Lidar Toolbox™

Reference

MATLAB®

R2020**b**

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

Lidar Toolbox™ Reference

Revision History

September 2020	Online only	New for Version 1.0 (R2020b)

Contents

1

Apps

Lidar Labeler

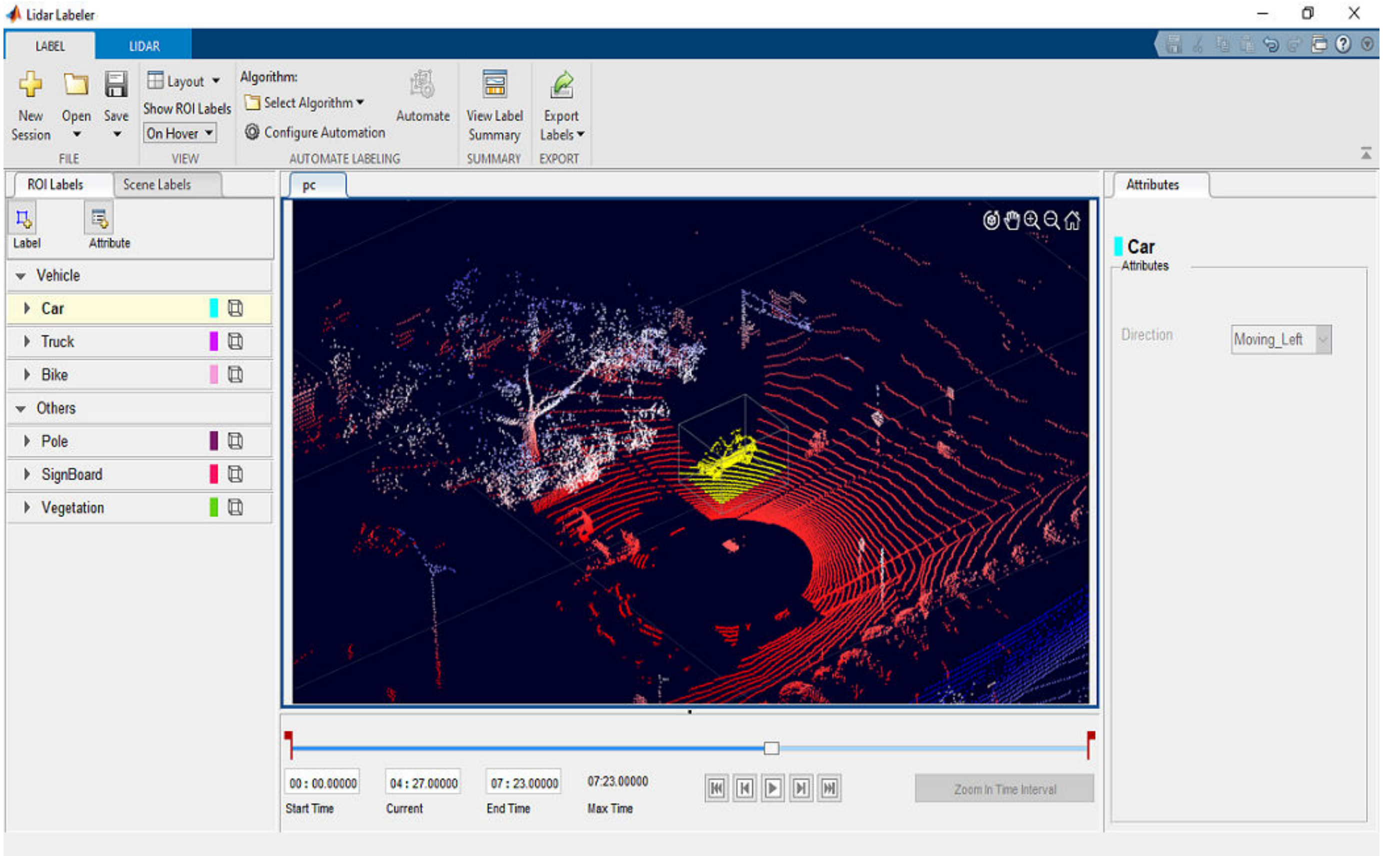
Label ground truth data in lidar point clouds

Description

The **Lidar Labeler** app enables you to label objects in a point cloud or a point cloud sequence. The app reads point cloud data from PLY, PCAP, LAS, LAZ, ROS and PCD files. Using the app, you can:

- Define cuboid region of interest (ROI) labels and scene labels. Use them to interactively label your ground truth data.
- Define attributes for the labels and use them to provide further detail about the labels.
- Use built-in algorithms for clustering, ground plane segmentation, automated labeling, and tracking.
- Save label definitions, point cloud data, and ground truth data to a session file for future use.
- Use the **Projected View** option to view the labels in top, front and side views simultaneously.
- Use the **Camera View** option to create and reuse custom views of the point cloud data.
- Use the **Auto Align** option to rotate and best fit the cuboid to the cluster.
- Use the `lidar.syncImageViewer.SyncImageViewer` class to sync the app to an external visualization or analysis tool.
- Write, import, and use a custom automation algorithm for automated labeling.
- Evaluate the performance of your label automation algorithms with a visual summary.
- Export the labeled ground truth as a `groundTruthLidar` object. This object can be used for system verification and training an object detector.

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



Open the Lidar Labeler App

- MATLAB® Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the app icon.
- MATLAB command prompt: Enter `lidarLabeler`.

Examples

- “Get Started with the Lidar Labeler”
- “Choose an App to Label Ground Truth Data”
- “Keyboard Shortcuts and Mouse Actions for Lidar Labeler”

Programmatic Use

`lidarLabeler` opens a new session of the app, enabling you to label ground truth data in point clouds.

`lidarLabeler(velodyneLidarFileName,deviceModel,calibrationFile)` opens the app and loads the `velodyneLidarFileName`.

lidarLabeler(ptCloudSeqFolder) opens the app and loads the point cloud sequence from the folder ptCloudSeqFolder, where ptCloudSeqFolder is a string scalar or character vector specifying a folder that contains point cloud files. The point cloud files must have extensions supported by pcformats, and are loaded in the order returned by the dir function.

lidarLabeler(lasSeqFolder) opens the app and loads the LAS sequence from the folder lasSeqFolder, where lasSeqFolder is a string scalar or character vector specifying a folder contains LAS files. LAS files must have extensions supported by lasformats, and are loaded in the order returned by the dir function.

lidarLabeler(___ ,'SyncImageViewerTargetHandle',syncImageViewer) opens the app and loads both of these components:

- A point cloud signal, specified using any of the input argument combinations from previous syntaxes.
- An external video or image sequence display tool that is time-synchronized with the specified point cloud signal.

The syncImageViewer input is a handle to a lidar.syncImageViewer.SyncImageViewer class that implements the external tool.

For example, this code opens the app with a point cloud signal and synchronized video visualization tool.

```
sourceName = fullfile(toolboxdir('lidar'),'lidardata','lcc', ...
                      'HDL64','pointCloud');
lidarLabeler(sourceName,'SyncImageViewerTargetHandle',@SyncImageDisplay)
```

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

Limitations

- The labels do not support sublabels.
- The Label Summary window does not support sublabels.

More About

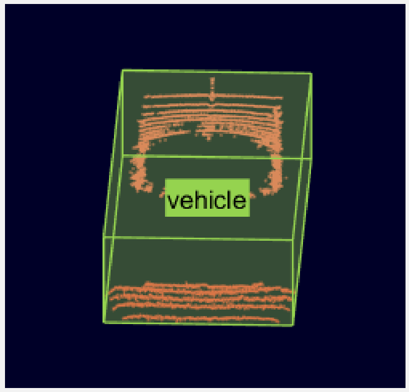
ROI Labels and Attributes

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

The app supports the definition of ROI labels and attributes.

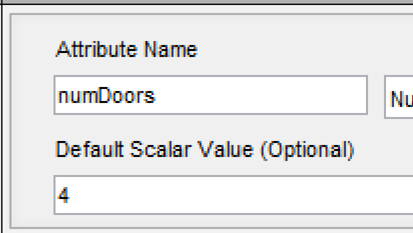
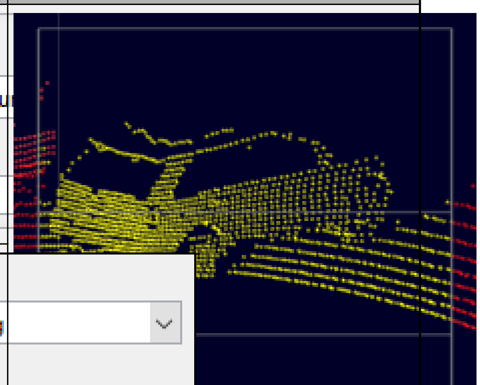
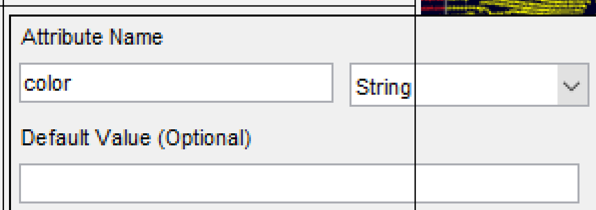
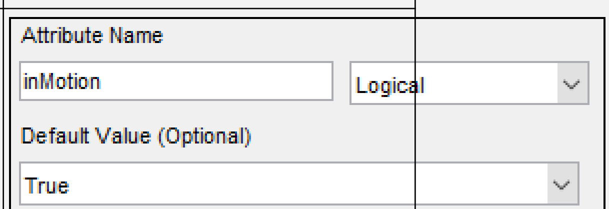
ROI Labels

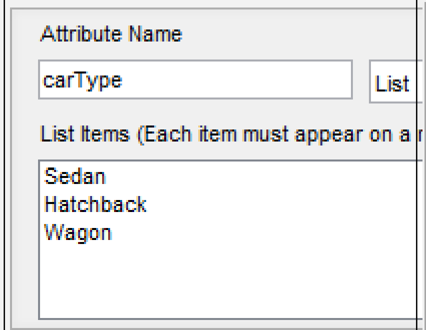
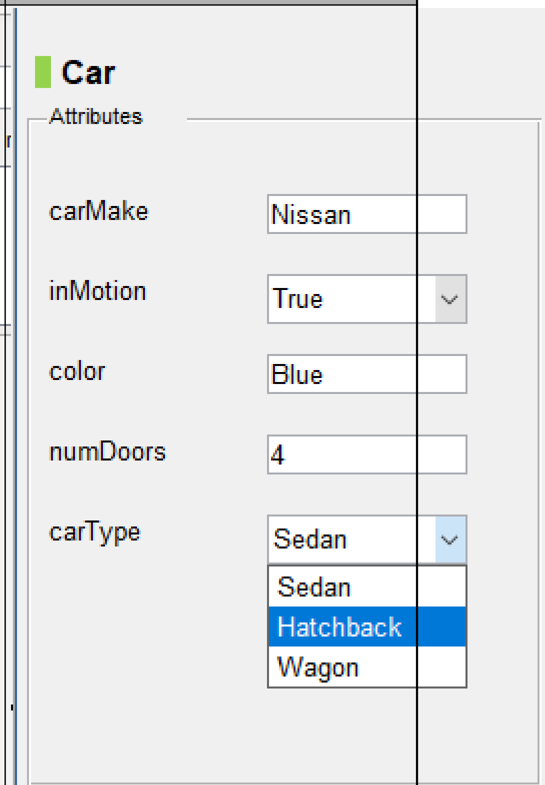
An ROI label is a label that corresponds to an ROI in a signal frame. This table describes the supported label type.

ROI Label	Description	Example
`Cuboid`	Draw cuboidal ROI labels around objects.	

ROI Attributes

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

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

Attribute Type	Sample Attribute Definition	Sample Default Values
`List`		

Tips

- Use the `lidar.syncImageViewer.SyncImageViewer` class to create a tool for viewing the image corresponding to the point cloud data.
- Remove the ground plane to clearly view the created object labels.
- Use the rotate, translate, expand, and shrink options to edit the cuboids after drawing them.
- Use the **Camera View** option to save the a view of the data from the current angle and direction.
- To avoid having to relabel ground truth with new labels, organize the labeling scheme you want to use before you begin marking your ground truth.
- You can copy and paste the labels between signals that are of the same type.

Algorithms

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

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

Lidar Object Tracker

Track an object through the point cloud frame. To use this algorithm, you must draw a cuboid ROI on an object you wish to track. You can also draw multiple cuboid ROIs to track more than one label. Running the algorithm provides tracking data of the labels that you can accept or reject. You can also undo the run and perform it again.

The step by step procedure is displayed on app when you select the **Lidar Object Tracker** algorithm.

Point Cloud Temporal Interpolator

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

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

See Also

Apps
Image Labeler | **Video Labeler**

Objects
`groundTruthLidar` | `labelDefinitionCreatorLidar`

Classes
`lidar.syncImageViewer.SyncImageViewer`

Topics
"Get Started with the Lidar Labeler"
"Choose an App to Label Ground Truth Data"
"Keyboard Shortcuts and Mouse Actions for Lidar Labeler"

Introduced in R2020b

Objects

cuboidModel

Parametric cuboid model

Description

The `cuboidModel` object stores the parameters of a parametric cuboid model. After you create a `cuboidModel` object, you can extract cuboid corner points, and points within the cuboid using the object functions. Cuboid models are used to store the output of `pcfitcuboid` function. It is a shape fitting function which fits a cuboid over a point cloud.

Creation

Syntax

```
model = cuboidModel(params)
model = pcfitcuboid(ptCloudIn)
model = pcfitcuboid(ptCloudIn,indices)
```

Description

`model = cuboidModel(params)` constructs a parametric cuboid model from the 1-by-9 input vector, `params`.

`model = pcfitcuboid(ptCloudIn)` fits a cuboid over the input point cloud data. The `pcfitcuboid` function stores the properties of the cuboid in a parametric cuboid model object, `model`.

`model = pcfitcuboid(ptCloudIn,indices)` fits a cuboid over a selected set of points, `indices`, in the input point cloud.

For more information on how to use this function, visit `pcfitcuboid` function reference page.

Properties

`Parameters` — Cuboid model parameters
nine-element row vector

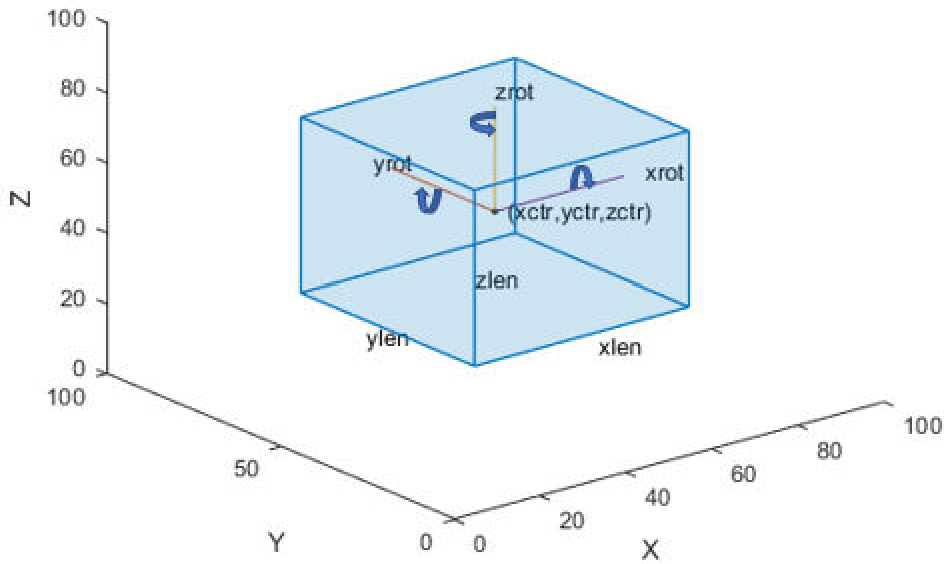
This property is read-only.

Cuboid model parameters, stored as a nine-element row vector of the form [x_{ctr} y_{ctr} z_{ctr} x_{len} y_{len} z_{len} x_{rot} y_{rot} z_{rot}].

- x_{ctr}, y_{ctr}, and z_{ctr} specify the center of the cuboid.
- x_{len}, y_{len}, and z_{len} specify the length of the cuboid along the *x*-, *y*-, and *z*-axis, respectively, before rotation has been applied.

- x_{rot}, y_{rot}, and z_{rot} specify the rotation angles for the cuboid along the *x*-, *y*-, and *z*-axis, respectively. These angles are clockwise-positive when looking in the forward direction of their corresponding axes.

The figure shows how these values determine the position of a cuboid.



These parameters are specified by the `params` input argument.

Data Types: `single` | `double`

`Center` — Center of cuboid
three-element row vector

This property is read-only.

Center of the cuboid, stored as a three-element row vector of the form [x_{ctr} y_{ctr} z_{ctr}]. The vector contains the 3-D coordinates of the cuboid center in the *x*-, *y*-, and *z*-axis, respectively.

This property is derived from the `Parameters` property.

Data Types: `single` | `double`

`Dimensions` — Dimensions of cuboid
three-element row vector

This property is read-only.

Dimensions of the cuboid, stored as a three-element row vector of the form [x_{len} y_{len} z_{len}]. The vector contains the length of the cuboid along the *x*-, *y*-, and *z*-axis, respectively.

This property is derived from the `Paramareters` property.

Data Types: `single` | `double`

`Orientation` — Orientation of cuboid
three-element row vector

This property is read-only.

Orientation of the cuboid, stored as a three-element row vector of the form, [x_{rot} y_{rot} z_{rot}], in degrees. The vector contains the rotation of the cuboid along the *x*-, *y*-, and *z*-axis, respectively.

This property is derived from the `Paramareters` property.

Data Types: `single` | `double`

Object Functions

getCornerPoints	Get corner points of cuboid model
findPointsInsideCuboid	Find points enclosed by cuboid model
plot	Plot cuboid model

Examples

Detect Cuboid in Point Cloud

Detect cuboid in a point cloud using `pcfitcuboid` function. The function stores the cuboid parameters as a `cuboidModel` object.

Read point cloud data into the workspace.

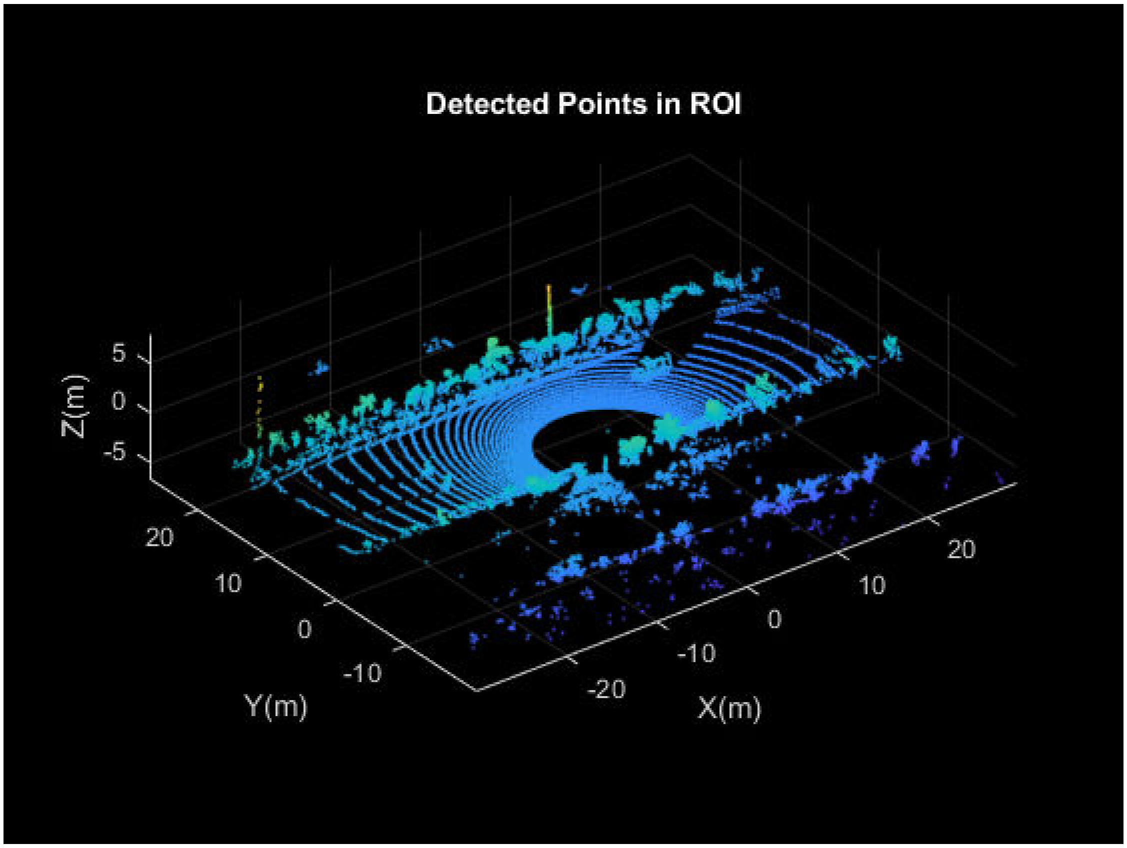
```
ptCloud = pcread('highwayScene.pcd');
```

Search the point cloud within a specified region of interest (ROI). Create a point cloud of only the detected points.

```
roi = [-30 30 -20 30 -8 13];
in = findPointsInROI(ptCloud,roi);
ptCloudIn = select(ptCloud,in);
```

Plot the point cloud of detected points.

```
figure
pcshow(ptCloudIn.Location)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
title('Detected Points in ROI')
```



Find the indices of the points in a specified ROI within the point cloud.

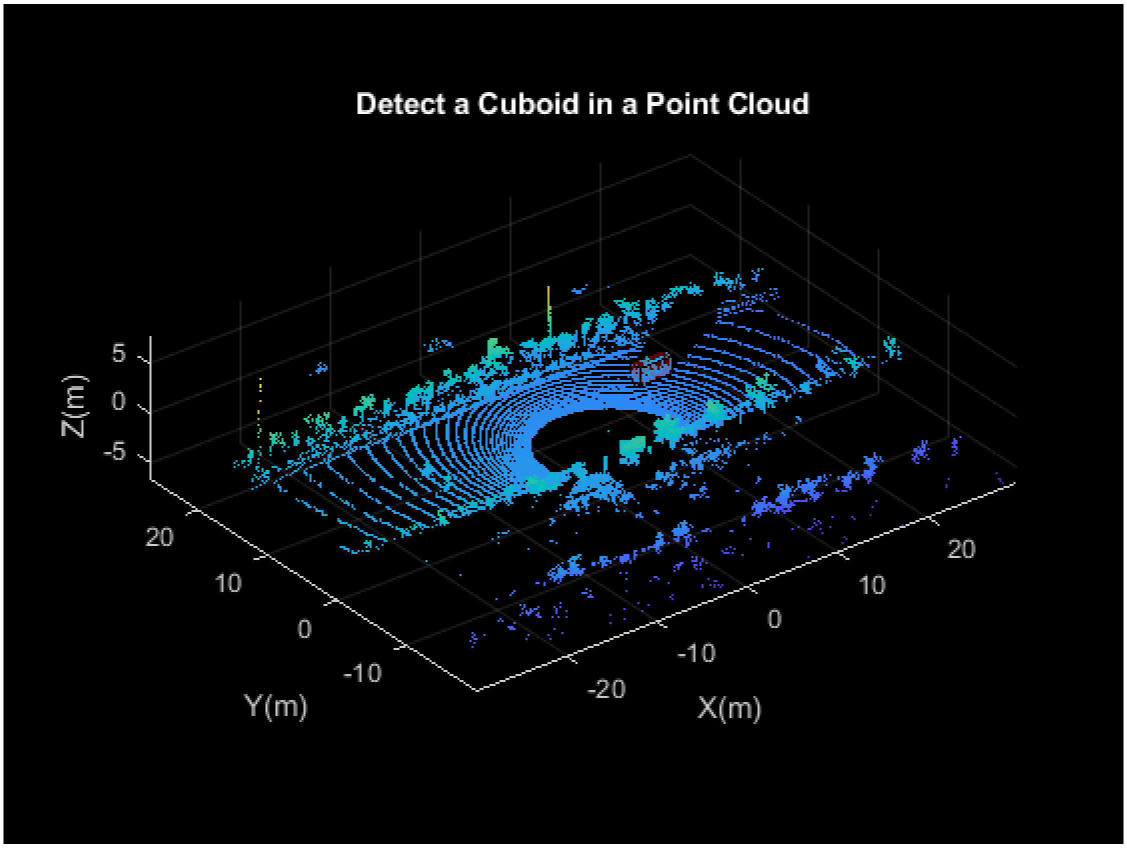
```
roi = [9.6 13.8 7.9 9.3 -2.5 3];
sampleIndices = findPointsInROI(ptCloudIn,roi);
```

Fit a cuboid to the selected set of points in the point cloud.

```
 model = pcfitcuboid(ptCloudIn,sampleIndices);
  figure
  pcshow(ptCloudIn.Location)
  xlabel('X(m)')
  ylabel('Y(m)')
  zlabel('Z(m)')
  title('Detect a Cuboid in a Point Cloud')
```

Plot the cuboid box in the point cloud.

```
hold on
plot(model)
```



Display internal properties of `cuboidModel` object.

```
model
```

```
model = 
  cuboidModel with properties:

     Parameters: [11.4873 8.5997 -1.6138 3.6713 1.3220 1.7576 0 0 0.9999]
         Center: [11.4873 8.5997 -1.6138]
     Dimensions: [3.6713 1.3220 1.7576]
    Orientation: [0 0 0.9999]
```

Fit Cuboid Over Point Cloud Data

Fit bounding boxes around clusters in a point cloud.

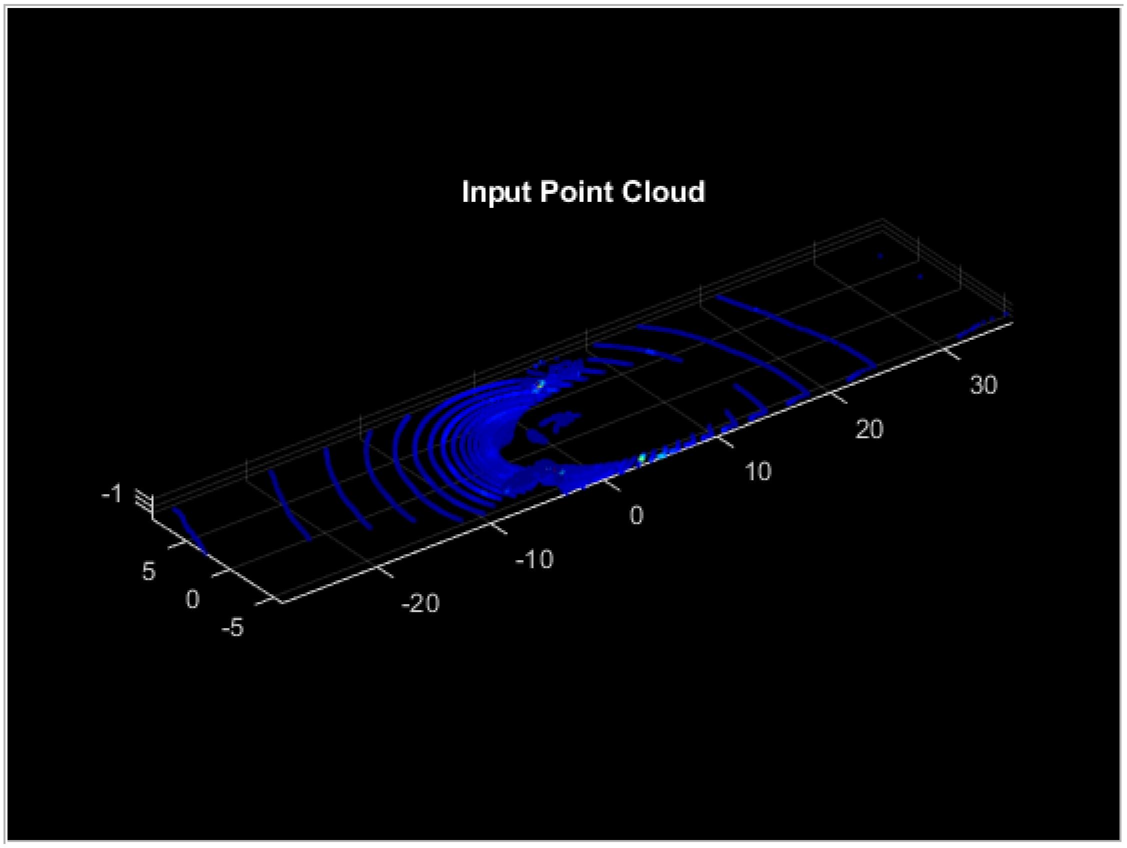
Load the point cloud data into the workspace.

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

Define and crop a region of interest (ROI) from the point cloud. Visualize the selected ROI of the point cloud.

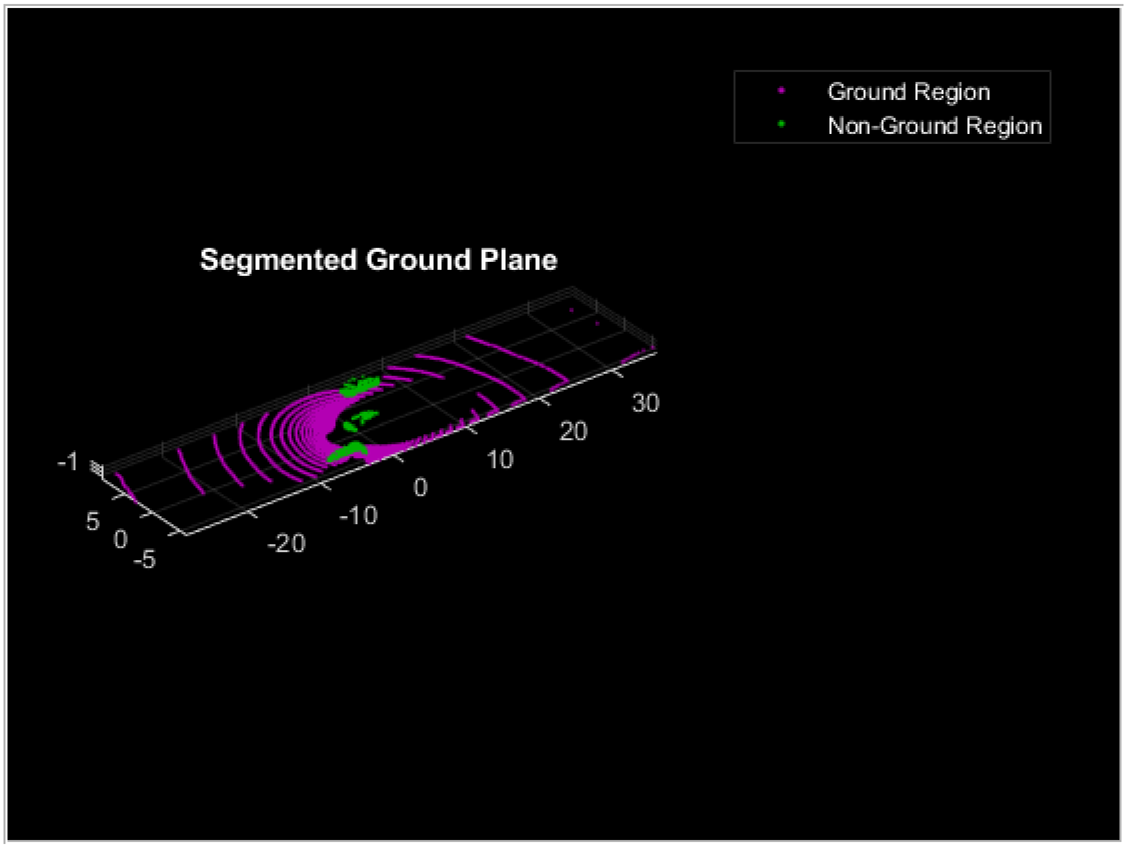
```
roi = [-40 40 -6 9 -2 1];
in = findPointsInROI(data.ptCloud,roi);
```

```
ptCloudIn = select(data.ptCloud,in);
hcluster = figure;
panel = uipanel('Parent',hcluster,'BackgroundColor',[0 0 0]);
ax = axes('Parent',panel,'Color',[0 0 0]);
pcshow(ptCloudIn,'MarkerSize',30,'Parent',ax)
title('Input Point Cloud')
```



Segment the ground plane. Visualize the segmented ground plane.

```
maxDistance = 0.3;
referenceVector = [0 0 1];
[~,inliers,outliers] = pcfitplane(ptCloudIn,maxDistance,referenceVector);
ptCloudWithoutGround = select(ptCloudIn,outliers,'OutputSize','full');
hSegment = figure;
panel = uipanel('Parent',hSegment,'BackgroundColor',[0 0 0]);
ax = axes('Parent',panel,'Color',[0 0 0]);
pcshowpair(ptCloudIn,ptCloudWithoutGround,'Parent',ax)
legend('Ground Region','Non-Ground Region','TextColor', [1 1 1])
title('Segmented Ground Plane')
```

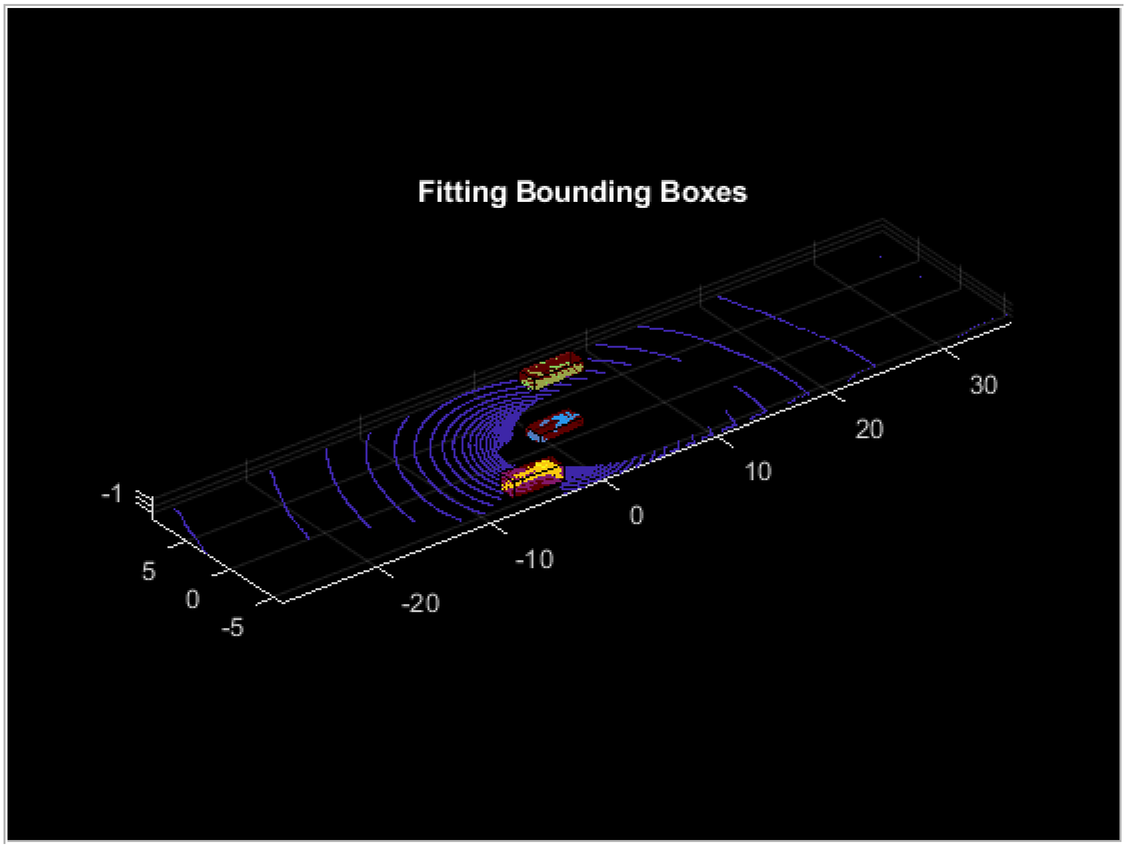


Segment the non-ground region of the point cloud into clusters. Visualize the segmented point cloud.

```
distThreshold = 1;
[labels,numClusters] = pcsegdist(ptCloudWithoutGround,distThreshold);
labelColorIndex = labels;
hCuboid = figure;
panel = uipanel('Parent',hCuboid,'BackgroundColor',[0 0 0]);
ax = axes('Parent',panel,'Color',[0 0 0]);
pcshow(ptCloudIn.Location,labelColorIndex,'Parent',ax)
title('Fitting Bounding Boxes')
hold on
```

Fit bounding box on each cluster, visualized as orange highlights.

```
for i = 1:numClusters
    idx = find(labels == i);
    model = pcfitcuboid(ptCloudWithoutGround,idx);
    plot(model)
end
```



Extended Capabilities

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

See Also

Functions
`findPointsInsideCuboid` | `getCornerPoints` | `pcfitcuboid` | `plot`

Objects
`cylinderModel` | `planeModel` | `pointCloud` | `sphereModel`

Introduced in R2020b

findPointsInsideCuboid

Find points enclosed by cuboid model

Syntax

```
Indices = findPointsInsideCuboid(model,ptCloudIn)
```

Description

`Indices = findPointsInsideCuboid(model,ptCloudIn)` returns the linear indices of to the points enclosed by a cuboid model, `model`, in an input point cloud, `ptCloudIn`.

Examples

Extract Points Inside Cuboid Model

Extract points enclosed by a cuboid model in a point cloud. Create the cuboid model as a `cuboidModel` object.

Read point cloud data into the workspace.

```
ptCloudIn = pcread('highwayScene.pcd');
```

Define a cuboid model as a `cuboidModel` object.

```
params = [11.4873085 8.59969 -1.613766 3.6712 1.3220...
    1.75755, 0, 0, 0.017451];
model = cuboidModel(params);
```

Find the points inside the cuboid.

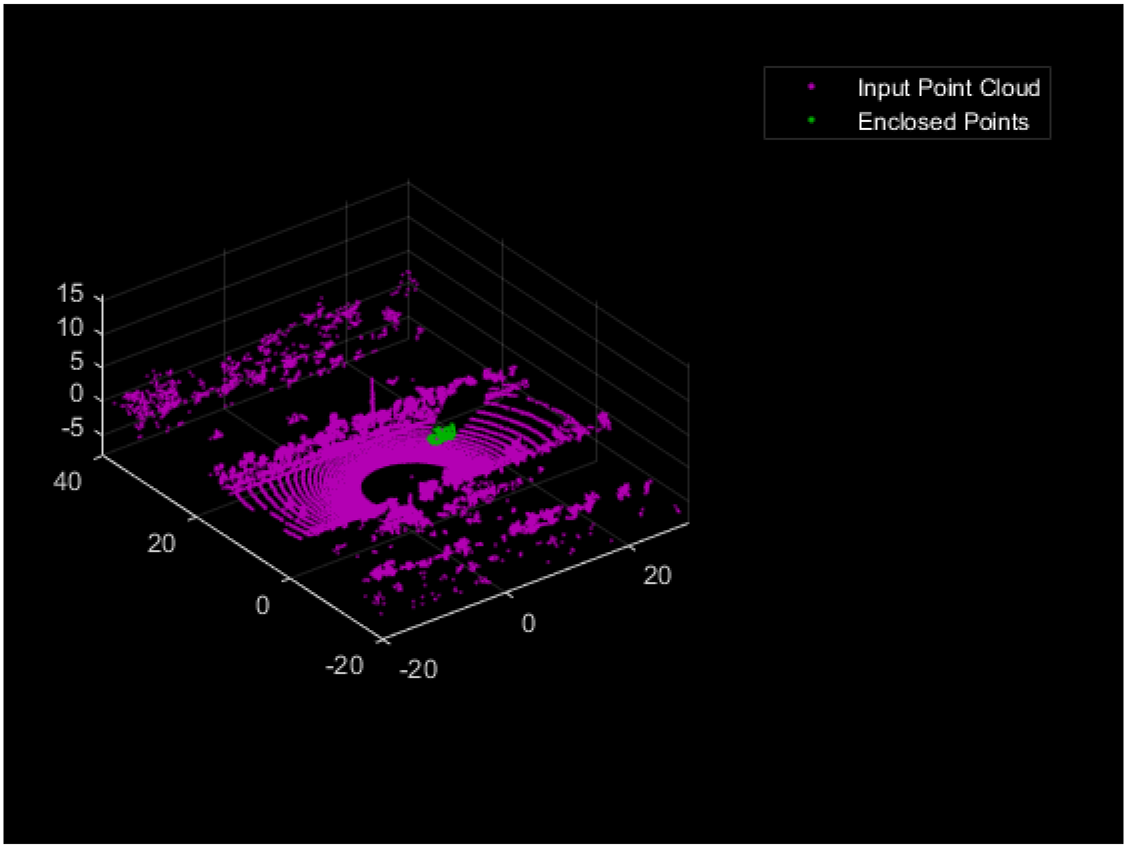
```
indices = findPointsInsideCuboid(model,ptCloudIn);
```

Select the corresponding points in the input point cloud.

```
cubPtCloud = select(ptCloudIn,indices);
```

Plot the point cloud and the points enclosed by the cuboid.

```
pcshowpair(ptCloudIn,cubPtCloud)
xlim([-20 30])
ylim([-20 40])
legend("Input Point Cloud","Enclosed Points",'TextColor',[1 1 1])
```



Input Arguments

model — Cuboid model
cuboidModel object

Cuboid model, specified as a `cuboidModel` object.

ptCloudIn — Point cloud
pointCloud object

Point cloud, specified as a `pointCloud` object.

Output Arguments

Indices — Indices of enclosed points
N-element column vector

Indices of enclosed points, returned as *N*-element column vector. *N* is the number of enclosed points. Use the `select` function to select the corresponding points in the input point cloud `ptCloudIn`.

Extended Capabilities

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

See Also

Functions
getCornerPoints | pcfitcuboid | plot

Objects
cuboidModel

Introduced in R2020b

getCornerPoints

Get corner points of cuboid model

Syntax

```
points = getCornerPoints(model)
```

Description

`points = getCornerPoints(model)` returns the corner points of a cuboid model as 3-D coordinates.

Examples

Get Corner Points of Cuboid Model

Create a cuboid model object using the `cuboidModel` creation function, and get the corner points of the cuboid model as 3-D coordinates.

Read point cloud data into the workspace.

```
ptCloudIn = pcread('highwayScene.pcd');
```

Define a cuboid model as a `cuboidModel` object.

```
params = [11.4873085 8.59969 -1.613766 3.6712 1.3220,...
    1.75755 0 0 0.017451];
model = cuboidModel(params);
```

Get the corner points of the cuboid model.

```
points = getCornerPoints(model)
```

```
points = 8×3

   13.3227    9.2612   -0.7350
    9.6515    9.2601   -0.7350
    9.6519    7.9381   -0.7350
   13.3231    7.9392   -0.7350
   13.3227    9.2612   -2.4925
    9.6515    9.2601   -2.4925
    9.6519    7.9381   -2.4925
   13.3231    7.9392   -2.4925
```

The columns represent the *x*, *y*, and *z* coordinates, respectively, of the eight corners of the cuboid model. Each row represents a corner point.

Input Arguments

`model` — Cuboid model
`cuboidModel` object

Cuboid model, specified as a `cuboidModel` object.

Output Arguments

`points` — 3-D coordinates of corner points
8-by-3 matrix of real values

3-D coordinates of the corner points, returned as an 8-by-3 matrix of real values.

Extended Capabilities

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

See Also

Functions
`findPointsInsideCuboid` | `pcfitcuboid` | `plot`

Objects
`cuboidModel`

Introduced in R2020b

plot

Plot cuboid model

Syntax

```
H = plot(model)
H = plot(model,'Parent',ax)
```

Description

`H = plot(model)` plots a cuboid model within the axes limits of the current figure.

`H = plot(model,'Parent',ax)` plots a cuboid model on a specified output axes.

Examples

Detect Cuboid in Point Cloud

Detect cuboid in a point cloud using `pcfitcuboid` function. The function stores the cuboid parameters as a `cuboidModel` object.

Read point cloud data into the workspace.

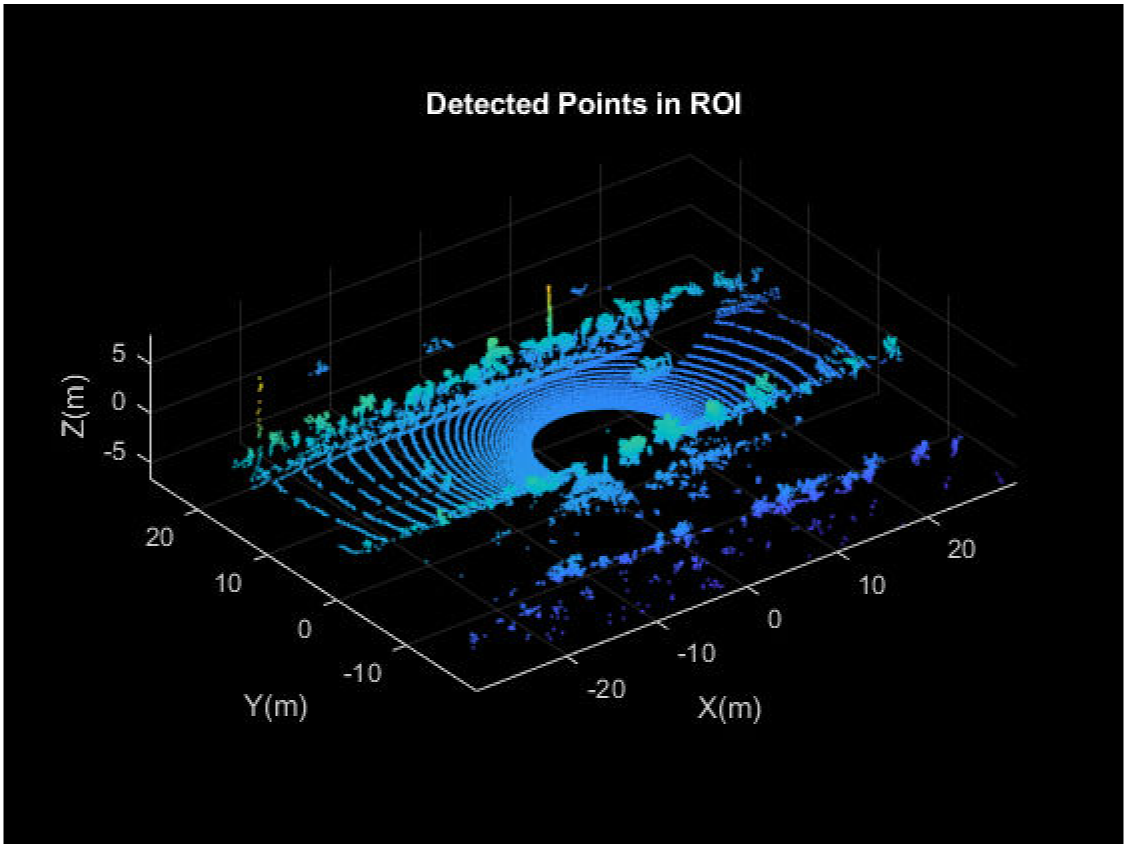
```
ptCloud = pcread('highwayScene.pcd');
```

Search the point cloud within a specified region of interest (ROI). Create a point cloud of only the detected points.

```
roi = [-30 30 -20 30 -8 13];
in = findPointsInROI(ptCloud,roi);
ptCloudIn = select(ptCloud,in);
```

Plot the point cloud of detected points.

```
figure
pcshow(ptCloudIn.Location)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
title('Detected Points in ROI')
```



Find the indices of the points in a specified ROI within the point cloud.

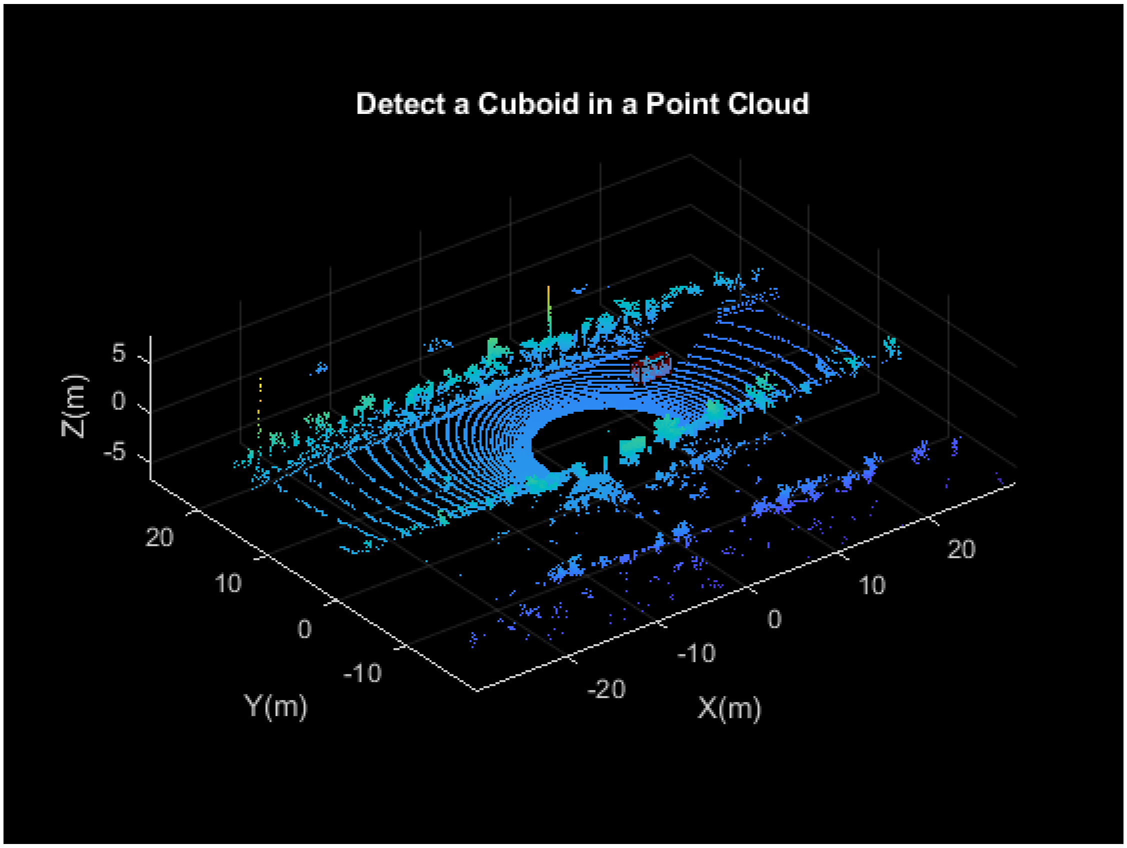
```
roi = [9.6 13.8 7.9 9.3 -2.5 3];
sampleIndices = findPointsInROI(ptCloudIn,roi);
```

Fit a cuboid to the selected set of points in the point cloud.

```
 model = pcfitcuboid(ptCloudIn,sampleIndices);
  figure
  pcshow(ptCloudIn.Location)
  xlabel('X(m)')
  ylabel('Y(m)')
  zlabel('Z(m)')
  title('Detect a Cuboid in a Point Cloud')
```

Plot the cuboid box in the point cloud.

```
hold on
plot(model)
```



Display internal properties of `cuboidModel` object.

```
model
```

```
model = 
  cuboidModel with properties:

     Parameters: [11.4873 8.5997 -1.6138 3.6713 1.3220 1.7576 0 0 0.9999]
         Center: [11.4873 8.5997 -1.6138]
     Dimensions: [3.6713 1.3220 1.7576]
    Orientation: [0 0 0.9999]
```

Input Arguments

`model` — Cuboid model
`cuboidModel` object

Cuboid model, specified as a `cuboidModel` object.

`ax` — Output axes
`gca` (default) | axes

Output axes, specified as the axes on which to display the cuboid model.

Output Arguments

`H` — Patch object
`patch` object

Patch object, returned as a `patch` object.

See Also

Functions
`findPointsInsideCuboid` | `getCornerPoints` | `pcfitcuboid`

Objects
`cuboidModel`

Introduced in R2020b

groundTruthLidar

Lidar ground truth label data

Description

The `groundTruthLidar` object contains information about lidar ground truth labels. The data source used to create the object is a collection of lidar point cloud data. You can create, export, or import a `groundTruthLidar` object from the **Lidar Labeler** app.

Creation

To export a `groundTruthLidar` object from the **Lidar Labeler** app, on the app toolstrip, select **Export Labels** > **To Workspace**. The app exports the object to the MATLAB workspace. To create a `groundTruthLidar` object programmatically, use the `groundTruthLidar` function (described here).

Syntax

```
gTruth = groundTruthLidar(dataSource,labelDefs,labelData)
```

Description

`gTruth = groundTruthLidar(dataSource,labelDefs,labelData)` returns an object containing lidar ground truth labels that can be imported into the **Lidar Labeler** app.

- `dataSource` specifies the source of the lidar point cloud data and sets the `DataSource` property.
- `labelDefs` specifies the definitions of ROI and scene labels containing information such as `Name`, `Type`, and `Group`, and sets the `LabelDefinitions` property.
- `labelData` specifies the identifying information, position, and timestamps for the marked region of interest (ROI) labels and scene labels, and sets the `LabelData` property.

Properties

`DataSource` — Source of ground truth lidar data
`PointCloudSequenceSource` object | `VelodyneLidarSource` object | `LasFileSequenceSource` object | `RosbagSource` object

Source of ground truth lidar data, specified as a `PointCloudSequenceSource`, `VelodyneLidarSource`, `LasFileSequenceSource`, or `RosbagSource` object. This object contains the information that describes the source from which the ground truth lidar data was labeled. This table provides more details about the type of objects that you can specify.

Object Name	Data Source	Class Reference
`PointCloudSequenceSource`	Point cloud sequence folder	`vision.labeler.loading.PointCloudSequenceSource`

Object Name	Data Source	Class Reference
VelodyneLidarSource	Velodyne® packet capture (PCAP) file	vision.labeler.loading.VelodyneLidarSource
LasFileSequenceSource	LAS or LAZ file sequence folder	lidar.labeler.loading.LasFileSequenceSource
RosbagSource	Rosbag file	lidar.labeler.loading.RosbagSource

LabelDefinitions — Label definitions
table

This property is read-only.

Label definitions, specified as a table. To create this table, use one of these options.

- In the **Lidar Labeler** app, create label definitions, and then export them as part of a groundTruthLidar object.
- Use a labelDefinitionCreatorLidar object to generate a label definitions table. If you save this table to a MAT-file, you can then load the label definitions into a **Lidar Labeler** app session by selecting **Open** > **Label Definitions** from the app toolstrip.
- Create the label definitions table at the MATLAB command line.

This table describes the required and optional columns of the table specified in the LabelDefinitions property.

Column	Description	Required or Optional
Name	Strings or character vectors specifying the name of each label definition.	Required
Type	labelType enumerations that specify the type of each label definition. • For ROI label definitions, the only valid labelType enumeration is labelType.Cuboid. • For scene label definitions, the only valid labelType enumeration is labelType.Scene.	Required

Column	Description	Required or Optional
`LabelColor`	RGB triplets that specify the colors of the label definitions. Values are in the range [0, 1]. The color yellow (RGB triplet [1 1 0]) is reserved for the color of selected labels in the **Lidar Labeler** app.	Optional When you define labels in the **Lidar Labeler** app, you must specify a color. Therefore, an exported label definitions table always includes this column. When you create label definitions using the `labelDefinitionCreatorLidar` object without specifying colors, the returned label definition table includes this column, but all column values are empty.

Column	Description	Required or Optional
`Group`	Strings or character vectors specifying the group to which each label definition belongs.	Optional If you create the label definitions table at the MATLAB command line, you do not need to include a `Group` column. If you export label definitions from the **Lidar Labeler** app or create them using a `labelDefinitionCreatorLidar` object, the label definitions table includes this column, even if you did not specify groups. The app assigns each label definition a `Group` value of `'None'`.

Column	Description	Required or Optional
`Description`	Strings or character vectors that describe each label definition.	Optional If you create the label definitions table at the MATLAB command line, you do not need to include a `Description` column. If you export label definitions from the **Lidar Labeler** app or create them using a `labelDefinitionCreatorLidar` object, the label definitions table includes this column, even if you did not specify descriptions. The `Description` for these label definitions is an empty character vector.

<table>
<tr><th>Column</th><th>Description</th><th>Required or Optional</th></tr>
<tr><td>Hierarchy</td><td>Structures containing attribute information for each label definition.
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>AttributeName1,...,AttributeNameN</td><td>Attribute information

Each defined attribute has its own field, where the name of the field corresponds to the attribute name. The attribute value is a structure containing these fields:
• DefaultValue — Default value of the attribute, specified as a numeric scalar for Numeric attributes, a string for String attributes, or a logical scalar or empty array for Logical attributes. List attributes do not contain this field.
• ListItems — List items of the attribute, specified as a cell array of character vectors. Only List attributes contain this field.
• Description — Description of the attribute, specified as a character vector.</td></tr>
<tr><td>Type</td><td>Type of parent label for the attributes, specified as a string or character vector.</td></tr>
<tr><td>Description</td><td>Description of parent label for the attributes, specified as a string or character vector.</td></tr>
</table>
If a label definition does not contain attributes, then the table entry for that label definition is empty.</td><td>Optional

When you define sublabels or attributes in the Lidar Labeler app or the labelDefinitionCreatorMultisignal object, the generated label definitions table includes this column.</td></tr>
</table>

LabelData — Label data for each ROI and scene label
timetable

This property is read-only.

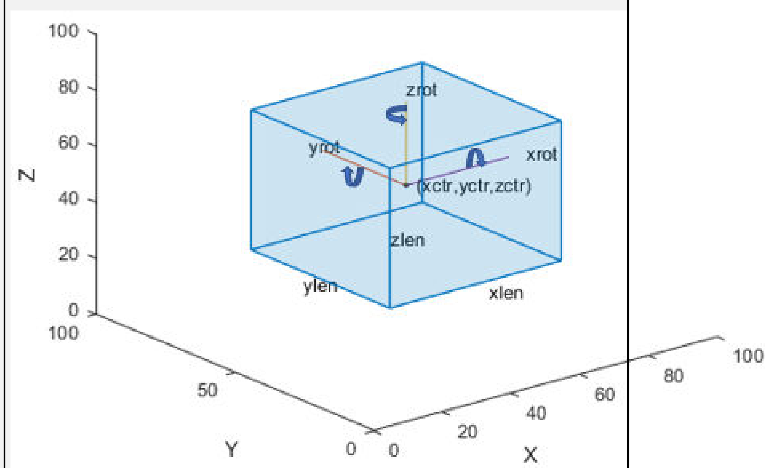
Label data for each ROI and scene label, specified as a `timetable`. Each column of `LabelData` holds labels for a single label definition and corresponds to the `Name` value for each row in `LabelDefinitions`. The storage format for the label data depends on the label type.

<table>
<tr><th>Label Type</th><th>Storage Format for Labels at Each Timestamp</th></tr>
<tr><td><code>labelType.Cuboid</code></td><td>

M-by-9 numeric matrix with rows of the form <code>[xctr, yctr, zctr, xlen, ylen, zlen, xrot, yrot, zrot]</code>, where:

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

The figure shows how these values determine the position of a cuboid.



</td></tr>
<tr><td><code>labelType.Scene</code></td><td>Logical vector, where <code>true</code> indicates the presence of the label at that timestamp.</td></tr>
</table>

If the `Cuboid` ROI label data includes attributes, then the labels at each timestamp must be specified as structures instead. The structure includes these fields.

Label Structure Field	Description
`Position`	Positions of the parent labels at the given timestamp The format of `Position` for labels of type `Cuboid` is described in the previous table.

Label Structure Field	Description
`AttributeName1,...,AttributeNameN`	Attributes of the parent labels Each defined attribute has its own field, where the name of the field corresponds to the attribute name. The attribute value is a character vector for a `List` or `String` attribute, a numeric scalar for a `Numeric` attribute, or a logical scalar for a `Logical` attribute. If the attribute is unspecified, then the attribute value is an empty vector.

Object Functions

changeFilePaths	Change file paths in ground truth data
selectLabels	Select ground truth data by label name or type
selectLabelsByGroup	Select ground truth data by label group name
selectLabelsByName	Select ground truth data by label name
selectLabelsByType	Select ground truth data by label type

Examples

Create Ground Truth Lidar Object

Create ground truth data for a Velodyne lidar source that captures a car on the road. Specify the signal sources, label definitions, and ROI label data.

Create a Velodyne data source.

```
sourceName = fullfile(toolboxdir('vision'),'visiondata', ...
    'lidarData_ConstructionRoad.pcap');

sourceParams = struct();
sourceParams.DeviceModel = 'HDL32E';
sourceParams.CalibrationFile = fullfile(matlabroot,'toolbox','shared', ...
    'pointclouds','utilities','velodyneFileReaderConfiguration', ...
    'HDL32E.xml');
```

Load the data source.

```
dataSource = vision.labeler.loading.VelodyneLidarSource;
dataSource.loadSource(sourceName,sourceParams);
```

Create label definitions.

```
ldc = labelDefinitionCreatorLidar;
addLabel(ldc,'Car','Cuboid');
labelDefs = ldc.create;
```

Create ground truth data for lidar sequence.

```
numPCFrames = numel(dataSource.Timestamp{1});
carData = cell(numPCFrames,1);
carData{1} = [1.0223 13.2884 1.1456 8.3114 3.8382 3.1460 0 0 0];
lidarData = timetable(dataSource.Timestamp{1},carData, ...
    'VariableNames',{'Car'});
```

Create the ground truth lidar object.

```
gTruth = groundTruthLidar(dataSource,labelDefs,lidarData)
```

```
gTruth = 

  groundTruthLidar with properties:

          DataSource: [1×1 vision.labeler.loading.VelodyneLidarSource]
    LabelDefinitions: [1×5 table]
           LabelData: [1238×1 timetable]
```

See Also

Objects
attributeType | labelDefinitionCreatorLidar | labelType

Introduced in R2020b

changeFilePaths

Change file paths in ground truth data

Syntax

```
unresolvedPaths = changeFilePaths(gTruth,alternativePaths)
```

Description

`unresolvedPaths = changeFilePaths(gTruth,alternativePaths)` changes the file paths in a `groundTruthLidar` object `gTruth` based on the specified pairs of current paths and alternative paths `alternativePaths`. If `gTruth` is a vector of `groundTruthLidar` objects, the function changes the file paths across all objects. The function returns the unresolved paths in `unresolvedPaths`. An unresolved path is any current path in `alternativePaths` not found in `gTruth` or any alternative path in `alternativePaths` not found at the specified path location. In both cases, `unresolvedPaths` returns only the current paths.

Examples

Change File Path in Ground Truth Lidar Object

Change the file paths to the data sources in a `groundTruthLidar` object.

Load a `groundTruthLidar` object containing multiple labels of groups, types and names into the workspace. The data source contains the file paths corresponding to the point cloud sequence showing multiple vehicles. MATLAB® displays a warning that the path to the data source cannot be found.

```
load('groundTruthLidar.mat');
```

```
Warning: The data source for the following source names could not be loaded. C:\Source
```

Display the current path to the data source.

```
gTruth.DataSource
```

```
ans = 
  PointCloudSequenceSource with properties:

            Name: "Point Cloud Sequence"
     Description: "A PointCloud sequence reader"
      SourceName: "C:\Source"
    SourceParams: [1×1 struct]
      SignalName: "Source"
      SignalType: PointCloud
       Timestamp: {[0 sec]}
      NumSignals: 1
```

Specify the current path to the data source and an alternative path and store these paths in a cell array. Use the `changeFilePaths` function to update the data source path based on the paths in the cell array.

The function updates the paths for all labels. As the function resolves all paths, it returns an empty array of unresolved paths.

```
currentPathDataSource = "C:\Source";
newPathDataSource = fullfile(matlabroot, 'toolbox', 'lidar', 'lidardata');
alternativeFilePaths = {[currentPathDataSource newPathDataSource]};
unresolvedPaths = changeFilePaths(gTruth, alternativeFilePaths)
```

```
unresolvedPaths =

     []
```

To view the new data source path, use the `gTruth.DataSource` command.

Input Arguments

`gTruth` — Ground truth lidar data
`groundTruthLidar` object | vector of `groundTruthLidar` objects

Ground truth lidar data, specified as a `groundTruthLidar` object or vector of `groundTruthLidar` objects.

`alternativePaths` — Alternative file paths
two-element row vector of strings | cell array of two-element row vector of strings

Alternative file paths, specified as a two-element row vector of strings or cell array of two-element row vectors of strings, where each vector is of the form [$p_{current}$ p_{new}].

- $p_{current}$ is a current file path in `gTruth`. This file path can be from the data source or pixel label data of the `gTruth` input. Specify $p_{current}$ using backslashes as the path separators.
- p_{new} is the new path to which to change $p_{current}$. Specify p_{new} using either forward slashes or backslashes as the path separators.

You can specify alternative paths to signal data sources. The `DataSource` property of `gTruth` contains one `groundTruthLidar` object per signal. The `changeFilePaths` function updates the signal paths stored in these objects.

If `gTruth` is a vector of `groundTruthLidar` objects, the function changes the file paths across all objects.

Output Arguments

`unresolvedPaths` — Unresolved file paths
string array

Unresolved file paths, returned as a string array. If the `changeFilePaths` function cannot find either the specified current path in the `gTruth` input or the specified new path in the specified path location, then it returns the unresolved current path.

If the function finds and resolves all file paths, then it returns `unresolvedPaths` as an empty string array.

See Also
`groundTruthLidar`

Introduced in R2020b

selectLabels

Select ground truth data by label name or type

Syntax

```
gtLabel = selectLabels(gTruth,labels)
```

Description

`gtLabel = selectLabels(gTruth,labels)` selects ground truth data of the specified label names or types `labels` from a `groundTruthLidar` object `gTruth`. The function returns a corresponding `groundTruthLidar` object `gtLabel` that contains only the selected labels. If `gTruth` is a vector of `groundTruthLidar` objects, then the function returns a vector of corresponding `groundTruthLidar` objects that contain only the selected labels.

Examples

Select Ground Truth Lidar Labels by Label Name or Label Type

Load a `groundTruthLidar` object containing labels of various groups, types, and names into the workspace.

```
lidarDir = fullfile(matlabroot,'toolbox','lidar','lidardata','lidarLabeler');
addpath(lidarDir)
load('lidarLabelerGTruth.mat')
```

Inspect the label definitions. The object contains label definitions of types `Cuboid` and `Scene` with various label names.

```
lidarLabelerGTruth.LabelDefinitions
```

```
ans =

  5×5 table

         Name           Type       LabelColor         Group       Description
    ______________    ______    _____________    ___________    ___________

    {'car'       }    Cuboid    {1×3 double}    {'vehicle'}    {0×0 char}
    {'bike'      }    Cuboid    {1×3 double}    {'vehicle'}    {0×0 char}
    {'pole'      }    Cuboid    {1×3 double}    {'None'   }    {0×0 char}
    {'vegetation'}    Cuboid    {1×3 double}    {'None'   }    {0×0 char}
    {'road'      }    Scene     {1×3 double}    {'None'   }    {0×0 char}
```

Create a new `groundTruthLidar` object that contains only the label definitions with the name `"car"`.

```
labelNames = "car";
gtLidarLabel = selectLabels(lidarLabelerGTruth,labelNames);
```

```
gtLidarLabel =

  groundTruthLidar with properties:

          DataSource: [1×1 vision.labeler.loading.PointCloudSequenceSource]
```

```
    LabelDefinitions: [1×5 table]
           LabelData: [1×1 timetable]
```

View the label definitions of the returned `groundTruthLidar` object.

```
gtLidarLabel.LabelDefinitions
```

```
ans =

  1×5 table

     Name        Type       LabelColor          Group        Description
    _______     ______     ____________     ___________     ___________

    {'car'}     Cuboid     {1×3 double}     {'vehicle'}     {0×0 char}
```

Create a new `groundTruthLidar` object that contains the label definitions from `lidarLabelerGTruth` for only the labels of type `Cuboid`.

```
labelType = labelType.Cuboid;
gtLidarLabel = selectLabels(lidarLabelerGTruth,labelType)
```

```
gtLidarLabel =

  groundTruthLidar with properties:

          DataSource: [1×1 vision.labeler.loading.PointCloudSequenceSource]
    LabelDefinitions: [4×5 table]
           LabelData: [1×4 timetable]
```

View the label definitions of the returned `groundTruthLidar` object.

```
gtLidarLabel.LabelDefinitions
```

```
ans =

  4×5 table

         Name          Type       LabelColor          Group        Description
    ______________    ______    ____________     ___________     ___________

    {'car'       }    Cuboid    {1×3 double}     {'vehicle'}     {0×0 char}
    {'bike'      }    Cuboid    {1×3 double}     {'vehicle'}     {0×0 char}
    {'pole'      }    Cuboid    {1×3 double}     {'None'   }     {0×0 char}
    {'vegetation'}    Cuboid    {1×3 double}     {'None'   }     {0×0 char}
```

Input Arguments

`gTruth` — Ground truth lidar data
`groundTruthLidar` object | vector of `groundTruthLidar` objects

Ground truth lidar data, specified as a `groundTruthLidar` object or vector of `groundTruthLidar` objects.

`labels` — Label names or types
one or more label names | one or more label types

Label names or types, specified as one or more label names or one or more label types. Specify one or more label names as a character vector, string scalar, cell array of character vectors, or vector of strings. Specify one or more label types as a `labelType` enumeration or vector of `labelType` enumerations.

To view all distinct label names in a `groundTruthLidar` object, enter the first of these commands at the MATLAB command prompt. To view all distinct label types in a `groundTruthLidar` object, enter the second.

```
unique(gTruth.LabelDefinitions.Name)
unique(gTruth.LabelDefinitions.Type)
```

Example: `'car'`

Example: `"car"`

Example: `{'car','lane'}`

Example: `["car" "lane"]`

Example: `labelType.Cuboid`

Example: `[labelType.Cuboid labelType.Scene]`

Output Arguments

`gtLabel` — Ground truth with only selected labels
`groundTruthLidar` object | vector of `groundTruthLidar` objects

Ground truth with only the selected labels, returned as a `groundTruthLidar` object or vector of `groundTruthLidar` objects.

Each `groundTruthLidar` object in the `gtLabel` output corresponds to a `groundTruthLidar` object in the `gTruth` input. The returned objects contain only those labels from the input ground truth objects that are of the label types or the label names specified in the `labels` input.

See Also

Objects
`groundTruthLidar`

Functions
`selectLabelsByGroup` | `selectLabelsByType` | `selectLabelsByName`

Introduced in R2020b

selectLabelsByGroup

Select ground truth data by label group name

Syntax

`gtLabel = selectLablesByGroup(gTruth,labelGroups)`

Description

`gtLabel = selectLablesByGroup(gTruth,labelGroups)` selects ground truth data with the specified label group names `labelGroups` from a `groundTruthLidar` object `gTruth`. The function returns a corresponding `groundTruthLidar` object `gtLabel` that contains only the selected labels. If `gTruth` is a vector of `groundTruthLidar` objects, then the function returns a vector of corresponding `groundTruthLidar` objects that contain only the selected labels.

Examples

Select Ground Truth Lidar Labels by Group Name

Load a `groundTruthLidar` object containing multiple labels of groups, types and names.

```
lidarDir = fullfile(matlabroot,'toolbox','lidar','lidardata','lidarLabeler');
addpath(lidarDir)
load('lidarLabelerGTruth.mat')
```

Inspect the label definitions. The object contains two label definitions in a `'vehicle'` group. Ungrouped labels are in the group named `'None'`.

```
lidarLabelerGTruth.LabelDefinitions
```

```
ans =

  5×5 table

        Name          Type       LabelColor         Group        Description
    ______________    ______    ____________    ___________    ___________

    {'car'       }    Cuboid    {1×3 double}    {'vehicle'}    {0×0 char}
    {'bike'      }    Cuboid    {1×3 double}    {'vehicle'}    {0×0 char}
    {'pole'      }    Cuboid    {1×3 double}    {'None'   }    {0×0 char}
    {'vegetation'}    Cuboid    {1×3 double}    {'None'   }    {0×0 char}
    {'road'      }    Scene     {1×3 double}    {'None'   }    {0×0 char}
```

Create a new `groundTruthLidar` object that contains only the label definitions in the group `'Vehicle'` group.

```
groupNames = 'vehicle';
gtLidarLabel = selectLabelsByGroup(lidarLabelerGTruth,groupNames)
```

```
gtLidarLabel =

  groundTruthLidar with properties:

          DataSource: [1×1 vision.labeler.loading.PointCloudSequenceSource]
    LabelDefinitions: [2×5 table]
           LabelData: [1×2 timetable]
```

View the labels returned by the function.

```
gtLidarLabel.LabelDefinitions
```

```
ans =

  2×5 table

      Name        Type       LabelColor         Group        Description
    ________     ______    ____________    ___________    ___________

    {'car' }    Cuboid    {1×3 double}    {'vehicle'}    {0×0 char}
    {'bike'}    Cuboid    {1×3 double}    {'vehicle'}    {0×0 char}
```

Input Arguments

gTruth — Ground truth lidar data
groundTruthLidar object | vector of groundTruthLidar objects

Ground truth lidar data, specified as a groundTruthLidar object or vector of groundTruthLidar objects.

labelGroups — Label group names
character vector | string scalar | cell array of character vectors | vector of strings

Label group names, specified as a character vector, string scalar, cell array of character vectors, or vector of strings.

To view all distinct label group names in a groundTruthLidar object, enter this command at the MATLAB command prompt.

```
unique(gTruth.LabelDefinitions.Group)
```

Example: 'Vehicles'

Example: "Vehicles"

Example: {'Vehicles','Signs'}

Example: ["Vehicles" "Signs"]

Output Arguments

gtLabel — Ground truth with only selected labels
groundTruthLidar object | vector of groundTruthLidar objects

Ground truth with only the selected labels, returned as a groundTruthLidar object or vector of groundTruthLidar objects.

Each groundTruthLidar object in the gtLabel output corresponds to a groundTruthLidar object in the gTruth input. The returned objects contain only those labels from the input ground truth objects that are of the label groups specified by the labelGroup input.

See Also

Objects
`groundTruthLidar`

Functions
`selectLabels` | `selectLabelsByType` | `selectLabelsByName`

selectLabelsByName

Select ground truth data by label name

Syntax

`gtLabel = selectLabelsByName(gTruth,labelNames)`

Description

`gtLabel = selectLabelsByName(gTruth,labelNames)` selects ground truth data of the specified label names `labelNames` from a `groundTruthLidar` object `gTruth`. The function returns a corresponding `groundTruthLidar` object `gtLabel` that contains only the selected labels. If `gTruth` is a vector of `groundTruthLidar` objects, then the function returns a vector of corresponding `groundTruthLidar` objects that contain only the selected labels.

Examples

Select Ground Truth Lidar Labels by Label Name

Load a `groundTruthLidar` object containing labels of various groups, types, and names.

```
lidarDir = fullfile(matlabroot,'toolbox','lidar','lidardata','lidarLabeler');
addpath(lidarDir)
load('lidarLabelerGTruth.mat')
```

Inspect the label definitions. The object contains label definitions with various names.

```
lidarLabelerGTruth.LabelDefinitions
```

```
ans =

  5×5 table

        Name           Type       LabelColor        Group       Description
    ______________    ______    ____________    ___________    ___________

    {'car'       }    Cuboid    {1×3 double}    {'vehicle'}    {0×0 char}
    {'bike'      }    Cuboid    {1×3 double}    {'vehicle'}    {0×0 char}
    {'pole'      }    Cuboid    {1×3 double}    {'None'   }    {0×0 char}
    {'vegetation'}    Cuboid    {1×3 double}    {'None'   }    {0×0 char}
    {'road'      }    Scene     {1×3 double}    {'None'   }    {0×0 char}
```

Create a new `groundTruthLidar` object that contains only the label definitions with the name `'car'`.

```
labelNames = 'car';
gtLidarLabel = selectLabelsByName(lidarLabelerGTruth,labelNames)
```

```
gtLidarLabel =

  groundTruthLidar with properties:

          DataSource: [1×1 vision.labeler.loading.PointCloudSequenceSource]
    LabelDefinitions: [1×5 table]
           LabelData: [1×1 timetable]
```

View the label definitions of the returned groundTruthLidar object.

```
gtLidarLabel.LabelDefinitions
```

```
ans =

  1×5 table

     Name       Type       LabelColor        Group        Description
    _______    ______    ____________    ___________    ___________

    {'car'}    Cuboid    {1×3 double}    {'vehicle'}    {0×0 char}
```

Input Arguments

gTruth — Ground truth lidar data
groundTruthLidar object | vector of groundTruthLidar objects

Lidar ground truth data, specified as a groundTruthLidar object or vector of groundTruthLidar objects.

labelNames — Label names
character vector | string scalar | cell array of character vectors | vector of strings

Label names, specified as a character vector, string scalar, cell array of character vectors, or vector of strings.

To view all distinct label names in a groundTruthLidar object gTruth, enter this command at the MATLAB command prompt.

```
unique(gTruth.LabelDefinitions.Name)
```

Example: 'car'

Example: "car"

Example: {'car','lane'}

Example: ["car" "lane"]

Output Arguments

gtLabel — Ground truth with only selected labels
groundTruthLidar object | vector of groundTruthLidar objects

Ground truth with only the selected labels, returned as a groundTruthLidar object or vector of groundTruthLidar objects.

Each groundTruthLidar object in gtLabel corresponds to a groundTruthLidar object in the gTruth input. The returned objects contain only the labels that are of the label names specified by the labelNames input.

See Also

Objects
groundTruthLidar

Functions
`selectLabels` | `selectLabelsByGroup` | `selectLabelsByType`

Introduced in R2020b

selectLabelsByType

Select ground truth data by label type

Syntax

```
gtLabel = selectLabelsByType(gTruth,labelTypes)
```

Description

`gtLabel = selectLabelsByType(gTruth,labelTypes)` selects labels of the types specified by `labelTypes` from a `groundTruthLidar` object `gTruth`. The function returns a corresponding `groundTruthLidar` object `gtLabel` that contains only the selected labels. If `gTruth` is a vector of `groundTruthLidar` objects, then the function returns a vector of corresponding `groundTruthLidar` objects that contain only the selected labels.

Examples

Select Ground Truth Lidar Labels by Label Type

Load a `groundTruthLidar` object containing labels of various groups, types, and names into the workspace.

```
lidarDir = fullfile(matlabroot,'toolbox','lidar','lidardata','lidarLabeler');
addpath(lidarDir)
load('lidarLabelerGTruth.mat')
```

Inspect the label definitions. The object contains label definitions of type `Cuboid` and `Scene`.

```
lidarLabelerGTruth.LabelDefinitions
```

```
ans =

  5×5 table

        Name           Type       LabelColor        Group        Description
    ______________    ______    ____________    ___________    ___________

    {'car'       }    Cuboid    {1×3 double}    {'vehicle'}    {0×0 char}
    {'bike'      }    Cuboid    {1×3 double}    {'vehicle'}    {0×0 char}
    {'pole'      }    Cuboid    {1×3 double}    {'None'   }    {0×0 char}
    {'vegetation'}    Cuboid    {1×3 double}    {'None'   }    {0×0 char}
    {'road'      }    Scene     {1×3 double}    {'None'   }    {0×0 char}
```

Create a new `groundTruthLidar` object that contains only the label definitions with the type `'Cuboid'`.

```
labelType = labelType.Cuboid;
gtLidarLabel = selectLabelsByType(lidarLabelerGTruth,labelType)
```

```
 =

  groundTruthLidar with properties:

          DataSource: [1×1 vision.labeler.loading.PointCloudSequenceSource]
    LabelDefinitions: [4×5 table]
           LabelData: [1×4 timetable]
```

View the label definitions of the returned `groundTruthLidar` object.

```
lidarLabelerGTruth.LabelDefinitions
```

```
ans =

  4×5 table

        Name          Type      LabelColor        Group       Description
    ______________    ______    ____________    ___________    ___________

    {'car'       }    Cuboid    {1×3 double}    {'vehicle'}    {0×0 char}
    {'bike'      }    Cuboid    {1×3 double}    {'vehicle'}    {0×0 char}
    {'pole'      }    Cuboid    {1×3 double}    {'None'   }    {0×0 char}
    {'vegetation'}    Cuboid    {1×3 double}    {'None'   }    {0×0 char}
```

Input Arguments

`gTruth` — Ground truth lidar data
`groundTruthLidar` object | vector of `groundTruthLidar` objects

Lidar ground truth data, specified as a `groundTruthLidar` object or vector of `groundTruthLidar` objects.

`labelTypes` — Label types
`labelType` enumeration | vector of `labelType` enumerations

Label types, specified as a `labelType` enumeration or vector of `labelType` enumerations.

To view all distinct label types in a `groundTruthLidar` object, enter this command at the MATLAB command prompt.

```
unique(gTruth.LabelDefinitions.LabelType)
```

Example: `labelType.Cuboid`

Example: `[labelType.Cuboid labelType.Scene]`

Output Arguments

`gtLabel` — Ground truth with only selected labels
`groundTruthLidar` object | vector of `groundTruthLidar` objects

Ground truth with only the selected labels, returned as a `groundTruthLidar` object or vector of `groundTruthLidar` objects.

Each `groundTruthLidar` object in `gtLabel` corresponds to a `groundTruthLidar` object in the `gTruth` input. The returned objects contain only the labels that are of the label types specified by the `labelTypes` input.

See Also

Objects
`groundTruthLidar`

Functions
`selectLabels` | `selectLabelsByGroup` | `selectLabelsByName`

Introduced in R2020b

ibeoLidarReader

Ibeo data container (IDC) file reader

Description

Ibeo Automotive Systems is a manufacturer of lidar sensor-based devices. The data captured by these devices is stored in IDC files. An IDC file reader object reads Ibeo FUSION SYSTEM or ECU scan data and Ibeo point cloud plane data from IDC files.

The reader currently supports message data types `0x2205` and `0x7510` in IDC files. These data types represent the Ibeo FUSION SYSTEM or ECU scan data and Ibeo point cloud plane data, respectively.

Creation

Syntax

```
ibeoReader = ibeoLidarReader(fileName)
```

Description

`ibeoReader = ibeoLidarReader(fileName)` creates an `ibeoLidarReader` object that reads metadata from IDC file.

Properties

`FileName` — Name of IDC file
character vector | string scalar

This property is read-only.

Name of IDC file, stored as a character vector or string scalar.

`MessageTypes` — List of supported message types
string scalar | vector of strings

This property is read-only.

List of supported message types available in the IDC file, stored as a string scalar or as a vector of strings. The possible values of this property are `"Scan"`, `"PointCloudPlane"`, or a vector containing both.

`NumMessages` — Total number of supported messages
positive integer

This property is read-only.

Total number of supported messages available in the IDC file, stored as a positive integer.

FileInfo — Information on supported messages
table object

This property is read-only.

Information on supported messages, stored as a table object.

MessageType	DataType	Description	NumMessages	TimeStamps
"Scan"	"0x2205"	"Ibeo FUSION SYSTEM/ECU scan data"	30	30-by-1 datetime arrays
"PointCloudPlane"	"0x7510"	"Ibeo point cloud plane"	40	40-by-1 datetime arrays

- MessageType – Type of message
- DataType – Data type of message.
- Description – Message data description.
- NumMessages – Number of messages available in the file.
- TimeStamps – Timestamp values for each message in the file, stored as a NumMessages-element column vector of datetime arrays.

Object Functions

readMessages   Read Ibeo scan data and point cloud plane messages

See Also

Functions
pcread | pcshow | readMessages

Objects
lasFileReader | pointCloud | velodyneFileReader

Introduced in R2020b

readMessages

Read Ibeo scan data and point cloud plane messages

Syntax

```
ptCloud = readMessages(ibeoReader)
[ptCloud,messageData] = readMessages(ibeoReader)
___ = readMessages(ibeoReader,Name,Value)
```

Description

`ptCloud = readMessages(ibeoReader)` reads Ibeo FUSION SYSTEM/ECU scan data and Ibeo point cloud plane messages from an Ibeo data container (IDC) file. The function returns an array of `pointCloud` objects, where each object contains individual message data.

`[ptCloud,messageData] = readMessages(ibeoReader)` additionally returns the message type and timestamp for each message. If the message is a point cloud plane message, the function also returns additional plane information.

`___ = readMessages(ibeoReader,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input argument. For example, `'Messages',"Scan"` sets the message type to read from the IDC file to `"Scan"`.

Input Arguments

`ibeoReader` — IDC file reader
`ibeoLidarReader` object

IDC file reader, specified as an `ibeoLidarReader` 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: `'Messages',"Scan"` sets the `readMessages` function to only read Ibeo scan data messages from the IDC file.

`Messages` — Message types to read
`["Scan" "PointCloudPlane"]` (default) | string scalar | vector of strings | character vector | cell array of character vectors

Message types to read from the IDC file, specified as the comma-separated pair consisting of `'Messages'` and a string scalar, vector of strings, character vector, or a cell array of character vectors. Each element must be a valid message type.

Valid Message Types

- `"Scan"`

- "PointCloudPlane"

Data Types: string | char | cell

Time — Timestamps of messages
total file duration (default) | datetime arrays | 2-element vector of datetime arrays

Timestamps of messages, specified as the comma-separated pair consisting of 'Time' and one of these options-

- datetime array – Represents a single timestamp
- 1-by-2 datetime array – Represents all timestamps in the range [*startTime endTime*].

Data Types: datetime

Output Arguments

ptCloud — Point cloud array
array of pointCloud objects

Point cloud array, returned as an array of pointCloud objects. Each element of the returned array is a point cloud that contains the data of a single message.

messageData — Information on messages read from file
cell array of structures

Information on messages read from the file, returned as a cell array of structures. Each structure contains this information for a single message.

- MessageType – Type of message, returned as "Scan" or "PointCloudPlane".
- TimeStamp – Timestamp value for each message in the file, returned as a datetime array.

If the value of the MessageType field for a message is "PointCloudPlane", then the structure contains this additional plane information.

- Label – Classification type of all points in the point cloud, returned as one of these values.
  - "Undefined"
  - "ScanPoint"
  - "LanePoint"
  - "CurbstonePoint"
  - "GuardrailPoint"
  - "RoadmarkingPoint"
  - "OffRoadMarkingPoint"
- ReferencePoint – Reference point for the plane points, returned as a three-element vector that contains the longitude and latitude of the point in degrees and the altitude in meters.
- PlaneOrientation – Plane orientation, returned as a three-element vector that contains the yaw, pitch, and roll of the plane in degrees.

See Also

Functions
`pcread` | `pcshow`

Objects
`ibeoLidarReader` | `lasFileReader` | `pointCloud` | `velodyneFileReader`

Introduced in R2020b

labelDefinitionCreatorLidar

Store, modify, and create label definitions tables for lidar

Description

The `labelDefinitionCreatorLidar` object stores definitions of labels and attributes to label ground truth data for a lidar workflow. Use various “Object Functions” on page 2-48 to add, remove, modify, or display label definitions. Use the `create` object function to create a label definitions table from the `labelDefinitionCreatorLidar` object. You can use this label definitions table with the Lidar Labeler app.

Creation

Syntax

```
ldc = labelDefinitionCreatorLidar
ldc = labelDefinitionCreatorLidar(labelDefs)
```

Description

`ldc = labelDefinitionCreatorLidar` creates an empty label definition creator object, `ldc`, for the lidar workflow. Add label definitions to this object, as well as modify or remove them, using various “Object Functions” on page 2-48. Use the `info` object function to inspect the stored labels and attributes.

`ldc = labelDefinitionCreatorLidar(labelDefs)` creates a label definition creator object, `ldc`, for a lidar workflow that contains the definitions from the label definitions table `labelDefs`.

Input Arguments

`labelDefs` — Label definitions
table

Label definitions, returned as a table with up to eight columns. The possible columns are *Name*, *Type*, *Group*, *Description*, *LabelColor*, and *Hierarchy*. This table contains the definitions and attributes of labels used for labeling ground truth lidar data. For more details, see the `labelDefinitions` property of the `groundTruthLidar` object.

Object Functions

addLabel	Add label to label definition creator object for lidar workflow
addAttribute	Add attribute to label in label definition creator for lidar workflow
editLabelGroup	Modify label group name in label definition creator object for lidar workflow
editLabelDescription	Modify label description in label definition creator for lidar workflow
editAttributeDescription	Modify attribute description in label definition creator object for lidar workflow

editGroupName	Change group name in label definition creator for lidar workflow
removeLabel	Remove label from label definition creator for lidar workflow
removeAttribute	Remove attribute from label in label definition creator for lidar workflow
create	Create label definitions table from label definition creator object for lidar workflow
info	Display label or attribute information stored in label definition creator for lidar workflow

Examples

Create Label Definition Creator Object for Lidar Workflow and Add Label Definitions

Create an empty `labelDefinitionCreatorLidar` object.

```
ldc = labelDefinitionCreatorLidar;
```

Add a `Cuboid` label, `Vehicle`, to the label definition creator.

```
addLabel(ldc,'Vehicle','Cuboid')
```

Add a `Color` attribute to the `Vehicle` label as a list of three strings.

```
addAttribute(ldc,'Vehicle','Color','List',{'Red','White','Green'})
```

Display the details of the updated label definition creator object.

```
ldc
```

```
ldc = 

labelDefinitionCreatorLidar contains the following labels:

    Vehicle with 1 attributes and belongs to None group.    (info)

For more details about attributes, use the info method.
```

Create a label definitions table from the definition stored in the object.

```
labelDefs = create(ldc)
```

```
labelDefs =

  1×6 table

       Name         Type      LabelColor     Group       Description     Hierarchy  
    ___________    _______    __________    ________    ___________    ____________

    {'Vehicle'}    Cuboid     {0×0 char}    {'None'}       {' '}        {1×1 struct}
```

Create Label Definition Creator Object for Lidar Workflow from Label Definitions Table

Load a lidar label definitions table into the workspace.

```
lidarDir = fullfile(matlabroot,'toolbox','lidar','lidardata','lidarLabeler');
addpath(lidarDir)
load('lidarLabelerGTruth.mat')
```

Create a `labelDefinitionCreatorLidar` object from the label definitions table.

```
ldc = labelDefinitionCreatorLidar(lidarLabelerGTruth.LabelDefinitions)
```

```
ldc = 

labelDefinitionCreatorLidar contains the following labels:

    car with 0 attributes and belongs to vehicle group.    (info)
    bike with 0 attributes and belongs to vehicle group.    (info)
    pole with 0 attributes and belongs to None group.    (info)
    vegetation with 0 attributes and belongs to None group.    (info)
    road with 0 attributes and belongs to None group.    (info)

For more details about attributes, use the info method.
```

Add a new attribute to the `car` label.

```
addAttribute(ldc,'car','Color','List',{'Red','Green','Blue'})
```

Display the details of the updated `labelDefinitionCreatorLidar` object.

```
ldc
```

```
ldc = 

labelDefinitionCreatorLidar contains the following labels:

    car with 1 attributes and belongs to vehicle group.    (info)
    bike with 0 attributes and belongs to vehicle group.    (info)
    pole with 0 attributes and belongs to None group.    (info)
    vegetation with 0 attributes and belongs to None group.    (info)
    road with 0 attributes and belongs to None group.    (info)
```

See Also

Apps
Lidar Labeler

Objects
`groundTruthLidar`

Introduced in R2020b

addAttribute

Add attribute to label in label definition creator for lidar workflow

Syntax

```
addAttribute(ldc,labelName,attributeName,typeOfAttribute,attributeDefault)
addAttribute( ___ ,Name,Value)
```

Description

`addAttribute(ldc,labelName,attributeName,typeOfAttribute,attributeDefault)` adds an attribute with the specified name and type to the indicated label. The attribute is added to the hierarchy of the specified label in the `labelDefinitionCreatorLidar` object `ldc`.

`addAttribute( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Add Label and Attribute Using Label Definition Creator for Lidar Workflow

Create an empty `labelDefinitionCreatorLidar` object.

```
ldc = labelDefinitionCreatorLidar
```

Add a `Cuboid` label, `Vehicle`, to the label definition creator.

```
addLabel(ldc, 'Vehicle', 'Cuboid');
```

Add a `Color` attribute to the `Vehicle` label as a string.

```
addAttribute(ldc,'Vehicle','Color','String','Red')
```

Display the details of the updated label definition creator object.

```
ldc
```

```
ldc =

labelDefinitionCreatorLidar contains the following labels:

    Vehicle with 1 attributes and belongs to None group.    (info)

For more details about attributes, use the info method.
```

Display information about the label `Vehicle` using the `info` object function .

```
info(ldc,'Vehicle')
```

```
          Name: "Vehicle"
          Type: Cuboid
     LabelColor: {''}
```

```
        Group: "None"
   Attributes: "Color"
  Description: ' '
```

Display information about the `Color` attribute of the `Vehicle` label using the `info` object function.

```
info(ldc,'Vehicle/Color')
```

```
          Name: "Color"
          Type: String
  DefaultValue: 'Red'
   Description: ' '
```

Input Arguments

`ldc` — Label definition creator for lidar workflow
`labelDefinitionCreatorLidar` object

Label definition creator for the lidar workflow, specified as a `labelDefinitionCreatorLidar` object.

`labelName` — Label name
character vector | string scalar

Label name, specified as a character vector or string scalar. This sets the label to which to add the attribute.

`attributeName` — Attribute name
character vector | string scalar

Attribute name, specified as a character vector or string scalar. This sets the attribute to add to the label.

`typeOfAttribute` — Type of attribute
`attributeType` enumeration | character vector | string scalar

Type of attribute, specified using one of these options:

- `attributeType` enumeration — Specify the attribute as a `Numeric`, `Logical`, `String`, or `List` `attributeType` enumerator. For example, `attributeType.String` specifies a `String` attribute type.
- Character vector or string scalar — Specify a value that partially or fully matches one of the `attributeType` enumerators. For example, `Str` specifies a String attribute type.

`attributeDefault` — Default value of attribute
valid attribute value

Default value of the attribute, specified as a valid attribute value depending on the value of the `typeOfAttribute` argument:

- Numeric — Specify the value as a numeric scalar.
- Logical — Specify the value as a logical scalar.
- String — Specify the value as a character vector or string scalar.

- List — Specify the value as a cell array of character vectors or string scalars. The first element of the cell array is the default value.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'Description','car' sets the description of the added label attribute to 'car'.

`Description` — Attribute description
`' '` (default) | character vector | string scalar

Attribute description, specified as the comma-separated pair consisting of `'Description'` and a character vector or string scalar. Use this name-value pair argument to describe the attribute.

See Also

Objects
`labelDefinitionCreatorLidar`

Functions
`addLabel` | `editAttributeDescription` | `removeAttribute`

Introduced in R2020b

addLabel

Add label to label definition creator object for lidar workflow

Syntax

```
addLabel(ldc,labelName,typeOfLabel)
addLabel( ___ ,Name,Value)
```

Description

`addLabel(ldc,labelName,typeOfLabel)` adds a label with the specified name and type to the `labelDefinitionCreatorLidar` object `ldc`.

`addLabel( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax. For example, `Group`, `truck` sets the group of the added label to `truck`.

Examples

Add Labels Using Label Definition Creator for Lidar Workflow

Create an empty `labelDefinitionCreatorLidar` object.

```
ldc = labelDefinitionCreatorLidar;
```

Add a `Cuboid` label, `Vehicle`, to the label definition creator object.

```
addLabel(ldc, 'Vehicle', 'Cuboid');
```

Add a `Scene` label, `Bike`, to the object.

```
addLabel(ldc, 'Bike', 'Scene');
```

Display the details of the updated label definition creator object.

```
ldc
```

```
ldc = 

labelDefinitionCreatorLidar contains the following labels:

    Vehicle with 0 attributes and belongs to None group.    (info)
    Bike with 0 attributes and belongs to None group.    (info)

For more details about attributes, use the info method.
```

Display information about the `Vehicle` label using the `info` object function.

```
info(ldc,'Vehicle')
```

```
           Name: "Vehicle"
           Type: Cuboid
```

```
     LabelColor: {''}
          Group: "None"
     Attributes: []
    Description: ' '
```

Add Label with Additional Details

Create an empty `labelDefinitionCreatorLidar` object.

```
ldc = labelDefinitionCreatorLidar;
```

Add a `Cuboid` label, `Vehicle`, to the label definition creator object. Include `Group` and `LabelColor` information for the label.

```
addLabel(ldc, 'Vehicle', 'Cuboid','Group',"Transport",'LabelColor',[1 0 0]);
```

Add a `Scene` label, `TrafficSign`, to the object. Include Group information for the label.

```
addLabel(ldc, 'TrafficSign', 'Scene','Group',"Data");
```

Display the details of the updated label definition creator object.

```
ldc
```

```
ldc = 

labelDefinitionCreatorLidar contains the following labels:

    Vehicle with 0 attributes and belongs to Transport group.    (info)
    TrafficSign with 0 attributes and belongs to Data group.    (info)

For more details about attributes, use the info method.
```

Display information about the `Vehicle` label using the `info` object function.

```
info(ldc,'Vehicle')
```

```
           Name: "Vehicle"
           Type: Cuboid
     LabelColor: {[1 0 0]}
          Group: "Transport"
     Attributes: []
    Description: ' '
```

Input Arguments

`ldc` — Label definition creator for lidar workflow
`labelDefinitionCreatorLidar` object

Label definition creator for the lidar workflow, specified as a `labelDefinitionCreatorLidar` object.

`labelName` — Label name
character vector | string scalar

Label name, specified as a character vector or string scalar. This sets the name of the label in the label definition creator object.

typeOfLabel — Type of label
labelType enumerator | character vector | string scalar

Type of label, specified using one of these options. For example, `labelType.Cuboid` specifies a `Cuboid` label type.

- `labelType` enumeration — Specify the type of label as a `Scene` or `Cuboid` `labelType` enumerator.
- Character vector or string scalar — Specify a value that partially or fully matches one of the `labelType` enumerators. For example, `Cub` specifies a `Cuboid` label type.

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`.

For example, `Group`, `truck` sets the group of the added label to `truck`.

Group — Group name
'None' (default) | character vector | string scalar

Group name, specified as a comma-separated pair consisting of `'Group'` and the character vector or string scalar. Use this name-value pair arguments to specify a name for a group of labels.

Description — Label description
' ' (default) | character vector | string scalar

Label description, specified as a comma-separated pair consisting of `'Description'` and the character vector or string scalar. Use this name-value pair arguments to describe the label.

See Also

Objects
`labelDefinitionCreatorLidar`

Functions
`addAttribute` | `editLabelDescription` | `removeLabel`

Introduced in R2020b

create

Create label definitions table from label definition creator object for lidar workflow

Syntax

```
labelDefs = create(ldc)
```

Description

`labelDefs = create(ldc)` creates a label definitions table, `labelDefs`, from the `labelDefinitionCreatorLidar` object `ldc`. You can import the `labelDefs` table into the Lidar Labeler app to label ground truth lidar data.

Examples

Create Label Definitions Table from Label Definition Creator for Lidar Workflow

Create an empty `labelDefinitionCreatorLidar` object.

```
ldc = labelDefinitionCreatorLidar;
```

Add a `Cuboid` label, `Vehicle`, to the label definition creator object.

```
addLabel(ldc, 'Vehicle', 'Cuboid', 'Description', 'Use this label for Cars and buses.')
```

Add a `logical` attribute, `IsCar`, to the `Vehicle` label.

```
addAttribute(ldc,'Vehicle','IsCar','logical',true,'Description','Type of vehicle')
```

Create a label definitions table from the definitions stored in the object.

```
labelDefs = create(ldc)
```

```
labelDefs =

  1×6 table

       Name          Type       LabelColor      Group                  Description                      Hierarchy
    ___________    ______    __________    ________    ________________________________________    ____________

    {'Vehicle'}    Cuboid    {0×0 char}    {'None'}    {'Use this label for Cars and buses.'}    {1×1 struct}
```

Input Arguments

ldc — Label definition creator for lidar workflow
`labelDefinitionCreatorLidar` object

Label definition creator for the lidar workflow, specified as a `labelDefinitionCreatorLidar` object. The object defines the labels and attributes used for generating the label definitions table `labelDefs`.

Output Arguments

labelDefs — Label definitions
table

Label definitions, returned as a table with up to eight columns. The possible columns are *Name*, *Type*, *Group*, *Description*, *LabelColor*, and *Hierarchy*. This table contains the definitions and attributes of labels used for labeling ground truth lidar data. For more details, see the `labelDefinitions` property of the `groundTruthLidar` object.

See Also

Objects
`labelDefinitionCreatorLidar`

Functions
`addAttribute` | `addLabel` | `info`

Introduced in R2020b

editAttributeDescription

Modify attribute description in label definition creator object for lidar workflow

Syntax

`editAttributeDescription(ldc,labelName,attributeName,description)`

Description

`editAttributeDescription(ldc,labelName,attributeName,description)` modifies the description of the specified attribute `attributeName` of the label `labelName`. The label must be contained within the `labelDefinitionCreatorLidar` object `ldc`.

Examples

Modify Attribute Description in Label Definition Creator for Lidar Workflow

Create an empty `labelDefinitionCreatorLidar` object.

```
ldc = labelDefinitionCreatorLidar;
```

Add a `Cuboid` label, `Vehicle`, to the label definition creator object.

```
addLabel(ldc, 'Vehicle', 'Cuboid');
```

Add a `Color` attribute to the `Vehicle` label.

```
addAttribute(ldc,'Vehicle','Color','String','Red')
```

Display the created attribute.

```
info(ldc,'Vehicle/Color')
```

```
            Name: "Color"
            Type: String
    DefaultValue: 'Red'
     Description: ' '
```

Modify the attribute description.

```
editAttributeDescription(ldc,'Vehicle','Color','Color of the vehicle in RGB format - [1 0 0]')
```

Display the attribute details to confirm the updated description field.

```
info(ldc,'Vehicle/Color')
```

```
            Name: "Color"
            Type: String
    DefaultValue: 'Red'
     Description: 'Color of the vehicle in format RGB - [1 0 0]'
```

Input Arguments

`ldc` — Label definition creator for lidar workflow
`labelDefinitionCreatorLidar` object

Label definition creator for the lidar workflow, specified as a `labelDefinitionCreatorLidar` object.

`labelName` — Label name
character vector | string scalar

Label name, specified as a character vector or string scalar. This identifies the label with which the attribute is associated.

`attributeName` — Attribute name
character vector | string scalar

Attribute name, specified as a character vector or string scalar. This identifies the attribute to modify.

`description` — Description
character vector | string scalar

Description, specified as a character vector or string scalar. This sets the new description for the attribute specified by the `attributeName`.

See Also

Objects
`labelDefinitionCreatorLidar`

Functions
`editLabelDescription`

Introduced in R2020b

editGroupName

Change group name in label definition creator for lidar workflow

Syntax

`editGroupName(ldc,oldname,newname)`

Description

`editGroupName(ldc,oldname,newname)` changes the existing group name `oldname` to the specified group name `newname`. This function changes the group name for all label definitions that have the group name `oldname`.

Examples

Edit Label Group in Label Definition Creator for Lidar Workflow

Create an empty `labelDefinitionCreatorLidar` object.

```
ldc = labelDefinitionCreatorLidar;
```

Add a `Cuboid` label, `Vehicle`, to the label definition creator object.

```
addLabel(ldc,'Vehicle','Cuboid')
```

Display information about the label.

```
info(ldc,'Vehicle')
```

```
           Name: "Vehicle"
           Type: Cuboid
     LabelColor: {''}
          Group: "None"
     Attributes: []
    Description: ' '
```

Edit the group name of the label.

```
editGroupName(ldc,'None','Transport')
```

Display the information of the label. Confirm that the `Group` field is updated.

```
info(ldc,'Vehicle')
```

```
           Name: "Vehicle"
           Type: Cuboid
     LabelColor: {''}
          Group: "Transport"
     Attributes: []
    Description: ' '
```

Input Arguments

ldc — Label definition creator for lidar workflow
labelDefinitionCreatorLidar object

Label definition creator for the lidar workflow, specified as a labelDefinitionCreatorLidar object.

oldname — Existing group name
character vector | string scalar

Existing group name, specified as a character vector or string scalar. This identifies group name to modify. The group name must already exist within the specified label definition creator object.

newname — New group name
character vector | string scalar

New group name, specified as a character vector or string scalar. This sets the new group name.

See Also

Objects
labelDefinitionCreatorLidar

Functions
editLabelDescription | editLabelGroup

Introduced in R2020b

editLabelDescription

Modify label description in label definition creator for lidar workflow

Syntax

`editLabelDescription(ldc,labelName,description)`

Description

`editLabelDescription(ldc,labelName,description)` modifies the description of the specified label `labelName`. The label must be contained within the `labelDefinitionCreatorLidar` object `ldc`.

Examples

Modify Label Description in Label Definition Creator for Lidar Workflow

Create an empty `labelDefinitionCreatorLidar` object.

```
ldc = labelDefinitionCreatorLidar;
```

Add a `Cuboid` label, `Vehicle`, to the label definition creator object.

```
addLabel(ldc,'Vehicle','Cuboid')
```

Display the label information using the `info` object function.

```
           Name: "Vehicle"
           Type: Cuboid
     LabelColor: {''}
          Group: "None"
     Attributes: []
    Description: ' '
```

Modify the description of the `Vehicle` label.

```
editLabelDescription(ldc, 'Vehicle', 'Use this label for cars and buses.')
```

Display the label information. Confirm that the `Description` field has been updated.

```
info(ldc, 'Vehicle')
```

```
           Name: "Vehicle"
           Type: Cuboid
     LabelColor: {''}
          Group: "None"
     Attributes: []
    Description: 'Use this label for cars and buses.'
```

Input Arguments

ldc — Label definition creator for lidar workflow
`labelDefinitionCreatorLidar` object

Label definition creator for the lidar workflow, specified as a `labelDefinitionCreatorLidar` object.

`labelName` — Label name
character vector | string scalar

Label name, specified as a character vector or string scalar. This identifies the label to update.

`description` — Description
character vector | string scalar

Description, specified as a character vector or string scalar. This sets the new description for the label specified by the `labelName` argument.

See Also

Objects
`labelDefinitionCreatorLidar`

Functions
`editAttributeDescription`

Introduced in R2020b

editLabelGroup

Modify label group name in label definition creator object for lidar workflow

Syntax

`editLabelGroup(ldc,labelName,groupName)`

Description

`editLabelGroup(ldc,labelName,groupName)` modifies the group name of the specified label identified by `labelName`. The label must be contained within the `labelDefinitionCreatorLidar` object `ldc`.

Examples

Modify Label Group Name in Label Definition Creator for Lidar Workflow

Create an empty `labelDefinitionCreatorLidar` object.

```
ldc = labelDefinitionCreatorLidar;
```

Add a `Cuboid` label, `Vehicle`, to the label definition creator object.

```
addLabel(ldc,'Vehicle','Cuboid','Group','Transport')
```

Add a `Cuboid` label, `Car`, to the label definition creator object.

```
addLabel(ldc,'Car','Cuboid','Group','Four Wheeler')
```

Display the label definition creator object.

```
ldc
```

```
ldc = 

labelDefinitionCreatorLidar contains the following labels:

    Vehicle with 0 attributes and belongs to Transport group.    (info)
    Car with 0 attributes and belongs to Four Wheeler group.    (info)

For more details about attributes, use the info method.
```

Change the group of the `Car` label from `Four Wheeler` to `Transport`.

```
editLabelGroup(ldc,'Car','Transport')
```

```
ldc = 

labelDefinitionCreatorLidar contains the following labels:

    Vehicle with 0 attributes and belongs to Transport group.    (info)
    Car with 0 attributes and belongs to Transport group.    (info)
```

```
For more details about attributes, use the info method.
```

Input Arguments

`ldc` — Label definition creator for lidar workflow
`labelDefinitionCreatorLidar` object

Label definition creator for the lidar workflow, specified as a `labelDefinitionCreatorLidar` object.

`labelName` — Label name
character vector | string scalar

Label name, specified as a character vector or string scalar. This identifies the label to modify.

`groupName` — Group name
character vector | string scalar

Group name, specified as a character vector or string scalar. This sets the group to which the function assigns the label specified by the `labelName` argument.

See Also

Objects
`labelDefinitionCreatorLidar`

Functions
`editGroupName` | `editLabelDescription`

Introduced in R2020b

info

Display label or attribute information stored in label definition creator for lidar workflow

Syntax

```
info(ldc,name)
infoStruct = info(ldc,name)
```

Description

`info(ldc,name)` displays information about the specified label or attribute `name` stored in the `labelDefinitionCreatorLidar` object `ldc`.

`infoStruct = info(ldc,name)` returns the information as a structure.

Examples

Save Definitions from Label Definition Creator for Lidar Workflow

Create an empty `labelDefinitionCreatorLidar` object.

```
ldc = labelDefinitionCreatorLidar;
```

Add a `Cuboid` label, `Vehicle`, with `Group` and `Description` information to the label definition creator object.

```
addLabel(ldc,'Vehicle','Cuboid','Group','Transport','Description','Use this label for cars and buses')
```

Create a structure array containing the label information.

```
infoStruct = info(ldc,'Vehicle')
```

```
infoStruct = 

  struct with fields:

           Name: "Vehicle"
           Type: Cuboid
     LabelColor: {''}
          Group: "Transport"
     Attributes: []
    Description: 'Use this label for cars and buses'
```

Input Arguments

`ldc` — Label definition creator for lidar workflow
`labelDefinitionCreatorLidar` object

Label definition creator for the lidar workflow, specified as a `labelDefinitionCreatorLidar` object.

`name` — Name of label or attribute
character vector | string scalar

Name of the label or attribute in the `ldc` object, specified as a character vector or string scalar. The form of the argument depends on the type of name specified.

- To specify a label, use the form '*labelName*'. For example, `'TrafficLight'` specifies the label with the label name `TrafficLight`.
- To specify an attribute, use the form '*labelName/attributeName*'. For example, `'TrafficLight/Active'` specifies the `Active` attribute of the label with the label name `TrafficLight`.

Output Arguments

`infoStruct` — Information structure
structure

Information structure, returned as a structure that contains the fields `Name`, `SignalType` (for labels), `LabelType` (for labels), `Type` (for attributes), `Description`, `Attributes` (when pertinent), `DefaultValue` (for attributes), and `ListItems` (for List attributes).

See Also

Objects
`labelDefinitionCreatorLidar`

Functions
`addLabel` | `create`

Introduced in R2020b

removeAttribute

Remove attribute from label in label definition creator for lidar workflow

Syntax

`removeAttribute(ldc,labelName,attributeName)`

Description

`removeAttribute(ldc,labelName,attributeName)` removes the specified attribute `attributeName`from the label `labelName` in the `labelDefinitionCreatorLidar` object `ldc`.

Examples

Remove Attribute from Label in Label Definition Creator Lidar

Create an empty `labelDefinitionCreatorLidar` object.

```
ldc = labelDefinitionCreatorLidar;
```

Add a `Cuboid` label, `Vehicle`, to the label definition creator object.

```
addLabel(ldc,'Vehicle','Cuboid')
```

Add a `String` attribute, `Color`, to the `Vehicle` label.

```
addAttribute(ldc,'Vehicle','Color','String','Red')
```

Add another `String` attribute, `Classification`, to the label.

```
addAttribute(ldc,'Vehicle','Classification','String','Car')
```

Display the label information using the `info` object function.

```
info(ldc,'Vehicle')
```

```
           Name: "Vehicle"
           Type: Cuboid
     LabelColor: {''}
          Group: "None"
     Attributes: ["Color"    "Classification"]
    Description: ' '
```

Remove an attribute from the `Vehicle` label.

```
removeAttribute(ldc,'Vehicle','Color')
```

Display the label information. Confirm that the `Attributes` field has been updated.

```
info(ldc,'Vehicle')
```

```
           Name: "Vehicle"
           Type: Cuboid
```

```
     LabelColor: {''}
          Group: "None"
     Attributes: "Classification"
    Description: ' '
```

Input Arguments

ldc — Label definition creator for lidar workflow
labelDefinitionCreatorLidar object

Label definition creator for the lidar workflow, specified as a labelDefinitionCreatorLidar object.

labelName — Label name
character vector | string scalar

Label name, specified as a character vector or string scalar. This identifies the label from which to remove the attribute.

attributeName — Attribute name
character vector | string scalar

Attribute name, specified as a character vector or string scalar. This identifies the attribute to remove from the label specified by the labelName argument.

See Also

Objects
labelDefinitionCreatorLidar

Functions
addAttribute | addLabel | removeLabel

Introduced in R2020b

removeLabel

Remove label from label definition creator for lidar workflow

Syntax

`removeLabel(ldc,labelName)`

Description

`removeLabel(ldc,labelName)` removes the specified label `labeName` from the `labelDefinitionCreatorLidar` object `ldc`.

Examples

Remove Label from Label Definition Creator for Lidar Workflow

Create an empty `labelDefinitionCreatorLidar` object.

```
ldc = labelDefinitionCreatorLidar;
```

Add a `Cuboid` label, `Vehicle`, to the label definition creator object.

```
addLabel(ldc,'Vehicle','Cuboid')
```

Add a `Cuboid` label, `Car`, to the object.

```
addLabel(ldc,'Car','Cuboid')
```

Display the label definition creator object.

```
ldc
```

```
ldc =

labelDefinitionCreatorLidar contains the following labels:

    Vehicle with 0 attributes and belongs to None group.    (info)
    Car with 0 attributes and belongs to None group.    (info)

For more details about attributes, use the info method.
```

Remove the `'Car'` label and display the object to confirm that the label has been removed.

```
removeLabel(ldc,'Car')
ldc
```

```
ldc =

labelDefinitionCreatorLidar contains the following labels:

    Vehicle with 0 attributes and belongs to None group.    (info)

For more details about attributes, use the info method.
```

Input Arguments

`ldc` — Label definition creator for lidar workflow
`labelDefinitionCreatorLidar` object

Label definition creator for the lidar workflow, specified as a `labelDefinitionCreatorLidar` object.

`labelName` — Label name
character vector | string scalar

Label name, specified as a character vector or string scalar. This identifies the label to remove from the label definition creator object.

See Also

Objects
`labelDefinitionCreatorLidar`

Functions
`addLabel` | `addAttribute` | `removeAttribute`

Introduced in R2020b

vision.labeler.loading.MultiSignalSource class

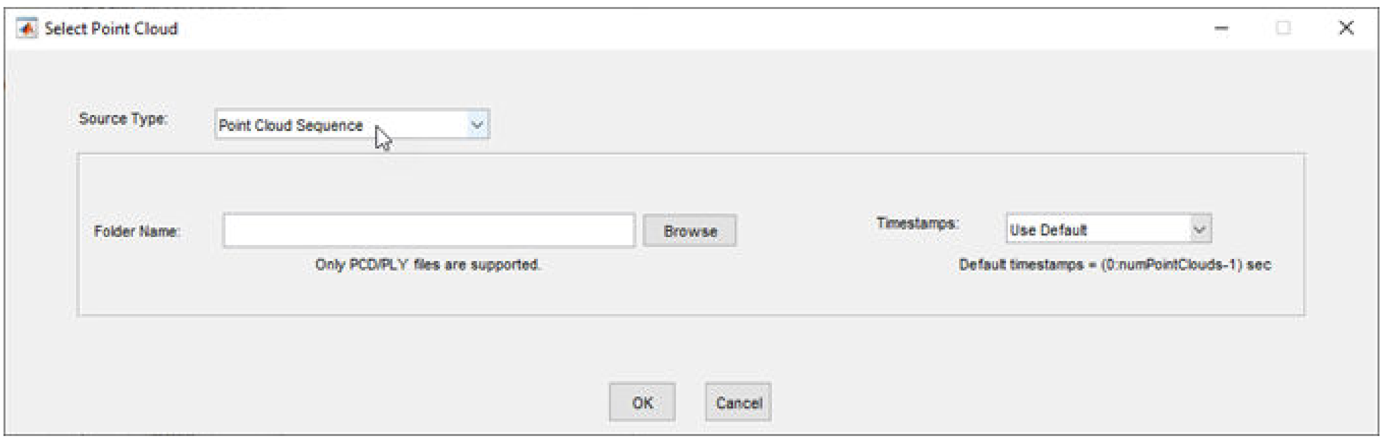
Package: `vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading`
Superclasses: `matlab.mixin.Heterogeneous`

Interface for loading signal data into Lidar Labeler app

Description

The `vision.labeler.loading.MultiSignalSource` class creates an interface for loading a point cloud signal from a data source into the **Lidar Labeler** app.

The interface created using this class enables you to customize the panel for loading data sources in the Select Point Cloud dialog box of the app. The figure shows a sample loading panel.



The class also provides an interface to read frames from loaded signals. The app renders these frames for labeling.

The class supports loading these data sources:

- `vision.labeler.loading.PointCloudSequenceSource` — Point cloud sequence folder
- `vision.labeler.loading.VelodyneLidarSource` — Velodyne packet capture (PCAP) file
- `lidar.labeler.loading.LasFileSequenceSource` — LAS or LAZ file
- `lidar.labeler.loading.RosbagSource` — Rosbag file

The `vision.labeler.loading.MultiSignalSource` class is a `handle` class.

Class Attributes

`Abstract`	`true`

For information on class attributes, see “Class Attributes”.

Properties

Name — Name of source type
string scalar

Name of the type of source that this class loads, specified as a string scalar.

Attributes:

GetAccess	public
Abstract	true
Constant	true
NonCopyable	true

Description — Description of class functionality
string scalar

Description of the functionality that this class provides, specified as a string scalar.

Attributes:

GetAccess	public
Abstract	true
Constant	true
NonCopyable	true

SourceName — Name of data source
string scalar

Name of the data source, specified as a string scalar. Typically, `SourceName` is the name of the file from which the signal is loaded.

Attributes:

GetAccess	public
SetAccess	protected

SourceParams — Parameters for loading signals from data source
structure

Parameters for loading signals from the data source into the app, specified as a structure. The fields of this structure contain values that the `loadSource` method requires to load the signal.

Attributes:

GetAccess	public
SetAccess	protected

SignalName — Names of signals in data source
string vector

Names of the signals that can be loaded from the data source, specified as a string vector.

Attributes:

GetAccess	public
SetAccess	protected

SignalType — Types of signals in data source
vector of vision.labeler.loading.SignalType enumerations

Types of the signals that can be loaded from the data source, specified as a vector of vision.labeler.loading.SignalType enumerations. Each signal listed in the SignalName property is of the type in the corresponding position of SignalType.

Attributes:

GetAccess	public
SetAccess	protected

Timestamp — Timestamps of signals in data source
cell array of duration vectors

Timestamps of the signals that can be loaded from the data source, specified as a cell array of duration vectors. Each signal listed in the SignalName property has the timestamps in the corresponding position of Timestamp.

Attributes:

GetAccess	public
SetAccess	protected

NumSignals — Number of signals in data source
nonnegative integer

Number of signals that can be read from the data source, specified as a nonnegative integer. NumSignals is equal to the number of signals in the SignalName property.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true
NonCopyable	true

Methods

Public Methods

customizeLoadPanel	customizeLoadPanel(sourceObj,panel) Customize the loading panel for the data source object. In the loading dialog box of the app, this method is invoked when you select the data source type from the **Source Type** list. Abstract: true

<table>
<tr><td><code>getLoadPanelData</code></td><td>
<code>[sourceName,sourceParams] = getLoadPanelData(sourceObj)</code>
Obtain the data needed to load the data source object currently selected in the loading panel. In the loading dialog box of the app, this method is invoked when you add a source. The method returns these outputs.

<code>sourceName</code> is a string capturing the name of the data source object.
<code>sourceParams</code> is a structure with fields containing the parameters required to load the data source object.

Both of these outputs are passed to the <code>loadSource</code> method.
<table><tr><td>Abstract</td><td>true</td></tr></table>
</td></tr>
<tr><td><code>loadSource</code></td><td>
<code>loadSource(sourceObj,sourceName,sourceParams)</code>
Load a data source object into the app. In the loading dialog box of the app, this method is invoked after you add a source and the <code>getLoadPanelData</code> method executes successfully. This method is also invoked when you load the data source object into the MATLAB workspace. When you load the data source object, MATLAB expects that the source has the name <code>sourceName</code> and parameters <code>sourceParams</code> that are needed to load that source and read data from it.
<table><tr><td>Abstract</td><td>true</td></tr></table>
</td></tr>
<tr><td><code>readFrame</code></td><td>
<code>frame = readFrame(sourceObj,signalName,tsIndex)</code>
Read a frame of data from a signal contained in a data source object at the specified timestamp index. The index must be in the bounds of the length of the timestamps for that signal.
<table><tr><td>Abstract</td><td>true</td></tr></table>
</td></tr>
<tr><td><code>loadPanelChecker</code></td><td>
<code>loadPanelChecker</code>
Check the load panel for the loading dialog box of the app. This method opens a dialog box similar to the loading dialog box that you open from the Open menu on the app toolstrip. Use this method to preview how the <code>customizeLoadPanel</code> method populates the loading panel for the selected data source object.
<table><tr><td>Static</td><td>true</td></tr></table>
</td></tr>
</table>

See Also

Apps
Lidar Labeler

Introduced in R2020b

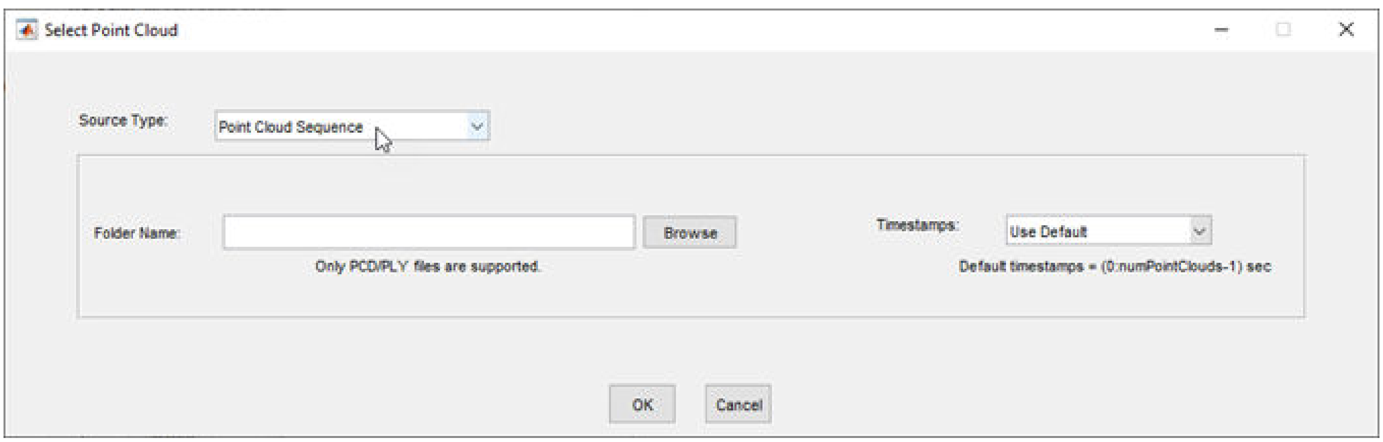
vision.labeler.loading.PointCloudSequenceSource class

Package: `vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading`
Superclasses: `vision.labeler.loading.MultiSignalSource`

Load signals from point cloud sequence sources into Lidar Labeler app

Description

The `vision.labeler.loading.PointCloudSequenceSource` class creates an interface for loading a signal from a point cloud sequence data source into the **Lidar Labeler** app. In the Select Point Cloud dialog box of the app, when **Source Type** is set to `Point Cloud Sequence`, this class controls the parameters in that dialog box.



To access this dialog box, in the app, select **Open > Add Point Cloud**.

This class loads point cloud sequences composed of PCD or PLY files.

The `vision.labeler.loading.PointCloudSequenceSource` class is a `handle` class.

Creation

When you export labels from a **Lidar Labeler** app session that contains a point cloud sequence source, the exported `groundTruthLidar` object stores an instance of this class in its `DataSource` property.

To create a `PointCloudSequenceSource` object programmatically, such as when programmatically creating a `groundTruthLidar` object, use the `vision.labeler.loading.PointCloudSequenceSource` function (described here).

Syntax

`pcSeqSource = vision.labeler.loading.PointCloudSequenceSource`

Description

`pcSeqSource = vision.labeler.loading.PointCloudSequenceSource` creates a `PointCloudSequenceSource` object for loading a signal from a point cloud sequence data source. To specify the data source and the parameters required to load the source, use the `loadSource` method.

Properties

`Name` — Name of source type
`"Point Cloud Sequence"` (default) | string scalar

Name of the type of source that this class loads, specified as a string scalar.

Attributes:

GetAccess	public
Constant	true
NonCopyable	true

`Description` — Description of class functionality
`"A PointCloud sequence reader"` (default) | string scalar

Description of the functionality that this class provides, specified as a string scalar.

Attributes:

GetAccess	public
Constant	true
NonCopyable	true

`SourceName` — Name of data source
`[]` (default) | string scalar

Name of the data source, specified as a string scalar. Typically, `SourceName` is the name of the file from which the signal is loaded.

Attributes:

GetAccess	public
SetAccess	protected

`SourceParams` — Parameters for loading point cloud sequence signal from data source
`[]` (default) | structure

Parameters for loading a point cloud sequence signal from a data source, specified as a structure.

This table describes the required and optional fields of the `SourceParams` structure.

Field	Description	Required or Optional
Timestamps	Timestamps for the point cloud sequence signal, specified as a cell array containing a single duration vector of timestamps. In the Select Point Cloud dialog box of the app, if you set the **Timestamps** parameter to From Workspace and read the timestamps from a variable in the MATLAB workspace, then the SourceParams property stores these timestamps in the Timestamps field.	Optional If you set the **Timestamps** parameter to Use Default and use the default timestamps for point cloud sequence signals, then the structure does not include this field, and the SourceParams property is empty, []. For point cloud sequence signals, the default timestamp duration vector has elements from 0 to the number of valid point cloud files minus 1. Units are in seconds.

Attributes:

GetAccess	public
SetAccess	protected

SignalName — Names of signals in data source

[] (default) | string vector

Names of the signals that can be loaded from the data source, specified as a string vector.

Attributes:

GetAccess	public
SetAccess	protected

SignalType — Types of signals in data source

[] (default) | vector of vision.labeler.loading.SignalType enumerations

Types of the signals that can be loaded from the data source, specified as a vector of vision.labeler.loading.SignalType enumerations. Each signal listed in the SignalName property is of the type in the corresponding position of SignalType.

Attributes:

GetAccess	public
SetAccess	protected

Timestamp — Timestamps of signals in data source

[] (default) | cell array of duration vectors

Timestamps of the signals that can be loaded from the data source, specified as a cell array of duration vectors. Each signal listed in the SignalName property has the timestamps in the corresponding position of Timestamp.

Attributes:

GetAccess	public
SetAccess	protected

`NumSignals` — Number of signals in data source

`0` (default) | integer

Number of signals that can be read from the data source, specified as a nonnegative integer. `NumSignals` is equal to the number of signals in the `SignalName` property.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`
`NonCopyable`	`true`

Methods

Public Methods

`customizeLoadPanel`	`customizeLoadPanel(sourceObj,panel)` Customize the loading panel for the data source object. In the loading dialog box of the app, this method is invoked when you select the data source type from the **Source Type** list.
`getLoadPanelData`	`[sourceName,sourceParams] = getLoadPanelData(sourceObj)` Obtain the data needed to load the data source object currently selected in the loading panel. In the loading dialog box of the app, this method is invoked when you add a source. The method returns these outputs. • `sourceName` is a string capturing the name of the data source object. • `sourceParams` is a structure with fields containing the parameters required to load the data source object. Both of these outputs are passed to the `loadSource` method.
`loadSource`	`loadSource(sourceObj,sourceName,sourceParams)` Load a data source object into the app. In the loading dialog box of the app, this method is invoked after you add a source and the `getLoadPanelData` method executes successfully. This method is also invoked when you load the data source object into the MATLAB workspace. When you load the data source object, MATLAB expects that the source has the name `sourceName` and parameters `sourceParams` that are needed to load that source and read data from it.

readFrame	`frame = readFrame(sourceObj,signalName,tsIndex)` Read a frame of data from a signal contained in a data source object at the specified timestamp index. The index must be in the bounds of the length of the timestamps for that signal.
loadPanelChecker	`loadPanelChecker` Check the load panel for the loading dialog box of the app. This method opens a dialog box similar to the loading dialog box that you open from the **Open** menu on the app toolstrip. Use this method to preview how the `customizeLoadPanel` method populates the loading panel for the selected data source object. Static: true

Examples

Create Point Cloud Sequence Source

Specify the path to a folder containing a point cloud sequence.

```
pcSeqFolder = fullfile(toolboxdir('vision'),'visiondata', ...
    'pcdmapseq');
```

Create a point cloud sequence source. The sequence does not have a separate timestamps file to load with it, so specify the source parameters as empty. Load the folder path and the empty source parameters into the `PointCloudSequenceSource` object.

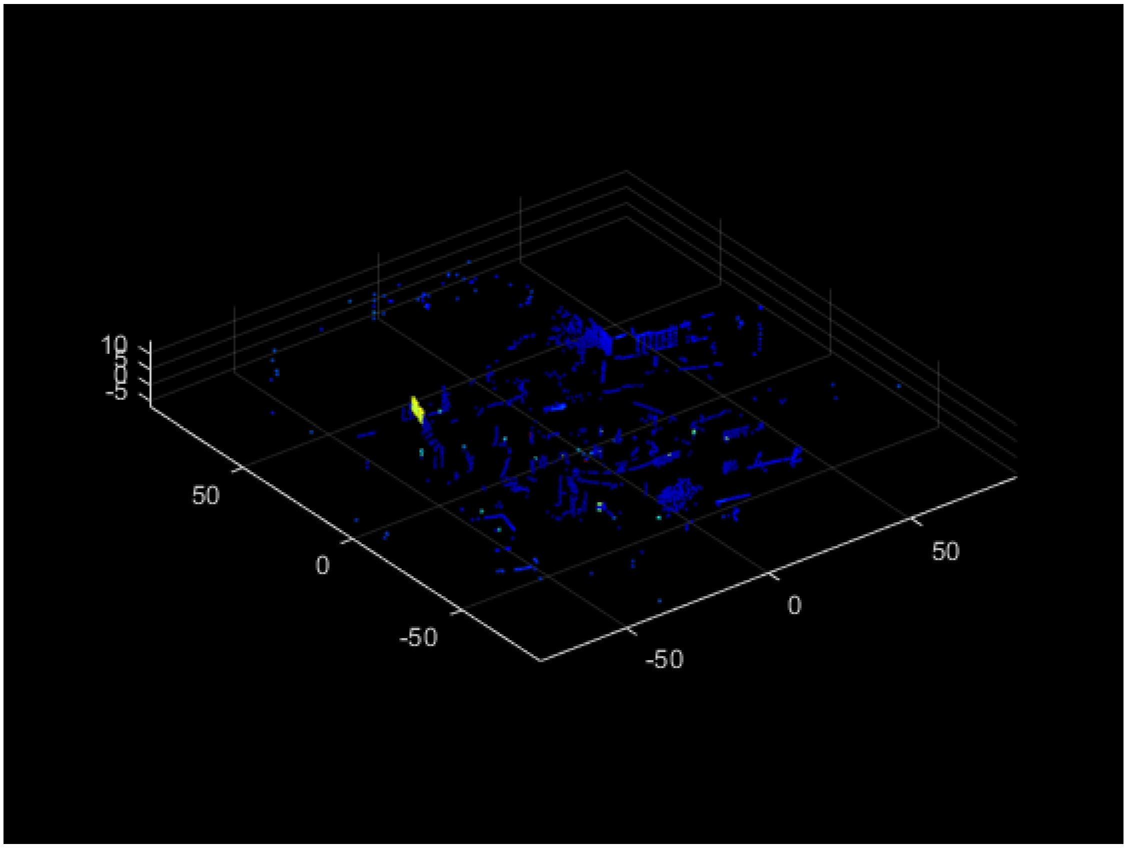
```
sourceName = pcSeqFolder;
sourceParams = [];

pcseqSource = vision.labeler.loading.PointCloudSequenceSource;
loadSource(pcseqSource,sourceName,sourceParams)
```

Read the first frame in the sequence. Display the frame.

```
signalName = pcseqSource.SignalName;
pc = readFrame(pcseqSource,signalName,1);

figure
pcshow(pc)
```



See Also

Apps
Lidar Labeler

Classes
`lidar.labeler.loading.LasFileSequenceSource` | `lidar.labeler.loading.RosbagSource` | `vision.labeler.loading.VelodyneLidarSource`

Introduced in R2020b

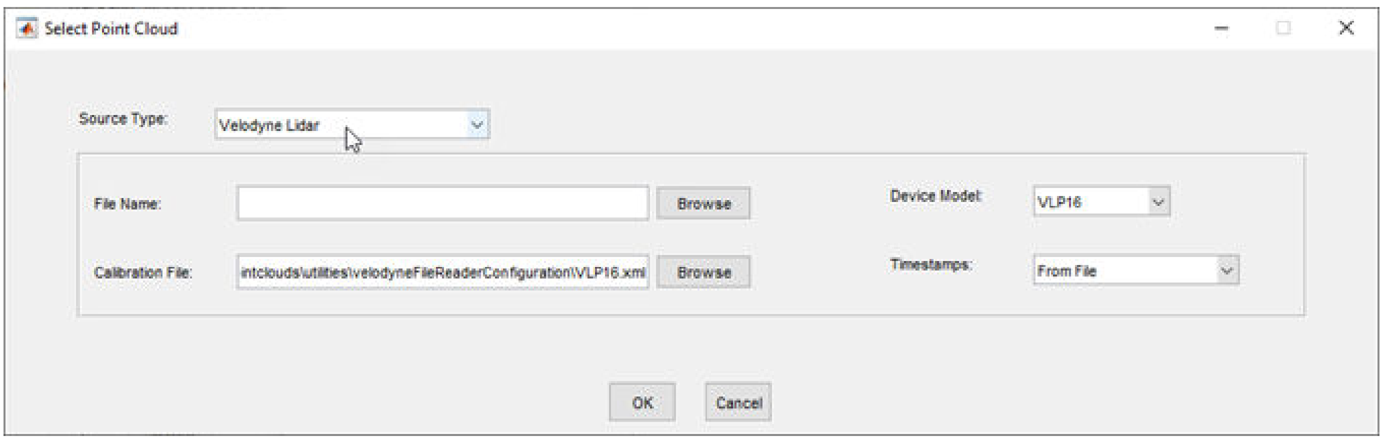
vision.labeler.loading.VelodyneLidarSource class

Package: vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading vision.labeler.loading
Superclasses: vision.labeler.loading.MultiSignalSource

Load signals from Velodyne lidar sources into Lidar Labeler app

Description

The vision.labeler.loading.VelodyneLidarSource class creates an interface for loading a signal from a Velodyne packet capture (PCAP) lidar data source into the **Lidar Labeler** app. In the Select Point Cloud dialog box of the app, when **Source Type** is set to Velodyne Lidar, this class controls the parameters in that dialog box.



To access this dialog box, in the app, select **Open** > **Add Point Cloud**.

This class loads Velodyne PCAP files from the device models accepted by the velodyneFileReader function.

The vision.labeler.loading.VelodyneLidarSource class is a handle class.

Creation

When you export labels from a **Lidar Labeler** app session that contains a Velodyne lidar source, the exported groundTruthLidar object stores an instance of this class in its DataSource property.

To create a VelodyneLidarSource object programmatically, such as when programmatically creating a groundTruthLidar object, use the vision.labeler.loading.VelodyneLidarSource function (described here).

Syntax

```
velodyneSource = vision.labeler.loading.VelodyneLidarSource
```

Description

`velodyneSource = vision.labeler.loading.VelodyneLidarSource` creates a `VelodyneLidarSource` object for loading a signal from a Velodyne lidar data source. To specify the data source and the parameters required to load the source, use the `loadSource` method.

Properties

`Name` — Name of source type
`"Velodyne Lidar"` (default) | string scalar

Name of the type of source that this class loads, specified as a string scalar.

Attributes:

GetAccess	public
Constant	true
NonCopyable	true

`Description` — Description of class functionality
`"A Velodyne file reader"` (default) | string scalar

Description of the functionality that this class provides, specified as a string scalar.

Attributes:

GetAccess	public
Constant	true
NonCopyable	true

`SourceName` — Name of data source
`[]` (default) | string scalar

Name of the data source, specified as a string scalar. Typically, `SourceName` is the name of the file from which the signal is loaded.

Attributes:

GetAccess	public
SetAccess	protected

`SourceParams` — Parameters for loading Velodyne lidar signal from data source
`[]` (default) | structure

Parameters for loading a Velodyne lidar signal from a data source, specified as a structure.

This table describes the required and optional fields of the `SourceParams` structure.

Field	Description	Required or Optional
`Timestamps`	Timestamps for the Velodyne lidar signal, specified as a cell array containing a single `duration` vector of timestamps. In the Select Point Cloud dialog box of the app, if you set the **Timestamps** parameter to `From Workspace` and read the timestamps from a variable in the MATLAB workspace, then the `SourceParams` property stores these timestamps in the `Timestamps` field.	Optional In the Select Point Cloud dialog box of the app, if you set the **Timestamps** parameter to `From File` and read the timestamps from the Velodyne PCAP file, then the structure does not include this field.
`DeviceModel`	Velodyne device model name, specified as one of these options. If you specify the incorrect device model for your Velodyne PCAP file, the app loads an improperly calibrated point cloud. In the Select Point Cloud dialog box of the app, select the device model from the **Device Model** parameter. The **Calibration File** parameter updates to the calibration file of the selected device model.	Required

Field	Description	Required or Optional

`CalibrationFile`	Name of the Velodyne calibration XML file, specified as a character vector or string scalar.
	
	To specify one of the calibration files included with your MATLAB installation, at the MATLAB command prompt, enter this code. Replace `<DeviceModel>` with the name of the device model that you specify in the `DeviceModel` field of this structure (without quotes).
	
	`calibrationFile = fullfile( ...`
	`    matlabroot,'toolbox', ...`
	`    'shared','pointclouds','utilities', ...`
	`    'velodyneFileReaderConfiguration', ...`
	`    '<DeviceModel>.xml')`
	
	By default, the `CalibrationFile` field is set to the full path to the `VLP16.xml` file, which is the calibration file for the VLP-16 device model.
	
	In the Select Point Cloud dialog box of the app, when you change the **Device Model** parameter selection, the **Calibration File** parameter updates to the corresponding calibration file for the selected device model. You can also browse for or enter a path to a different calibration file in the **Calibration File** box.
		Required

For more details on device models and calibration files, see the `velodyneFileReader` object reference page.

Attributes:

`GetAccess`	`public`
`SetAccess`	`protected`

`SignalName` — Names of signals in data source

[ ] (default) | string vector

Names of the signals that can be loaded from the data source, specified as a string vector.

Attributes:

GetAccess	public
SetAccess	protected

SignalType — Types of signals in data source
[] (default) | vector of vision.labeler.loading.SignalType enumerations

Types of the signals that can be loaded from the data source, specified as a vector of vision.labeler.loading.SignalType enumerations. Each signal listed in the SignalName property is of the type in the corresponding position of SignalType.

Attributes:

GetAccess	public
SetAccess	protected

Timestamp — Timestamps of signals in data source
[] (default) | cell array of duration vectors

Timestamps of the signals that can be loaded from the data source, specified as a cell array of duration vectors. Each signal listed in the SignalName property has the timestamps in the corresponding position of Timestamp.

Attributes:

GetAccess	public
SetAccess	protected

NumSignals — Number of signals in data source
0 (default) | integer

Number of signals that can be read from the data source, specified as a nonnegative integer. NumSignals is equal to the number of signals in the SignalName property.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true
NonCopyable	true

Methods

Public Methods

customizeLoadPanel	customizeLoadPanel(sourceObj,panel) Customize the loading panel for the data source object. In the loading dialog box of the app, this method is invoked when you select the data source type from the **Source Type** list.

<table>
<tr><td><code>getLoadPanelData</code></td><td><code>[sourceName,sourceParams] = getLoadPanelData(sourceObj)</code>

Obtain the data needed to load the data source object currently selected in the loading panel. In the loading dialog box of the app, this method is invoked when you add a source. The method returns these outputs.

• <code>sourceName</code> is a string capturing the name of the data source object.
• <code>sourceParams</code> is a structure with fields containing the parameters required to load the data source object.

Both of these outputs are passed to the <code>loadSource</code> method.</td></tr>
<tr><td><code>loadSource</code></td><td><code>loadSource(sourceObj,sourceName,sourceParams)</code>

Load a data source object into the app. In the loading dialog box of the app, this method is invoked after you add a source and the <code>getLoadPanelData</code> method executes successfully. This method is also invoked when you load the data source object into the MATLAB workspace. When you load the data source object, MATLAB expects that the source has the name <code>sourceName</code> and parameters <code>sourceParams</code> that are needed to load that source and read data from it.</td></tr>
<tr><td><code>readFrame</code></td><td><code>frame = readFrame(sourceObj,signalName,tsIndex)</code>

Read a frame of data from a signal contained in a data source object at the specified timestamp index. The index must be in the bounds of the length of the timestamps for that signal.</td></tr>
<tr><td><code>loadPanelChecker</code></td><td><code>loadPanelChecker</code>

Check the load panel for the loading dialog box of the app. This method opens a dialog box similar to the loading dialog box that you open from the Open menu on the app toolstrip. Use this method to preview how the <code>customizeLoadPanel</code> method populates the loading panel for the selected data source object.

<table><tr><td><code>Static</code></td><td><code>true</code></td></tr></table></td></tr>
</table>

Examples

Create Velodyne Lidar Source

Specify the name of the Velodyne® lidar data source, a packet capture (PCAP) file.

```
sourceName = fullfile(toolboxdir('vision'),'visiondata', ...
    'lidarData_ConstructionRoad.pcap');
```

Specify information needed to load the source, including the device model of the lidar and the calibration file.

```
sourceParams = struct;
sourceParams.DeviceModel = 'HDL32E';
sourceParams.CalibrationFile = fullfile(matlabroot,'toolbox','shared', ...
    'pointclouds','utilities','velodyneFileReaderConfiguration', ...
    'HDL32E.xml');
```

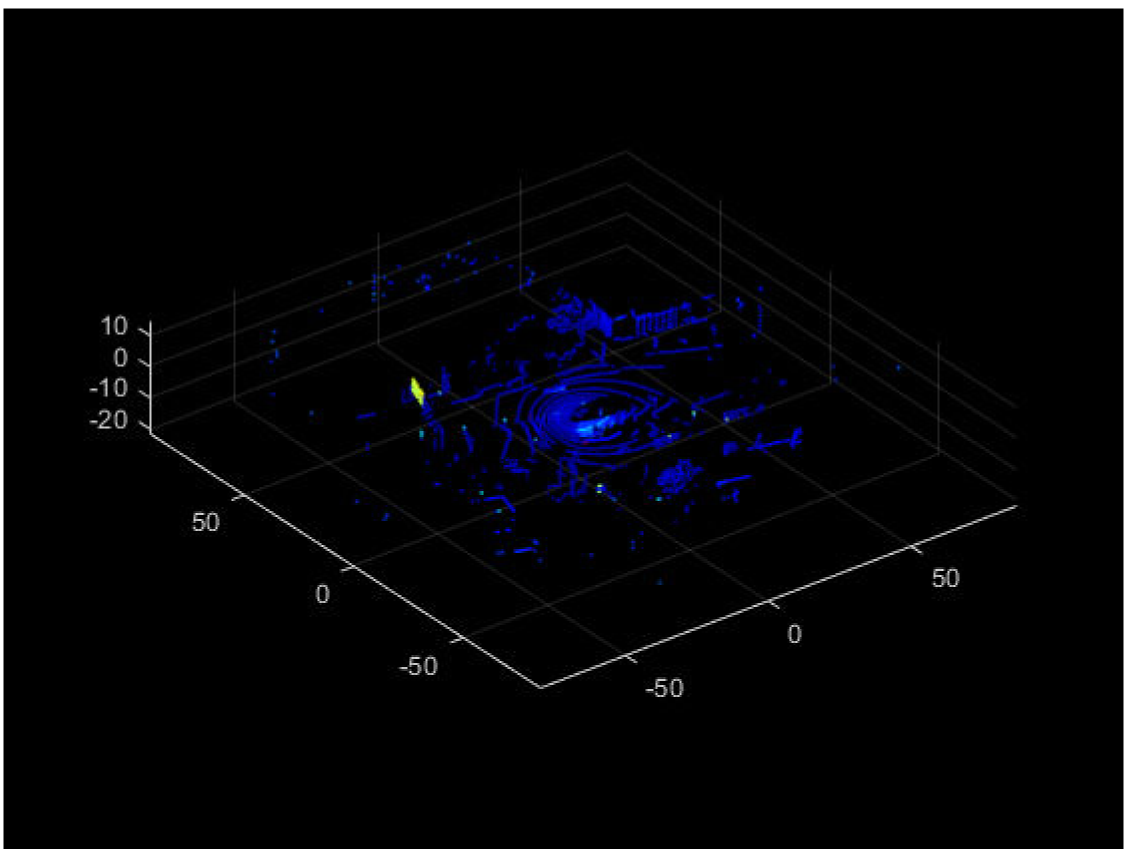
Create the Velodyne lidar data source. Load the data source path, device model, and calibration file path into the `VelodyneLidarSource` object.

```
velodyneSource = vision.labeler.loading.VelodyneLidarSource;
loadSource(velodyneSource,sourceName,sourceParams)
```

Read the first frame from the source. Display the frame.

```
signalName = velodyneSource.SignalName;
pc = readFrame(velodyneSource,signalName,1);

figure
pcshow(pc)
```



See Also

Apps
Lidar Labeler

Classes
`lidar.labeler.loading.LasFileSequenceSource` |
`lidar.labeler.loading.RosbagSource` |
`vision.labeler.loading.PointCloudSequenceSource`

Introduced in R2020b

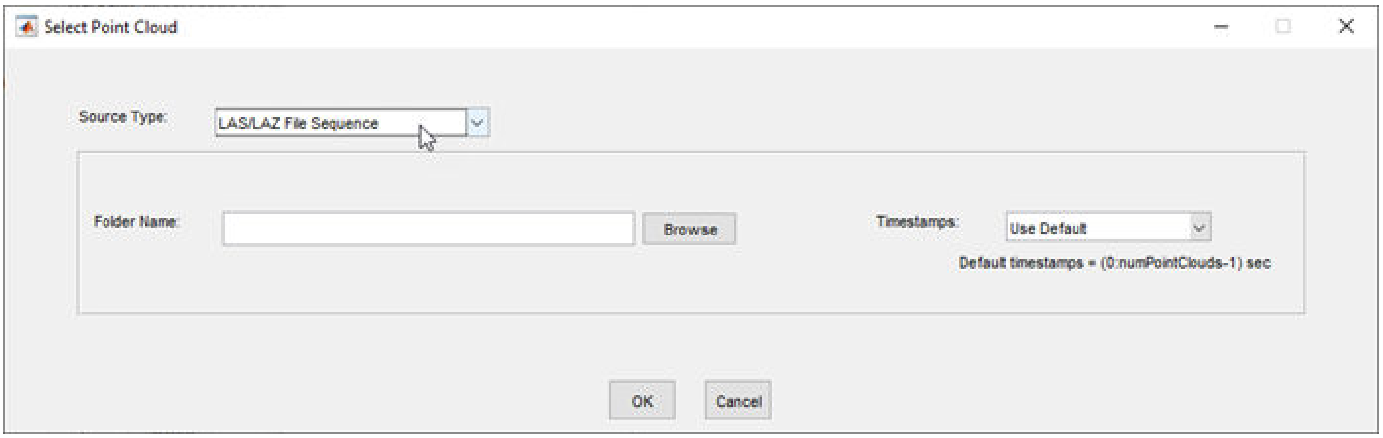
lidar.labeler.loading.LasFileSequenceSource class

Package: `lidar.labeler.loading lidar.labeler.loading lidar.labeler.loading lidar.labeler.loading lidar.labeler.loading lidar.labeler.loading`
Superclasses: `vision.labeler.loading.MultiSignalSource`

Load signals from LAS or LAZ file sequence sources into Lidar Labeler app

Description

The `lidar.labeler.loading.LasFileSequenceSource` class creates an interface for loading a signal from a LAS or LAZ file sequence data source into the **Lidar Labeler** app. In the Select Point Cloud dialog box of the app, when **Source Type** is set to `LAS/LAZ File Sequence`, this class controls the parameters in that dialog box.



To access this dialog box, in the app, select **Open** > **Add Point Cloud**.

The `lidar.labeler.loading.LasFileSequenceSource` class is a `handle` class.

Creation

When you export labels from a **Lidar Labeler** app session that contains a LAS or LAZ file sequence source, the exported `groundTruthLidar` object stores an instance of this class in its `DataSource` property.

To create a `LasFileSequenceSource` object programmatically, such as when programmatically creating a `groundTruthLidar` object, use the `lidar.labeler.loading.LasFileSequenceSource` function (described here).

Syntax

```
lasSeqSource = lidar.labeler.loading.LasFileSequenceSource
```

Description

`lasSeqSource = lidar.labeler.loading.LasFileSequenceSource` creates a `LasFileSequenceSource` object for loading a signal from a LAS or LAZ file sequence data source. To specify the data source and the parameters required to load the source, use the `loadSource` method.

Properties

Name — Name of source type
`"LAS/LAZ File Sequence"` (default) | string scalar

Name of the type of source that this class loads, specified as a string scalar.

Attributes:

GetAccess	public
Constant	true
NonCopyable	true

`Description` — Description of class functionality
`"A LAS/LAZ file sequence reader"` (default) | string scalar

Description of the functionality that this class provides, specified as a string scalar.

Attributes:

GetAccess	public
Constant	true
NonCopyable	true

`SourceName` — Name of data source
`[]` (default) | string scalar

Name of the data source, specified as a string scalar. Typically, `SourceName` is the name of the file from which the signal is loaded.

Attributes:

GetAccess	public
SetAccess	protected

`SourceParams` — Parameters for loading LAS or LAZ file sequence signal from data source
`[]` (default) | structure

Parameters for loading a LAS or LAZ file sequence signal from a data source, specified as a structure.

This table describes the required and optional fields of the `SourceParams` structure.

Field	Description	Required or Optional
`Timestamps`	Timestamps for the LAS or LAZ file sequence signal, specified as a cell array containing a single `duration` vector of timestamps. In the Select Point Cloud dialog box of the app, if you set the **Timestamps** parameter to `From Workspace` and read the timestamps from a variable in the MATLAB workspace, then the `SourceParams` property stores these timestamps in the `Timestamps` field.	Optional If you set the **Timestamps** parameter to `Use Default` and use the default timestamps for LAS or LAZ file sequence signals, then the structure does not include this field, and the `SourceParams` property is empty, `[]`. For LAS or LAZ file sequence signals, the default timestamp `duration` vector has elements from 0 to the number of valid LAS or LAZ files minus 1. Units are in seconds.

Attributes:

GetAccess	public
SetAccess	protected

`SignalName` — Names of signals in data source

`[]` (default) | string vector

Names of the signals that can be loaded from the data source, specified as a string vector.

Attributes:

GetAccess	public
SetAccess	protected

`SignalType` — Types of signals in data source

`[]` (default) | vector of `vision.labeler.loading.SignalType` enumerations

Types of the signals that can be loaded from the data source, specified as a vector of `vision.labeler.loading.SignalType` enumerations. Each signal listed in the `SignalName` property is of the type in the corresponding position of `SignalType`.

Attributes:

GetAccess	public
SetAccess	protected

`Timestamp` — Timestamps of signals in data source

`[]` (default) | cell array of `duration` vectors

Timestamps of the signals that can be loaded from the data source, specified as a cell array of `duration` vectors. Each signal listed in the `SignalName` property has the timestamps in the corresponding position of `Timestamp`.

Attributes:

GetAccess	public
SetAccess	protected

`NumSignals` — Number of signals in data source
`0` (default) | integer

Number of signals that can be read from the data source, specified as a nonnegative integer. `NumSignals` is equal to the number of signals in the `SignalName` property.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`
`NonCopyable`	`true`

Methods

Public Methods

`customizeLoadPanel`	`customizeLoadPanel(sourceObj,panel)` Customize the loading panel for the data source object. In the loading dialog box of the app, this method is invoked when you select the data source type from the **Source Type** list.
`getLoadPanelData`	`[sourceName,sourceParams] = getLoadPanelData(sourceObj)` Obtain the data needed to load the data source object currently selected in the loading panel. In the loading dialog box of the app, this method is invoked when you add a source. The method returns these outputs. • `sourceName` is a string capturing the name of the data source object. • `sourceParams` is a structure with fields containing the parameters required to load the data source object. Both of these outputs are passed to the `loadSource` method.
`loadSource`	`loadSource(sourceObj,sourceName,sourceParams)` Load a data source object into the app. In the loading dialog box of the app, this method is invoked after you add a source and the `getLoadPanelData` method executes successfully. This method is also invoked when you load the data source object into the MATLAB workspace. When you load the data source object, MATLAB expects that the source has the name `sourceName` and parameters `sourceParams` that are needed to load that source and read data from it.

`readFrame`	`frame = readFrame(sourceObj,signalName,tsIndex)` Read a frame of data from a signal contained in a data source object at the specified timestamp index. The index must be in the bounds of the length of the timestamps for that signal.
`loadPanelChecker`	`loadPanelChecker` Check the load panel for the loading dialog box of the app. This method opens a dialog box similar to the loading dialog box that you open from the **Open** menu on the app toolstrip. Use this method to preview how the `customizeLoadPanel` method populates the loading panel for the selected data source object. Static: true

Examples

Create LAS File Sequence Source

Specify the path to a folder containing a LAS file sequence.

```
lasSeqFolder = fullfile(toolboxdir('lidar'),'lidardata','las');
```

The LAS file consists of two point cloud frames that occur at one-second intervals. Specify the timestamps of the frames as a `duration` vector of two seconds.

```
timestamps = seconds(1:2);
```

Create a LAS file sequence source. Load the folder path and timestamps into the `LasFileSequenceSource` object.

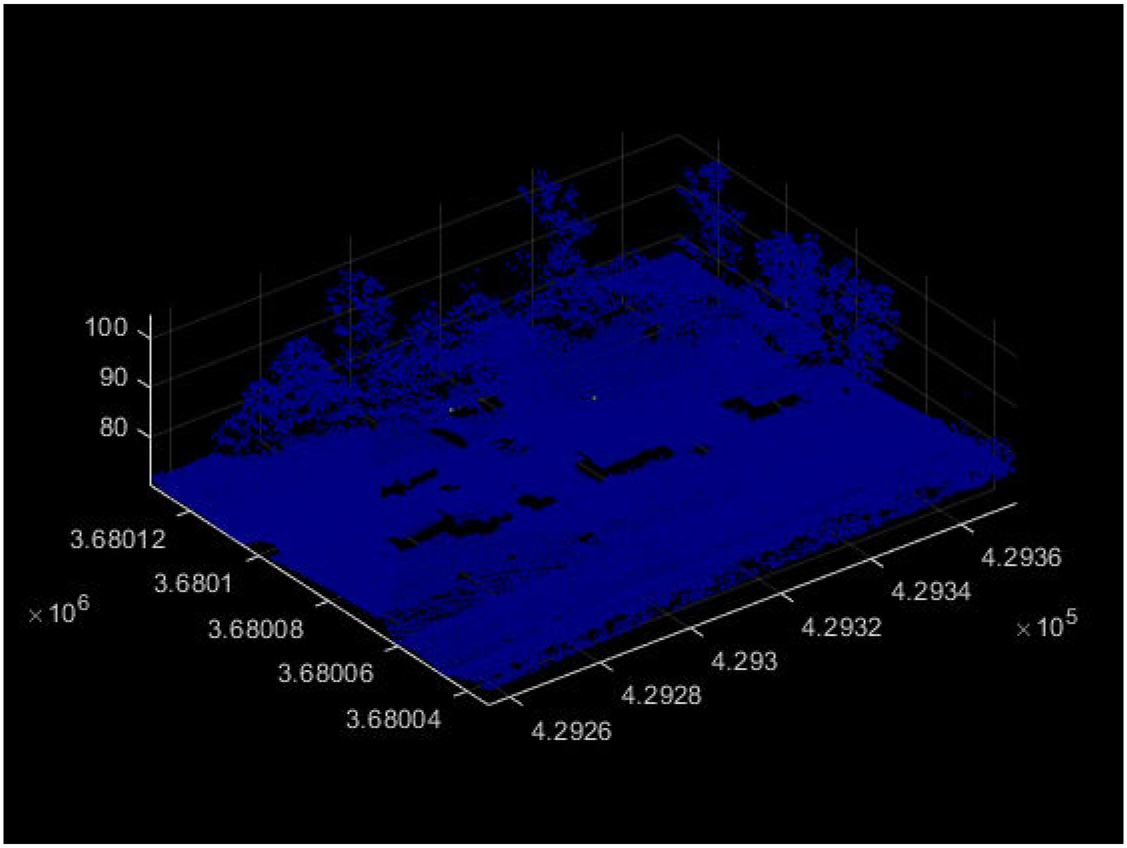
```
sourceName = lasSeqFolder;
sourceParams = struct;
sourceParams.Timestamps = timestamps;

lasSeqSource = lidar.labeler.loading.LasFileSequenceSource;
loadSource(lasSeqSource,sourceName,sourceParams)
```

Read the second frame in the sequence. Display the frame.

```
signalName = lasSeqSource.SignalName;
pc = readFrame(lasSeqSource,signalName,2);

figure
pcshow(pc)
```



See Also

Apps
Lidar Labeler

Classes
`lidar.labeler.loading.RosbagSource` | `vision.labeler.loading.PointCloudSequenceSource` | `vision.labeler.loading.VelodyneLidarSource`

Introduced in R2020b

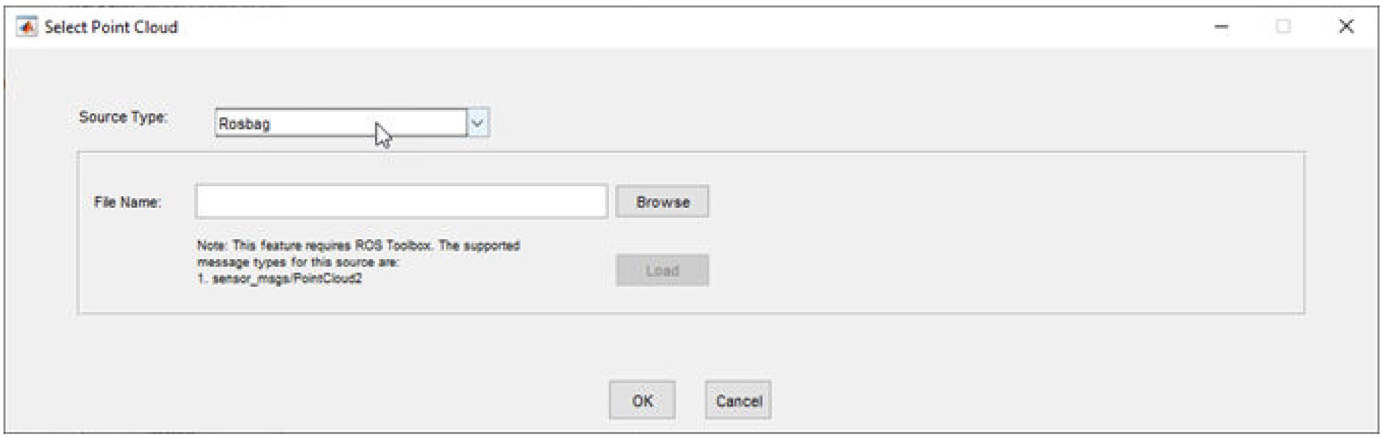
lidar.labeler.loading.RosbagSource class

Package: `lidar.labeler.loading lidar.labeler.loading lidar.labeler.loading lidar.labeler.loading lidar.labeler.loading lidar.labeler.loading`
Superclasses: `vision.labeler.loading.MultiSignalSource`

Load signals from rosbag sources into Lidar Labeler app

Description

The `lidar.labeler.loading.RosbagSource` class creates an interface for loading a signal from a rosbag file into the **Lidar Labeler** app. In the Select Point Cloud dialog box of the app, when **Source Type** is set to `Rosbag`, this class controls the parameters in that dialog box.



To access this dialog box, in the app, select **Open** > **Add Point Cloud**.

This class loads signals from the `sensor_msgs/PointCloud2` ROS message type only.

Note This class requires ROS Toolbox.

The `lidar.labeler.loading.RosbagSource` class is a `handle` class.

Creation

When you export labels from a **Lidar Labeler** app session that contains a rosbag source, the exported `groundTruthLidar` object stores an instance of this class in its `DataSource` property.

To create a `RosbagSource` object programmatically, such as when programmatically creating a `groundTruthLidar` object, use the `lidar.labeler.loading.RosbagSource` function (described here).

Syntax

```
rosbagSource = lidar.labeler.loading.RosbagSource
```

Description

`rosbagSource = lidar.labeler.loading.RosbagSource` creates a `RosbagSource` object for loading a signal from a rosbag data source. To specify the data source and the parameters required to load the source, use the `loadSource` method.

Properties

`Name` — Name of source type
`"Rosbag"` (default) | string scalar

Name of the type of source that this class loads, specified as a string scalar.

Attributes:

`GetAccess`	`public`
`Constant`	`true`
`NonCopyable`	`true`

`Description` — Description of class functionality
`"A rosbag reader"` (default) | string scalar

Description of the functionality that this class provides, specified as a string scalar.

Attributes:

`GetAccess`	`public`
`Constant`	`true`
`NonCopyable`	`true`

`SourceName` — Name of data source
[ ] (default) | string scalar

Name of the data source, specified as a string scalar. Typically, `SourceName` is the name of the file from which the signal is loaded.

Attributes:

`GetAccess`	`public`
`SetAccess`	`protected`

`SourceParams` — Parameters for loading signals from rosbag data source
[ ] (default) | empty structure

Parameters for loading signals from a rosbag data source, specified as an empty structure. When you load a point cloud signal from a rosbag, do not specify the signal timestamps or any other parameters. The `loadSource` method reads these parameters from the rosbag.

Attributes:

`GetAccess`	`public`
`SetAccess`	`protected`

`SignalName` — Names of signals in data source
[ ] (default) | string vector

Names of the signals that can be loaded from the data source, specified as a string vector.

Attributes:

GetAccess	public
SetAccess	protected

SignalType — Types of signals in data source
[] (default) | vector of vision.labeler.loading.SignalType enumerations

Types of the signals that can be loaded from the data source, specified as a vector of vision.labeler.loading.SignalType enumerations. Each signal listed in the SignalName property is of the type in the corresponding position of SignalType.

Attributes:

GetAccess	public
SetAccess	protected

Timestamp — Timestamps of signals in data source
[] (default) | cell array of duration vectors

Timestamps of the signals that can be loaded from the data source, specified as a cell array of duration vectors. Each signal listed in the SignalName property has the timestamps in the corresponding position of Timestamp.

Attributes:

GetAccess	public
SetAccess	protected

NumSignals — Number of signals in data source
0 (default) | integer

Number of signals that can be read from the data source, specified as a nonnegative integer. NumSignals is equal to the number of signals in the SignalName property.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true
NonCopyable	true

Methods

Public Methods

customizeLoadPanel	customizeLoadPanel(sourceObj,panel) Customize the loading panel for the data source object. In the loading dialog box of the app, this method is invoked when you select the data source type from the **Source Type** list.

<table>
<tr><td><code>getLoadPanelData</code></td><td><code>[sourceName,sourceParams] = getLoadPanelData(sourceObj)</code>

Obtain the data needed to load the data source object currently selected in the loading panel. In the loading dialog box of the app, this method is invoked when you add a source. The method returns these outputs.

• <code>sourceName</code> is a string capturing the name of the data source object.
• <code>sourceParams</code> is a structure with fields containing the parameters required to load the data source object.

Both of these outputs are passed to the <code>loadSource</code> method.</td></tr>
<tr><td><code>loadSource</code></td><td><code>loadSource(sourceObj,sourceName,sourceParams)</code>

Load a data source object into the app. In the loading dialog box of the app, this method is invoked after you add a source and the <code>getLoadPanelData</code> method executes successfully. This method is also invoked when you load the data source object into the MATLAB workspace. When you load the data source object, MATLAB expects that the source has the name <code>sourceName</code> and parameters <code>sourceParams</code> that are needed to load that source and read data from it.</td></tr>
<tr><td><code>readFrame</code></td><td><code>frame = readFrame(sourceObj,signalName,tsIndex)</code>

Read a frame of data from a signal contained in a data source object at the specified timestamp index. The index must be in the bounds of the length of the timestamps for that signal.</td></tr>
<tr><td><code>loadPanelChecker</code></td><td><code>loadPanelChecker</code>

Check the load panel for the loading dialog box of the app. This method opens a dialog box similar to the loading dialog box that you open from the Open menu on the app toolstrip. Use this method to preview how the <code>customizeLoadPanel</code> method populates the loading panel for the selected data source object.

<table><tr><td><code>Static</code></td><td><code>true</code></td></tr></table></td></tr>
</table>

See Also

Apps
Lidar Labeler

Classes
`lidar.labeler.loading.LasFileSequenceSource` | `vision.labeler.loading.PointCloudSequenceSource` | `vision.labeler.loading.VelodyneLidarSource`

Introduced in R2020b

lidar.syncImageViewer.SyncImageViewer class

Package: `lidar.syncImageViewer`

Interface to connect external tool to Lidar Labeler app

Description

The `lidar.syncImageViewer.SyncImageViewer` class creates an interface between a custom visualization or analysis tool and a point cloud signal in the **Lidar Labeler** app. You can use the `SyncImageViewer` class to sync video and image sequence signals to the app only.

Creation

The `SyncImageViewer` specifies the interface for connecting an external tool to the **Lidar Labeler** app. An external tool can be a custom visualization tool or custom analysis tool. The class that inherits from the `SyncImageViewer` interface is called the client. The client performs these tasks:

- Syncs an external tool to each frame change event for a specific signal loaded into the **Lidar Labeler** app. Syncing enables you to control the external tool through the range slider and playback controls of the app.
- Controls the current time in the external tool and the corresponding display in the app.

To connect an external tool to the **Lidar Labeler** app, follow these steps:

1. Define a client class that inherits from `lidar.syncImageViewer.SyncImageViewer`. You can use the `SyncImageViewer` class template to define a class and implement your custom visualization or analysis tool. At the MATLAB command prompt, enter this code:

   ```
   lidar.syncImageViewer.SyncImageViewer.openTemplateInEditor
   ```

   Follow the steps in the template.
2. Save the file to any folder on the MATLAB path. Alternatively, save the file to a folder outside the MATLAB path and add the folder to MATLAB path by using the `addpath` function.

Properties

`VideoStartTime` — Start time of signal
real scalar in seconds

Start time of the signal, specified as a real scalar in seconds.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

`VideoEndTime` — End time of signal
real scalar in seconds

End time of the signal, specified as a real scalar in seconds.

Attributes:

GetAccess	public
SetAccess	private

StartTime — Start of time interval in app

real scalar in seconds

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

Attributes:

GetAccess	public
SetAccess	private

CurrentTime — Time of frame currently displaying in app

real scalar in seconds

Time of the frame currently displaying in the app for the connected signal, specified as a real scalar in seconds. If the slider is between two timestamps, then the currently displaying frame is the frame that is at the previous timestamp.

Attributes:

GetAccess	public
SetAccess	private

EndTime — End of time interval in app

real scalar in seconds

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

Attributes:

GetAccess	public
SetAccess	private

TimeVector — Timestamps for connected signal

`duration` vector

Timestamps for the connected signal, specified as a `duration` vector. This signal must be the master signal. If you change the master signal, the `TimeVector` property updates to the timestamps for new master signal.

Attributes:

GetAccess	public
SetAccess	private

Methods

Public Methods

dataSourceChangeListener	Update external tool when connecting to signal being loaded into Lidar Labeler app

frameChangeListener	Update external tool when new frame is displayed in Lidar Labeler app
updateLabelerCurrentTime	Update current time in Lidar Labeler app
close	Close external tool connected to Lidar Labeler app
disconnect	Disconnect external tool from Lidar Labeler app

Examples

Connect Image Display to Lidar Labeler

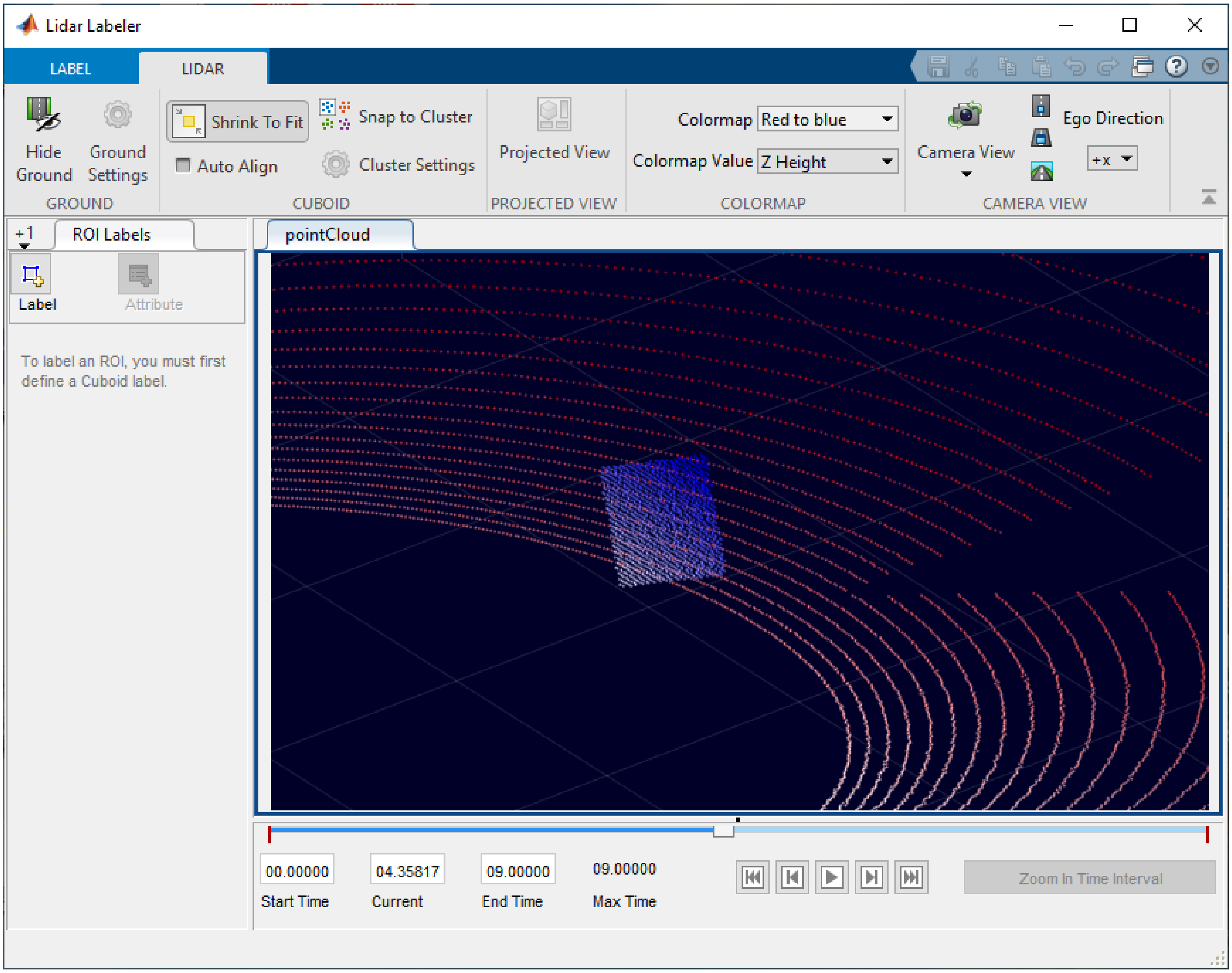
Connect an image display tool to the **Lidar Labeler** app. Use the app and tool to display synchronized lidar and image data.

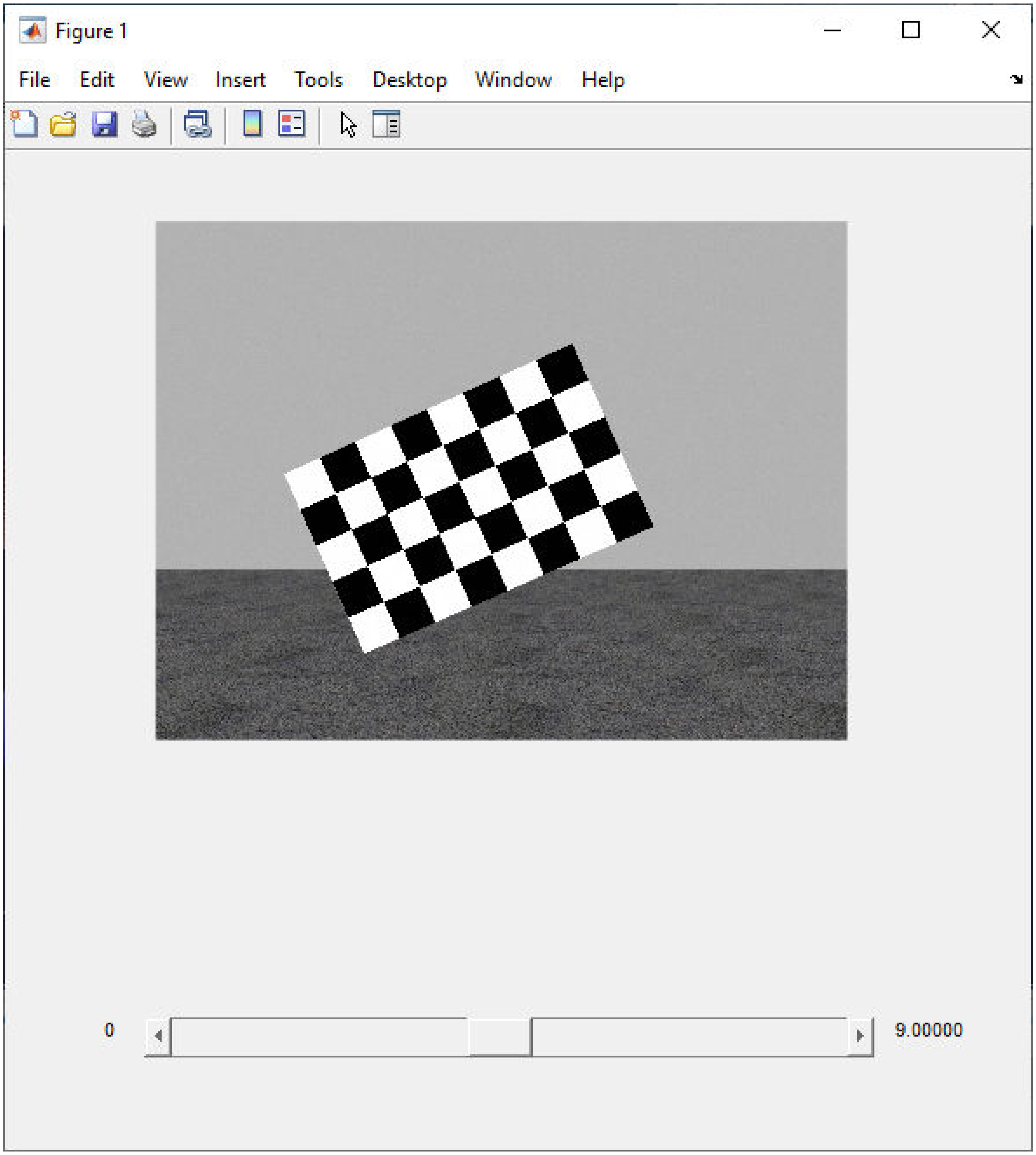
Specify the source for the image data and lidar data to load into the app.

```
sourceName = fullfile(toolboxdir('lidar'),'lidardata','lcc', ...
                    'HDL64','pointCloud');
```

Display the image data in sync with the lidar data.

```
lidarLabeler(sourceName,'SyncImageViewerTargetHandle',@SyncImageDisplay);
```


Lidar Labeler
LABEL
LIDAR
Hide Ground
Ground Settings
GROUND
Shrink To Fit
Snap to Cluster
Auto Align
Cluster Settings
CUBOID
Projected View
PROJECTED VIEW
Colormap Red to blue
Colormap Value Z Height
COLORMAP
Camera View
Ego Direction
+x
CAMERA VIEW
ROI Labels
Label
Attribute
To label an ROI, you must first define a Cuboid label.
pointCloud
00.00000
Start Time
04.35817
Current
09.00000
End Time
09.00000
Max Time
Zoom In Time Interval



Tips

- For an example of an external tool, see the `SyncImageDisplay` implementation of the `lidar.syncImageViewer.SyncImageViewer` class. This class implements an image display tool. You can use this code as a starting point for creating your own tools.

  ```
  edit SyncImageDisplay
  ```

See Also

Apps
Lidar Labeler

Introduced in R2020b

close

Class: `lidar.syncImageViewer.SyncImageViewer`
Package: `lidar.syncImageViewer`

Close external tool connected to Lidar Labeler app

Syntax

`close(syncImageObj)`

Description

`close(syncImageObj)` provides the option to close the external tool that is connected to the **Lidar Labeler** app when the app closes. The app calls this method using the `syncImageObj` object.

Note The client class can optionally implement this method.

Input Arguments

`syncImageObj` — Synced image viewer
`SyncImageViewer` object

Synced image viewer, specified as a `lidar.syncImageViewer.SyncImageViewer` object.

See Also

Lidar Labeler | `lidar.syncImageViewer.SyncImageViewer`

Introduced in R2020b

dataSourceChangeListener

Class: `lidar.syncImageViewer.SyncImageViewer`
Package: `lidar.syncImageViewer`

Update external tool when connecting to signal being loaded into Lidar Labeler app

Syntax

`dataSourceChangeListener(syncImageObj)`

Description

`dataSourceChangeListener(syncImageObj)` provides the option to update the external tool when loading a new data source is loaded into the **Lidar Labeler** app. The app calls this method using the `syncImageObj` object.

Note The client class can optionally implement this method.

Input Arguments

`syncImageObj` — Synced image viewer
`SyncImageViewer` object

Synced image viewer, specified as a `lidar.syncImageViewer.SyncImageViewer` object.

See Also

Lidar Labeler | `lidar.syncImageViewer.SyncImageViewer`

Introduced in R2020b

disconnect

Class: `lidar.syncImageViewer.SyncImageViewer`
Package: `lidar.syncImageViewer`

Disconnect external tool from Lidar Labeler app

Syntax

`disconnect(syncImageObj)`

Description

`disconnect(syncImageObj)` disconnects the interface between an external tool and the **Lidar Labeler** app. The client calls this method using the `syncImageObj` object. After the external tool is disconnected, the **Lidar Labeler** app no longer calls the `frameChangeListener` method in the client class.

Note The client class can call this method.

Input Arguments

`syncImageObj` — Synced image viewer
`SyncImageViewer` object

Synced image viewer, specified as a `lidar.syncImageViewer.SyncImageViewer` object.

See Also

Lidar Labeler | `lidar.syncImageViewer.SyncImageViewer`

Introduced in R2020b

frameChangeListener

Class: `lidar.syncImageViewer.SyncImageViewer`
Package: `lidar.syncImageViewer`

Update external tool when new frame is displayed in Lidar Labeler app

Syntax

`frameChangeListener(syncImageObj)`

Description

`frameChangeListener(syncImageObj)` provides an option to synchronize an external tool with the frame changes in the **Lidar Labeler** app. The app calls this method when a new frame is displayed in the app. If the slider is between two timestamps, then the app displays the frame that is at the previous timestamp.

Note The client class must implement this method.

Input Arguments

`syncImageObj` — Synced image viewer
`SyncImageViewer` object

Synced image viewer, specified as a `lidar.syncImageViewer.SyncImageViewer` object.

See Also

Lidar Labeler | `lidar.syncImageViewer.SyncImageViewer`

Introduced in R2020b

updateLabelerCurrentTime

Class: `lidar.syncImageViewer.SyncImageViewer`
Package: `lidar.syncImageViewer`

Update current time in Lidar Labeler app

Syntax

`updateLabelerCurrentTime(syncImageObj,newTime)`

Description

`updateLabelerCurrentTime(syncImageObj,newTime)` updates the current time in the **Lidar Labeler** app to `newTime`. The client calls this method using the `syncImageObj` object.

Note The client class can call this method.

Input Arguments

`syncImageObj` — Synced image viewer
`SyncImageViewer` object

Synced image viewer, specified as a `lidar.syncImageViewer.SyncImageViewer` object.

`newTime` — Current time for app
real scalar in seconds

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

See Also

Lidar Labeler | `lidar.syncImageViewer.SyncImageViewer`

Introduced in R2020b

lasFileReader

LAS or LAZ file reader

Description

The LAS file format is an industry-standard binary format for storing lidar data, developed and maintained by the American Society for Photogrammetry and Remote Sensing (ASPRS). The LAZ file format is a compressed version of the LAS file format.

A LAS file contains a public header, which has lidar metadata, followed by lidar point records. Each point record contains attributes such as 3-D coordinates, intensity and GPS timestamp.

A `lasFileReader` object stores the metadata present in the LAS or LAZ file as read-only properties. The object function, `readPointCloud`, uses these properties to read point cloud data from the file.

Creation

Syntax

`lasReader = lasFileReader(fileName)`

Description

`lasReader = lasFileReader(fileName)` reads the metadata from a LAS or LAZ file, `fileName`, and stores it as properties of an output `lasFileReader` object, `lasReader`. The `fileName` input sets the `FileName` property.

Properties

`FileName` — Name of LAS or LAZ file
character vector | string scalar

This property is read-only.

Name of the LAS or LAZ file, specified as a character vector or string scalar.

`Count` — Number of available point records
positive integer

This property is read-only.

Number of available point records in the file, specified as a positive integer.

`LasVersion` — LAS or LAZ file version
character vector

This property is read-only.

LAS or LAZ file version, specified as a character vector.

`XLimits` — Range of coordinates along *x*-axis
two-element row vector

This property is read-only.

Range of coordinates along the *x*-axis, specified as a two-element row vector.

`YLimits` — Range of coordinates along *y*-axis
two-element row vector

This property is read-only.

Range of coordinates along the *y*-axis, specified as a two-element row vector.

`ZLimits` — Range of coordinates along *z*-axis
two-element row vector

This property is read-only.

Range of coordinates along the *z*-axis, specified as a two-element row vector.

`GPSTimeLimits` — Range of GPS timestamps
1-by-2 `duration` vector

This property is read-only.

Range of GPS timestamp readings, specified as a 1-by-2 `duration` vector.

`NumReturns` — Maximum of all point laser returns
`1` (default) | positive integer

This property is read-only.

Maximum of all point laser returns, specified as a positive integer.

`NumClasses` — Maximum of all point classification values
`1` (default) | positive integer

This property is read-only.

Maximum of all point classification values, specified as a positive integer.

Object Functions

readPointCloud Read point cloud data from a LAS or LAZ file

Examples

Read Point Cloud Data from LAZ File

Create a `lasFileReader` object for a LAZ file. Then, use the `readPointCloud` function to read point cloud data from the LAZ file and generate a `pointCloud` object.

Create a `lasFileReader` object to access the LAZ file data.

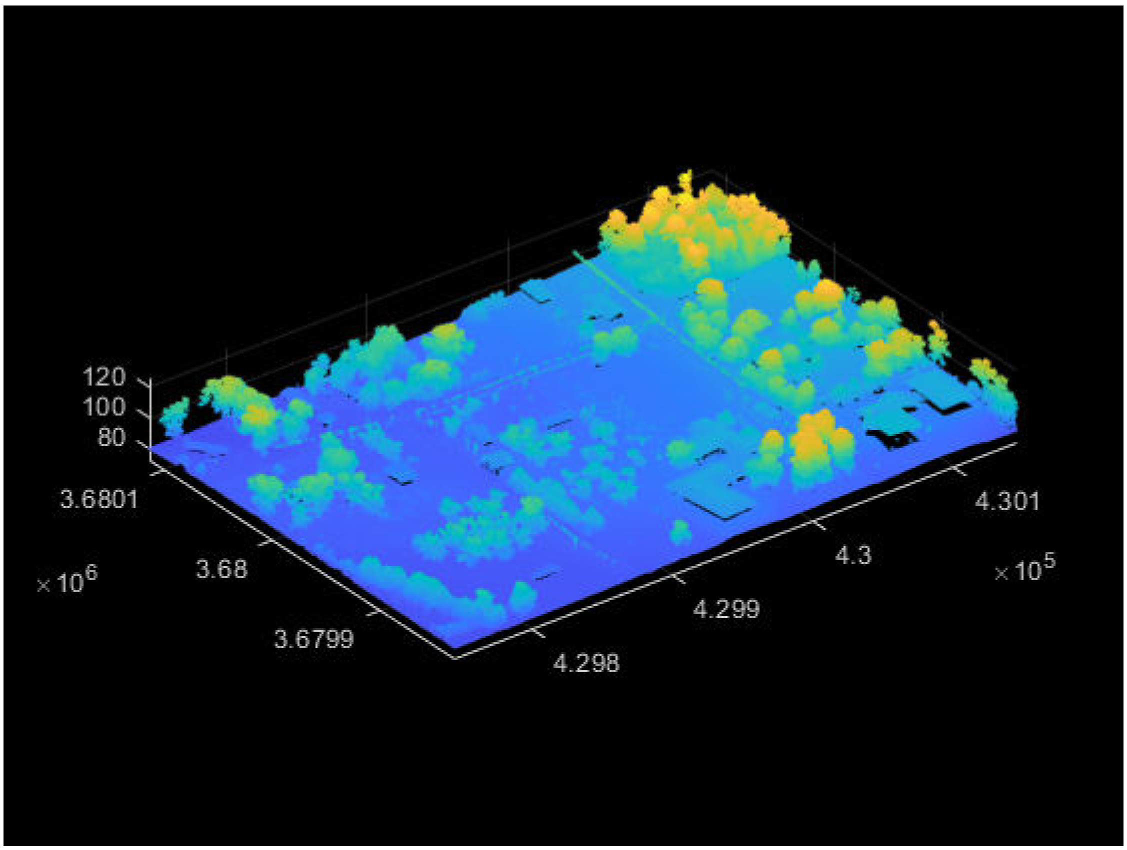
```
path = fullfile(toolboxdir('lidar'),'lidardata', ...
    'las','aerialLidarData.laz');
lasReader = lasFileReader(path);
```

Read point cloud data from the LAZ file using the `readPointCloud` function.

```
ptCloud = readPointCloud(lasReader);
```

Visualize the point cloud.

```
figure
pcshow(ptCloud.Location)
```



Visualize Point Cloud Based on Classification Data from LAZ File

Segregate and visualize point cloud data based on classification data from a LAZ file.

Create a `lasFileReader` object to access data from the LAZ file.

```
path = fullfile(toolboxdir('lidar'),'lidardata', ...
    'las','aerialLidarData.laz');
lasReader = lasFileReader(path);
```

Read point cloud data and associated classification point attributes from the LAZ file using the `readPointCloud` function.

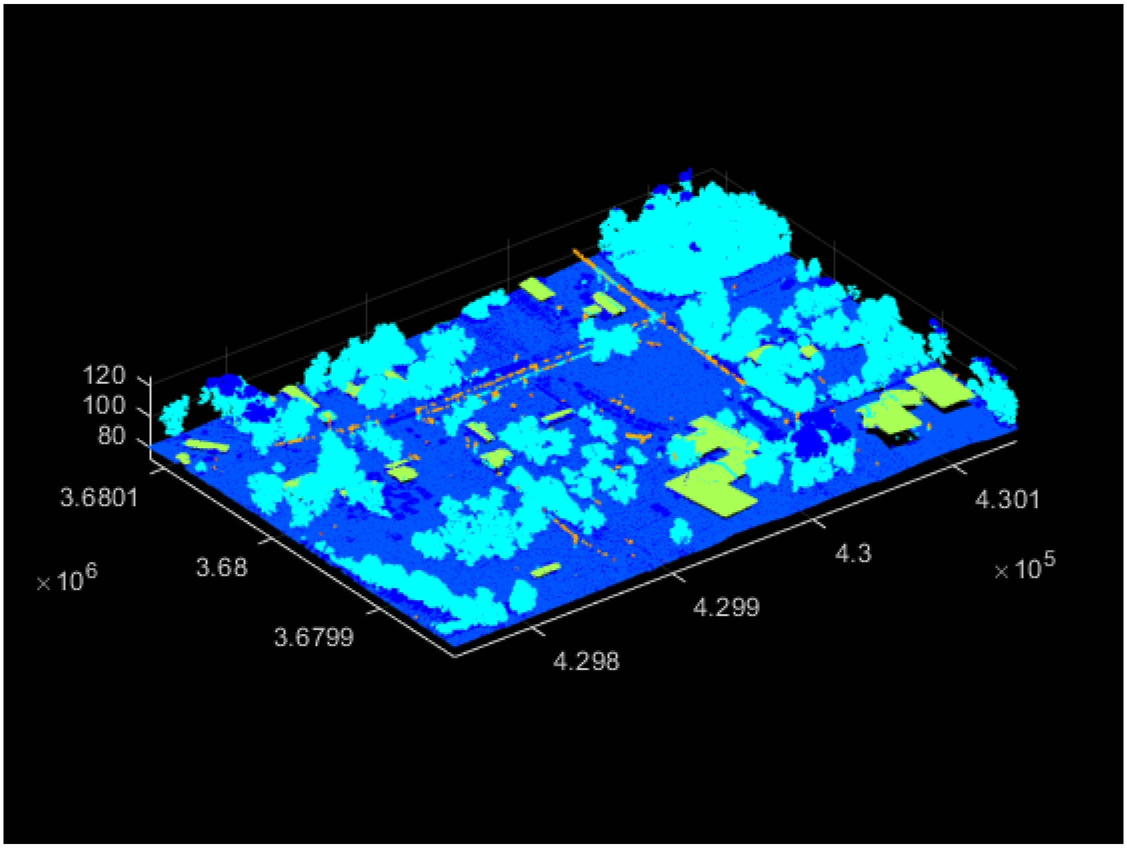
```
[ptCloud,pointAttributes] = readPointCloud(lasReader,'Attributes','Classification');
```

Color the points based on their classification attributes.

```
colorData = reshape(label2rgb(pointAttributes.Classification),[],3);
```

Visualize the color-coded point cloud.

```
figure
pcshow(ptCloud.Location,colorData)
```



See Also

Functions
`pcread` | `pcshow` | `readPointCloud`

Objects
`ibeoLidarReader` | `pointCloud` | `velodyneFileReader`

Introduced in R2020b

readPointCloud

Read point cloud data from a LAS or LAZ file

Syntax

```
ptCloud = readPointCloud(lasReader)
[ptCloud,ptAttributes] = readPointCloud(lasReader,'Attributes',ptAtt)
[ ___ ] = readPointCloud( ___ ,Name,Value)
```

Description

`ptCloud = readPointCloud(lasReader)` reads the point cloud data from the LAS or LAZ file indicated by the input `lasFileReader` object and returns it as a `pointCloud` object, `ptCloud`.

`[ptCloud,ptAttributes] = readPointCloud(lasReader,'Attributes',ptAtt)` reads specified point attributes, `ptAtt`, from a LAS or LAZ file. In addition to the point cloud, the function returns a structure, `ptAttributes`, containing the specified attributes of each point in the point cloud.

`[ ___ ] = readPointCloud( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to any of the argument combinations in previous syntaxes. For example, `'ROI',[5 10 5 10 5 10]` sets the region of interest (ROI) in which the function reads the point cloud.

Examples

Read Point Cloud Data from LAZ File

Create a `lasFileReader` object for a LAZ file. Then, use the `readPointCloud` function to read point cloud data from the LAZ file and generate a `pointCloud` object.

Create a `lasFileReader` object to access the LAZ file data.

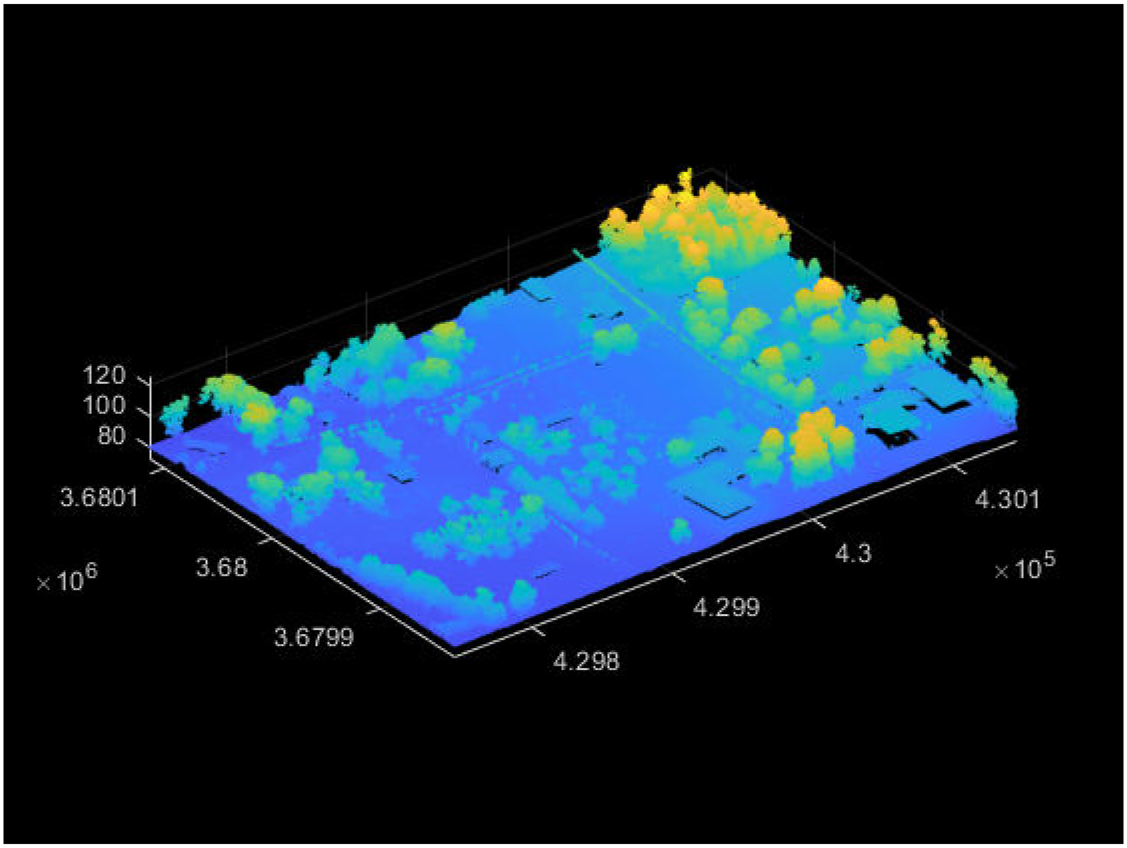
```
path = fullfile(toolboxdir('lidar'),'lidardata', ...
    'las','aerialLidarData.laz');
lasReader = lasFileReader(path);
```

Read point cloud data from the LAZ file using the `readPointCloud` function.

```
ptCloud = readPointCloud(lasReader);
```

Visualize the point cloud.

```
figure
pcshow(ptCloud.Location)
```



Visualize Point Cloud Based on Classification Data from LAZ File

Segregate and visualize point cloud data based on classification data from a LAZ file.

Create a `lasFileReader` object to access data from the LAZ file.

```
path = fullfile(toolboxdir('lidar'),'lidardata', ...
    'las','aerialLidarData.laz');
lasReader = lasFileReader(path);
```

Read point cloud data and associated classification point attributes from the LAZ file using the `readPointCloud` function.

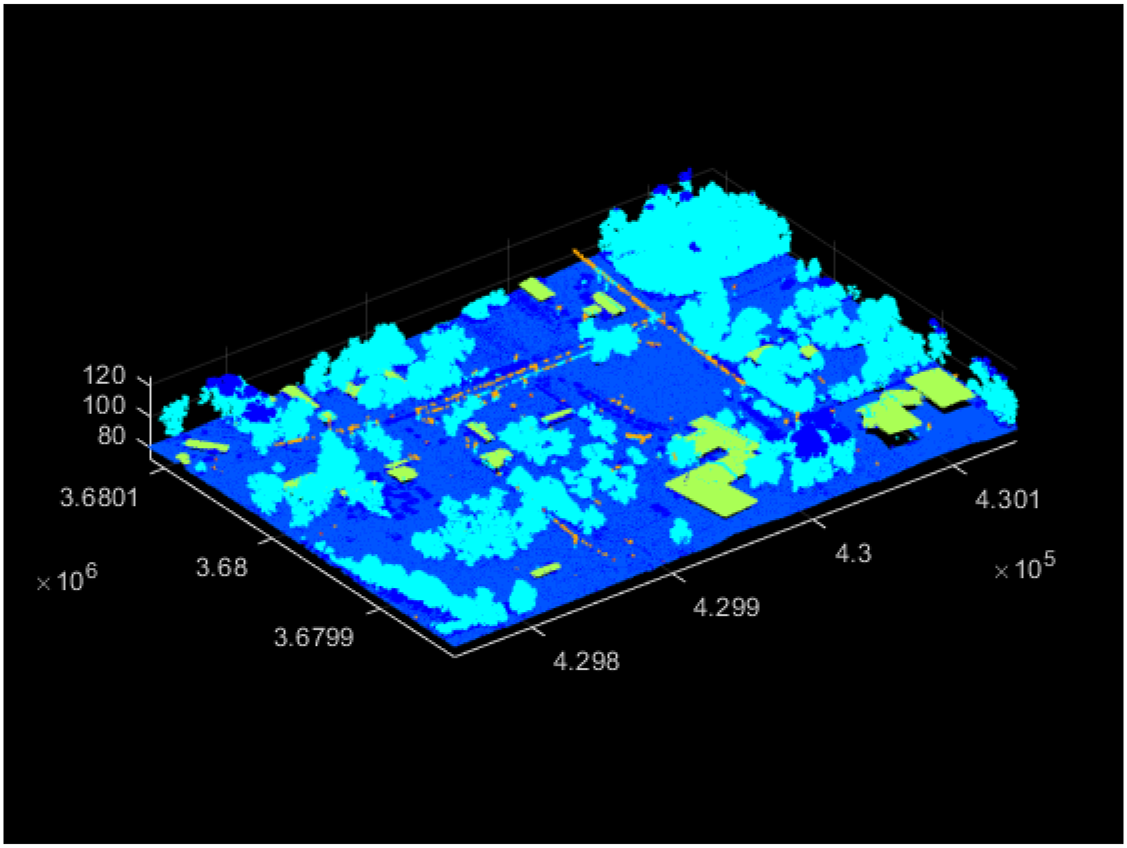
```
[ptCloud,pointAttributes] = readPointCloud(lasReader,'Attributes','Classification');
```

Color the points based on their classification attributes.

```
colorData = reshape(label2rgb(pointAttributes.Classification),[],3);
```

Visualize the color-coded point cloud.

```
figure
pcshow(ptCloud.Location,colorData)
```



Input Arguments

lasReader — LAS or LAZ file reader
lasFileReader object

LAS or LAZ file reader, specified as a `lasFileReader` object.

ptAtt — Point attributes
[ ] (default) | character vector | string scalar | cell array of character vectors | string array

Point attributes, specified as a character vector, string scalar, cell array of character vectors, or string array. The input must contain one or more of these options:

- `"Classification"`
- `"GPSTimeStamp"`
- `"LaserReturns"`
- `"NearIR"`
- `"ScanAngle"`

Each specified point attribute is returned as a field in the name of structure `ptAttributes`.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'ROI',[5 10 5 10 5 10] sets the region of interest (ROI) in which the function reads the point cloud.

ROI — ROI to read in the point cloud
[-inf inf -inf inf -inf inf] (default) | six-element row vector

ROI to read in the point cloud, specified as the comma-separated pair consisting of 'ROI' and a six-element row vector in the order, [x_{min} x_{max} y_{min} y_{max} z_{min} z_{max}]. Each element must be a real number. The values specify the ROI boundaries in the *x*-, *y*-, and *z*-axis.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

GpsTimeSpan — Range of GPS timestamps
lasReader.GPSTimeLimits (default) | two-element vector of duration objects

Range of GPS timestamps, specified as the comma-separated pair consisting of 'GpsTimeSpan' and a two-element vector of duration objects, that denotes [*startTime endTime*]. The timestamps must be positive.

Data Types: duration

Classification — Classification numbers of interest
0:lasReader.NumClasses - 1 (default) | vector of valid classification numbers

Classification numbers of interest, specified as the comma-separated pair consisting of 'Classification' and a vector of valid classification numbers.

Valid classification numbers range from 0 to 255.

Classification Number	Classification Type
0	Created, never classified
1	Unclassified
2	Ground
3	Low vegetation
4	Medium vegetation
5	High vegetation
6	Building
7	Low point (noise)
8	Reserved
9	Water
10	Rail
11	Road surface
12	Reserved

Classification Number	Classification Type
13	Wire guard (shield)
14	Wire conductor (phase)
15	Transmission tower
16	Wire-structure connector (insulator)
17	Bridge deck
18	High noise
19 - 63	Reserved
64 - 255	User-defined

These are standard classes and class-object mappings might differ from the standard classes depending on the application that created the LAS or LAZ file.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LaserReturns` — Number of points segregated by their return numbers
`1:lasReader.NumReturns` (default) | vector of valid return numbers

Number of points segregated by their return numbers, specified as the comma-separated pair consisting of `'LaserReturns'` and a vector of valid return numbers. Valid return numbers are integers from 1 to the value of the NumReturns property of the input `lasFileReader` object. For each value, *i*, in the vector, the function returns a point cloud of only the points that have *i* as their return number.

The return number is the number of times a laser pulse reflects back to the sensor.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, returned as a `pointCloud` object.

`ptAttributes` — Point attributes data
structure

Point attributes data, returned as a structure of fields that correspond to point attributes. The `ptAtt` input specifies the fields for this structure. The structure contains data for the specified attributes of each point in the `ptCloud` output.

Data Types: `struct`

See Also

Functions
`pcread` | `pcshow`

Objects
`ibeoLidarReader` | `lasFileReader` | `pointCloud` | `velodyneFileReader`

Introduced in R2020b

lidarScan

Create object for storing 2-D lidar scan

Description

A `lidarScan` object contains data for a single 2-D lidar (light detection and ranging) scan. The lidar scan is a laser scan for a 2-D plane with distances (`Ranges`) measured from the sensor to obstacles in the environment at specific angles (`Angles`). Use this laser scan object as an input to other robotics algorithms such as `matchScans`, `controllerVFH`, or `monteCarloLocalization`.

Creation

Syntax

```
scan = lidarScan(ranges,angles)
scan = lidarScan(cart)
```

Description

`scan = lidarScan(ranges,angles)` creates a `lidarScan` object from the `ranges` and `angles`, that represent the data collected from a lidar sensor. The `ranges` and `angles` inputs are vectors of the same length and are set directly to the `Ranges` and `Angles` properties.

`scan = lidarScan(cart)` creates a `lidarScan` object using the input Cartesian coordinates as an *n*-by-2 matrix. The `Cartesian` property is set directly from this input.

`scan = lidarScan(scanMsg)` creates a `lidarScan` object from a `LaserScan` ROS message object.

Properties

`Ranges` — Range readings from lidar
vector

Range readings from lidar, specified as a vector. This vector is the same length as `Angles`, and the vector elements are measured in meters.

Data Types: `single` | `double`

`Angles` — Angle of readings from lidar
vector

Angle of range readings from lidar, specified as a vector. This vector is the same length as `Ranges`, and the vector elements are measured in radians. Angles are measured counter-clockwise around the positive *z*-axis.

Data Types: `single` | `double`

`Cartesian` — Cartesian coordinates of lidar readings
`[x y]` matrix

Cartesian coordinates of lidar readings, returned as an `[x y]` matrix. In the lidar coordinate frame, positive *x* is forward and positive *y* is to the left.

Data Types: `single` | `double`

`Count` — Number of lidar readings
scalar

Number of lidar readings, returned as a scalar. This scalar is also equal to the length of the `Ranges` and `Angles` vectors or the number of rows in `Cartesian`.

Data Types: `double`

Object Functions

plot	Display laser or lidar scan readings
removeInvalidData	Remove invalid range and angle data

Examples

Plot Lidar Scan and Remove Invalid Points

Specify lidar data as vectors of ranges and angles. These values include readings outside of the sensors range.

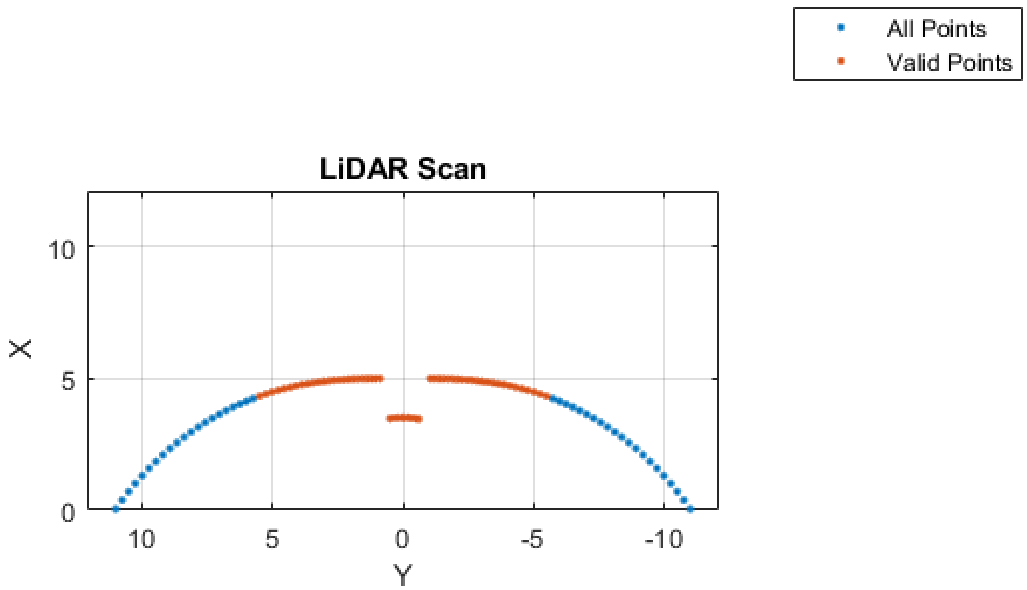
```
x = linspace(-2,2);
ranges = abs((1.5).*x.^2 + 5);
ranges(45:55) = 3.5;
angles = linspace(-pi/2,pi/2,numel(ranges));
```

Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

```
scan = lidarScan(ranges,angles);
plot(scan)
```

Remove invalid points based on a specified minimum and maximum range.

```
minRange = 0.1;
maxRange = 7;
scan2 = removeInvalidData(scan,'RangeLimits',[minRange maxRange]);
hold on
plot(scan2)
legend('All Points','Valid Points')
```



Match Lidar Scans

Create a reference lidar scan using `lidarScan`. Specify ranges and angles as vectors.

```
refRanges = 5*ones(1,300);
refAngles = linspace(-pi/2,pi/2,300);
refScan = lidarScan(refRanges,refAngles);
```

Using the `transformScan` (Robotics System Toolbox) function, generate a second lidar scan at an `x,y` offset of `(0.5,0.2)`.

```
currScan = transformScan(refScan,[0.5 0.2 0]);
```

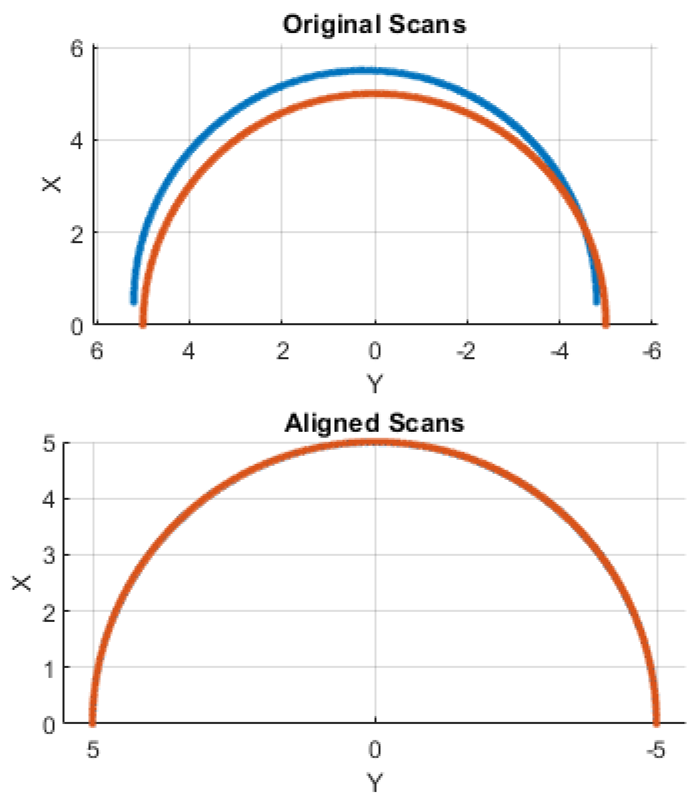
Match the reference scan and the second scan to estimate the pose difference between them.

```
pose = matchScans(currScan,refScan);
```

Use the `transformScan` function to align the scans by transforming the second scan into the frame of the first scan using the relative pose difference. Plot both the original scans and the aligned scans.

```
currScan2 = transformScan(currScan,pose);
subplot(2,1,1);
hold on
plot(currScan)
plot(refScan)
```

```
title('Original Scans')
hold off
subplot(2,1,2);
hold on
plot(currScan2)
plot(refScan)
title('Aligned Scans')
xlim([0 5])
hold off
```



Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

Lidar scans require a limited size in code generation. The lidar scans are limited to 4000 points (range and angles) as a maximum.

See Also

`matchScans`

Introduced in R2020b

plot

Display laser or lidar scan readings

Syntax

```
plot(scanObj)
plot( ___ ,Name,Value)
linehandle = plot( ___ )
```

Description

`plot(scanObj)` plots the lidar scan readings specified in `scanObj`.

`plot( ___ ,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments.

`linehandle = plot( ___ )` returns a column vector of line series handles, using any of the arguments from previous syntaxes. Use `linehandle` to modify properties of the line series after it is created.

Examples

Plot Lidar Scan and Remove Invalid Points

Specify lidar data as vectors of ranges and angles. These values include readings outside of the sensors range.

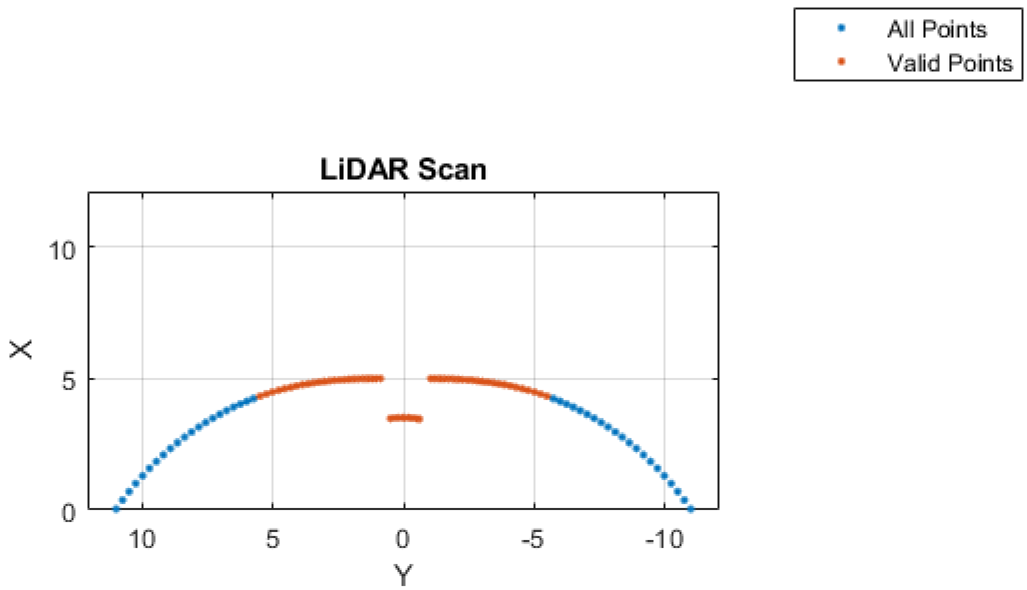
```
x = linspace(-2,2);
ranges = abs((1.5).*x.^2 + 5);
ranges(45:55) = 3.5;
angles = linspace(-pi/2,pi/2,numel(ranges));
```

Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

```
scan = lidarScan(ranges,angles);
plot(scan)
```

Remove invalid points based on a specified minimum and maximum range.

```
minRange = 0.1;
maxRange = 7;
scan2 = removeInvalidData(scan,'RangeLimits',[minRange maxRange]);
hold on
plot(scan2)
legend('All Points','Valid Points')
```



Input Arguments

scanObj — Lidar scan readings
lidarScan object

Lidar scan readings, specified as a `lidarScan` 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: "MaximumRange",5

Parent — Parent of axes
axes object

Parent of axes, specified as the comma-separated pair consisting of `"Parent"` and an axes object in which the laser scan is drawn. By default, the laser scan is plotted in the currently active axes.

MaximumRange — Range of laser scan
scan.RangeMax (default) | scalar

Range of laser scan, specified as the comma-separated pair consisting of `"MaximumRange"` and a scalar. When you specify this name-value pair argument, the minimum and maximum *x*-axis and the maximum *y*-axis limits are set based on specified value. The minimum *y*-axis limit is automatically determined by the opening angle of the laser scanner.

This name-value pair only works when you input `scanMsg` as the laser scan.

Outputs

`linehandle` — One or more chart line objects
scalar | vector

One or more chart line objects, returned as a scalar or a vector. These are unique identifiers, which you can use to query and modify properties of a specific chart line.

See Also
`matchScans`

Introduced in R2020b

removeInvalidData

Remove invalid range and angle data

Syntax

```
validScan = removeInvalidData(scan)
validScan = removeInvalidData(scan,Name,Value)
```

Description

`validScan = removeInvalidData(scan)` returns a new `lidarScan` object with all `Inf` and `NaN` values from the input `scan` removed. The corresponding angle readings are also removed.

`validScan = removeInvalidData(scan,Name,Value)` provides additional options specified by one or more `Name,Value` pairs.

Examples

Plot Lidar Scan and Remove Invalid Points

Specify lidar data as vectors of ranges and angles. These values include readings outside of the sensors range.

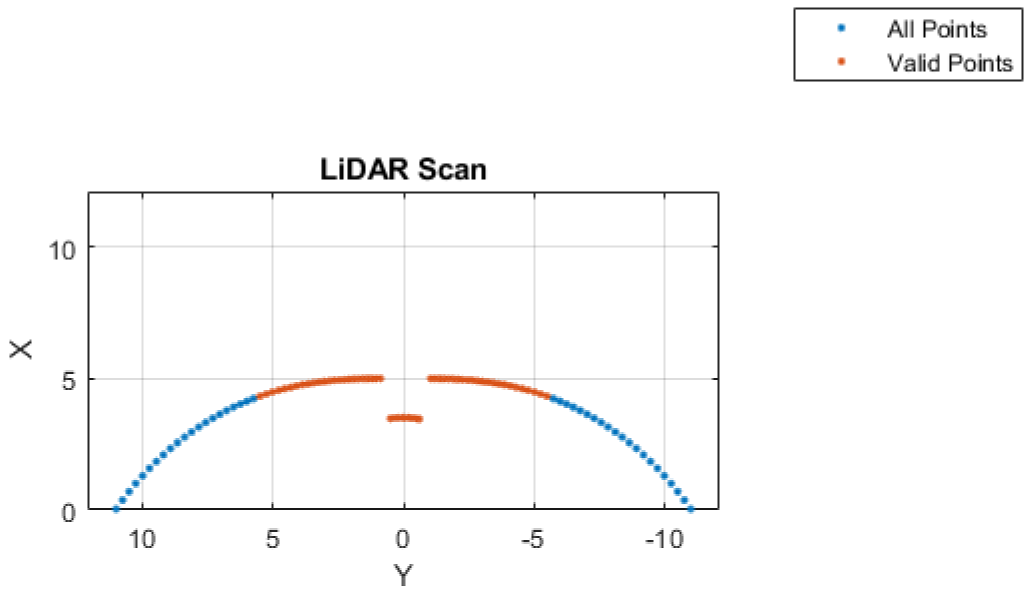
```
x = linspace(-2,2);
ranges = abs((1.5).*x.^2 + 5);
ranges(45:55) = 3.5;
angles = linspace(-pi/2,pi/2,numel(ranges));
```

Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

```
scan = lidarScan(ranges,angles);
plot(scan)
```

Remove invalid points based on a specified minimum and maximum range.

```
minRange = 0.1;
maxRange = 7;
scan2 = removeInvalidData(scan,'RangeLimits',[minRange maxRange]);
hold on
plot(scan2)
legend('All Points','Valid Points')
```



Input Arguments

`scan` — Lidar scan readings

`lidarScan` object

Lidar scan readings, specified as a `lidarScan` 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: `["RangeLimits",[0.05 2]`

`RangeLimits` — Range reading limits

two-element vector

Range reading limits, specified as a two-element vector, `[minRange maxRange]`, in meters. All range readings and corresponding angles outside these range limits are removed

Data Types: `single` | `double`

`AngleLimits` — Angle limits

two-element vector

Angle limits, specified as a two-element vector, `[minAngle maxAngle]` in radians. All angles and corresponding range readings outside these angle limits are removed.

Angles are measured counter-clockwise around the positive*z*-axis.

Data Types: `single` | `double`

Output Arguments

`validScan` — Lidar scan readings
`lidarScan` object

Lidar scan readings, specified as a `lidarScan` object. All invalid lidar scan readings are removed.

See Also
`matchScans`

Introduced in R2020b

rangeSensor

Simulate range-bearing sensor readings

Description

The `rangeSensor` System object™ is a range-bearing sensor that is capable of outputting range and angle measurements based on the given sensor pose and occupancy map. The range-bearing readings are based on the obstacles in the occupancy map.

To simulate a range-bearing sensor using this object:

1. Create the `rangeSensor` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
rbsensor = rangeSensor
rbsensor = rangeSensor(Name,Value)
```

Description

`rbsensor = rangeSensor` returns a `rangeSensor` System object, `rbsensor`. The sensor is capable of outputting range and angle measurements based on the sensor pose and an occupancy map.

`rbsensor = rangeSensor(Name,Value)` sets properties for the sensor using one or more name-value pairs. Unspecified properties have default values. 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.

Range — Minimum and maximum detectable range
`[0 20]` (default) | 1-by-2 positive real-valued vector

The minimum and maximum detectable range, specified as a 1-by-2 positive real-valued vector. Units are in meters.

Example: `[1 15]`

Tunable: Yes

Data Types: `single` | `double`

`HorizontalAngle` — Minimum and maximum horizontal detection angle
`[-pi pi]` (default) | 1-by-2 real-valued vector

Minimum and maximum horizontal detection angle, specified as a 1-by-2 real-valued vector. Units are in radians.

Example: `[-pi/3 pi/3]`

Data Types: `single` | `double`

`HorizontalAngleResolution` — Resolution of horizontal angle readings
`0.0244` (default) | positive scalar

Resolution of horizontal angle readings, specified as a positive scalar. The resolution defines the angular interval between two consecutive sensor readings. Units are in radians.

Example: `0.01`

Data Types: `single` | `double`

`RangeNoise` — Standard deviation of range noise
`0` (default) | positive scalar

The standard deviation of range noise, specified as a positive scalar. The range noise is modeled as a zero-mean white noise process with the specified standard deviation. Units are in meters.

Example: `0.01`

Tunable: Yes

Data Types: `single` | `double`

`HorizontalAngleNoise` — Standard deviation of horizontal angle noise
`0` (default) | positive scalar

The standard deviation of horizontal angle noise, specified as a positive scalar. The range noise is modeled as a zero-mean white noise process with the specified standard deviation. Units are in radians.

Example: `0.01`

Tunable: Yes

Data Types: `single` | `double`

`NumReadings` — Number of output readings
`258` (default) | positive integer

This property is read-only.

Number of output readings for each pose of the sensor, specified as a positive integer. This property depends on the `HorizonalAngle` and `HorizontalAngleResolution` properties.

Data Types: `single` | `double`

Usage

Syntax

```
[ranges,angles] = rbsensor(pose,map)
```

Description

`[ranges,angles] = rbsensor(pose,map)` returns the range and angle readings from the 2-D pose information and the ground-truth map.

Input Arguments

pose — Pose of sensor in map
N-by-3 real-valued matrix

Poses of the sensor in the 2-D map, specified as an *N*-by-3 real-valued matrix, where *N* is the number of poses to simulate the sensor. Each row of the matrix corresponds to a pose of the sensor in the order of [x, y, θ]. x and y represent the position of the sensor in the map frame. The units of x and y are in meters. θ is the heading angle of the sensor with respect to the positive x-direction of the map frame. The units of θ are in radians.

map — Ground-truth map
`occupancyMap` object | `binaryOccupancyMap` object

Ground-truth map, specified as an `occupancyMap` or a `binaryOccupancyMap` object. For the `occupancyMap` input, the range-bearing sensor considers a cell as occupied and returns a range reading if the occupancy probability of the cell is greater than the value specified by the `OccupiedThreshold` property of the occupancy map.

Output Arguments

ranges — Range readings
R-by-*N* real-valued matrix

Range readings, specified as an *R*-by-*N* real-valued matrix. *N* is the number of poses for which the sensor is simulated, and *R* is the number of sensor readings per pose of the sensor. *R* is same as the value of the `NumReadings` property.

angles — Angle readings
R-by-1 real-valued vector

Angle readings, specified as an *R*-by-1 real-valued vector. *R* is the number of sensor readings per pose of the sensor. *R* is same as the value of the `NumReadings` property.

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)
```

Common to All System Objects

step Run System object algorithm
clone Create duplicate System object

Examples

Obtain Range and Bearing Readings

Create a range-bearing sensor.

```
rbsensor = rangeSensor;
```

Specify the pose of the sensor and the ground-truth map.

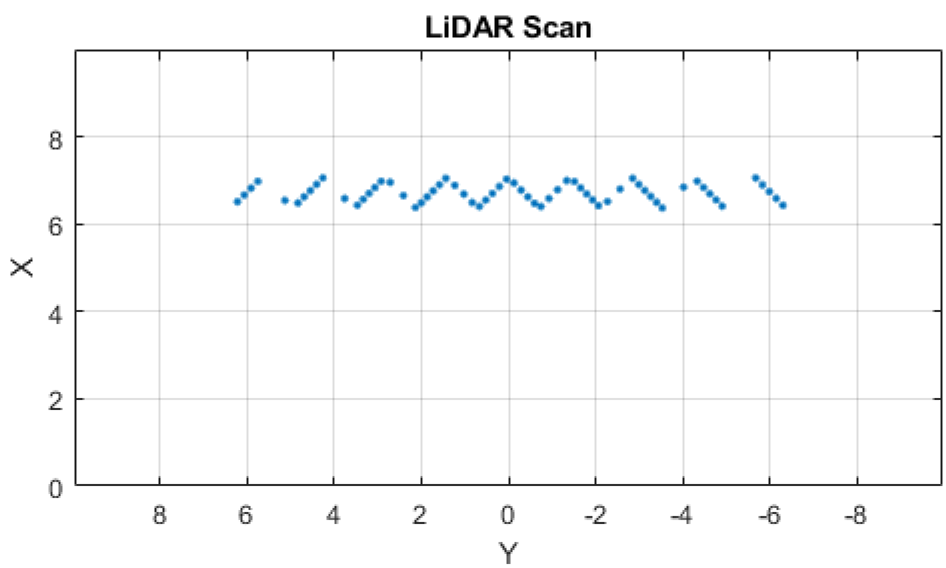
```
truePose = [0 0 pi/4];
trueMap = binaryOccupancyMap(eye(10));
```

Generate the sensor readings.

```
[ranges, angles] = rbsensor(truePose, trueMap);
```

Visualize the results using `lidarScan`.

```
scan = lidarScan(ranges, angles);
figure
plot(scan)
```



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

`binaryOccupancyMap` | `lidarScan` | `occupancyMap`

Introduced in R2020b

Functions

pcfitcuboid

Fit cuboid over point cloud

Syntax

```
model = pcfitcuboid(ptCloudIn)
model = pcfitcuboid(ptCloudIn,indices)
model = pcfitcuboid( ___ ,Name,Value)
```

Description

`model = pcfitcuboid(ptCloudIn)` fits a cuboid over the input point cloud data. The function stores the properties of the cuboid in the `cuboidModel` object, `model`.

`model = pcfitcuboid(ptCloudIn,indices)` fits a cuboid over a selected set of points, `indices`, in the input point cloud.

`model = pcfitcuboid( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to any of the input argument combinations in previous syntaxes. For example, `'AzimuthRange,[-75 75]'` sets the angular range for the azimuth angles of the function.

Examples

Fit Cuboid Over Point Cloud Data

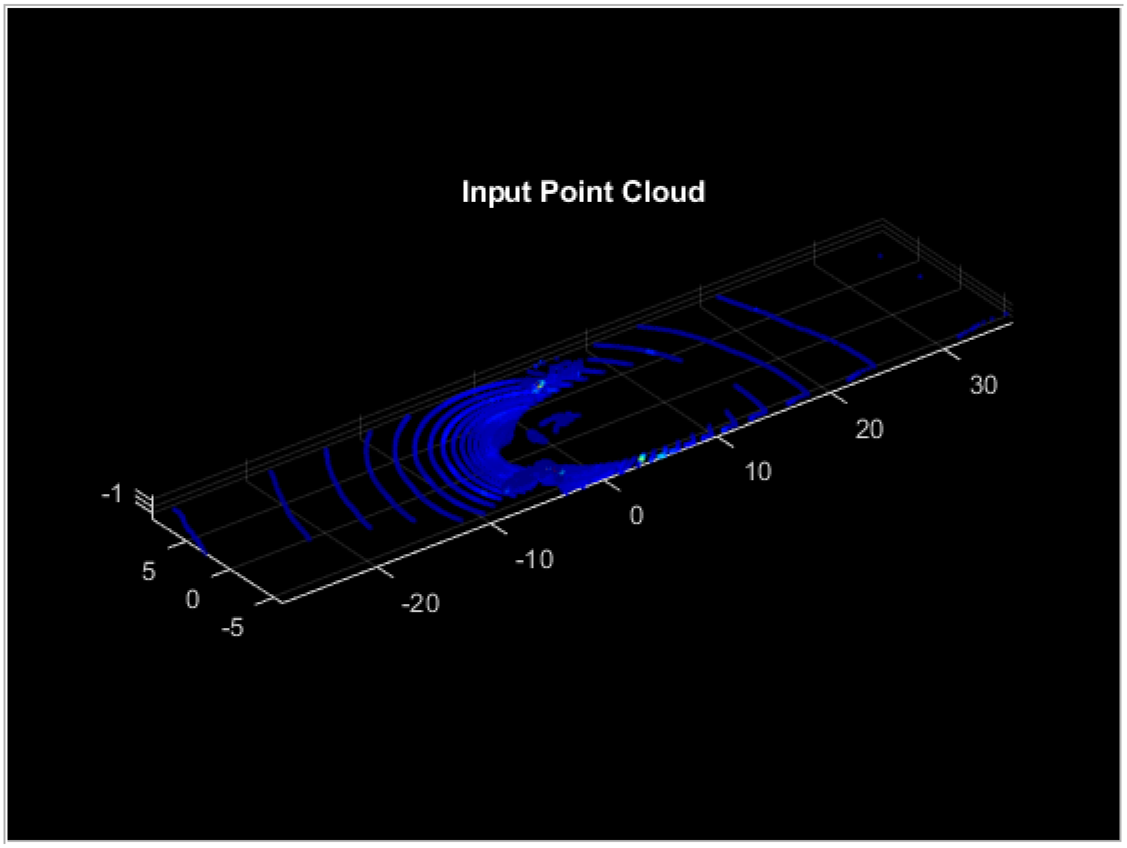
Fit bounding boxes around clusters in a point cloud.

Load the point cloud data into the workspace.

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

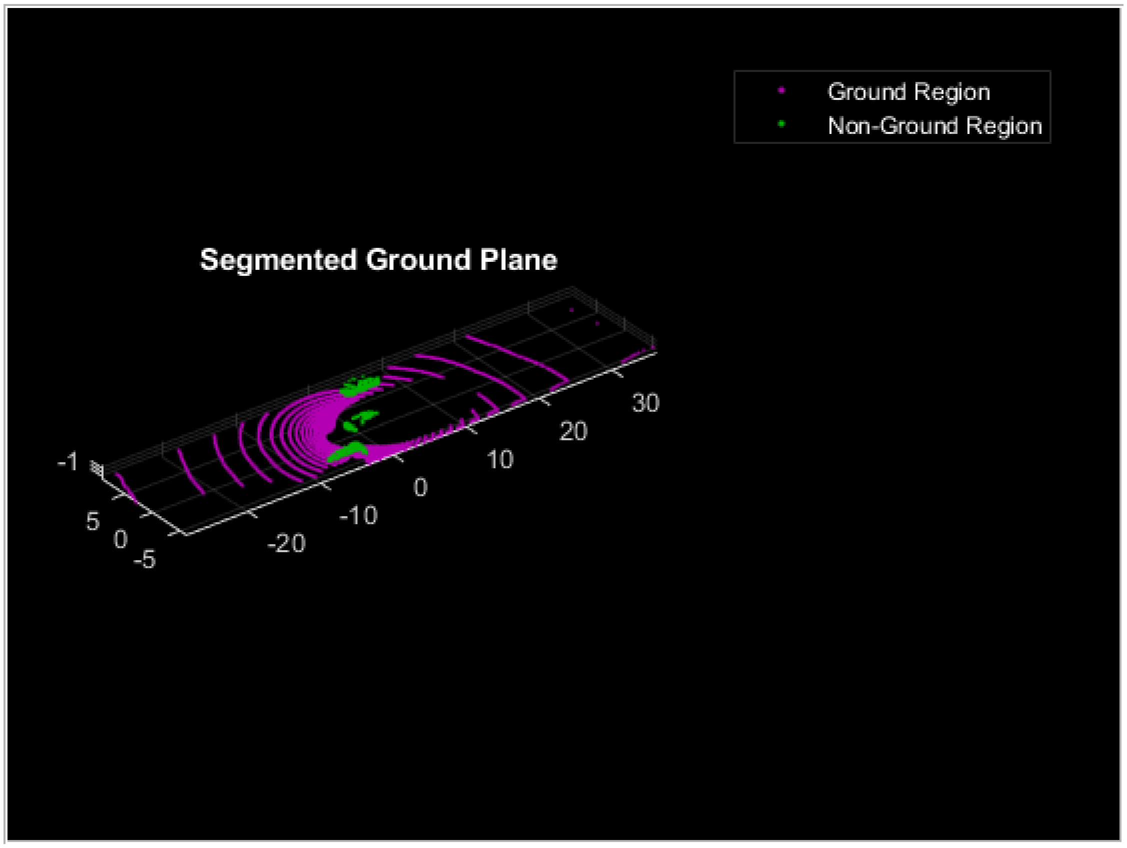
Define and crop a region of interest (ROI) from the point cloud. Visualize the selected ROI of the point cloud.

```
roi = [-40 40 -6 9 -2 1];
in = findPointsInROI(data.ptCloud,roi);
ptCloudIn = select(data.ptCloud,in);
hcluster = figure;
panel = uipanel('Parent',hcluster,'BackgroundColor',[0 0 0]);
ax = axes('Parent',panel,'Color',[0 0 0]);
pcshow(ptCloudIn,'MarkerSize',30,'Parent',ax)
title('Input Point Cloud')
```



Segment the ground plane. Visualize the segmented ground plane.

```
maxDistance = 0.3;
referenceVector = [0 0 1];
[~,inliers,outliers] = pcfitplane(ptCloudIn,maxDistance,referenceVector);
ptCloudWithoutGround = select(ptCloudIn,outliers,'OutputSize','full');
hSegment = figure;
panel = uipanel('Parent',hSegment,'BackgroundColor',[0 0 0]);
ax = axes('Parent',panel,'Color',[0 0 0]);
pcshowpair(ptCloudIn,ptCloudWithoutGround,'Parent',ax)
legend('Ground Region','Non-Ground Region','TextColor', [1 1 1])
title('Segmented Ground Plane')
```

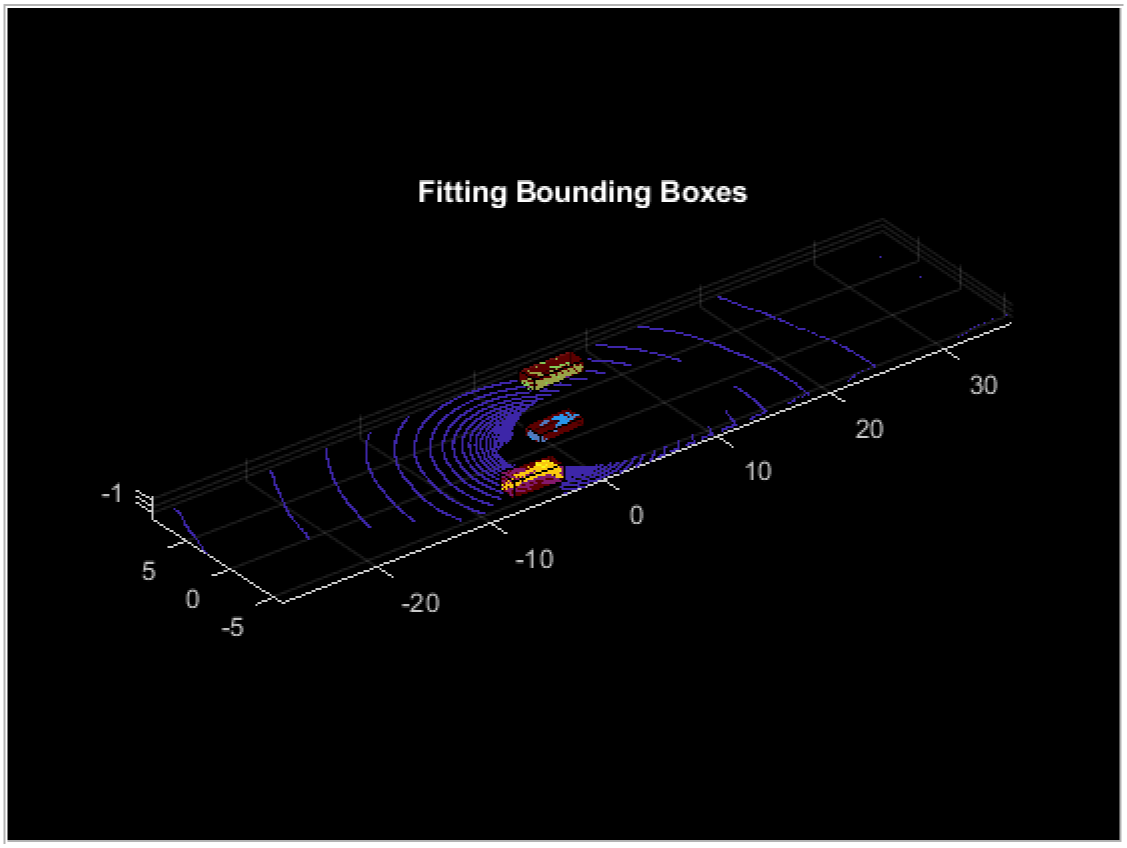


Segment the non-ground region of the point cloud into clusters. Visualize the segmented point cloud.

```
distThreshold = 1;
[labels,numClusters] = pcsegdist(ptCloudWithoutGround,distThreshold);
labelColorIndex = labels;
hCuboid = figure;
panel = uipanel('Parent',hCuboid,'BackgroundColor',[0 0 0]);
ax = axes('Parent',panel,'Color',[0 0 0]);
pcshow(ptCloudIn.Location,labelColorIndex,'Parent',ax)
title('Fitting Bounding Boxes')
hold on
```

Fit bounding box on each cluster, visualized as orange highlights.

```
for i = 1:numClusters
    idx = find(labels == i);
    model = pcfitcuboid(ptCloudWithoutGround,idx);
    plot(model)
end
```



Input Arguments

ptCloudIn — Point cloud
pointCloud object

Point cloud, specified as a pointCloud object.

indices — Indices of selected valid points
vector of positive integers

Indices of selected valid points, specified as a vector of positive integers.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

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: 'AzimuthRange',[-75 75] sets the angular range for the azimuth angles of the function.

AzimuthRange — Range of azimuth angles
[0 90] (default) | two-element row vector of real values

Range of azimuth angles over which to identify the orientation of the cuboid, specified as the comma-separated pair consisting of 'AzimuthRange' and a two-element row vector of real values in the range [0, 90].

Data Types: single | double

Resolution — Step size of search window
1 (default) | positive scalar

Step size of search window, specified as the comma-separated pair consisting of 'Resolution' and a positive scalar. The specified value must be less than or equal to the distance between the upper and lower bounds of the range of azimuth angles. For example, if the range of azimuth angles is [0, 90], the specified value must be less than or equal to 90.

Note Decreasing the resolution will increase the computation time and memory footprint.

Data Types: single | double

Output Arguments

model — Cuboid model
cuboidModel object

Cuboid model, returned as a cuboidModel object.

References

[1] Xiao Zhang, Wenda Xu, Chiyu Dong and John M. Dolan, "Efficient L-Shape Fitting for Vehicle Detection Using Laser Scanners", IEEE Intelligent Vehicles Symposium, June 2018

Extended Capabilities

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

See Also

Functions
pcfitcylinder | pcfitplane | pcfitsphere

Objects
cuboidModel | pointCloud

Introduced in R2020b

extractFPFHFeatures

Extract fast point feature histogram (FPFH) descriptors from point cloud

Syntax

```
features = extractFPFHFeatures(ptCloudIn)
features = extractFPFHFeatures(ptCloudIn,indices)
features = extractFPFHFeatures(ptCloudIn,row,column)
[ ___ ,validIndices] = extractFPFHFeatures( ___ )
[ ___ ] = extractFPFHFeatures( ___ ,Name,Value)
```

Description

`features = extractFPFHFeatures(ptCloudIn)` extracts FPFH descriptors for each valid point in the input point cloud object. The function returns descriptors as an *N*-by-33 matrix, where *N* is the number of valid points in the input point cloud.

`features = extractFPFHFeatures(ptCloudIn,indices)` extracts FPFH descriptors for the valid points located at the specified linear indices, `indices`.

`features = extractFPFHFeatures(ptCloudIn,row,column)` extracts FPFH descriptors for the valid points at the specified 2-D indices of the input organized point cloud `ptCloudIn`. Specify the row and column indices of the points as `row` and `column`, respectively.

`[ ___ ,validIndices] = extractFPFHFeatures( ___ )` returns the linear indices of valid points in the point cloud for which FPFH descriptors have been extracted.

`[ ___ ] = extractFPFHFeatures( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to any combination of arguments in previous syntaxes.

Descriptors can be extracted using KNN search method, radius search method or a combination of both. The `extractFPFHFeatures` function uses KNN search method to extract descriptors by default. The users can choose the method of extraction through the name-value pair arguments. For example, `'NumNeighbors',8` selects the KNN search method to extract descriptors and sets maximum number of neighbors to consider in the k-nearest neighbor (KNN) search method to eight.

Examples

Extract FPFH Descriptors at Selected Indices in Point Cloud

Load point cloud data into the workspace.

```
ptObj = pcread('teapot.ply');
```

Downsample the point cloud data.

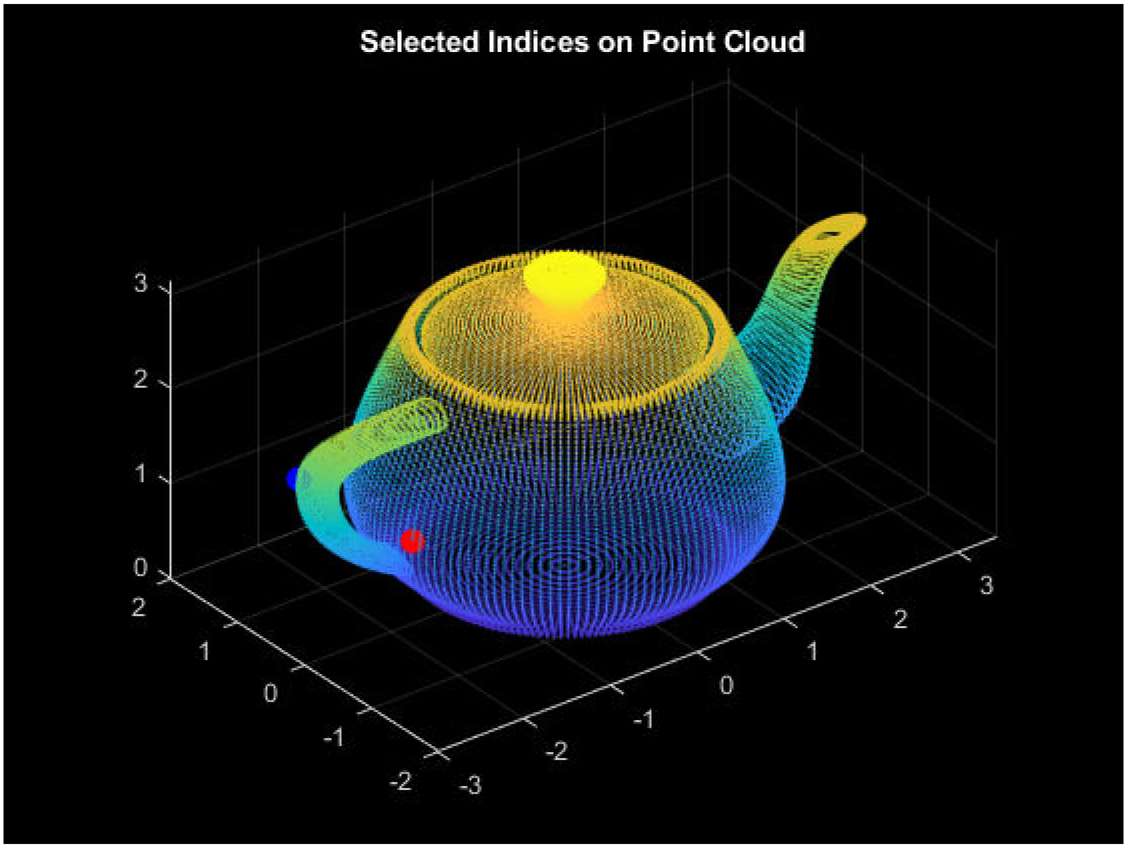
```
ptCloudIn = pcdownsample(ptObj,'gridAverage',0.05);
```

Extract FPFH descriptors for the points at specified key indices.

```
keyInds = [6565 10000];
features = extractFPFHFeatures(ptCloudIn,keyInds);
```

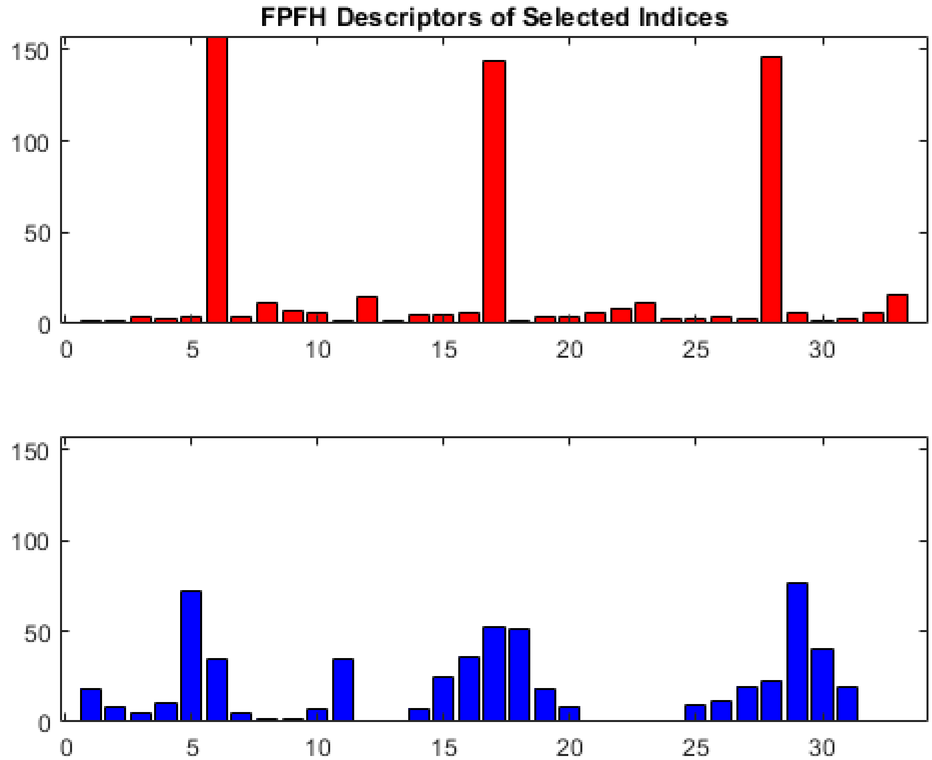
Display the key points on the point cloud.

```
ptKeyObj = pointCloud(ptCloudIn.Location(keyInds,:),'Color',[255 0 0;0 0 255]);
figure
pcshow(ptObj)
title('Selected Indices on Point Cloud')
hold on
pcshow(ptKeyObj,'MarkerSize',1000)
hold off
```



Display the extracted FPFH descriptors at key points.

```
figure
ax1 = subplot(2,1,1);
bar(features(1,:),'FaceColor',[1 0 0])
title('FPFH Descriptors of Selected Indices')
ax2 = subplot(2,1,2);
bar(features(2,:),'FaceColor',[0 0 1])
linkaxes([ax1 ax2],'xy')
```



Input Arguments

ptCloudIn — Point cloud
pointCloud object

Point cloud, specified as a `pointCloud` object.

indices — Linear indices of selected points
vector of positive integers

Linear indices of selected points, specified as a vector of positive integers.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

row — Row indices of selected points
vector of positive integers

Row indices of selected points in an organized point cloud, specified as a vector of positive integers.

The `row` and `column` vectors must be of the same length.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

column — Column indices of selected points
vector of positive integers

Column indices of selected points in an organized point cloud, specified as a vector of positive integers.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

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: `'NumNeighbors'`,8 sets the maximum number of neighbors to consider in the k-nearest neighbor (KNN) search method to eight.

`NumNeighbors` — Number of neighbors for KNN search
`50` (default) | positive integer

Number of neighbors for the KNN search method, specified as the comma-separated pair consisting of `'NumNeighbors'` and a positive integer.

KNN search method calculates the distance between a point and its adjacent points in a point cloud and sorts them in ascending order. Closest points are considered as neighbors. `'NumNeighbors'` sets the upper limit for the number of neighbors to consider.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Radius` — Radius considered for radius search
`0.05` (default) | positive real-valued scalar

Radius considered for radius search method, specified as the comma-separated pair consisting of `'Radius'` and a positive real-valued scalar.

Radius search method sets a particular radius around a point and selects all the adjacent points within that given radius as neighbors.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Note If you specify values for both the `'NumNeighbors'` and `'Radius'` name-value pair arguments, the `extractFPFHFeatures` function performs the KNN search method, and then selects only those of that set within the given radius.

If you specify large values for `'NumNeighbors'` and `'Radius'`, the memory footprint and computation time increase.

Output Arguments

`features` — FPFH descriptors
N-by-33 matrix of positive real values

FPFH descriptors, returned as a *N*-by-33 matrix of positive real values. *N* is the number of valid points from which the function extracts FPFH descriptors. Each column contains the FPFH descriptors for a valid point in the point cloud. To additionally return the indices of the extracted points, use the `validIndices` output argument.

Data Types: `double`

`validIndices` — Linear indices of valid points
vector of positive integers

Linear indices of valid points, specified as a vector of positive integers. The vector contains the indices of only those points for which the function extracts features.

Data Types: `double`

References

[1] Rusu, Radu Bogdan, Nico Blodow, and Michael Beetz. "Fast point feature histograms (FPFH) for 3D registration." In *2009 IEEE International Conference on Robotics and Automation*, pp. 3212-3217. IEEE, 2009.

Extended Capabilities

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

See Also

Functions
`pcdownsample` | `pcnormals` | `pcread` | `pcshow`

Objects
`pointCloud`

Introduced in R2020b

pcmedian

Median filtering 3-D point cloud data

Syntax

```
ptCloudOut = pcmedian(ptCloudIn)
ptCloudOut = pcmedian( ___ ,Name,Value)
```

Description

`ptCloudOut = pcmedian(ptCloudIn)` performs median filtering of 3-D point cloud data. The function filters each channel of the point cloud individually. The output is a filtered point cloud. Each output location property value is the median of neighborhood around the corresponding input location property value. The `pcmedian` function doesn't pad zeros on the edges. Rather, it operates only on the available neighborhood values.

If the input point cloud is an organized point cloud, the `pcmedian` function uses *N*-by-*N* neighborhood method. If the point cloud is unorganized, the function uses radial neighborhood method.

`ptCloudOut = pcmedian( ___ ,Name,Value)` specifies options using one or more name-value pair arguments. For example, `'FilterSize',3` sets the size of the median filter for organized point clouds to 3.

Examples

Median Filter Noisy Point Cloud

Use the median filter to remove noise from a point cloud. First, add random noise to a point cloud. Then, use the `pcmedian` function to filter the noise.

Create a point cloud.

```
gv = 0:0.01:1;
[X,Y] = meshgrid(gv,gv);
Z = X.^2 + Y.^2;
ptCloud = pointCloud(cat(3,X,Y,Z));
```

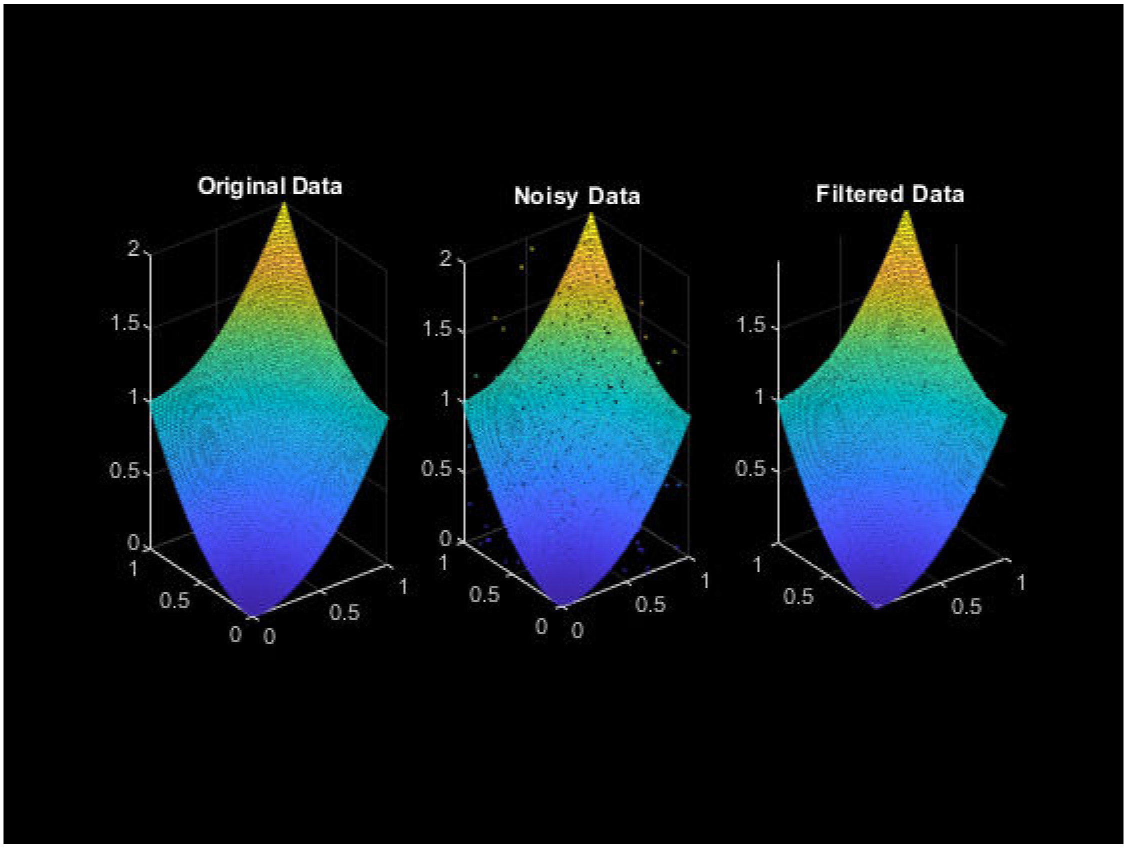
Add random noise along the *z*-axis.

```
temp = ptCloud.Location;
count = numel(temp(:,:,3));
temp((2*count) + randperm(count,100)) = rand(1,100);
temp(count + randperm(count,100)) = rand(1,100);
temp(randperm(count,100)) = rand(1,100);
ptCloudA = pointCloud(temp);
```

Apply the median filter and display the three point clouds (original, noisy, and filtered).

```
ptCloudB = pcmedian(ptCloudA);
```

```
subplot(1,3,1)
pcshow(ptCloud)
title('Original Data')
subplot(1,3,2)
pcshow(ptCloudA)
title('Noisy Data')
subplot(1,3,3)
pcshow(ptCloudB)
title('Filtered Data')
```



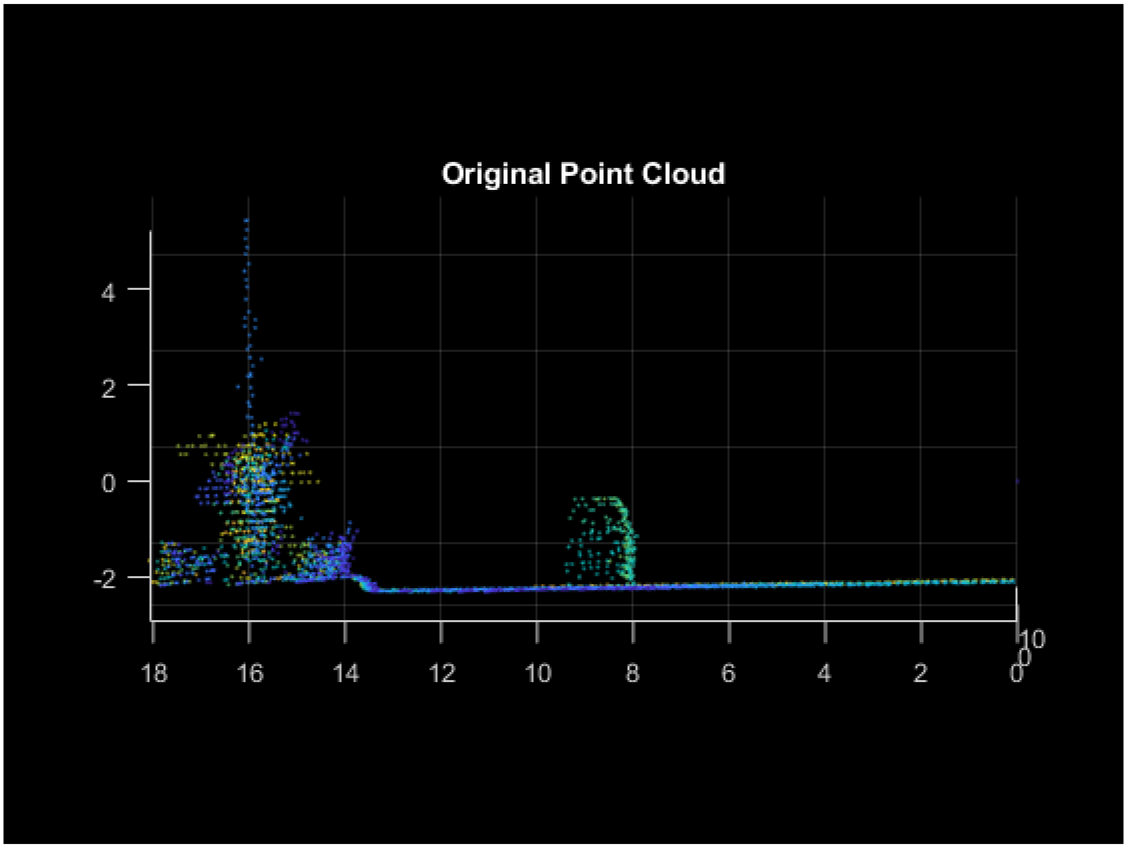
Apply Median Filter on Unorganized Point Cloud Data

Load point cloud data into the workspace.

```
ptCloud = pcread('highwayScene.pcd');
roi = [0 20 0 20 -5 15];
indices = findPointsInROI(ptCloud,roi);
ptCloud = select(ptCloud,indices);
ptCloud = pcdownsample(ptCloud,'gridAverage',0.2);
```

Display the point cloud data. Each point is color-coded based on its x-coordinate.

```
figure
pcshow(ptCloud.Location,ptCloud.Location(:,1))
view(-90,2)
title('Original Point Cloud')
```

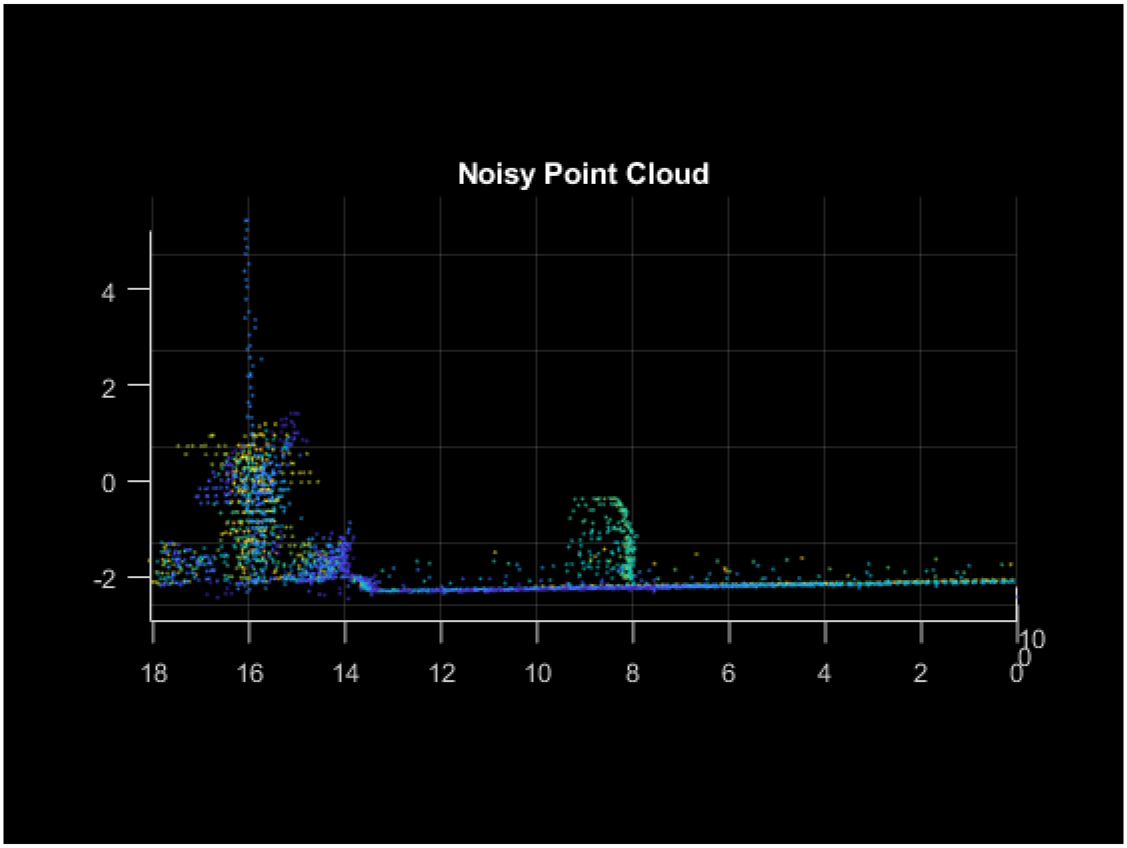


Add noise along z-channel in the interval (a,b). Values of a and b are chosen to make the noise appear close to the ground.

```
temp = ptCloud.Location;
count = numel(temp(:,3));
a = -2.5;
b = -2;
temp((2*count)+randperm(count,200)) = a+(b-a).*rand(1,200);
ptCloudA = pointCloud(temp);
```

Display the noisy point cloud. Each point is color-coded based on its x-coordinate.

```
figure
pcshow(ptCloudA.Location,ptCloudA.Location(:,1))
view(-90,2)
title('Noisy Point Cloud')
```

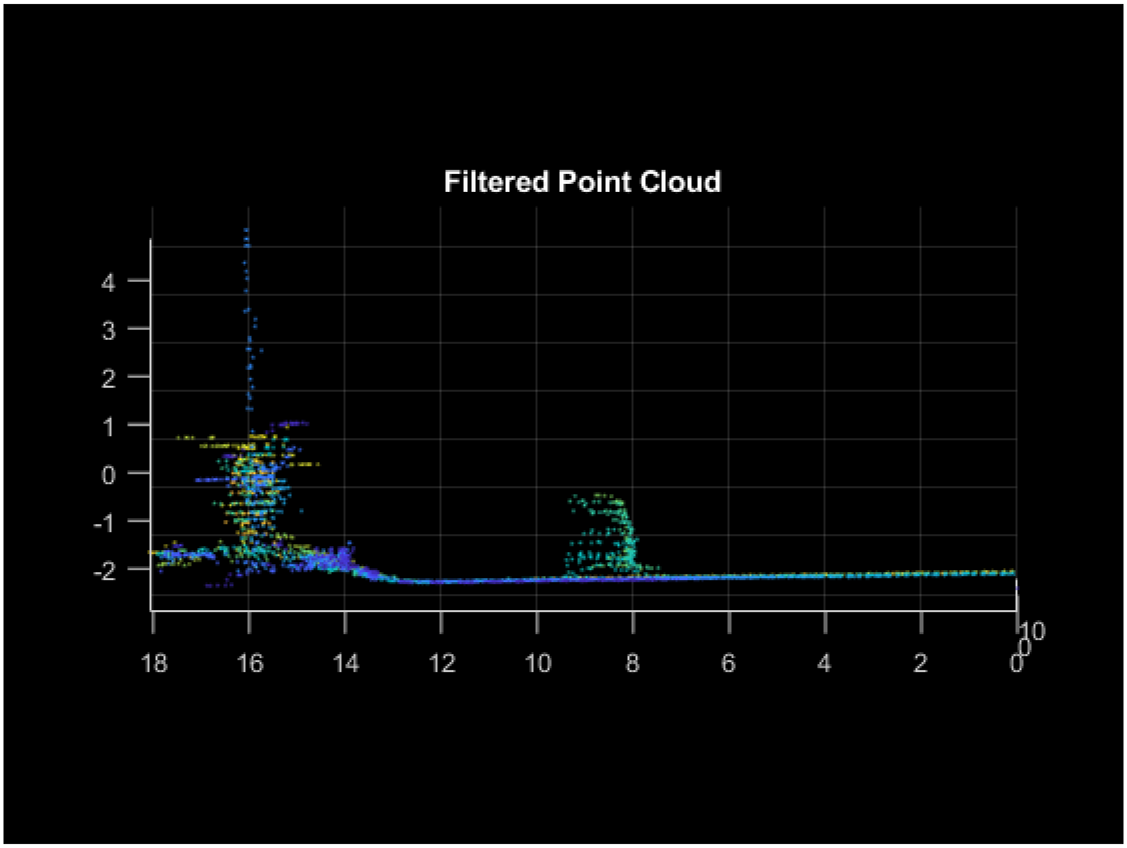


Apply median filter on the point cloud.

```
ptCloudB = pcmedian(ptCloudA,'Dimensions',3,'Radius',1);
```

Display the filtered point cloud. Each point is color-coded based on its x-coordinate.

```
figure
pcshow(ptCloudB.Location, ptCloudB.Location(:,1))
view(-90,2)
title('Filtered Point Cloud')
```



Input Arguments

ptCloudIn — Point cloud
pointCloud object

Point cloud, specified as a pointCloud object with at least one valid point. If the input point cloud is organized, the size of the point cloud must be at least 3-by-3-by-3.

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: 'FilterSize',3 specifies a median filter size of 3.

Dimensions — Point cloud dimensions of interest
[1 2 3] (default) | vector of integers in the range [1 3]

Point cloud dimensions of interest, specified as a vector of integers in the range [1 3]. The values 1, 2, and 3 correspond to the *x*-, *y*-, and *z*-axis respectively. You must specify dimensions in ascending order.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

FilterSize — Size of the median filter for organized point cloud
3 (default) | odd integer in the range [3, *N*]

Size of the median filter for an organized point cloud, specified as an odd integer in the range [3, *N*]. *N* is the smallest of channel dimensions in the point cloud.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Radius — Radius of neighborhood for unorganized point cloud
0.05 (default) | positive scalar

Radius of the neighborhood for unorganized point cloud, specified as a positive scalar.

Data Types: single | double

Output Arguments

ptCloudOut — Filtered point cloud
pointCloud object

Filtered point cloud, returned as a pointCloud object.

Extended Capabilities

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

See Also

Functions
pcdenoise | pcdownsample | pcmerge | pcshow | pctransform

Objects
pointCloud

Introduced in R2020b

estimateCheckerboardCorners3d

Estimate world frame coordinates of checkerboard corner points in image

Syntax

```
imageCorners3d = estimateCheckerboardCorners3d(I,cameraIntrinsic,checkerSize)
[imageCorners3d,boardDimensions] = estimateCheckerboardCorners3d(I,
cameraIntrinsic,checkerSize)
[ ___ ,imagesUsed] = estimateCheckerboardCorners3d(imageFileNames,
cameraIntrinsic,checkerSize)
[ ___ ] = estimateCheckerboardCorners3d(imageArray,cameraIntrinsic,checkerSize)
[ ___ ] = estimateCheckerboardCorners3d( ___ ,Name,Value)
```

Description

`imageCorners3d = estimateCheckerboardCorners3d(I,cameraIntrinsic,checkerSize)` estimates the world frame coordinates of the corner points of a checkerboard in an image, `I`, by using the camera intrinsic parameters `cameraIntrinsic` and the size of the checkerboard squares `checkerSize`.

`[imageCorners3d,boardDimensions] = estimateCheckerboardCorners3d(I, cameraIntrinsic,checkerSize)` additionally returns the checkerboard dimensions `boardDimensions`.

`[ ___ ,imagesUsed] = estimateCheckerboardCorners3d(imageFileNames, cameraIntrinsic,checkerSize)` estimates the world frame coordinates of the corner points of a checkerboard from a set of image files, `imageFileNames`. The function returns a logical vector that indicates in which images it detected a checkerboard, `imagesUsed`, in addition to any combination of output arguments from previous syntaxes.

`[ ___ ] = estimateCheckerboardCorners3d(imageArray,cameraIntrinsic,checkerSize)` estimates the world frame coordinates of the corner points of a checkerboard from an array of images, `imageArray`.

`[ ___ ] = estimateCheckerboardCorners3d( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to any combination of arguments from previous syntaxes. For example, `'MinCornerMetric',0.2` sets the threshold for the checkerboard corner metric to 0.2.

Examples

Detect Checkerboard Corners in Image

Detect a checkerboard in an image using the `estimateCheckerboardCorners3d` function and estimate the world frame coordinates of the checkerboard corners.

Read the image into the workspace.

```
Image = imread('CheckerboardImage.png');
```

Load the camera parameters into the workspace.

```
intrinsic = load('calibration.mat');
```

Set the size of the checkerboard squares in millimeters.

```
squareSize = 200;
```

Estimate the checkerboard corners.

```
boardCorners = estimateCheckerboardCorners3d(Image, ...
    intrinsic.cameraParams,squareSize)
```

```
boardCorners = 4×3

    1.2840   -0.5216    8.8913
    2.8614    0.5774    8.3401
    1.8230    2.0470    8.2984
    0.2455    0.9480    8.8496
```

Plot the corners on the input image.

```
imPts = projectLidarPointsOnImage(boardCorners,intrinsic.cameraParams,rigid3d());
J = undistortImage(Image,intrinsic.cameraParams);
imshow(J)
hold on
plot(imPts(:,1),imPts(:,2),'.r','MarkerSize',30)
title('Detected Checkerboard Corners')
hold off
```

Input Arguments

I — Image for detection
H-by-*W*-by-*C* array

Image for detection, specified as an *H*-by-*W*-by-*C* array where:

- *H* – Height of the image in pixels
- *W* – Width of the image in pixels
- *C* – Number of color channels

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

`imageFileNames` — Image file names
character vector | cell array of character vectors

Image file names, specified as a character vector or cell array of character vectors If specifying more than one file name, you must use a cell array of character vectors.

Data Types: `char` | `cell`

`imageArray` — Set of images
H-by-*W*-by-*C*-by-*N* array

Set of images, specified as an *H*-by-*W*-by-*C*-by-*N* array where:

- *H* – Height of the tallest image in the array
- *W* – Width of the widest image in the array
- *C* – Number of color channels
- *N* – Number of images in the array

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

`cameraIntrinsic` — Camera intrinsic parameters
`cameraIntrinsics` object

Camera intrinsic parameters, specified as a `cameraIntrinsics` object.

`checkerSize` — Size of checkerboard square
scalar

Size of a checkerboard square, specified as a scalar in millimeters. This value specifies the length of each side of a checkerboard square.

Data Types: `single` | `double`

Name-Value Pair Arguments

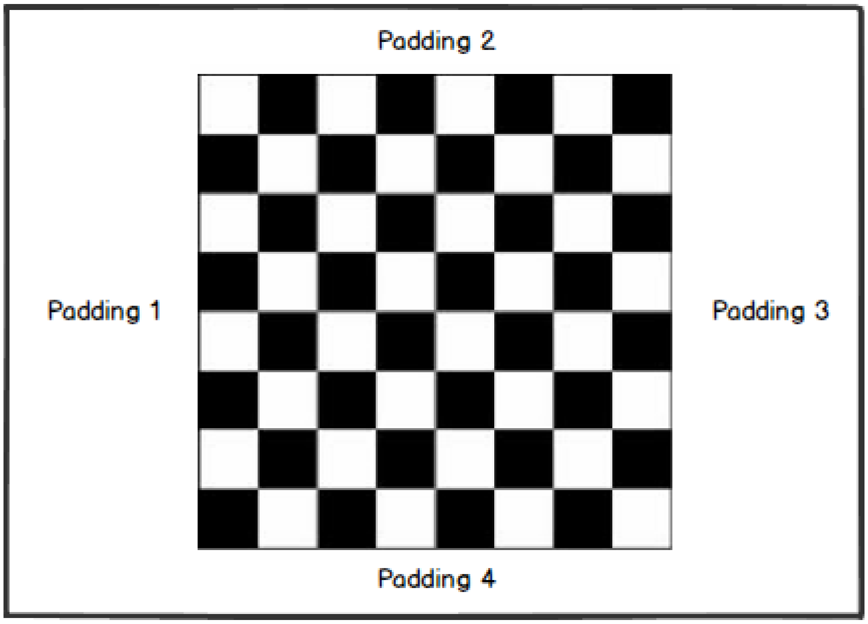
Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MinCornerMetric',0.2` sets the threshold for the checkerboard corner metric to 0.2.

`Padding` — Padding along each side of checkerboard
`[0 0 0 0]` (default) | four-element row vector

Padding along each side of checkerboard, specified as the comma-separated pair consisting of `'Padding'` and a four-element row vector of nonnegative values in millimeters.

The figure shows how the elements of the vector pad the sides.



Checkerboard Padding

Data Types: `single` | `double`

`MinCornerMetric` — Threshold for checkerboard corner metric
`0.15` (default) | nonnegative scalar

Threshold for the checkerboard corner metric, specified as the comma-separated pair consisting of `'MinCornerMetric'` and a nonnegative scalar. Using a higher threshold value can reduce the number of false detections in a noisy or highly textured image.

Data Types: `single` | `double`

`ShowProgressBar` — Display function progress
`false` (default) | `true`

Display function progress in a progress bar, specified as the comma-separated pair consisting of `'ShowProgressBar'` and a logical `false` or `true`.

Data Types: `logical`

Output Arguments

`imageCorners3d` — Estimated location of checkerboard corners
4-by-3 matrix | 4-by-3-by-*P* array

Estimated location of checkerboard corners, returned as a 4-by-3 matrix or 4-by-3-by-*P* array. For one image, the function returns the 3-D world frame coordinates of the four checkerboard corners. Each row represents the *x*-, *y*- , *z*-axis coordinates of a corner point in meters. For multiple images, the coordinates are returned as a 4-by-3-by-*P* array, where *P* is the number of images in which a checkerboard was detected.

`boardDimensions` — Checkerboard dimensions
two-element row vector

Checkerboard dimensions, returned as a two-element row vector. The elements represent the width and length of the checkerboard respectively, in millimeters. The dimensions of the checkerboard are expressed in terms of the number of squares. The function calculates the dimensions of the checkerboard by multiplying the size of the checkerboard squares, `checkerSize`, by the number of detected squares along a side.

`imagesUsed` — Pattern detection flag
N-by-1 logical array

Pattern detection flag, returned as an *N*-by-1 logical array. *N* is the number of images in the first input argument. A value of `1` (true) indicates that the function detected a checkerboard pattern in the corresponding image. A value of `0` (false) indicates that the function did not detect a checkerboard pattern in the corresponding image.

See Also

Functions
`detectRectangularPlanePoints` | `estimateLidarCameraTransform`

Topics
"Lidar and Camera Calibration"

Introduced in R2020b

detectRectangularPlanePoints

Detect rectangular plane of specified dimensions in point cloud

Syntax

```
ptCloudPlanes = detectRectangularPlanePoints(ptCloudIn,planeDimensions)
[ptCloudPlanes,ptCloudUsed] = detectRectangularPlanePoints(ptCloudArray,
planeDimensions)
___ = detectRectangularPlanePoints(ptCloudFileNames,planeDimensions)
[___,indicesCell] = detectRectangularPlanePoints(ptCloudFileNames,
planeDimensions)
[___] = detectRectangularPlanePoints(___,Name,Value)
```

Description

`ptCloudPlanes = detectRectangularPlanePoints(ptCloudIn,planeDimensions)` detects and extracts a rectangular plane, `ptCloudPlanes`, of specified dimensions, `planeDimensions`, from the input point cloud `ptCloudIn`.

`[ptCloudPlanes,ptCloudUsed] = detectRectangularPlanePoints(ptCloudArray, planeDimensions)` detects rectangular planes from a set of point clouds, `ptCloudArray`. In addition, the function can return is a logical vector, `ptCloudUsed`, that indicates the point clouds in which it detected a rectangular plane.

`___ = detectRectangularPlanePoints(ptCloudFileNames,planeDimensions)` detects rectangular planes from a set of point cloud files, `ptCloudFileNames`, and returns any combination of output arguments from previous syntaxes.

`[___,indicesCell] = detectRectangularPlanePoints(ptCloudFileNames, planeDimensions)` returns indices to the points within the detected rectangular plane in each point cloud, in addition to any previous combination of arguments.

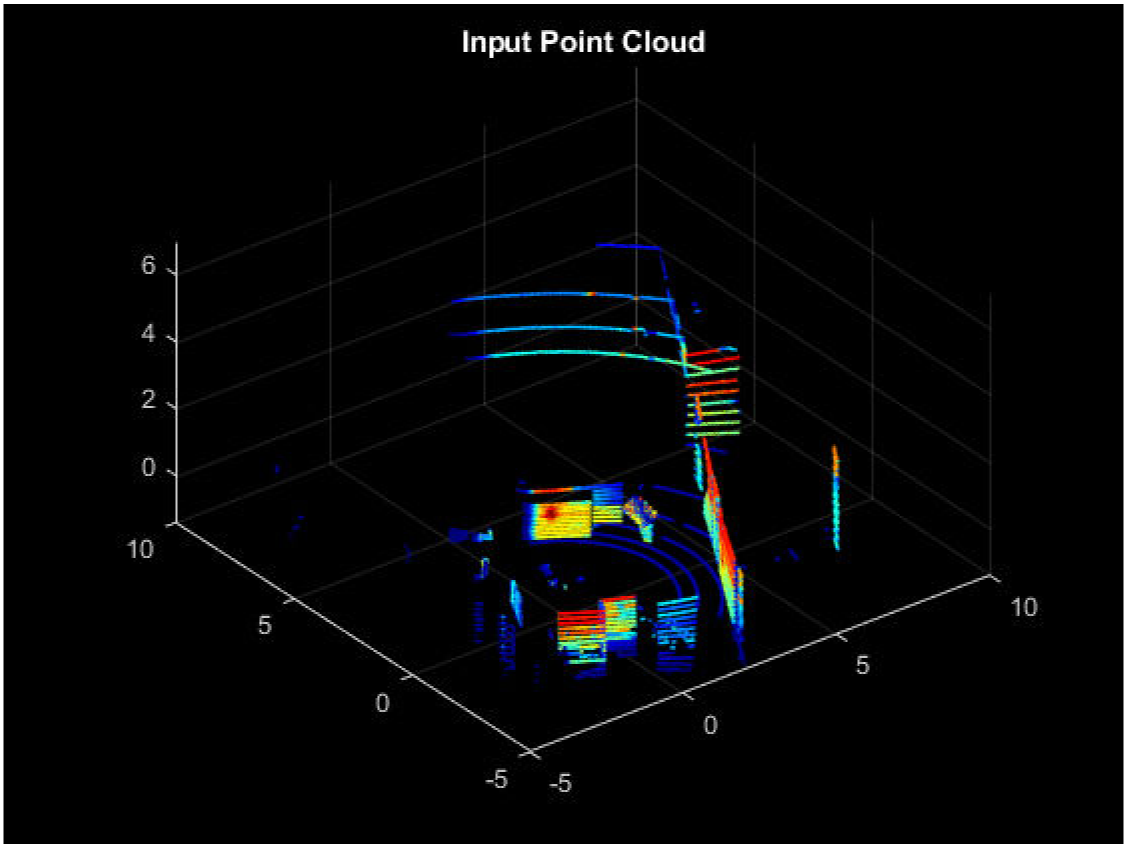
`[___] = detectRectangularPlanePoints(___,Name,Value)` specifies options using one or more name-value pair arguments. For example, `'RemoveGround',true` sets the `'RemoveGround'` flag to true, which removes the ground plane from the input point cloud before processing.

Examples

Detect Checkerboard Plane in Point Cloud

Load point cloud data into the workspace. Visualize the point cloud.

```
ptCloud = pcread('pcCheckerboard.pcd');
pcshow(ptCloud)
title('Input Point Cloud')
xlim([-5 10])
ylim([-5 10])
```

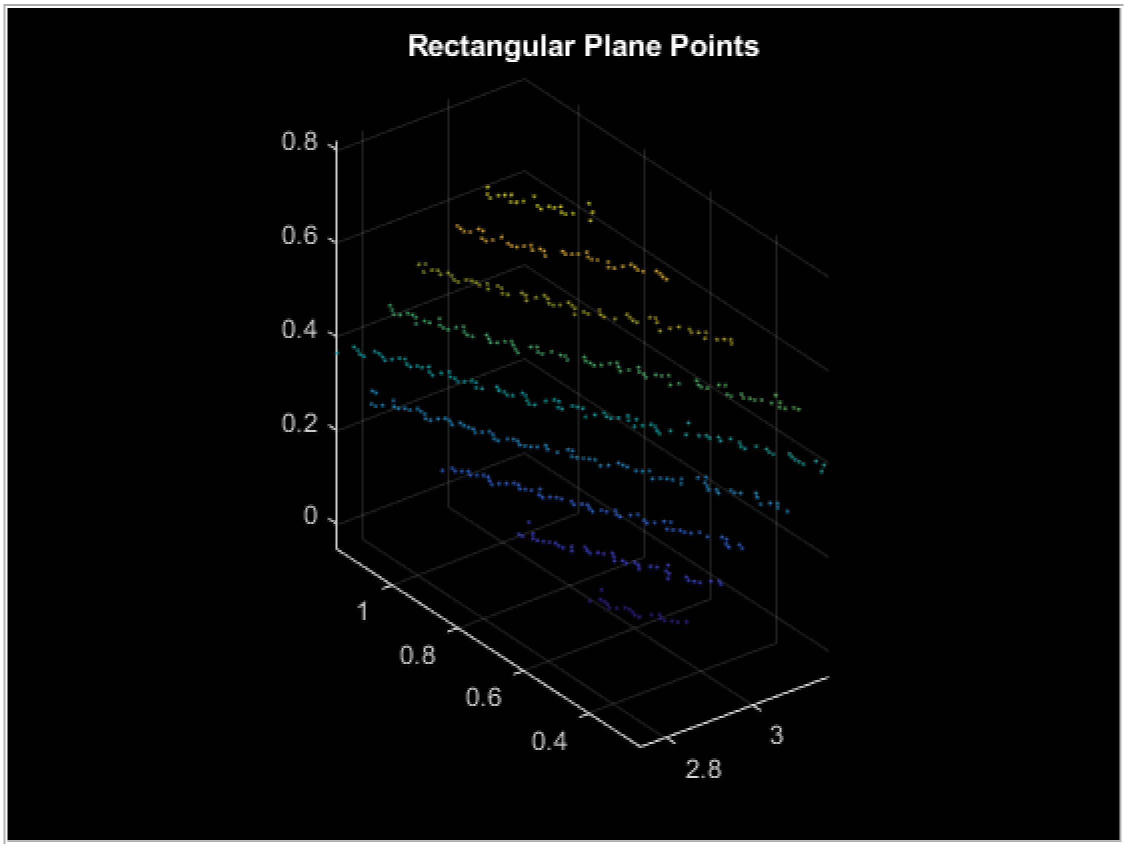


Set the search dimensions for the rectangular plane.

```
boardSize = [729 810];
```

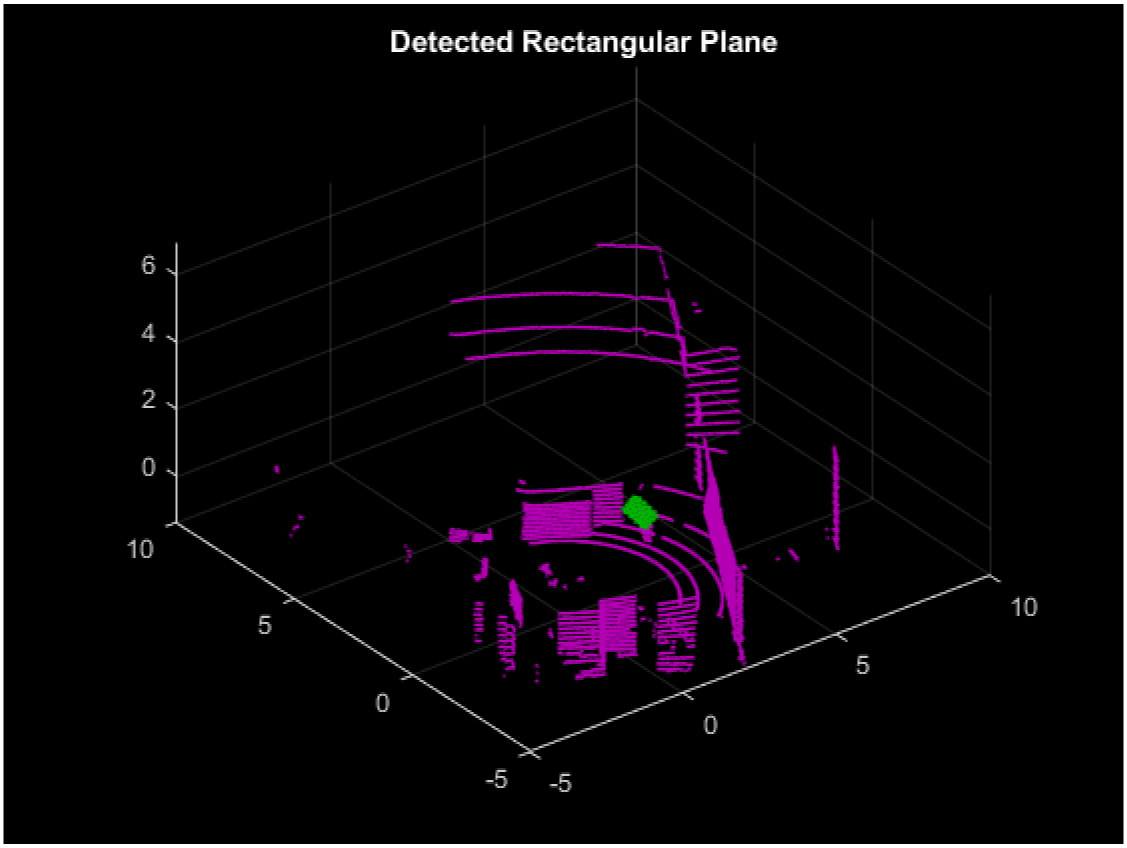
Search for the rectangular plane in the point cloud. Visualize the detected rectangular plane.

```
lidarCheckerboardPlane = detectRectangularPlanePoints(ptCloud,boardSize, ...
    'RemoveGround',true);
hRect = figure;
panel = uipanel('Parent',hRect,'BackgroundColor',[0 0 0]);
ax = axes('Parent',panel,'Color',[0 0 0]);
pcshow(lidarCheckerboardPlane,'Parent',ax)
title('Rectangular Plane Points')
```



Visualize the detected rectangular plane on the input point cloud.

```
figure
pcshowpair(ptCloud,lidarCheckerboardPlane)
title('Detected Rectangular Plane')
xlim([-5 10])
ylim([-5 10])
```



Input Arguments

ptCloudIn — Point cloud
pointCloud object

Point cloud, specified as a pointCloud object. The function searches within this point cloud for a rectangular plane.

ptCloudArray — Point cloud array
array of pointCloud objects

Point cloud array, specified as a *P*-by-1 array of pointCloud objects. *P* is the number of pointCloud objects in the array. The function searches within each point cloud for a rectangular plane.

ptCloudFileNames — Point cloud file names
character vector | cell array of character vectors

Point cloud file names, specified as a character vector or cell array of character vectors. If specifying multiple file names, you must use a cell array of character vectors.

Data Types: char | cell

planeDimensions — Rectangular plane dimensions
two-element vector

Rectangular plane dimensions, specified as a two-element vector of positive real numbers. The elements specify the width and length of the rectangular plane respectively, in millimeters. The function searches the input point cloud for a plane with the same dimensions as `planeDimensions`.

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: `'RemoveGround',true` sets the `'RemoveGround'` flag to true, which removes the ground plane from the input point cloud before processing.

`MinDistance` — Clustering threshold for two adjacent points

`0.5` (default) | positive scalar

Clustering threshold for two adjacent points, specified as the comma-separated pair consisting of `'MinDistance'` and a positive scalar in meters. The clustering process is based on the Euclidean distance between adjacent points. If the distance between two adjacent points is less than the clustering threshold, both points belong to the same cluster.

Data Types: `single` | `double`

`ROI` — Region of interest for detection

vector of form [*xmin*, *xmax*, *ymin*, *ymax*, *zmin*, *zmax*]

Region of interest (ROI) for detection, specified as the comma-separated pair consisting of `'ROI'` and a vector of the form [*xmin*, *xmax*, *ymin*, *ymax*, *zmin*, *zmax*]. The vector specifies the *x*, *y*, and *z* limits of the ROI as the pairs *xmin* and *xmax*, *ymin* and *ymax*, *zmin* and *zmax* respectively.

Data Types: `single` | `double`

`DimensionTolerance` — Tolerance for uncertainty in rectangular plane dimensions

`0.05` (default) | positive scalar in the range [0 1]

Tolerance for uncertainty in the rectangular plane dimensions, specified as the comma-separated pair consisting of `'DimensionTolerance'` and a positive scalar in the range [0 1]. A higher `'DimensionTolerance'` indicates a more tolerant range for the rectangular plane dimensions

Data Types: `single` | `double`

`RemoveGround` — Remove ground plane from point cloud

`false` (default) | `true`

Remove the ground plane from the point cloud, specified as the comma-separated pair consisting of `'RemoveGround'` and a logical `false` or `true`.

The normal of the plane is assumed to point towards the positive direction of the *z*-axis with the reference vector `[0 0 1]`.

Data Types: `logical`

`Verbose` — Display function progress

`false` (default) | `true`

Display function progress, specified as the comma-separated pair consisting of `'Verbose'` and a logical `false` or `true`.

Data Types: `logical`

Output Arguments

`ptCloudPlanes` — Detected rectangular planes
`pointCloud` object | 1-by-*P* array of `pointCloud` objects

Detected rectangular planes, returned as a `pointCloud` object or 1-by-*P* array of `pointCloud` objects, where *P* specifies the number of input point clouds in which a rectangular plane was detected.

`ptCloudUsed` — Pattern detection flag
1-by-*N* logical vector

Pattern detection flag, returned as a 1-by-*N* logical vector. *N* is the number of input point clouds. A `true` value indicates that the function detected a rectangular plane in the corresponding point cloud. A `false` value indicates that the function did not detect a rectangular plane.

`indicesCell` — Indices of detected rectangular planes
1-by-*P* cell array

Indices of detected rectangular planes, returned as a 1-by-*P* cell array, where *P* is the number of input point clouds in which a rectangular plane was detected. Each cell contains a logical vector that specifies the indices of the corresponding point cloud at which the function detected a rectangular plane. The indices can be used to extract the detected plane from the point cloud data.

See Also

Functions
`estimateCheckerboardCorners3d` | `estimateLidarCameraTransform` | `projectLidarPointsOnImage`

Topics
"Lidar and Camera Calibration"

Introduced in R2020b

estimateLidarCameraTransform

Estimate rigid transformation from lidar sensor to camera

Syntax

```
tform = estimateLidarCameraTransform(ptCloudPlanes,imageCorners3d)
[tform,errors] = estimateLidarCameraTransform( ___ )
[ ___ ] = estimateLidarCameraTransform( ___ ,Name,Value)
```

Description

`tform = estimateLidarCameraTransform(ptCloudPlanes,imageCorners3d)` estimates the transformation between a lidar sensor and a camera using the checkerboard calibration pattern features extracted from each sensor.

`[tform,errors] = estimateLidarCameraTransform( ___ )` returns the inaccuracy in estimating the transformation matrix `errors` using the input arguments from the previous syntax.

`[ ___ ] = estimateLidarCameraTransform( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to any combination of arguments in previous syntaxes. For example, `'Verbose',true` sets the function to display progress.

Examples

Estimate Rigid Transform from Lidar to Camera

Estimate the rigid transformation from a lidar sensor to a camera using data captured from the lidar sensor and camera calibration parameters. Use these three steps:

1. Load the data into the workspace.
2. Extract the required features from images and point cloud data.
3. Estimate the rigid transformation using the extracted features.

Load Data

Load images and point cloud data into the workspace.

```
imageDataPath = fullfile(toolboxdir('lidar'),'lidardata',...
    'lcc','vlp16','images');
imds = imageDatastore(imageDataPath);
imageFileNames = imds.Files;
ptCloudFilePath = fullfile(toolboxdir('lidar'),'lidardata',...
'lcc','vlp16','pointCloud');
pcds = fileDatastore(ptCloudFilePath,'ReadFcn',@pcread);
pcFileNames = pcds.Files;
```

Load camera calibration files into the workspace.

```
cameraIntrinsicFile = fullfile(imageDataPath,'calibration.mat');
intrinsic = load(cameraIntrinsicFile);
```

Feature Extraction

Specify the size of the checkerboard squares in millimeters.

```
squareSize = 81;
```

Estimate the checkerboard corner coordinates for the images.

```
[imageCorners3d,planeDimension,imagesUsed] = estimateCheckerboardCorners3d( ...
    imageFileNames,intrinsic.cameraParams,squareSize);
```

Filter the point clouds based on the images used.

```
pcFileNames = pcFileNames(imagesUsed);
```

Detect the checkerboard planes in the filtered point clouds using the plane parameters `planeDimension`.

```
[lidarCheckerboardPlanes,framesUsed] = detectRectangularPlanePoints( ...
pcFileNames,planeDimension,'RemoveGround',true);
```

Extract the images, checkerboard corners, and point clouds in which you detected features.

```
imageFileNames = imageFileNames(framesUsed);
pcFileNames = pcFileNames(framesUsed);
imageCorners3d = imageCorners3d(:,:,framesUsed);
```

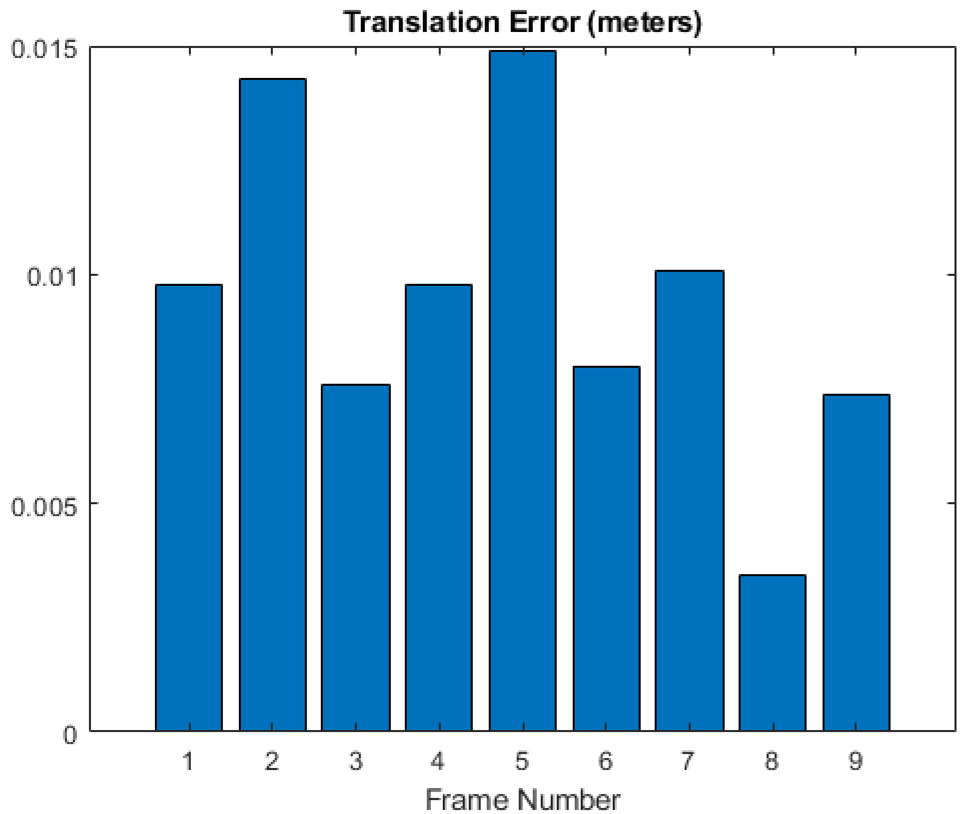
Estimate Transformation

Estimate the transformation using checkerboard planes from the point clouds and 3-D checkerboard corner points from the images.

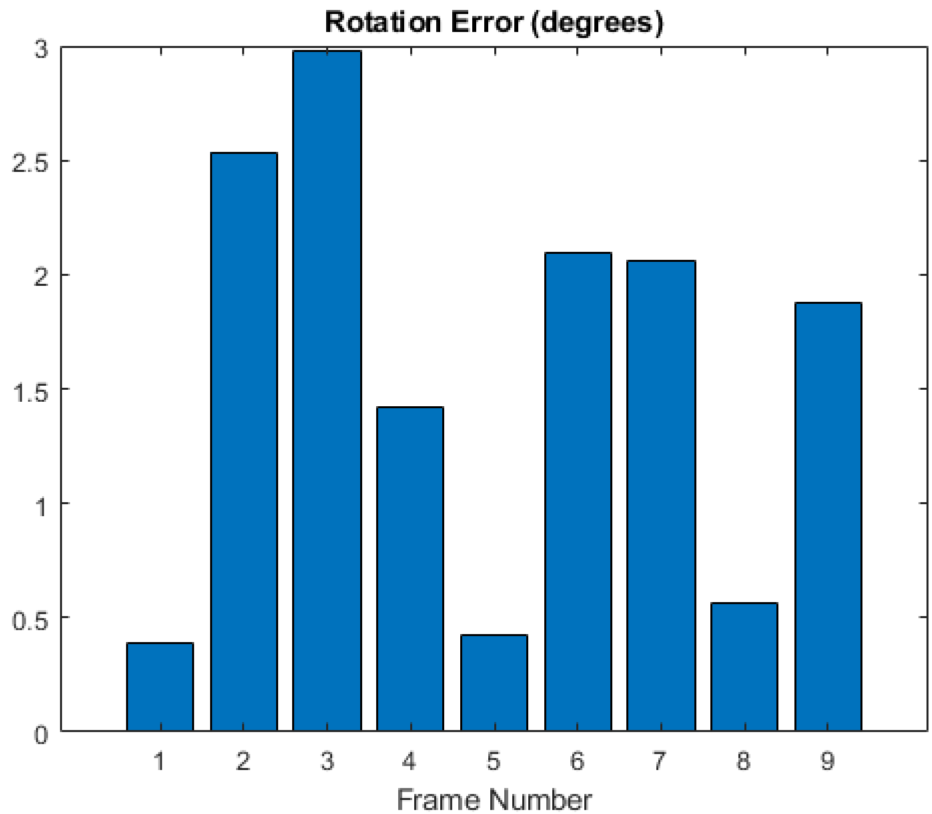
```
[tform,errors] = estimateLidarCameraTransform(lidarCheckerboardPlanes, ...
imageCorners3d,'CameraIntrinsic',intrinsic.cameraParams);
```

Display translation, rotation, and reprojection errors as bar graphs.

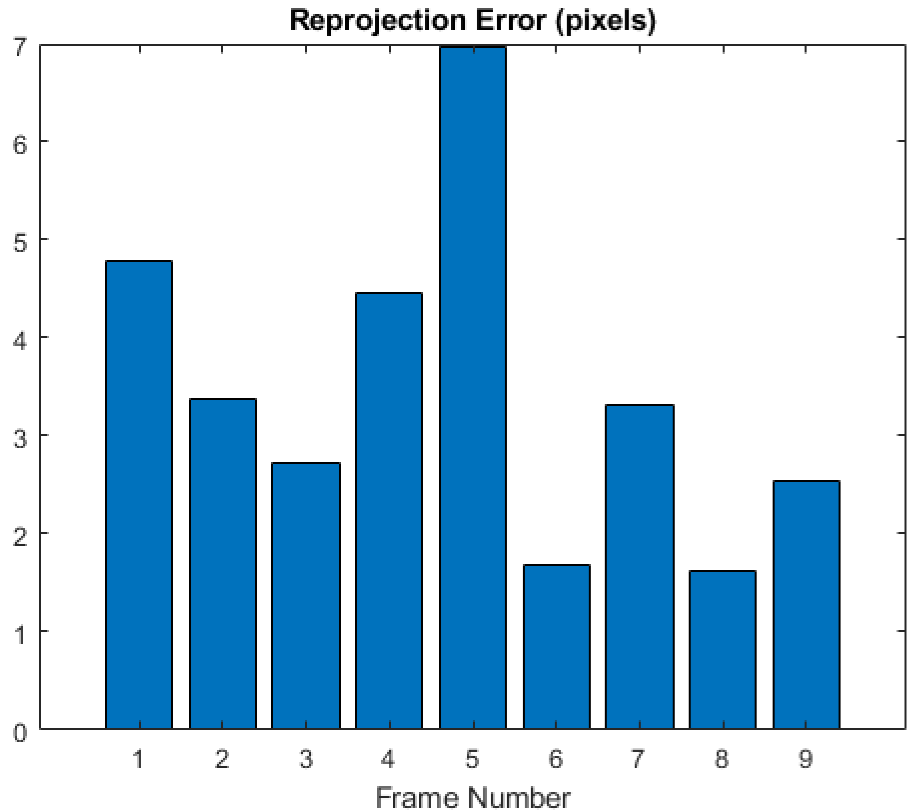
```
figure
bar(errors.TranslationError)
xlabel('Frame Number')
title('Translation Error (meters)')
```




```
figure
bar(errors.RotationError)
xlabel('Frame Number')
title('Rotation Error (degrees)')
```




```
figure
bar(errors.ReprojectionError)
xlabel('Frame Number')
title('Reprojection Error (pixels)')
```



Input Arguments

ptCloudPlanes — Segmented checkerboard planes
P-by-1 array of pointCloud objects

Segmented checkerboard planes, specified as a pointCloud object or *P*-by-1 array of pointCloud objects. *P* is the number of point clouds. Each pointCloud object must contain points that represent a checkerboard (rectangular) plane.

P must be equal for both the ptCloudPlanes and imageCorners3d arguments. This means that number of point clouds and number of images used for detection must also be equal.

imageCorners3d — 3-D coordinates of checkerboard corners
4-by-3-by-*P* array

3-D coordinates of the checkerboard corners, specified as a 4-by-3 matrix or 4-by-3-by-*P* array. *P* represents the number of camera images used for detection. Each row of a channel contains the 3-D coordinates, in the form of [*x*,*y*,*z*], of a checkerboard corner in meters extracted from the corresponding camera image. *P* must be equal for both the ptCloudPlanes and imageCorners3d arguments. This means that number of point clouds and number of images used for detection must also be equal.

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: `'Verbose',true` sets the function to display progress.

`Lidar3DCorners` — Checkerboard corners in lidar frame
4-by-3-by-*P* array

Checkerboard corners in the lidar frame, specified as the comma-separated pair consisting of `'Lidar3DCorners'` and a 4-by-3-by-*P* array where *P* is the number of point clouds.

If the user specifies the checkerboard corners in the lidar frame, then the function does not calculate them internally.

Data Types: `single` | `double`

`InitialTransform` — Initial rigid transformation
identity transformation as a `rigid3d` object (default) | `rigid3d` object

Initial rigid transformation, specified as the comma-separated pair consisting of `'InitialTransformation'` and a `rigid3d` object.

The function assumes the rotation angle between the lidar sensor and the camera is in the range [-45 45] along each axis. For any other range of the rotation angle, use this name-value pair to specify an initial transformation to improve function accuracy.

`CameraIntrinsic` — Camera intrinsic parameters
`cameraIntrinsics` object | `cameraParameters` object

Camera intrinsic parameters, specified as the comma-separated pair consisting of `'CameraIntrinsic'` and a `cameraIntrinsics` object or `cameraParameters` object.

`Verbose` — Display function progress
`false` or `0` (default) | `true` or `1`

Display function progress, specified as the comma-separated pair consisting of `'Verbose'` and a logical `0` (`false`) or logical `1` (`true`).

Data Types: `logical`

Output Arguments

`tform` — Lidar to camera rigid transformation
`rigid3d` object

Lidar to camera rigid transformation, returned as a `rigid3d` object. The returned object registers the point cloud data from a lidar sensor to the coordinate frame of a camera.

`errors` — Inaccuracy of the transformation matrix estimation
structure

Inaccuracy of the transformation matrix estimation, returned as a structure. The structure contains these fields.

- `RotationError` — The difference between the normal angles defined by the checkerboard planes in the point clouds (lidar frame) and those in the images (camera frame). The function estimates the plane in the image using the checkerboard corner coordinates. The function returns the error values in degrees, as a *P*-element numeric array. *P* is the number of point clouds.
- `TranslationError` — The difference between the centroid coordinates of checkerboard planes in the point clouds and those in the images. The function returns the error values in meters, as a *P*-element numeric array. *P* is the number of point clouds.

If you specify camera intrinsic parameters to the function using `'CameraIntrinsic'` name-value pair, then the structure contains this additional field.

- `ReprojectionError` — The difference between the projected (transformed) centroid coordinates of the checkerboard planes from the point clouds and those in the images. The function returns the error values in pixels, as a *P*-element numeric array. *P* is the number of point clouds.

Data Types: `struct`

See Also

Functions
`bboxCameraToLidar` | `detectRectangularPlanePoints` | `estimateCheckerboardCorners3d` | `fuseCameraToLidar` | `projectLidarPointsOnImage`

Topics
"Lidar and Camera Calibration"

Introduced in R2020b

projectLidarPointsOnImage

Project lidar point cloud data onto image coordinate frame

Syntax

```
imPts = projectLidarPointsOnImage(ptCloudIn,intrinsics,tform)
imPts = projectLidarPointsOnImage(worldPoints,intrinsics,tform)
[imPts,indices] = projectLidarPointsOnImage( ___ )
[ ___ ] = projectLidarPointsOnImage( ___ ,Name,Value)
```

Description

`imPts = projectLidarPointsOnImage(ptCloudIn,intrinsics,tform)` projects lidar point cloud data onto an image coordinate frame using a rigid transformation between the lidar sensor and camera, `tform`, and a set of camera intrinsic parameters, `intrinsics`. The output `imPts` contains the 2-D coordinates of the projected points in the image frame.

`imPts = projectLidarPointsOnImage(worldPoints,intrinsics,tform)` projects lidar points, specified as 3-D coordinates in the world frame, onto image coordinate frame.

`[imPts,indices] = projectLidarPointsOnImage( ___ )` returns the linear indices of the projected points in the point cloud using any combination of input arguments in previous syntaxes.

`[ ___ ] = projectLidarPointsOnImage( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to any combination of arguments in previous syntaxes. For example, `'ImageSize',[250 400]` sets the size of the image on which to project the points to 250-by-400 pixels.

Examples

Overlay Projected Lidar Points on Image

Load ground truth data from a MAT-file into the workspace. Extract the image and point cloud data from the ground truth data.

```
dataPath = fullfile(toolboxdir('lidar'),'lidardata','lcc','sampleColoredPtCloud.mat');
gt  = load(dataPath);
img = gt.im;
pc = gt.ptCloud;
```

Extract the camera intrinsic parameters from the ground truth data.

```
intrinsics = gt.camParams;
```

Extract the camera to lidar transformation matrix from the ground truth data, and invert to find the lidar to camera transformation matrix.

```
tform = invert(gt.tform);
```

Downsample the point cloud data.

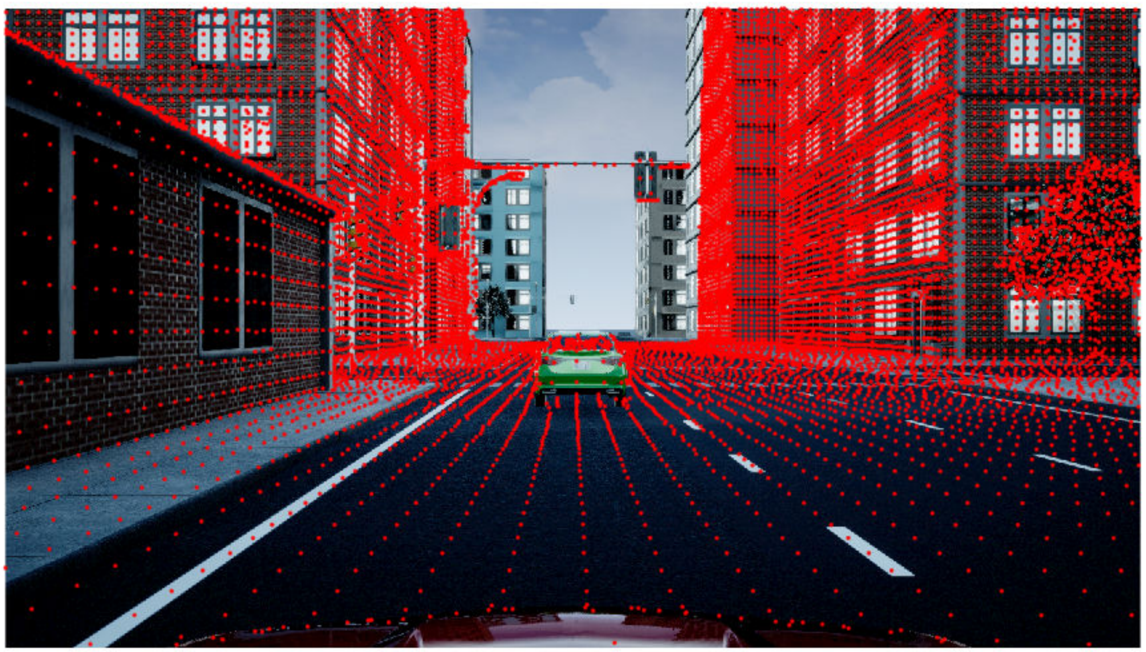
```
p1 = pcdownsample(pc,'gridAverage',0.5);
```

Project the point cloud onto the image frame.

```
imPts = projectLidarPointsOnImage(p1,intrinsics,tform);
```

Overlay the projected points on the image.

```
figure
imshow(img)
hold on
plot(imPts(:,1),imPts(:,2),'.','Color','r')
hold off
```



Input Arguments

ptCloudIn — Point cloud
pointCloud object

Point cloud, specified as a pointCloud object.

worldPoints — Points in world coordinate frame
M-by-3 matrix | *M*-by-*N*-by-3 array

Points in the world coordinate frame, specified as an *M*-by-3 matrix or *M*-by-*N*-by-3 array. If you specify an *M*-by-3 matrix, each row contains 3-D world coordinates of a point in an unorganized point cloud that contains *M* points in total. If you specify an *M*-by-*N*-by-3 array, *M* and *N* represent the number of rows and columns, respectively, in an organized point cloud. Each channel of the array contains the 3-D world coordinates of that point.

Data Types: `single` | `double`

`intrinsics` — Camera intrinsic parameters
`cameraIntrinsics` object

Camera intrinsic parameters, specified as a `cameraIntrinsics` object.

`tform` — Lidar to camera rigid transformation
`rigid3d` object

Lidar to camera rigid transformation, specified as a `rigid3d` 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: `'ImageSize',[250 400]` sets the size of the image on which to project the points to 250-by-400 pixels.

`Indices` — Indices selected for projection onto image coordinate frame
vector of positive integers

Indices selected for projection onto image coordinate frame, specified as the comma-separated pair consisting of `'Indices'` and a vector of positive integers.

Data Types: `single` | `double`

`ImageSize` — Size of image on which points are projected
[`ceil`(2C_x) `ceil`(2C_y)] (default) | two-element row vector

Size of the image on which the points are projected, specified as the comma-separated pair consisting of `'ImageSize'` and a two-element row vector of the form [*width height*] in pixels. The function uses the specified dimensions to filter out the projected points that are not in the field of view of the camera.

If you do no specify the`'ImageSize'` argument, then the function uses the optical center coordinates from the camera intrinsic parameters `intrinsics` to estimate the field of view of the camera. The optical center coordinates are of the form [C_x C_y] where C_x and C_y represent the *x*- and *y*-axis coordinates of the optical center, respectively, in pixels.

Note If you specify an `'ImageSize'` argument greater than the default argument, then the function uses the default argument.

Data Types: `single` | `double`

Output Arguments

`imPts` — Points projected on image
M-by-2 matrix

Points projected on image, returned as an *M*-by-2 matrix. Each row contains the 2-D coordinates, in the form [*x y*], a point in the image frame.

Data Types: `single` | `double`

`indices` — Linear indices of projected points
vector of positive integers

Linear indices of the projected points of the point cloud, returned as a vector of positive integers.

Data Types: `single` | `double`

See Also

Functions
`bboxCameraToLidar` | `detectRectangularPlanePoints` | `estimateCheckerboardCorners3d` | `estimateLidarCameraTransform` | `fuseCameraToLidar`

Topics
"Lidar and Camera Calibration"

Introduced in R2020b

fuseCameraToLidar

Fuse image information to lidar point cloud

Syntax

```
ptCloudOut = fuseCameraToLidar(I,ptCloudIn,intrinsics)
ptCloudOut = fuseCameraToLidar(I,ptCloudIn,intrinsics,tform)
ptCloudOut = fuseCameraToLidar( ___ ,nonoverlapcolor)
[ptCloudOut,colormap] = fuseCameraToLidar( ___ )
[ ___ ,indices] = fuseCameraToLidar( ___ )
```

Description

`ptCloudOut = fuseCameraToLidar(I,ptCloudIn,intrinsics)` fuses information from an image, `I`, to a specified point cloud, `ptCloudIn`, using the camera intrinsic parameters, `intrinsics`.

The function crops the fused point cloud, `ptCloudOut`, so that it contains only the points present in the field of view of the camera.

`ptCloudOut = fuseCameraToLidar(I,ptCloudIn,intrinsics,tform)` uses the camera to lidar rigid transformation `tform` to bring the point cloud into image frame before fusing it to the image information. Use this syntax when the point cloud data is not in the camera coordinate frame.

`ptCloudOut = fuseCameraToLidar( ___ ,nonoverlapcolor)` returns a fused point cloud of the same size as the input point cloud. The function uses the specified color `nonoverlapcolor` for points that are outside the field of view of the camera in addition to any combination of input arguments from previous syntaxes.

`[ptCloudOut,colormap] = fuseCameraToLidar( ___ )` returns the colors of the points `colormap` of the fused point cloud.

`[ ___ ,indices] = fuseCameraToLidar( ___ )` returns linear indices of the points in the fused point cloud that are in the field of view of the camera in addition to any combination of arguments from previous syntaxes.

Examples

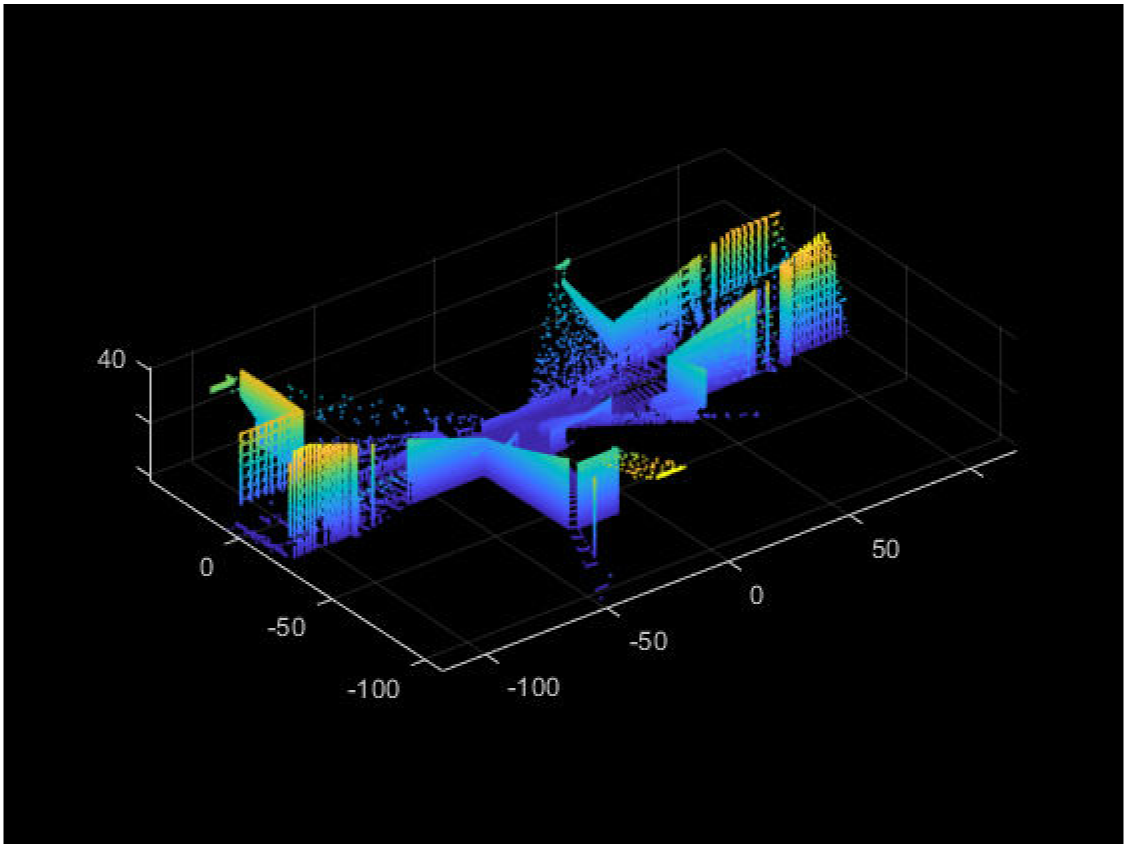
Fuse Color Information from Camera to Lidar

Load a MAT-file containing ground truth data into the workspace. Extract the image and point cloud from data.

```
dataPath = fullfile(toolboxdir('lidar'),'lidardata','lcc','sampleColoredPtCloud.mat');
gt = load(dataPath);
im = gt.im;
ptCloud = gt.ptCloud;
```

Plot the extracted point cloud.

```
pcshow(ptCloud)
title('Original Point Cloud')
```



Extract the lidar to camera transformation matrix and camera intrinsic parameters from the ground truth data.

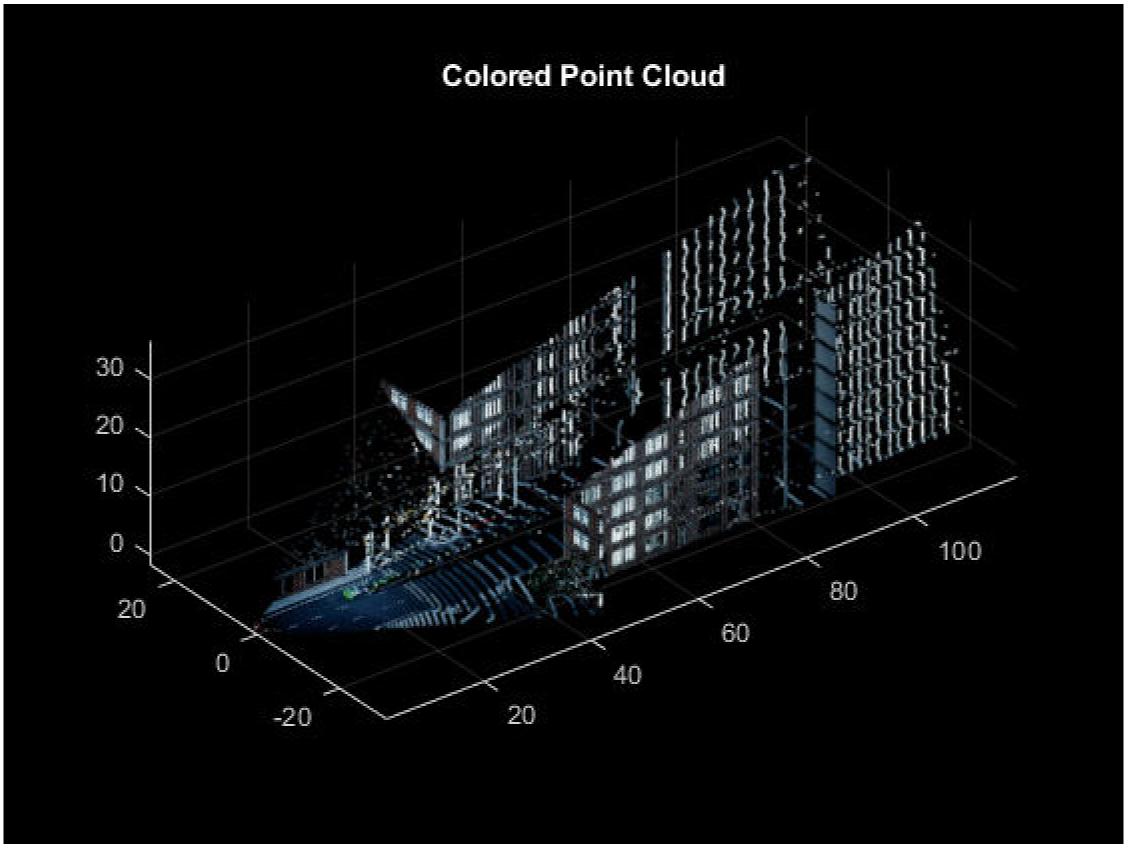
```
intrinsics = gt.camParams;
camToLidar = gt.tform;
```

Fuse the image to the point cloud.

```
ptCloudOut = fuseCameraToLidar(im,ptCloud,intrinsics,camToLidar);
```

Visualize the fused point cloud.

```
pcshow(ptCloudOut)
title('Colored Point Cloud')
```



Input Arguments

I — Color or grayscale image
H-by-*W*-by-*C* array

Color or grayscale image, specified as an *H*-by-*W*-by-*C* array.

- *H* — This specifies the height of the image.
- *W* — This specifies the width of the image.
- *C* — This specifies the number of color channels in the image. The function supports up to three color channels in an image.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

`ptCloudIn` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

`intrinsics` — Camera intrinsic parameters
`cameraIntrinsics` object

Camera intrinsic parameters, specified as a `cameraIntrinsics` object.

tform — Camera to lidar rigid transformation
rigid3d object

Camera to lidar rigid transformation, specified as a rigid3d object.

nonoverlapcolor — Color specification for points outside camera field of view
color name | short color name | RGB Triplet

Color specification for points outside the camera field of view, specified as a color name, short color name, or RGB triplet.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7]. Alternatively, you can specify some common colors by name. This table lists the named color options and the equivalent RGB triplet values.

Color Name	Color Short Name	RGB Triplet	Appearance
'red'	'r'	[1 0 0]	
'green'	'g'	[0 1 0]	
'blue'	'b'	[0 0 1]	
'cyan'	'c'	[0 1 1]	
'magenta'	'm'	[1 0 1]	
'yellow'	'y'	[1 1 0]	
'black'	'k'	[0 0 0]	
'white'	'w'	[1 1 1]	

Data Types: single | double | char

Output Arguments

ptCloudOut — Fused point cloud
pointCloud object

Fused point cloud, returned as a pointCloud object.

colormap — Point cloud color map
M-by-3 matrix of real values in the range [0, 1] | *M*-by-*N*-by-3 array of real values in the range [0, 1]

Point cloud color map, returned as one of these options:

- *M*-by-3 matrix — For unorganized point clouds
- *M*-by-*N*-by-3 array — For organized point clouds

Each row of the matrix or channel of the array contains the RGB triplet for the corresponding point in the point cloud. The function returns them as real values in the range [0, 1]. If you do not specify a nonoverlapcolor argument, then the color value for points outside the field of view of the camera is [0 0 0] (black).

Data Types: uint8

indices — Linear indices of fused point cloud points in camera field of view
vector of positive integers

Linear indices of the fused point cloud points in the camera field of view, returned as a vector of positive integers.

Data Types: single | double

See Also

Functions
bboxCameraToLidar | detectRectangularPlanePoints | estimateCheckerboardCorners3d | estimateLidarCameraTransform | projectLidarPointsOnImage

Topics
"Lidar and Camera Calibration"

Introduced in R2020b

bboxCameraToLidar

Estimate 3-D bounding boxes in point cloud from 2-D bounding boxes in image

Syntax

```
bboxesLidar = bboxCameraToLidar(bboxesCamera,ptCloudIn,intrinsics,tform)
[bboxesLidar,indices] = bboxCameraToLidar( ___ )
[bboxesLidar,indices,boxesUsed] = bboxCameraToLidar( ___ )
[ ___ ] = bboxCameraToLidar( ___ ,Name,Value)
```

Description

`bboxesLidar = bboxCameraToLidar(bboxesCamera,ptCloudIn,intrinsics,tform)` estimates 3-D bounding boxes in a point cloud frame, `ptCloudIn`, from 2-D bounding boxes in an image, `bboxesCamera`. The function uses camera intrinsic parameters, `intrinsics`, and a camera to lidar transformation matrix, `tform`, to estimate the 3-D bounding boxes, `bboxesLidar`.

`[bboxesLidar,indices] = bboxCameraToLidar( ___ )` returns the indices of the point cloud points that are inside the 3-D bounding boxes using the input arguments from the previous syntax.

`[bboxesLidar,indices,boxesUsed] = bboxCameraToLidar( ___ )` indicates for which of the specified 2-D bounding boxes the function detected a corresponding 3-D bounding box in the point cloud.

`[ ___ ] = bboxCameraToLidar( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to any of the argument combinations in previous syntaxes. For example, `'ClusterThreshold',0.5` sets the Euclidean distance threshold for differentiating point cloud clusters to 0.5 world units.

Examples

Transfer Bounding Box from Image to Point Cloud

Load ground truth data from a MAT-file into the workspace. Extract the image, point cloud data, and camera intrinsic parameters from the ground truth data.

```
dataPath = fullfile(toolboxdir('lidar'),'lidardata','lcc','bboxGT.mat');
gt = load(dataPath);
im = gt.im;
pc = gt.pc;
intrinsics = gt.cameraParams;
```

Extract the camera to lidar transformation matrix from the ground truth data.

```
tform = gt.camToLidar;
```

Extract the 2-D bounding box information.

```
bboxImage = gt.box;
```

Display the 2-D bounding box overlaid on the image.

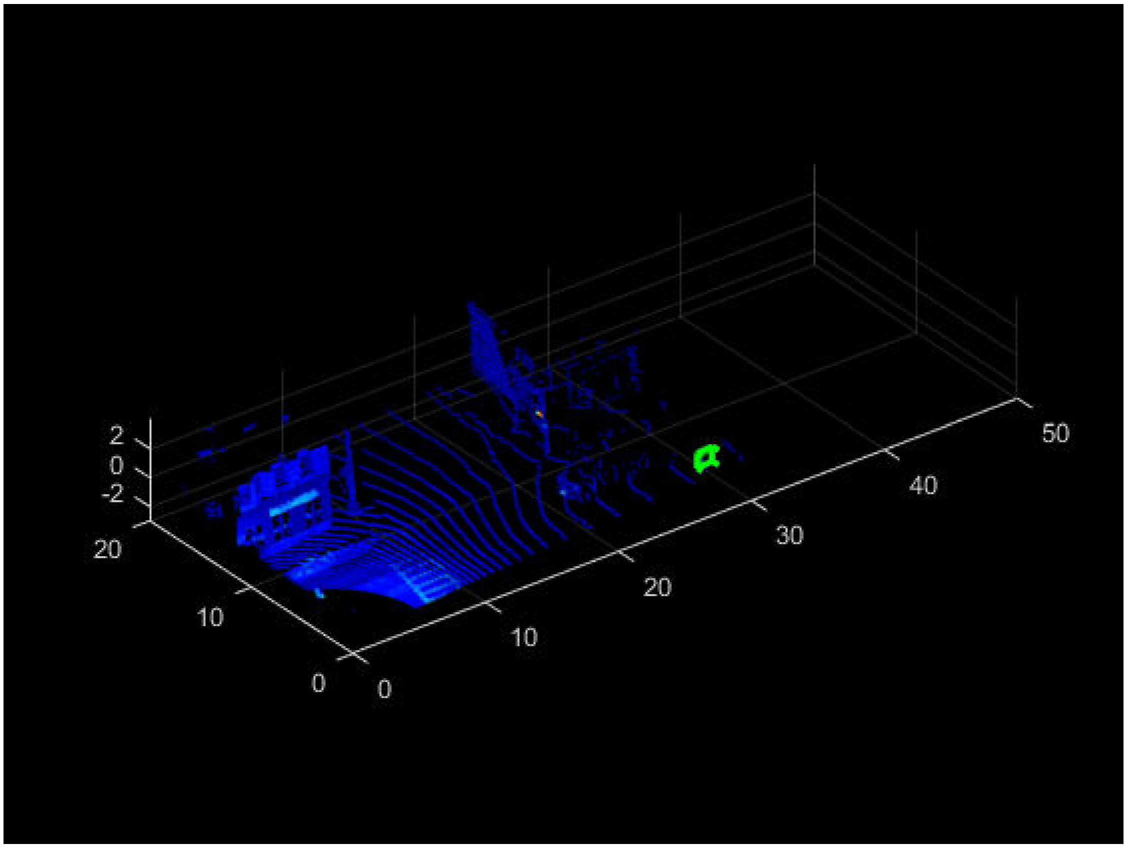
```
annotatedImage = insertObjectAnnotation(im,'Rectangle',bboxImage,'Vehicle');
figure
imshow(annotatedImage)
```

Estimate the bounding box in the point cloud.

```
[bboxLidar,indices] = ...
bboxCameraToLidar(bboxImage,pc,intrinsics,tform,'ClusterThreshold',1);
```

Display the 3-D bounding box overlaid on the point cloud.

```
figure
pcshow(pc)
xlim([0 50])
ylim([0 20])
showShape('cuboid',bboxLidar,'Opacity',0.5,'Color','green')
```



Input Arguments

bboxesCamera — 2-D bounding boxes in camera frame
M-by-4 matrix of real values

2-D bounding boxes in the camera frame, specified as an *M*-by-4 matrix of real values. Each row of the matrix contains the location and size of a rectangular bounding box in the form [*x y width height*]. The *x* and *y* elements specify the *x* and *y* coordinates, respectively, for the upper-left corner of the rectangle. The *width* and *height* elements specify the size of the rectangle. *M* is the number of bounding boxes.

Note The function assumes that the image data that corresponds to the 2-D bounding boxes and the point cloud data are time synchronized.

Data Types: `single` | `double`

ptCloudIn — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

Note The function assumes that the point cloud is in the vehicle coordinate system, where the *x*-axis points forward from the ego vehicle.

`intrinsics` — Camera intrinsic parameters
`cameraIntrinsics` object

Camera intrinsic parameters, specified as a `cameraIntrinsics` object.

`tform` — Camera to lidar rigid transformation
`rigid3d` object

Camera to lidar rigid transformation, specified as a `rigid3d` 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: `'ClusterThreshold',0.5` sets the Euclidean distance threshold for differentiating point cloud clusters to 0.5 world units.

`ClusterThreshold` — Clustering threshold for two adjacent points
`1` (default) | positive scalar

Clustering threshold for two adjacent points, specified as the comma-separated pair consisting of `'ClusterThreshold'` and a positive scalar. The clustering process is based on the Euclidean distance between two adjacent points. If the distance between two adjacent points is less than the specified clustering threshold, then the points belong to the same cluster. If the function returns a 3-D bounding box that is smaller than expected, try specifying a higher `'ClusterThreshold'` value.

Data Types: `single` | `double`

`MaxDetectionRange` — Range of detection from lidar sensor
[`1e–6 Inf`] (default) | two-element vector of real values in the range (`0, Inf`]

Range of detection from lidar sensor, specified as the comma-separated pair consisting of `'MaxDetectionRange'` and a real values in the range (`0, Inf`]. The first element of the vector specifies the shortest distance from the sensor at which to search for bounding boxes, and the second element specifies the distance at which the function stops searching. The value of Inf indicates the outermost points of the point cloud.

The first element must be smaller than the second element. Specify both in world units.

Data Types: `single` | `double`

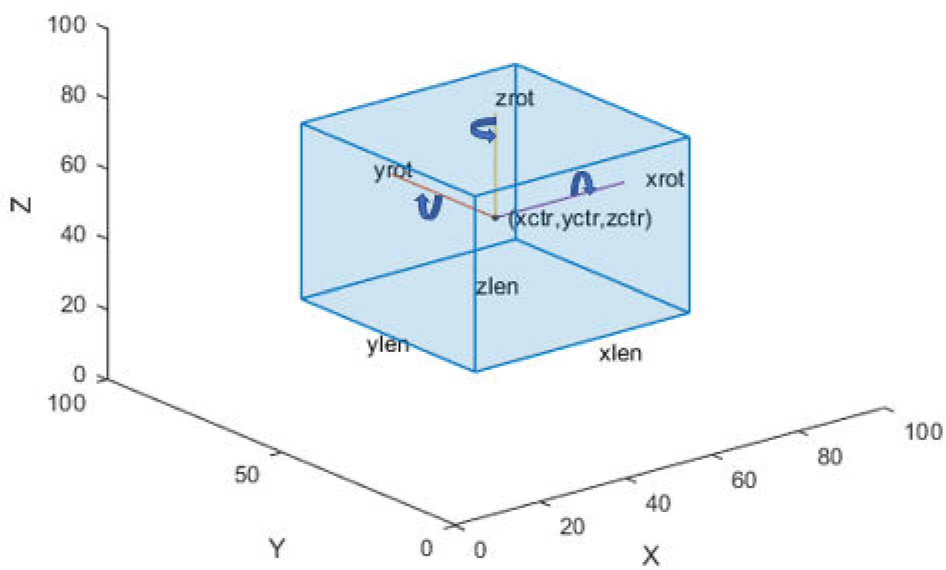
Output Arguments

`bboxesLidar` — 3-D bounding boxes in lidar frame
N-by-9 matrix of real values

3-D bounding boxes in the lidar frame, returned as an *N*-by-9 matrix of real values. *N* is the number of detected 3-D bounding boxes. Each row of the matrix has the form [x_{ctr} y_{ctr} z_{ctr} x_{len} y_{len} z_{len} x_{rot} y_{rot} z_{rot}].

- x_{ctr}, y_{ctr}, and z_{ctr} — These values specify the *x*-, *y*-, and *z*-axis coordinates, respectively, of the center of the cuboid bounding box.
- x_{len}, y_{len}, and z_{len} — These values specify the length of the cuboid along the *x*-, *y*-, and *z*-axis, respectively, before it is rotated.
- x_{rot}, y_{rot}, and z_{rot} — These values specify the rotation angles of the cuboid around the *x*-, *y*-, and *z*-axis, respectively. These angles are clockwise-positive when looking in the forward direction of their corresponding axes.

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



Data Types: `single` | `double`

indices — Indices of points inside 3-D bounding boxes
column vector | *N*-element cell array

Indices of the points inside the 3-D bounding boxes, returned as a column vector or an *N*-element cell array.

If the function detects only one 3-D bounding box in the point cloud, it returns a column vector. Each element of the vector is the point cloud index of a point detected in the 3-D bounding box.

If the function detects multiple 3-D bounding boxes, it returns an *N*-element cell array. *N* is the number of 3-D bounding boxes detected in the point cloud, and each element of the cell array contains the point cloud indices of the points detected in the corresponding 3-D bounding box.

Data Types: `single` | `double`

boxesUsed — Pattern detection flag
M-element row vector of logicals

Pattern detection flag, returned as an *M*-element row vector of logicals. *M* is the number of input 2-D bounding boxes. If the function detects a corresponding 3-D bounding box in the point cloud, then it returns a value of `true` for that input 2-D bounding box. If the function does not detect a corresponding 3-D bounding box, then it returns a value of `false`.

Data Types: `logical`

See Also

Functions
`detectRectangularPlanePoints` | `estimateCheckerboardCorners3d` | `estimateLidarCameraTransform` | `fuseCameraToLidar` | `projectLidarPointsOnImage`

Topics
"Lidar and Camera Calibration"

Introduced in R2020b

pcmatchfeatures

Find matching features between point clouds

Syntax

```
indexPairs = pcmatchfeatures(features1,features2)
indexPairs = pcmatchfeatures(features1,features2,ptCloud1,ptCloud2)
[indexPairs,scores] = pcmatchfeatures(___)
___ = pcmatchfeatures(___,Name,Value)
```

Description

`indexPairs = pcmatchfeatures(features1,features2)` finds matching features between the input matrices of extracted point cloud features and returns their indices within each feature matrix.

`indexPairs = pcmatchfeatures(features1,features2,ptCloud1,ptCloud2)` rejects ambiguous feature matches based on spatial relation information from the point clouds corresponding to the feature matrices.

`[indexPairs,scores] = pcmatchfeatures(___)` returns the normalized Euclidean distances between the matching features using any combination of input arguments from previous syntaxes.

`___ = pcmatchfeatures(___,Name,Value)` specifies options using one or more name-value pair arguments in addition to any combination of arguments in previous syntaxes. For example, `'MatchThreshold',0.03` sets the normalized distance threshold for matching features to `0.03`.

Examples

Match and Visualize Corresponding Features in Point Clouds

This example shows how to match corresponding point cloud features and visualize them using the `pcmatchfeatures` and `pcshowMatchedFeatures` functions.

Construct a `velodyneFileReader` object.

```
veloReader = velodyneFileReader('lidarData_ConstructionRoad.pcap','HDL32E');
```

Read two point clouds using the `velodyneFileReader` object.

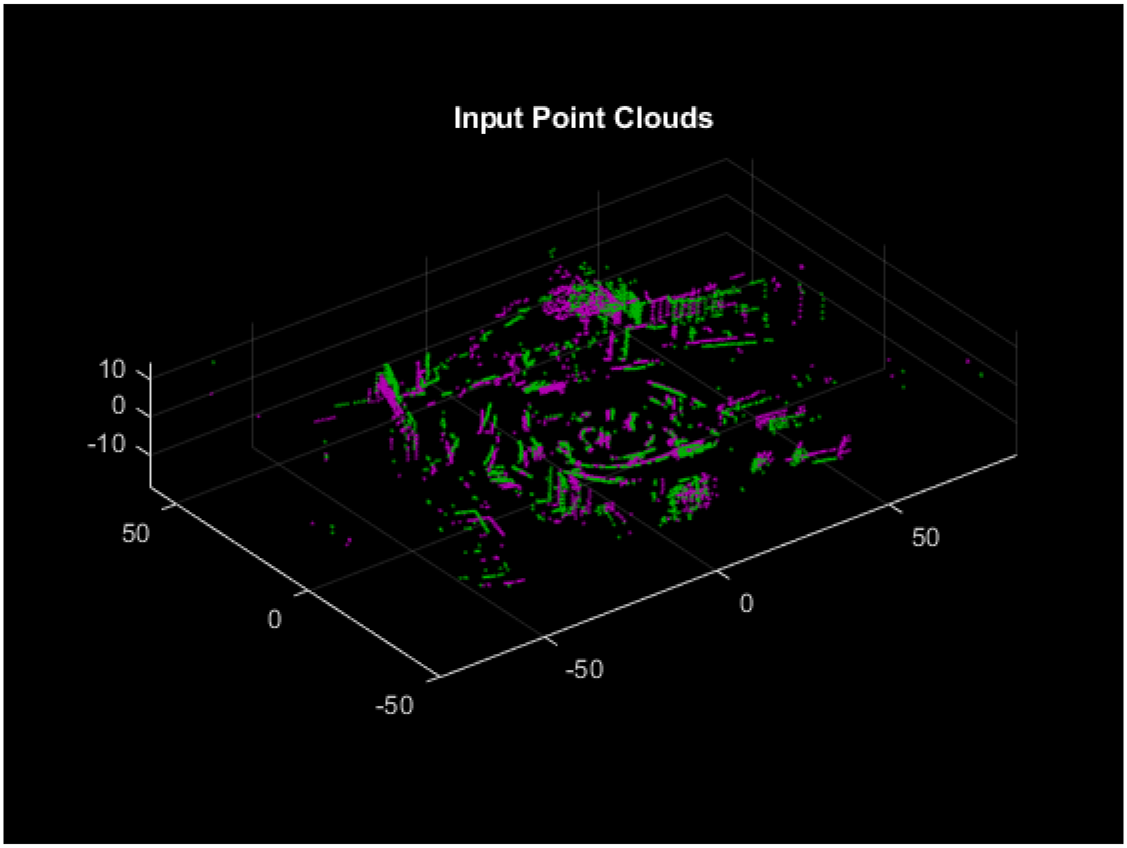
```
frameNumber = 1;
skipFrame = 5;
fixed = readFrame(veloReader,frameNumber);
moving = readFrame(veloReader,frameNumber + skipFrame);
```

Segment and remove the ground plane from the fixed point cloud.

```
groundPtsIdxFixed = segmentGroundFromLidarData(fixed);
fixedSeg = select(fixed,~groundPtsIdxFixed,'OutputSize','full');
```

Segment and remove the ground plane from the moving point cloud. Plot both point clouds.

```
groundPtsIdxMoving = segmentGroundFromLidarData(moving);
movingSeg = select(moving,~groundPtsIdxMoving,'OutputSize','full');
figure
pcshowpair(fixedSeg,movingSeg)
ylim([-50 60])
title('Input Point Clouds')
```



The superimposed input point clouds are color coded as follows:

- Magenta - Fixed point cloud
- Green - Moving Point Cloud

Downsample the point clouds.

```
fixedDownsampled = pcdownsample(fixedSeg,'gridAverage',0.2);
movingDownsampled = pcdownsample(movingSeg,'gridAverage',0.2);
```

Extract features from the point clouds.

```
[fixedFeature,fixedValidInds] = extractFPFHFeatures(fixedDownsampled);
[movingFeature,movingValidInds] = extractFPFHFeatures(movingDownsampled);
fixedValidPts = select(fixedDownsampled,fixedValidInds);
movingValidPts = select(movingDownsampled,movingValidInds);
```

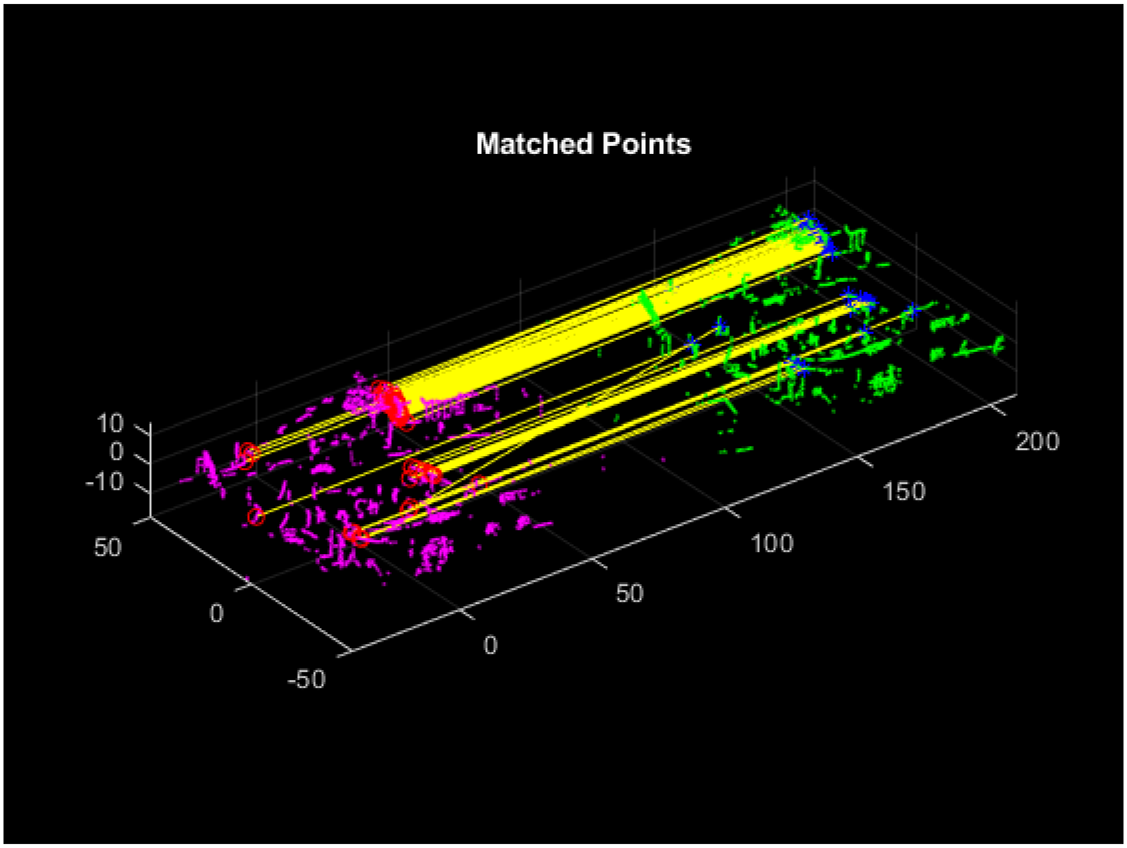
Match features between the point clouds.

```
indexPairs = pcmatchfeatures(movingFeature,fixedFeature,movingValidPts, ...
    fixedValidPts);
```

```
matchedFixedPts = select(fixedValidPts,indexPairs(:,2));
matchedMovingPts = select(movingValidPts,indexPairs(:,1));
```

Visualize the matches.

```
figure
pcshowMatchedFeatures(movingSeg,fixedSeg,matchedMovingPts,matchedFixedPts, ...
    'Method',"montage")
xlim([-40 210])
ylim([-50 50])
title('Matched Points')
```



The matched features and point clouds are color coded to improve visualization:

- Magenta - Moving point cloud.
- Green - Fixed point cloud.
- Red circle - Matched points in the moving point cloud.
- Blue asterisk - Matched points in the fixed point cloud.
- Yellow - Line connecting matched features.

Input Arguments

features1 — First feature set

M_1-by-N matrix

First feature set, specified as an M_1-by-N matrix. The matrix contains M_1 features, and N is the length of each feature vector. Each row represents a single feature.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`features2` — Second feature set
M_2-by-N matrix

Second feature set, specified as an M_2-by-N matrix. The matrix contains M_2 features, and N is the length of each feature vector. Each row represents a single feature.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`ptCloud1` — First point cloud
`pointCloud` object

First point cloud, specified as a `pointCloud` object.

`ptCloud2` — Second point cloud
`pointCloud` object

Second point cloud, specified as a `pointCloud` 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: `'MatchThreshold',0.03` sets the normalized distance threshold for matching features to `0.03`.

`Method` — Matching method
`'Exhaustive'` (default) | `'Approximate'`

Matching method, specified as the comma-separated pair consisting of `'Method'` and either `'Exhaustive'` or `'Approximate'`. The method determines how the function finds the nearest neighbors between `features1` and `features2`. Two feature vectors match when the distance between them is less or equal to the matching threshold.

- `'Exhaustive'` — Compute the pairwise distance between the specified feature vectors.
- `'Approximate'` — Use an efficient approximate nearest neighbor search. Use this method for large feature sets. [1]

Data Types: `char` | `string`

`MatchThreshold` — Matching threshold
`0.01` (default) | scalar in the range (`0, 1`]

Matching threshold, specified as the comma-separated pair consisting of `'MatchThreshold'` and a scalar in the range (`0, 1`].

Two feature vectors match when the normalized Euclidean distance between them is less than or equal to the matching threshold. A higher value may result in additional matches, but increases the risk of false positives.

Data Types: `single` | `double`

`RejectRatio` — Spatial relation threshold
`0.95` (default) | scalar in the range (`0, 1`)

Spatial relation threshold, specified as the comma-separated pair consisting of `'RejectRatio'` and a scalar in the range (`0, 1`).

The function uses point cloud data to estimate the spatial relation between the points associated with potential feature matches and reject matches based on the spatial relation threshold. A lower spatial relation threshold may result in additional matches, but increases the risk of false positives.

The function does not consider the spatial relation threshold if you do not specify values for the `ptCloud1` and `ptCloud2` input arguments.

Data Types: `single` | `double`

Output Arguments

`indexPairs` — Indices of matched features
P-by-2 matrix

Indices of matched features, returned as a *P*-by-2 matrix. *P* is the number of matched features. Each row corresponds to a matched feature between the `features1` and `features2` inputs, where the first element is the index of the feature in `features1` and the second element is the index of the matching feature in `features2`.

Data Types: `uint32`

`scores` — Normalized Euclidean distance between matching features
P-element column vector

Normalized Euclidean distance between matching features, returned as a *P*-element column vector. The *i*th element of the vector is the distance between the matched features in the *i*th row of the `indexPairs` output.

Data Types: `single` | `double`

References

[1] Muja, Marius and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration." In *Proceedings of the Fourth International Conference on Computer Vision Theory and Applications*, 331-40. Lisboa, Portugal: SciTePress - Science and Technology Publications, 2009. https://doi.org/10.5220/0001787803310340.

[2] Zhou, Qian-Yi, Jaesik Park, and Vladlen Koltun. "Fast global registration." In *European Conference on Computer Vision*, pp. 766-782. Springer, Cham, 2016.

Extended Capabilities

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

See Also

Functions
extractFPFHFeatures | pcshowMatchedFeatures

Introduced in R2020b

pcshowMatchedFeatures

Display point clouds with matched feature points

Syntax

```
pcshowMatchedFeatures(ptCloud1,ptCloud2,matchedPtCloud1,matchedPtCloud2)
ax = pcshowMatchedFeatures( ___ )
[ ___ ] = pcshowMatchedFeatures( ___ ,Name,Value)
```

Description

`pcshowMatchedFeatures(ptCloud1,ptCloud2,matchedPtCloud1,matchedPtCloud2)` displays point clouds, `ptCloud1` and `ptCloud2`, with their matched feature points, `matchedPtCloud1` and `matchedPtCloud2`, color coded by point cloud and each connected to the corresponding point in the other point cloud by a line.

`ax = pcshowMatchedFeatures( ___ )` additionally returns an `axes` object using the input arguments from the previous syntax.

`[ ___ ] = pcshowMatchedFeatures( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to any combination of arguments in previous syntaxes. For example, `'Method','montage'` visualizes the point clouds next to each other in the axes.

Examples

Match and Visualize Corresponding Features in Point Clouds

This example shows how to match corresponding point cloud features and visualize them using the `pcmatchfeatures` and `pcshowMatchedFeatures` functions.

Construct a `velodyneFileReader` object.

```
veloReader = velodyneFileReader('lidarData_ConstructionRoad.pcap','HDL32E');
```

Read two point clouds using the `velodyneFileReader` object.

```
frameNumber = 1;
skipFrame = 5;
fixed = readFrame(veloReader,frameNumber);
moving = readFrame(veloReader,frameNumber + skipFrame);
```

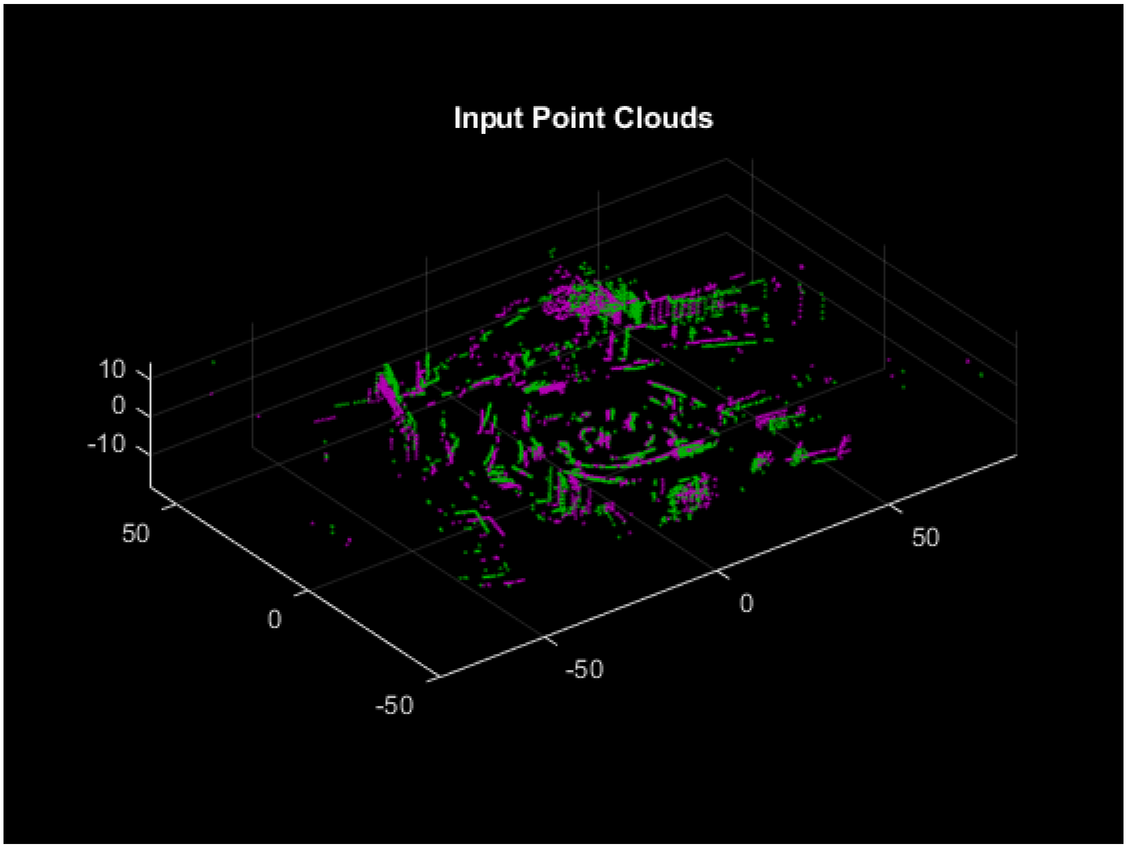
Segment and remove the ground plane from the fixed point cloud.

```
groundPtsIdxFixed = segmentGroundFromLidarData(fixed);
fixedSeg = select(fixed,~groundPtsIdxFixed,'OutputSize','full');
```

Segment and remove the ground plane from the moving point cloud. Plot both point clouds.

```
groundPtsIdxMoving = segmentGroundFromLidarData(moving);
movingSeg = select(moving,~groundPtsIdxMoving,'OutputSize','full');
```

```
figure
pcshowpair(fixedSeg,movingSeg)
ylim([-50 60])
title('Input Point Clouds')
```



The superimposed input point clouds are color coded as follows:

- Magenta - Fixed point cloud
- Green - Moving Point Cloud

Downsample the point clouds.

```
fixedDownsampled = pcdownsample(fixedSeg,'gridAverage',0.2);
movingDownsampled = pcdownsample(movingSeg,'gridAverage',0.2);
```

Extract features from the point clouds.

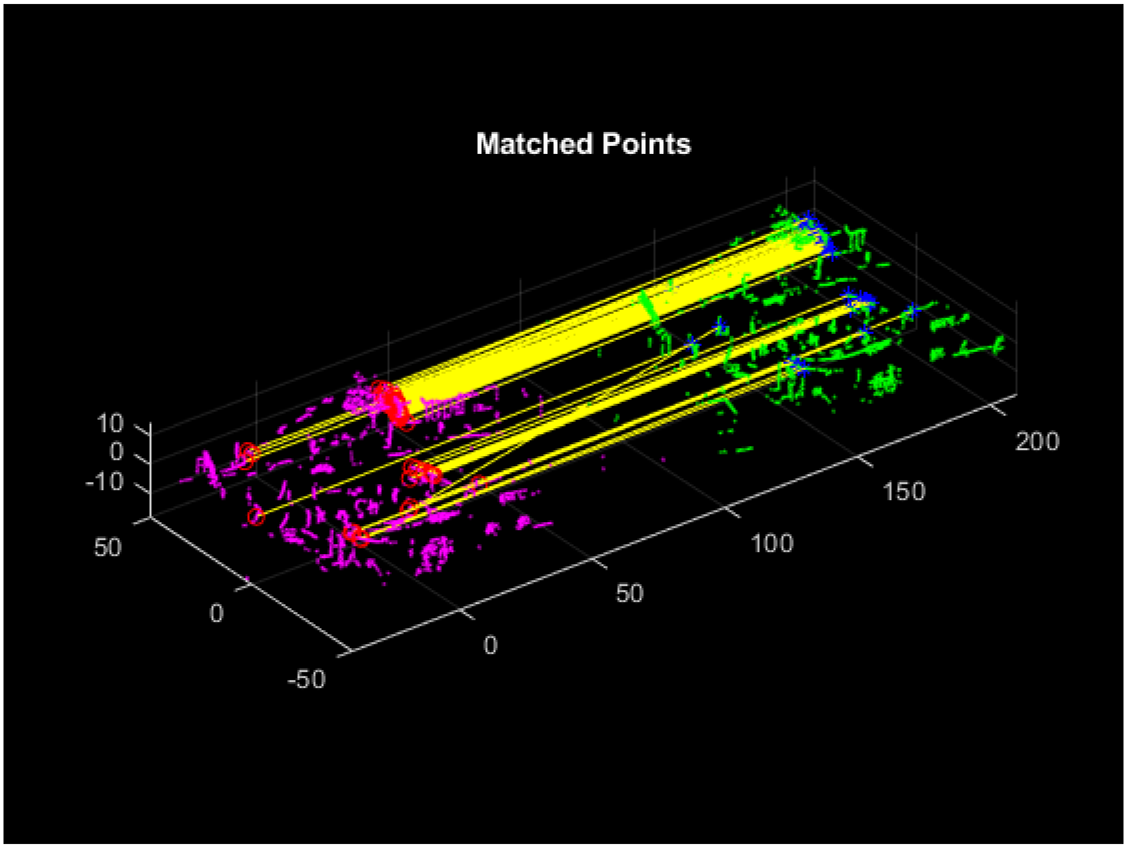
```
[fixedFeature,fixedValidInds] = extractFPFHFeatures(fixedDownsampled);
[movingFeature,movingValidInds] = extractFPFHFeatures(movingDownsampled);
fixedValidPts = select(fixedDownsampled,fixedValidInds);
movingValidPts = select(movingDownsampled,movingValidInds);
```

Match features between the point clouds.

```
indexPairs = pcmatchfeatures(movingFeature,fixedFeature,movingValidPts, ...
    fixedValidPts);
matchedFixedPts = select(fixedValidPts,indexPairs(:,2));
matchedMovingPts = select(movingValidPts,indexPairs(:,1));
```

Visualize the matches.

```
figure
pcshowMatchedFeatures(movingSeg,fixedSeg,matchedMovingPts,matchedFixedPts, ...
    'Method',"montage")
xlim([-40 210])
ylim([-50 50])
title('Matched Points')
```



The matched features and point clouds are color coded to improve visualization:

- Magenta - Moving point cloud.
- Green - Fixed point cloud.
- Red circle - Matched points in the moving point cloud.
- Blue asterisk - Matched points in the fixed point cloud.
- Yellow - Line connecting matched features.

Input Arguments

ptCloud1 — First point cloud
pointCloud object

First point cloud, specified as a `pointCloud` object.

`ptCloud2` — Second point cloud
`pointCloud` object

Second point cloud, specified as a `pointCloud` object.

`matchedPtCloud1` — Matched points in first point cloud
`pointCloud` object

Matched points in the first point cloud, specified as a `pointCloud` object. Each point is a feature match for the point with the corresponding index in `matchedPtCloud2`.

`matchedPtCloud2` — Matched points in second point cloud
`pointCloud` object

Matched points in the second point cloud, specified as a `pointCloud` object. Each point is a feature match for the point with the corresponding index in `matchedPtCloud1`.

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: `'Method','montage'` visualizes the point clouds next to each other in the axes.

`Method` — Display method
`'overlay'` (default) | `'montage'`

Display method, specified as the comma-separated pair consisting of `'Method'` and one of these options:

- `'overlay'` — Overlay `ptCloud2` on `ptCloud1`.
- `'montage'` — Display `ptCloud1` and `ptCloud2` next to each other in the same axes.

Data Types: `char` | `string`

`PlotOptions` — Line style and color options
`{'ro','b*','y-'}` (default) | cell array of character vectors

Line style and color options, specified as the comma-separated pair consisting of `'PlotOptions'` and a cell array of character vectors of the form {*MarkerStyle1*, *MarkerStyle2*, *LineStyle*}. *MarkerStyle1* specifies the color and marker symbol for the matched points `matchedPtCloud1` in the first point cloud `ptCloud1`. *MarkerStyle2* specifies the color and marker symbol for the matched points `matchedPtCloud2` in the second point cloud `ptCloud2`. *LineStyle* specifies the color and line style of the lines connecting the matched points of the first point cloud to the matched points of the second. For more information on line styles, marker symbols, and colors, see `LineSpec`.

Data Types: `char`

`Parent` — Output axes
`axes` graphics object

Output axes, specified as the comma-separated pair consisting of `'Parent'` and an `axes` graphics object.

Output Arguments

ax — Axes handle
`axes` graphics object

Axes handle, returned as an `axes` graphics object.

See Also

Functions
`extractFPFHFeatures` | `pcmatchfeatures`

Introduced in R2020b

squeezesegv2Layers

Create SqueezeSegV2 segmentation network for organized lidar point cloud

Syntax

```
lgraph = squeezesegv2Layers(inputSize,numClasses)
lgraph = squeezesegv2Layers( ___ ,Name,Value)
```

Description

`lgraph = squeezesegv2Layers(inputSize,numClasses)` returns a SqueezeSegV2 layer graph `lgraph` for organized point clouds of size `inputSize` and the number of classes `numClasses`.

SqueezeSegV2 is a convolutional neural network that predicts pointwise labels for an organized lidar point cloud.

Use the `squeezesegv2Layers` function to create the network architecture for SqueezeSegV2. This function requires Deep Learning Toolbox™.

`lgraph = squeezesegv2Layers( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax. For example, `'NumEncoderModules',4` sets the number of encoders used to create the network to four.

Examples

Create Standard SqueezeSegV2 Network

Set the network input parameters.

```
inputSize = [64 512 5];
numClasses = 4;
```

Create a SqueezeSegV2 layer graph.

```
lgraph = squeezesegv2Layers(inputSize,numClasses)
```

```
lgraph = 
  LayerGraph with properties:

         Layers: [168x1 nnet.cnn.layer.Layer]
    Connections: [186x2 table]
     InputNames: {'input'}
    OutputNames: {'focalloss'}
```

Display the network.

```
analyzeNetwork(lgraph)
```

Create Custom SqueezeSegV2 Network

Set the network input parameters.

```
inputSize = [64 512 6];
numClasses = 2;
```

Create a custom SqueezeSegV2 layer graph.

```
lgraph = squeezesegv2Layers(inputSize,numClasses,...
'NumEncoderModules',4,'NumContextAggregationModules',2)
```

```
lgraph =
  LayerGraph with properties:

         Layers: [232x1 nnet.cnn.layer.Layer]
    Connections: [257x2 table]
     InputNames: {'input'}
    OutputNames: {'focalloss'}
```

Display the network.

```
analyzeNetwork(lgraph)
```

Input Arguments

inputSize — Size of network input

two-element row vector | three-element row vector

Size of the network input, specified as one of these options:

- Two-element vector of the form [*height width*].
- Three-element vector of the form [*height width channels*], where *channels* specifies the number of input channels. Set *channels* to 3 for RGB images, to 1 for grayscale images, or to the number of channels for multispectral and hyperspectral images.

numClasses — Number of classes

integer greater than 1

Number of semantic segmentation classes, specified as an integer greater than 1.

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: `'NumEncoderModules',4` sets the number of encoders used to create the network to four.

NumEncoderModules — Number of encoder modules

2 (default) | nonnegative integer

Number of encoder modules used to create the network, specified as the comma-separated pair consisting of `'NumEncoderModules'` and a nonnegative integer. Each encoder module consists of two fire modules and one max-pooling layer connected sequentially. If you specify `0`, then the function returns a network with a default encoder that consists of convolution and max-pooling layers with no fire modules. Use this name-value pair to customize the number of fire modules in the network.

`NumContextAggregationModules` — Number of context aggregation modules
`3` (default) | integer in the range `[0,3]`

Number of context aggregation modules (CAMs), specified as the comma-separated pair consisting of `'NumContextAggregationModules'` and an integer in the range [`0,3`]. If you specify `0`, then the function creates a network without a CAM.

Output Arguments

`lgraph` — Layers
`LayerGraph` object

Layers that represent the SqueezeSegV2 network architecture, returned as a `layerGraph` object.

More About

SqueezeSegV2 Network

- A SqueezeSegV2 network consists of encoder modules, CAMs, intermediate fixed fire modules [1] for feature extraction, and decoder modules. The function automatically configures the number of decoder modules based on the specified number of encoder modules.
- The function uses narrow-normal weight initialization method to initialize the weights of each convolution layer within encoder and decoder subnetworks .
- The function initializes all bias terms to zero.
- The function adds the padding for all convolution and max-pooling layers such that the output has the same size as the input (if the stride equals 1).
- The height of the input tensor is significantly lower than the width in organized lidar point cloud data. To address this, the network downsamples the width dimension of the input data in convolution and max-pooling layers. The width of the input data must be a multiple of $2^{(D+2)}$, where *D* is the number of encoder modules used to create the network.
- This function does not provide a recurrent conditional random field (CRF) layer.

References

[1] Wu, Bichen, Xuanyu Zhou, Sicheng Zhao, Xiangyu Yue, and Kurt Keutzer. "SqueezeSegV2: Improved Model Structure and Unsupervised Domain Adaptation for Road-Object Segmentation from a LiDAR Point Cloud." In *2019 International Conference on Robotics and Automation (ICRA)*, 4376–82. Montreal, QC, Canada: IEEE, 2019.https://doi.org/10.1109/ICRA.2019.8793495.

Extended Capabilities

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

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Functions
`evaluateSemanticSegmentation` | `semanticseg` | `trainNetwork`

Objects
`DAGNetwork` | `focalLossLayer` | `layerGraph` | `pixelClassificationLayer`

Topics
“Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network”

Introduced in R2020b

matchScans

Estimate pose between two laser scans

Syntax

```
pose = matchScans(currScan,refScan)
pose = matchScans(currRanges,currAngles,refRanges,refAngles)
[pose,stats] = matchScans( ___ )
[ ___ ] = matchScans( ___ ,Name,Value)
```

Description

`pose = matchScans(currScan,refScan)` finds the relative pose between a reference `lidarScan` and a current `lidarScan` object using the normal distributions transform (NDT).

`pose = matchScans(currRanges,currAngles,refRanges,refAngles)` finds the relative pose between two laser scans specified as ranges and angles.

`[pose,stats] = matchScans( ___ )` returns additional statistics about the scan match result using the previous input arguments.

`[ ___ ] = matchScans( ___ ,Name,Value)` specifies additional options specified by one or more `Name,Value` pair arguments.

Examples

Match Lidar Scans

Create a reference lidar scan using `lidarScan`. Specify ranges and angles as vectors.

```
refRanges = 5*ones(1,300);
refAngles = linspace(-pi/2,pi/2,300);
refScan = lidarScan(refRanges,refAngles);
```

Using the `transformScan` (Robotics System Toolbox) function, generate a second lidar scan at an `x,y` offset of `(0.5,0.2)`.

```
currScan = transformScan(refScan,[0.5 0.2 0]);
```

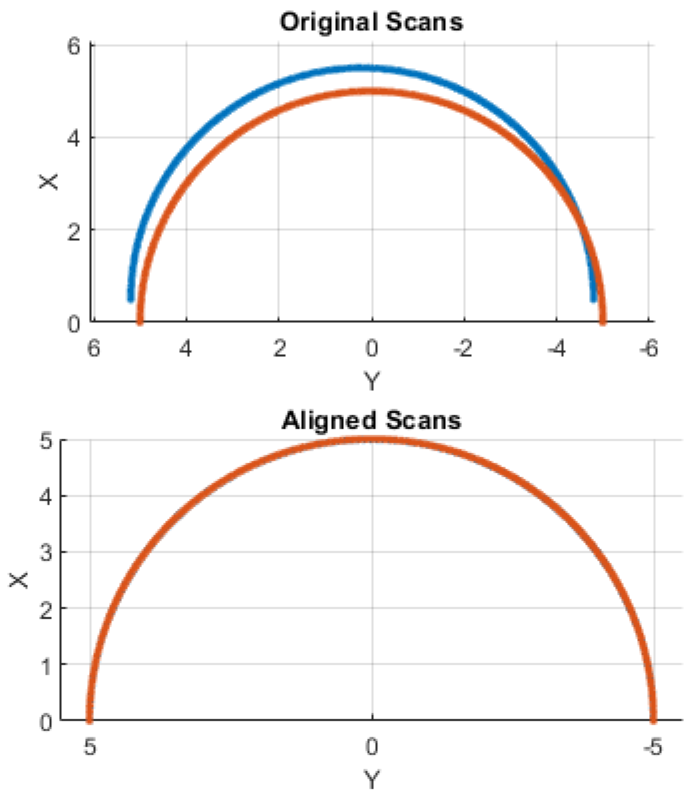
Match the reference scan and the second scan to estimate the pose difference between them.

```
pose = matchScans(currScan,refScan);
```

Use the `transformScan` function to align the scans by transforming the second scan into the frame of the first scan using the relative pose difference. Plot both the original scans and the aligned scans.

```
currScan2 = transformScan(currScan,pose);
subplot(2,1,1);
hold on
plot(currScan)
```

```
plot(refScan)
title('Original Scans')
hold off
subplot(2,1,2);
hold on
plot(currScan2)
plot(refScan)
title('Aligned Scans')
xlim([0 5])
hold off
```



Input Arguments

currScan — Current lidar scan readings
lidarScan object

Current lidar scan readings, specified as a `lidarScan` object.

Your lidar scan can contain `Inf` and `NaN` values, but the algorithm ignores them.

refScan — Reference lidar scan readings
lidarScan object

Reference lidar scan readings, specified as a `lidarScan` object.

Your lidar scan can contain `Inf` and `NaN` values, but the algorithm ignores them.

`currRanges` — Current laser scan ranges
vector in meters

Current laser scan ranges, specified as a vector. Ranges are given as distances to objects measured from the laser sensor.

Your laser scan ranges can contain `Inf` and `NaN` values, but the algorithm ignores them.

`currAngles` — Current laser scan angles
vector in radians

Current laser scan angles, specified as a vector in radians. Angles are given as the orientations of the corresponding range measurements.

`refRanges` — Reference laser scan ranges
vector in meters

Reference laser scan ranges, specified as a vector in meters. Ranges are given as distances to objects measured from the laser sensor.

Your laser scan ranges can contain `Inf` and `NaN` values, but the algorithm ignores them.

`refAngles` — Reference laser scan angles
vector in radians

Reference laser scan angles, specified as a vector in radians. Angles are given as the orientations of the corresponding range measurements.

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: `"InitialPose",[1 1 pi/2]`

`SolverAlgorithm` — Optimization algorithm
`"trust-region"` (default) | `"fminunc"`

Optimization algorithm, specified as either `"trust-region"` or `"fminunc"`. Using `"fminunc"` requires an Optimization Toolbox™ license.

`InitialPose` — Initial guess of current pose
[0 0 0] (default) | `[x y theta]`

Initial guess of the current pose relative to the reference laser scan, specified as the comma-separated pair consisting of `"InitialPose"` and an `[x y theta]` vector. `[x y]` is the translation in meters and `theta` is the rotation in radians.

`CellSize` — Length of cell side
1 (default) | numeric scalar

Length of a cell side in meters, specified as the comma-separated pair consisting of `"CellSize"` and a numeric scalar. `matchScans` uses the cell size to discretize the space for the NDT algorithm.

Tuning the cell size is important for proper use of the NDT algorithm. The optimal cell size depends on the input scans and the environment of your robot. Larger cell sizes can lead to less accurate matching with poorly sampled areas. Smaller cell sizes require more memory and less variation between subsequent scans. Sensor noise influences the algorithm with smaller cell sizes as well. Choosing a proper cell size depends on the scale of your environment and the input data.

`MaxIterations` — Maximum number of iterations

400 (default) | scalar integer

Maximum number of iterations, specified as the comma-separated pair consisting of `"MaxIterations"` and a scalar integer. A larger number of iterations results in more accurate pose estimates, but at the expense of longer execution time.

`ScoreTolerance` — Lower bounds on the change in NDT score

`1e-6` (default) | numeric scalar

Lower bound on the change in NDT score, specified as the comma-separated pair consisting of `"ScoreTolerance"` and a numeric scalar. The NDT score is stored in the `Score` field of the output `stats` structure. Between iterations, if the score changes by less than this tolerance, the algorithm converges to a solution. A smaller tolerance results in more accurate pose estimates, but requires a longer execution time.

Output Arguments

pose — Pose of current scan

`[x y theta]`

Pose of current scan relative to the reference scan, returned as `[x y theta]`, where `[x y]` is the translation in meters and `theta` is the rotation in radians.

`stats` — Scan matching statistics

structure

Scan matching statistics, returned as a structure with the following fields:

- `Score` — Numeric scalar representing the NDT score while performing scan matching. This score is an estimate of the likelihood that the transformed current scan matches the reference scan. `Score` is always nonnegative. Larger scores indicate a better match.
- `Hessian` — 3-by-3 matrix representing the Hessian of the NDT cost function at the given `pose` solution. The Hessian is used as an indicator of the uncertainty associated with the pose estimate.

References

[1] Biber, P., and W. Strasser. "The Normal Distributions Transform: A New Approach to Laser Scan Matching." *Intelligent Robots and Systems Proceedings*. 2003.

[2] Magnusson, Martin. "The Three-Dimensional Normal-Distributions Transform -- an Efficient Representation for Registration, Surface Analysis, and Loop Detection." PhD Dissertation. Örebro University, School of Science and Technology, 2009.

Extended Capabilities

C/C++ Code Generation

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

Code generation is supported for the default `SolverAlgorithm`, `"trust-region"`. You cannot use the `"fminunc"` algorithm in code generation.

See Also

Functions
`lidarScan` | `matchScansGrid` | `matchScansLine`

Classes
`monteCarloLocalization` | `occupancyMap`

Introduced in R2020b

matchScansGrid

Estimate pose between two lidar scans using grid-based search

Syntax

```
pose = matchScansGrid(currScan,refScan)
[pose,stats] = matchScansGrid( ___ )
[ ___ ] = matchScansGrid( ___ ,Name,Value)
```

Description

`pose = matchScansGrid(currScan,refScan)` finds the relative pose between a reference `lidarScan` and a current `lidarScan` object using a grid-based search. `matchScansGrid` converts lidar scan pairs into probabilistic grids and finds the pose between the two scans by correlating their grids. The function uses a branch-and-bound strategy to speed up computation over large discretized search windows.

`[pose,stats] = matchScansGrid( ___ )` returns additional statistics about the scan match result using the previous input arguments.

`[ ___ ] = matchScansGrid( ___ ,Name,Value)` specifies options using one or more `Name,Value` pair arguments. For example, `'InitialPose',[1 1 pi/2]` specifies an initial pose estimate for scan matching.

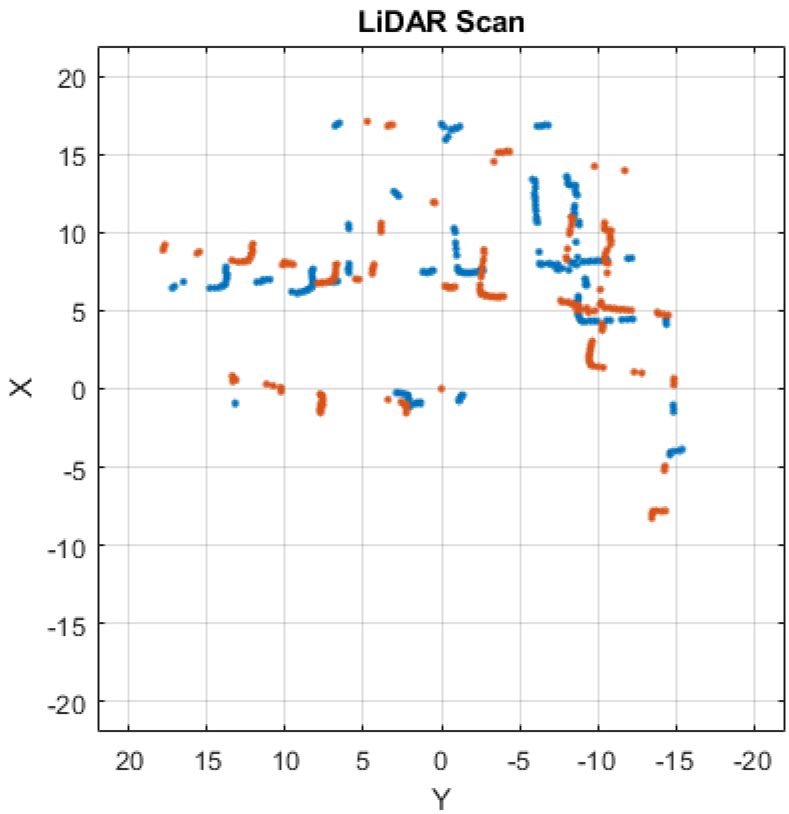
Examples

Match Scans Using Grid-Based Search

Perform scan matching using a grid-based search to estimate the pose between two laser scans. Generate a probabilistic grid from the scans and estimate the pose difference from those grids.

Load the laser scan data. These two scans are from an actual lidar sensor with changes in the robot pose and are stored as `lidarScan` objects.

```
load laserScans.mat scan scan2
plot(scan)
hold on
plot(scan2)
hold off
```

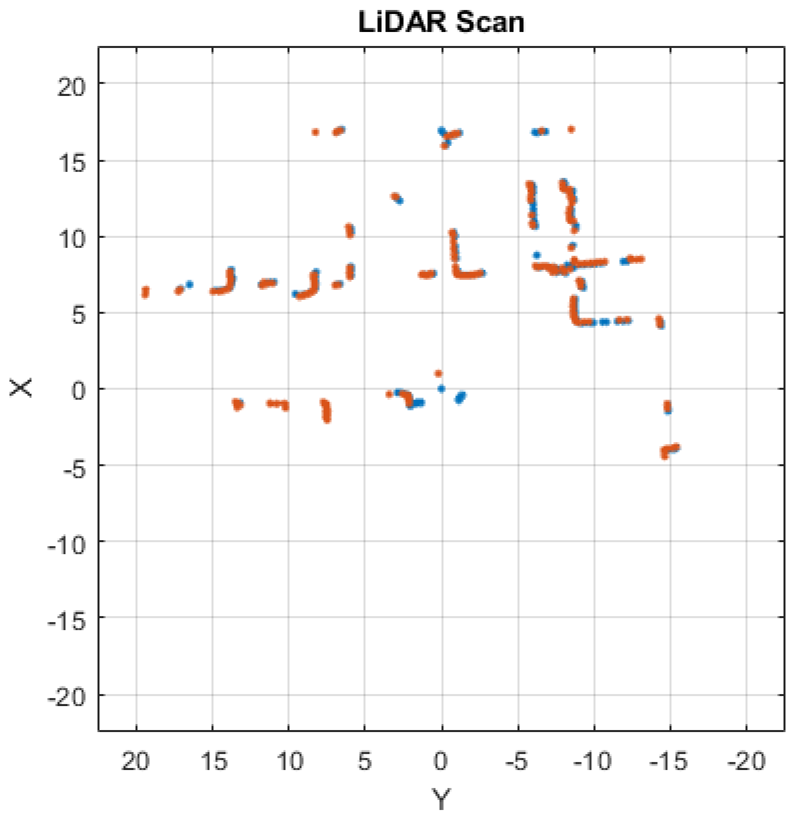


Use `matchScansGrid` to estimate the pose between the two scans.

```
relPose = matchScansGrid(scan2,scan);
```

Using the estimated pose, transform the current scan back to the reference scan. The scans overlap closely when you plot them together.

```
scan2Tformed = transformScan(scan2,relPose);
plot(scan)
hold on
plot(scan2Tformed)
hold off
```



Input Arguments

currScan — Current lidar scan readings
lidarScan object

Current lidar scan readings, specified as a lidarScan object.

Your lidar scan can contain Inf and NaN values, but the algorithm ignores them.

refScan — Reference lidar scan readings
lidarScan object

Reference lidar scan readings, specified as a lidarScan object.

Your lidar scan can contain Inf and NaN values, but the algorithm ignores them.

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: 'InitialPose',[1 1 pi/2]

InitialPose — Initial guess of current pose
[0 0 0] (default) | [x y theta]

Initial guess of the current pose relative to the reference laser scan, specified as the comma-separated pair consisting of `'InitialPose'` and an `[x y theta]` vector. `[x y]` is the translation in meters and `theta` is the rotation in radians.

`Resolution` — Grid cells per meter
`20` (default) | positive integer

Grid cells per meter, specified as the comma-separated pair consisting of `'Resolution'` and a positive integer. The accuracy of the scan matching result is accurate up to the grid cell size.

`MaxRange` — Maximum range of lidar sensor
`8` (default) | positive scalar

Maximum range of lidar sensor, specified as the comma-separated pair consisting of `'MaxRange'` and a positive scalar.

`TranslationSearchRange` — Search range for translation
`[4 4]` (default) | `[x y]` vector

Search range for translation, specified as the comma-separated pair consisting of `'TranslationSearchRange'` and an `[x y]` vector. These values define the search window in meters around the initial translation estimate given in `InitialPose`. If the `InitialPose` is given as `[x0 y0]`, then the search window coordinates are `[x0-x x0+x]` and `[y0-y y0+y]`. This parameter is used only when `InitialPose` is specified.

`RotationSearchRange` — Search range for rotation
`pi/4` (default) | positive scalar

Search range for rotation, specified as the comma-separated pair consisting of `'RotationSearchRange'` and a positive scalar. This value defines the search window in radians around the initial rotation estimate given in `InitialPose`. If the `InitialPose` rotation is given as `th0`, then the search window is `[th0-a th0+a]`, where `a` is the rotation search range. This parameter is used only when `InitialPose` is specified.

Output Arguments

`pose` — Pose of current scan
`[x y theta]` vector

Pose of current scan relative to the reference scan, returned as an `[x y theta]` vector, where `[x y]` is the translation in meters and `theta` is the rotation in radians.

`stats` — Scan matching statistics
structure

Scan matching statistics, returned as a structure with the following field:

- `Score` — Numeric scalar representing the score while performing scan matching. This score is an estimate of the likelihood that the transformed current scan matches the reference scan. `Score` is always nonnegative. Larger scores indicate a better match, but values vary depending on the lidar data used.
- `Covariance` — Estimated covariance representing the confidence of the computed relative pose, returned as a 3-by-3 matrix.

References

[1] Hess, Wolfgang, Damon Kohler, Holger Rapp, and Daniel Andor. "Real-Time Loop Closure in 2D LIDAR SLAM." *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016.

Extended Capabilities

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

See Also

Functions
`lidarScan` | `matchScans` | `matchScansLine`

Classes
`lidarSLAM`

Introduced in R2020b

matchScansLine

Estimate pose between two laser scans using line features

Syntax

```
relpose = matchScansLine(currScan,refScan,initialRelPose)
[relpose,stats] = matchScansLine( ___ )
[relpose,stats,debugInfo] = matchScansLine( ___ )
[ ___ ] = matchScansLine( ___ ,Name,Value)
```

Description

`relpose = matchScansLine(currScan,refScan,initialRelPose)` estimates the relative pose between two scans based on matched line features identified in each scan. Specify an initial guess on the relative pose, `initialRelPose`.

`[relpose,stats] = matchScansLine( ___ )` returns additional information about the covariance and exit condition in `stats` as a structure using the previous inputs.

`[relpose,stats,debugInfo] = matchScansLine( ___ )` returns additional debugging info, `debugInfo`, from the line-based scan matching result.

`[ ___ ] = matchScansLine( ___ ,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

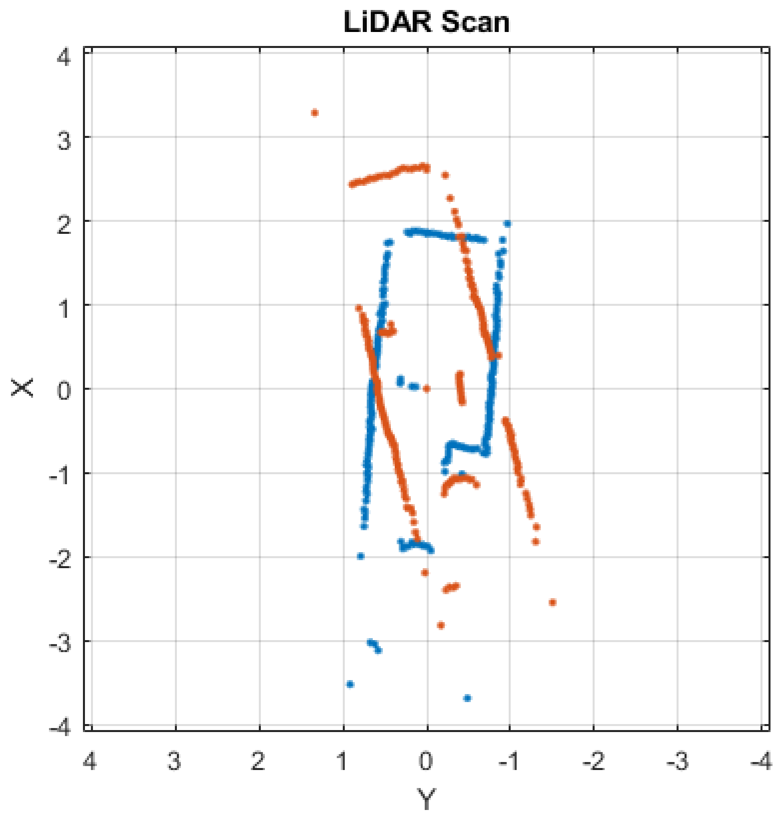
Examples

Estimate Pose of Scans with Line Features

This example shows how to use the `matchScansLine` function to estimate the relative pose between lidar scans given an initial estimate. The identified line features are visualized to show how the scan-matching algorithm associates features between scans.

Load a pair of lidar scans. The `.mat` file also contains an initial guess of the relative pose difference, `initGuess`, which could be based on odometry or other sensor data.

```
load tb3_scanPair.mat
plot(s1)
hold on
plot(s2)
hold off
```



Set parameters for line feature extraction and association. The noise of the lidar data determines the smoothness threshold, which defines when a line break occurs for a specific line feature. Increase this value for more noisy lidar data. The compatibility scale determines when features are considered matches. Increase this value for looser restrictions on line feature parameters.

```
smoothnessThresh = 0.2;
compatibilityScale = 0.002;
```

Call `matchScansLine` with the given initial guess and other parameters specified as name-value pairs. The function calculates line features for each scan, attempts to match them, and uses an overall estimate to get the difference in pose.

```
[relPose, stats, debugInfo] = matchScansLine(s2, s1, initGuess, ...
                                   'SmoothnessThreshold', smoothnessThresh, ...
                                   'CompatibilityScale', compatibilityScale);
```

After matching the scans, the `debugInfo` output gives you information about the detected line feature parameters, `[rho alpha]`, and the hypothesis of which features match between scans.

`debugInfo.MatchHypothesis` states that the first, second, and sixth line feature in `s1` match the fifth, second, and fourth features in `s2`.

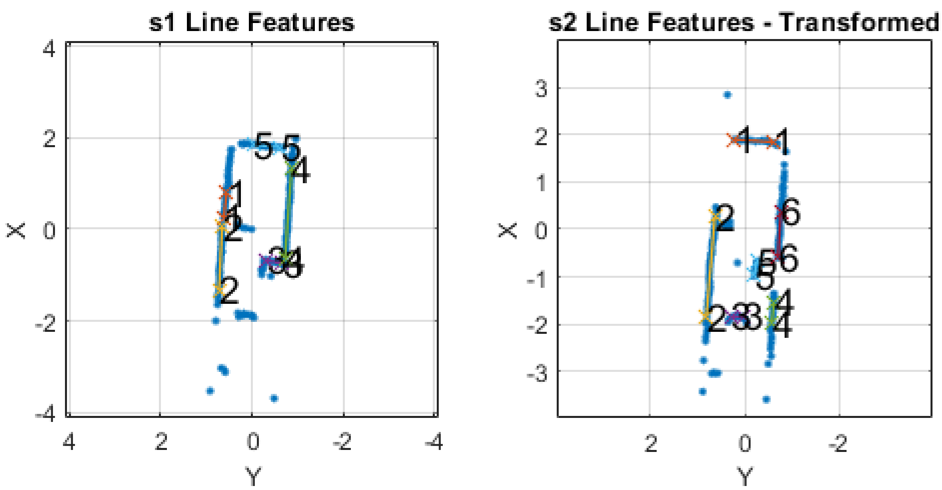
```
debugInfo.MatchHypothesis
```

```
ans = 1×6

     5     2     0     0     0     4
```

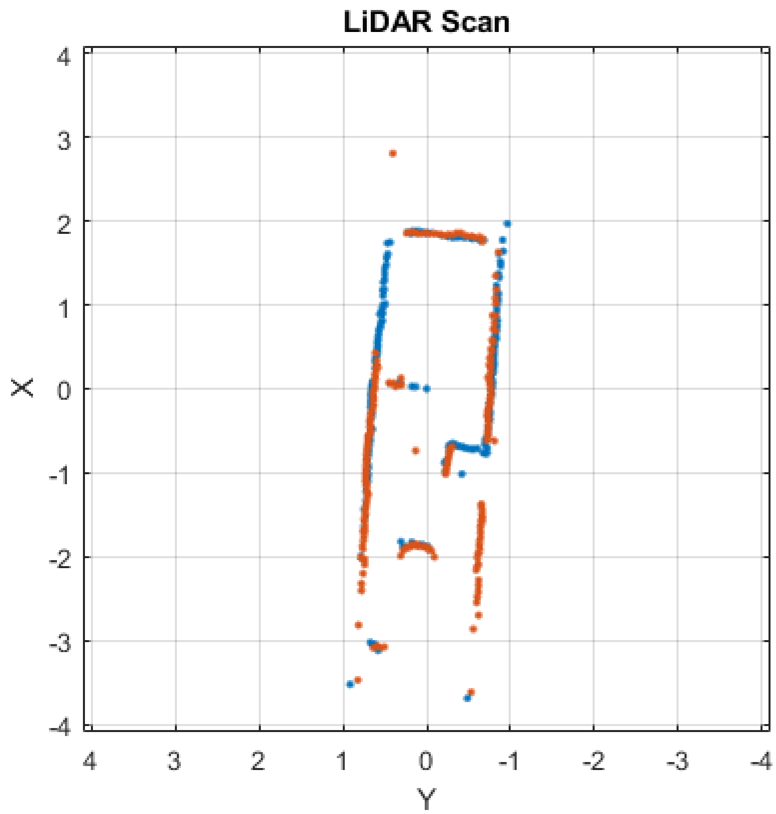
The provided helper function plots these two scans and the features extracted with labels. `s2` is transformed to be in the same frame based on the initial guess for relative pose.

```
exampleHelperShowLineFeaturesInScan(s1, s2, debugInfo, initGuess);
```



Use the estimated relative pose from `matchScansLine` to transform `s2`. Then, plot both scans to show that the relative pose difference is accurate and the scans overlay to show the same environment.

```
s2t = transformScan(s2,relPose);
clf
plot(s1)
hold on
plot(s2t)
hold off
```



Input Arguments

currScan — Current lidar scan readings
lidarScan object

Current lidar scan readings, specified as a `lidarScan`object.

Your lidar scan can contain `Inf` and `NaN` values, but the algorithm ignores them.

refScan — Reference lidar scan readings
lidarScan object

Reference lidar scan readings, specified as a `lidarScan` object.

Your lidar scan can contain `Inf` and `NaN` values, but the algorithm ignores them.

initialRelPose — Initial guess of relative pose
[x y theta]

Initial guess of the current pose relative to the reference laser scan frame, specified an `[x y theta]` vector. `[x y]` is the translation in meters and `theta` is the rotation in radians.

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: `"LineMergeThreshold",[0.10 0.2]`

`SmoothnessThreshold` — Threshold to detect line break points in scan
`0.1` (default) | scalar

Threshold to detect line break points in scan, specified as a scalar. Smoothness is defined by calling `diff(diff(scanData))` and assumes equally spaced scan angles. Scan points corresponding to smoothness values higher than this threshold are considered break points. For lidar scan data with a higher noise level, increase this threshold.

`MinPointsPerLine` — Minimum number of scan points in each line feature
`10` (default) | positive integer greater than 3

Minimum number of scan points in each line feature, specified as a positive integer greater than 3.

A line feature cannot be identified from a set of scan points if the number of points in that set is below this threshold. When the lidar scan data is noisy, setting this property too small may result in low-quality line features being identified and skew the matching result. On the other hand, some key line features may be missed if this number is set too large.

`LineMergeThreshold` — Threshold on line parameters to merge line features
`[0.05 0.1]` (default) | two-element vector `[rho alpha]`

Threshold on line parameters to merge line features, specified as a two-element vector `[rho alpha]`. A line is defined by two parameters:

- `rho` –– Distance from the origin to the line along a vector perpendicular to the line, specified in meters.
- `alpha` –– Angle between the `x`-axis and the `rho` vector, specified in radians.

If the difference between these parameters for two line features is below the given threshold, the line features are merged.

`MinCornerPromenance` — Lower bound on prominence value to detect a corner
`0.05` (default) | positive scalar

Lower bound on prominence value to detect a corner, specified as a positive scalar.

Prominence measures how much a local extrema stands out in the lidar data. Only values higher than this lower bound are considered a corner. Corners help identify line features, but are not part of the feature itself. For noisy lidar scan data, increase this lower bound.

`CompatibilityScale` — Scale used to adjust the compatibility thresholds for feature association
`0.0005` (default) | positive scalar

Scale used to adjust the compatibility thresholds for feature association, specified as a positive scalar. A lower scale means tighter compatibility threshold for associating features. If no features are found in lidar data with obvious line features, increase this value. For invalid feature matches, reduce this value.

Output Arguments

`relpose` — Pose of current scan
`[x y theta]`

Pose of current scan relative to the reference scan, returned as `[x y theta]`, where `[x y]` is the translation in meters and `theta` is the rotation in radians.

`stats` — Scan matching information
structure

Scan matching information, returned as a structure with the following fields:

- `Covariance` –– 3-by-3 matrix representing the covariance of the relative pose estimation. The `matScansLine` function does not provide covariance between the `(x,y)` and the `theta` components of the relative pose. Therefore, the matrix follows the pattern: `[Cxx, Cxy 0; Cyx Cyy 0; 0 0 Ctheta]`.
- `ExitFlag` –– Scalar value indicating the exit condition of the solver:
  - `0` –– No error.
  - `1` –– Insufficient number of line features (< 2) are found in one or both of the scans. Consider using different scans with more line features.
  - `2` –– Insufficient number of line feature matches are identified. This may indicate the `initialRelPose` is invalid or scans are too far apart.

`debugInfo` — Debugging information for line-based scan matching result
structure

Debugging information for line-based scan matching result, returned as a structure with the following fields:

- `ReferenceFeatures` –– Line features extracted from the reference scan as an *n*-by-2 matrix. Each line feature is represented as `[rho alpha]` for the parametric equation, *rho* = *x*•cos(*alpha*) + *y*•sin(*alpha*).
- `ReferenceScanMask` –– Mask indicating which points in the reference scan are used for each line feature as an *n*-by-*p* matrix. Each row corresponds to a row in `ReferenceFeatures` and contains zeros and ones for each point in `refScan`.
- `CurrentFeatures` –– Line features extracted from the current scan as an *n*-by-2 matrix. Each line feature is represented as `[rho alpha]` for the parametric equation, *rho* = *x*•cos(*alpha*) + *y*•sin(*alpha*).
- `CurrentScanMask` –– Mask indicating which points in the current scan are used for each line feature as an *n*-by-*p* matrix. Each row corresponds to a row in `ReferenceFeatures` and contains zeros and ones for each point in `refScan`.
- `MatchHypothesis` –– Best line feature matching hypothesis as an *n* element vector, where *n* is the number of line features in `CurrentFeatures`. Each element represents the corresponding feature in `ReferenceFeatures`and gives the index of the matched feature in `ReferenceFeatures` is an index match the

- `MatchValue` –– Scalar value indicating a score for each `MatchHypothesis`. A lower value is considered a better match. If two elements of `MatchHypothesis` have the same index, the feature with a lower score is used.

References

[1] Neira, J., and J.d. Tardos. "Data Association in Stochastic Mapping Using the Joint Compatibility Test." *IEEE Transactions on Robotics and Automation* 17, no. 6 (2001): 890–97. https://doi.org/10.1109/70.976019.

[2] Shen, Xiaotong, Emilio Frazzoli, Daniela Rus, and Marcelo H. Ang. "Fast Joint Compatibility Branch and Bound for Feature Cloud Matching." *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016. https://doi.org/10.1109/iros.2016.7759281.

See Also

`matchScans` | `matchScansGrid`

Introduced in R2020b