot handle

Unpopulated bird's-eye plot, specified as a `birdsEyePlot` handle that you can update with various plotters.

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: `'FaceAlpha'`,`0.5`

`FaceAlpha` — Transparency within each outline
`0.75` (default) | scalar

Transparency within each outline, specified as the comma-separated pair consisting of '`FaceAlpha`' and a scalar between 0 and 1. A value of 1 is fully opaque and a value of 0 is fully transparent.

`Tag` — Tag to identify plot of coverage area
`'PlotterN'` (default) | character vector | string scalar

Tag to identify the plot of the coverage area, specified as the comma-separated pair consisting of '`Tag`' and a character vector or string scalar. The default `'Tag'` used is `'PlotterN'`, where *N* is an integer.

Output Arguments

`olPlotter` — Outline plotter
plotter object

Outline plotter to use for the bird's-eye plot, returned as a plotter object.

See Also

Functions
`birdsEyePlot` | `plotOutline`

Introduced in R2017b

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. 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(occGrid)
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(occGrid)` creates a vehicle costmap from the occupancy grid `occGrid`. Use of this syntax requires Robotics System 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',3)` uses three circles 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
numeric matrix with values in the range [0, 1]

Cost values, specified as a numeric matrix with values 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 scalar

Width of costmap, in world units, specified as a positive scalar.

mapLength — Length of costmap
positive scalar

Length of costmap, in world units, specified as a positive scalar.

costVal — Uniform cost value
scalar in the range [0, 1]

Uniform cost value applied to all cells in the costmap, specified as a 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`.

occGrid — Occupancy grid
`robotics.OccupancyGrid` object | `robotics.BinaryOccupancyGrid` object

Occupancy grid, specified as a `robotics.OccupancyGrid` or `robotics.BinaryOccupancyGrid` object. Use of this argument requires Robotics System Toolbox.

mapLocation — Costmap location
`[0 0]` (default) | two-element numeric vector of form [*mapX* *mapY*]

Costmap location, specified as a two-element numeric 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) | scalar in the range [0, 1]

Threshold below which a grid cell is free, specified as a numeric 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) | scalar in the range [0, 1]

Threshold above which a grid cell is occupied, specified as a numeric 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 scalar

Side length of each square cell, in world units, specified as a positive 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

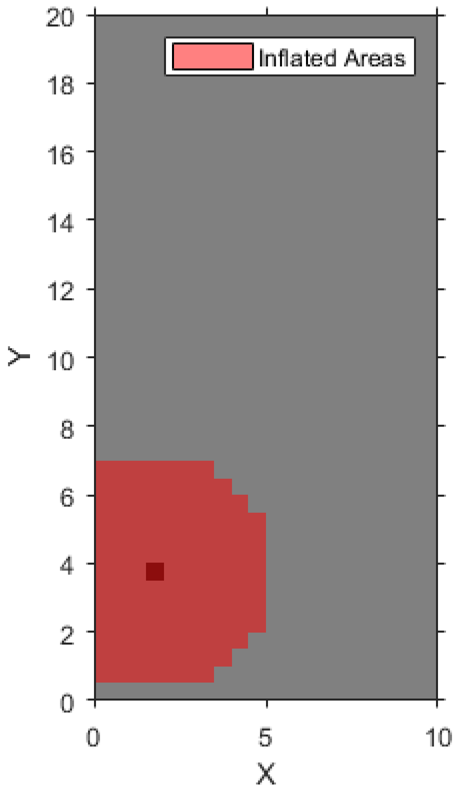
Create a 10-by-20 meter costmap that is divided into square cells of size 0.5-by-0.5 meters. Specify a default cost value of 0.5 for all cells.

```
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: [1×1 driving.costmap.InflationCollisionChecker]
             CellSize: 0.5000
              MapSize: [40 20]
            MapExtent: [0 10 0 20]
```

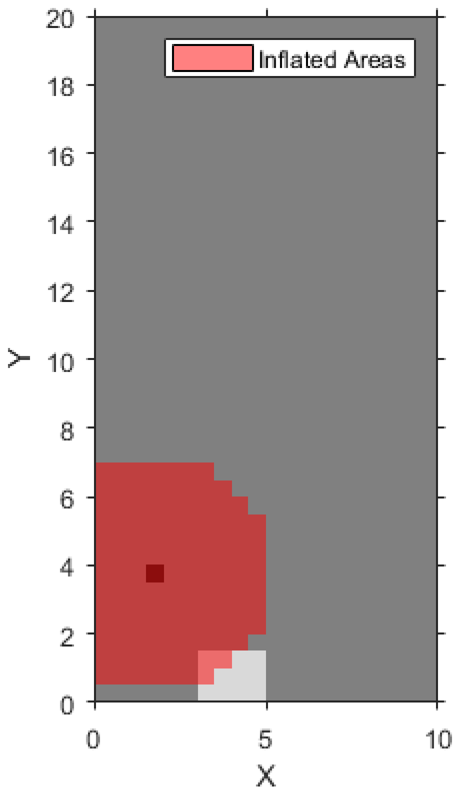
Mark an obstacle on the costmap. Display the costmap.

```
occupiedVal = 0.9;
xyPoint = [2,4];
setCosts(costmap,xyPoint,occupiedVal)

plot(costmap)
```



Mark an obstacle-free area on the costmap. Display the costmap again.

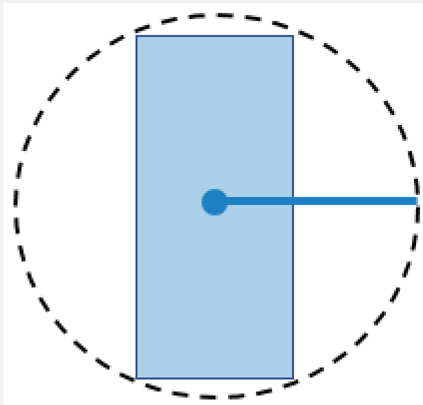
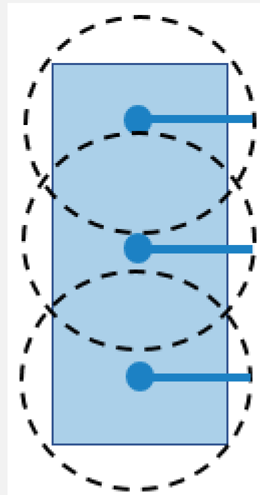
```
freeVal = 0.15;
[X,Y] = meshgrid(3.5:cellSize:5,0.5:cellSize:1.5);
setCosts(costmap,[X(:),Y(:)],freeVal)
plot(costmap)
```



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:

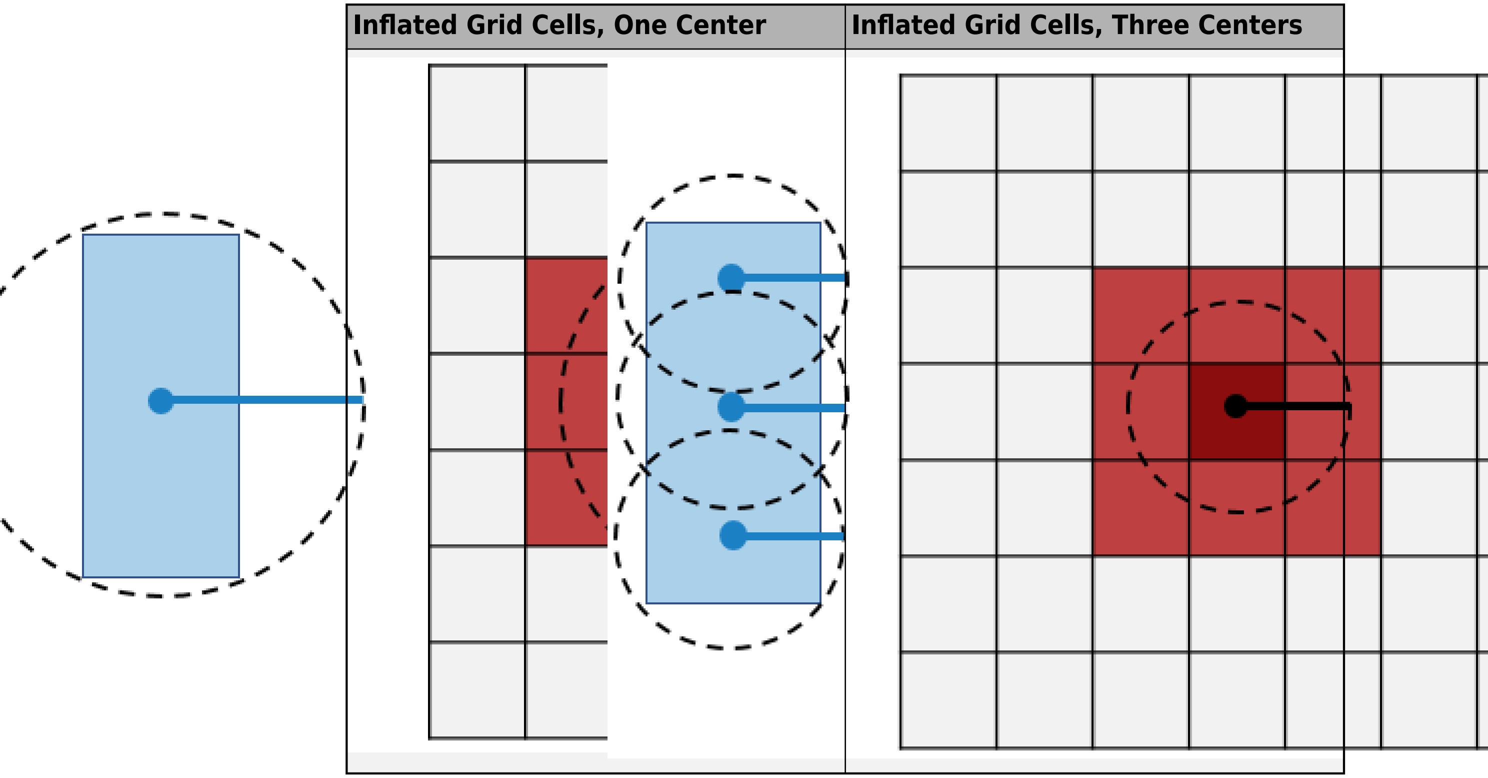
1. Calculate the inflation radius, in world units, from the vehicle dimensions. The default inflation radius is equal to the radius of the smallest set of overlapping circles required to completely enclose the vehicle. The center points of the circles lie along the longitudinal axis of the vehicle. Increasing 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.

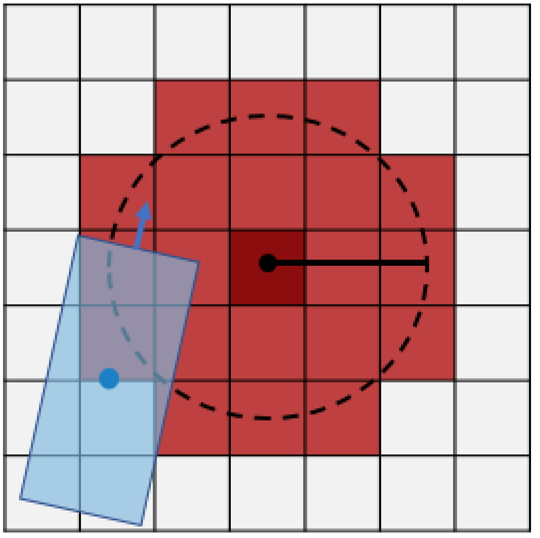
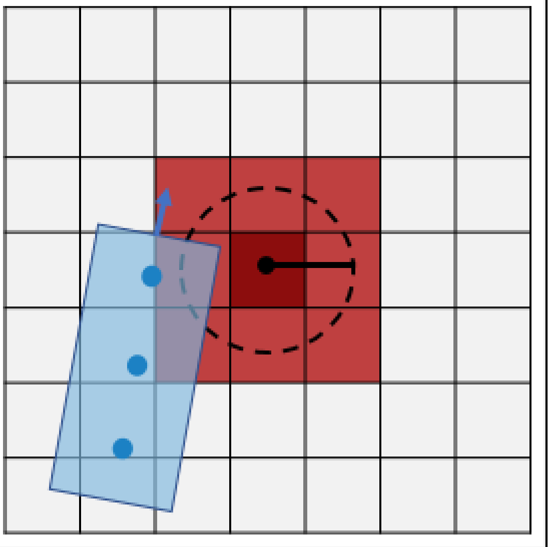
The diagrams show occupied cells in dark red. Cells in the inflated area are colored in light red. The solid black line shows the original inflation radius. In the diagram on the left, *R* is 3. In the diagram on the right, *R* is 2.



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 are not recommended

Not recommended starting in R2018b

The `InflationRadius` and `VehicleDimensions` properties of `vehicleCostmap` are not recommended. Instead:

1. Use the `inflationCollisionChecker` function to create an `InflationCollisionChecker` object, which has the properties `InflationRadius` and `VehicleDimensions`.
2. Specify this object as the value of the `CollisionChecker` property of `vehicleCostmap`.

There are no current plans to remove the `InflationRadius` and `VehicleDimensions` properties of `vehicleCostmap`. If you do specify these properties, the values in the corresponding properties of `CollisionChecker` are updated to match.

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.

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.

Discouraged 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);`

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

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 4-505 on the `vehicleCostmap` reference page.

Syntax

```
free = checkFree(costmap,vehiclePoses)
free = checkFree(costmap,xyPoints)
freeMat = checkFree(costmap)
```

Description

`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

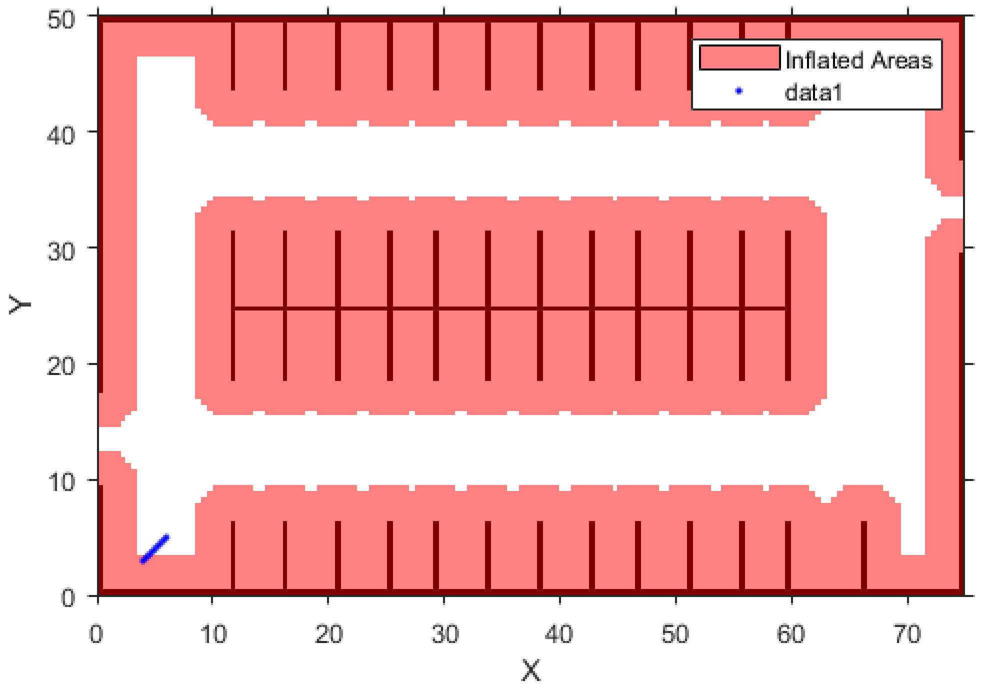
Check If Sequence of Poses Is Collision-Free

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.

`xyPoints` — Points
M-by-2 numeric vector

Points, specified as an *M*-by-2 numeric vector 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.

See Also

Objects
inflationCollisionChecker | pathPlannerRRT | vehicleCostmap

Functions
checkOccupied | checkPathValidity

Introduced in R2018a

checkOccupied

Check vehicle costmap for occupied poses or points

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 4-505 on the `vehicleCostmap` reference page.

Syntax

```
occ = checkOccupied(costmap,vehiclePoses)
occ = checkOccupied(costmap,xyPoints)
occMat = checkOccupied(costmap)
```

Description

`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

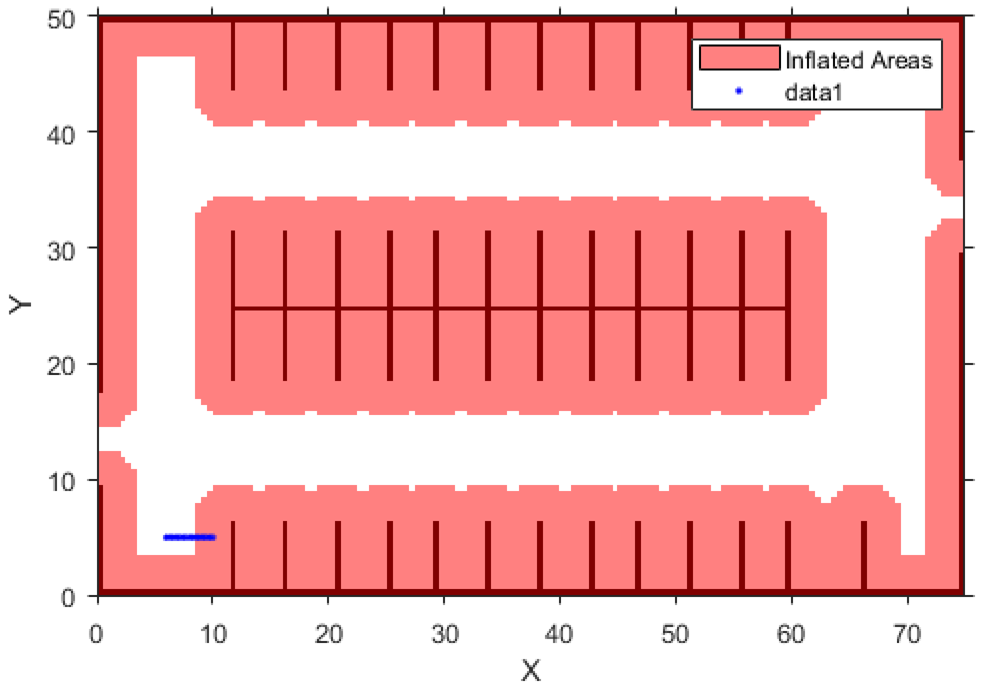
Check If Sequence of Poses Enters Occupied Cell

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 numeric vector

Vehicle poses, specified as an *M*-by-3 numeric vector. Each row corresponds to a pose of the form [*x*, *y*, *theta*]. The coordinates *x* and *y* must be specified with respect to the center of the rear axle of the vehicle, and are in world units. The heading angle *theta* is measured in degrees with respect to the *x*-axis.

Example: `[3.4 2.6 0]` specifies a vehicle with the center of the rear axle at (3.4, 2.6) and a heading angle of 0 degrees with respect to the x-axis.

`xyPoints` — Points

M-by-2 numeric vector

Points, specified as an *M*-by-2 numeric vector 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.

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.

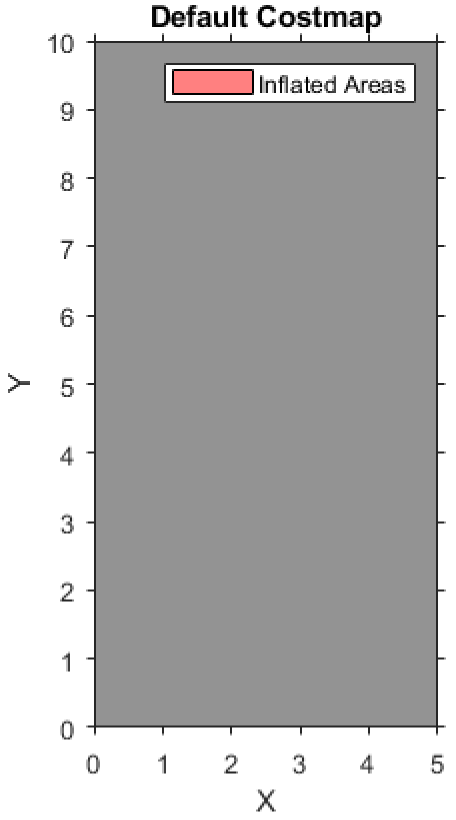
`costMat = getCosts(costmap)` returns a matrix, `costMat`, that contains the cost of each cell in the costmap.

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')
```



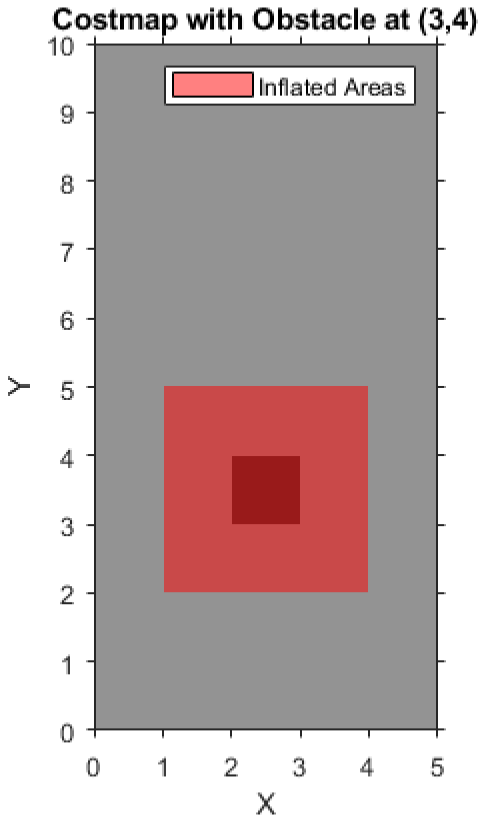

```
getCosts(costmap)
```

```
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 numeric vector

Points, specified as an *M*-by-2 numeric vector 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 numeric vector

Cost of points in `xyPoints`, returned as an *M*-element numeric vector.

`costMat` — Cost of all cells
numeric matrix

Cost of all cells in `costmap`, returned as a numeric matrix of the same size as the costmap grid. This size is specified by the `MapSize` property of the costmap.

See Also

`setCosts` | `vehicleCostmap`

Introduced in R2018a

plot

Plot vehicle costmap

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.

Syntax

```
plot(costmap)
plot(costmap,Name,Value)
```

Description

`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`).

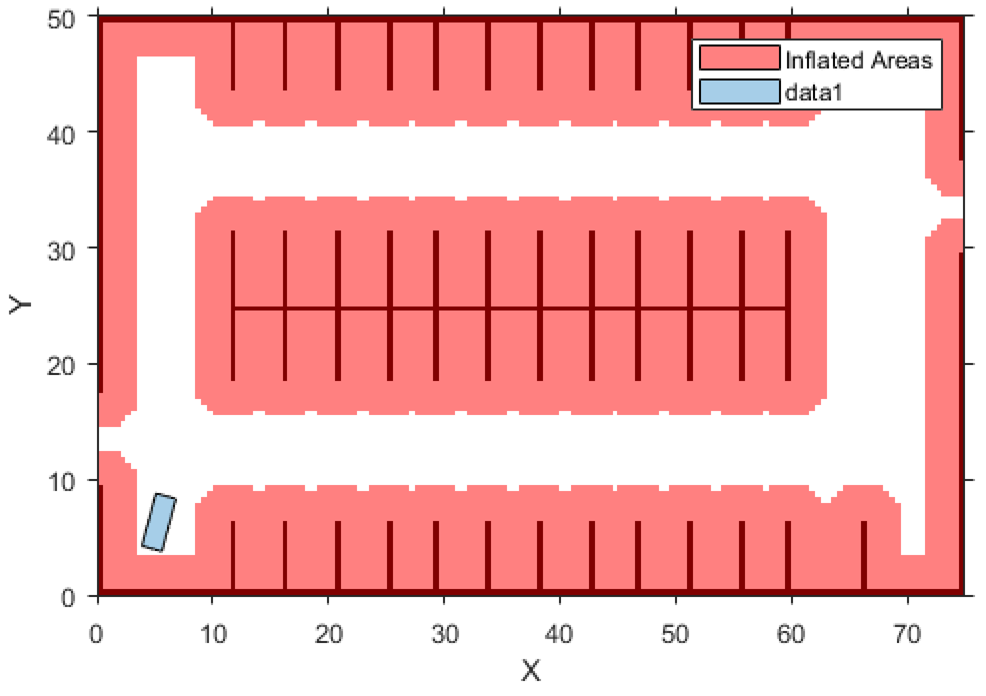
```
move = @(s,x,y,theta) rotate(translate(s,[x,y]), ...
    theta,[x,y]);
```

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 5-by-10 meter vehicle costmap. Cells have side length 1, in the world units of meters. Specify the inflation radius as 2 meters.

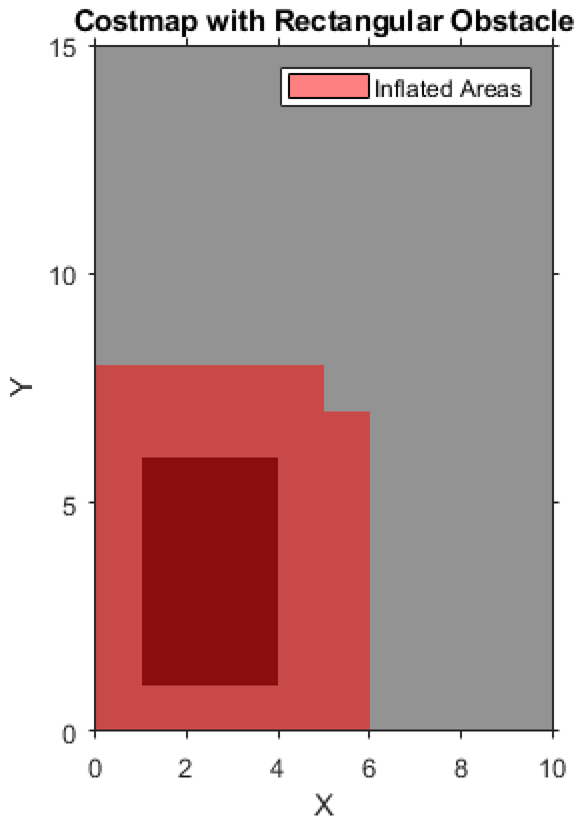
```
costmap = vehicleCostmap(10,15,'InflationRadius',2);
```

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 rectangular 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 numeric vector

Points, specified as an *M*-by-2 numeric vector 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 numeric vector

Cost of points in `xyPoints`, specified as an *M*-element numeric vector.

Example: `0.8` specifies the cost of a single point

Example: `[0.2 0.5 0.8]` specifies the cost of three points

See Also
`getCosts` | `vehicleCostmap`

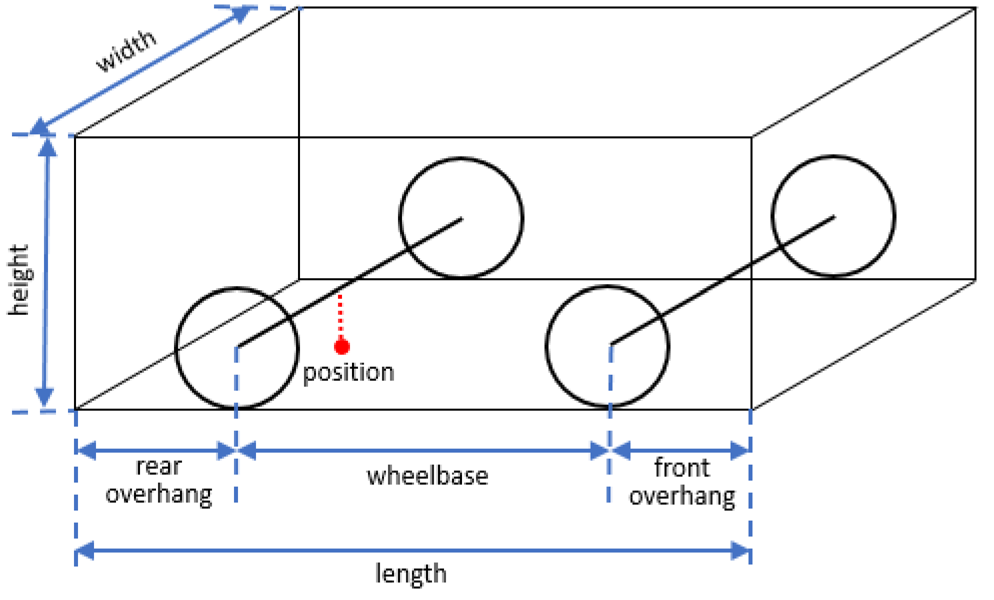
Introduced in R2018a

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 scalar

Length of vehicle, specified as a positive scalar.

Data Types: `double`

`Width` — Width of vehicle
`1.8` (default) | positive scalar

Width of vehicle, specified as a positive scalar.

Data Types: `double`

`Height` — Height of vehicle
`1.4` (default) | positive scalar

Height of vehicle, specified as a positive scalar.

Data Types: `double`

`FrontOverhang` — Front overhang of vehicle
`0.9` (default) | numeric scalar

Front overhang of vehicle, specified as a numeric 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) | numeric scalar

Rear overhang of vehicle, specified as a numeric 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 scalar

The distance between the front and rear axles of the vehicle, specified as a positive 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.

See Also

`vehicle` | `vehicleCostmap`

Introduced in R2018a

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 scalar

This property is read-only.

Length of the path, in world units, specified as a positive scalar.

Object Functions

interpolate	Interpolate poses along planned vehicle path
plot	Plot planned vehicle path

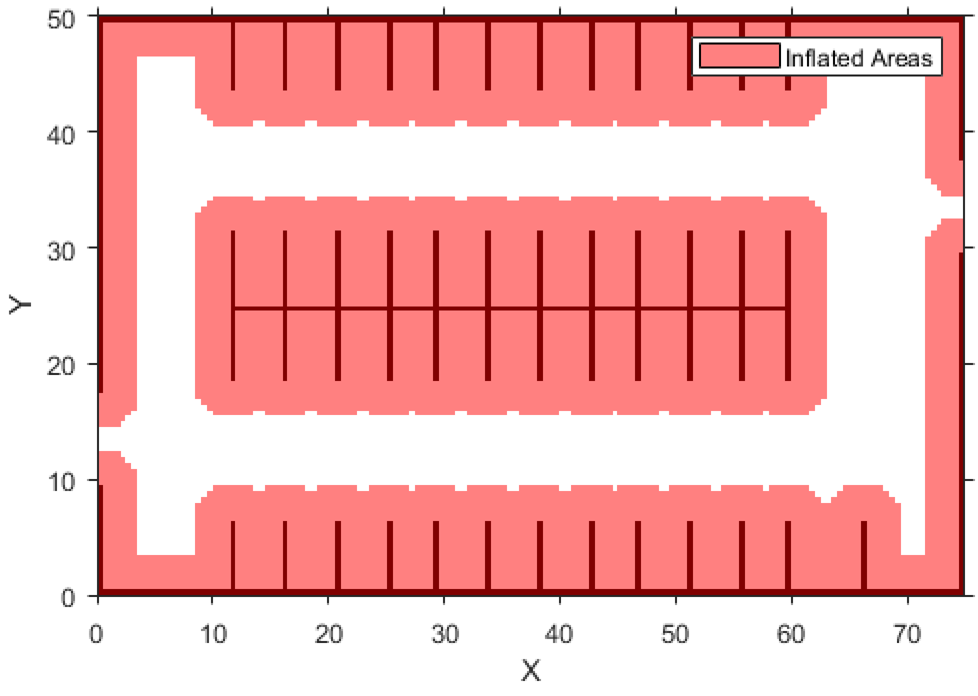
Examples

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)
```

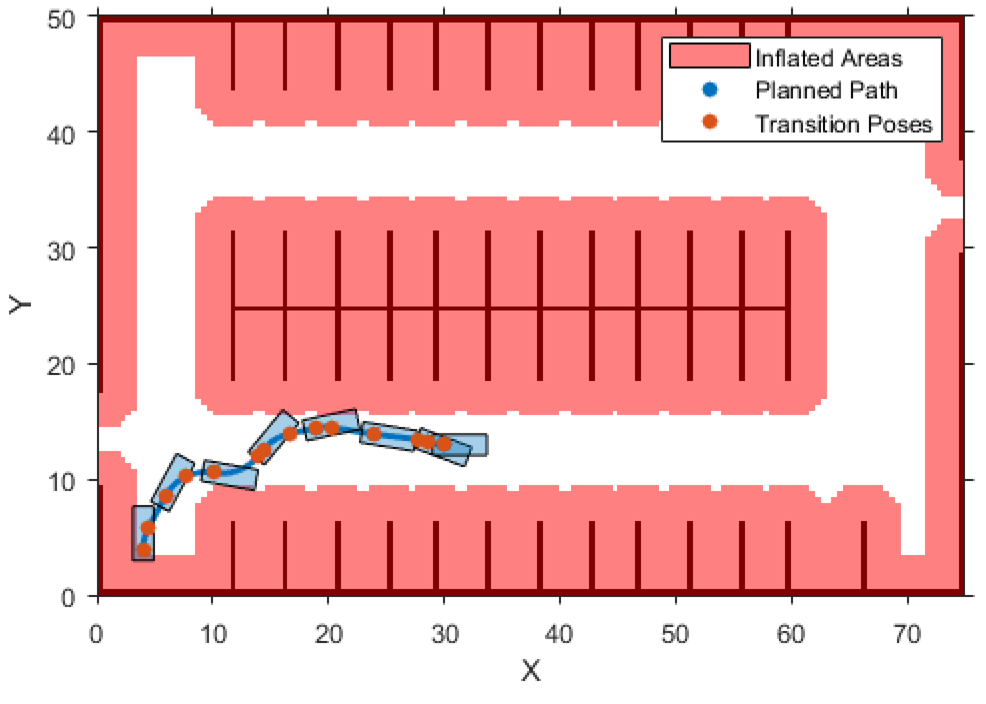
```
isPathValid = logical
   1
```

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
```

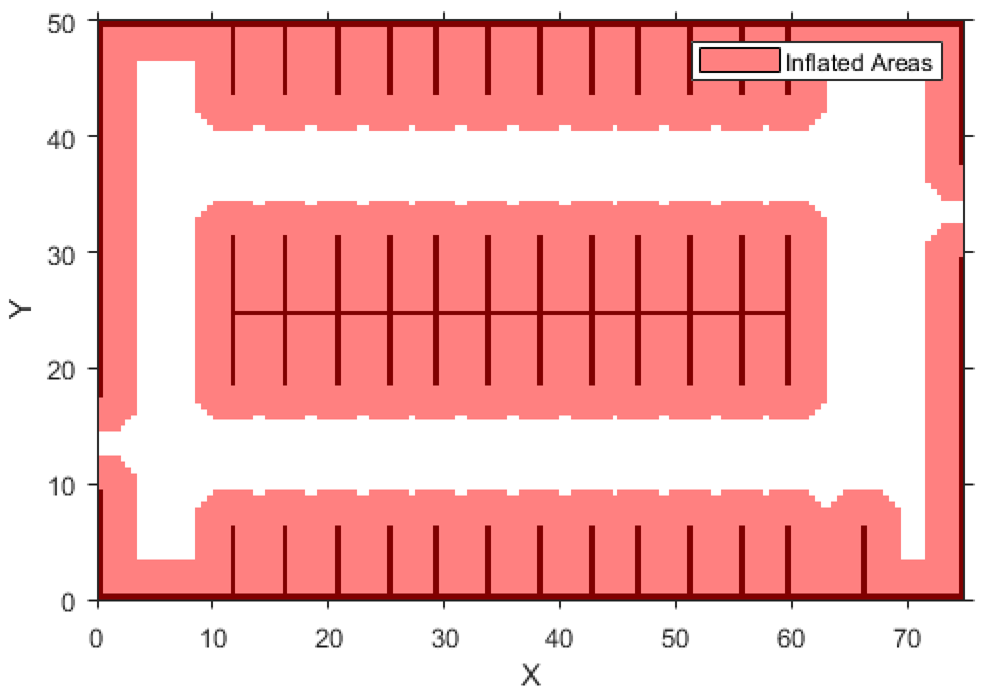


Plan Path and Interpolate Along Path

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.

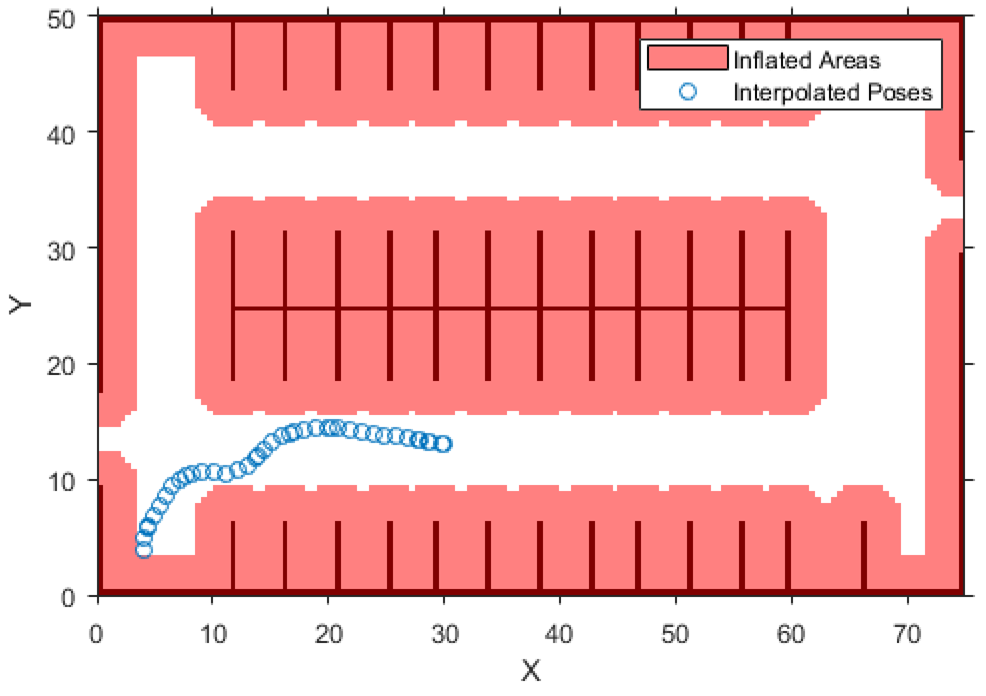
```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

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.

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 | plot

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.

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.

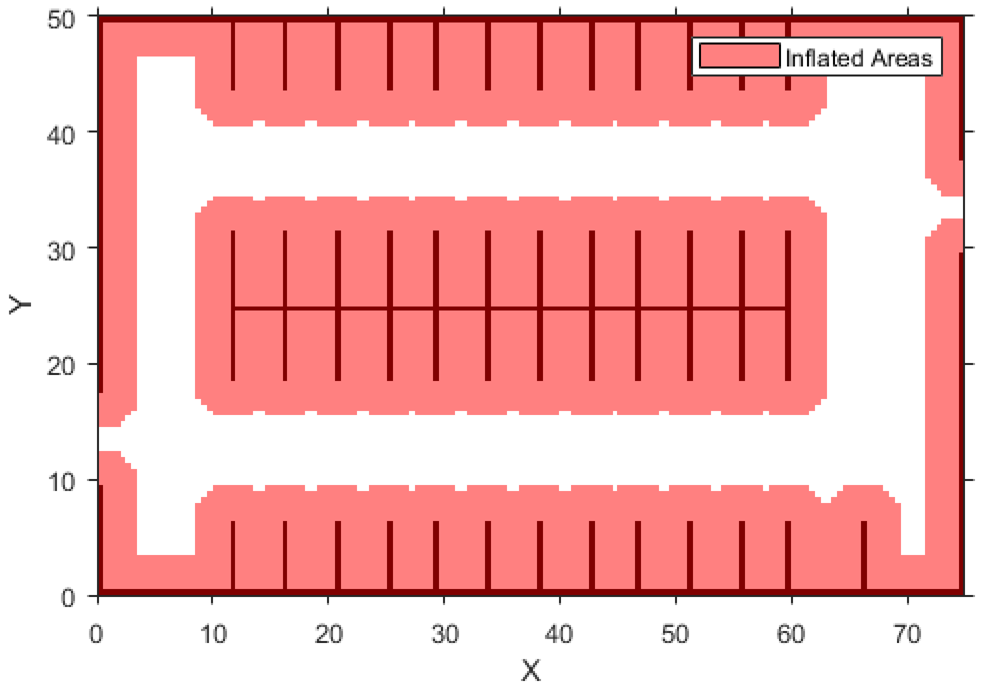
Examples

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)
```

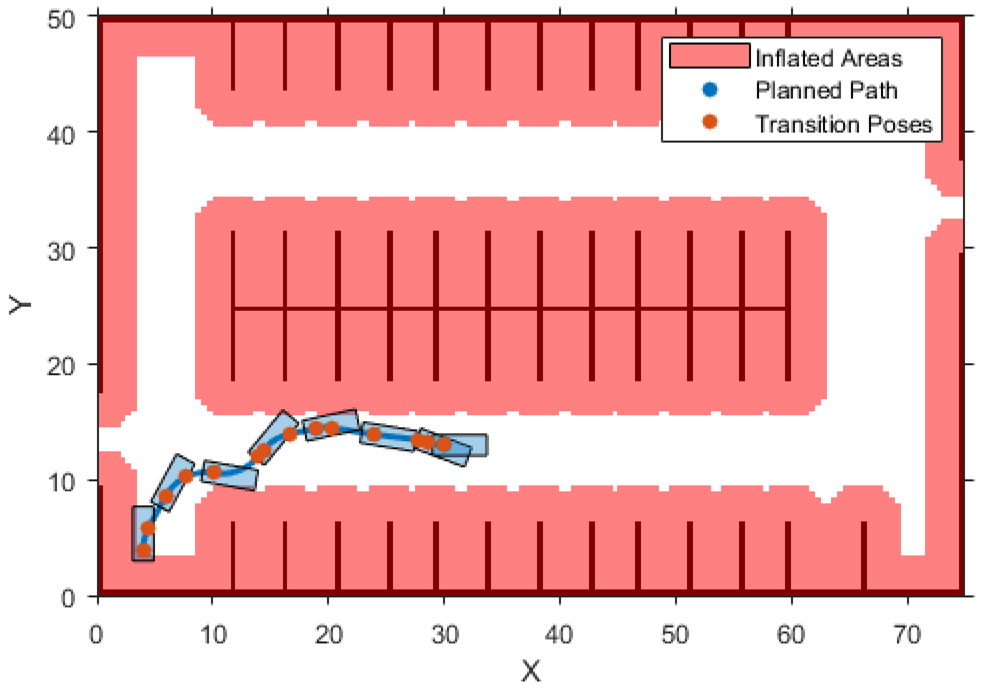
```
isPathValid = logical
   1
```

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` — Color of path
RGB triplet

Color of the path, specified as the comma-separated pair consisting of `'Color'` and an RGB triplet in the range [0, 1]. For details on specifying RGB triplets, see `ColorSpec (Color Specification)`.

`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 4-561.

`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.

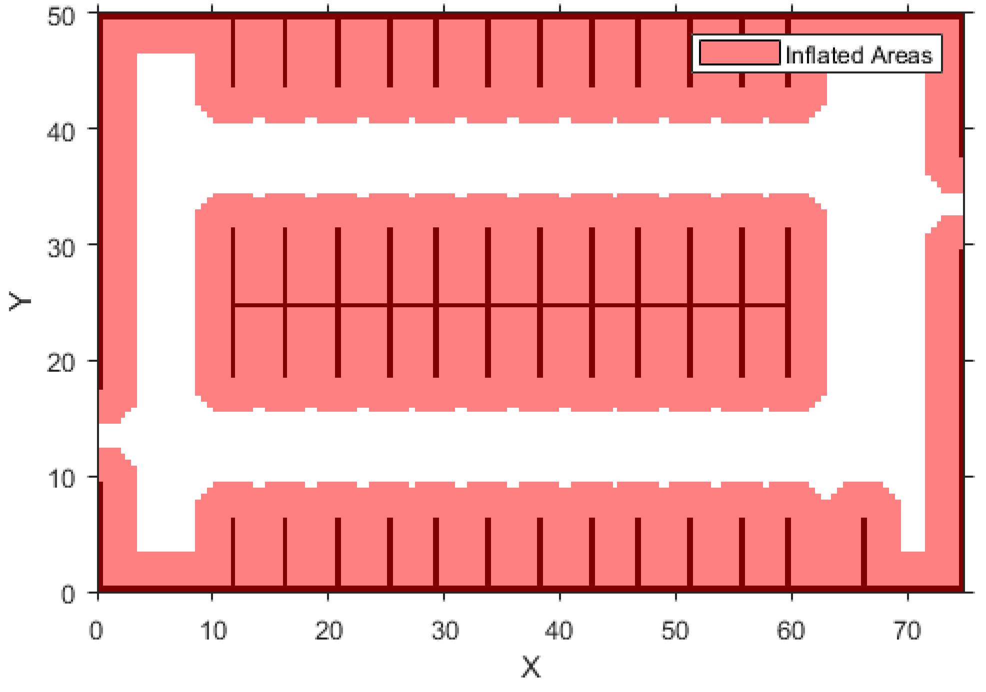
Examples

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)
```

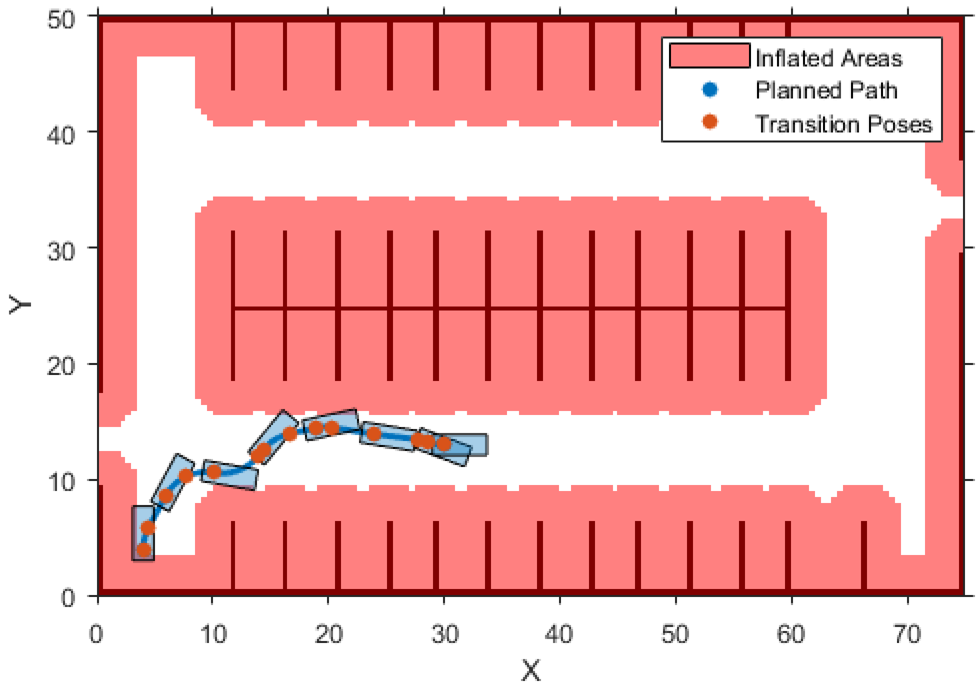
```
isPathValid = logical
   1
```

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
```

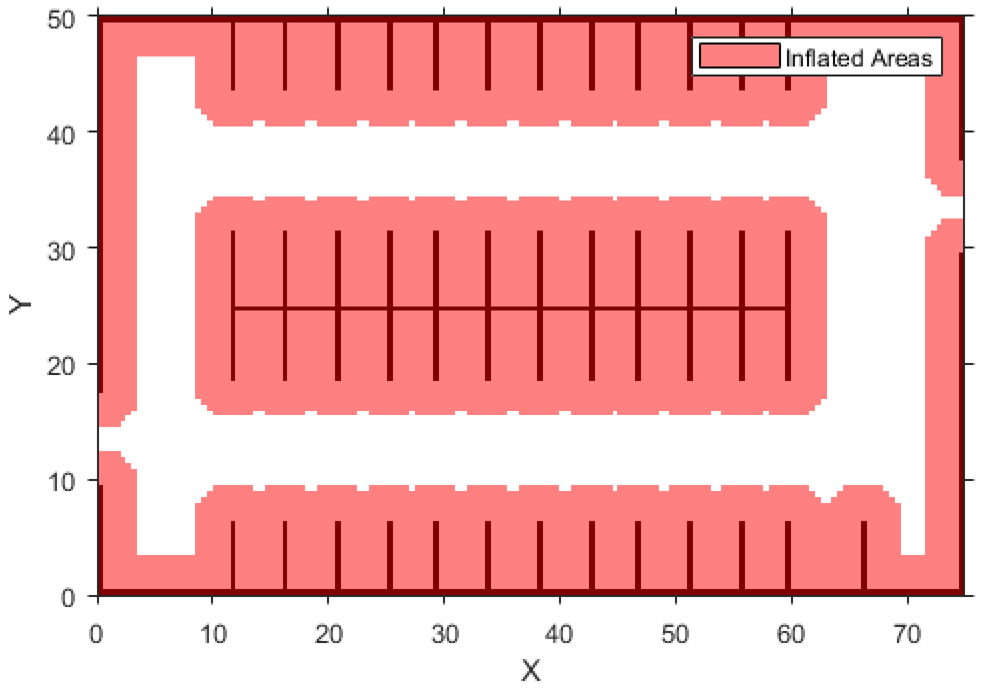


Plan Path and Interpolate Along Path

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.

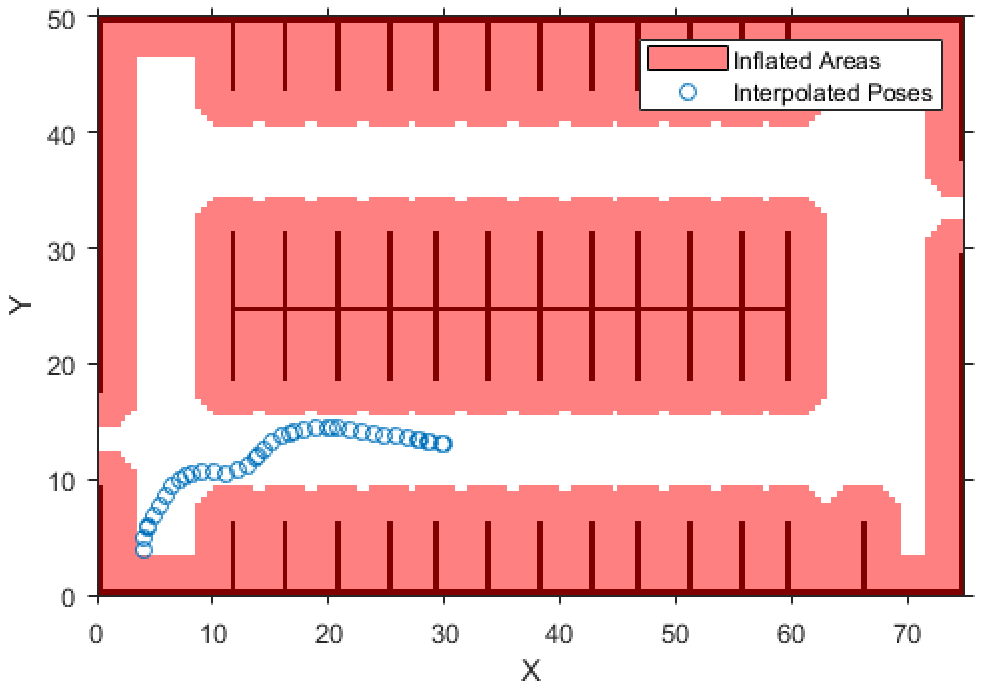
```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

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
numeric vector

Points along the length of the path, specified as a numeric 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)

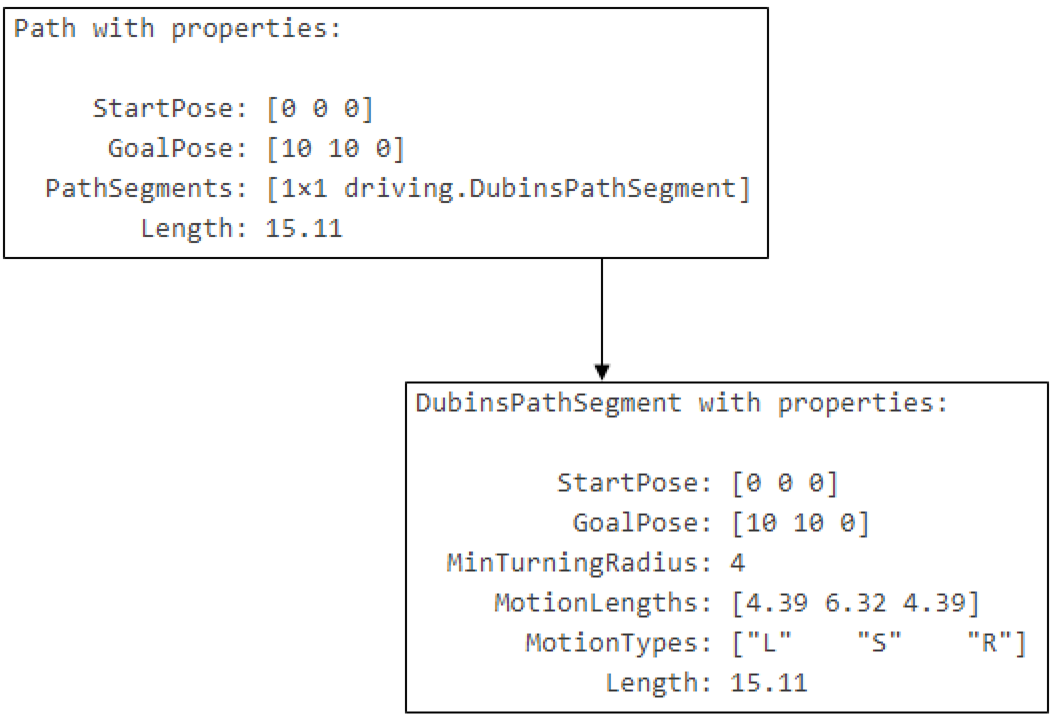
Motion directions of vehicle poses, returned as an *m*-by-1 vector of 1s (forward motion) and –1s (reverse motion). *m* is the number of returned poses. Each element of `directions` corresponds to a row of `poses`.

Definitions

Transition Poses

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.

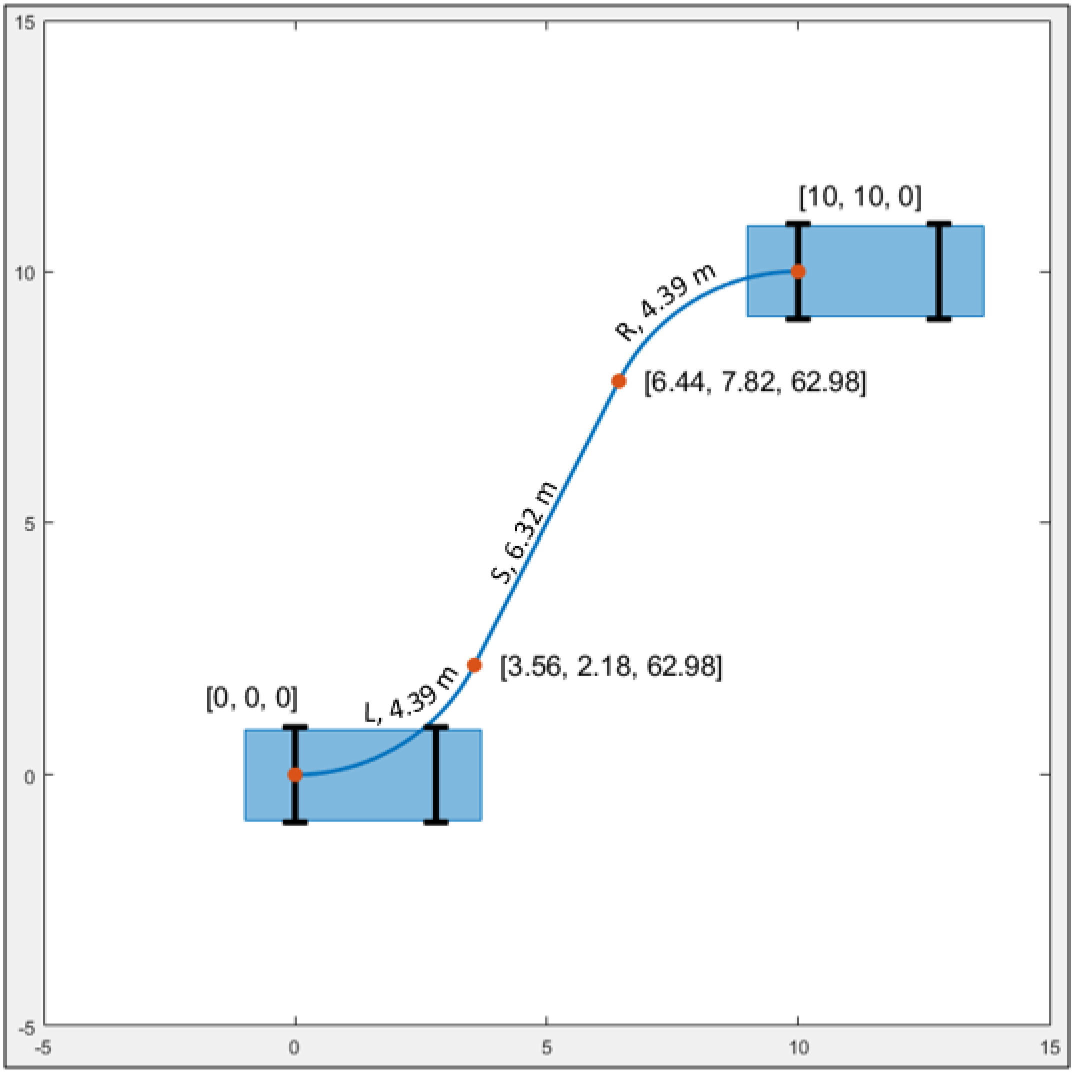
The path length between transition poses is given by the `MotionLengths` property of the path segments. For example, consider the following path, which is a `driving.Path` object composed of a single Dubins path segment. This segment consists of three motions, as described by the `MotionLengths` and `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.



See Also

Functions
`checkPathValidity`

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 scalar

This property is read-only.

Minimum turning radius of the vehicle, in world units, specified as a positive 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 numeric vector

This property is read-only.

Length of each motion in the path segment, in world units, specified as a three-element numeric 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 scalar

This property is read-only.

Length of the path segment, in world units, specified as a positive 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.

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 scalar

This property is read-only.

Minimum turning radius of the vehicle, in world units, specified as a positive 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 numeric vector

This property is read-only.

Length of each motion in the path segment, in world units, specified as a five-element numeric 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 scalar

This property is read-only.

Length of the path segment, in world units, specified as a positive 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.

See Also

Objects
`driving.DubinsPathSegment` | `driving.Path` | `pathPlannerRRT`

Topics
"Automated Parking Valet"

Introduced in R2018b

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 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) | scalar in the range [0, 1]

Probability of selecting the goal pose instead of a random pose, specified as a 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 scalar

Maximum distance between two connected poses, specified as a positive 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 scalar

Minimum turning radius of the vehicle, specified as a positive 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

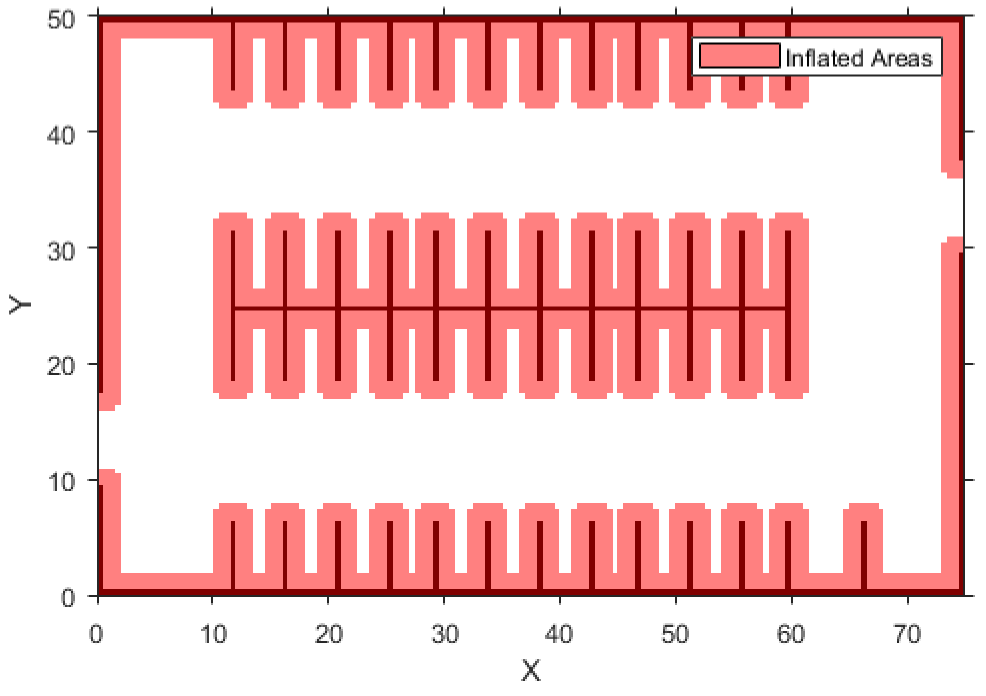
Examples

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.

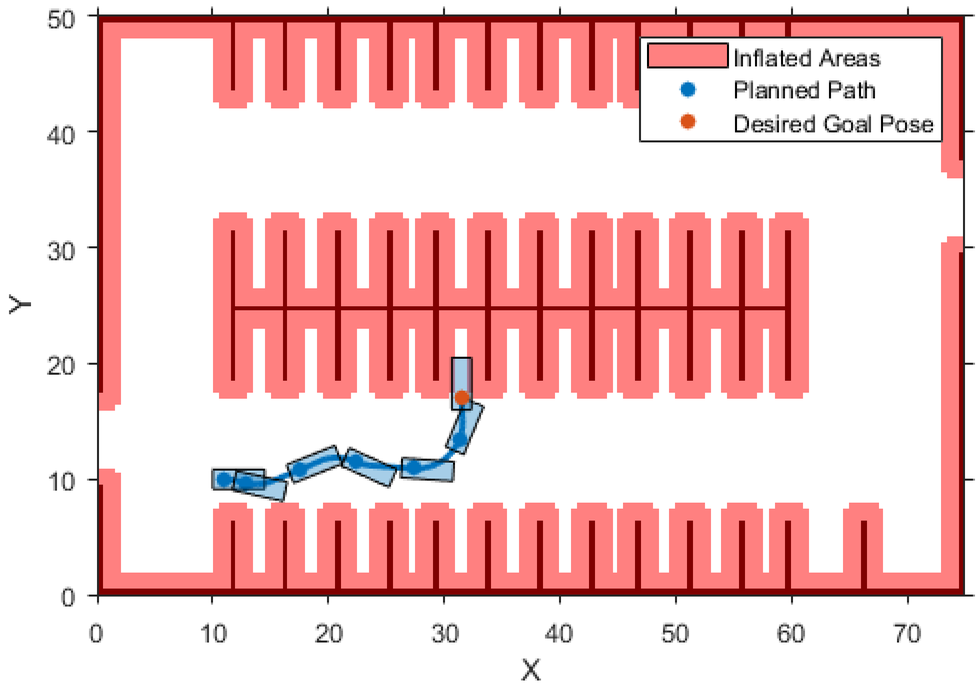
```
startPose = [11, 10, 0]; % [meters, meters, degrees]
goalPose  = [31.5, 17, 90];
```

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)
```

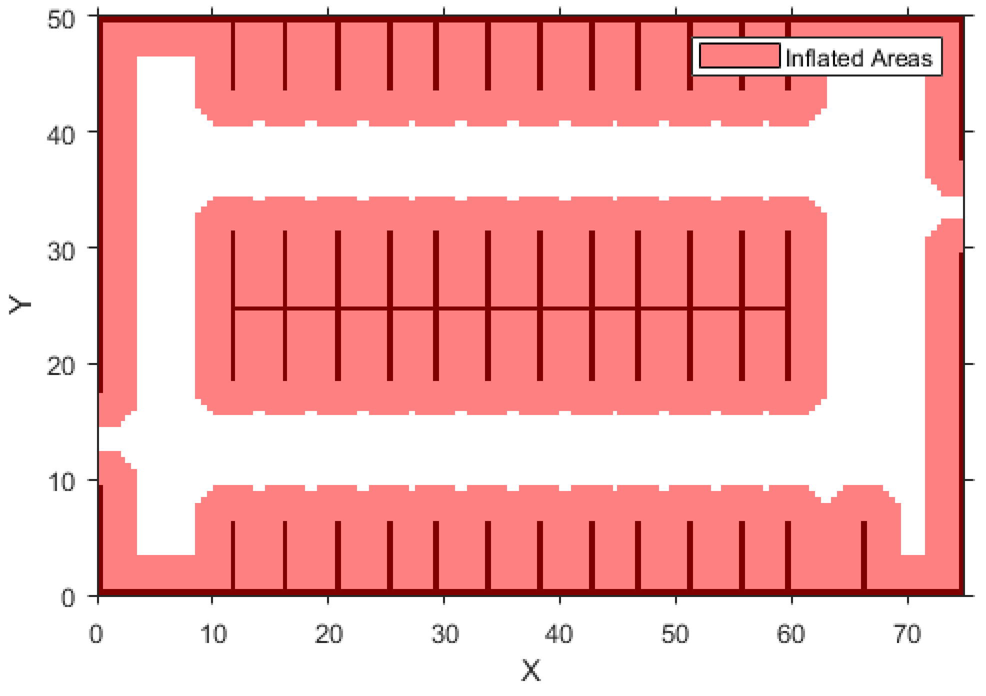


Plan Path and Check Its Validity

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)
```

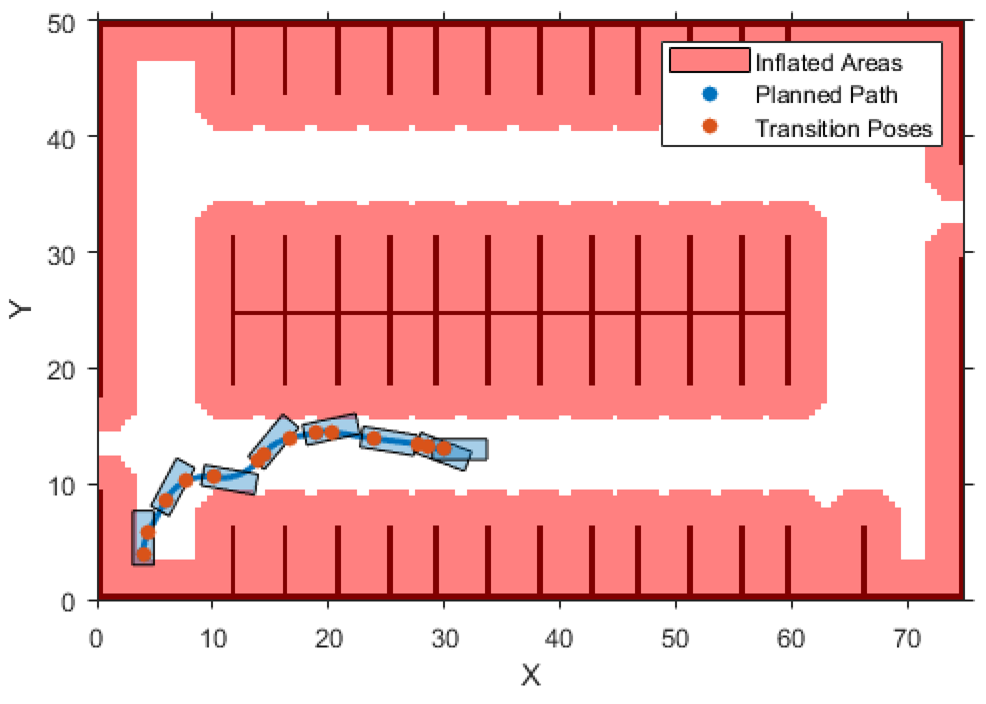
```
isPathValid = logical
   1
```

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.

See Also

Functions
`checkPathValidity` | `lateralControllerStanley` | `plan` | `plot`

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`.

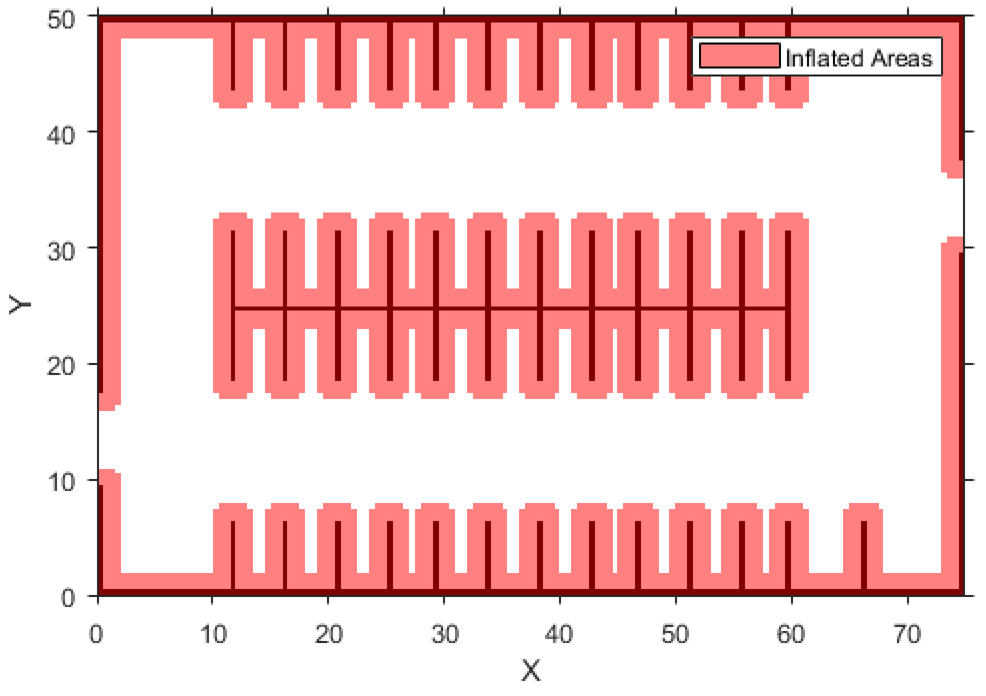
Examples

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.

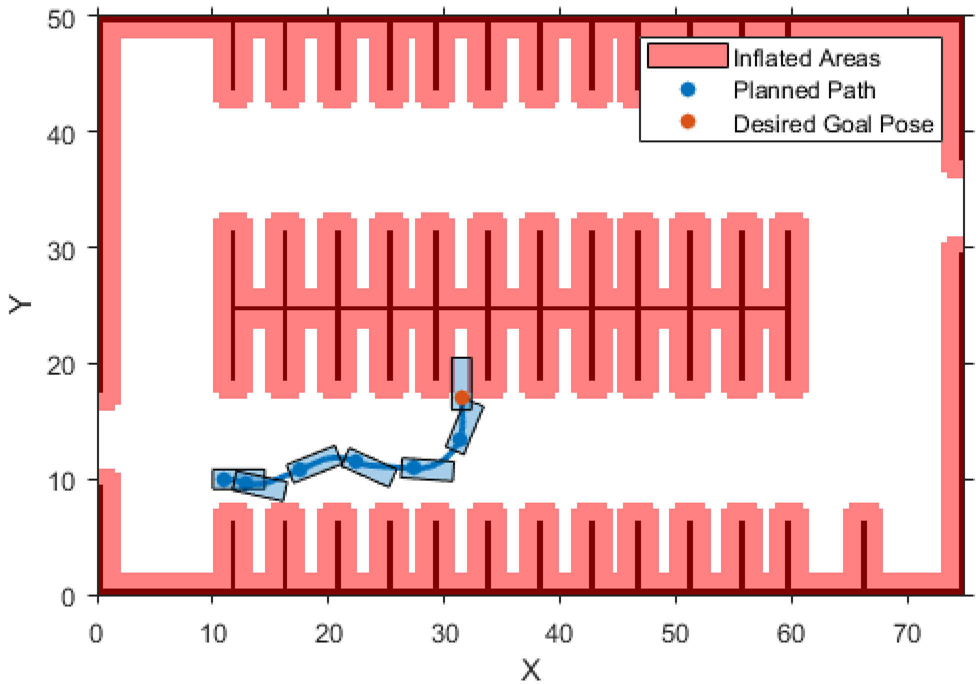
```
startPose = [11, 10, 0]; % [meters, meters, degrees]
goalPose  = [31.5, 17, 90];
```

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.

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.

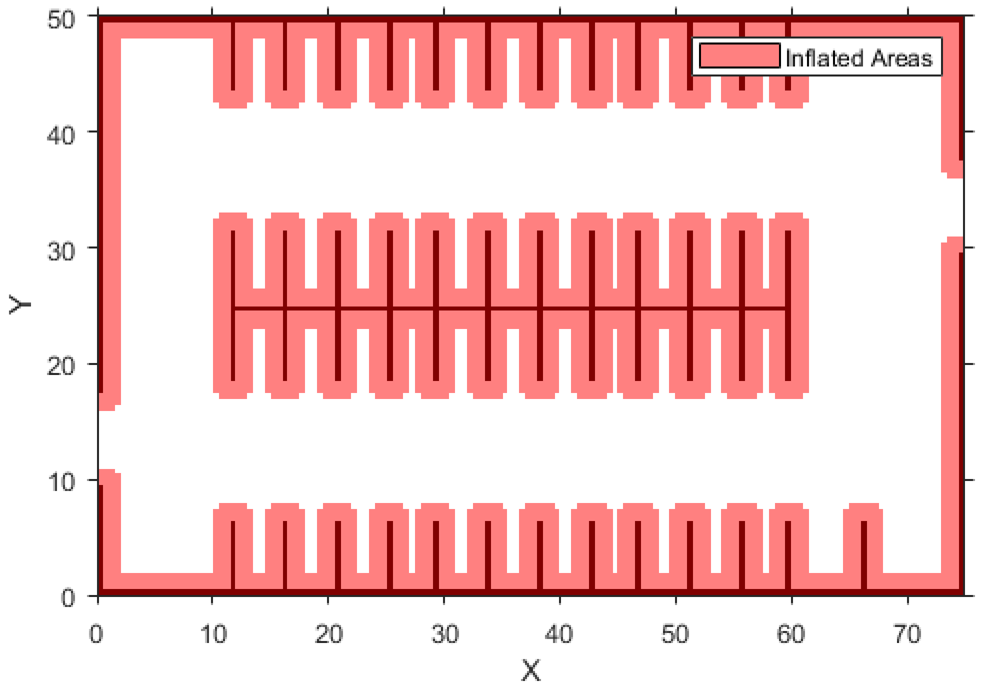
Examples

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.

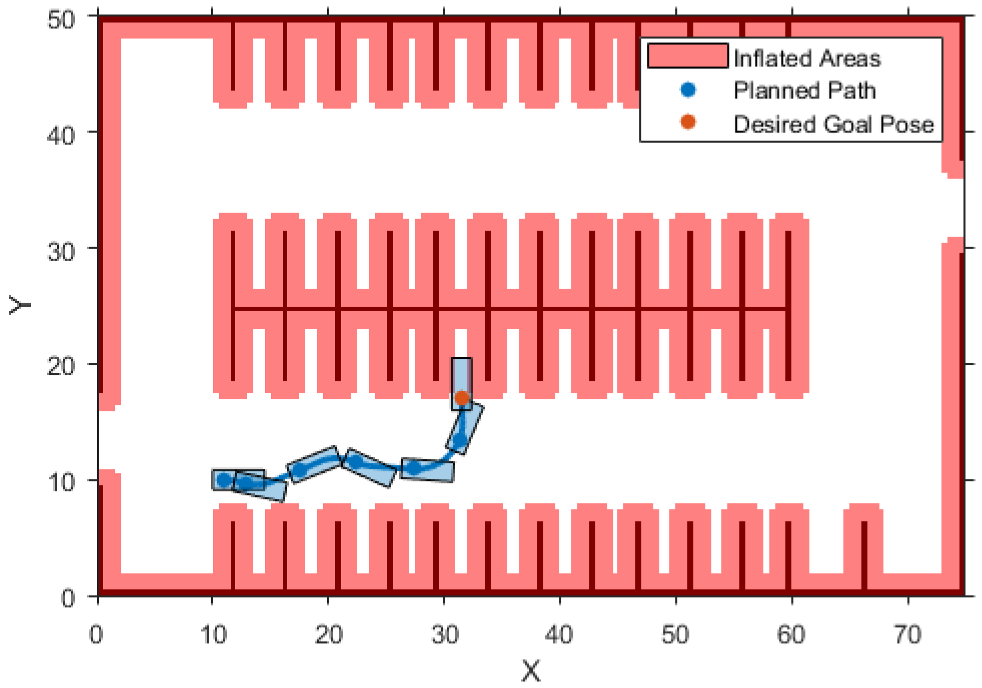
```
startPose = [11, 10, 0]; % [meters, meters, degrees]
goalPose  = [31.5, 17, 90];
```

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

lanespec class

Create road lane specifications

Description

The `lanespec` object defines road lane specifications used in the `road` method of the `drivingScenario` class.

Construction

`lnspec = lanespec(numlanes)` returns lane specifications for a road having `numlanes` lanes. All other properties take default values.

`lnspec = lanespec(numlanes,Name,Value)` returns lane specifications for a road having `numlanes` lanes. You can specify additional options using one or more `Name,Value` pair arguments. `Name` can also be a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`numlanes` — Number of lanes in road
positive integer | positive integer-valued 1-by-2 vector (N_L,N_R)

Number of lanes in the road, specified as a positive integer or a vector of positive integers of the form $[N_L,N_R]$. When `numlanes` is a scalar, all lanes flow in the same direction. When `numlanes` is a vector, the first entry is the number of lanes to the left and the number of lanes to the right. The total number of lanes is the sum, $N = N_L + N_R$. For the definitions of left and right, see “Meaning of Left and Right” on page 4-593.

Example: `[2 2]`

Data Types: `double`

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: `'Width',[3.5,3.7,3.7,3.5]`

`Width` — Lane widths

`3.6` (default) | positive scalar | 1-by-*N* vector of positive values

Lane widths, specified as a positive scalar or 1-by-*N* vector of positive values. *N* is the number of lanes defined by `numlanes`.

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 marking

lane marking object (default) | 1-by-*M* array of lane marking objects

Lane markings, specified as a `laneMarking` object or a 1-by-*M* array of `laneMarking` objects *N* lanes have $M = N + 1$ lane markings.

By default, for a one way road, the color of the lane marking of the leftmost lane is yellow. For two way roads, the color of the dividing lane marker is yellow.

Outputs

`lnspec` — Lane specification

lane specification object

Lane specification, returned as a `lanespec` lane specification object with these properties.

`NumLanes` -- The number of lanes specified by the `numlanes` argument.
`Width` -- The lane widths specified by the `'Width'` Name,Value pair.
`Marking` -- Lane markings specified by the `'Marking'` Name,Value pair.

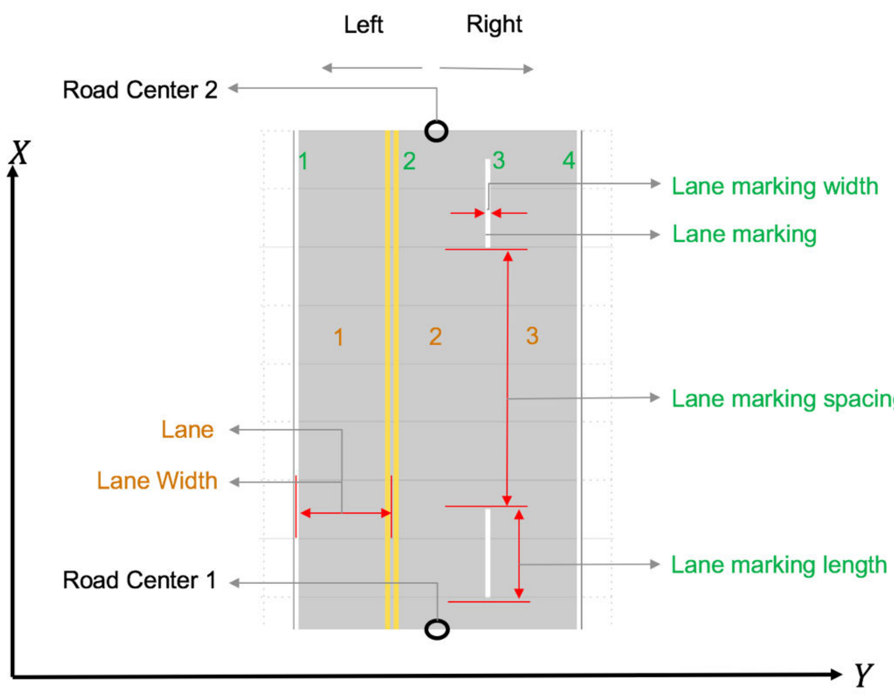
Limitations

- 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. There can only be one specification for a road.

Definitions

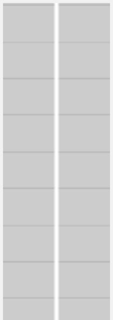
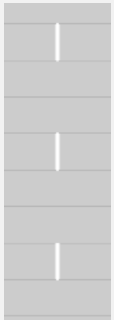
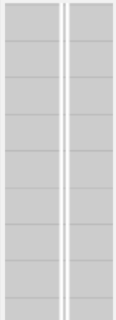
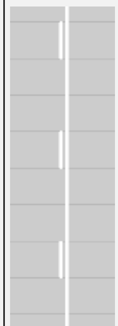
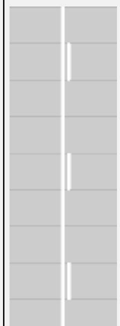
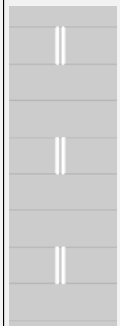
Lane Markings

This figure illustrates the lane marking geometric properties:



This figure illustrates the types of lane markings used in driving scenarios:

Lane Boundary Markings

Solid	Dashed	DoubleSolid	DashedSolid	SolidDashed	DoubleDashed
					

Meaning of Left and Right

Left and *right* are defined with respect to the road centers specified by the matrix of `roadCenters` input to the `road` method. The road centers create a directed line starting from the first row to the last row of the matrix. Left and right mean left and right of the directed line. The width of the road is the sum of all lane widths plus half the widths of the left-edge and right-edge boundary markings.

Examples

Create Straight Four-Lane Road

Construct a straight road with two lanes in each direction.

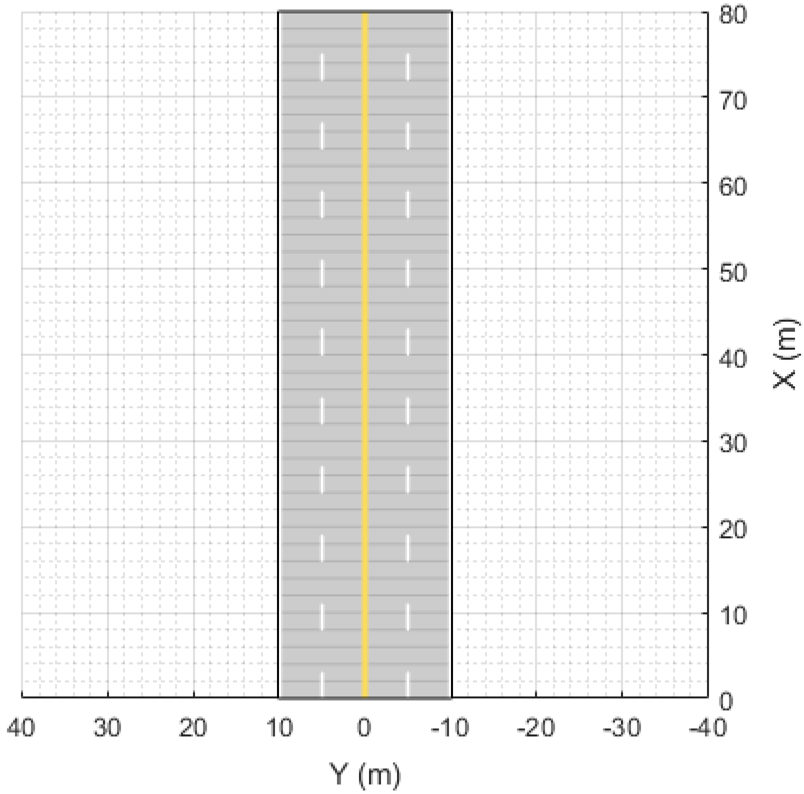
Create a lanespec object from lane marking objects. A four-lane road has five lane markings. The center line is a double-yellow line. The outermost lines are solid white lines while the inner lines are dashed.

```
sc = drivingScenario;
roadCenters = [0 0; 80 0];
solid_w = laneMarking('Solid','Width',0.3);
dash_w = laneMarking('Dashed','Space',5);
```

```
double_y = laneMarking('DoubleSolid','Color','yellow');
lspec = lanespec([2 2],'Width',[5,5,5,5],'Marking',[solid_w,dash_w,double_y,dash_w,soli
```

Display the road.

```
road(sc,roadCenters,'Lanes',lspec);
plot(sc)
```



Simulate Car Traveling on S-Curve

Simulate a driving scenario with one car traveling through an *S*-curve. Create and plot the lane boundaries.

Create the scenario with one road having an *S*-curve.

```
sc = 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.

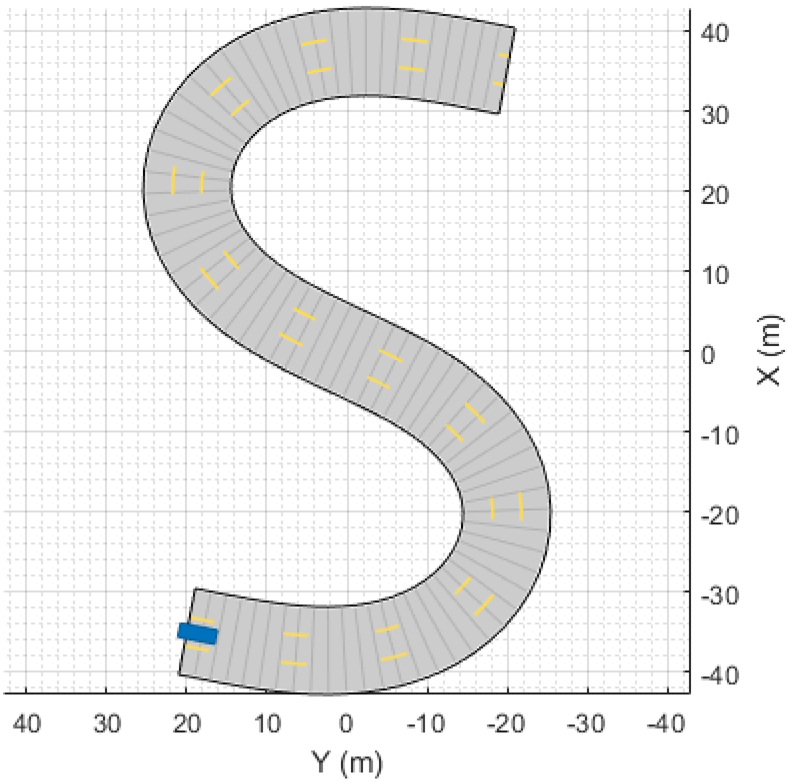
```
lm = [laneMarking('Solid','Color','w'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Solid','Color','w')];
ls = lanespec(3,'Marking',lm);
road(sc, roadCenters,'Lanes',ls);
```

Add an ego car and specify its trajectory from its speed and waypoints. The car travels at 30 m/s.

```
car = vehicle(sc, ...
    'ClassID', 1, ...
    'Position', [-35 20 0]);
waypoints = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
speed = 30;
trajectory(car,waypoints,speed);
```

Plot the scenario and corresponding chase plot.

```
plot(sc)
```



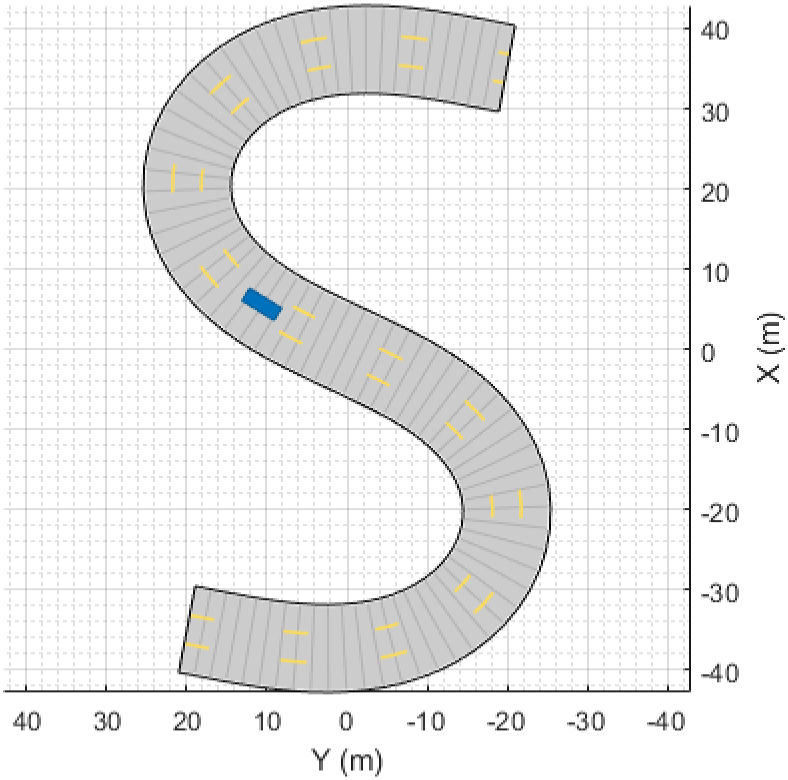

```
chasePlot(car)
```

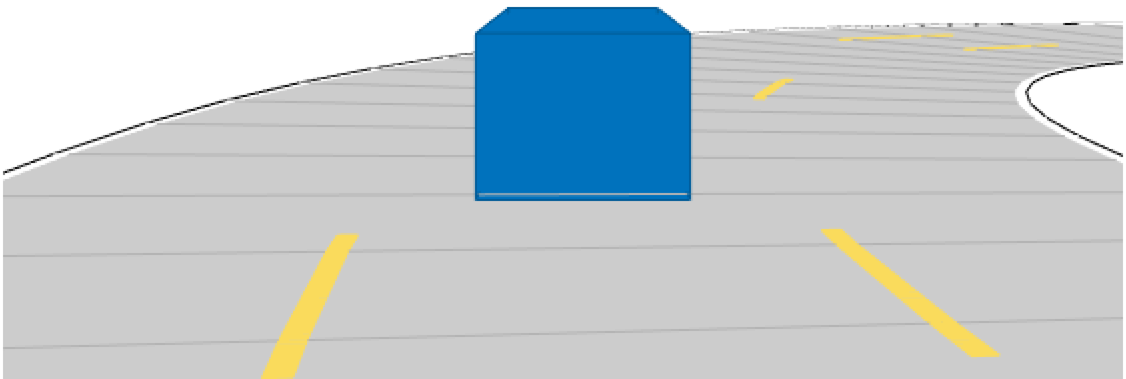
Run the simulation loop.

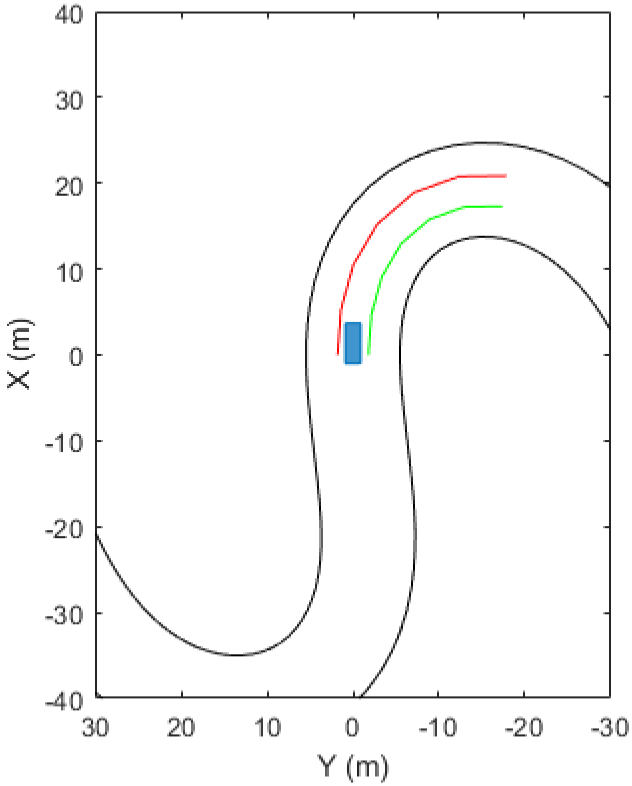
1. Initialize a bird's-eye plot and create an outline plotter, left-lane and right-lane boundary plotters, and a road boundary plotter.
2. Obtain the road boundaries and rectangular outlines.
3. Obtain the lane boundaries to the left and right of the vehicle.
4. Advance the simulation and update the 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(sc)
    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
```







See Also

`drivingScenario` | `laneMarking` | `road`

Introduced in R2018a

laneMarking class

Create road lane marking object

Description

The `laneMarking` class specifies the properties of lane markings which define the lane boundary lines on roads. You can use lane marking objects as input to the `lanespec` object when creating roads.

Construction

`lanemarking = laneMarking(Type)` returns a lane marking object, `lanemarking`, with default properties for the lane boundary type, `Type`.

`lanemarking = laneMarking(Type,Name,Value)` returns a lane marking object, `lanemarking`, with properties specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Output Arguments

`lanemarking` — Lane marking
laneMarking object

Lane marking, returned as a `laneMarking` object. A `laneMarking` object defines the characteristics of a lane boundary marker on a road.

Properties

Type — Type of lane boundary marker
`'Unmarked'` | `'Solid'` | `'Dashed'` | `'DoubleSolid'` | `'DoubleDashed'` | `'SolidDashed'` | `'DashedSolid'`

Lane boundary type, specified as one of the `LaneBoundaryType` enumerations: `'Unmarked'`, `'Solid'`, `'Dashed'`, `'DoubleSolid'`, `'DoubleDashed'`, `'SolidDashed'`, or `'DashedSolid'`. The lane boundary markers correspond to different types of lines painted on a road.

Example: `'DoubleSolid'`

Data Types: `char` | `string`

`Width` — Lane marking widths
`0.15` (default) | positive scalar

Lane marking widths, specified as a positive scalar. For a double lane marker, the same width is used for both lines. Units are in meters.

Example: `0.20`

Data Types: `double`

`Color` — Boundary line color
`[1 1 1]` (white) (default) | MATLAB color string | `[r g b]` vector

Boundary line color, specified as a MATLAB color string or as an `[r g b]` vector. For a double lane marker, the same color is used for both lines.

Example: `[.8 .8 .8]`

Data Types: `double` | `char` | `string`

`Strength` — Visibility of lane marking
`1` (default) | positive scalar from 0 to 1

Visibility of lane marking, specified as a scalar from 0 through 1. A value of 0 corresponds to a marking that is not visible and a value of 1 corresponds to a marking that is completely visible. Values in between are partially visible. For a double lane marker, the same strength is used for both lines.

Example: `0.20`

Data Types: `double`

`Length` — Length of dash in dashed lines
`3.0` (default) | positive scalar

Length of dash in dashed lines, specified as a positive scalar. For a double lane marker, 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`

Data Types: `double`

Space — Length of space between dashes in dashed lines
`9.0` (default) | positive scalar

Length of space between the end of a dash in a dashed line and beginning of the next dash, specified as a positive scalar. For a double lane marker, the same length is used for both lines. Units are in meters.

Example: `2.0`

Data Types: `double`

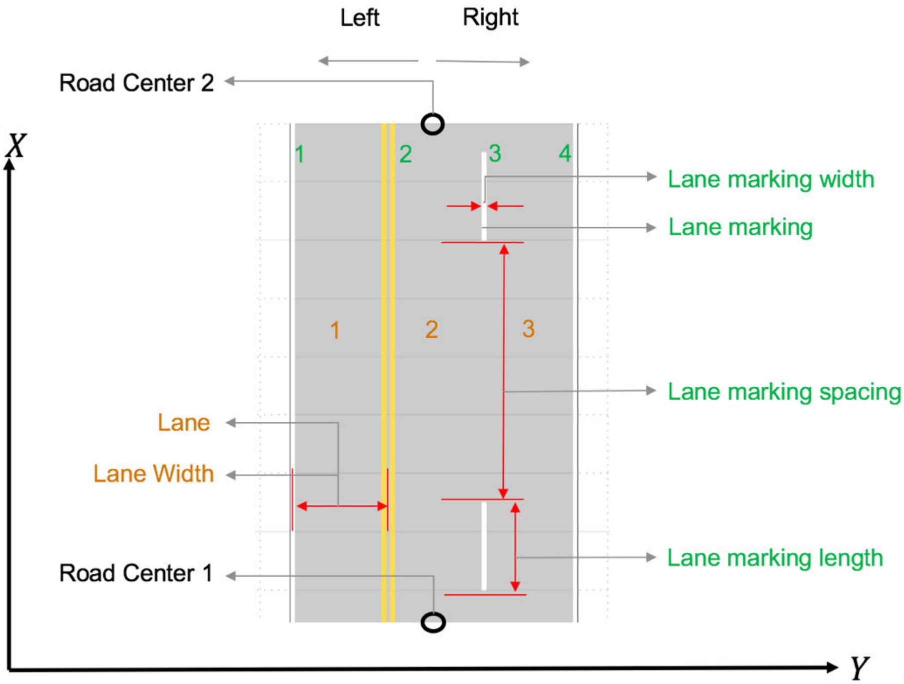
Limitations

- 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. There can only be one specification for a road.

Definitions

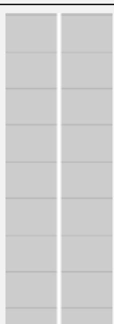
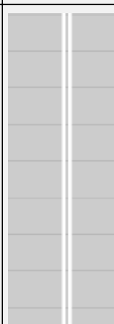
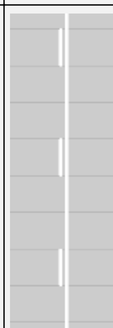
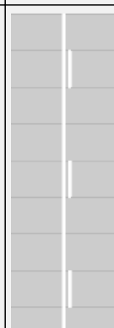
Lane Markings

This figure illustrates the lane marking geometric properties:



This figure illustrates the types of lane markings used in driving scenarios:

Lane Boundary Markings

Solid	Dashed	DoubleSolid	DashedSolid	SolidDashed	DoubleDashed
					

Examples

Create Straight Four-Lane Road

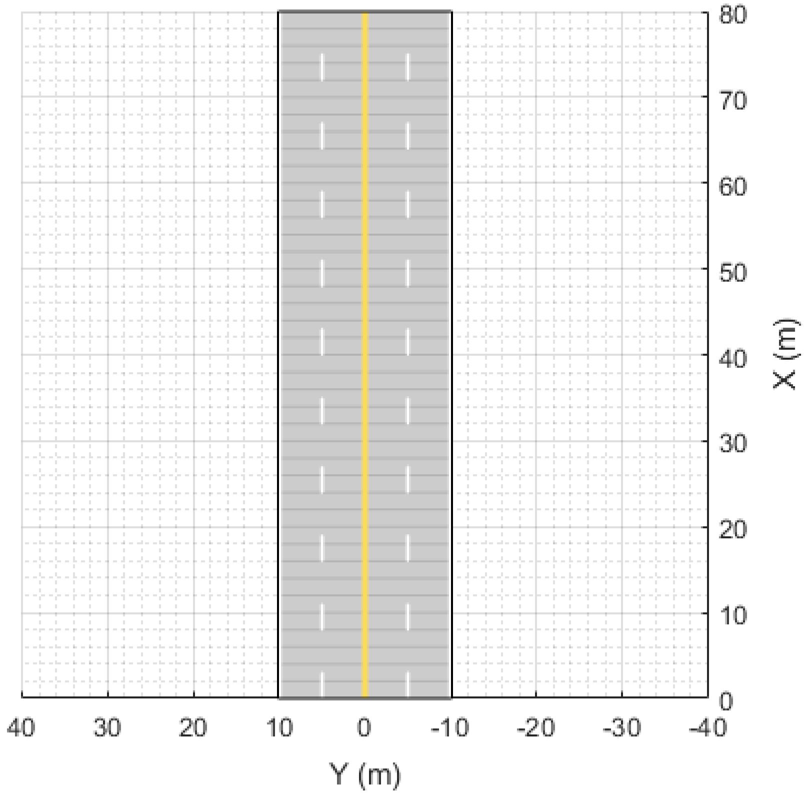
Construct a straight road with two lanes in each direction.

Create a lanespec object from lane marking objects. A four-lane road has five lane markings. The center line is a double-yellow line. The outermost lines are solid white lines while the inner lines are dashed.

```
sc = drivingScenario;
roadCenters = [0 0; 80 0];
solid_w = laneMarking('Solid','Width',0.3);
dash_w = laneMarking('Dashed','Space',5);
double_y = laneMarking('DoubleSolid','Color','yellow');
lspec = lanespec([2 2],'Width',[5,5,5,5],'Marking',[solid_w,dash_w,double_y,dash_w,sol
```

Display the road.

```
road(sc,roadCenters,'Lanes',lspec);
plot(sc)
```



Simulate Car Traveling on S-Curve

Simulate a driving scenario with one car traveling through an *S*-curve. Create and plot the lane boundaries.

Create the scenario with one road having an *S*-curve.

```
sc = 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.

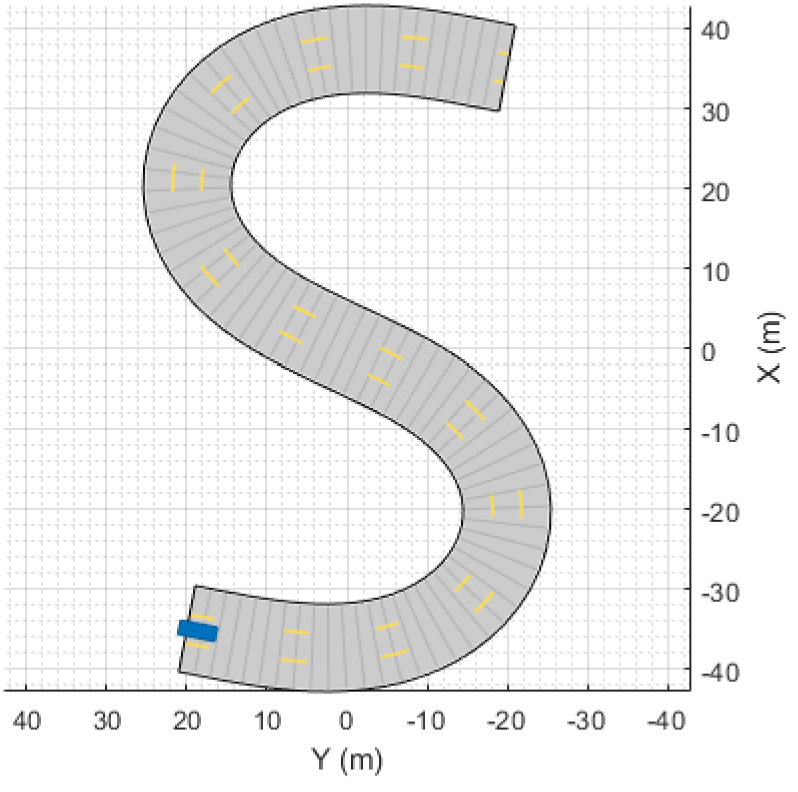
```
lm = [laneMarking('Solid','Color','w'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Solid','Color','w')];
ls = lanespec(3,'Marking',lm);
road(sc, roadCenters,'Lanes',ls);
```

Add an ego car and specify its trajectory from its speed and waypoints. The car travels at 30 m/s.

```
car = vehicle(sc, ...
    'ClassID', 1, ...
    'Position', [-35 20 0]);
waypoints = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
speed = 30;
trajectory(car,waypoints,speed);
```

Plot the scenario and corresponding chase plot.

```
plot(sc)
```



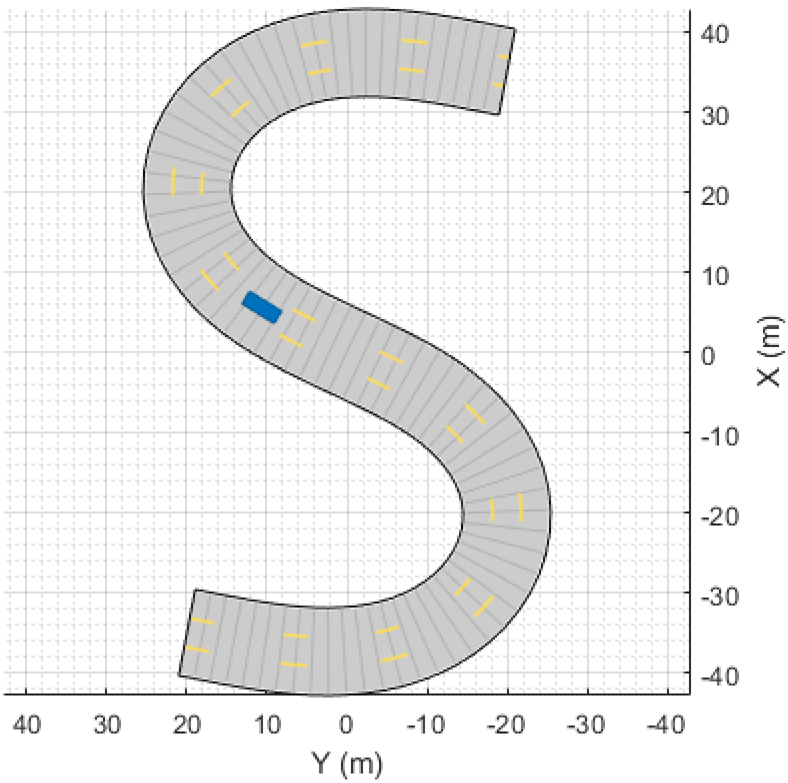

```
chasePlot(car)
```

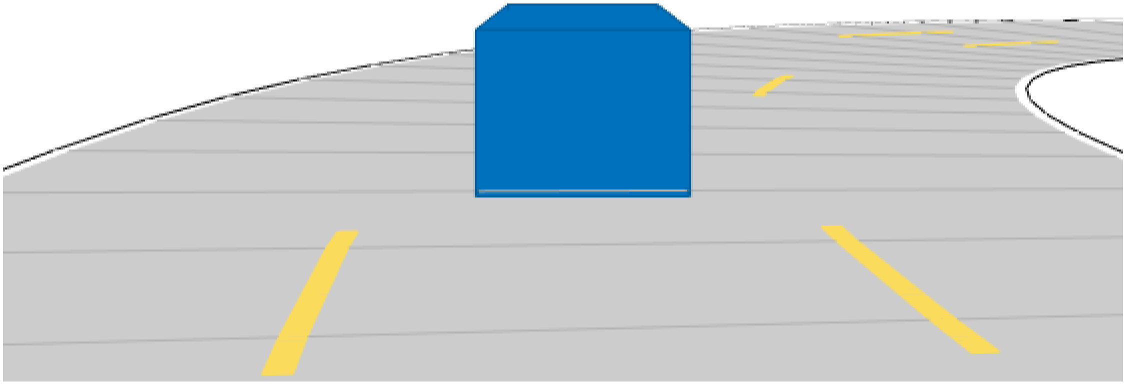
Run the simulation loop.

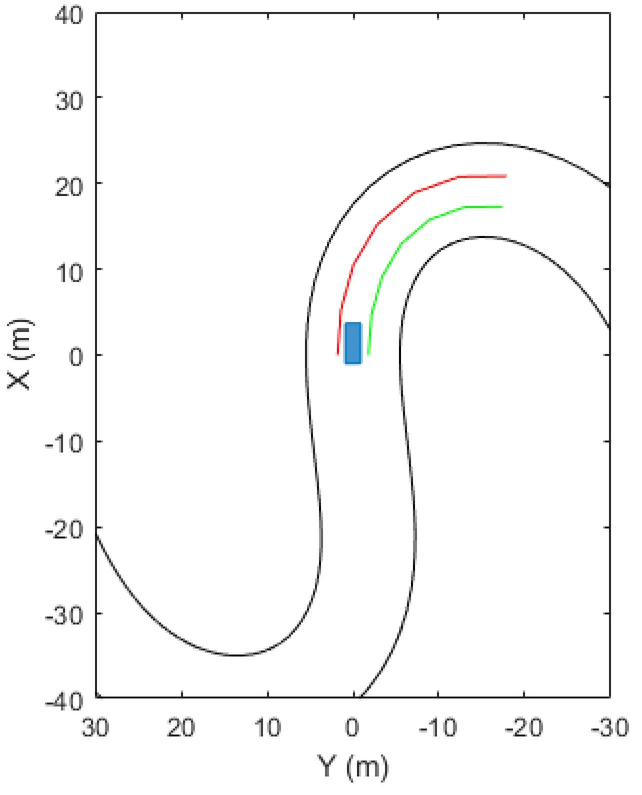
1. Initialize a bird's-eye plot and create an outline plotter, left-lane and right-lane boundary plotters, and a road boundary plotter.
2. Obtain the road boundaries and rectangular outlines.
3. Obtain the lane boundaries to the left and right of the vehicle.
4. Advance the simulation and update the 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(sc)
    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
```







See Also

`drivingScenario` | `lanespec` | `road`

Introduced in R2018a

laneMarkingVertices

Class: drivingScenario

Lane marking vertices and faces

Syntax

```
[lmv,lmf] = laneMarkingVertices(sc)
[lmv,lmf] = laneMarkingVertices(ac)
```

Description

`[lmv,lmf] = laneMarkingVertices(sc)` returns lane marking vertices, `lmv`, and lane marking faces, `lmf`, in driving scenario, `sc`, coordinates. Use lane marking vertices and faces to display lane markings in `laneMarkingPlotter`.

`[lmv,lmf] = laneMarkingVertices(ac)` returns lane marking vertices, `lmv`, and lane marking faces, `lmf`, in the coordinates of the actor, `ac`.

Input Arguments

`sc` — Driving scenario
`drivingScenario` object

Driving scenario, specified as a `drivingScenario` object.

Example: `sc = drivingScenario`

ac — Scenario actor
`Actor` object | `Vehicle` object

Scenario actor, specified as an `Actor` or `Vehicle` object. To create actors, use the `actor` or `vehicle` method.

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*-, and *z*- coordinates of a vertex. Lane marking vertices are defined in `patch`.

`lmf` — Lane marking faces
real-valued matrix

Lane marking faces, returned as a real-valued matrix. Each row of the matrix is a face that defines the connection between vertices for one lane marking. Lane marking faces are defined in `patch`.

Examples

Plot Lane Markings in Car and Pedestrian Scenario

Construct a driving scenario containing a car and pedestrian on a straight road. Then, create and display lane markings in a bird's-eye plot.

Create an empty driving scenario.

```
sc = drivingScenario;
```

Construct a straight road segment 25 m in length with two travel lanes in one direction.

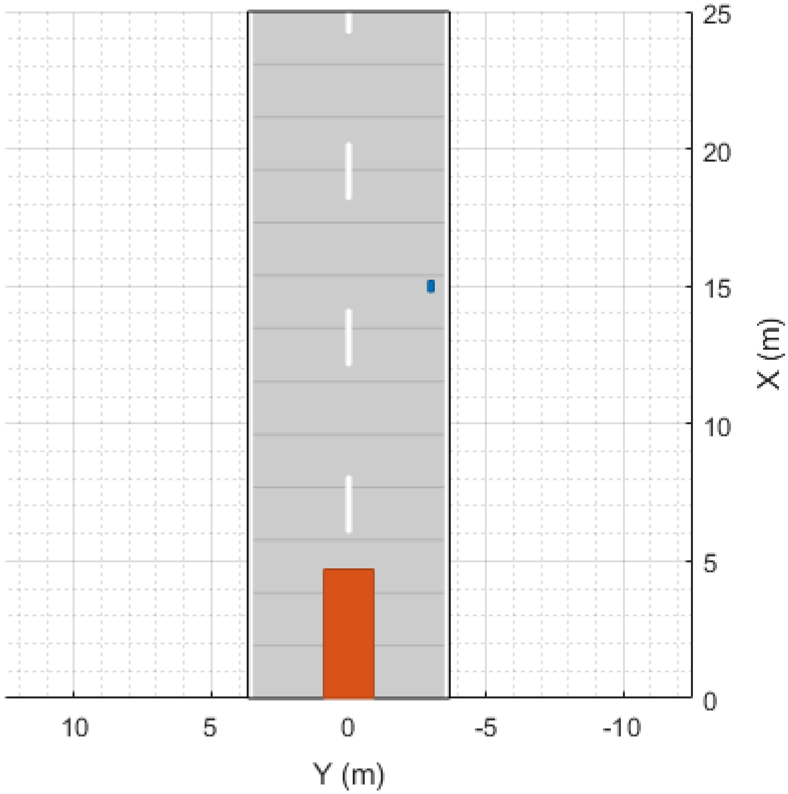
```
lm = [laneMarking('Solid')
    laneMarking('Dashed','Length',2,'Space',4)
    laneMarking('Solid')];
l = lanespec(2,'Marking',lm);
road(sc, [0 0 0; 25 0 0],'Lanes',l);
```

Add a pedestrian crossing the road at 1 m/s and a car following the road at 10 m/s.

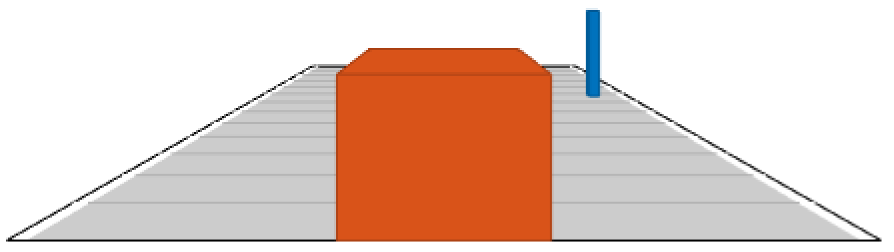
```
ped = actor(sc, 'Length', 0.2, 'Width', 0.4, 'Height', 1.7);
car = vehicle(sc);
trajectory(ped,[15 -3 0; 15 3 0], 1);
trajectory(car,[car.RearOverhang 0 0; 25-car.Length+car.RearOverhang 0 0], 10);
```

Display the scenario and corresponding chase plot.

```
plot(sc)
```




```
chasePlot(car)
```

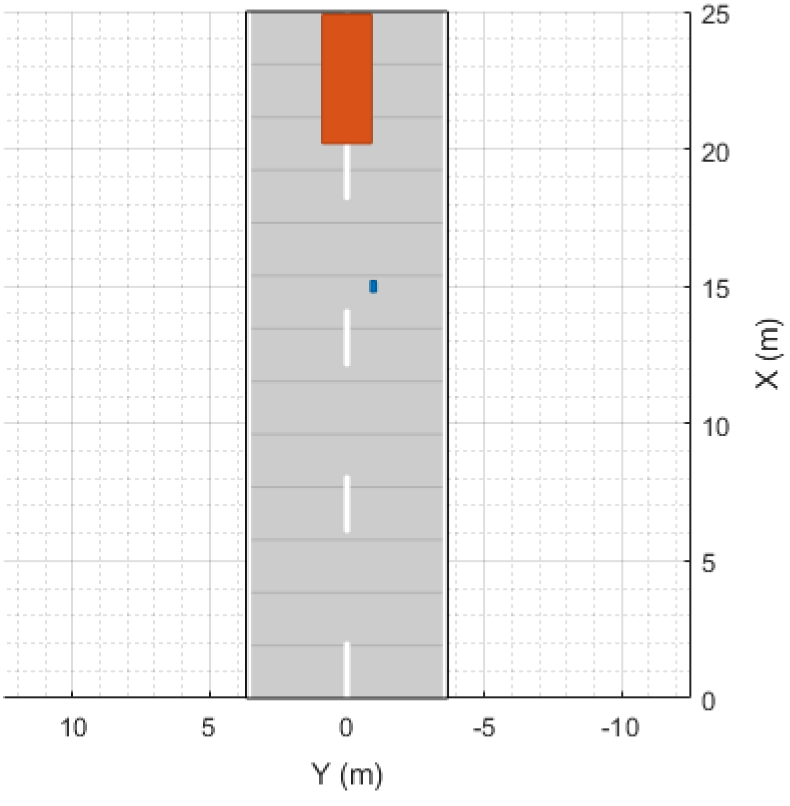


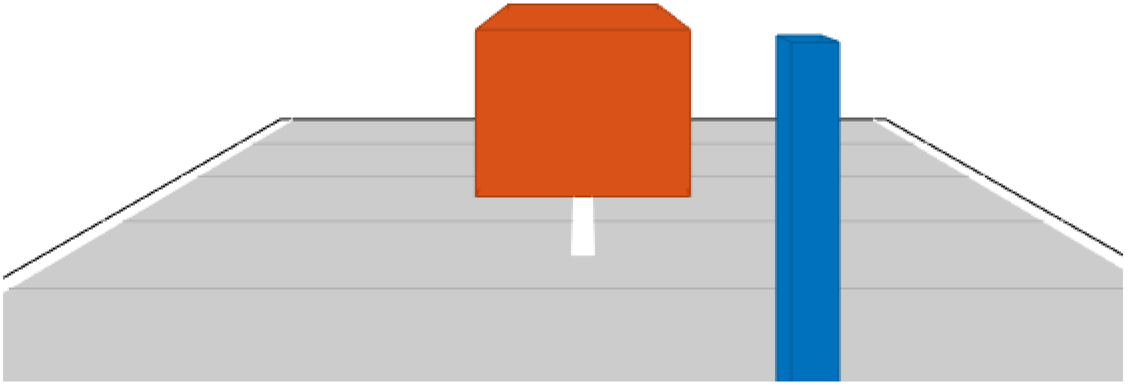
Run the simulation.

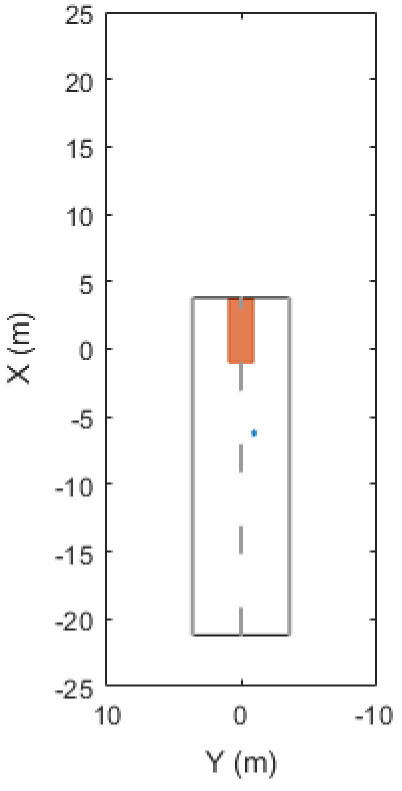
- Create the bird's eye plot and add an outline plotter, a lane boundary plotter and lane marking plotter.
- Get the road boundaries and target outlines.
- Get lane marking vertices and faces.
- Plot the boundaries and lane markers.
- Run the simulation loop.

```
bep = birdsEyePlot('XLim',[-25 25],'YLim',[-10 10]);
olPlotter = outlinePlotter(bep);
lbPlotter = laneBoundaryPlotter(bep);
```

```
lmPlotter = laneMarkingPlotter(bep,'DisplayName','Lanes');
legend('off');
while advance(sc)
    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
```







See Also

patch | laneMarking | laneMarkingPlotter | plotLaneMarking

Introduced in R2018a

laneBoundaries

Lane boundaries

Syntax

```
lbdry = laneBoundaries(ac)
lbdry = laneBoundaries(ac,Name,Value)
```

Description

`lbdry = laneBoundaries(ac)` returns the lane boundaries, `lbdry`, defined with respect to coordinates of the ego actor, `ac`.

`lbdry = laneBoundaries(ac,Name,Value)` specifies additional options using one or more `Name,Value` pair arguments. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Examples

Simulate Car Traveling on S-Curve

Simulate a driving scenario with one car traveling through an *S*-curve. Create and plot the lane boundaries.

Create the scenario with one road having an *S*-curve.

```
sc = 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'); ...
```

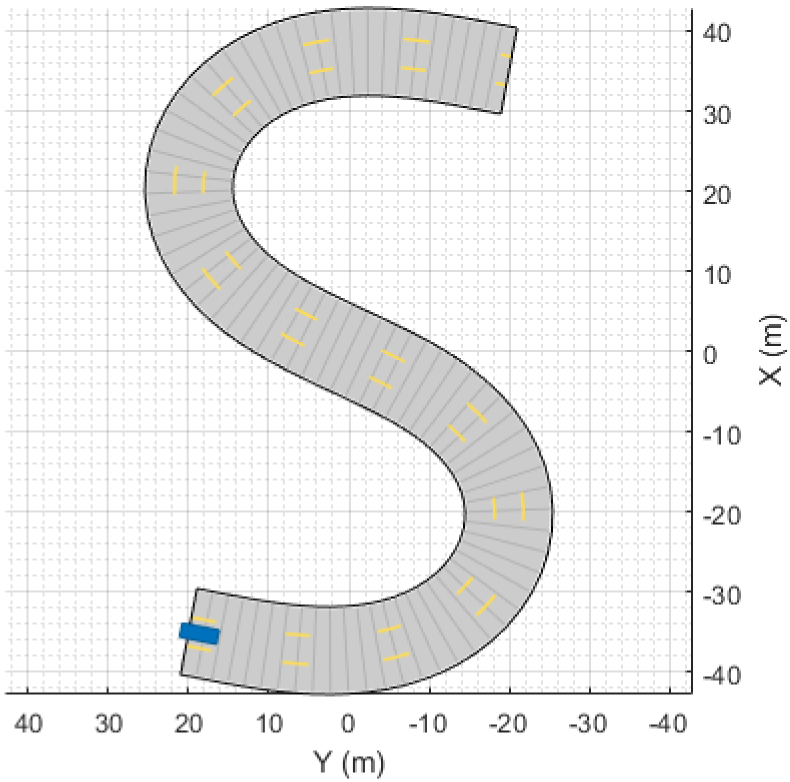
```
    laneMarking('Dashed','Color','y'); ...
    laneMarking('Solid','Color','w')];
ls = lanespec(3,'Marking',lm);
road(sc, roadCenters,'Lanes',ls);
```

Add an ego car and specify its trajectory from its speed and waypoints. The car travels at 30 m/s.

```
car = vehicle(sc, ...
    'ClassID', 1, ...
    'Position', [-35 20 0]);
waypoints = [-35 20 0; -20 -20 0; 0 0 0; 20 20 0; 35 -20 0];
speed = 30;
trajectory(car,waypoints,speed);
```

Plot the scenario and corresponding chase plot.

```
plot(sc)
```



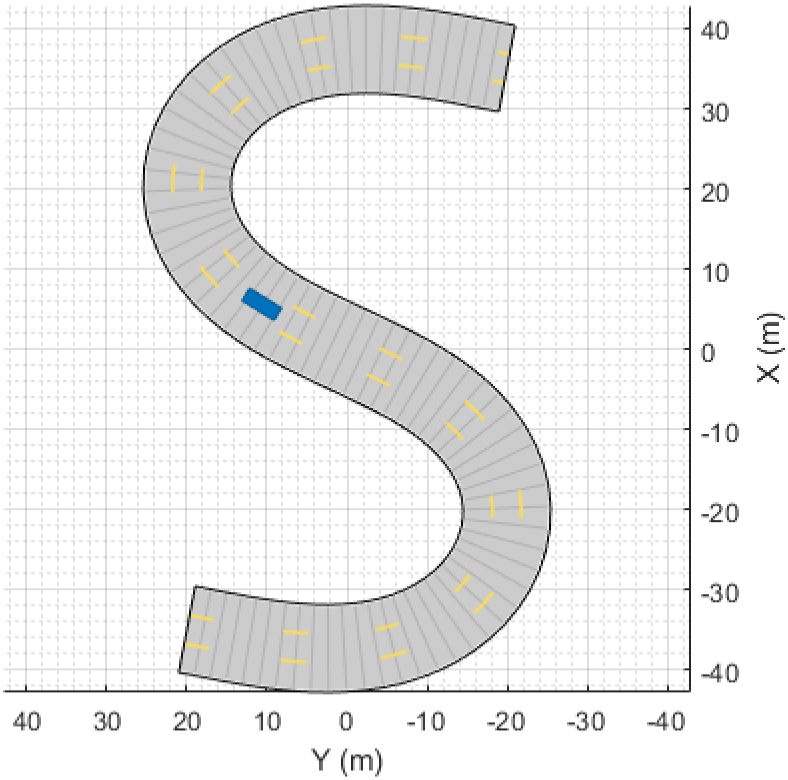

```
chasePlot(car)
```

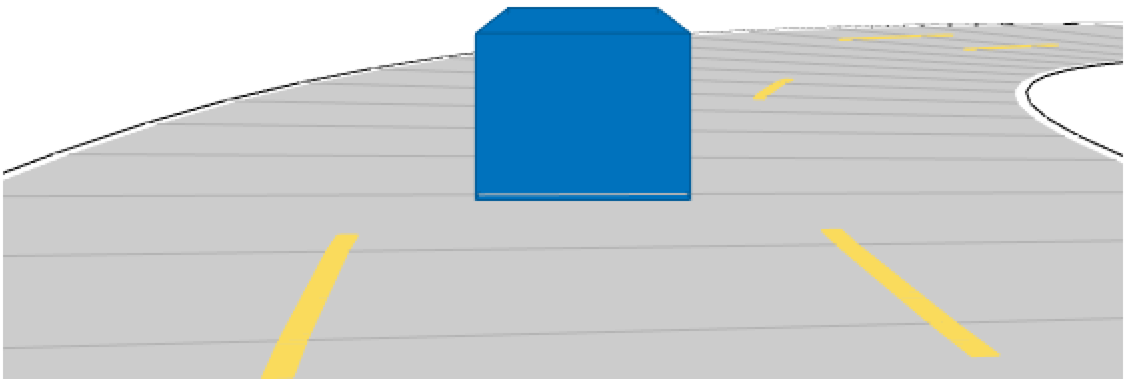
Run the simulation loop.

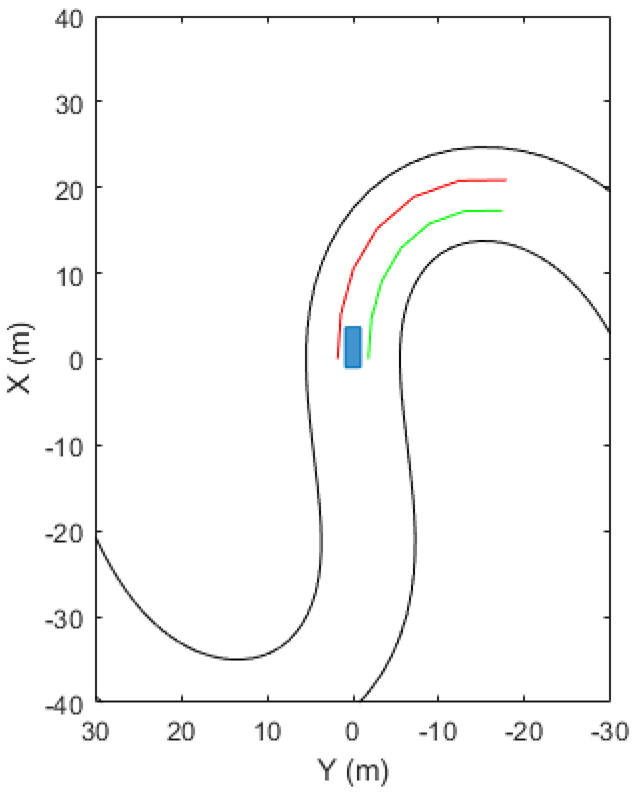
1. Initialize a bird's-eye plot and create an outline plotter, left-lane and right-lane boundary plotters, and a road boundary plotter.
2. Obtain the road boundaries and rectangular outlines.
3. Obtain the lane boundaries to the left and right of the vehicle.
4. Advance the simulation and update the 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(sc)
    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
```







Input Arguments

ac — Scenario actor
`Actor` object | `Vehicle` object

Scenario actor, specified as an `Actor` or `Vehicle` object. To create actors, use the `actor` or `vehicle` method.

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'`

`XDistance` — Distances ahead of ego actor
`0` (default) | *N*-element real-valued vector

Distances ahead of the ego actor position along the road at which to determine the lane boundaries, specified as an *N*-element real-valued vector.

Example: `1:0.1:10`

Data Types: `double`

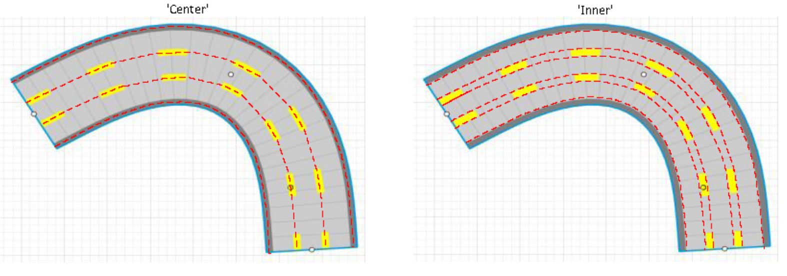
`LocationType` — Lane boundary location
`'Center'` (default) | `'Inner'`

Lane boundary location, specified as `'Center'` or `'Inner'`. For `'Center'`, returned boundaries are centered on the lane markings. For `'Inner'`, boundaries are placed at the inner edges of the lane markings.

Consider a three-lane road with four lane markings. Two lane markings are at the road edges. The other two lane markings divide the road into its three lanes.

- When `LocationType` is `'Center'`, the road has four lane boundaries, with one boundary per lane marking.
- When `LocationType` is `'Inner'`, the road has six lane boundaries, with two boundaries for each of the three lanes.

The following figure illustrates the two types of lane boundary locations.



Example: `'Inner'`

Data Types: `char` | `string`

`AllBoundaries` — Return lane boundary locations
`false` (default) | `true`

Return all lane boundary locations, specified as `false` or `true`. Lane boundaries are returned from left to right relative to the ego vehicle. When false, only the left and right lane boundaries next to the ego vehicle are returned.

Data Types: `logical`

Output Arguments

`lbdry` — Lane boundaries
array of structures

Lane boundaries, returned as an array of lane boundary structure fields defined in the table.

Lane Boundary Structure Fields

Field	Description
`Coordinates`	Lane boundary coordinates, specified as a real-valued *N*-by-3 matrix. Lane boundary coordinates define the position of points on the boundary at distances specified by `XDistance`. In addition, a set of boundary coordinates are inserted into the matrix at zero distance. 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 rows in the `Coordinates` matrix. Units are in degrees/m.
`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 rows in the `Coordinates` matrix. Units are in degrees/m. Units are in degrees/m^2.
`HeadingAngle`	Initial lane boundary heading, specified as a scalar. The heading angle of the lane boundary is relative to the ego car heading. Units are in degrees.
`LateralOffset`	Distance of the lane boundary from the ego vehicle position, specified as a scalar. An offset to a lane boundary to the left of the ego 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 the following: • `'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`	Strength of the lane boundary marking, specified as a scalar from 0 through 1. A value of 0 corresponds to a marking that is not visible and a value of 1 corresponds to a marking that is completely visible. Values in between are partially visible.
`Width`	Lane boundary width, specified as a positive scalar. In a double-line lane marker, the same width is used for both lines and the space between lines. Units are in meters.
`Length`	Length of dash in dashed lines, specified as a positive 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 scalar. In a dashed double-line lane marker the same space is used for both lines

See Also

`drivingScenario` | `laneBoundaryPlotter` | `laneMarking` | `laneMarkingPlotter` | `lanespec` | `plotLaneBoundary` | `plotLaneMarking` | `road`

Introduced in R2018a

clothoidLaneBoundary class

Clothoid-shaped lane boundary model

Description

`clothoidLaneBoundary` defines an object containing a clothoid lane boundary model. A clothoid is a type of curve whose rate of change of curvature varies linearly with distance.

Construction

`bdry = clothoidLaneBoundary` creates a clothoid lane boundary object, `bdry`.

Outputs

`bdry` — Lane boundary
`clothoidLaneBoundary` object

Lane boundary, returned as a `clothoidLaneBoundary` object.

Properties

`Curvature` — Lane boundary curvature
`0` (default) | scalar

Lane boundary curvature, specified as a scalar. This property represents the rate of change of lane boundary direction with respect to distance. Units are in degrees/m.

Example: `-0.1`

Data Types: `single` | `double`

`CurvatureDerivative` — Derivative of lane boundary curvature
`0` (default) | scalar

Derivative of lane boundary curvature, specified as a scalar. This property represents the rate of change of lane curvature with respect to distance. Units are in degrees/m^2.

Example: `0.01`

Data Types: `single` | `double`

`CurvatureLength` — Length of lane boundary along road
`0` (default) | positive scalar

Length of the lane boundary along the road, specified as a positive scalar. Units are in meters.

Example: `25`

Data Types: `single` | `double`

`HeadingAngle` — Initial lane boundary heading
`0` (default) | scalar

Initial lane boundary heading, specified as a scalar. The heading angle of the lane boundary is relative to the ego car heading. Units are in degrees.

Example: `10`

Data Types: `single` | `double`

`LateralOffset` — Distance of lane boundary
`0` (default) | real-valued vector

Distance of the lane boundary from the ego vehicle position, specified as a scalar. A lane boundary offset to the left of the ego is 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
`'Unmarked'` (default) | `'Solid'` | `'Dashed'` | `'DoubleSolid'` | `'DoubleDashed'` | `'SolidDashed'` | `'DashedSolid'`

Type of lane boundary marking, specified as one of the following:

- `'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

Example: 'SolidDashed'

Strength — Strength of lane boundary marking

1 (default) | scalar from 0 to 1

Strength of the lane boundary marking, specified as a scalar from 0 through 1. A value of 0 corresponds to a marking that is not visible and a value of 1 corresponds to a marking that is completely visible. Values in between are partially visible.

Example: 0.9

Data Types: single | double

XExtent — Extent of the lane boundary along x-axis

[0 Inf] (default) | 1-by-2 vector

Extent of the lane boundary along the x-axis, specified as a 1-by-2 vector of the form [Xmin Xmax]. Units are in meters.

Example: [0 100]

Data Types: single | double

Width — Lane boundary width

0 (default) | positive scalar

Lane boundary width, specified as a positive scalar. For a double-line lane marking, this value applies to both lines and the distance between the lines. Units are in meters.

Example: 0.15

Data Types: single | double

Methods

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.

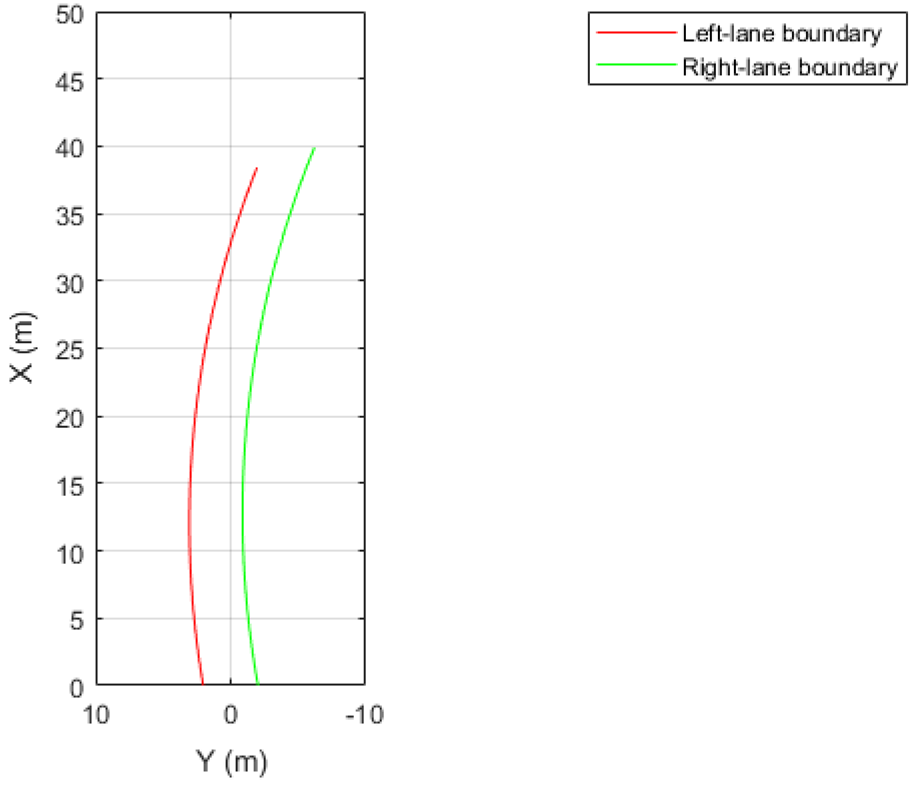
```
lb = clothoidLaneBoundary;
lb.BoundaryType = 'Solid';
lb.Strength = 1;
lb.Width = 0.2;
lb.CurveLength = 40;
lb.Curvature = -0.8;
lb.LateralOffset = 2;
lb.HeadingAngle = 10;
```

Create the right boundary with almost identical properties.

```
rb = lb;
rb.LateralOffset = -2;
```

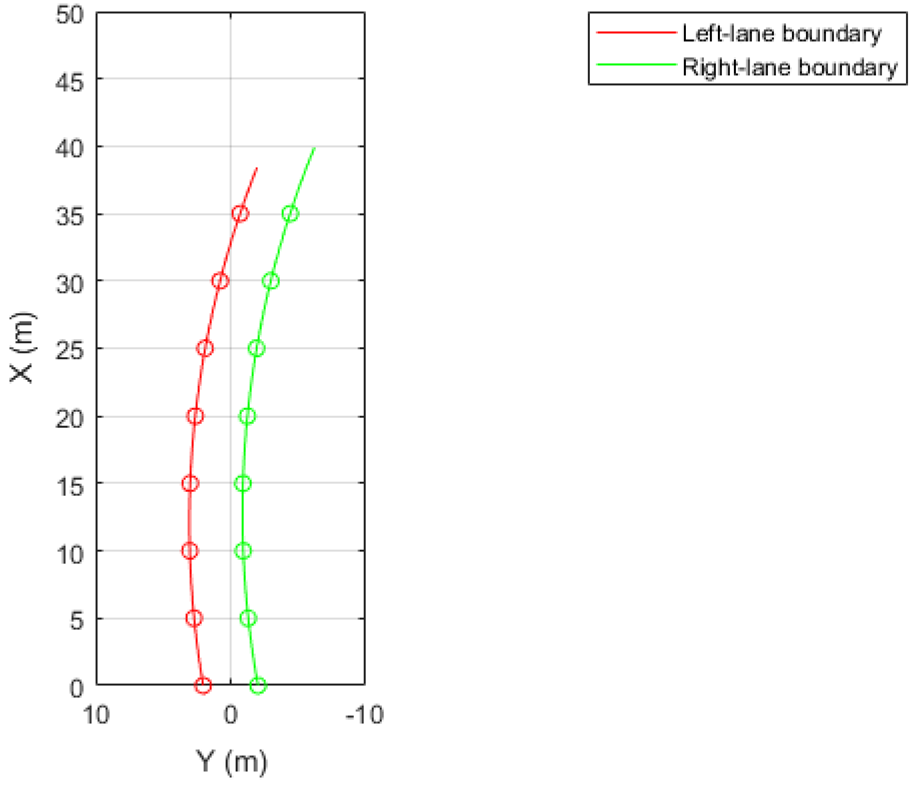
Create a bird's-eye plot. Then, create the lane boundary plotters and plot the boundaries.

```
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

laneBoundaries | laneBoundaryPlotter | laneMarking | lanespec | plotLaneBoundary

Introduced in R2018a

computeBoundaryModel

Class: clothoidLaneBoundary

Compute lane boundary points from clothoid lane boundary model

Syntax

```
yworld = computeBoundaryModel(boundary,xworld)
```

Description

`yworld = computeBoundaryModel(boundary,xworld)` returns lane boundary points, `yworld`, derived from a lane boundary, `boundary`, at points specified by the coordinates, `xworld`. The corresponding *y*-coordinates are returned in `yworld`.

Input Arguments

`boundary` — Lane boundary model
`clothoidLaneBoundary` object

Lane boundary model, specified as a `clothoidLaneBoundary` object.

`xworld` — *x*-world coordinates
N-length real-valued vector

x-world coordinates, specified as a *N*-length real-valued vector.

Example: `2:2.5:100`

Data Types: `single` | `double`

Output Arguments

`yworld` — *y*-world coordinates
N-length real-valued vector

y-world coordinates, returned as a *N*-length real-valued vector. The length and data type of `yWorld` are the same as for `xWorld`.

Data Types: `single` | `double`

Examples

Create Clothoid Lane Boundaries

Create clothoid curves to represent left and right lane boundaries. Then. plot the curves.

Create the left boundary.

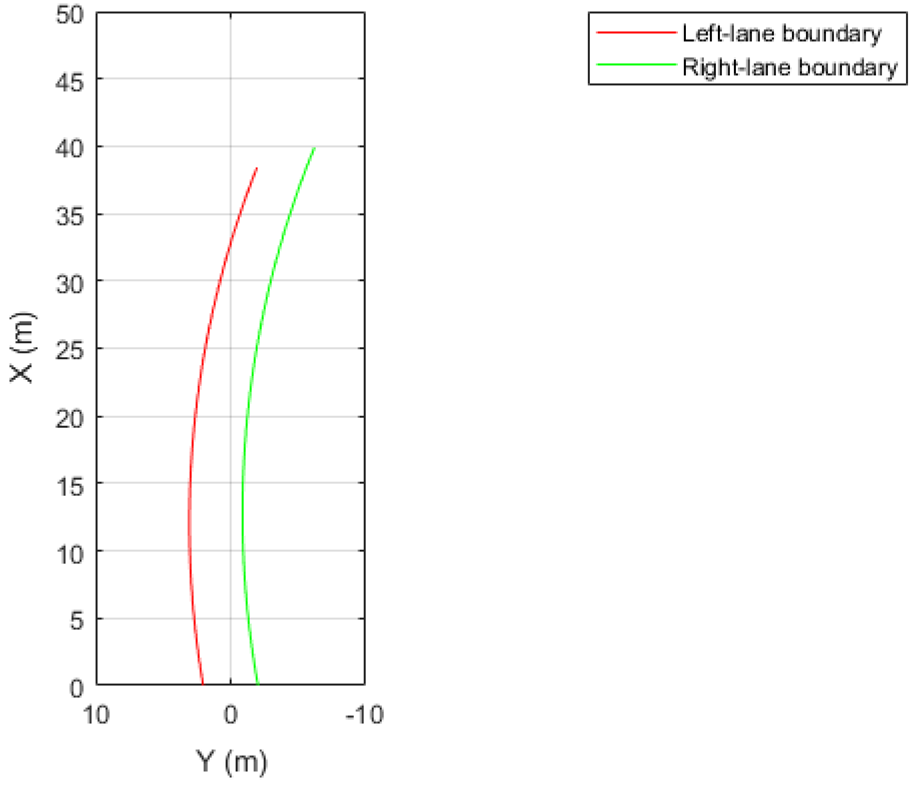
```
lb = clothoidLaneBoundary;
lb.BoundaryType = 'Solid';
lb.Strength = 1;
lb.Width = 0.2;
lb.CurveLength = 40;
lb.Curvature = -0.8;
lb.LateralOffset = 2;
lb.HeadingAngle = 10;
```

Create the right boundary with almost identical properties.

```
rb = lb;
rb.LateralOffset = -2;
```

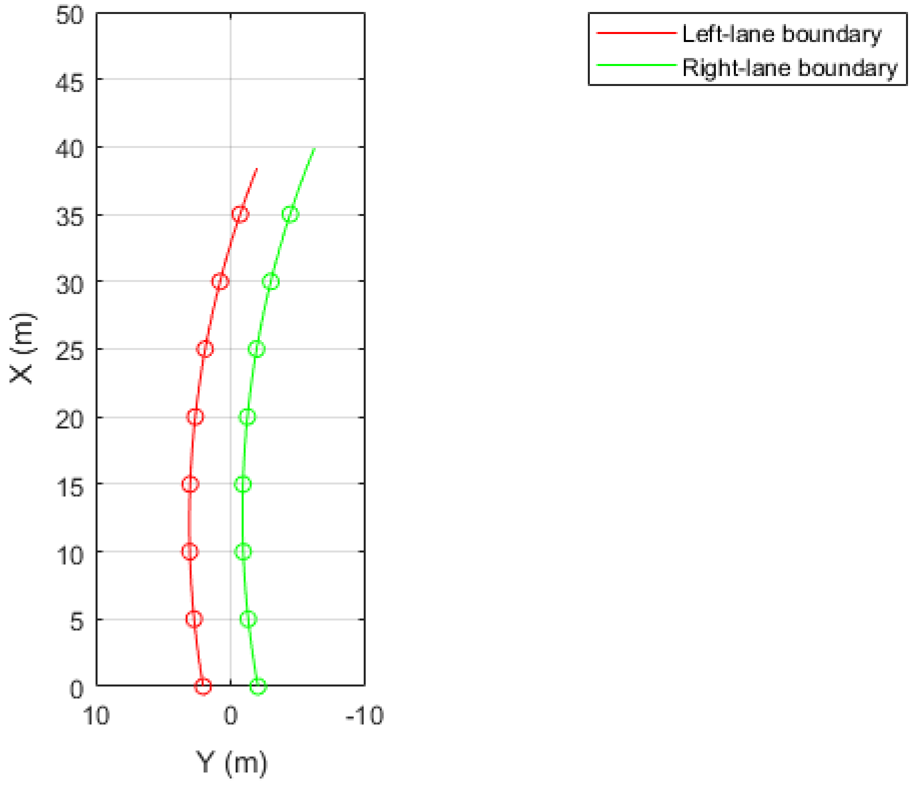
Create a bird's-eye plot. Then, create the lane boundary plotters and plot the boundaries.

```
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

`laneBoundaries`

Introduced in R2018a

currentLane

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

This example shows how to obtain the current lane of a car during a driving scenario simulation. The car is driving along a straight road at 20 m/s.

Create an empty driving scenario. Then, add a straight road with three lanes.

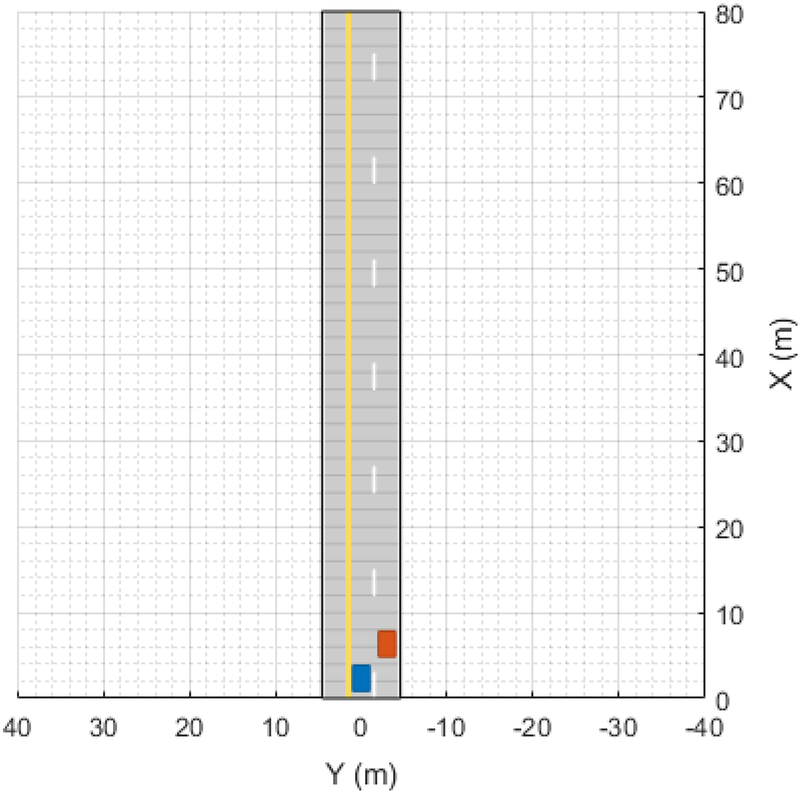
```
s = drivingScenario;
roadCenters = [0 0; 80 0];
road(s,roadCenters,'Lanes',lanespec([1 2],'Width',3));
```

Add an ego car moving at 20 m/s.

```
car1 = vehicle(s,'Position',[5 0 0],'Length',3,'Width',2,'Height',1.6);
trajectory(car1,[1 0 0; 20 0 0; 30 0 0;50 0 0],20);
car2 = vehicle(s,'Position',[5 0 0],'Length',3,'Width',2,'Height',1.6);
trajectory(car2,[5 -3 0; 20 -3 0; 30 -3 0;50 -3 0],10);
```

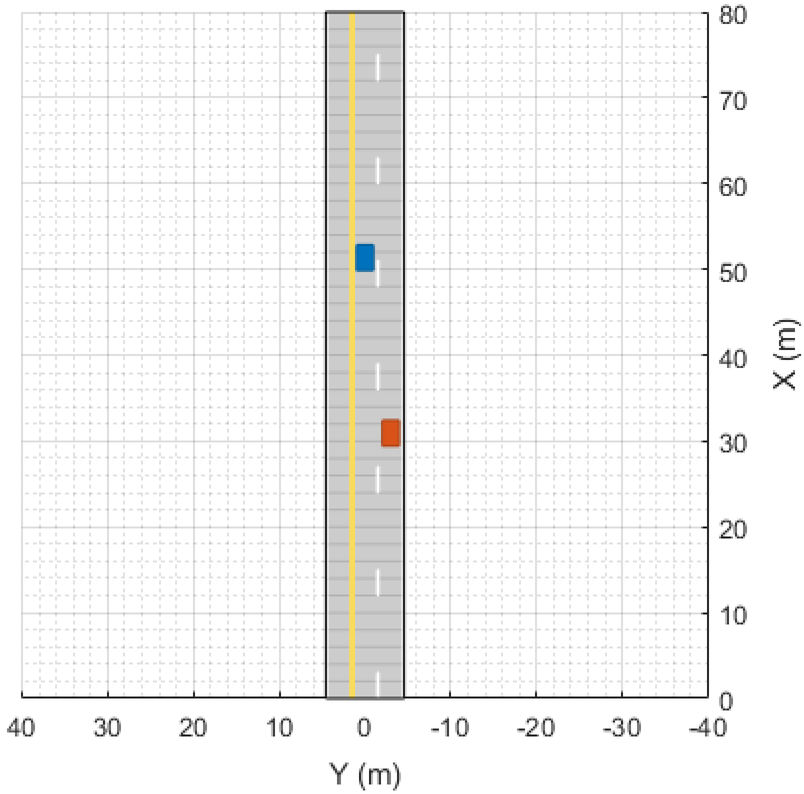
Plot the scenario.

```
plot(s)
```



Run the simulation loop.

```
while advance(s)
    [cl1,numlanes] = currentLane(car1);
    [cl2,numlanes] = currentLane(car2);
end
```



Display the current lane.

```
disp(cl1)
disp(cl2)
```

```
     2
```

```
    3
```

Input Arguments

ac — Scenario actor
`Actor` object | `Vehicle` object

Scenario actor, specified as an `Actor` or `Vehicle` object. To create actors, use the `actor` or `vehicle` method.

Output Arguments

`cl` — Road lane on which actor is traveling
positive integer | `[]`

Road lane on which actor is traveling, specified 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, empty values are returned.

Data Types: `double`

`numlanes` — Number of road lanes
positive integer | `[]`

Number of road lanes, specified as a positive integer. When the actor is not on a road or is on a road without any lanes specified, empty values are returned.

Data Types: `double`

See Also

`actor` | `laneBoundaries` | `vehicle`

Introduced in R2018a

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 4-656. 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 properties using one or more name-value pairs, in addition to the input arguments from preceding syntaxes. For example, `inflationCollisionChecker('InflationRadius',1.2,'CenterPlacements',[0.2 0.5 0.8])` sets specific values for the inflation radius and center placements. Enclose each property name in quotes.

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.

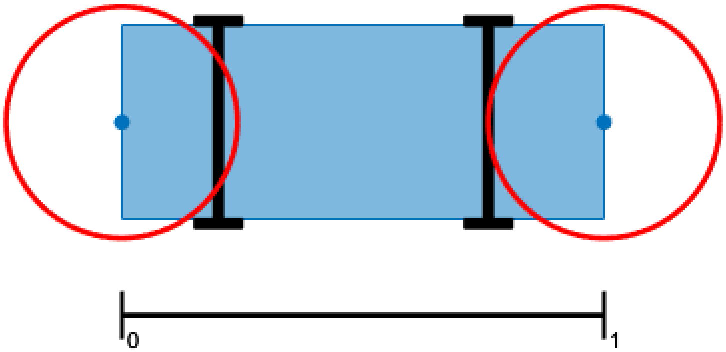
`CenterPlacements` — Normalized placement of circle centers

1-by-`NumCircles` numeric vector of values in the range [0, 1]

Normalized placement of circle centers along the longitudinal axis of the vehicle, specified as a 1-by-`NumCircles` numeric vector of 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. If you leave this property unspecified, 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 4-656.

Object Functions

plot Plot collision configuration

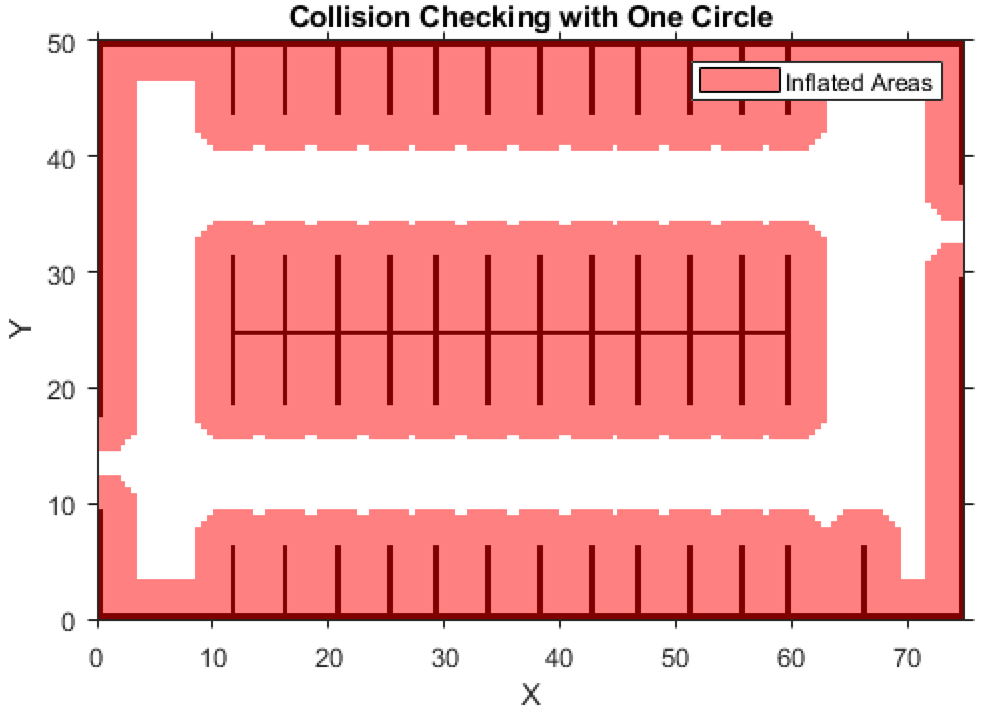
Examples

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.

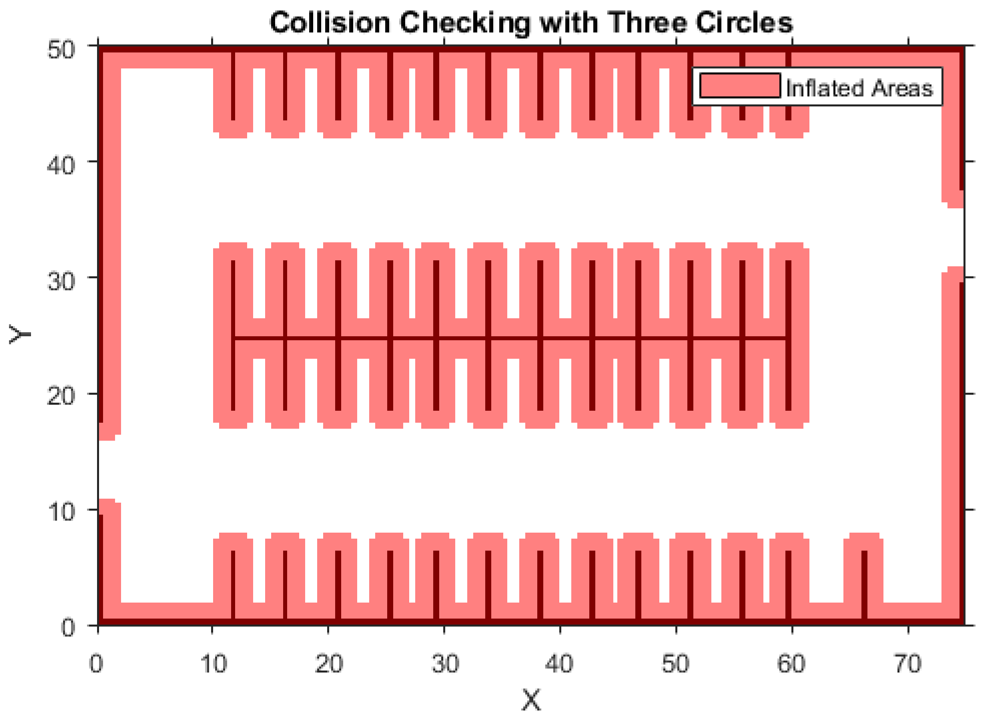
- To decrease inflation of the obstacles, increase the number of circles enclosing the vehicle.
- To specify the dimensions of the vehicle, use a `vehicleDimensions` object.

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')
```

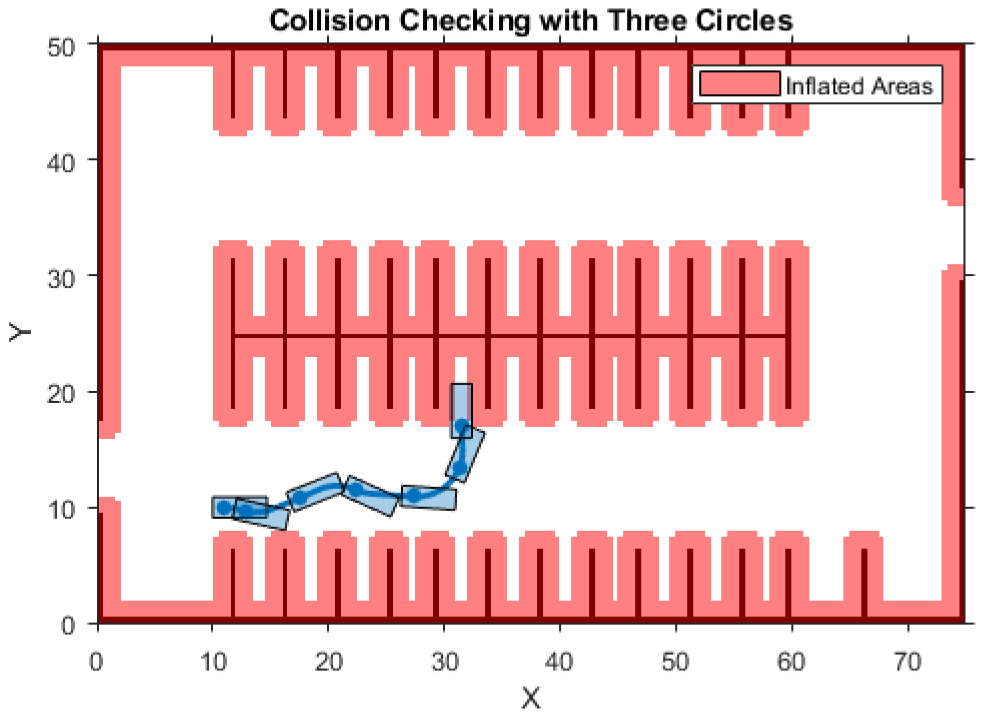


Define a planning problem: a vehicle starts near the left entrance of the parking lot and ends in a parking spot.

```
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.

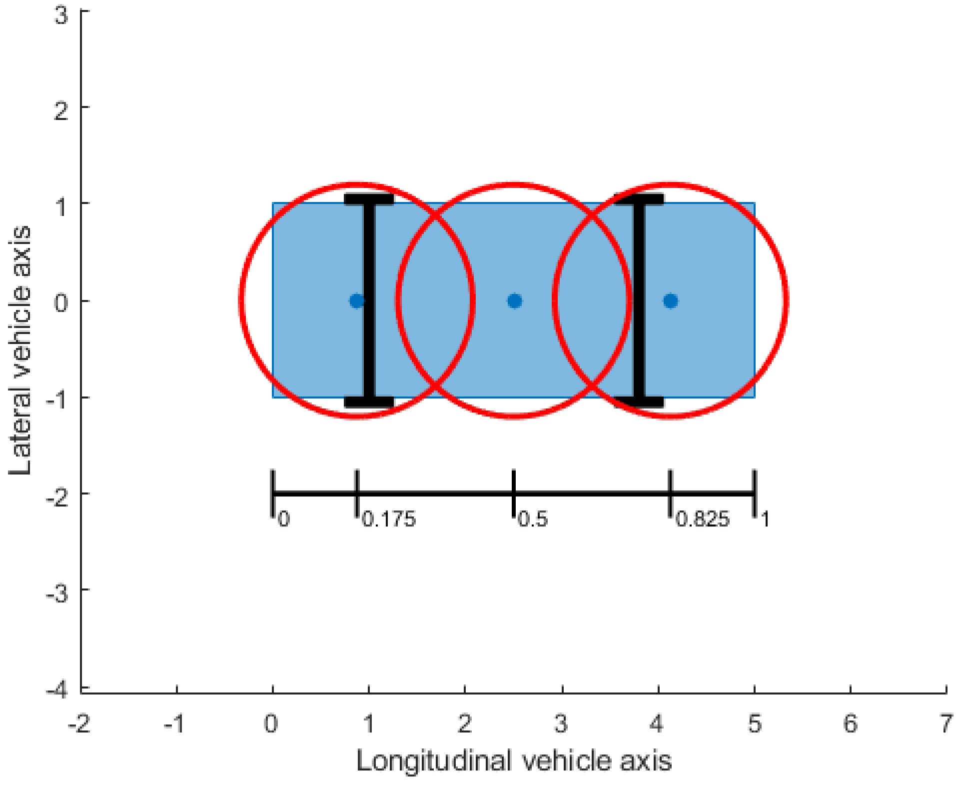
```
length = 5; % meters
width = 2; % meters
vehicleDims = vehicleDimensions(length,width);
```

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;
```

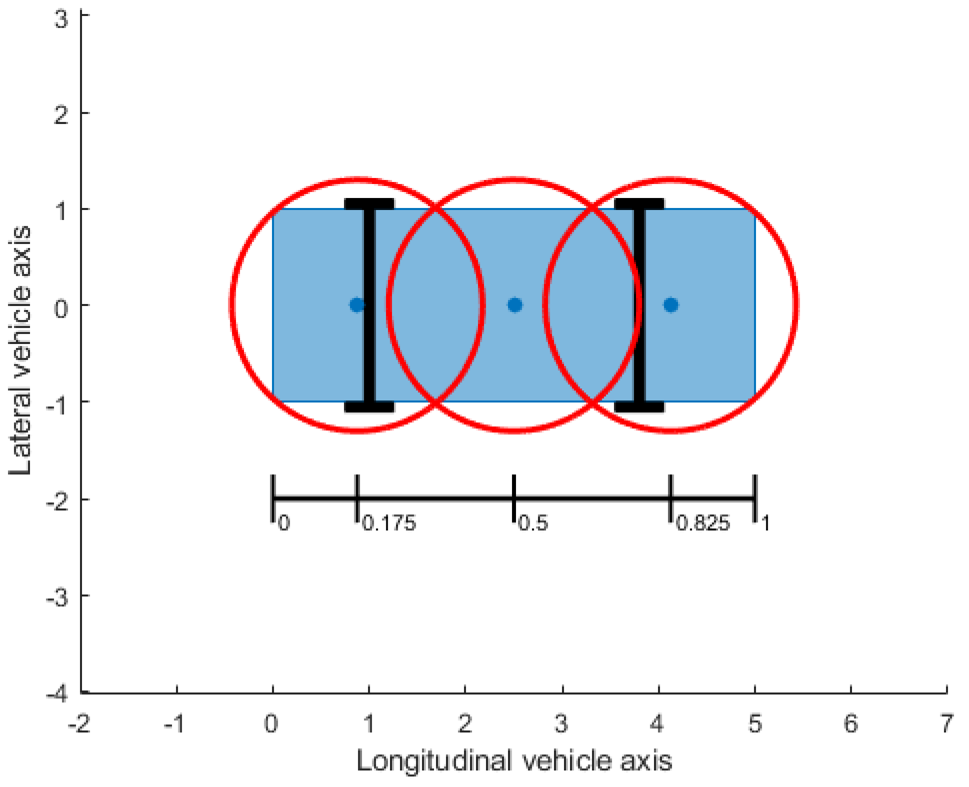
Create and display the collision-checking configuration.

```
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);
```

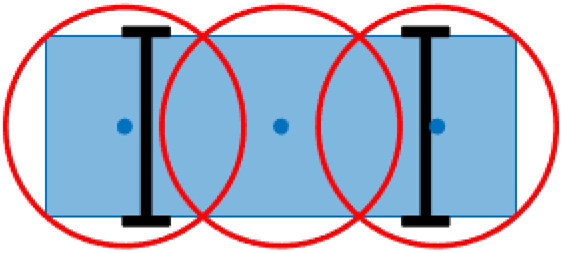
Tips

- 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 4-505 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.

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.

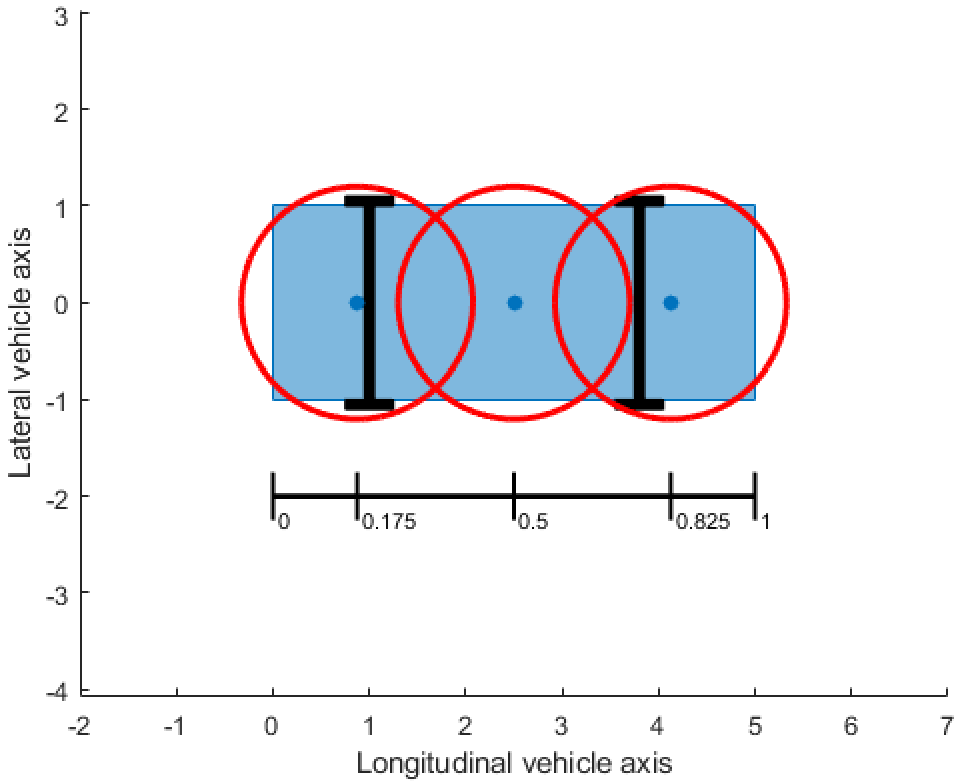
```
length = 5; % meters
width = 2; % meters
vehicleDims = vehicleDimensions(length,width);
```

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;
```

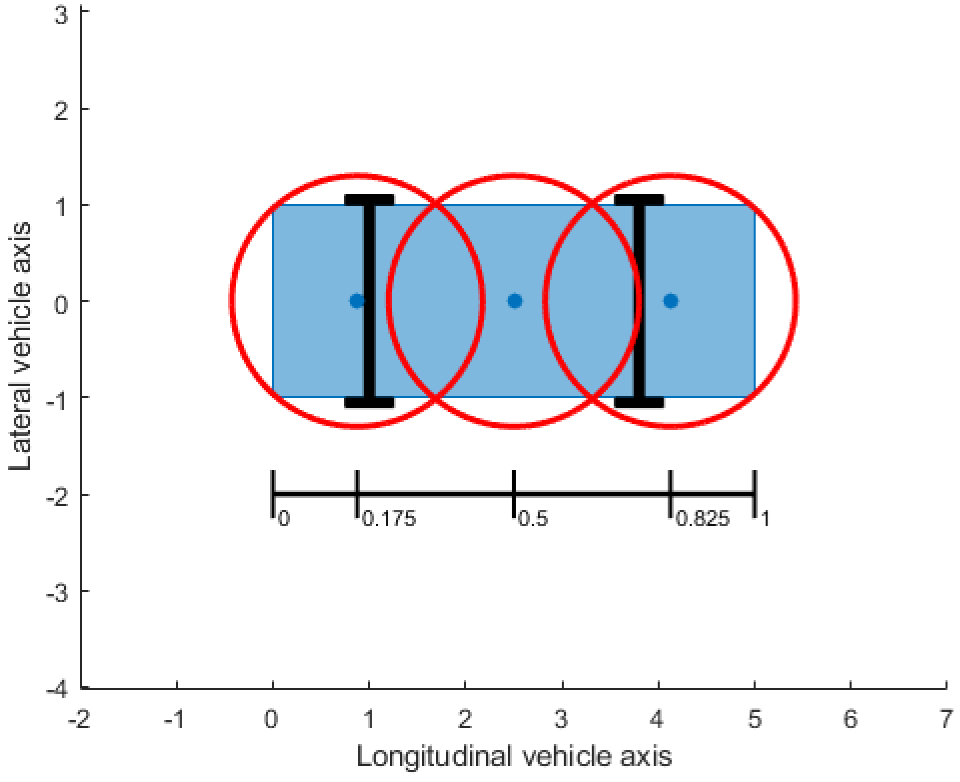
Create and display the collision-checking configuration.

```
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