Lidar Toolbox™

Reference
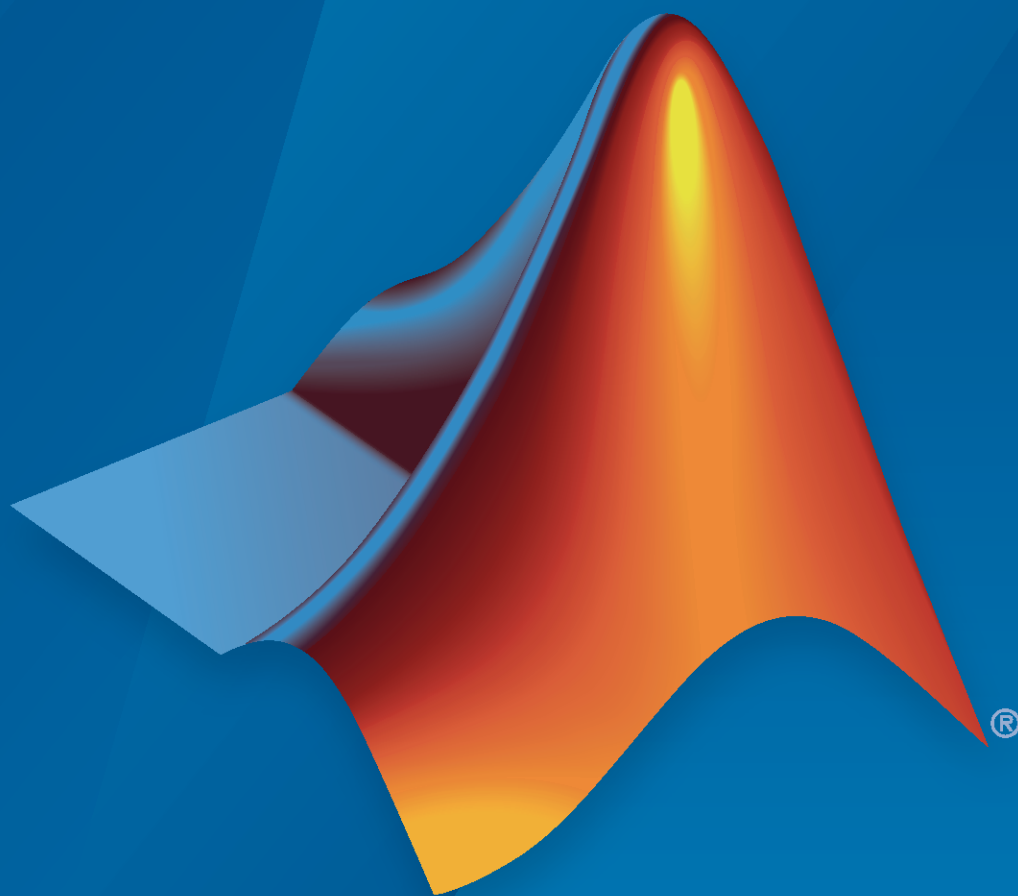
MATLAB®

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

Trademarks

Patents

Revision History

Contents

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), line, voxel 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".
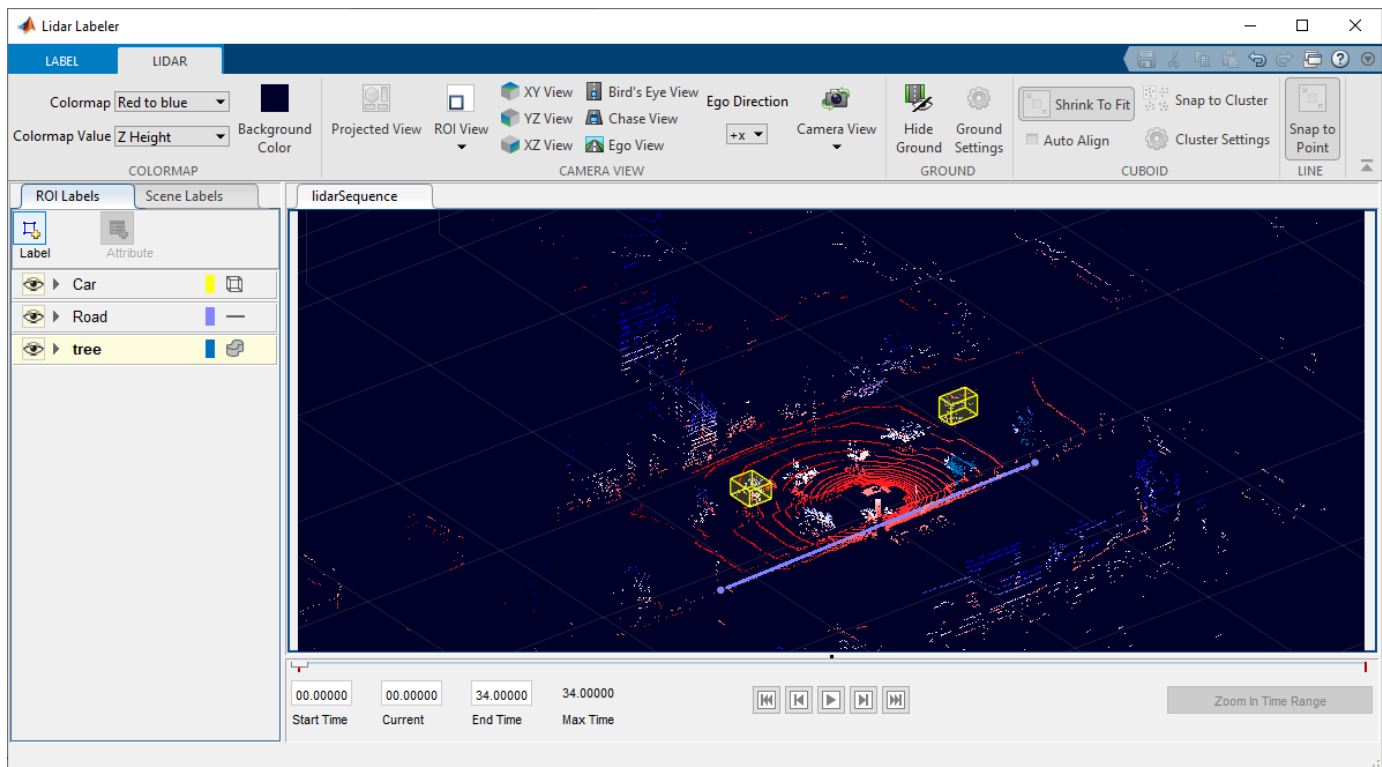
Open the Lidar Labeler App

- 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.

`lidarLabeler(gTruth)` opens the app and loads a `groundTruth` object .

Limitations

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

More About
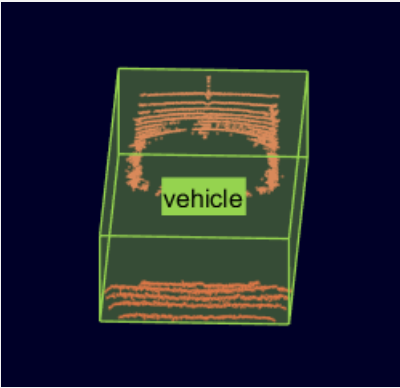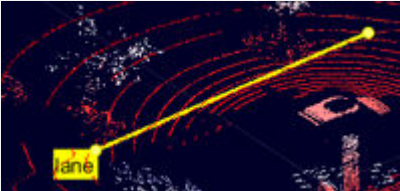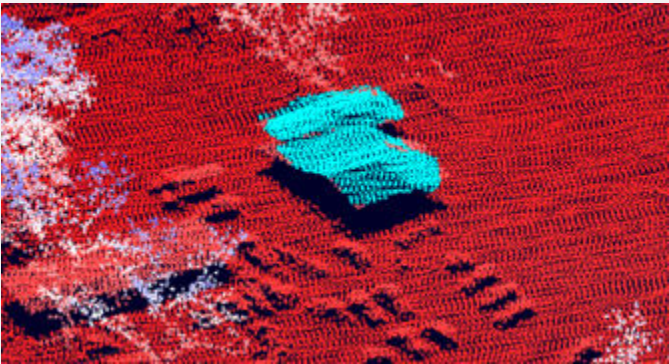
ROI Labels and Attributes

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.
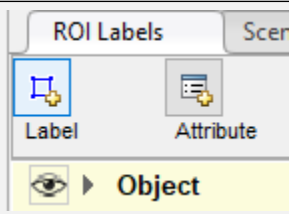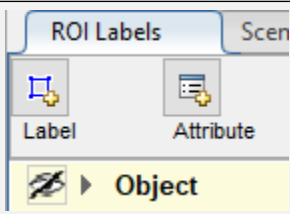
ROI Labels

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.	
Line	Draw line labels.	
Voxel	Draw voxel ROI labels around objects.	
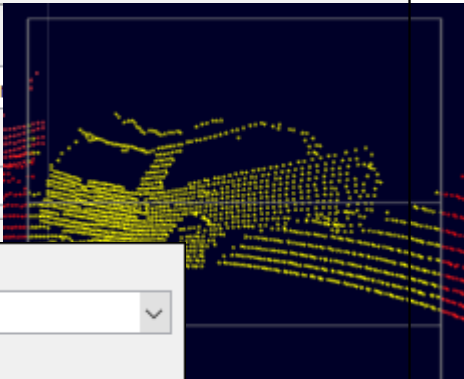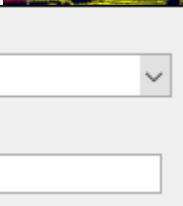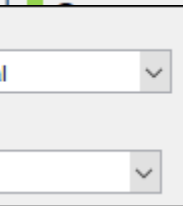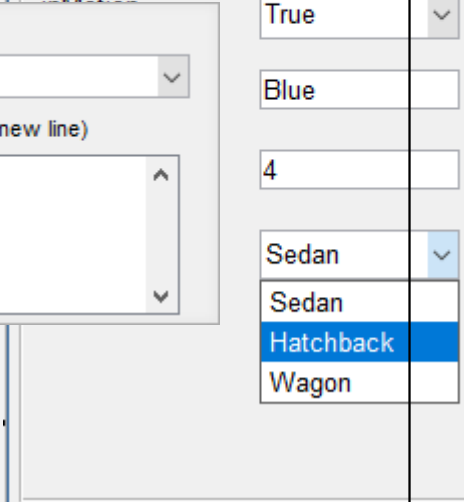
You can show or hide the labels by using the 👁 icon on the **ROI Labels** pane.

The 👁 appears only after you define a label. By default, the app displays all the labels. To hide a label, click on the 👁 icon alongside the label name. The app hides the corresponding label and displays the 🚫 icon.

Show Label	Hide Label

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	Attribute Name: numDoors / Num... Default Scalar Value (Optional): 4	
String	Attribute Name: color / String Default Value (Optional):	
Logical	Attribute Name: inMotion / Logical Default Value (Optional): True	Nissan / True
List	Attribute Name: carType / List List Items (Each item must appear on a new line): Sedan Hatchback Wagon	Blue / 4 / Sedan — Sedan / Hatchback / Wagon

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 built-in algorithms by following these steps:

1   Import 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 range. 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.

Version History
Introduced in R2020b

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"

Lidar Camera Calibrator

Interactively estimate rigid transformation between lidar sensor and camera

Description

The **Lidar Camera Calibrator** app enables you to interactively estimate the rigid transformation between a lidar sensor and a camera. The app performs calibration by reading the calibration images and point clouds captured by the user. The app reads point cloud data in the PLY and PCD formats, and images in any format supported by `imformats`.

Using the app, you can:

- Detect, extract, and visualize checkerboard features from image and point cloud data.
- Estimate the rigid transformation between the camera and the lidar using feature detection results.
- Use the calibration results to fuse data from both the sensors. You can visualize point cloud data projected onto the images, and color or grayscale information from the images fused with point cloud data.
- View the plotted calibration error metrics. You can remove outliers, using a threshold line, and recalibrate the remaining data.
- Define a region of interest (ROI) around the checkerboard to reduce the computation resources required by the transformation estimation process.
- Export the transformation and error metric data as workspace variables or MAT files. You can also create a MATLAB script for the entire workflow.
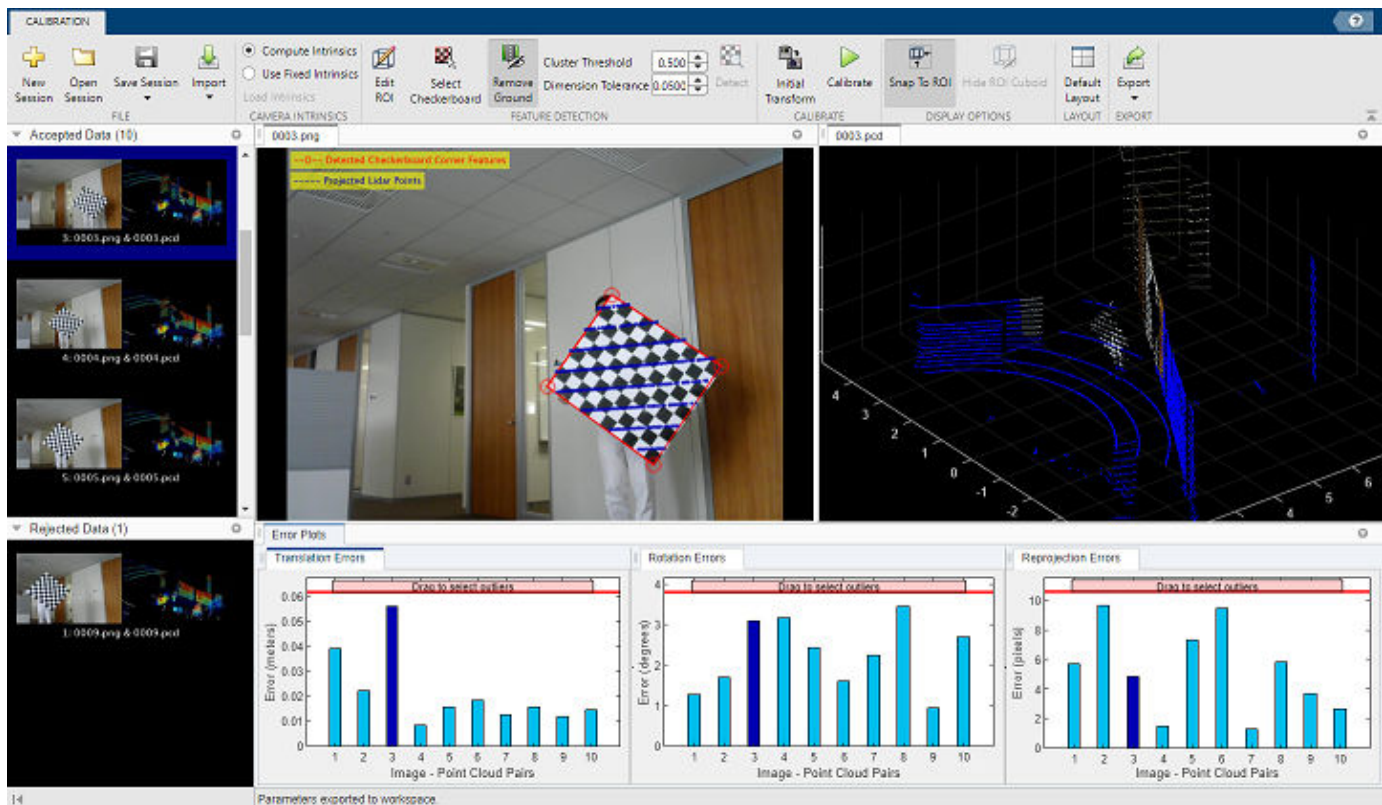
Open the Lidar Camera Calibrator App

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

Examples

Start and Load Parameters into Lidar Camera Calibrator App

Define paths to the image and point cloud files.

```
imageFilesPath = fullfile(toolboxdir('lidar'),'lidardata',...
    'lcc','vlp16','images');
pcFilesPath = fullfile(toolboxdir('lidar'),'lidardata',...
    'lcc','vlp16','pointCloud');
```

Load the checker size and padding values of the checkerboard.

```
checkerSize = 81; % millimeters
padding = [0 0 0 0]; % millimeters
```

Launch the app with these parameters.

```
lidarCameraCalibrator(imageFilesPath,pcFilesPath,checkerSize,padding)
```

- "Read Lidar and Camera Data from Rosbag File"

Programmatic Use

`lidarCameraCalibrator` opens a new session of the **Lidar Camera Calibrator** app.

`lidarCameraCalibrator(sessionFile)` opens the **Lidar Camera Calibrator** app and loads a previously saved app session, `sessionFile`.

`lidarCameraCalibrator(imageFilesPath,pcFilesPath,checkerSize,padding)` opens a new session of the app and loads the specified input data. The app reads image files from `imageFilesPath` and point cloud files from `pcFilesPath`. Both of these arguments must be valid folders containing images and point clouds, respectively. `checkerSize` is the square checker dimension of the checkerboard used in calibration and `padding` contains the padding values of the checkerboard, specified as a positive numeric scalar in millimeters.

Limitations

The **Lidar Camera Calibrator** app has these limitations:

- The point cloud axes tools and overall responsiveness are slow in Linux® machines.
- The script generated from **Export > Generate MATLAB Script** does not contain any manually selected checkerboard regions using the **Select Checkerboard** feature. In the script, the checkerboard region is detected in the specified ROI.
- After manually selected checkerboard regions using the **Select Checkerboard** feature, when the user comes back to the Calibration tab, you can see the selected points (highlighted in red) only while viewing the whole point cloud (i.e. when **SnapToROI** button is unselected).

Version History
Introduced in R2021a

See Also

Functions
`estimateCheckerboardCorners3d` | `detectRectangularPlanePoints` | `estimateLidarCameraTransform` | `projectLidarPointsOnImage` | `fuseCameraToLidar` | `bboxCameraToLidar` | `bboxLidarToCamera`

Topics
"Read Lidar and Camera Data from Rosbag File"
"What Is Lidar-Camera Calibration?"
"Calibration Guidelines"
"Get Started with Lidar Camera Calibrator"

Lidar Viewer

Visualize and analyze lidar data

Description

The **Lidar Viewer** app is a visualization, analysis, and preprocessing tool for lidar data. The app provides these features:

- Load and visualize point cloud data. The app can import `pointCloud` objects from the workspace and read point cloud data from PLY, PCAP, LAS, LAZ, PCD, rosbag files, or any custom source. You can export the processed point clouds as PCD or PLY files.

- Measure point cloud attributes such as distance, elevation, location, and volume.

- View and analyze point cloud data using the built-in camera views, color maps, and clustering options. You can also create and save custom camera views.

- Use built-in preprocessing algorithms for ground removal, organization, denoising, median filtering, cropping, and downsampling of lidar point cloud data.

- Import custom preprocessing algorithms for the lidar data. You can also create a user interface element to interactively tune the algorithm parameters.

- Export and reuse the preprocessing operations performed on a point cloud as a function script.
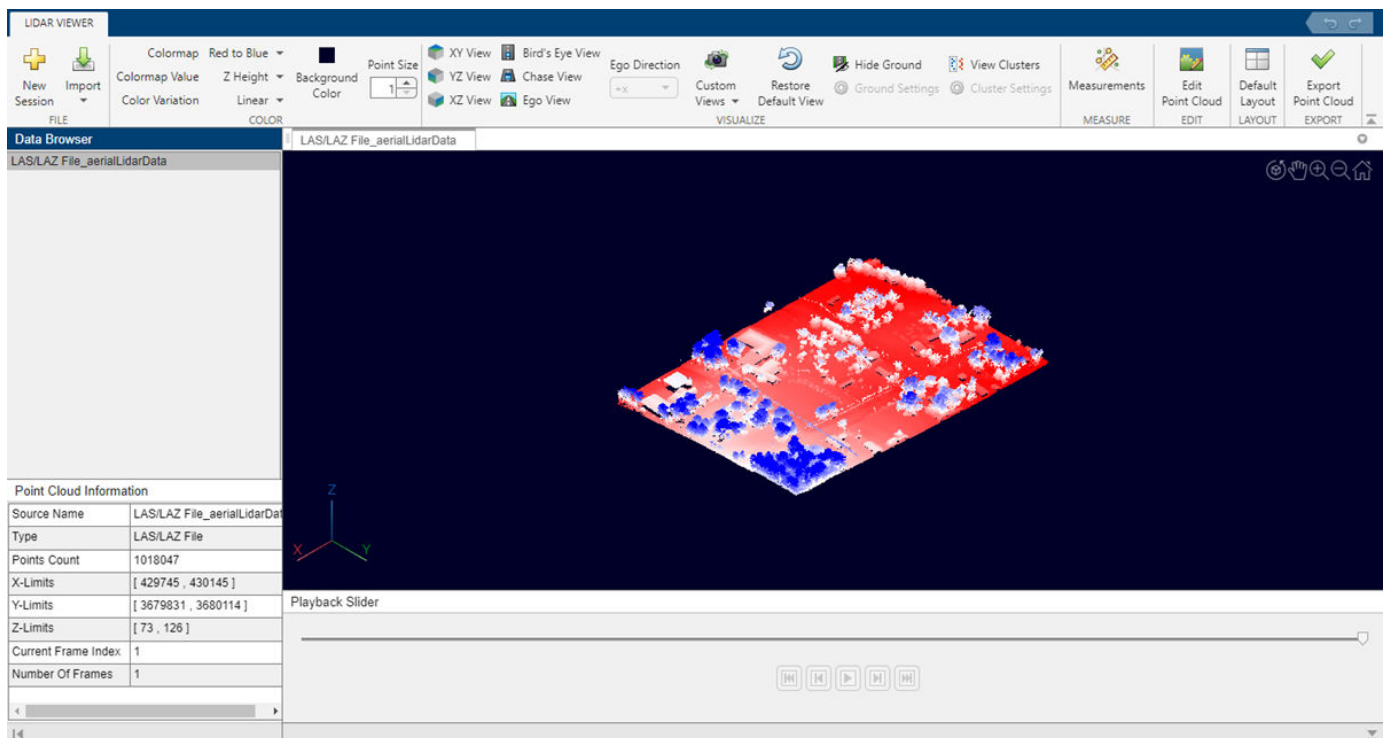
Open the Lidar Viewer App

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

Programmatic Use

`lidarViewer` opens a new session of the **Lidar Viewer** app.

Version History
Introduced in R2021b

See Also

Apps
Lidar Labeler | **Lidar Camera Calibrator**

Functions
pcshow | pointCloud | pcdownsample | pcmedian | pcdenoise | pcorganize | segmentGroundSMRF | pcfitplane | segmentGroundFromLidarData

Objects
pointCloud | lasFileReader

Topics
"Get Started with Lidar Viewer"
"Create Custom Preprocessing Workflow with Lidar Viewer"

Objects

pointCloudInputLayer

Point cloud input layer

Description

A point cloud input layer inputs 3-D point clouds to a network and applies data normalization. You can input any lidar data, such as 2-D lidar scans, to this layer, but the data must be a 2-D or a 3-D numeric array, as specified by the `InputSize` property.

Creation

Syntax

```
layer = pointCloudInputLayer(inputSize)
layer = pointCloudInputLayer(inputSize,Name=Value)
```

Description

`layer = pointCloudInputLayer(inputSize)` creates a point cloud input layer with the specified input size. The `inputSize` argument sets the `InputSize` property.

`layer = pointCloudInputLayer(inputSize,Name=Value)` specifies properties using one or more name-value arguments. For example, `Normalization="zscore"` applies z-score normalization to the layer.

Properties

3-D Point Cloud Input

InputSize — Size of the input
vector of positive integers

Size of the input data, specified as vector of positive integers. You can specify one of these options.

- For an unorganized point cloud, specify input size as two-element vector of the form [*M C*]. *M* is the number of points in the point cloud. *C* is the number of channels, which must be greater than or equal to 1.
- For an organized point cloud, specify input size as a three-element vector of the form [*M N C*]. *M* and *N* represent the number of rows and columns in the point cloud, respectively. *C* is the number of channels, which must be a positive integer greater than or equal to 1.

Normalization — Data normalization
'none' (default) | 'zerocenter' | 'zscore' | 'rescale-symmetric' | 'rescale-zero-one' | function handle

This property is read-only.

Data normalization to apply every time data is forward propagated through the input layer, specified as one of the following:

- `'zerocenter'` — Subtract the mean specified by Mean.
- `'zscore'` — Subtract the mean specified by Mean and divide by StandardDeviation.
- `'rescale-symmetric'` — Rescale the input to be in the range [-1, 1] using the minimum and maximum values specified by Min and Max, respectively.
- `'rescale-zero-one'` — Rescale the input to be in the range [0, 1] using the minimum and maximum values specified by Min and Max, respectively.
- `'none'` — Do not normalize the input data.
- function handle — Normalize the data using the specified function. The function must be of the form `Y = func(X)`, where `X` is the input data and the output `Y` is the normalized data.

Tip The software, by default, automatically calculates the normalization statistics at training time. To save time when training, specify the required statistics for normalization and set the `'ResetInputNormalization'` option in `trainingOptions` to `false`.

Data Types: `char` | `string`

NormalizationDimension — Normalization dimension
`'auto'` (default) | `'channel'` | `'element'` | `'all'`

Normalization dimension, specified as one of the following:

- `'auto'` – If the training option is `false` and you specify any of the normalization statistics (`Mean`, `StandardDeviation`, `Min`, or `Max`), then normalize over the dimensions matching the statistics. Otherwise, recalculate the statistics at training time and apply channel-wise normalization.
- `'channel'` – Channel-wise normalization.
- `'element'` – Element-wise normalization.
- `'all'` – Normalize all values using scalar statistics.

Data Types: `char` | `string`

Mean — Mean for zero-center and z-score normalization
`[]` (default) | matrix | 3-D array | numeric scalar

Mean for zero-center and z-score normalization, specified as a one of these options.

Point Cloud Format	Element-Wise Normalization	Channel-Wise Normalization
Unorganized point cloud	M-by-C numeric array. M is the number of points in the point cloud, and C is the number of channels.	1-by-C numeric array
Organized point cloud	M-by-N-by-C numeric array. M and N are the number of rows and columns in the point cloud, respectively, and C is the number of channels.	1-by-1-by-C numeric array

You can also specify this value as a scalar, in which case the function normalizes the entire input data set using the specified value.

> **Note** To specify the `Mean` property, `Normalization` must be `'zerocenter'` or `'zscore'`. If `Mean` is `[]`, the software calculates the mean during training.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

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

StandardDeviation — Standard deviation for z-score normalization
`[]` (default) | matrix | 3-D array | numeric scalar

Standard deviation for z-score normalization, specified as a one of these options.

Point Cloud Format	Element-Wise Normalization	Channel-Wise Normalization
Unorganized point cloud	M-by-C numeric array. M is the number of points in the point cloud, and C is the number of channels.	1-by-C numeric array
Organized point cloud	M-by-N-by-C numeric array. M and N are the number of rows and columns in the point cloud, respectively, and C is the number of channels.	1-by-1-by-C numeric array

You can also specify this value as a scalar, in which case the function normalizes the entire input data set using the specified value.

> **Note** To specify the `StandardDeviation` property, `Normalization` must be `'zscore'`. If `StandardDeviation` is `[]`, then the software calculates the standard deviation at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

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

Min — Minimum value for rescaling
`[]` (default) | matrix | 3-D array | numeric scalar

Minimum value for rescaling, specified as a one of these options.

Point Cloud Format	Element-Wise Normalization	Channel-Wise Normalization
Unorganized point cloud	M-by-C numeric array. M is the number of points in the point cloud, and C is the number of channels.	1-by-C numeric array

Point Cloud Format	Element-Wise Normalization	Channel-Wise Normalization
Organized point cloud	M-by-N-by-C numeric array. M and N are the number of rows and columns in the point cloud, respectively, and C is the number of channels.	1-by-1-by-C numeric array

You can also specify this value as a scalar, in which case the function normalizes the entire input data set using the specified value.

Note To specify the Min property, Normalization must be 'rescale-symmetric' or 'rescale-zero-one'. If Min is [], the software computes the minimum value at training time.

You can set this property when creating networks without training (for example, when assembling networks using assembleNetwork).

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

Max — Maximum value for rescaling
[] (default) | matrix | 3-D array | numeric scalar

Maximum value for rescaling, specified as one of these options.

Point Cloud Format	Element-Wise Normalization	Channel-Wise Normalization
Unorganized point cloud	M-by-C numeric array. M is the number of points in the point cloud, and C is the number of channels.	1-by-C numeric array
Organized point cloud	M-by-N-by-C numeric array. M and N are the number of rows and columns in the point cloud, respectively, and C is the number of channels.	1-by-1-by-C numeric array

You can also specify this value as a scalar, in which case the function normalizes the entire input data set using the specified value.

Note To specify the Max property, Normalization must be 'rescale-symmetric' or 'rescale-zero-one'. If Max is [], the software computes the maximum value at training time.

You can set this property when creating networks without training (for example, when assembling networks using assembleNetwork).

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

Layer

Name — Layer name
'' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with the name `''`.

Data Types: `char` | `string`

NumInputs — Number of inputs
0 (default)

This property is read-only.

Number of inputs of the layer. The layer has no inputs.

Data Types: `double`

InputNames — Input names
{} (default)

This property is read-only.

Input names of the layer. The layer has no inputs.

Data Types: `cell`

NumOutputs — Number of outputs
1 (default)

Number of outputs of the layer. The layer has one output.

Data Types: `double`

OutputNames — Output names
{'out'} (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Point Cloud Input Layer

Create a point cloud input layer using an unorganized point cloud with 1000 points 3 channels.

```
layer = pointCloudInputLayer([1000 3],Name="Input")

layer =
  PointCloudInputLayer with properties:

                    Name: 'Input'
               InputSize: [1000 3]

   Hyperparameters
           Normalization: 'none'
```

```
NormalizationDimension: 'auto'
```

Version History
Introduced in R2022b

See Also
trainNetwork | layerGraph | squeezesegv2Layers | pointnetplusLayers | pcsemanticseg

Topics
"Getting Started with Point Clouds Using Deep Learning"
"List of Deep Learning Layers" (Deep Learning Toolbox)
"Datastores for Deep Learning" (Deep Learning Toolbox)

removeDefects

Remove surface mesh defects

Syntax

```
removeDefects(mesh,"duplicate-vertices")
removeDefects(mesh,"duplicate-faces")
removeDefects(mesh,"unreferenced-vertices")
removeDefects(mesh,"degenerate-faces")
removeDefects(mesh,"nonmanifold-edges")
```

Description

`removeDefects(mesh,"duplicate-vertices")` removes duplicate vertices from the surface mesh `mesh`.

`removeDefects(mesh,"duplicate-faces")` removes duplicate faces from the surface mesh `mesh`.

`removeDefects(mesh,"unreferenced-vertices")` removes unreferenced vertices from the surface mesh `mesh`. Unreferenced vertices are those vertices that are not part of any face.

`removeDefects(mesh,"degenerate-faces")` removes degenerate faces from the surface mesh `mesh`. Degenerate faces are faces with an area of zero.

`removeDefects(mesh,"nonmanifold-edges")` removes nonmanifold edges from the surface mesh `mesh`.

Examples

Remove Unreferenced Vertices from Surface Mesh

Define mesh vertices and faces for the surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1];
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create and display the mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Original mesh")
```

Add new vertices to the mesh.

```
newVertices = [1, 2, 3; 4, 5, 6];
addVertices(mesh, newVertices)
```

Remove unreferenced vertices from the surface mesh.

```
removeDefects(mesh,"unreferenced-vertices")
```

Remove NonManifold Edges from Surface Mesh

Define mesh vertices and faces for the surface mesh.

```
vertices = [0 0 0; 0 0 1; 0 1 1; 0 0 2; 1 0.5 1];
faces = [1 2 3; 2 3 4; 2 3 5];
```

Create the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
```

Check if the mesh is edge-manifold.

```
allowBoundaryEdges = true;
edgeManifoldBefore = isEdgeManifold(mesh,allowBoundaryEdges)
```

```
edgeManifoldBefore = logical
   0
```

Remove nonmanifold edges from the surface mesh and check if the mesh is edge-manifold.

```
removeDefects(mesh,"nonmanifold-edges")
edgeManifoldAfter = isEdgeManifold(mesh,allowBoundaryEdges)
```

```
edgeManifoldAfter = logical
   1
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a surfaceMesh object.

Version History
Introduced in R2022b

See Also
surfaceMesh | isEdgeManifold | isVertexManifold | addVertices | addFaces | removeVertices | removeFaces

isWatertight

Check if surface mesh is watertight

Syntax

```
TF = isWatertight(mesh)
```

Description

`TF = isWatertight(mesh)` checks if the mesh is watertight. A mesh is watertight when it is edge-manifold and vertex-manifold, but not self-intersecting.

Examples

Check If Surface Mesh Is Watertight

Define mesh vertices and faces for the surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1];
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Input Mesh")
```

Check if the mesh is watertight.

```
TF = isWatertight(mesh)
```

```
TF = logical
   1
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a `surfaceMesh` object.

Output Arguments

TF — Surface mesh is watertight
0 | 1

Surface mesh is watertight, returned as a logical `0` (`false`) or `1` (`true`). The function returns `true` when the mesh is watertight. Otherwise, it returns `false`.

Version History
Introduced in R2022b

See Also
surfaceMesh | isEdgeManifold | isOrientable | isSelfIntersecting | isVertexManifold | isWatertight

isVertexManifold

Check if surface mesh is vertex-manifold

Syntax

```
TF = isVertexManifold(mesh)
```

Description

`TF = isVertexManifold(mesh)` checks if the surface mesh is vertex-manifold. A mesh is vertex-manifold if faces with a common vertex form an open or closed fan.

Closed fan   Open fan
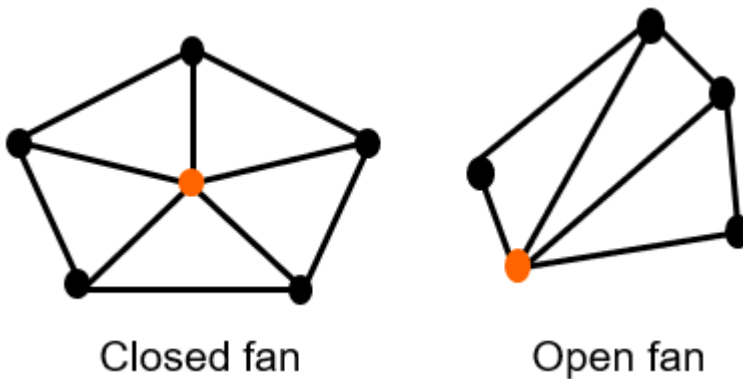
Examples

Check If Surface Mesh Is Vertex-Manifold

Define mesh vertices and faces for the surface mesh.

```
vertices = [0 0 0; 0 0 1; 0 1 1; 0 0 2; 1 0.5 1];
faces = [1 2 3; 2 3 4; 2 3 5];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Input Mesh")
```

Check if the mesh is vertex-manifold.

```
TF = isVertexManifold(mesh)
```

```
TF = logical
   1
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a `surfaceMesh` object.

Output Arguments

TF — Surface mesh is vertex-manifold
0 | 1

Surface mesh is vertex-manifold, returned as a logical `0` (`false`) or `1` (`true`). The function returns `true` when the mesh is vertex-manifold. Otherwise, it returns `false`.

Version History
Introduced in R2022b

See Also
surfaceMesh | isEdgeManifold | isOrientable | isSelfIntersecting | isVertexManifold | isWatertight

isSelfIntersecting

Check if surface mesh is self-intersecting

Syntax

```
TF = isSelfIntersecting(mesh)
```

Description

`TF = isSelfIntersecting(mesh)` checks if the surface mesh is self-intersecting. A mesh is self-intersecting if at least one of its face intersects another face.

Examples

Check If Surface Mesh Is Self-Intersecting

Define mesh vertices and faces for the mesh.

```
vertices =[0 0 0; 0 1 0; 1 0 0; 1 1 0; ...
          0.5 0.5 -1; 0 1 1; 1 0 1];
faces = [1 2 3; 2 3 4; 5 6 7];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Input Mesh")
```

Check if the mesh is self-intersecting.

```
TF = isSelfIntersecting(mesh)

TF = logical
   1
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a `surfaceMesh` object.

Output Arguments

TF — Surface mesh is self-intersecting
0 | 1

Surface mesh is self-intersecting, returned as a logical `0` (`false`) or `1` (`true`). The function returns `true` when the mesh is self-intersecting. Otherwise, it returns `false`.

Version History
Introduced in R2022b

See Also
surfaceMesh | isEdgeManifold | isOrientable | isSelfIntersecting | isVertexManifold | isWatertight

isOrientable

Check if surface mesh is orientable

Syntax

```
TF = isOrientable(mesh)
```

Description

`TF = isOrientable(mesh)` checks if the surface mesh is orientable. The orientation of a mesh is given by the cyclic order of its vertices. A mesh is orientable if all its face normals point outside.

Examples

Check If Surface Mesh Is Orientable

Read a surface mesh from an STL file.

```
fileName = fullfile(toolboxdir("lidar" ),"lidardata", ...
                    "surfaceMesh","mobius.stl");
mesh = readSurfaceMesh(fileName);
```

Display the surface mesh

```
surfaceMeshShow(mesh,Title="Input Mesh")
```

Check if the mesh is orientable.

```
TF = isOrientable(mesh)

TF = logical
   0
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a `surfaceMesh` object.

Output Arguments

TF — Surface mesh is orientable
0 | 1

Surface mesh is orientable, returned as a logical `0` (`false`) or `1` (`true`). The function returns `true` when the mesh is orientable. Otherwise, it returns `false`.

Version History
Introduced in R2020b

See Also
surfaceMesh | isEdgeManifold | isOrientable | isSelfIntersecting | isVertexManifold | isWatertight

isEdgeManifold

Check if surface mesh is edge-manifold

Syntax

```
TF = isEdgeManifold(mesh,allowBoundaryEdges)
```

Description

`TF = isEdgeManifold(mesh,allowBoundaryEdges)` checks if the surface mesh is edge-manifold. A mesh is edge-manifold if every edge of the mesh bounds either one or two faces.

Examples

Check If Surface Mesh Is Edge-Manifold

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Input Mesh")
```

Check if the mesh is edge-manifold.

```
allowBoundaryEdges = false;
TF = isEdgeManifold(mesh,allowBoundaryEdges);
disp(TF)

    1
```

Input Arguments

`mesh` — Surface mesh
surfaceMesh object

Surface mesh, specified as a `surfaceMesh` object.

`allowBoundaryEdges` — Allow boundary edges
true | false

Allow boundary edges, specified as a logical `true` or `false`. When this value is specified as `true`, the function ignores boundary edges while checking if the mesh is edge-manifold.

Data Types: `logical`

Output Arguments

TF — Surface mesh is edge-manifold
`0` | `1`

Surface mesh is edge-manifold, returned as a logical `0` (`false`) or `1` (`true`). The function returns `true` when the mesh is edge-manifold. Otherwise, it returns `false`.

The function also returns `false` when a mesh has boundary edges and `allowBoundaryEdges` is specified as `false`.

Version History
Introduced in R2022b

See Also
`surfaceMesh` | `isEdgeManifold` | `isOrientable` | `isSelfIntersecting` | `isVertexManifold` | `isWatertight`

subdivide

Subdivide surface mesh

Syntax

```
subdivide(mesh,"midpoint-split",numIterations)
subdivide(mesh,"loop",numIterations)
```

Description

subdivide(mesh,"midpoint-split",numIterations) subdivides the surface mesh mesh by using the midpoint-split method with the specified number of iterations. In this method, the function divides each face of the mesh into four faces in each iteration. New vertices lie at the midpoints of the edges of the original face.

subdivide(mesh,"loop",numIterations) subdivides the surface mesh by using a loop subdivision method with the specified number of iterations. In this methods, the function divides each triangular face into four by connecting the midpoints of the edges, then updates the new vertices as a weighted average of neighboring positions. The function divides each face into four faces in each iteration.

Examples

Subdivide Surface Mesh Using Midpoint-Split Method

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Original Mesh")
```

Subdivide the mesh using the midpoint-split method. Display the subdivided mesh.

```
numIterations = 4;
subdivide(mesh,"midpoint-split",numIterations);
surfaceMeshShow(mesh,Title="Subdivided Mesh",WireFrame=true)
```

Subdivide Surface Mesh Using Loop Method

Read a surface mesh from a PLY file.

```
fileName = fullfile(toolboxdir("lidar"),"lidardata", ...
            "surfaceMesh","sphere.ply");
mesh = readSurfaceMesh(fileName);
```

Display the surface mesh.

```
surfaceMeshShow(mesh,Title="Original Mesh",WireFrame=true)
```

Subdivide the mesh using the loop method, and display the result.

```
numIterations = 1;
subdivide(mesh,"loop",numIterations)
surfaceMeshShow(mesh,Title="Subdivided mesh",WireFrame=true)
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a surfaceMesh object.

numIterations — Number of iterations for subdivision method
positive integer

Number of iterations for the subdivision method, specified as a positive integer. At each iteration, the function divides each face of the mesh into four faces.

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

Version History
Introduced in R2022b

See Also
surfaceMesh | rotate | translate | transform | scale | crop | simplify | subdivide

simplify

Simplify surface mesh

Syntax

```
simplify(mesh)
simplify(mesh,Name=Value)
```

Description

`simplify(mesh)` simplifies the surface mesh `mesh` by using quadric decimation.

`simplify(mesh,Name=Value)` specifies options using one or more name-value arguments. For example, `SimplificationMethod="vertex-clustering"` simplifies the surface mesh by using the vertex-clustering method.

Examples

Simplify Surface Mesh Using Quadric Decimation

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Original Mesh")
```

Subdivide the mesh using the midpoint-split method, and display the subdivided mesh.

```
numIterations = 4;
subdivide(mesh,"midpoint-split",numIterations)
surfaceMeshShow(mesh,Title="Subdivided Mesh",WireFrame=true)
```

Simplify the surface mesh by using the quadric-decimation method.

```
 simplify(mesh,SimplificationMethod="quadric-decimation", ...
             TargetNumFaces=30)
```

Remove any unreferenced vertices, and display the simplified mesh.

```
removeDefects(mesh,"unreferenced-vertices")
surfaceMeshShow(mesh,Title="Simplified Mesh",WireFrame=true)
```

Simplify Surface Mesh Using Vertex Clustering

Read a surface mesh from an STL file.

```
fileName = fullfile(toolboxdir("lidar" ),"lidardata", ...
                  "surfaceMesh","mobius.stl");
mesh = readSurfaceMesh(fileName);
```

Display the surface mesh.

```
surfaceMeshShow(mesh,Title="Original Mesh",WireFrame=true)
```

Simplify the surface mesh by using the vertex-clustering method.

```
simplify(mesh,SimplificationMethod="vertex-clustering", ...
         VoxelSize=0.15,MergeMethod="Quadric")
```

Remove any unreferenced vertices, and display the simplified mesh.

```
removeDefects(mesh,"unreferenced-vertices")
surfaceMeshShow(mesh,Title="Simplified Mesh",WireFrame=true)
```

Input Arguments

`mesh` — Surface mesh
surfaceMesh object

Surface mesh, specified as a `surfaceMesh` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `simplify(mesh,SimplificationMethod="vertex-clustering")` simplifies the surface mesh by using the vertex-clustering method.

`SimplificationMethod` — Surface mesh simplification method
"quadric-decimation" (default) | "vertex-clustering"

Surface mesh simplification method, specified as `"quadric-decimation"` or `"vertex-clustering"`. For more information on these simplification methods, see the "Simplification Methods" on page 2-25 section.

Data Types: char | string

`TargetNumFaces` — Number of triangular faces in simplified mesh
positive integer

Number of triangular faces in the simplified mesh, specified as a positive integer. By default, the function computes this values as `0.2` times the number of faces in the input mesh.

Note This argument is applicable only when you specify `SimplificationMethod` as `"quadric-decimation"`.

Data Types: `single` | `double` | `int32` | `uint32`

MaxError — Maximum allowed error when vertex is merged
`Inf` (default) | positive scalar

Maximum allowed error when a vertex is merged, specified as a positive scalar.

Note This argument is applicable only when you specify `SimplificationMethod` as `"quadric-decimation"`.

Data Types: `single` | `double`

BoundaryWeight — Weight for edge vertices
`1.0` (default) | positive scalar

Weight for the edge vertices, specified as a positive scalar. The function uses this value to preserve the mesh boundaries.

Note This argument is applicable only when you specify `SimplificationMethod` as `"quadric-decimation"`.

Data Types: `single` | `double`

VoxelSize — Voxel size for vertex clustering
`0.01` (default) | positive scalar

Voxel size for vertex clustering, specified as a positive scalar. The function pools all the vertices within the specified voxel size to form clusters.

Note This argument is applicable only when you specify `SimplificationMethod` as `"voxel-clustering"`.

Data Types: `single` | `double`

MergeMethod — Vertex merging method
`"Average"` (default) | `"Quadric"`

Vertex merging method, specified as `"Average"` or `"Quadric"`. When the value is specified as `"Average"`, the function computes an average value of all vertices in a voxel. When the value is specified as `"Quadric"`, the function minimizes the distance between the adjacent planes to merge the vertices.

Note This argument is applicable only when you specify `SimplificationMethod` as `"voxel-clustering"`.

Data Types: `char` | `string`

Algorithms

Quadric decimation is a method that uses iterative contractions of vertex pairs to simplify a mesh, and maintains error approximation for all vertices using quadric matrices. The method consists of these steps.

1   Compute quadric matrices for all vertices, and select vertex pairs to merge.

2   Compute the contraction target vertex for each pair.

3   Iteratively minimize the error of the target vertices to construct the simplified mesh.

Vertex clustering is a method that uses bounding boxes to divide the mesh into voxels. The function clusters the vertices in each voxel into a single vertex and then updates the mesh faces accordingly.

Version History
Introduced in R2022b

See Also
`surfaceMesh` | `rotate` | `translate` | `transform` | `scale` | `crop` | `simplify` | `subdivide`

crop

Crop surface mesh

Syntax

```
crop(mesh,bbox)
```

Description

`crop(mesh,bbox)`crops the surface mesh `mesh` to the region specified by the 3-D bounding box `bbox`.

Examples

Crop Surface Mesh Using a Bounding Box

Read a surface mesh from a PLY file into the workspace.

```
fileName = fullfile(toolboxdir("lidar"),"lidardata", ...
        "surfaceMesh","sphere.ply");
sphereMesh = readSurfaceMesh(fileName);
```

Display the surface mesh.

```
removeDefects(sphereMesh,"unreferenced-vertices")
surfaceMeshShow(sphereMesh,Title="Original Mesh")
```

Define a bounding box to crop the sphere mesh to its lower half.

```
bbox3d = [-1 1 -1 1 -1 0];
crop(sphereMesh,bbox3d)
```

Display the cropped mesh.

```
removeDefects(sphereMesh,"unreferenced-vertices")
surfaceMeshShow(sphereMesh, Title="Cropped mesh")
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a `surfaceMesh` object.

bbox — 3-D bounding box dimensions
six-element vector

3-D bounding box dimensions, specified as a six-element vector. The vector is of the form [*minX maxX minY maxY minZ maxZ*], specifying the minimum and maximum limits of the box along the *x-*,*y-*, *z*-directions.

Version History
Introduced in R2022b

See Also
surfaceMesh | translate | rotate | transform | scale | simplify | subdivide

computeNormals

Compute unit normals for mesh vertices and faces

Syntax

```
computeNormals(mesh)
computeNormals(mesh,"vertices")
computeNormals(mesh,"faces")
```

Description

`computeNormals(mesh)` computes the unit normal vector for the vertices and the faces of the surface mesh `mesh`. The function overwrites the existing vertex and face normal vectors.

`computeNormals(mesh,"vertices")` computes the unit normal vector for only the mesh vertices. A vertex normal is the average of the face normal vectors of all the faces that share the vertex. The function uses the existing face normal vectors to compute the vertex normal vectors.

`computeNormals(mesh,"faces")` computes the unit normal vectors for only the mesh faces.

Examples

Compute Normal Vectors for Mesh Vertices and Faces

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
```

Compute the unit normal vectors for the mesh vertices and faces.

```
computeNormals(mesh);
mesh
```

```
mesh =
  surfaceMesh with properties:

          Vertices: [8x3 double]
             Faces: [12x3 int32]
     VertexNormals: [8x3 double]
      VertexColors: []
       FaceNormals: [12x3 double]
```

```
      FaceColors: []
    NumVertices: 8
       NumFaces: 12
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a `surfaceMesh` object.

Version History
Introduced in R2022b

See Also
surfaceMesh | vertexCenter | rotate | transform | scale | crop | simplify | subdivide

scale

Scale vertices of surface mesh

Syntax

```
scale(mesh,scalingFactor)
scale(mesh,scalingFactor,pivot)
```

Description

`scale(mesh,scalingFactor)` scales the vertices of the surface mesh `mesh` along the *x-*, *y-*, *z*-axes by the specified scaling factor `scalingFactor`. The function uses the mesh origin as the pivot point.

`scale(mesh,scalingFactor,pivot)` scales the mesh vertices along the *x-*, *y-*, *z*-axes about the specified pivot point.

Examples

Scale Surface Mesh

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Original Mesh")
```

Scale the vertices of the surface mesh by a factor of two, and display the scaled mesh.

```
scalingFactor = 2;
scale(mesh,scalingFactor)
surfaceMeshShow(mesh,Title="Scaled Mesh")
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a `surfaceMesh` object.

scalingFactor — Scaling factor for mesh
positive scalar

Scaling factor for the surface mesh, specified as a positive scalar.

Data Types: `single` | `double`

pivot — Pivot point for scaling
three-element vector

Pivot point for scaling the surface mesh, specified as a three-element vector of the form [x y z]. The values of the vector define the coordinates of the pivot point.

Version History
Introduced in R2022b

See Also
`surfaceMesh` | `rotate` | `transform` | `translate` | `crop` | `simplify` | `subdivide`

vertexCenter

Find vertex center of surface mesh

Syntax

```
vertexCenter(mesh)
```

Description

`vertexCenter(mesh)` finds the center of the mesh vertices of the surface mesh `mesh`. The vertex center is the mean value of all vertices.

Examples

Compute Vertex Center of Surface Mesh

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Cuboid Mesh")
```

Find the vertex center of the surface mesh and display its coordinates.

```
center = vertexCenter(mesh);
disp(center)

     0     0     0
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a `surfaceMesh` object.

Version History
Introduced in R2022b

See Also

surfaceMesh | computeNormals | rotate | scale | crop | simplify | subdivide

transform

Apply rigid transformation to surface mesh

Syntax

```
transform(mesh,tform)
```

Description

transform(mesh,tform) applies the rigid 3-D transformation specified by tform to the surface mesh mesh.

Examples

Apply Rigid 3-D Transformation to Surface Mesh

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Original Mesh")
```

Define rotation and translation values, then use them to generate a transformation matrix.

```
theta = 30;
rotationMat = [cosd(theta)  sind(theta) 0;
               -sind(theta) cosd(theta) 0;
                    0            0       1];
translationVector = [2 0 0];
tform = rigidtform3d(rotationMat,translationVector);
```

Apply the rigid transformation and visualize the output.

```
transform(mesh,tform);
surfaceMeshShow(mesh,Title="Transformed Mesh")
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a `surfaceMesh` object.

`tform` — Rigid 3-D transformation matrix
`rigidtform3d` object

Rigid 3-D transformation matrix, specified as a `rigidtform3d` object.

Version History
Introduced in R2022b

See Also
`surfaceMesh` | `translate` | `rotate` | `scale` | `crop` | `simplify` | `subdivide`

rotate

Rotate surface mesh

Syntax

```
rotate(mesh,"rotmat",rotMatrix)
rotate(mesh,"euler",E,rotSequence)
rotate(mesh,"quaternion",quat)
rotate( ___ ,pivot)
```

Description

`rotate(mesh,"rotmat",rotMatrix)` rotates the surface mesh `mesh` around its origin by the values specified in the rotation matrix `rotMatrix`.

`rotate(mesh,"euler",E,rotSequence)` rotates the surface mesh `mesh` around its origin using the Euler angles specified by the Euler vector `E` in the sequence of rotation `rotSequence`.

`rotate(mesh,"quaternion",quat)` rotates the surface mesh `mesh` around its origin using the values specified by the quaternion array `quat`.

`rotate( ___ ,pivot)` specifies a pivot point around which to rotate the surface mesh, in addition to any combination of input arguments from the previous syntaxes.

Examples

Rotate Surface Mesh About *z*-Axis Using Euler Angles

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Original Mesh")
```

Define the Euler angles and rotation sequence for rotating the surface mesh about the *z*-axis by 30 degrees.

```
eulerAngles = [30 0 0];
rotSequence = "ZYX";
```

Rotate the surface mesh and display the output.

```
rotate(mesh,"euler",eulerAngles,rotSequence)
surfaceMeshShow(mesh,Title="Rotated Mesh")
```

Rotate Surface Mesh Using Quaternion Array

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Original Mesh")
```

Define the quaternion array for rotatating the surface mesh about the *y*-axis by 60 degrees,

```
eulerAngles = [60 0 0];
quat = quaternion(eulerAngles,'eulerd','YZX','point');
```

Rotate surface mesh and display the output.

```
rotate(mesh,"quaternion",quat);
surfaceMeshShow(mesh,Title="Rotated Mesh")
```

Rotate Surface Mesh About Pivot Point

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Original Mesh")
```

Define the pivot point for rotation.

```
pivot = [-1 -1 0];
```

Define a rotation matrix for rotating the surface mesh by 45 degrees about the *z*-axis.

```
rotMatrix = [cosd(45)  -sind(45)  0;
             sind(45)   cosd(45)  0;
                 0          0      1];
```

Rotate the surface mesh about the pivot point and display the rotated mesh.

```
rotate(mesh,"rotmat",rotMatrix,pivot);
surfaceMeshShow(mesh,Title="Rotated Mesh");
```

Input Arguments

`mesh` — Surface mesh
`surfaceMesh` object

Surface mesh, specified as a `surfaceMesh` object.

`rotMatrix` — Rotation matrix
3-by-3 matrix

Rotation matrix, specified as a 3-by-3 matrix.

E — Euler angles
three-element vector

Euler angles, specified as a three-element vector. The values of the vector are in degrees.

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

`rotSequence` — Sequence of rotation
character vector | string scalar

Sequence of rotation, specified as a character vector or string scalar. You can specify one of these options.

- `"YZY"`
- `"YXY"`
- `"ZYZ"`
- `"ZXZ"`
- `"XYX"`
- `"XZX"`
- `"XYZ"`
- `"YZX"`
- `"ZXY"`
- `"XZY"`
- `"ZYX"`
- `"YXZ"`

For example, when you specify the sequence as `"YZX"`, the function first rotates the mesh about the *y*-axis, then the about new *z*-axis, followed by the new *x*-axis. E defines the angles of rotation.

Data Types: `char` | `string`

quat — Quaternion array
quaternion object

Quaternion array for 3-D axes rotation, specified as a `quaternion` object.

pivot — Pivot point
three-element vector

Pivot point about which to rotate the mesh, specified as a three-element vector. The values of the vector specify the *x*-, *y*-, and *z*-coordinates of the pivot point. The function first shifts the origin to the pivot point, then rotates the mesh, and lastly shifts the origin back to the original position.

Version History
Introduced in R2022b

See Also
surfaceMesh | translate | transform | scale | crop | simplify | subdivide

translate

Translate surface mesh

Syntax

```
translate(mesh,translationVector)
translate(mesh,translationVector,pivot)
```

Description

`translate(mesh,translationVector)` translates the surface mesh, with respect to the world origin, by the values specified in `translationVector` along the *x-*, *y-*, *z-* axes.

`translate(mesh,translationVector,pivot)` translates the mesh with respect to the specified pivot point.

Examples

Translate Surface Mesh

Define mesh vertices for a surface mesh.

```
vertices = [3 1 3; 3 3 3; 1 3 3; 1 1 3; ...
            3 1 1; 3 3 1; 1 3 1; 1 1 1];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
```

Define the translation vector, using the mesh center as the pivot point for translation.

```
translationVec = [1 2 3];
pivot = vertexCenter(mesh);
```

Translate the surface mesh with respect to its center.

```
translate(mesh,translationVec,pivot)
```

Display the translated mesh vertices.

```
 mesh.Vertices
```

```
ans = 8×3

     2     1     4
     2     3     4
     0     3     4
```

```
0     1     4
2     1     2
2     3     2
0     3     2
0     1     2
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a surfaceMesh object.

translationVector — Translation along *x*-, *y*-, and *z*-axes
three-element vector

Translation along *x*-, *y*-, *z*-axes, specified as a three-element vector of the form [*x y z*]. If you do not specify a pivot point, this translation is with respect to the world origin.

pivot — Pivot point
three-element vector

Pivot point coordinates, specified as a three-element vector of the form [*x y z*]. The function translates the mesh with respect to this point.

Version History
Introduced in R2022b

See Also
surfaceMesh | rotate | transform | scale | crop | simplify | subdivide

removeFaces

Remove faces from surface mesh

Syntax

```
removeFaces(mesh,faceIDs)
removeFaces(mesh,faceMask)
```

Description

removeFaces(mesh,faceIDs) removes the faces specified by the face IDs `faceIDs` from the `surfaceMesh` object `mesh`. The function also removes the corresponding normal vectors and colors from the mesh.

removeFaces(mesh,faceMask) remove the faces specified by the binary face mask `faceMask` from the `surfaceMesh` object `mesh`.

Examples

Remove Faces from Surface Mesh

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1; ...
            0 0 2; 0 0 -2];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7; ...
         1 4 9; 2 3 9; 5 8 10; 6 7 10];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Original Mesh")
```

Remove the faces 13th and 14th from the mesh by specifying their face IDs, remove the resulting unreferenced vertices from the mesh. Visualize the output.

```
faceIDs = [13 14];
removeFaces(mesh,faceIDs)
removeDefects(mesh,"unreferenced-vertices")
surfaceMeshShow(mesh,Title="Mesh After Removing Faces")
```

Remove Faces from Surface Mesh Using Binary Mask

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1; ...
            0 0 2; 0 0 -2];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7; ...
         1 4 9; 2 3 9; 5 8 10; 6 7 10];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Original Mesh")
```

Mark 13,14 vertices as `true` and remaining vertices as `false` to generate a binary mask.

```
faceMask = false(1,mesh.NumFaces);
faceMask([13 14]) = true;
```

Remove faces 13 and 14 face from the mesh by using the `removeFaces` function, and remove the resulting unreferenced vertices. Visualize the output.

```
removeFaces(mesh,faceMask);
removeDefects(mesh,"unreferenced-vertices")
surfaceMeshShow(mesh,Title="Mesh After Removing Faces")
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a `surfaceMesh` object.

faceIDs — Face IDs
n-element vector

Face IDs, specified as an *n*-element vector. Each face ID is a unique identifier for a face in the mesh and is equal to the row number of the face in the `Faces` property of the `surfaceMesh` object. *n* is the number of faces to remove from the mesh.

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

faceMask — Binary mask with mesh faces
N-element logical vector

Binary mask with mesh faces, specified as an *N*-element logical vector. *N* is the total number of mesh faces. If the value of an element is `true`, the function removes the corresponding face from the mesh. If an element is `false`, the function does not remove the corresponding face from the mesh.

Version History
Introduced in R2022b

See Also

addFaces | addVertices | removeVertices | surfaceMesh | pc2surfacemesh | readSurfaceMesh | writeSurfaceMesh | surfaceMeshShow

removeVertices

Remove vertices from surface mesh

Syntax

```
removeVertices(mesh,vertexIDs)
removeVertices(mesh,vertexMask)
```

Description

removeVertices(mesh,vertexIDs) removes the vertices specified by the vertex IDs `vertexIDs` from the `surfaceMesh` object `mesh`. The function also removes the corresponding normal vectors and colors from the mesh.

removeVertices(mesh,vertexMask) removes the vertices specified by the binary vertex mask `vertexMask` from the `surfaceMesh` object `mesh`. The function also removes the corresponding normal vectors and colors from the mesh.

Examples

Remove Vertices from Surface Mesh

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1; ...
            0 0 2; 0 0 -2];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7; ...
         1 4 9; 2 3 9; 5 8 10; 6 7 10];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Original Mesh")
```

Remove the 9th and 10th vertices of the surface mesh. Display the updated mesh.

```
vertexIDs = [9 10];
removeVertices(mesh,vertexIDs)
surfaceMeshShow(mesh,Title="Mesh After Removing Vertices")
```

Remove Vertices from Surface Mesh Using Binary Mask

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1; ...
            0 0 2; 0 0 -2];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7; ...
         1 4 9; 2 3 9; 5 8 10; 6 7 10];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Original Mesh")
```

Generate a binary mask by marking the 9th and 10th vertices as `true` and remaining vertices as `false`.

```
vertexMask = false(1,mesh.NumVertices);
vertexMask([9 10]) = true;
```

Remove vertices `9` and `10` by using the `removeVertices` function, and display the results.

```
removeVertices(mesh,vertexMask)
surfaceMeshShow(mesh,Title="Mesh After Removing Vertices")
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a `surfaceMesh` object.

vertexIDs — Vertex IDs
m-element vector

Vertex IDs, specified as an *m*-element vector. Each vertex ID is a unique identifier for a vertex in the mesh and is equal to the row number of the vertex in the `Vertices` property of the `surfaceMesh` object. *m* is the number of vertices to remove from the mesh.

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

vertexMask — Binary mask with mesh vertices
M-element logical vector

Binary mask with mesh vertices, specified as an *M*-element logical vector. *M* is the total number of mesh vertices in the mesh. If the value of an element is `true`, the function removes the vertex with the corresponding vertex ID from the mesh. If an element is `false`, the function does not remove the corresponding vertex from the mesh.

Version History
Introduced in R2022b

See Also

addVertices | addFaces | removeFaces | surfaceMesh | pc2surfacemesh | readSurfaceMesh | writeSurfaceMesh | surfaceMeshShow

addFaces

Add faces to surface mesh

Syntax

```
addFaces(mesh,faces)
addFaces( ___ ,Name=Value)
```

Description

addFaces(mesh,faces) adds the specified faces to the surfaceMesh object, mesh.

addFaces(___ ,Name=Value) specifies options using one or more name-value arguments in addition to the previous syntax. For example, addFaces(mesh,faces,FaceNormals=[1 2 0; 1 1 1; 0 2 2; 4 3 0; 2 1 2; 5 5 0]) specifies the face normals for the surface mesh.

Examples

Add Vertices and Faces to Surface Mesh

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Original Mesh")
```

Add a new vertex to the surface mesh.

```
addVertices(mesh,[0 0 2])
```

Add new faces to surface mesh and display the updated mesh.

```
faces = [1 4 9; 2 3 9];
addFaces(mesh,faces);
surfaceMeshShow(mesh,Title="Modified Mesh")
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a `surfaceMesh` object.

faces — Mesh triangular faces
N-by-3 array

Mesh triangular faces, specified as an *N*-by-3 matrix. Each row of the array is in the form [V_1 V_2 V_3], specifying the vertex IDs of the vertices which define the triangular face. *N* is the total number of faces to add to the mesh.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `addFaces(mesh,faces,FaceNormals=[1 2 0; 1 1 1; 0 2 2; 4 3 0; 2 1 2; 5 5 0])`

FaceNormals — Normals for mesh faces
N-by-3 matrix

Normal vectors for the mesh faces, specified as an *N*-by-3 matrix. Each row of the matrix of in the form [`x y z`], specifying the normal for a face. *N* is the total number of faces to add to the mesh.

Data Types: `single` | `double`

FaceColors — Colors for mesh faces
N-by-3 matrix

Color values for the mesh faces, specified as an *N*-by-3 matrix. Each row of the matrix is of the form [`R G B`], specifying the color value for a face. Each value must be in the range [`0,1`]. *N* is the total number of faces to add to the mesh.

Data Types: `single` | `double`

Version History
Introduced in R2022b

See Also
`removeFaces` | `addVertices` | `removeVertices` | `surfaceMesh` | `pc2surfacemesh` | `readSurfaceMesh` | `writeSurfaceMesh` | `surfaceMeshShow`

addVertices

Add vertices to surface mesh

Syntax

```
addVertices(mesh,vertices)
addVertices( ___ ,Name=Value)
```

Description

addVertices(mesh,vertices) adds the specified vertices to the surfaceMesh object mesh.

addVertices(___ ,Name=Value) specifies options using one or more name-value arguments in addition to the input arguments from the previous syntax. For example, VertexNormals=[8 -4 4; 4 4 8; -6 6 3; -3 -6 6; 3 -6 -6; 6 6 -3] specifies the normal vectors for the mesh vertices.

Examples

Add Vertices and Faces to Surface Mesh

Define mesh vertices for a surface mesh.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh,Title="Original Mesh")
```

Add a new vertex to the surface mesh.

```
addVertices(mesh,[0 0 2])
```

Add new faces to surface mesh and display the updated mesh.

```
faces = [1 4 9; 2 3 9];
addFaces(mesh,faces);
surfaceMeshShow(mesh,Title="Modified Mesh")
```

Input Arguments

mesh — Surface mesh
surfaceMesh object

Surface mesh, specified as a `surfaceMesh` object.

`vertices` — Mesh vertices
M-by-3 matrix

Mesh vertices, specified as an *M*-by-3 matrix. Each row of the matrix is of the form [x y z], specifying the coordinates of a vertex. Each vertex has a vertex ID equal to $N + M_{row}$, where *N* is the ID of the last vertex in `mesh` and M_{row} is the row number of the vertex in `vertices`. *M* is the total number of vertices to add to the mesh.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `mesh(vertices,faces,VertexNormals=[8 -4 4; 4 4 8; -6 6 3; -3 -6 6; 3 -6 -6; 6 6 -3])` specifies the normal vectors for the mesh vertices.

`VertexNormals` — Normals for mesh vertices
M-by-3 matrix

Normal vectors for the mesh vertices, specified as an *M*-by-3 matrix. Each row of the matrix is of the form [x y z], specifying the normal vector for a vertex. *M* is the total number of vertices to add to the mesh.

Data Types: `single` | `double`

`VertexColors` — Colors for mesh vertices
M-by-3 matrix

Color values for the mesh vertices, specified as an *M*-by-3 matrix. Each row of the matrix is of the form [R G B], specifying the color value for a vertex. Each values must be in the range [0,1]. *M* is the total number of vertices to add to the mesh.

Data Types: `single` | `double`

Version History
Introduced in R2022b

See Also
addFaces | removeVertices | removeFaces | surfaceMesh | pc2surfacemesh | readSurfaceMesh | writeSurfaceMesh | surfaceMeshShow

surfaceMesh

Create surface mesh

Description

A `surfaceMesh` object creates and stores a surface mesh. A surface mesh represents a geometric surface and consists of vertices, faces, and edges. Using the `surfaceMesh` object, you can:

- Add and remove mesh vertices and faces
- Perform geometric operations, such as rotate, translate, transform, and scale
- Compute mesh normals
- Crop, simplify, and subdivide a mesh
- Check mesh properties such as whether it is self-intersecting, watertight, or orientable
- Remove degenerate and unreferenced vertices and faces

Creation

Syntax

```
mesh = surfaceMesh(vertices,faces)
mesh = surfaceMesh( ___ ,Name=Value)
```

Description

`mesh = surfaceMesh(vertices,faces)` creates a `surfaceMesh` object with the specified vertices and faces.

`mesh = surfaceMesh( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to the arguments from the previous syntax. For example, `VertexNormals=[8 -4 4; 4 4 8; -6 6 3; -3 -6 6; 3 -6 -6; 6 6 -3]` specifies the normal vectors for the mesh vertices.

Properties

Vertices — Mesh vertices
M-by-3 matrix

Mesh vertices, specified as an *M*-by-3 matrix. Each row of the matrix is of the form [x y z], specifying the coordinates of a vertex. Each vertex has a vertex ID equal to its row number in the matrix. *M* is the total number of vertices in the mesh.

The `vertices` argument sets this property at object creation.

Data Types: `single` | `double`

Faces — Mesh triangular faces
N-by-3 matrix

Mesh triangular faces, specified as an *N*-by-3 matrix. Each row of the matrix is in the form [V_1 V_2 V_3], specifying the vertex IDs of the vertices that define the triangular face. *N* is the number of faces in the mesh.

The `faces` argument sets this property at object creation.

Data Types: `single` | `double`

VertexNormals — Normal vectors for mesh vertices
M-by-3 matrix

Normal vectors for the mesh vertices, specified as an *M*-by-3 matrix. Each row of the matrix is in the form [`x y z`], specifying the normal vector for a vertex. *M* is the total number of vertices in the mesh.

To set this property, specify it at object creation.

Example: `mesh(vertices,faces,VertexNormals=[8 -4 4; 4 4 8; -6 6 3; -3 -6 6; 3 -6 -6; 6 6 -3])`

Data Types: `single` | `double`

VertexColors — Color values for mesh vertices
M-by-3 matrix

Color values for the mesh vertices, specified as an *M*-by-3 matrix. Each row of the matrix is of the form [`R G B`], specifying the color value for a vertex. Each value must be in the range [`0, 1`]. *M* is the total number of vertices in the mesh.

To set this property, specify it at object creation.

Example: `mesh(vertices,faces,VertexColors=[1 0 0; 0 0 1; 0 0 0; 1 1 1 1; 1 1 0; 0 1 1])`

Data Types: `single` | `double`

FaceNormals — Normals for mesh faces
N-by-3 matrix

Normal vectors for the mesh faces, specified as an *N*-by-3 matrix. Each row of the matrix is of the form [`x y z`], specifying the normal for a face. *N* is the total number of faces in the mesh.

To set this property, specify it at object creation.

Example: `mesh(vertices,faces,FaceNormals=[8 -4 4; 4 4 8; -6 6 3; -3 -6 6; 3 -6 -6; 6 6 -3])`

Data Types: `single` | `double`

FaceColors — Color for mesh faces
N-by-3 matrix

Color values for the mesh faces, specified as an *N*-by-3 matrix. Each row of the matrix is of the form [`R G B`], specifying the color value for a face. Each value must be in the range [`0, 1`]. *N* is the total number of faces in the mesh.

To set this property, specify it at object creation.

Example: `mesh(vertices,faces,FaceColors=[1 0 0; 0 0 1; 0 0 0; 1 1 1; 1 1 0; 0 1 1])`

Data Types: `single` | `double`

NumVertices — Number of mesh vertices
positive integer

Number of mesh vertices, stored as a positive integer.

This property is read-only.

Data Types: `unit32`

NumFaces — Number of mesh faces
positive integer

Number of mesh faces, stored as a positive integer.

This property is read-only.

Data Types: `unit32`

Object Functions

addVertices	Add vertices to surface mesh
addFaces	Add faces to surface mesh
removeVertices	Remove vertices from surface mesh
removeFaces	Remove faces from surface mesh
translate	Translate surface mesh
rotate	Rotate surface mesh
transform	Apply rigid transformation to surface mesh
vertexCenter	Find vertex center of surface mesh
scale	Scale vertices of surface mesh
computeNormals	Compute unit normals for mesh vertices and faces
crop	Crop surface mesh
simplify	Simplify surface mesh
subdivide	Subdivide surface mesh
isEdgeManifold	Check if surface mesh is edge-manifold
isOrientable	Check if surface mesh is orientable
isSelfIntersecting	Check if surface mesh is self-intersecting
isVertexManifold	Check if surface mesh is vertex-manifold
isWatertight	Check if surface mesh is watertight
removeDefects	Remove surface mesh defects

Examples

Create Cuboid Surface Mesh

Define mesh vertices for a cuboid.

```
vertices = [1 -1  1; 1 1 1; -1 1 1; -1 -1 1; ...
            1 -1 -1; 1 1 -1; -1 1 -1; -1 -1 -1];
```

Define the mesh faces using the vertices.

```
faces = [6 2 1; 1 5 6; 8 4 3; 3 7 8; 6 7 3; 3 2 6; ...
         5 1 4; 4 8 5; 4 1 2; 2 3 4; 7 6 5; 5 8 7];
```

Create the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
```

Display the surface mesh.

```
surfaceMeshShow(mesh,Title="Cuboid Mesh")
```

Version History
Introduced in R2022b

See Also
pc2surfacemesh | readSurfaceMesh | writeSurfaceMesh | surfaceMeshShow

findPose

Find absolute pose of 2-D lidar scan in the map

Syntax

```
absPose = findPose(scanMapObj,scan)
absPose = findPose(scanMapObj,scan,positionEstimate)
[absPose,scanID,score] = findPose( ___ )
[ ___ ] = findPose( ___ ,Name=Value)
```

Description

`absPose = findPose(scanMapObj,scan)` finds the absolute pose for a scan in the map matching the input scan.

`absPose = findPose(scanMapObj,scan,positionEstimate)` specifies a position estimate for the input scan with respect to the map. This reduces the computation time of the function.

`[absPose,scanID,score] = findPose( ___ )` returns the scan ID of the matching scan and the corresponding correlation score, using any combination of input arguments from previous syntaxes.

`[ ___ ] = findPose( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to any combination of arguments from previous syntaxes. For example, `findPose(scanMapObj,scan,positionEstimate,SearchRadius=10)` searches for a matching scan in the map within a 10 meter radius of the position estimate.

Examples

Find Absolute Pose of 2-D Lidar Scans in Map

Load a MAT file containing 2-D lidar scans and a warehouse map into the workspace.

```
data = load("wareHouse.mat");
scanMapObj = data.wareHouseMap;
lidarScans = data.wareHouseScans;
```

Display the warehouse map.

```
ax = show(scanMapObj,ShowTrajectory=false);
```

Find the absolute pose of the first scan in the map by using the `findPose` function. Specify the pose estimate of the scan as [0 0] and the search radius as 5 meters.

```
absPose = findPose(scanMapObj,lidarScans{1},[0 0],SearchRadius=5);
```

Display the pose of the first scan in the map.

```
showShape("circle",[absPose(1:2) 0.2],Color="g",Parent=ax)
```
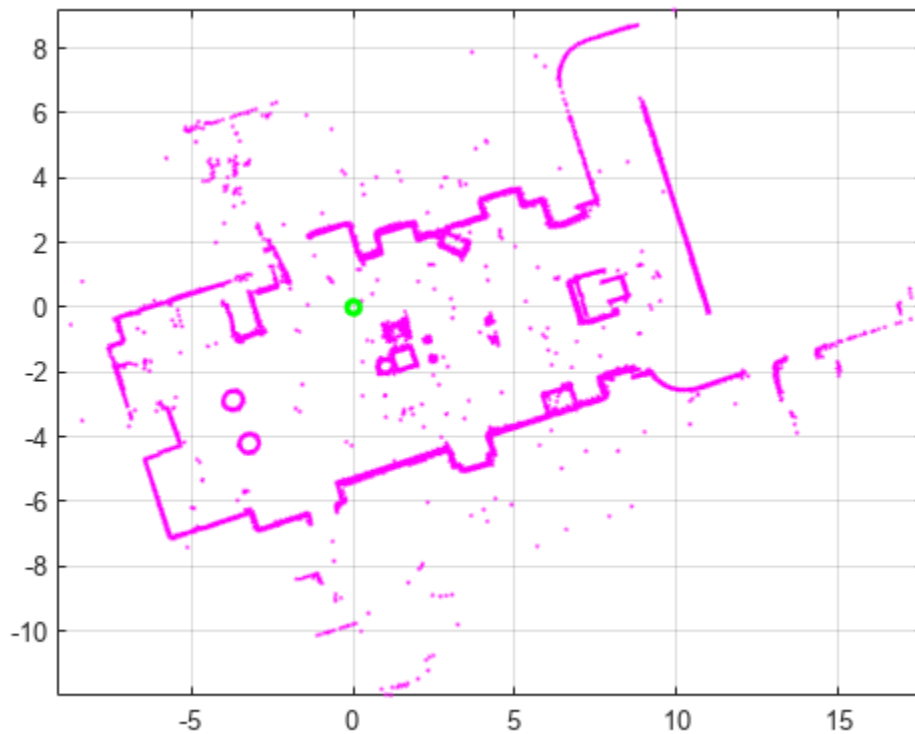
Input Arguments

scanMapObj — 2-D lidar scan map
`lidarscanmap` object

2-D lidar scan map, specified as a `lidarscanmap` object.

scan — Scan for which to find absolute pose
`lidarScan` object

Scan for which to find the absolute pose, specified as `lidarScan` object

positionEstimate — Position estimate of input scan
two-element vector

Position estimate of the input scan with respect to the map, specified as a two-element vector of the form [*x y*], where *x* and *y* represent the position of the scan in meters. The values are relative to the world origin.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `findPose(scanMapObj,scan,SearchRadius=10)` searches for a matching scan in the map within a 10 meter radius of the position estimate.

`SearchRadius` — Search radius for identifying matching scan
8 (default) | positive scalar

Search radius for identifying a matching scan in the map, specified as a positive scalar. This argument specifies the radius to search around the specified `positionEstimate`, in meters. Tune this argument based on the vehicle trajectory. Increasing this value can increase the computation time.

Note Use this argument only when you specify the `positionEstimate` the input.

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

`MatchThreshold` — Minimum correlation score for matching scan
100 (default) | positive scalar

Minimum correlation score for a matching scan in the map, specified as a positive scalar. A higher value can result in a better match.

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

`NumMatches` — Maximum number of matching scans
1 (default) | positive integer

Maximum number of matching scans to identify, specified as a positive integer.

Note When this value is greater than 1, the function returns `absPose` as a matrix, where each row corresponds to a matching scan. The number of rows in the matrix is equal to the number of matching scans.

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

Output Arguments

`absPose` — Absolute pose of scan
three-element vector | M-by-3 matrix

Absolute pose of the scan with respect to the map, returned as a three-element vector of the form [x y Θ], where x and y define the position in meters, and Θ defines the orientation of the input scan in radians. The values are relative to the world origin.

Note When the value of `NumMatches` is greater than 1, the function returns `absPose` as an M-by-3 matrix, where each row corresponds to a matching scan. M is the number of matching scans.

`scanID` — Scan ID of matching scan
positive integer | vector

Scan ID of the matching scan in the map, returned as a positive integer.

> **Note** When the value of `NumMatches` is greater than 1, the function returns `scanID` as an *M*-element vector, where each value corresponds to a matching scan. *M* is the number of matching scans.

Data Types: `double`

`score` — Correlation score of matching scan
positive scalar | vector

Correlation score of the matching scan to the input scan, returned as a positive scalar. A higher score indicates a better match.

> **Note** When the value of `NumMatches` is greater than 1, the function returns `score` as an *M*-element vector, where each value corresponds to a matching scan. *M* is the number of matching scans.

Data Types: `double`

Version History
Introduced in R2022b

See Also
`lidarscanmap` | `poseGraph` | `addScan` | `updateScanPoses` | `detectLoopClosure` | `deleteLoopClosure` | `show`

Topics
"Build Map from 2-D Lidar Scans Using SLAM"

show

Display 2-D lidar scans and lidar sensor trajectory

Syntax

```
show(scanMapObj)
ax = show(scanMapObj)
[ ___ ] = show( ___ ,Name=Value)
```

Description

show(scanMapObj) displays the map defined by the lidarscanmap object scanMapObj, with all lidar scans overlaid at their estimate poses in the map. The function also displays the sensor trajectory.

ax = show(scanMapObj) displays the lidar scan map and returns the axes handle ax.

[___] = show(___ ,Name=Value) specifies options using one or more name-value arguments in addition to any combination of arguments from previous syntaxes. For example, show(scanMapObj,ShowTrajectory=False) displays the pose graph of lidar scans, but not the sensor trajectory.

Examples

Build and Visualize Map by Adding 2-D Lidar Scans

Load a MAT file containing 2-D lidar scans into the workspace.

```
data = load("wareHouse.mat");
scans = data.wareHouseScans;
```

Create a lidarscanmap object.

```
scanMapObj = lidarscanmap;
```

Add the first 15 scans from the input data to the scanMapObj object by using the addScan function.

```
for currentID = 1:15
    addScan(scanMapObj,scans{currentID});
end
```

Visualize the map and the sensor trajectory.

```
figure
show(scanMapObj);
```

Input Arguments

scanMapObj — 2-D lidar scan map
`lidarscanmap` object

2-D lidar scan map, specified as a `lidarscanmap` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `show(scanMapObj,ShowTrajectory=False)` displays the pose graph of lidar scans, but not the sensor trajectory.

Parent — Axes on which to plot map
`Axes` object | `UIAxes` object

Axes on which to plot lidar scan map, specified as an `Axes` object or an `UIAxes` object. For more information, see `axes`, `uiaxes` documentation.

ShowTrajectory — Sensor trajectory map
`true` or `1` (default) | `false` or `0`

Sensor trajectory map, specified as a logical `1` (`true`) or `0` (`false`). Specify this as `false` to not display the sensor trajectory.

Data Types: `logical`

ShowMap — Lidar scan map environment
`true` or 1 (default) | `false` or 0

Lidar scan map environment, specified as a logical `1` (`true`) or `0` (`false`). Specify this as `false` to not display the map environment.

Data Types: `logical`

Output Arguments

ax — Axes of lidar scan map plot
`Axes` object | `UIAxes` object

Axes of the lidar scan map plot, returned as an `Axes` object or an `UIAxes` object. For more information, see `axes`, `uiaxes` documentation.

Version History
Introduced in R2022b

See Also
`lidarscanmap` | `poseGraph` | `addScan` | `findPose` | `updateScanPoses` | `copy`

Topics
"Build Map from 2-D Lidar Scans Using SLAM"

updateScanPoses

Update absolute poses of 2-D lidar scans

Syntax

```
updateScanPoses(scanMapObj,absPoses)
```

Description

`updateScanPoses(scanMapObj,absPoses)` updates the absolute poses of all lidar scans in the `lidarscanmap` object `scanMapObj`.

Examples

Update Absolute Poses of 2-D Lidar Scans in Map

Load a MAT file containing 2-D lidar scans into the workspace.

```
data = load("wareHouse.mat");
scans = data.wareHouseScans;
```
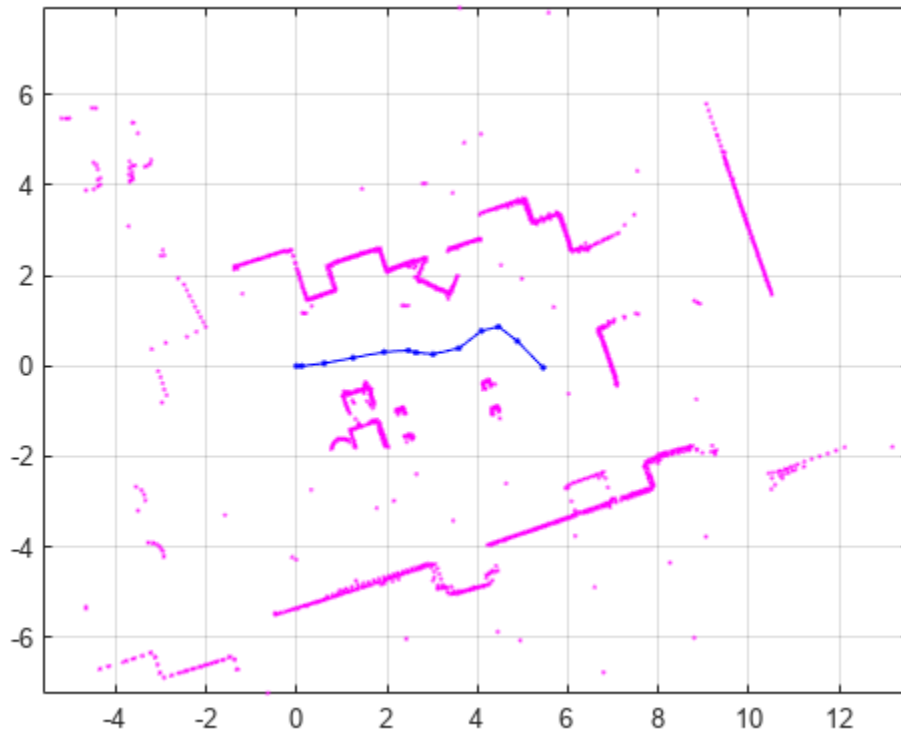
Create a `lidarscanmap` object.

```
scanMapObj = lidarscanmap;
```

Add the first 3 scans from the input data to the `scanMapObj` object by using the `addScan` function.

```
for currentID = 1:3
    addScan(scanMapObj,scans{currentID});
end
```

Update absolute poses of the scans.

```
absPoses = [1 0 0; 1 -2 -5; 1 3 -1];
updateScanPoses(scanMapObj,absPoses);
```

Input Arguments

scanMapObj — 2-D lidar scan map
`lidarscanmap` object

2-D lidar scan map, specified as a `lidarscanmap` object.

absPoses — Absolute poses of 2-D lidar scans
M-by-3 matrix

Absolute poses of 2-D lidar scans in the map, specified as an *M*-by-3 matrix, where *M* is the number of scans in the `lidarscanmap` object. Each row of the matrix is of the form [*x* *y* *Θ*], where *x* and *y* define the position in meters, and *Θ* defines the orientation of the scan in radians.

Version History
Introduced in R2022b

See Also
lidarscanmap | poseGraph | findPose | addScan | updateScanPoses | detectLoopClosure | show

Topics
"Build Map from 2-D Lidar Scans Using SLAM"

poseGraph

Create 2-D pose graph from lidar scan map

Syntax

```
pGraph = poseGraph(scanMapObj)
```

Description

`pGraph = poseGraph(scanMapObj)` creates a pose graph from the 2-D lidar scan map object `scanMapObj`. You can use this pose graph for inspection, visualization, and pose graph optimization.

This function requires Navigation Toolbox™ version 2.3 or higher.

To optimize the pose graph output, use the `optimizePoseGraph` function.

Examples

Create Pose Graph from 2-D Lidar Scans

Load a MAT file containing 2-D lidar scans into the workspace.

```
data = load("wareHouse.mat");
scans = data.wareHouseScans;
```

Create a `lidarscanmap` object.

```
scanMapObj = lidarscanmap;
```

Add the first 15 scans from the input data to the `scanMapObj` object by using the `addScan` function.

```
for currentID = 1:15
    addScan(scanMapObj,scans{currentID});
end
```

Create a pose graph for the `scanMapObj`.

```
poseGraph(scanMapObj);
```

Visualize the map.

```
figure
show(scanMapObj);
```
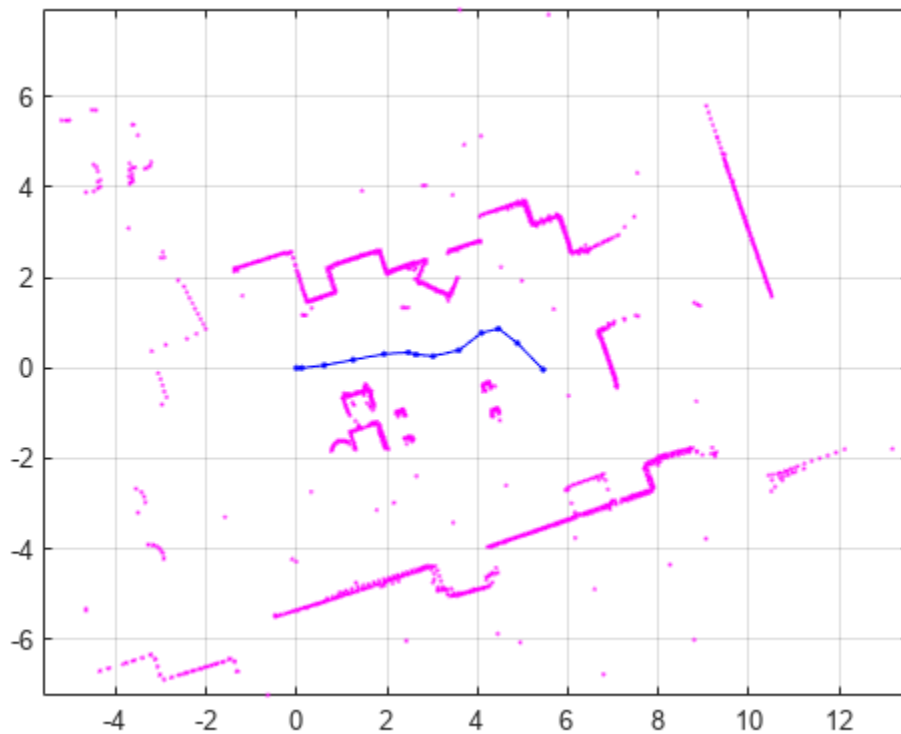
Input Arguments

scanMapObj — 2-D lidar scan map
lidarscanmap object

2-D lidar scan map, specified as a lidarscanmap object.

Output Arguments

pGraph — Pose graph
poseGraph object

Pose graph of the input lidar scan map, returned as a poseGraph object.

Version History
Introduced in R2022b

See Also
lidarscanmap | poseGraph | findPose | addScan | updateScanPoses | detectLoopClosure | show

Topics
"Build Map from 2-D Lidar Scans Using SLAM"

copy

Create a copy of `lidarscanmap` object

Syntax

```
newMapObj = copy(scanMapObj)
```

Description

`newMapObj = copy(scanMapObj)` creates a copy of `lidarscanmap` object `scanMapObj` with the same object properties.

Examples

Create Copy of `lidarscanmap` Object

Load a MAT file containing 2-D lidar scans into the workspace.

```
data = load("wareHouse.mat");
scans = data.wareHouseScans;
```

Create a `lidarscanmap` object.

```
orgMapObj = lidarscanmap;
```

Add the first scan to the `orgMapObj` object.

```
addScan(orgMapObj,scans{1});
```

Create a copy of the `orgMapObj` object, then add the second scan to the copied object, `newMapObj`.

```
newMapObj = copy(orgMapObj);
addScan(newMapObj,scans{2});
```

Display the number of scans in the original and the copied objects.

```
 disp([orgMapObj.NumScans newMapObj.NumScans])

     1     2
```

Input Arguments

scanMapObj — 2-D lidar scan map
`lidarscanmap` object

2-D lidar scan map, specified as a `lidarscanmap` object.

Output Arguments

newMapObj — Copy of input lidar scan map
`lidarscanmap` object

Copy of the input lidar scan map, returned as a `lidarscanmap` object.

Version History
Introduced in R2022b

See Also
`lidarscanmap` | `poseGraph` | `addScan` | `findPose` | `updateScanPoses` | `detectLoopClosure` | `show`

Topics
"Build Map from 2-D Lidar Scans Using SLAM"

deleteLoopClosure

Delete loop closure between 2-D lidar scans

Syntax

```
deleteLoopClousure(scanMapObj,fromScanID,toScanID)
```

Description

`deleteLoopClousure(scanMapObj,fromScanID,toScanID)` deletes the existing loop closure connection between the scans with scan IDs specified by `fromScanID` and `toScanID` from the `lidarscanmap` object `scanMapObj`.

Examples

Delete Loop Closure Connections from Map

Load a MAT file containing a map of a warehouse into the workspace.

```
data = load("wareHouse.mat");
scanMapObj = data.wareHouseMap;
```

Display the list of loop closure connections in the `lidarscanmap` object `scanMapObj`. Also, display the number of loop closures.

```
scanMapObj.LoopClosureIDs
```

```
ans=13×2 table
    FromScanID     ToScanID
    _____     _____

        4             59
        4             60
        4             61
        4             62
        4             63
        4             64
        4             65
       11             66
       11             67
       11             68
       11             69
       11             70
       11             71
```

```
disp(scanMapObj.NumLoopClosures)
```

```
    13
```

Delete the loop closure connection between scans 4 and 59 by using the `deleteLoopClosure` function. Then check the number of loop closures in `scanMapObj` object.

```
deleteLoopClosure(scanMapObj,4,59);
disp(scanMapObj.NumLoopClosures)

    12
```

Input Arguments

scanMapObj — 2-D lidar scan map
lidarscanmap object

2-D lidar scan map, specified as a lidarscanmap object.

fromScanID — Scan ID of scan at beginning of loop closure
positive integer | vector

Scan ID of the scan at the beginning of the loop closure, specified as a positive integer or vector. To delete multiple loop closure connections in the map, you must specify the value as a vector.

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

toScanID — Scan ID of scan at end of loop closure
positive integer | vector

Scan ID of the scan at the end of the loop closure, specified as a positive integer or vector. To delete multiple loop closure connections in the map, you must specify the value as a vector.

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

Version History
Introduced in R2022b

See Also
lidarscanmap | poseGraph | detectLoopClosure | addLoopClosure | findPose | updateScanPoses

Topics
"Build Map from 2-D Lidar Scans Using SLAM"

addLoopClosure

Add loop closure to the map

Syntax

```
addLoopClosure(scanMapObj,fromScanID,toScanID,relPose)
addLoopClosure(scanMapObj,fromScanID,toScanID,relPose,informationMatrix)
```

Description

addLoopClosure(scanMapObj,fromScanID,toScanID,relPose) adds a loop closure between the scans with scan IDs fromScanID and toScanID to the lidarscanmap object, where relPose specifies the relative pose between the scans.

addLoopClosure(scanMapObj,fromScanID,toScanID,relPose,informationMatrix) specifies the information matrix that represents the uncertainty in the relative pose measurement.

Examples

Detect and Add Loop Closures to Map

Load a MAT file containing 2-D lidar scans into the workspace.

```
data = load("wareHouse.mat");
scans = data.wareHouseScans;
```

Create a lidarscanmap object.

```
scanMapObj = lidarscanmap;
```

Add the first 70 scans from the input data to the scanMapObj object by using the addScan function.

```
for currentID = 1:70
    addScan(scanMapObj,scans{currentID});
end
```

Detect loop closures in the map by using the detectLoopClosure function. The function searches the map for a previous scan matching the most recent scan. For best results, adjust the excluded views and the search radius according to the sensor trajectory.

```
[~,scanID,~] = detectLoopClosure(scanMapObj,NumExcludeViews=20,SearchRadius=5)
```

```
scanID = 11
```

Add the loop closure detected between the scans 70 and 11 to the scanMapObj by using the addLoopClosure function. Display the loop closure connection.

```
addLoopClosure(scanMapObj,70,11,[2 2 0])
disp(scanMapObj.LoopClosureIDs)
```

```
    FromScanID      ToScanID
    _____      _____
```

```
        70              11
```

Visualize the map and the sensor trajectory.

```
figure
show(scanMapObj);
```


Input Arguments

scanMapObj — 2-D lidar scan map
lidarscanmap object

2-D lidar scan map, specified as a lidarscanmap object.

fromScanID — Scan ID of scan at beginning of loop closure
positive integer | vector

Scan ID of the scan at the beginning of the loop closure, specified as a positive integer or vector. To add multiple loop closure connections in the map, you must specify the value as a vector.

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

toScanID — Scan ID of scan at end of loop closure
positive integer | vector

Scan ID of the scan at the end of the loop closure, specified as a positive integer or vector. To add multiple loop closure connections in the map, you must specify the value as a vector.

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

`relPose` — Relative pose between scans
three-element vector | *M*-by-3 matrix

Relative pose between the scans in the loop closure, specified as a three-element vector or an *M*-by-3 matrix. To add more than one loop closure connection, specify the value as a matrix, where *M* is the number of loop closure connections. Each row of the matrix is of the form [*x* *y* Θ], where *x* and *y* define the translational offset in meters, and Θ defines the rotational offset between the scans in radians.

`informationMatrix` — Uncertainty in relative pose measurement
M-element cell array

Uncertainty in the relative pose measurement, specified as an *M*-element cell array, where *M* is the number of loop closure connections. Each cell contains a 3-by-3 positive definite matrix.

Version History
Introduced in R2022b

See Also
`lidarscanmap` | `poseGraph` | `addScan` | `findPose` | `updateScanPoses` | `detectLoopClosure` | `deleteLoopClosure` | `show`

Topics
"Build Map from 2-D Lidar Scans Using SLAM"

detectLoopClosure

Detect loop closure in 2-D lidar scan map

Syntax

```
relPose = detectLoopClosure(scanMapObj)
[relPose,scanID,score] = detectLoopClosure(scanMapObj)
[ ___ ] = detectLoopClosure( ___ ,Name=Value)
```

Description

`relPose = detectLoopClosure(scanMapObj)` searches for a scan matching the most recent scan of the `scanMapObj` object to detect loop closures in the map. The function returns the relative pose `relPose` between the matched scan and the most recent scan.

`[relPose,scanID,score] = detectLoopClosure(scanMapObj)` returns the scan ID of the matched scan and its corresponding correlation score.

`[ ___ ] = detectLoopClosure( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to any combination of arguments from previous syntaxes. For example, `detectLoopClosure(scanMapObj,SearchRadius=10)` searches for a matching scan in the map within a 10 meter radius of the most recent scan.

Examples

Detect and Add Loop Closures to Map

Load a MAT file containing 2-D lidar scans into the workspace.

```
data = load("wareHouse.mat");
scans = data.wareHouseScans;
```

Create a `lidarscanmap` object.

```
scanMapObj = lidarscanmap;
```

Add the first 70 scans from the input data to the `scanMapObj` object by using the `addScan` function.

```
for currentID = 1:70
    addScan(scanMapObj,scans{currentID});
end
```

Detect loop closures in the map by using the `detectLoopClosure` function. The function searches the map for a previous scan matching the most recent scan. For best results, adjust the excluded views and the search radius according to the sensor trajectory.

```
[~,scanID,~] = detectLoopClosure(scanMapObj,NumExcludeViews=20,SearchRadius=5)

scanID = 11
```

Add the loop closure detected between the scans 70 and 11 to the `scanMapObj` by using the `addLoopClosure` function. Display the loop closure connection.

```
addLoopClosure(scanMapObj,70,11,[2 2 0])
disp(scanMapObj.LoopClosureIDs)
```
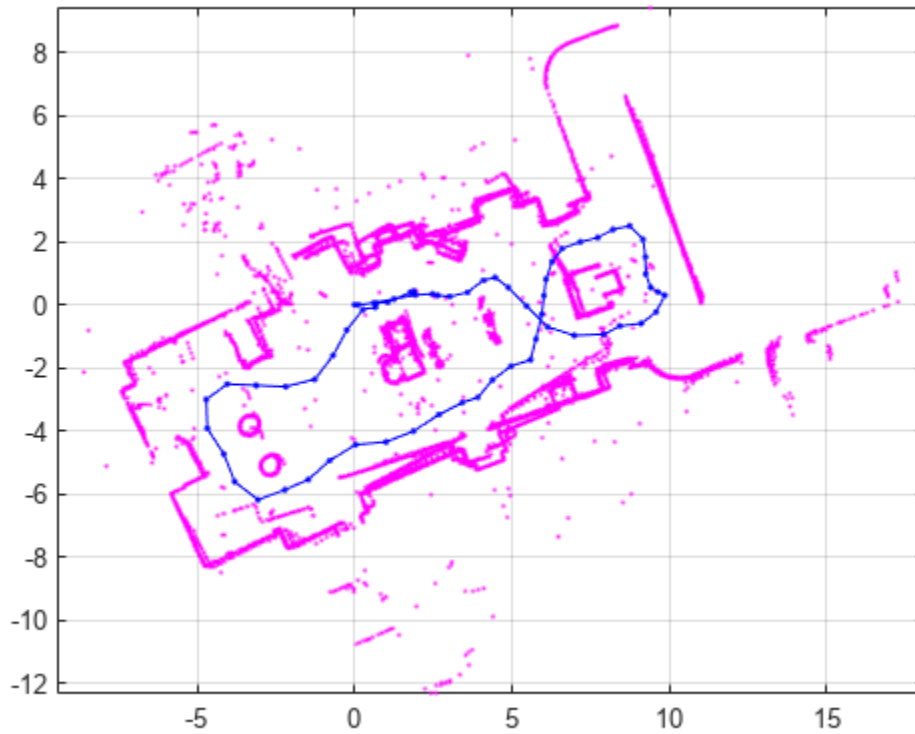
```
    FromScanID    ToScanID
    _____    _____

        70           11
```

Visualize the map and the sensor trajectory.

```
figure
show(scanMapObj);
```


Input Arguments

scanMapObj — 2-D lidar scan map
lidarscanmap object

2-D lidar scan map, specified as a `lidarscanmap` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `detectLoopClosure(scanMapObj,SearchRadius=10)` searches for a matching scan within a 10 meter radius of the most recent scan.

`SearchRadius` — Search radius to identify matching scan
8 (default) | positive scalar

Search radius to identify a matching scan in the map, specified as a positive scalar. The function searches in the specified radius around the most recent scan. Increasing this value can increase the computation time of the function.

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

`MatchThreshold` — Minimum correlation score to identify matching scan
100 (default) | positive scalar

Minimum correlation score to identify a matching scan, specified as a positive scalar. A higher value can result a better match and decrease false positives.

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

`NumExcludeViews` — Number of recently added views to exclude
10 (default) | positive integer

Number of recently added views to exclude for loop closure detection, specified as a positive integer. Increase this value when multiple recently added views belong to the same region of the map.

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

`NumMatches` — Number of matches to output
1 (default) | positive integer

Number of matches to output, specified as a positive integer.

Note When this value is greater than 1, the function returns `relPose` as a matrix, where each row specifies a matching scan. The number of rows in the matrix is equal to the `NumMatches` value.

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

Output Arguments

`relPose` — Relative pose between matched scan and most recent scan
three-element vector (default) | matrix

Relative pose between the matched scan and the most recent scan of the map, returned as a three-element vector of the form [x y Θ], where x and y define the translational offset in meters, and Θ defines the rotational offset in radians.

Note When the value of `NumMatches` is greater than 1, the function returns `relPose` as a matrix, where each row specifies a matching scan. The number of rows in the matrix is equal to the `NumMatches` value.

scanID — Scan ID of matching scan
positive integer | vector

Scan ID of the matching scan in the map, returned as a positive integer.

Note When the value of `NumMatches` is greater than 1, the function returns `scanID` as an *M*-element vector, where each value corresponds to a matching scan. *M* is the number of matching scans.

Data Types: `double`

score — Correlation score of matching scan
positive scalar | vector

Correlation score of the matching scan to the input scan, returned as a positive scalar. A higher score indicates a better match.

Note When the value of `NumMatches` is greater than 1, the function returns `score` as an *M*-element vector, where each value corresponds to a matching scan. *M* is the number of matching scans.

Data Types: `double`

Version History
Introduced in R2022b

See Also
`lidarscanmap` | `poseGraph` | `detectLoopClosure` | `addLoopClosure` | `deleteLoopClosure` | `findPose` | `updateScanPoses`

Topics
"Build Map from 2-D Lidar Scans Using SLAM"

addScan

Add 2-D lidar scan to map

Syntax

```
addScan(scanMapObj,currScan)
addScan(scanMapObj,currScan,relPose)
addScan( ___ ,Name=Value)
isScanAdded = addScan( ___ )
```

Description

addScan(scanMapObj,currScan) adds the specified 2-D lidar scan currScan to the most recent scan of the lidarscanmap object scanMapObj. The function uses scan matching to correlate this scan to the most recent one, then adds the scan to the scanMapObj.

addScan(scanMapObj,currScan,relPose) specifies the relative pose between the input scan and the most recent scan of the map.

addScan(___ ,Name=Value) specifies options using one or more name-value arguments in addition to any combination of arguments from previous syntaxes. For example, addScan(scanMapObj,currScan,ScanID=5) adds the input scan relative to the fifth scan in scanMapObj.

isScanAdded = addScan(___) returns an indication of whether the input scan is added or rejected.

Examples

Build and Visualize Map by Adding 2-D Lidar Scans

Load a MAT file containing 2-D lidar scans into the workspace.

```
data = load("wareHouse.mat");
scans = data.wareHouseScans;
```

Create a lidarscanmap object.

```
scanMapObj = lidarscanmap;
```

Add the first 15 scans from the input data to the scanMapObj object by using the addScan function.

```
for currentID = 1:15
    addScan(scanMapObj,scans{currentID});
end
```

Visualize the map and the sensor trajectory.

```
figure
show(scanMapObj);
```
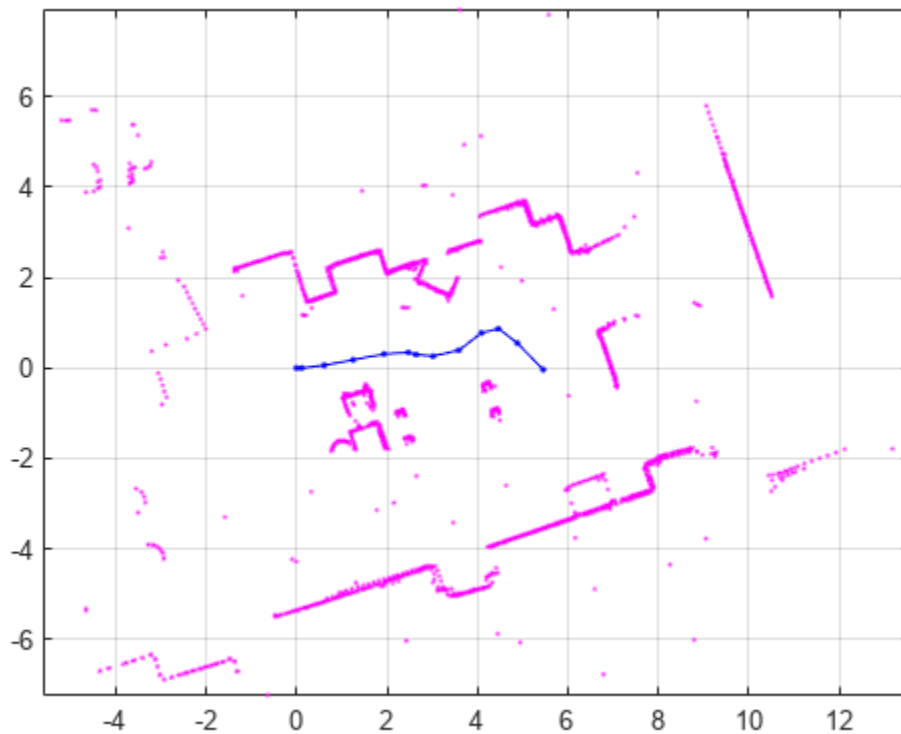
Input Arguments

scanMapObj — Map of 2-D lidar scans
lidarscanmap object

Map of 2-D lidar scans, specified as a lidarscanmap object

currScan — Input scan to add to map
lidarScan object

Input scan to add to the map, specified as a lidarScan object

relPose — Relative pose between input scan and most recent scan
three-element vector

Relative pose between the input scan and the most recent scan of the map, specified as a three-element vector of the form [*x* *y* Θ], where *x* and *y* define the translational offset in meters, and Θ defines the rotational offset of the input scan in radians.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: addScan(scanMapObj,currScan,ScanID=5) adds the input scan relative to the fifth scan in the scanMapObj object.

RelPoseEstimate — Initial estimate for relative pose between input scan and most recent scan
[0 0 0] (default) | three-element vector

Initial estimate for relative pose between the input scan and the most recent scan of the map, specified as a three-element vector of the form [*x y Θ*], where *x* and *y* define the translational offset in meters, and *Θ* defines the rotational offset between the scans in radians. The values are relative to the world origin.

The function computes the relative pose faster when the RelPoseEstimate is close to the true relative pose.

Note Use RelPoseEstimate only when you do not specify the relPose input.

PoseTolerance — Tolerance for relative pose estimate
three-element vector

Tolerance for the relative pose estimate between the input scan and the most recent scan of the map, specified as a three-element vector of the form [*x y Θ*], where *x* and *y* define the translational tolerance in meters, and *Θ* defines the rotational tolerance in radians.

If you do not specify this value, the function internally computes it as [scanMapObj.MaxLidarRange/2 scanMapObj.MaxLidarRange/2 pi/2].

Note Use PoseTolerance only when you do not specify the relPose input.

InformationMatrix — Uncertainty in relative pose measurement
eye(3) (default) | 3-by-3 positive definite matrix

Uncertainty in the relative pose measurement between the input scan and the most recent scan of the map, specified as a 3-by-3 positive definite matrix.

MovementThreshold — Minimum pose change required to add input scan
[0 0] (default) | two-element vector

Minimum pose change required to add the input scan to the map, specified as a two-element vector of the form [*translation rotation*], where *translation* and *rotation* specify the translational and the rotational thresholds, respectively. The function adds the input scan to the scanMapObj object only when the relative pose change of the input scan exceeds both values in the MovementThreshold.

ScanID — Scan to which input scan is relative
positive integer

Scan ID of the scan to which the input scan is relative to, specified as a positive integer. The value must be in the range [1, scanMapObj.NumScans]. By default, the function adds the input scan relative to the last scan in the scanMapObj object.

Note You can specify ScanID only when the scanMapObj object has at least one scan.

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

Output Arguments

isScanAdded — Check if input scan is added
`true` | `false`

Check if the input scan is added to the `scanMapObj` object, returned as a logical `true` or `false`. The function returns `true` when the scan is added to the map.

Version History
Introduced in R2022b

See Also
`lidarscanmap` | `poseGraph` | `updateScanPoses`

Topics
"Build Map from 2-D Lidar Scans Using SLAM"

lidarscanmap

Simultaneous localization and mapping using 2-D lidar scans

Description

A `lidarscanmap` object performs simultaneous localization and mapping (SLAM) using the 2-D lidar scans. The `lidarscanmap` object uses a graph-based SLAM algorithm to create a map of an environment from 2-D lidar scans. First, the algorithm builds a pose graph by linking the input scans using their absolute poses. Then, it uses a scan-matching approach to detect loop closures to minimize any odometry drift.

Using the `lidarscanmap` object, you can:

- Store and add lidar scans incrementally.
- Detect, add, and delete loop closures.
- Find and update the absolute poses of the scans.
- Generate and visualize a pose graph.

To further optimize the pose graph, use the `optimizePoseGraph` function.

Creation

Syntax

```
scanMapObj = lidarscanmap
scanMapObj = lidarscanmap(gridResolution,maxLidarRange)
```

Description

`scanMapObj = lidarscanmap` creates a `lidarscanmap` object with default property values.

`scanMapObj = lidarscanmap(gridResolution,maxLidarRange)` specifies the resolution of the occupancy grid map and maximum range of the lidar sensor. The `gridResolution` and `maxLidarRange` arguments set the `GridResolution` and the `MaxLidarRange` properties, respectively.

Properties

ScanAttributes — 2-D lidar scans and absolute poses
table

2-D lidar scans and their absolute poses, stored as a table, where each row represents a lidar scan. The table has these columns.

- `ScanID` — Unique identifier for the scan, stored as a positive integer.
- `LidarScan` — 2-D lidar scan, stored as a `lidarScan` object.

- **AbsolutePose** — Absolute pose of the lidar scan, stored as a three-element vector of the form [*x y Θ*], where *x* and *y* define the position in meters, and *Θ* defines the orientation of the input scan in radians.

This property is read-only.

ConnectionAttributes — Connections between lidar scans
table

Connections between lidar scans, stored as a table, where each row represents a connection. The table has these columns.

- **FromScanID** — **ScanID** of the scan at the beginning of the connection, stored as a positive integer.
- **ToScanID** — **ScanID** of the scan at the end of the connection, stored as a positive integer.
- **RelativePose** — Relative pose of the corresponding scan specified by **ToScanID** with respect to the connected scan specified by **FromScanID**, stored as a three-element vector of the form [*x y Θ*], where *x* and *y* define the translational offset in meters, and *Θ* defines the rotational offset between the scans in radians.
- **InformationMatrix** — Uncertainty in the relative pose measurement, stored as a 3-by-3 matrix. This matrix is the inverse of the covariance matrix.

This property is read-only.

LoopClosureIDs — Loop closure between lidar scans
table

Loop closure connections between lidar scans, stored as a table, where each row represents a loop closure. The table has these columns.

- **FromScanID** — **ScanID** of the scan at the beginning of the connection, stored as a positive integer.
- **ToScanID** — **ScanID** of the scan at the end of the connection, stored as a positive integer.

This property is read-only.

NumScans — Number of lidar scans
positive integer

Number of lidar scans in the **lidarscanmap** object, specified as a positive integer.

This property is read-only.

Data Types: double

NumConnections — Number of connections
positive integer

Number of connections in the **lidarscanmap** object, specified as a positive integer.

This property is read-only.

Data Types: double

NumLoopClosures — Number of loop closure connections
positive integer

Number of loop closure connections in the `lidarscanmap` object, specified as a positive integer.

This property is read-only.

Data Types: `double`

GridResolution — Resolution of occupancy grid map
positive scalar

Resolution of the occupancy grid map, specified as a positive scalar.

To set this property, you must specify it at object creation.

Data Types: `double`

MaxLidarRange — Maximum range of lidar sensor
positive scalar

Maximum range of the lidar sensor, specified as a positive scalar. Units are in meters.

To set this property, you must specify it at object creation.

Data Types: `double`

Object Functions

addScan	Add 2-D lidar scan to map
detectLoopClosure	Detect loop closure in 2-D lidar scan map
addLoopClosure	Add loop closure to the map
deleteLoopClosure	Delete loop closure between 2-D lidar scans
poseGraph	Create 2-D pose graph from lidar scan map
findPose	Find absolute pose of 2-D lidar scan in the map
updateScanPoses	Update absolute poses of 2-D lidar scans
copy	Create a copy of lidarscanmap object
show	Display 2-D lidar scans and lidar sensor trajectory

Examples

Localize 2-D Lidar Scans in Map

Load a MAT file containing 2-D lidar scans and a warehouse map into the workspace.

```
data = load("wareHouse.mat");
scans = data.wareHouseScans;
```

Create a `lidarscanmap` object.

```
scanMapObj = lidarscanmap;
```

Add the scans from the input data to the `scanMapObj` object by using the `addScan` function.

```
for currentID = 1:70
    addScan(scanMapObj,scans{currentID});
end
```

Display the warehouse map.

```
ax = show(scanMapObj,ShowTrajectory=false);
hold on
```

Find the absolute pose of the first scan in the map by using the `findPose` function. Specify the pose estimate of the scan as [0 0] and the search radius as 5 meters.

```
absPose = findPose(scanMapObj,scans{1},[0 0],SearchRadius=5);
```

Display the pose of the first scan in the map.

```
showShape("circle",[absPose(1:2) 0.2],Color="g",Parent=ax)
```

Iterate the previous two steps to localize the remaining scans in the map and visualize the results.

```
for n = 2:numel(scans)
    currentScan = scans{n};

    % Use the absolute pose of the previous scan as the pose estimate for
    % the next scan
    poseEstimate = absPose(1:2);
    absPose = findPose(scanMapObj,currentScan,poseEstimate,SearchRadius=5);

    % Display the pose of the current scan in the map
    pose = [absPose(1:2) 0.05];
    showShape("circle",pose,Color="b")
end
hold off
```
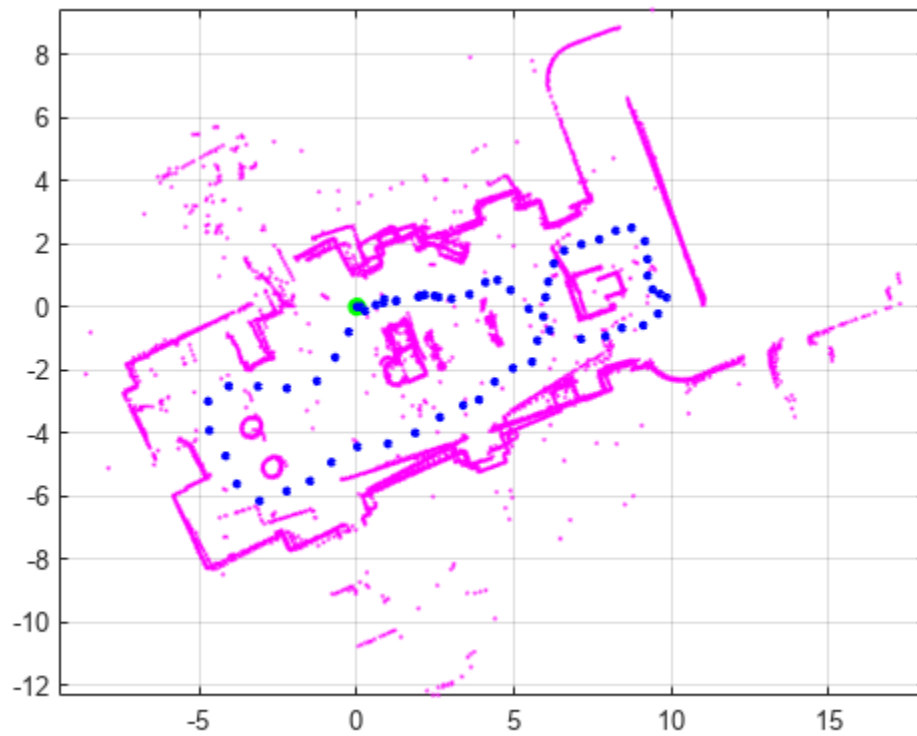
Version History
Introduced in R2022b

See Also
`poseGraph` | `findPose` | `addScan` | `detectLoopClosure` | `show`

Topics
"Build Map from 2-D Lidar Scans Using SLAM"

pcmaploam

Create map of LOAM feature points for map building

Description

The `pcmaploam` object creates a map of lidar odometry and mapping (LOAM) feature points. LOAM feature points represent edge points and surface points that are detected using the LOAM algorithm. Use this object for incremental map building workflows. Use the `findPose` function to find the optimized absolute pose that aligns the points to the map, and use the `addPoints` function to add points to the map.

Creation

Syntax

```
loamMap = pcmaploam(voxelSize)
loamMap = pcmaploam(voxelSize,mapSize)
```

Description

`loamMap = pcmaploam(voxelSize)` returns an empty LOAM map with the voxel size set by the `voxelSize` argument.

`loamMap = pcmaploam(voxelSize,mapSize)` also specifies the size of the map along each axis (*x*, *y*, and *z*). By default, using the `addPoints` function to add points outside the existing LOAM map expands the map. If you specify the `mapSize` argument, the `pcmaploam` object instead discards points outside the specified boundaries. The `mapSize` argument sets the `MapSize` property.

Use this syntax to improve the speed of the `findPose` and `addPoints` functions when mapping large areas.

Properties

`Points` — Points in LOAM map
M-by-3 matrix

Points in the LOAM map, specified as an *M*-by-3 matrix, where *M* is the number of points. Each row specifies the [*x y z*] coordinates of a point.

`VoxelSize` — Voxel size to use for downsampling map points
positive scalar

Voxel size to use for downsampling map points, specified as a positive scalar.

`MapSize` — Size of LOAM map
[[Inf Inf Inf] | three-element vector

Size of the LOAM map, specified as a three-element vector of the form [*dx dy dz*].

XLimits — Range of LOAM map along *x*-axis
[0 0] (default) | two-element vector

Range of the LOAM map along the *x*-axis, specified as a two-element vector of the form [*xmin xmax*].

YLimits — Range of LOAM map along *y*-axis
[0 0] (default) | two-element vector

Range of the LOAM map along the *y*-axis, specified as a two-element vector of the form [*ymin ymax*].

ZLimits — Range of LOAM map along *z*-axis
[0 0] (default) | two-element vector

Range of LOAM map along the *z*-axis, specified as a two-element vector of the form [*zmin zmax*].

Object Functions
addPoints   Add LOAM points to map
findPose   Find absolute pose of points in map
show   Visualize LOAM map

Examples

Create LOAM Map

Create a map to store LOAM feature points.

```
voxelSize = 0.1;
loamMap = pcmaploam(voxelSize);
```

Create a `velodyneFileReader` object to read point cloud data from a PCAP file.

```
veloReader = velodyneFileReader("lidarData_ConstructionRoad.pcap","HDL32E");
```

Read the first point cloud from the file into the workspace.

```
ptCloud1 = readFrame(veloReader,1);
```

Detect LOAM feature points in the point cloud.

```
points1 = detectLOAMFeatures(ptCloud1);
```

Downsample the less planar surface points to improve registration speed.

```
gridStep = 1;
points1 = downsampleLessPlanar(points1,gridStep);
```

Add the LOAM points of the first point cloud to the map.

```
absPose = rigidtform3d;
addPoints(loamMap,points1,absPose)
```

Read the fifth point cloud, and detect the LOAM feature points in it.

```
ptCloud2 = readFrame(veloReader,5);
points2 = detectLOAMFeatures(ptCloud2);
```

Downsample the less planar surface points.

```
points2 = downsampleLessPlanar(points2,gridStep);
```

Get a relative pose estimate by using the `pcregisterloam function`.

```
relPose = pcregisterloam(points2,points1);
```

Find the absolute pose of the points from the fifth point cloud in the map.

```
absPose = findPose(loamMap,points2,relPose);
```

Add the LOAM points from the fifth point cloud to the map.

```
addPoints(loamMap,points2,absPose)
```

Visualize the map.

```
show(loamMap,MarkerSize=20)
```
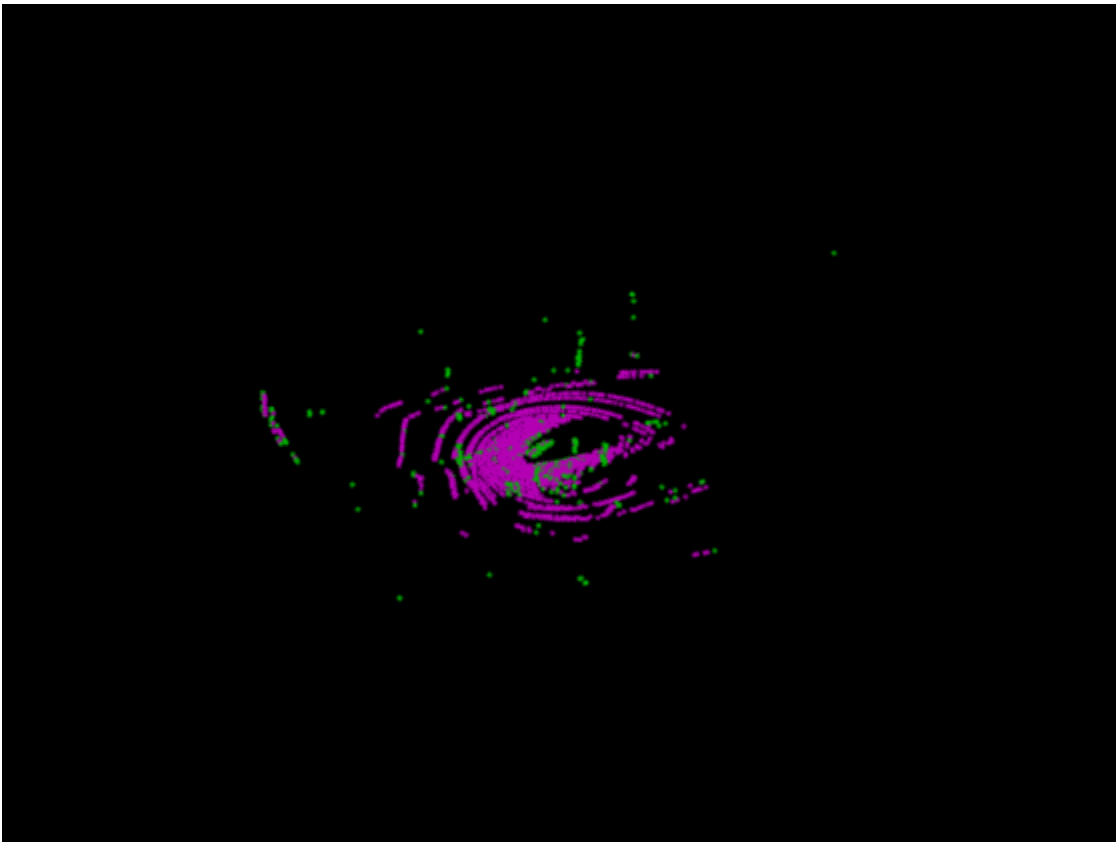

Version History
Introduced in R2022b

See Also

Objects
LOAMPoints

Functions
pcregisterloam | detectLOAMFeatures

lidarSensor

Simulate lidar sensor

Description

The `lidarSensor` System object simulates a lidar sensor mounted on an ego vehicle and outputs point cloud data for a given scene. The generated data is with respect to the ego vehicle coordinate system based on the sensor pose and the actors present in the scene. You can use the `drivingScenario` object to create a scenario containing actors and trajectories, then generate the point cloud data for the scenario by using the `lidarSensor` object.

To simulate lidar sensor using this object:

1   Create the `lidarSensor` object and set its properties.
2   Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
lidar = lidarSensor
lidar = lidarSensor(Name=Value)
```

Description

`lidar = lidarSensor` creates a `lidarSensor` object with default property values. You can use this object to generate lidar point cloud data for a given 3-D environment.

`lidar = lidarSensor(Name=Value)` sets the properties of the object using one or more name-value arguments. For example, `lidarSensor(UpdateRate=0.2)` creates a `lidarSensor` object that generates point cloud detections at every `0.2` seconds.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

SensorIndex — Unique identifier for sensor
1 (default) | positive integer

Unique identifier for the sensor, specified as a positive integer. In a multisensor system, this index distinguishes different sensors from one another.

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

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

`ActorID` of the ego vehicle, specified as a positive integer. The ego vehicle is the actor on which the sensor is mounted, and `ActorID` is the unique identifier for an actor.

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

UpdateRate — Time interval between consecutive sensor updates
0.1 (default) | positive scalar

Time interval between two consecutive sensor updates, specified as a positive scalar. The `lidarSensor` object generates new detections at the interval specified by this property. The value must be an integer multiple of the simulation time. Updates requested from the sensor in between the update intervals contain no detections. Units are in seconds.

Data Types: `single` | `double`

Position — Sensor center position
[1.5 0 1.6] (default) | [*x y height*]

Sensor center position, specified as a three-element vector of the form [*x y height*]. The values of *x* and *y* represent the location of the sensor with respect to the *x*- and *y*-axes of the ego vehicle coordinate system. *height* is the height of the sensor above the ground. The default value defines a lidar sensor mounted on the front edge of the roof of a sedan. Units are in meters.

Data Types: `single` | `double`

Orientation — Sensor orientation
[0 0 0] (default) | three-element vector of form [*roll pitch yaw*]

Sensor orientation, specified as a three-element vector of the form, [*roll pitch yaw*]. These values are with respect to the ego vehicle coordinate system. Units are in degrees.

- *roll* — The roll angle is the angle of rotation around the front-to-back axis, which is the *x*-axis of the ego vehicle coordinate system. A positive roll angle corresponds to a clockwise rotation when looking in the positive direction of the *x*-axis.
- *pitch* — The pitch angle is the angle of rotation around the side-to-side axis, which is the *y*-axis of the ego vehicle coordinate system. A positive roll angle corresponds to a clockwise rotation when looking in the positive direction of the *y*-axis.
- *yaw* — The yaw angle is the angle of rotation around the vertical axis,which is the *z*-axis of the ego vehicle coordinate system. A positive roll angle corresponds to a clockwise rotation when looking in the positive direction of the *z*-axis. This rotation appears counter-clockwise when viewing the vehicle from above.

Data Types: `single` | `double`

MaxRange — Maximum detection range of sensor
120 (default) | positive scalar

Maximum detection range of the lidar sensor, specified as a positive scalar in meters. The sensor cannot detect roads and actors beyond this range.

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

RangeAccuracy — Accuracy of sensor range measurement
`0.002` (default) | positive scalar

Accuracy of the sensor range measurement, specified as a positive scalar. Units are in meters.

Data Types: `single` | `double`

HasNoise — Point cloud data has added noise
`true` (default) | `false`

Point cloud data has added noise, specified as `true` or `false`. When set to `true`, the function adds random Gaussian noise to each point in the point cloud using the `RangeAccuracy` property as one standard deviation. Otherwise, the data has no noise.

Data Types: `logical`

HasOrganizedOutput — Output point cloud is organized
`true` (default) | `false`

Output point cloud is organized, specified as `true` or `false`.

- `true` — The function returns an organized point cloud of the form *M*-by-*N*-by-3, where *M* is the number of elevation channels and *N* is the number of azimuth channels in the point cloud.

- `false` — The function returns an unorganized point cloud of the form *P*-by-3, where *P* is the number of points in the point cloud.

Data Types: `logical`

AzimuthResolution — Azimuth resolution of lidar sensor
`0.16` (default) | positive scalar

Azimuth resolution of the lidar sensor, specified as a positive scalar in degrees.

Data Types: `single` | `double`

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

Elevation resolution of the lidar sensor, specified as a positive scalar in degrees.

Data Types: `single` | `double`

AzimuthLimits — Azimuth limits of lidar sensor
`[-180 180]` (default) | two-element vector

Azimuth limits of the lidar sensor, specified as a two-element vector of the form [*min max*]. The values must be in the range `[-180, 180]`, *max* must be greater than *min*. Units are in degrees.

Data Types: `single` | `double`

ElevationLimits — Elevation limits of lidar sensor
`[-20 20]` (default) | two-element vector

Elevation limits of the lidar sensor, specified as a two-element vector of the form [*min max*]. The values must be in the range [-180, 180], *max* must be greater than *min*. Units are in degrees.

Data Types: single | double

ElevationAngles — Elevation angles of lidar sensor
[] (default) | *N*-element vector

Elevation angles of the lidar sensor, specified as an *N*-element vector. *N* is the number of elevation channels. The values must be in the range [-180, 180]. Units are in degrees.

Data Types: single | double

ActorProfiles — Physical characteristics of actors
[] (default) | structure | *L*-element array of structures

Physical characteristics of the actors in the scene, specified as a structure or as an *L*-element array of structures. *L* is the number of actors in the scene.

To generate an array of actor profile structures for your driving scenario, use the `actorProfiles` function. You can also create these structures manually. This table shows the valid structure fields.

Field	Description	Value
ActorID	Unique identifier for the actor. In a scene with multiple actors, this value distinguishes different actors from one another.	Positive integer
ClassID	User-defined classification ID for the actor. ClassID / Class Name table: 1 — Car 2 — Truck 3 — Bicycle 4 — Pedestrian 5 — Jersey Barrier 6 — Guardrail	Positive scalar
Length	Length of the actor in meters.	Positive scalar
Width	Width of the actor in meters.	Positive scalar
Height	Height of the actor in meters.	Positive scalar
OriginOffset	Offset of the rotational center of the actor from its geometric center. The rotational center, or origin, is located at the bottom center of the actor. For vehicles, the rotational center is the point on the ground beneath the center of the rear axle.	A three-element vector of the form [x y z]. Units are in meters.

Field	Description	Value
MeshVertices	Vertices of the actor in mesh representation.	*N*-by-3 numeric matrix, where each row defines a vertex in 3-D space.
MeshFaces	Face of the actor in mesh representation.	*M*-by-3 integer matrix, where each row represents a triangle defined by vertex IDs, which are the row numbers of MeshVertices.
MeshTargetReflectances	Material reflectance for each triangular face of the actor.	*M*-by-1 numeric vector, where *M* is the number of triangle faces of the actor. Each value must be in the range [0, 1].

For more information about these structure fields, see the `actor` and `vehicle` functions.

Usage

Syntax

```
ptCloud = lidar(tgtPoses,time)
[ptCloud,isValidTime,clusters] = lidar(tgtPoses,time)
```

Description

`ptCloud = lidar(tgtPoses,time)` generates a lidar point cloud, `ptCloud`, using the actor poses `tgtPoses` at the specified simulation time `time`. The function generates data at time intervals specified by the `UpdateRate` property of `lidarSensor` object `lidar`.

`[ptCloud,isValidTime,clusters] = lidar(tgtPoses,time)` additionally returns `isValidTime`, which indicates whether the simulation time is valid, and `clusters`, which contain the classification data of the output point cloud.

Input Arguments

tgtPoses — Actor poses in scene
L- element array of structures

Actor poses in the scene, specified as an *L*- element array of structures. Each structure corresponds to an actor. *L* is the number of actors used.

You can generate this structure using the `actorPoses` function. You can also create these structures manually. Each structure has these fields:

Field	Description	Value
ActorID	Unique identifier for the actor.	Positive scalar
Position	Position of the actor with respect to the ego vehicle coordinate system, in meters.	Vector of the form [x y z]

Field	Description	Value
Velocity	Velocity (*V*) of the actor, in meters per second, along the *x*-, *y*-, and *z*- directions.	A vector of the form [V_x V_y V_z] Default: [0 0 0]
Roll	Roll angle of the actor in degrees.	Numeric scalar Default: 0
Pitch	Pitch angle of the actor in degrees.	Numeric scalar Default: 0
Yaw	Yaw angle of the actor in degrees.	Numeric scalar Default: 0
AngularVelocity	Angular velocity (*ω*) of the actor, in degrees per second, along the *x*-, *y*-, and *z*- directions.	Vector of the form [ω_x ω_y ω_z] Default: [0 0 0]

time — Simulation time
positive scalar

Simulation time, specified as a positive scalar. The `lidarSensor` object generates new detections at the interval specified by the `UpdateRate` property. The value of the `UpdateRate` property must be an integer multiple of the simulation time interval. Updates requested from the sensor between update intervals do not generate a point cloud.

Data Types: `single` | `double`

Output Arguments

ptCloud — Point cloud data
`pointCloud` object

Point cloud data generated from the scene, returned as a `pointCloud` object.

isValidTime — Valid simulation time
0 | 1

Valid simulation time, returned as a logical 0(`false`) or 1(`true`). The value is 0 for updates requested at times between the update interval specified by the `UpdateRate` property.

clusters — Classification data of actors
M-by-*N*-by-2 array | *P*-by-2 matrix

Classification data of actors in the scene, returned as an *M*-by-*N*-by-2 array for an organized point cloud or a *P*-by-2 matrix for an unorganized point cloud. The first column contains the `ActorID`s and the second column contains the `ClassID`s of the target actors. *M*, *N* are the number of rows and columns in the organized point cloud, and *P* is the number of points in the unorganized point cloud.

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
release   Release resources and allow changes to System object property values and input
            characteristics
reset   Reset internal states of System object

Examples

Generate Lidar Point Cloud Data

Load synthetic scene data containing actor profiles and target poses generated using the
`drivingScenario` (Automated Driving Toolbox) object into the workspace.

```
sceneData = load("scene_data.mat");
sceneActorProfiles = sceneData.ActorProfiles;
sceneTargetPoses = sceneData.TargetPoses;
```

Load the target material reflectance data.

```
reflectanceData = load("scene_target_reflectances.mat");
targetReflectance = reflectanceData.TargetReflectances;
```

Define the reflectances for each actor.

```
for i = 1:numel(sceneActorProfiles)
    sceneActorProfiles(i).MeshTargetReflectances = targetReflectance{i};
end
```

Create a `lidarSensor` System object, and define the actor profiles for the object.

```
lidar = lidarSensor(AzimuthResolution=0.5);
lidar.ActorProfiles = sceneActorProfiles;
```

Create a `pcplayer` object to visualize the lidar sensor point cloud detections.

```
player = pcplayer([-100 100],[-20 20],[0 5]);
```

Generate and visualize the point cloud detections at valid simulation times.

```
for i = 1:5:numel(sceneTargetPoses)
    if(~player.isOpen)
        break
    end
    [ptCloud,isValid] = lidar(sceneTargetPoses{i},i*0.1);
    if(isValid)
        view(player,ptCloud)
    end
end
```
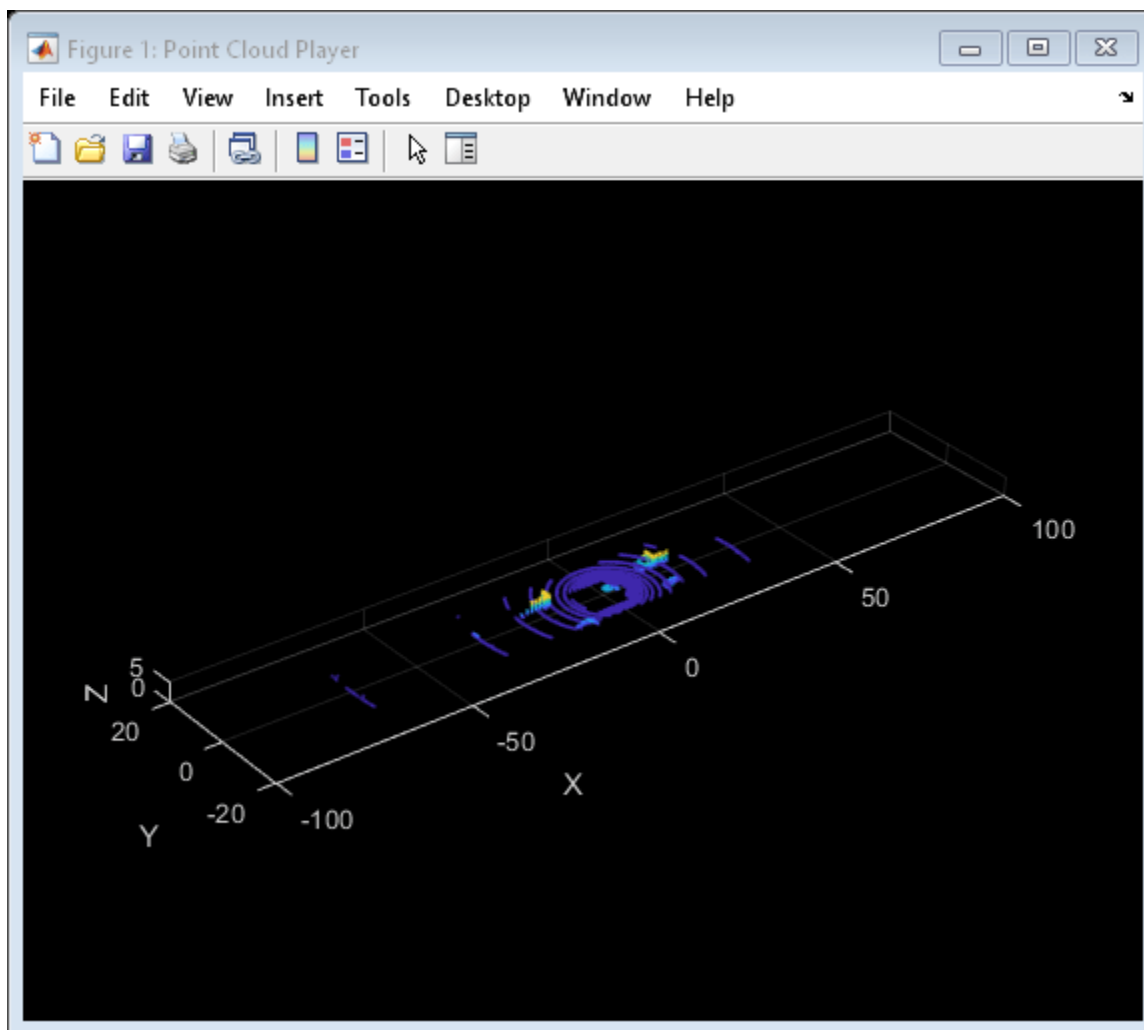
Version History
Introduced in R2022a

See Also

Apps
Lidar Labeler | **Lidar Viewer** | **Lidar Camera Calibrator**

Functions
rangeSensor | drivingScenario | actorProfiles | actorPoses

Topics
"Coordinate Systems in Lidar Toolbox"
"Generate Lidar Point Cloud Data for Driving Scenario with Multiple Actors"

LOAMPoints

Object for storing LOAM feature points

Description

The `LOAMPoints` object enables you to store lidar odometry and mapping (LOAM) feature points for registration. Use the `detectLOAMFeatures` function to detect feature points and store them in a `LOAMPoints` object. Use the `pcregisterloam` function to find the transformation between two `LOAMPoints` objects.

Creation

Syntax

points = LOAMPoints(location,label,laserID)

Description

points = LOAMPoints(location,label,laserID) constructs a `LOAMPoints` object from the specified 3-D point coordinates `location`. Specify the categorical label `label` for each point, and an identifier `laserID` for each point that relates to the laser that detected the point.

Input Arguments

`location` — Point locations
M-by-3 matrix of [*x y z*] coordinates

Point locations, specified by an *M*-by-3 matrix of [*x y z*] coordinates, where *M* is the number of points.

`label` — Categorical label of each point
M-element categorical array

Categorical label of each point, specified as an *M*-element categorical array, where *M* is the number of points. Each point can be of category `'sharp-edge'`, `'less-sharp-edge'`, `'planar-surface'`, or `'less-planar-surface'`.

`laserID` — Laser identifier
M-element vector of positive integers

Laser identifier, specified as an *M*-element vector of positive integers, where *M* is the number of points. Each element is the ID of the laser that detected the corresponding point.

Properties

`Location` — LOAM point locations
M-by-3 matrix of [*x y z*] coordinates

This property is read-only.

Point locations, specified by an *M*-by-3 matrix of [*x y z*] coordinates, where *M* is the number of points.

Label — Categorical label of each point
M-element categorical array

This property is read-only.

Categorical label of each point, specified as an *M*-element categorical array, where *M* is the number of points. Each point can be of category `'sharp-edge'`, `'less-sharp-edge'`, `'planar-surface'`, or `'less-planar-surface'`.

Count — Number of points
positive integer

This property is read-only.

Number of points, specified as a positive integer.

Object Functions

downsampleLessPlanar	Downsample less planar surface points
transformPointsForward	Apply forward geometric transformation to points
show	Visualize LOAM feature points

Examples

Detect and Visualize LOAM Feature Points

Load an organized lidar point cloud from a MAT file into the workspace.

```
ld = load("drivingLidarPoints.mat");
ptCloudOrg = ld.ptCloud;
```

Detect lidar odometry and mapping (LOAM) feature points.

```
points = detectLOAMFeatures(ptCloudOrg);
```

Visualize the LOAM points.

```
figure
show(points)
```

Version History
Introduced in R2022a
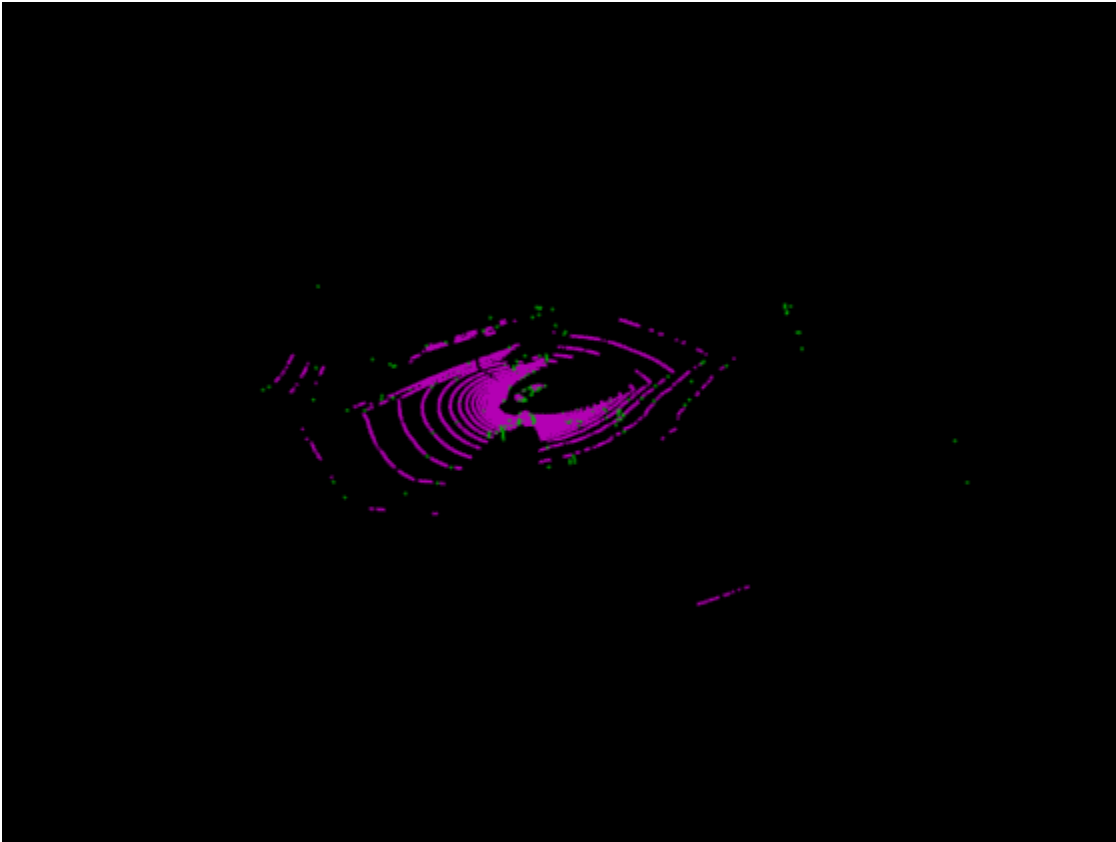
References

[1] Zhang, Ji, and Sanjiv Singh. "LOAM: Lidar Odometry and Mapping in Real-Time." In *Robotics: Science and Systems X*. Robotics: Science and Systems Foundation, 2014. https://doi.org/ 10.15607/RSS.2014.X.007.

See Also

Objects
pointCloud | rigidtform3d

Functions
detectLOAMFeatures | pcregisterloam

downsampleLessPlanar

Downsample less planar surface points

Syntax

```
pointsOut = downsampleLessPlanar(pointsIn,gridStep)
```

Description

`pointsOut = downsampleLessPlanar(pointsIn,gridStep)` downsamples the less planar surface points `pointsIn` using a box grid filter with 3-D boxes of the specified size `gridStep`. The function merges input points within the same box to a single point in the output.

To speed up LOAM registration, downsample the less planar surface points using the `downsampleLessPlanar` function, then register the LOAM points using the `pcregisterloam` function.

Examples

Downsample Less Planar Surface Points

Load an organized lidar point cloud into the MATLAB® workspace from a MAT file.

```
ld = load("drivingLidarPoints.mat");
ptCloud = ld.ptCloud;
```

Detect LOAM feature points in the point cloud.

```
points = detectLOAMFeatures(ptCloud);
```

Downsample the less planar surface points.

```
gridStep = 0.5;
pointsOut = downsampleLessPlanar(points,gridStep);
```

Visualize the downsampled LOAM points.

```
figure
show(pointsOut)
```
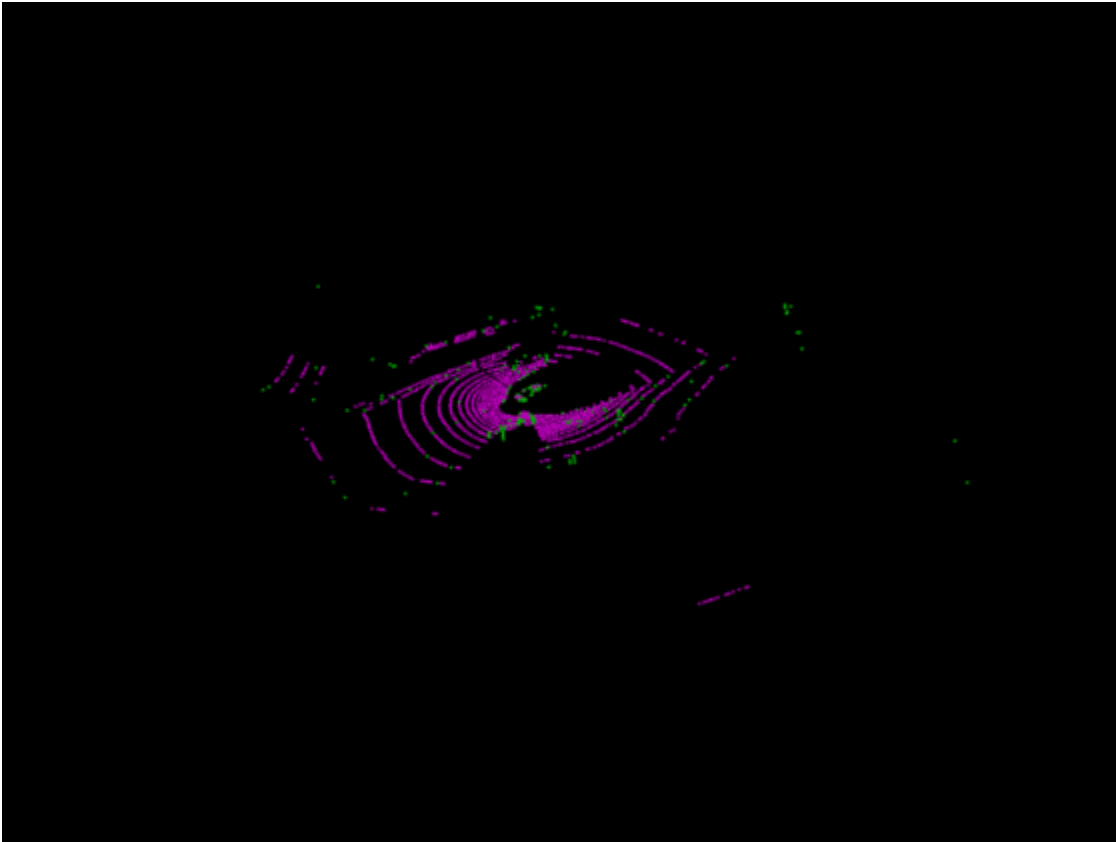
Input Arguments

pointsIn — Input points
LOAMPoints object

Input points, specified as a LOAMPoints object.

gridStep — Size of 3-D box for downsampling less planar surface points
positive scalar

Size of the 3-D box for downsampling less planar surface points, specified as a positive scalar.

Output Arguments

pointsOut — Downsampled points
LOAMPoints object

Downsampled points, returned as a LOAMPoints object.

Version History
Introduced in R2022a

See Also

Functions
detectLOAMFeatures | pcregisterloam | pcdownsample

Objects
LOAMPoints

transformPointsForward

Apply forward geometric transformation to points

Syntax

```
pointsOut = transformPointsForward(pointsIn,tform)
```

Description

`pointsOut = transformPointsForward(pointsIn,tform)` applies the forward geomentric transformation `tform` to the input `pointsIn` and returns the transformed LOAM feature points.

Examples

Transform LOAM Feature Points

Load an organized lidar point cloud from a MAT file.

```
ld = load("drivingLidarPoints.mat");
ptCloud = ld.ptCloud;
```

Detect LOAM feature points.

```
points = detectLOAMFeatures(ptCloud);
```

Define a rigid transformation object.

```
tform = rigidtform3d([0 0 0],[2 1 4]);
```

Transform the LOAM feature points.

```
tformedPoints = transformPointsForward(points,tform);
```

Visualize the transformed LOAM points.

```
figure
show(tformedPoints)
```
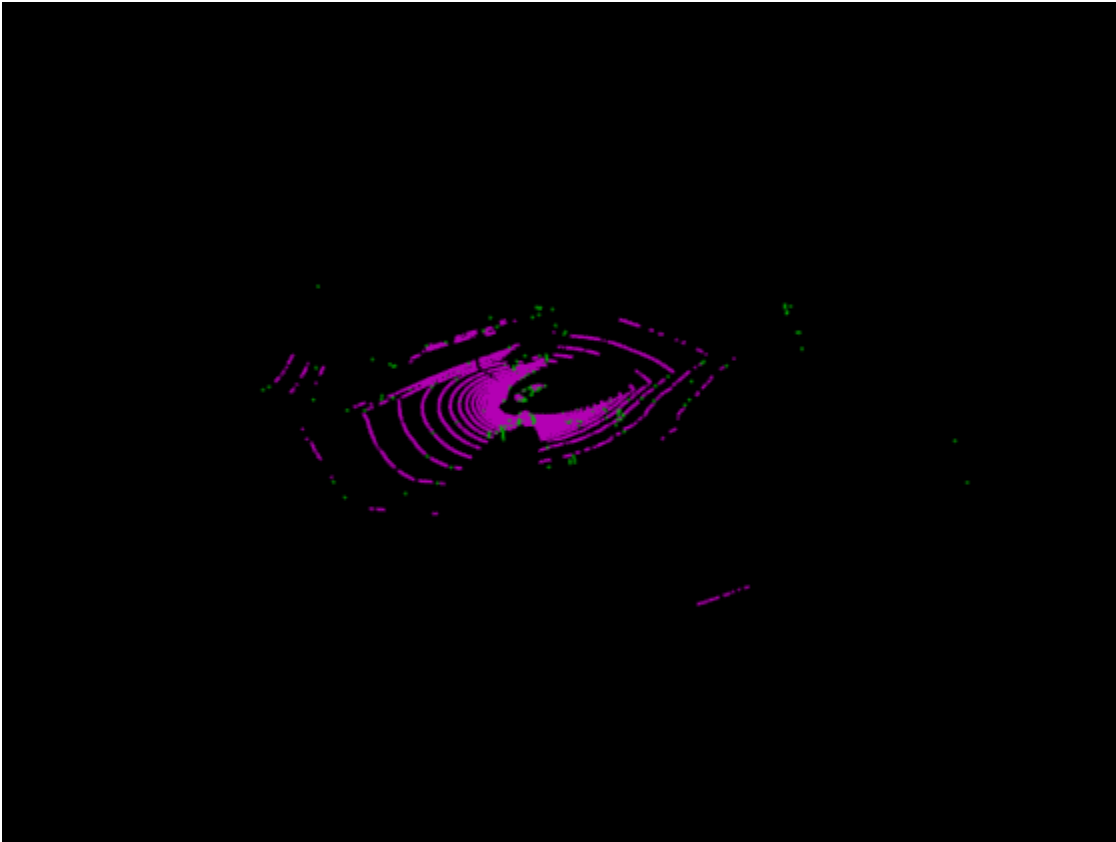
Input Arguments

`pointsIn` — Input points
`LOAMPoints` object

Input points, specified as a `LOAMPoints` object.

`tform` — Rigid 3-D transformation
`rigidtform3d` object

Rigid 3-D transformation, specified as a `rigidtform3d` object.

Output Arguments

`pointsOut` — Transformed points
`LOAMPoints` object

Transformed points, returned as a `LOAMPoints` object.

Version History
Introduced in R2022a

Supports `rigidtform3d` objects

You can now specify `tform`, as a `rigidtform3d` object, which uses the premultiply convention. Although you can still specify `tform` as a `rigid3d` object, this object is not recommended because it uses the postmultiply convention. For more information, see "Migrate Geometric Transformations to Premultiply Convention".

See Also

Functions
detectLOAMFeatures | pcregisterloam

Objects
LOAMPoints | rigidtform3d

show

Visualize LOAM feature points

Syntax

```
show(points)
show(points,Name=Value)
ax = show( ___ )
```

Description

show(`points`) displays the specified lidar odometry and mapping (LOAM) feature points. Surface points are displayed in magenta and edge points are displayed in green.

show(`points,Name=Value`) specifies additional options using one or more name-value argument. For example, `MarkerSize=5` sets the approximate diameter for the marker, in points, to `5`. Unspecified arguments have default values.

ax = show(___) returns the plot axes using any combination of input arguments from previous syntaxes.

Examples

Detect and Visualize LOAM Feature Points

Load an organized lidar point cloud from a MAT file into the workspace.

```
ld = load("drivingLidarPoints.mat");
ptCloudOrg = ld.ptCloud;
```

Detect lidar odometry and mapping (LOAM) feature points.

```
points = detectLOAMFeatures(ptCloudOrg);
```

Visualize the LOAM points.

```
figure
show(points)
```
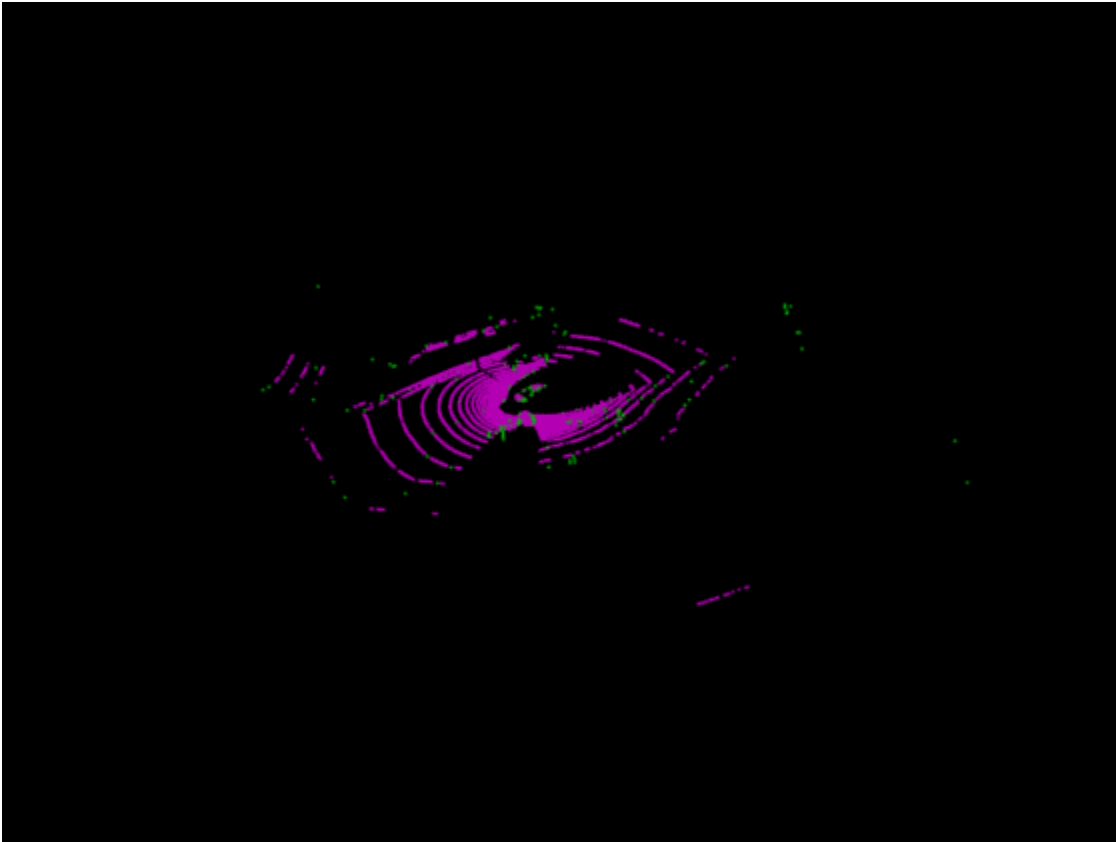
Input Arguments

`points` — Input points
`LOAMPoints` object

Input points, specified as a `LOAMPoints` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `show(points,MarkerSize=5)` sets the approximate diameter for the marker, in points, to 5.

Parent — Axes on which to display visualization
`Axes` graphics object

Axes on which to display the visualization, specified as an `Axes` object. To create an `Axes` object, use the `axes` function. To display the visualization in a new figure, leave `Parent` unspecified.

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

Diameter of the marker, specified as a positive scalar. The value specifies the approximate diameter of the point marker in points, defined by MATLAB graphics. A marker size greater than 6 can reduce rendering performance.

Output Arguments

ax — Plot axes
Axes graphics object

Plot axes, returned as an axes graphics object. You can set the default center of rotation for the point cloud viewer to the axes center a point in a plot. Set the default behavior using the "Computer Vision Toolbox Preferences".

Version History
Introduced in R2022a

See Also

Functions
detectLOAMFeatures | pcregisterloam | pcshowpair

Objects
LOAMPoints

eigenFeature

Object for storing eigenvalue-based features

Description

The `eigenFeature` object stores an eigenvalue-based feature vector extracted from point cloud data.

Creation

Syntax

```
features = eigenFeature(featureVector,centroid)
```

Description

`features = eigenFeature(featureVector,centroid)` constructs an `eigenFeature` object from the feature vector `featureVector` and the centroid `centroid`. The `featureVector` argument sets the `Feature` property, and the `centroid` argument sets the `Centroid` property.

Properties

Feature — Feature vector
seven-element vector

Feature vector, specified as a seven-element vector of the form [*linearity planarity scattering,omnivariance anisotropy eigenentropy change in curvature*].

Centroid — Centroid
three-element vector

Centroid, specified as a three-element vector in the form [*x y z*].

Examples

Create `eigenFeature` Object

Create a feature vector and set the centroid for the `eigenFeature` object.

```
featureVector = rand(1,7);
centroid = rand(1,3);
```

Create an `eigenFeature` object.

```
eFeature = eigenFeature(featureVector,centroid)

eFeature =
  eigenFeature with properties:
```

```
   Feature: [0.8147 0.9058 0.1270 0.9134 0.6324 0.0975 0.2785]
   Centroid: [0.5469 0.9575 0.9649]
```

Version History
Introduced in R2021a

See Also

Functions
extractEigenFeatures

Objects
pcmapsegmatch | pointCloud

Topics
"Build Map and Localize Using Segment Matching"
"Implement Point Cloud SLAM in MATLAB"

pcmapsegmatch

Map of segments and features for localization and loop closure detection

Description

The `pcmapsegmatch` object creates a map of segments and features, and uses the segment matching (SegMatch [1]) algorithm for place recognition. This segment matching approach is robust to dynamic obstacles and reliable on large scale environments. The object stores the features, and segments, and their corresponding view IDs. Use the view IDs to link the features to a view in the point cloud view set object, `pcviewset`, for map building. `pcmapsegmatch` implements feature matching using eigenvalue-based features. It uses the Euclidean distance between segment features to find segment matches.

Creation

Syntax

```
sMap = pcmapsegmatch
sMap = pcmapsegmatch('CentroidDistance',dist)
```

Description

`sMap = pcmapsegmatch` returns a default `pcmapsegmatch` object. Use the `addView` object function to add views and their corresponding segments and features to the map.

`sMap = pcmapsegmatch('CentroidDistance',dist)` additionally specifies the minimum distance between segment centroids when adding segments and their corresponding features to the map. Segments with centroids closer than the specified distance `dist`, are not added to the map. `dist` is specified as a positive scalar with a default value of `0.1`.

Properties

`ViewIds` — View identifier
M-element vector

This property is read-only.

View identifier, specified as an *M*-element vector of integers, where *M* is the number of views added to `pcmapsegmatch`.

`Features` — Feature vector
N-element vector of `eigenFeature` objects

This property is read-only.

Feature vector, specified as an *N*-element vector of `eigenFeature` objects, where *N* is the number of features.

Use the `addView` object function to add features for unique segments to the map. When you update the map using the `updateMap` object function, features that correspond to duplicate segments are removed from the map if they are within the `CentroidDistance`.

Segments — Point cloud segments
N-element vector of `pointCloud` objects

This property is read-only.

Point cloud segments, specified as an *N*-element vector of `pointCloud` objects, where *N* is the number of point cloud segments.

A segment is a group of 3-D points that are close together and represent a partial or full object.

SelectedSubmap — Currently selected submap
entire map (default) | 6-element vector

This property is read-only.

Currently selected submap, specified as a 6-element vector of the form [*xmin,xmax ymin ymax zmin zmax*] that describes the range of the submap along each axis. The elements of the vector describe the region of interest represented by the submap.

XLimits — Range of map along *x*-axis
2-element vector

This property is read-only.

Range of the map along the *x*-axis, specified as a 2-element vector of the form [*xmin xmax*] .

YLimits — Range of map along the *Y*-axis
2-element vector

This property is read-only.

Range of the map along the *Y*-axis, specified as a 2-element vector of the form [*ymin ymax*] .

ZLimits — Range of map along the *z*-axis
2-element vector

This property is read-only.

Range of the map along the *z*-axis, specified as a 2-element vector of the form [*zmin zmax*] .

CentroidDistance — Minimum distance between segment centroids
positive scalar

This property is read-only.

Minimum distance between segment centroids, specified as a positive scalar. The object uses the minimum distance when adding segments and corresponding features to the map as unique segments and features.

Object Functions

addView   Add view to map
deleteView   Delete view from map
findView   Retrieve feature and segment indices corresponding to map view
hasView   Check if view is in the map
deleteSegments   Delete all segments in map
findPose   Find absolute pose in map that aligns segment matches
updateMap   Update centroid and point cloud segment locations in map
selectSubmap   Select submap within map
isInsideSubmap   Check if query position is inside selected submap
show   Visualize the point cloud segments in the map

Examples

Lidar Localization Using Segment Matching

Load a map of segments and features from a MAT file into the workspace. The point cloud data in the map has been collected using the Simulation 3D Lidar (UAV Toolbox) block.

```
data = load('segmatchMapFullParkingLot.mat');
sMap = data.segmatchMapFullParkingLot;
```

Load point cloud scans from a MAT file.

```
data = load('fullParkingLotData.mat');
ptCloudScans = data.fullParkingLotData;
```

Display the map of segments.

```
ax = show(sMap);
```

Change the viewing angle to top-view.

```
view(2)
pause(0.2)
```

Set the radius for selecting a cylindrical neighborhood.

```
outerCylinderRadius = 20;
innerCylinderRadius = 3;
```

Set the threshold parameters for segmentation.

```
distThreshold = 0.5;
angleThreshold = 180;
```

Set the size and submap threshold parameters for the selected submap

```
sz = [65 30 20];
submapThreshold = 10;
```

Set the radius parameter for visualization.

```
radius = 0.5;
```

Segment each point cloud and localize by finding segment matches.

```matlab
for n = 1:numel(ptCloudScans)
    ptCloud = ptCloudScans(n);

    % Segment and remove the ground plane.
    groundPtsIdx = segmentGroundFromLidarData(ptCloud,'ElevationAngleDelta',11);
    ptCloud = select(ptCloud,~groundPtsIdx,'OutputSize','full');

    % Select the cylindrical neighborhood.
    dists = sqrt(ptCloud.Location(:,:,1).^2 + ptCloud.Location(:,:,2).^2);
    cylinderIdx = dists <= outerCylinderRadius & dists > innerCylinderRadius;
    ptCloud = select(ptCloud,cylinderIdx,'OutputSize','full');

    % Segment the point cloud.
    labels = segmentLidarData(ptCloud,distThreshold,angleThreshold,'NumClusterPoints',[50 5000]);

    % Extract features from the point cloud.
    [features,segments] = extractEigenFeatures(ptCloud,labels);

    % Localize by finding the absolute pose in the map that aligns the segment matches.
    [absPoseMap,~,inlierFeatures,inlierSegments] = findPose(sMap,features,segments);

    if isempty(absPoseMap)
        continue;
    end

    % Display the position estimate in the map.
    poseTranslation = absPoseMap.Translation;
    pos = [poseTranslation(1:2) radius];
    showShape('circle',pos,'Color','r','Parent',ax);
    pause(0.2)

    % Determine if the selected submap needs to be updated.
    [isInside,distToEdge] = isInsideSubmap(sMap,poseTranslation);
    needSelectSubmap = ~isInside ...                       % Current pose is outside submap
        || any(distToEdge(1:2) < submapThreshold) ...      % Current pose is close to submap edge
        || n == 1;                                         % 1st time localizing using whole map

    % Select a new submap.
    if needSelectSubmap
        sMap = selectSubmap(sMap,poseTranslation,sz);
    end
end
```

```
% Visualize the last segment matches.
figure;
pcshowMatchedFeatures(inlierSegments(:,1),inlierSegments(:,2),inlierFeatures(:,1),inlierFeatures
```

Version History
Introduced in R2021a

References

[1] Dube, Renaud, Daniel Dugas, Elena Stumm, Juan Nieto, Roland Siegwart, and Cesar Cadena. "SegMatch: Segment Based Place Recognition in 3D Point Clouds." In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 5266–72. Singapore, Singapore: IEEE, 2017. https://doi.org/10.1109/ICRA.2017.7989618.

See Also

Functions
segmentLidarData | segmentGroundFromLidarData | pcsegdist | pcshowMatchedFeatures | extractEigenFeatures

Objects
pcmapndt | pcviewset

Topics
"Build Map and Localize Using Segment Matching"

addView

Add view to map

Syntax

```
sMapOut = addView(sMapIn,viewId,features)
sMapOut = addView(sMapIn,viewId,features,segments)
```

Description

sMapOut = addView(sMapIn,viewId,features) adds a view, viewId, that contains the specified features features to the map sMapIn.

sMapOut = addView(sMapIn,viewId,features,segments) adds the segments segments that correspond to each feature.

Examples

Add Features and Segments to a Map

Create a map representation to hold point cloud segments and features.

```
sMap = pcmapsegmatch('CentroidDistance',1);
```

Load point cloud scans.

```
data = load('fullParkingLotData.mat');
ptCloudScans = data.fullParkingLotData;
```

Set the radius to select a cylindrical neighborhood.

```
outerCylinderRadius = 30;
innerCylinderRadius = 3;
```

Set the threshold parameters for segmentation.

```
distThreshold = 0.5;
angleThreshold = 180;
```

Segment each point cloud and add the features and point cloud segments to the map.

```
for n = 1:numel(ptCloudScans);
    ptCloud = ptCloudScans(n);

    % Segment and remove the ground plane.
    groundPtsIdx = segmentGroundFromLidarData(ptCloud,'ElevationAngleDelta',11);
    ptCloud = select(ptCloud,~groundPtsIdx,'OutputSize','full');

    % Select cylindrical neighborhood.
    dists = sqrt(ptCloud.Location(:,:,1).^2 + ptCloud.Location(:,:,2).^2);
    cylinderIdx = dists <= outerCylinderRadius ...
```

```
        & dists > innerCylinderRadius;
    ptCloud = select(ptCloud,cylinderIdx,'OutputSize','full');

    % Segment the point cloud.
    [labels, numClusters] = segmentLidarData(ptCloud,distThreshold,angleThreshold,'NumClusterPoi

    % Extract features from the point cloud.
    [features,segments] = extractEigenFeatures(ptCloud,labels);

    % Add the features and segments to the map.
    sMap = addView(sMap,n,features,segments);
end
```

Display the map of segments.

```
figure; show(sMap);
```
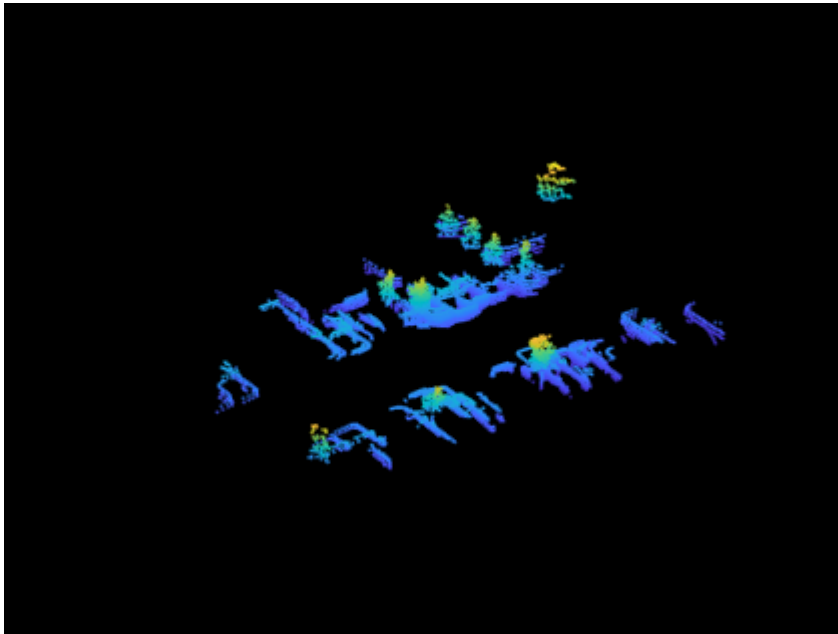

Input Arguments

sMapIn — Original map of segments and features
pcmapsegmatch object

Original map of segments and features, specified as a `pcmapsegmatch` object.

viewId — View identifier
positive integer

View identifier, specified as an integer. Each view identifiers is unique to a specific view.

features — Eigenvalue-based features
vector of eigenFeature objects

Eigenvalue-based features, specified as a vector of `eigenFeature` objects. The function filters out features that already exist in the map are filtered out as duplicates based on their centroid location and the distance specified by the `CentroidDistance` property of the map.

You should extract new features from only a point cloud registered to the point clouds of existing features

segments — Point cloud segments
vector of `pointCloud` objects

Point cloud segments, specified as a vector of `pointCloud` objects. To use the `show` object function for visualization, you must specify this argument.

For improved performance, do not include `segments` in the map with `findPose` and `updateMap` object functions. Alternatively, you can use the `deleteSegment` object function to remove the existing segments before using `findPose` or `updateMap`.

Output Arguments

sMapOut — Updated map of segments and features
`pcmapsegmatch` object

Updated map of segments and features, returned as a `pcmapsegmatch` object.

Version History
Introduced in R2021a

See Also

Functions
`findPose` | `findView`

Objects
`pcmapsegmatch` | `eigenFeature`

deleteSegments

Delete all segments in map

Syntax

```
sMapOut = deleteSegments(sMapIn)
```

Description

sMapOut = deleteSegments(sMapIn) deletes all segments in the map sMapIn. Removing the segments from the map improves the performance of the findPose and updateMap object functions.

Examples

Delete Segments Segment Map

Load a map of segments and features from a MAT file.

```
data = load('segmatchMapFullParkingLot.mat');
sMap = data.segmatchMapFullParkingLot;
```

Remove the segments from the map, leaving only the corresponding features in the map.

```
sMapNoSegments = deleteSegments(sMap);
```

Verify the number of segments in the map before and after removal.

```
numBefore = numel(sMap.Segments);
numAfter = numel(sMapNoSegments.Segments);
disp("Number of Segments Before Deleting Segments: " + num2str(numBefore))
```

```
Number of Segments Before Deleting Segments: 464
```

```
disp("Number of Segments After Deleting Segments: " + num2str(numAfter))
```

```
Number of Segments After Deleting Segments: 0
```

Input Arguments

sMapIn — Original map of segments and features
pcmapsegmatch object

Original map of segments and features, specified as a pcmapsegmatch object.

Output Arguments

sMapOut — Updated map of segments and features
pcmapsegmatch object

Updated map of segments and features, returned as a `pcmapsegmatch` object.

Version History
Introduced in R2021a

See Also

Objects
`pcmapsegmatch`

Functions
`findPose` | `updateMap`

deleteView

Delete view from map

Syntax

```
sMapOut = deleteView(sMapIn,viewIds)
```

Description

sMapOut = deleteView(sMapIn,viewIds) deletes the specified views viewIds, along with their corresponding features and segments.

Examples

Delete Views from Map

Load a map of segments and features from a MAT file into the workspace.

```
data = load('segmatchMapFullParkingLot.mat');
sMap = data.segmatchMapFullParkingLot;
```

Visualize the map.

```
figure
show(sMap)
title('Map Before Deleting Views')
```
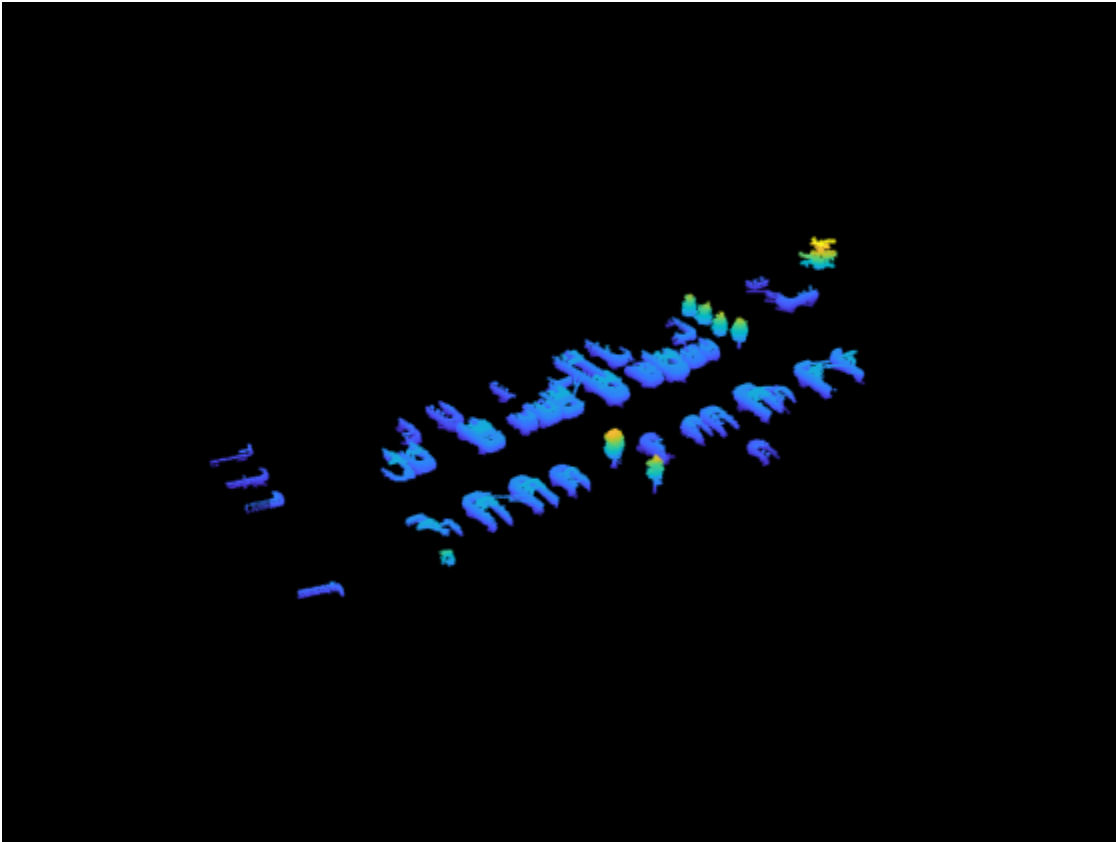
Delete the first 50 views from the map.

```
viewIds = 1:50;
sMap = deleteView(sMap,viewIds);
```

Visualize the map after deleting the views.

```
figure
show(sMap)
title('Map After Deleting the First 50 Views')
```
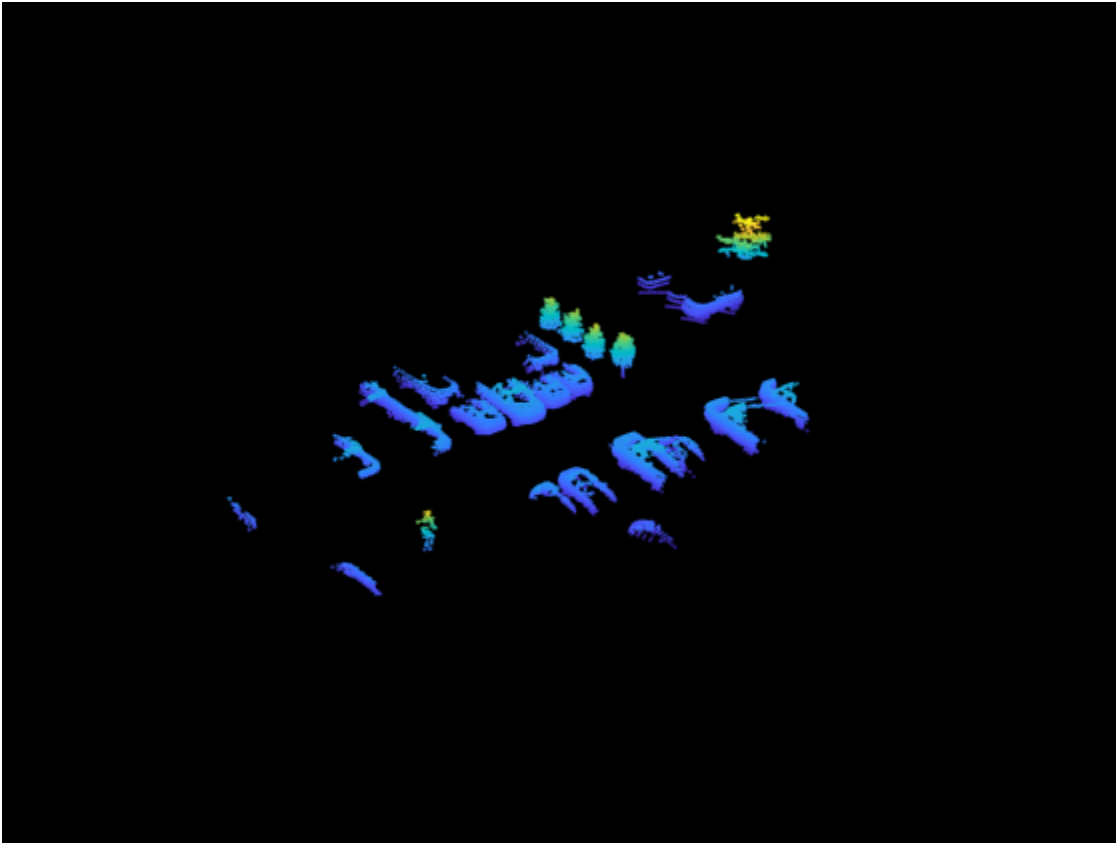
Input Arguments

sMapIn — Original map of segments and features
pcmapsegmatch object

Original map of segments and features, specified as a `pcmapsegmatch` object.

viewIds — View identifiers
M-element vector

View identifiers, specified as an *M*-element vector. *M* is the number of views to delete. Each view identifier is unique to a specific view.

Output Arguments

sMapOut — Updated map of segments and features
pcmapsegmatch object

Updated map of segments and features, returned as a `pcmapsegmatch` object.

Version History
Introduced in R2021a

See Also

Functions
deleteSegments | addView

Objects
pcmapsegmatch

findPose

Find absolute pose in map that aligns segment matches

Syntax

```
absPoseMap = findPose(sMap,refPose)
[absPoseMap,matchViewId] = findPose(sMap,refPose)

absPoseMap = findPose(sMap,currentFeatures)
absPoseMap = findPose(sMap,currentFeatures,currentSegments)

[ ___ ,inlierFeatures,inlierSegments] = findPose( ___ )

[ ___ ] = findPose( ___ ,Name,Value)
```

Description

Map Building

`absPoseMap = findPose(sMap,refPose)` finds the absolute pose of the last added view that aligns the segment matches of the detected loop closure. The function looks for segment matches between the last added view and the segment features inside the submap specified by the `SelectedSubmap` property of `sMap`.

`[absPoseMap,matchViewId] = findPose(sMap,refPose)` returns the view identifier for the view that contains the most inliers. Use `matchViewId` to add the loop closure as a connection in a `pcviewset`, using the `addConnection` object function. Correct for accumulated drift using `optimizePoses`.

Localization

`absPoseMap = findPose(sMap,currentFeatures)` finds the absolute pose that aligns the segments that correspond to the current features `currentFeatures` to the segments in the submap specified by the `SelectedSubmap` property of `sMap`.

`absPoseMap = findPose(sMap,currentFeatures,currentSegments)` specifies the segments `currentSegments` that correspond to the current features `currentFeatures`.

Visualization

`[ ___ ,inlierFeatures,inlierSegments] = findPose( ___ )` returns the inlier features `inlierFeatures` and inlier segments `inlierSegments` in addition to any combination of arguments from previous syntaxes.

Optional Name-Value Arguments

`[ ___ ] = findPose( ___ ,Name,Value)` specifies options using one or more name-value arguments in addition to the input arguments in previous syntaxes. For example, `'MaxThreshold',1.5` sets the matching threshold to 1.5 percent.

Examples

Lidar Localization Using Segment Matching

Load a map of segments and features from a MAT file into the workspace. The point cloud data in the map has been collected using the Simulation 3D Lidar (UAV Toolbox) block.

```
data = load('segmatchMapFullParkingLot.mat');
sMap = data.segmatchMapFullParkingLot;
```

Load point cloud scans from a MAT file.

```
data = load('fullParkingLotData.mat');
ptCloudScans = data.fullParkingLotData;
```

Display the map of segments.

```
ax = show(sMap);
```

Change the viewing angle to top-view.

```
view(2)
pause(0.2)
```

Set the radius for selecting a cylindrical neighborhood.

```
outerCylinderRadius = 20;
innerCylinderRadius = 3;
```

Set the threshold parameters for segmentation.

```
distThreshold = 0.5;
angleThreshold = 180;
```

Set the size and submap threshold parameters for the selected submap

```
sz = [65 30 20];
submapThreshold = 10;
```

Set the radius parameter for visualization.

```
radius = 0.5;
```

Segment each point cloud and localize by finding segment matches.

```
for n = 1:numel(ptCloudScans)
    ptCloud = ptCloudScans(n);

    % Segment and remove the ground plane.
    groundPtsIdx = segmentGroundFromLidarData(ptCloud,'ElevationAngleDelta',11);
    ptCloud = select(ptCloud,~groundPtsIdx,'OutputSize','full');

    % Select the cylindrical neighborhood.
    dists = sqrt(ptCloud.Location(:,:,1).^2 + ptCloud.Location(:,:,2).^2);
    cylinderIdx = dists <= outerCylinderRadius & dists > innerCylinderRadius;
    ptCloud = select(ptCloud,cylinderIdx,'OutputSize','full');

    % Segment the point cloud.
    labels = segmentLidarData(ptCloud,distThreshold,angleThreshold,'NumClusterPoints',[50 5000])

    % Extract features from the point cloud.
```

```
    [features,segments] = extractEigenFeatures(ptCloud,labels);

    % Localize by finding the absolute pose in the map that aligns the segment matches.
    [absPoseMap,~,inlierFeatures,inlierSegments] = findPose(sMap,features,segments);

    if isempty(absPoseMap)
        continue;
    end

    % Display the position estimate in the map.
    poseTranslation = absPoseMap.Translation;
    pos = [poseTranslation(1:2) radius];
    showShape('circle',pos,'Color','r','Parent',ax);
    pause(0.2)

    % Determine if the selected submap needs to be updated.
    [isInside,distToEdge] = isInsideSubmap(sMap,poseTranslation);
    needSelectSubmap = ~isInside ...                        % Current pose is outside submap
        || any(distToEdge(1:2) < submapThreshold) ...   % Current pose is close to submap edge
        || n == 1;                                        % 1st time localizing using whole map

    % Select a new submap.
    if needSelectSubmap
        sMap = selectSubmap(sMap,poseTranslation,sz);
    end
end
```

```
% Visualize the last segment matches.
figure;
pcshowMatchedFeatures(inlierSegments(:,1),inlierSegments(:,2),inlierFeatures(:,1),inlierFeatures
```


Input Arguments

sMap — Map of segments and features
pcmapsegmatch object

Map of segments and features, specified as a pcmapsegmatch object.

refPose — Reference pose of last added view
rigidtform3d object

Reference pose of the last added view, specified as a rigidtform3d object. The reference pose is the estimated absolute pose used to transform the point cloud from the sensor frame to the world frame for feature extraction.

currentFeatures — Current features
M-element vector of eigenFeature objects

Current features, specified as an *M*-element vector of eigenFeature objects.

currentSegments — Current segments
M-element vector of pointCloud objects

Current segments, specified as an *M*-element vector of `pointCloud` objects.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `MatchThreshold=1.5` sets the matching threshold to 1.5 percent.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'MatchThreshold',1.5` sets the matching threshold to 1.5 percent.

MatchThreshold — Matching threshold
`1.5` (default) | scalar in range (0, 100]

Matching threshold, specified as a scalar in the range (0, 100]. The threshold is the maximum percentage of the distance from a perfect match. The function classifies segments are classified as possible matches if the distance between their feature vectors is lower than the threshold.

MinNumInliers — Minimum number of inliers
`4` (default) | scalar

Minimum number of inliers, specified as a scalar greater than or equal to 3. Decreasing this value can result in false positives. If the number of detected inliers is less than `'MinNumInliers'`, the function returns an empty output for `absPoseMap` .

NumExcludedViews — Number of most recently added views to exclude
`auto` (default) | integer

Number of most recently added views to exclude, specified as an integer. For loop closure detection, exclude the most recently added views to avoid matches against the most recent features. Specify a larger value for this argument if many consecutive views correspond to the same area, such as scans from a slow-moving vehicle.

The function uses a default value of `10` for map building and `0` for localization.

MaxDistance — Maximum distance for inlier centroid match
`1` (default) | positive numeric scalar

Maximum distance for inlier centroid match, specified as a positive numeric scalar. This value is the maximum distance that a centroid can differ from the projected location of its centroid match to be considered an inlier in the geometric verification step.

NumNearestNeighbor — Number of closest features selected as feature match candidates
`100` (default) | positive integer

Number of closest features selected as feature match candidates, specified as a positive integer. For each feature in the last added view, or in the current features `currentFeatures`, the function selects the closest `'NumNearestNeighbor'` features as candidate feature matches. Specify a larger value for this argument for maps with numerous similar features.

NumSelectedClusters — Number of feature clusters to check for matches
`Inf` (default) | positive integer

Number of feature clusters to check for matches, specified as a positive integer. The function clusters candidate features based on their centroid locations. If you specify `refPose`, then the `findPose` function selects the clusters closest to the centroids of the last added view `currentFeatures`. Decrease this value to improve performance at the expense of increasing the likelihood of false negatives.

Output Arguments

absPoseMap — Absolute pose in the map
`rigidtform3d` object

Absolute pose in the map, returned as a `rigidtform3d` object. This object specifies the absolute pose that aligns the segment matches.

matchViewId — View identifier containing most inlier matches
integer

View identifier containing the most inlier matches, returned as an integer. The inliers used to compute the absolute pose map can come from several views.

inlierFeatures — Inlier features
N-by-2 matrix of `eigenFeature` objects

Inlier features, returned as an *N*-by-2 matrix of `eigenFeature` objects. The first column corresponds to the inliers in the map, and the second column corresponds to the inliers in the last added view or the current features input.

inlierSegments — Inlier segments
N-by-2 matrix of `pointCloud` objects

Inlier segments, returned as an *N*-by-2 matrix of `pointCloud` objects. The first column corresponds to the inliers in the map, and the second column corresponds to the inliers in the last added view or the current segments input.

Tips

* Removing the segments from the map using `deleteSegments`, before using the `findPose` function, can improve performance.

Algorithms

`findPose` finds the absolute pose of a segmented point cloud using the SegMatch [1 on page 2-135] algorithm for place recognition. It uses the Euclidean distance between segment features to find segment matches. The function finds the matches between the segments of interest and the segments in the map, and returns the absolute pose that aligns the segment matches in the map.

* **Map Building: Loop Closure Detection** — Loop closure starts with finding the absolute pose by finding the segment matches between the last added view and the segment features in the selected submap, which is specified by the `SelectedSubmap` property of the map.

  The last added view corresponds to a loop closure when the `findPose` function can estimate a valid geometric transformation. If the function cannot estimate this transformation, then the function returns an empty value for `absPoseMap`.

- **Map Building: Correct Drift** — To correct for drift, add the view that contains the most inliers for loop closure as a connection to the point cloud view set `pcviewset` object as a connection using the `addConnection` object function. Use the `optimizePoses` function to correct for accumulated drift.

- **Localization** — To find the absolute pose of the point cloud in the map, the function looks for segment matches between the current features `currentFeatures` and the submap specified by the `SelectedSubmap` property of `sMap`. If it cannot estimate a valid geometric transformation cannot be estimated, the function returns an empty value for the `absPoseMap` output argument.

- **Visualization** — Use the `inlierFeatures` and `inlierSegments` output arguments with the `pcshowMatchedFeatures` function to visualize the segment matches between the features and segments included in the map.

Version History
Introduced in R2021a

Supports `rigidtform3d` objects
Behavior changed in R2022b

You can now specify `refPose` as a `rigidtform3d` object, which uses the premultiply convention. Although you can still specify `refPose` as a `rigid3d` object, this object is not recommended because it uses the postmultiply convention. For more information, see "Migrate Geometric Transformations to Premultiply Convention".

When you use a syntax that includes the `refPose` argument, the `findPose` function returns `absPoseMap` as an object of the same type. When you use a syntax that includes the `currentFeatures` argument, the `findPose` function now returns `absPoseMap` as a `rigidtform3d` object. Before, the function returned `absPoseMap` as a `rigid3d` object.

References

[1] Dube, Renaud, Daniel Dugas, Elena Stumm, Juan Nieto, Roland Siegwart, and Cesar Cadena. "SegMatch: Segment Based Place Recognition in 3D Point Clouds." In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 5266–72. Singapore, Singapore: IEEE, 2017. https://doi.org/10.1109/ICRA.2017.7989618.

See Also

Objects
pcmapsegmatch | rigidtform3d | pcviewset | pointCloud | eigenFeature

Functions
updateMap | estgeotform3d | extractEigenFeatures | pcshowMatchedFeatures | segmentLidarData | pcsegdist

Topics
"Build Map and Localize Using Segment Matching"

findView

Retrieve feature and segment indices corresponding to map view

Syntax

```
idx = findView(sMap,viewIds)
```

Description

`idx = findView(sMap,viewIds)` retrieves the indices of the features and segments that correspond to the specified views `viewIds`.

Examples

Select Segments from Specific Views

Load a map of segments and features into the workspace.

```
data = load('segmatchMapFullParkingLot.mat');
sMap = data.segmatchMapFullParkingLot;
```

Retrieve the feature and segment indices corresponding to specific views.

```
viewIds = 20:25;
idx = findView(sMap,viewIds);
```

Select the segments that correspond to these views.

```
segments = sMap.Segments(idx);
```

Visualize the segments.

```
ptCloud = pccat(segments);
figure
pcshow(ptCloud)
```
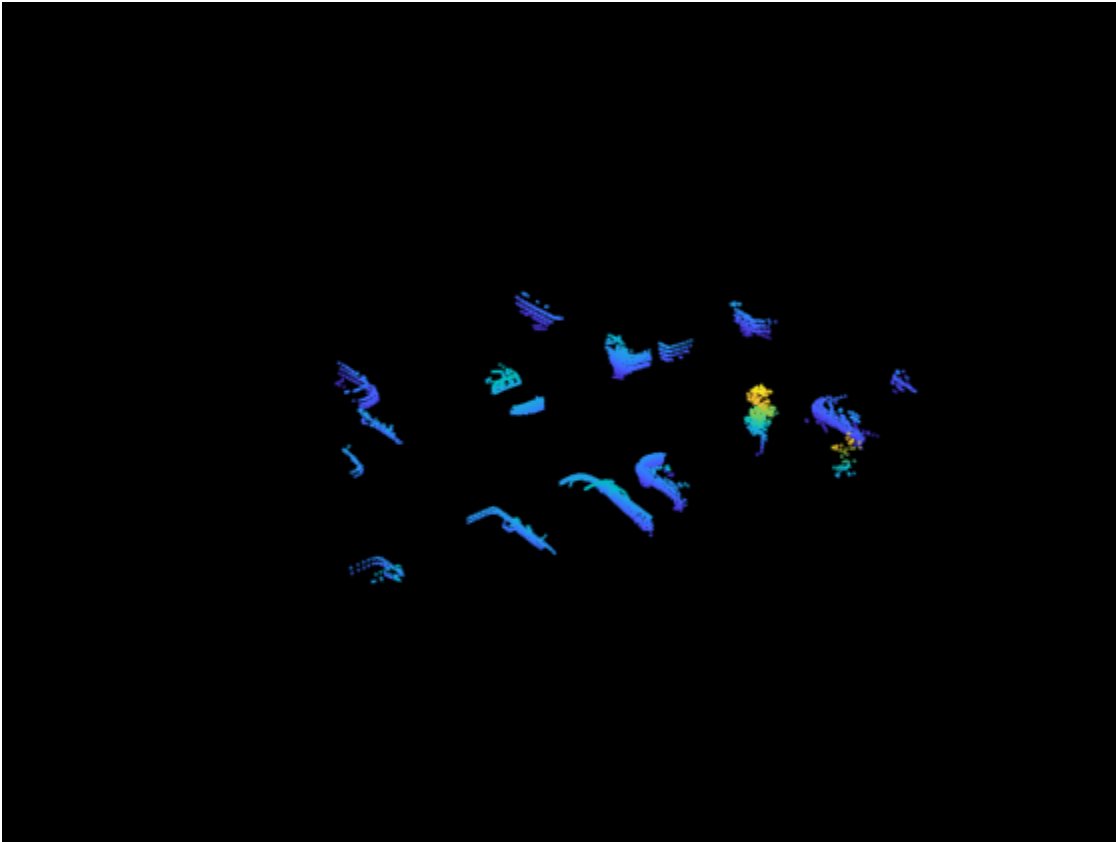
Input Arguments

sMap — Map of segments and features
pcmapsegmatch object

Map of segments and features, specified as a `pcmapsegmatch` object.

viewIds — View identifiers
M-element vector

View identifiers, specified as an *M*-element vector. *M* is the number of views to delete. Each view identifier is unique to a specific view.

Output Arguments

idx — Indices of features and segments in specified views
N-element vector

Indices of the index to features and segments in the specified views, returned as an *N*-element vector. *N* is the total number of features and segments in the map. If an element of `idx` is 1 (`true`), then the corresponding feature belongs to a specified view.

Version History
Introduced in R2021a

See Also

Functions
addView | hasView

Objects
pcmapsegmatch

hasView

Check if view is in the map

Syntax

```
tf = hasView(sMap,viewIds)
```

Description

`tf = hasView(sMap,viewIds)` checks if the views specified by `viewIds` are in the map.

Examples

Check if Views Exist

Load a map of segments and features from a MAT file.

```
data = load('segmatchMapFullParkingLot.mat');
sMap = data.segmatchMapFullParkingLot;
```

Specify a set of indices for views.

```
viewIds = [10,500,2,100];
```

Check if the specified indices correspond to existing view identifiers.

```
tf = hasView(sMap,viewIds)
```

```
tf = 1x4 logical array

   1   0   1   0
```

Input Arguments

sMap — Map of segments and features
`pcmapsegmatch` object

Map of segments and features, specified as a `pcmapsegmatch` object.

viewIds — View identifiers
M-element vector

View identifiers, specified as an *M*-element vector of integers. *M* is the number of views to delete. Each view identifier is unique to a specific view.

Output Arguments

tf — Views that exist in map
M-element vector

Views that exist in map, returned as an *M*-element vector. The function returns a value of 1 (true) if the view specified in the corresponding element of view Ids is in the map. The function returns 0 (false) if the view is not in the map.

Version History
Introduced in R2021a

See Also

Objects
pcmapsegmatch

Functions
deleteView

isInsideSubmap

Check if query position is inside selected submap

Syntax

```
isInside = isInsideSubmap(sMap,pos)
[isInside,distToEdge] = isInsideSubmap(sMap,pos)
```

Description

`isInside = isInsideSubmap(sMap,pos)` check if the query position `pos`, is inside the selected submap of the map `sMap`.

`[isInside,distToEdge] = isInsideSubmap(sMap,pos)` also returns the distance from the query position to the closest edge of the submap along the *X-,Y-*, and *Z*-axes respectively.

Examples

Check If Positions Are in Selected Submap

Load a map of segments and features from a MAT file.

```
data = load('segmatchMapFullParkingLot.mat');
sMap = data.segmatchMapFullParkingLot;
```

Select a submap within the map.

```
center = [0 30 0];
sz = [40 24 10];
sMap = selectSubmap(sMap,center,sz);
```

Check three positions to see if they are inside the submap.

```
pos1 = [0 30 0]; % center
[isInside1,distToEdge1] = isInsideSubmap(sMap,pos1)

isInside1 = logical
   1


distToEdge1 = 1x3 single row vector

   20.0000   12.0000    0.0649



pos2 = [60 0 0]; % completely outside
[isInside2,distToEdge2] = isInsideSubmap(sMap,pos2)

isInside2 = logical
   0
```

```
distToEdge2 = 1x3 single row vector

    40.0000    18.0000     0.0649
```

```
pos3 = [15 30 0]; % inside, 5 meters from edge in x direction
[isInside3,distToEdge3] = isInsideSubmap(sMap,pos3)

isInside3 = logical
    1


distToEdge3 = 1x3 single row vector

     5.0000    12.0000     0.0649
```

Input Arguments

sMap — Map of segments and features
pcmapsegmatch object

Map of segments and features, specified as a `pcmapsegmatch` object.

pos — Query position
3-element vector

Query position, specified as a 3-element vector of the form [*x y z*].

Output Arguments

isInside — Indication of position inside submap
true | false

Indication of position inside submap, returned as a logical `true` or `false`.

distToEdge — Distance from the query position to closest edge of the submap
3-element vector

Distance from the query position to the closest edge of the submap in the *X*-, *Y*-, and *Z*-axes respectively, returned as a 3-element vector.

Version History
Introduced in R2021a

See Also

Objects
pcmapsegmatch

Functions
selectSubmap | findPose

selectSubmap

Select submap within map

Syntax

```
sMapOut = selectSubmap(sMapIn,roi)
sMapOut = selectSubmap(sMapIn,center,sz)
```

Description

`sMapOut = selectSubmap(sMapIn,roi)` selects a submap within the `sMapIn` using the specified region of interest `roi`.

Use this function to confine the search space for localization using coarse position estimates.

`sMapOut = selectSubmap(sMapIn,center,sz)` selects the submap specified by the `center` and size `sz` of the submap.

Examples

Select and Visualize Submap

Load a segment map from a MAT file into the workspace.

```
data = load('segmatchMapFullParkingLot.mat');
sMap = data.segmatchMapFullParkingLot;
```

Visualize the currently selected submap.

```
figure
show(sMap,'submap')
title('Initial Selected Submap')
```
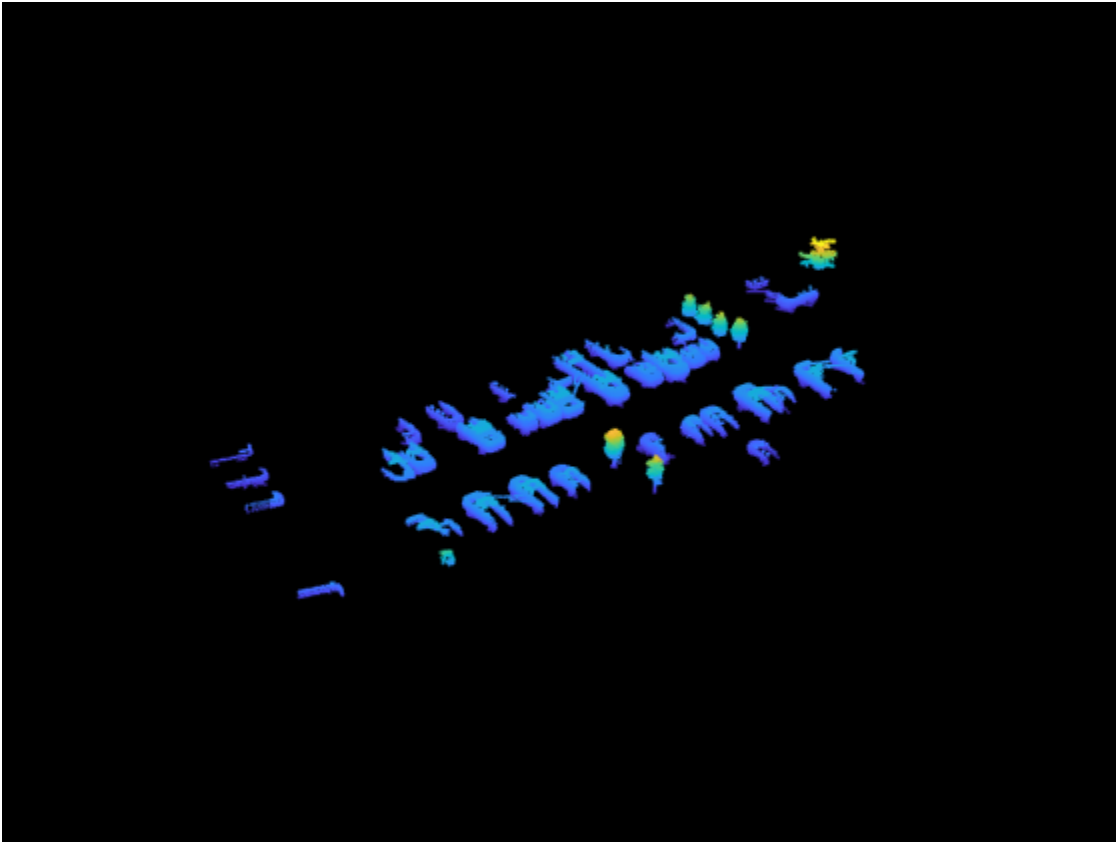
Select a new submap within the map.

```
center = [0 30 0];
sz = [40 25 10];
sMap = selectSubmap(sMap,center,sz);
```

Visualize the selected submap.

```
figure
show(sMap,'submap')
title('New Selected Submap')
```

Input Arguments

sMapIn — Original map of segments and features
pcmapsegmatch object

Original map of segments and features, specified as a `pcmapsegmatch` object.

roi — Region of interest
6-element vector

Region of interest, specified as a 6-element vector of the form [*xmin xmax ymin ymax zmin zmax*] .

center — Center of submap
3-element vector

Center of the submap, specified as 3-element vector of the form [*xc yc zc*].

sz — Size of submap along each axis
3-element vector

Size of the submap along each axis, specified as 3-element vector of the form [*xsz ysz zsz*].

Output Arguments

sMapOut — Updated map of segments and features
`pcmapsegmatch` object

Updated map of segments and features, returned as a `pcmapsegmatch` object with the updated `SelectedSubmap` property.

Tips

* Use a submap size large enough to include the uncertainty of the position estimates and the range of the sensor used withe `findPose`.A larger submap can increase computation time during each call to the `findPose` function, but it can reduce the frequency of submap updates.

Version History
Introduced in R2021a

See Also

Objects
`pcmapsegmatch`

Functions
`isInsideSubmap` | `findPose`

show

Visualize the point cloud segments in the map

Syntax

```
show(sMap)
show(sMap,spatialExtent)

show( ___ ,Name,Value)

ax = show( ___ )
```

Description

show(sMap) displays the point cloud segments in the map.

show(sMap,spatialExtent) displays point cloud segments within the spatial map or submap specified by spatialExtent.

show(___ ,Name,Value) specifies options using one or more name-value arguments in addition to any combination of input arguments in previous syntaxes. For example, 'MarkerSize',6 sets the marker size to 6 points.

ax = show(___) returns the axes used to plot the point cloud segments specified with previous syntaxes.

Examples

Visualize Full Map and Selected Submap

Load a map of segments and features from a MAT file into the workspace.

```
data = load('segmatchMapFullParkingLot.mat');
sMap = data.segmatchMapFullParkingLot;
```

Select a submap within the map.

```
center = [0 30 0];
sz = [40 25 8];
sMap = selectSubmap(sMap,center,sz);
```

Visualize the full map.

```
figure
show(sMap)
title('Full Map')
```

Highlight the selected submap on the full map.

```
pos = [center sz zeros(1,3)];
showShape('cuboid',pos,'Color','y','Opacity',0.2);
```
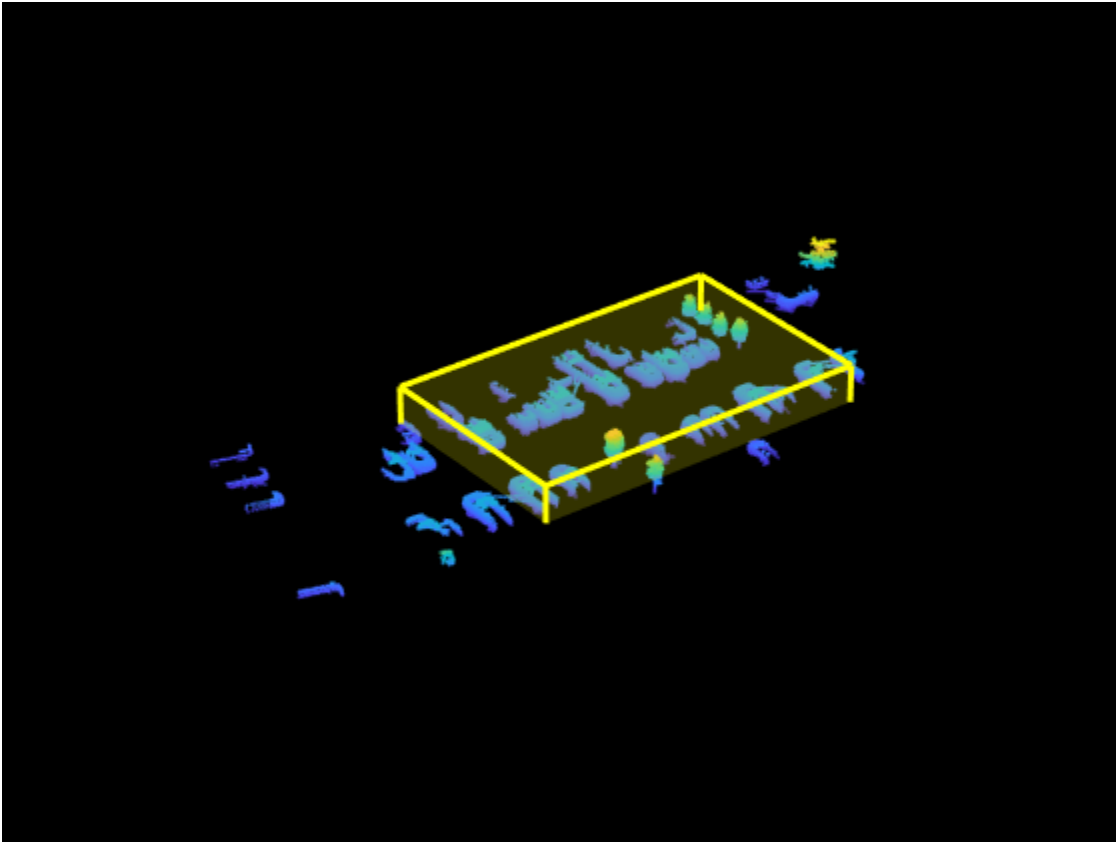
Visualize the selected submap.

```
figure
show(sMap,'submap')
title('Selected Submap')
```

Input Arguments

sMap — Map of segments and features
`pcmapsegmatch` object

Map of segments and features, specified as a `pcmapsegmatch` object.

spatialExtent — Spatial extent
`'map'` | `'submap'`

Spatial extent, specified as `'map'` or `'submap'`. When you specify `'submap'`, only points within the current submap are displayed.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'MarkerSize'`,6 sets the marker size to 6 points.

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

Diameter of marker, specified as a positive scalar. This value specifies the approximate diameter of the point marker. Units are in points. A marker size larger than six can reduce rendering performance.

Parent — Axes on which to display visualization
Axes object

Axes on which to display the visualization, specified as an Axes object. To create an Axes object, use the axes function. To display the visualization in a new figure, leave 'Parent' unspecified.

Output Arguments

ax — Plot axes
Axes object

Plot axes, returned as an axes graphics object.

Version History
Introduced in R2021a

See Also

Objects
pcmapsegmatch

Functions
pcshow | pcshowMatchedFeatures

updateMap

Update centroid and point cloud segment locations in map

Syntax

```
sMapOut = updateMap(sMapIn,tforms)
```

Description

`sMapOut = updateMap(sMapIn,tforms)` updates the centroid and point cloud segment locations by applying the specified transformation `tforms`.

Examples

Apply Translation and Rotation To Entire Map

Load a map of segments and features from a MAT file into the workspace.

```
data = load("segmatchMapFullParkingLot.mat");
sMap = data.segmatchMapFullParkingLot;
```

Visualize the map.

```
figure
show(sMap)
```

Change the viewing angle to top-view.

```
view(2)
title("Initial Map")
```
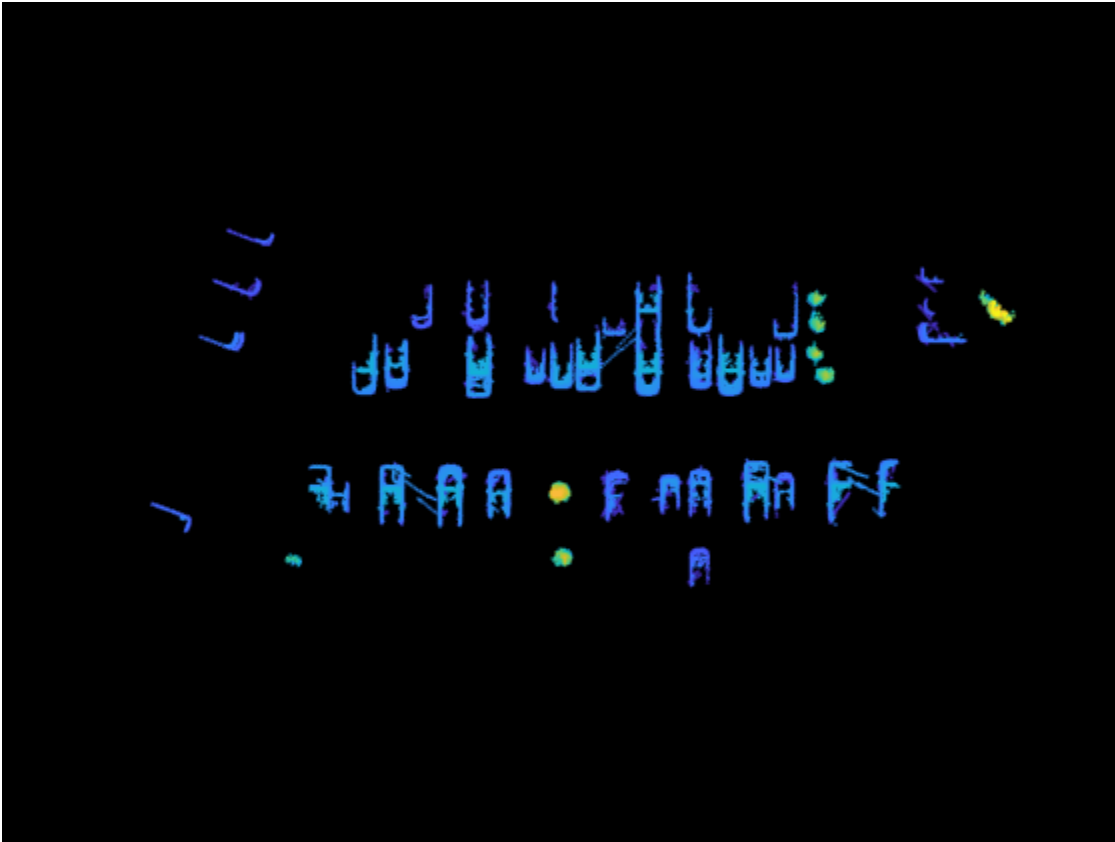
Define the transformation.

```
eulerAngles = [0 0 45]; % degrees
trans = [100 200 0];
tform = rigidtform3d(eulerAngles,trans);
numViews = numel(sMap.ViewIds);
tforms = repmat(tform,numViews,1);
```

Update the segments and features of each view with the defined transformation.

```
sMap = updateMap(sMap,tforms);
```

Visualize the transformed map.

```
figure
show(sMap)
```

Change the viewing angle to top-view.

```
view(2)
title("Transformed Map")
```
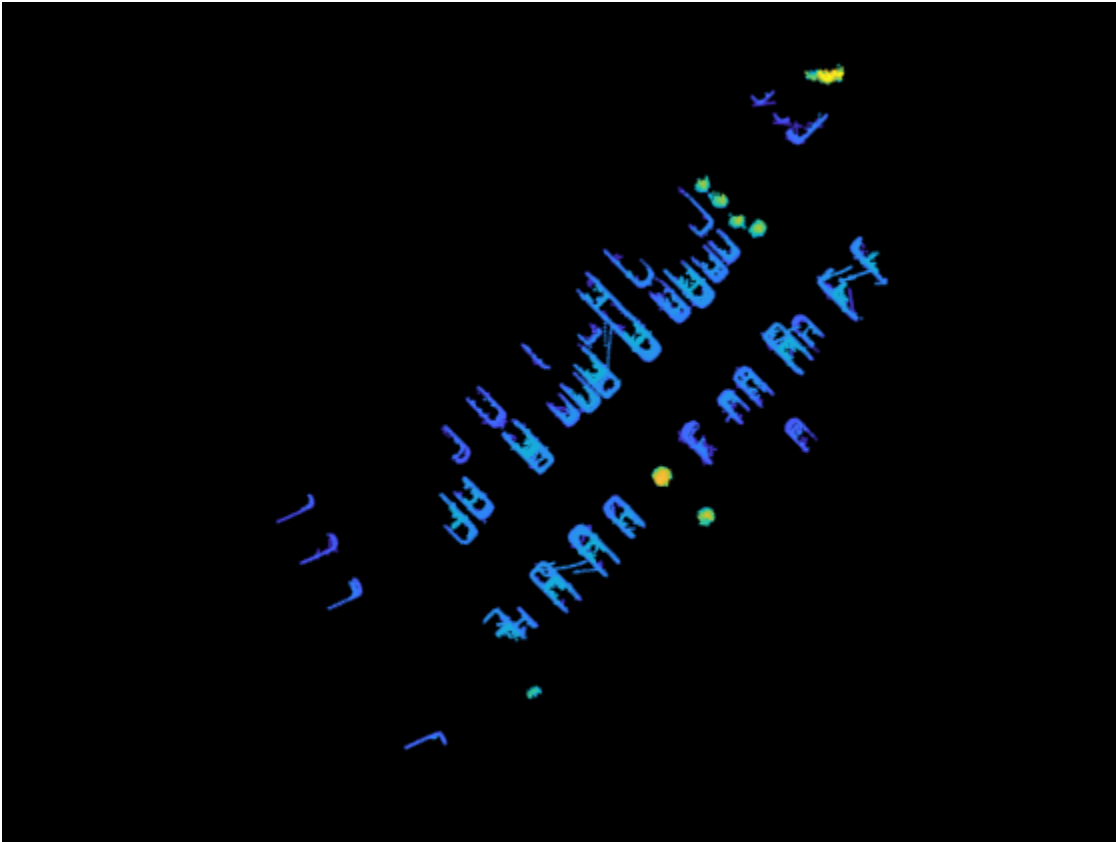
Input Arguments

sMapIn — Original map of segments and features
pcmapsegmatch object

Original map of segments and features, specified as a `pcmapsegmatch` object.

tforms — Transformations
M-element vector of `rigidtform3d` objects

Transformations, specified as an *M*-element vector of `rigidtform3d` objects. *M* is the number of views in the map.

Output Arguments

sMapOut — Updated map of segments and features
pcmapsegmatch object

Updated map of segments and features, returned as a `pcmapsegmatch` object. After the function updates the locations, it removes possible duplicates in the map based on the `CentroidDistance` property of the map.

The function resets the selected submap, specified by the `SelectedSubmap` property of the `pcmapsegmatch` object, to the extent of the map based on the centroid locations.

Tips

- To improve performance, remove all segments from the map using the `deleteSegments` function.

Version History
Introduced in R2021a

Supports `rigidtform3d` objects

You can now specify `tform` as a `rigidtform3d` object, which uses the premultiply convention. Although you can still specify `tform` as a `rigid3d` object, this object is not recommended because it uses the postmultiply convention. For more information, see "Migrate Geometric Transformations to Premultiply Convention".

See Also

Functions
`findPose`

Objects
`pcmapsegmatch` | `rigidtform3d`

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

There are two ways to create a `cuboidModel` object.

- Create a cuboid model by specifying the cuboid parameters in the `cuboidModel` function.
- Fit a cuboid model over a point cloud using the `pcfitcuboid` function.

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.

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 `Parameters` 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 `Parameters` 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 a 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 the internal properties of the `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 cuboid 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
```

Version History
Introduced in R2020b

Extended Capabilities

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

See Also

Functions
pcfitcuboid | getCornerPoints | findPointsInsideCuboid | plot

Objects
pointCloud | planeModel | cylinderModel | sphereModel

findPointsInsideCuboid

Find points enclosed by cuboid model

Syntax

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

Description

`Indices = findPointsInsideCuboid(model,ptCloudIn)` returns the linear indices of 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])
```
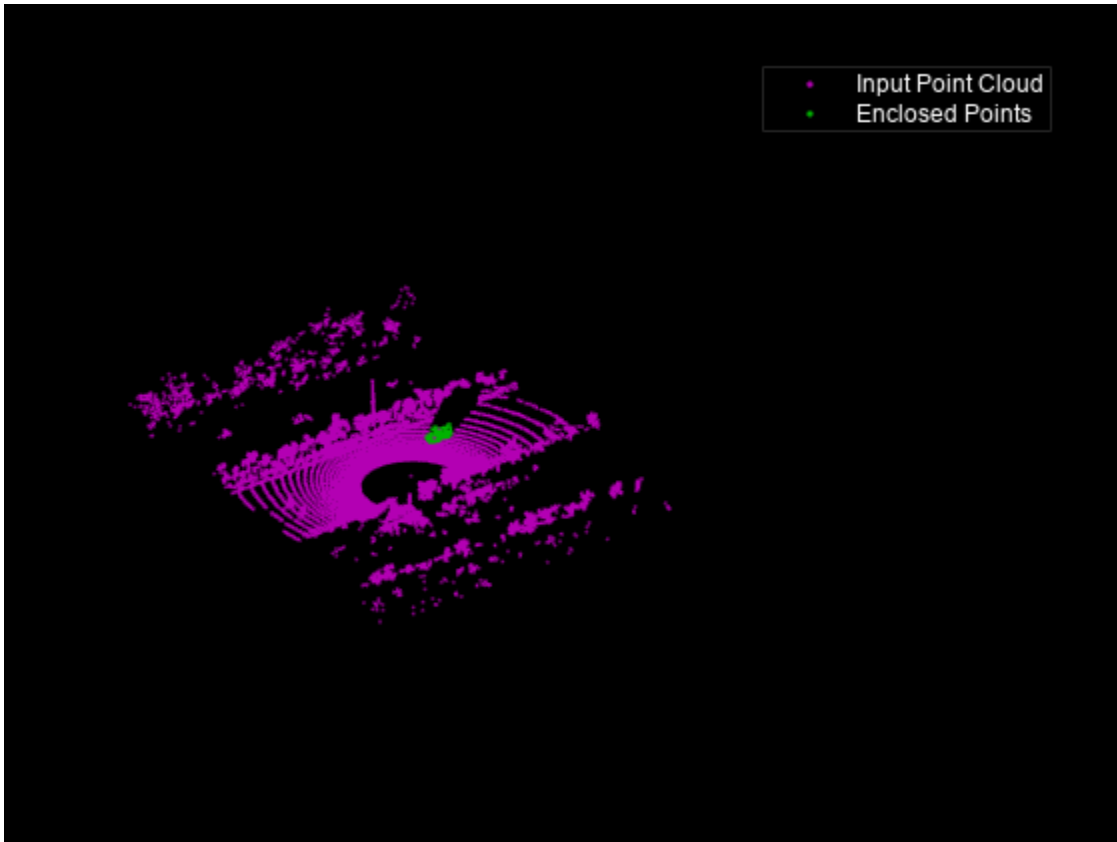
Input Arguments

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 an *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`.

Version History
Introduced in R2020b

Extended Capabilities

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

See Also

Functions
getCornerPoints | plot | pcfitcuboid

Objects
cuboidModel

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.

Version History
Introduced in R2020b

Extended Capabilities

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

See Also

Functions
findPointsInsideCuboid | plot | pcfitcuboid

Objects
cuboidModel

plot

Plot cuboid model

Syntax

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

Description

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

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

H = plot(___) additionally returns the cuboid model plot (figure) as a patch object.

Examples

Detect Cuboid in Point Cloud

Detect a 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)
```

Detect a Cuboid in a Point Cloud

Display the internal properties of the `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 object

Output axes, specified as an `Axes` object, on which to display the cuboid model. For a list of properties, see Axes Properties.

Output Arguments

H — Patch object
patch object

Patch object, returned as a `patch` object.

Version History
Introduced in R2020b

See Also

Functions
getCornerPoints | findPointsInsideCuboid | pcfitcuboid

Objects
cuboidModel

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** > **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 region of interest (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 ROI labels and scene labels, and sets the `LabelData` property.

Properties

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

Source of ground truth lidar data, specified as a `PointCloudSequenceSource`, `VelodyneLidarSource`, `LasFileSequenceSource`, `CustomPointCloudSource`, 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`
`VelodyneLidarSource`	Velodyne® packet capture (PCAP) file	`vision.labeler.loading.VelodyneLidarSource`
`LasFileSequenceSource`	LAS or LAZ file sequence folder	`lidar.labeler.loading.LasFileSequenceSource`
`CustomPointCloudSource`	Point cloud data from custom sources	`lidar.labeler.loading.CustomPointCloudSource`
`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'`.
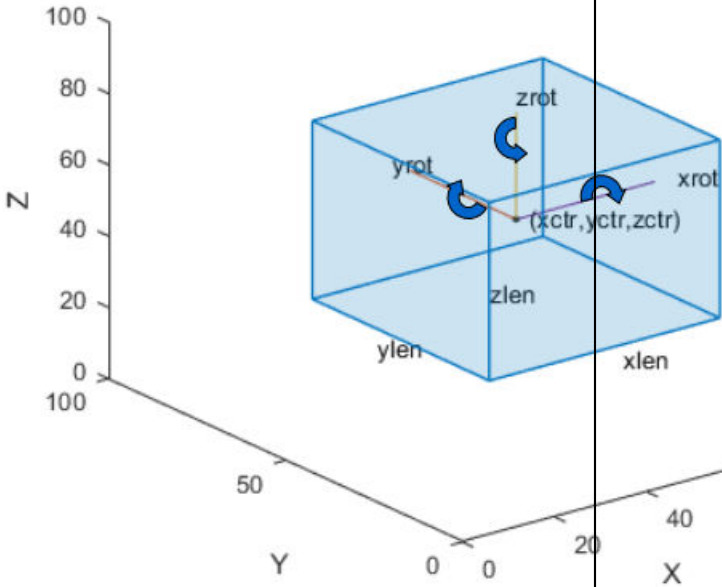
Column	Description	Required or Optional
Description	Strings or character vectors that describe each label definition.	Optional If you create the label definitions table at the MATLAB command line, you do not need to include a `Description` column. If you export label definitions from the **Lidar Labeler** 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.

Column	Description			Required or Optional
Hierarchy	Structures containing attribute information for each label definition.			Optional When you define sublabels or attributes in the **Lidar Labeler** app or the `labelDefinitionCreatorMultisignal` object, the generated label definitions table includes this column.
	Field		**Description**	
	`AttributeName1,...,AttributeNameN`		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.	
	`Type`		Type of parent label for the attributes, specified as a string or character vector.	
	`Description`		Description of parent label for the attributes, specified as a string or character vector.	
	If a label definition does not contain attributes, then the table entry for that label definition is empty.			

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.

Label Type	Storage Format for Labels at Each Timestamp
labelType.Cuboid	M-by-9 numeric matrix with rows of the form `[xctr, yctr, zctr, xlen, ylen, zlen, xrot, yrot, zrot]`, where: • M is the number of labels in the frame. • `xctr`, `yctr`, and `zctr` specify the center of the cuboid. • `xlen`, `ylen`, and `zlen` specify the length of the cuboid along the *x*-axis, *y*-axis, and *z*-axis, respectively, before rotation has been applied. • `xrot`, `yrot`, and `zrot` specify the rotation angles for the cuboid along the *x*-axis, *y*-axis, and *z*-axis, respectively. These angles are clockwise-positive when looking in the forward direction of their corresponding axes. The figure shows how these values determine the position of a cuboid.
labelType.Scene	Logical vector, where `true` indicates the presence of the label at that timestamp.

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.
`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: [1x1 vision.labeler.loading.VelodyneLidarSource]
    LabelDefinitions: [1x5 table]
           LabelData: [40x1 timetable]
```

Version History
Introduced in R2020b

See Also

Objects
labelDefinitionCreatorLidar | labelType | attributeType

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.

Version History
Introduced in R2020b

See Also
`groundTruthLidar`

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=4×5 table
         Name            Type          LabelColor            Group       Description
    _____      _____     _____    _____    _____

    {'car'       }      Cuboid     {[0.5862 0.8276 0.3103]}  {'vehicle'}   {0x0 char}
    {'bike'      }      Cuboid     {[    0.5172 0.5172 1]}   {'vehicle'}   {0x0 char}
    {'pole'      }      Cuboid     {[0.6207 0.3103 0.2759]}  {'None'   }   {0x0 char}
    {'vegetation'}      Cuboid     {[         0 1 0.7586]}   {'None'   }   {0x0 char}
```

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

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

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

```
gtLidarLabel.LabelDefinitions
```

```
ans=1×5 table
     Name        Type           LabelColor            Group        Description
    _____     _____     _____     _____     _____

    {'car'}     Cuboid     {[0.5862 0.8276 0.3103]}   {'vehicle'}    {0x0 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: [1x1 vision.labeler.loading.PointCloudSequenceSource]
    LabelDefinitions: [4x5 table]
           LabelData: [3x4 timetable]
```

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

```
gtLidarLabel.LabelDefinitions
```

```
ans=4×5 table
        Name           Type           LabelColor             Group        Description
    _____     _____     _____     _____     _____

    {'car'       }     Cuboid     {[0.5862 0.8276 0.3103]}   {'vehicle'}    {0x0 char}
    {'bike'      }     Cuboid     {[     0.5172 0.5172 1]}   {'vehicle'}    {0x0 char}
    {'pole'      }     Cuboid     {[0.6207 0.3103 0.2759]}   {'None'   }    {0x0 char}
    {'vegetation'}     Cuboid     {[         0 1 0.7586]}    {'None'   }    {0x0 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.

Version History
Introduced in R2020b

See Also

Objects
`groundTruthLidar`

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

selectLabelsByGroup

Select ground truth data by label group name

Syntax

```
gtLabel = selectLabelsByGroup(gTruth,labelGroups)
```

Description

`gtLabel = selectLabelsByGroup(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=4×5 table
        Name            Type            LabelColor            Group        Description
    _____    _____    _____    _____    _____

    {'car'       }    Cuboid    {[0.5862 0.8276 0.3103]}    {'vehicle'}    {0x0 char}
    {'bike'      }    Cuboid    {[    0.5172 0.5172 1]}    {'vehicle'}    {0x0 char}
    {'pole'      }    Cuboid    {[0.6207 0.3103 0.2759]}    {'None'   }    {0x0 char}
    {'vegetation'}    Cuboid    {[        0 1 0.7586]}    {'None'   }    {0x0 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: [1x1 vision.labeler.loading.PointCloudSequenceSource]
    LabelDefinitions: [2x5 table]
```

```
        LabelData: [3x2 timetable]
```

View the labels returned by the function.

`gtLidarLabel.LabelDefinitions`

ans=2×5 table

Name	Type	LabelColor	Group	Description
{'car' }	Cuboid	{[0.5862 0.8276 0.3103]}	{'vehicle'}	{0x0 char}
{'bike'}	Cuboid	{[0.5172 0.5172 1]}	{'vehicle'}	{0x0 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.

Version History
Introduced in R2020b

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=4×5 table
         Name            Type          LabelColor             Group        Description
    _____      _____    _____    _____    _____

    {'car'       }      Cuboid    {[0.5862 0.8276 0.3103]}    {'vehicle'}    {0x0 char}
    {'bike'      }      Cuboid    {[    0.5172 0.5172 1]}     {'vehicle'}    {0x0 char}
    {'pole'      }      Cuboid    {[0.6207 0.3103 0.2759]}    {'None'   }    {0x0 char}
    {'vegetation'}      Cuboid    {[        0 1 0.7586]}      {'None'   }    {0x0 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: [1x1 vision.labeler.loading.PointCloudSequenceSource]
    LabelDefinitions: [1x5 table]
           LabelData: [3x1 timetable]
```

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

`gtLidarLabel.LabelDefinitions`

```
ans=1×5 table
    Name       Type          LabelColor              Group        Description
   _____    _____    _____    _____    _____

   {'car'}    Cuboid    {[0.5862 0.8276 0.3103]}    {'vehicle'}    {0x0 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.

Version History
Introduced in R2020b

See Also

Objects
groundTruthLidar

Functions
selectLabels | selectLabelsByGroup | selectLabelsByType

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=4×5 table
        Name            Type            LabelColor            Group        Description
    _____    _____    _____    _____    _____

    {'car'       }     Cuboid     {[0.5862 0.8276 0.3103]}    {'vehicle'}    {0x0 char}
    {'bike'      }     Cuboid     {[     0.5172 0.5172 1]}    {'vehicle'}    {0x0 char}
    {'pole'      }     Cuboid     {[0.6207 0.3103 0.2759]}    {'None'   }    {0x0 char}
    {'vegetation'}     Cuboid     {[         0 1 0.7586]}    {'None'   }    {0x0 char}
```

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

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

```
gtLidarLabel =
  groundTruthLidar with properties:

          DataSource: [1x1 vision.labeler.loading.PointCloudSequenceSource]
    LabelDefinitions: [4x5 table]
```

```
      LabelData: [3x4 timetable]
```

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

`lidarLabelerGTruth.LabelDefinitions`

```
ans=4×5 table
        Name              Type            LabelColor              Group        Description
    _____      _____      _____      _____      _____

    {'car'       }      Cuboid      {[0.5862 0.8276 0.3103]}   {'vehicle'}     {0x0 char}
    {'bike'      }      Cuboid      {[     0.5172 0.5172 1]}   {'vehicle'}     {0x0 char}
    {'pole'      }      Cuboid      {[0.6207 0.3103 0.2759]}   {'None'   }     {0x0 char}
    {'vegetation'}      Cuboid      {[          0 1 0.7586]}   {'None'   }     {0x0 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.

Version History
Introduced in R2020b

See Also

Objects
groundTruthLidar

Functions
selectLabels | selectLabelsByGroup | selectLabelsByName

ibeoLidarReader

Ibeo data container (IDC) file reader

Description

An `ibeoLidarReader` object stores lidar data present in an Ibeo data container (IDC) file. The IDC file is captured by Ibeo lidar sensors. The object function, `readMessages`, uses the object properties to read Ibeo FUSION SYSTEM or ECU scan data and Ibeo point cloud plane data from IDC files. Ibeo Automotive Systems is a manufacturer of lidar sensor-based devices. The data captured by these devices is stored in 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 an IDC file. The `fileName` input sets the `FileName` property.

Properties

FileName — Name of IDC file
character vector | string scalar

This property is read-only.

Name of the 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

Version History
Introduced in R2020b

See Also

Functions
pcread | pcshow

Objects
lasFileReader | pointCloud | velodyneFileReader

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

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 one of these 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.

Version History
Introduced in R2020b

See Also

Functions
pcread | pcshow

Objects
ibeoLidarReader | lasFileReader | pointCloud | velodyneFileReader

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 2-202 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 2-202. 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]}	{0x0 char}	{'None'}	{' '}	{1x1 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)

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)

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

Version History
Introduced in R2020b

See Also

Apps
Lidar Labeler

Objects
groundTruthLidar

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 Lidar Label Definition Creator

Create an empty labelDefinitionCreatorLidar object.

```
ldc = labelDefinitionCreatorLidar

ldc =
labelDefinitionCreatorLidar
```

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

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

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

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

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

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

2-205

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

```
info(ldc,'Vehicle')

        Name: "Vehicle"
        Type: {[Cuboid]}
  LabelColor: {''}
       Group: "Transport"
  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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History
Introduced in R2020b

See Also

Objects
`labelDefinitionCreatorLidar`

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

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

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]}    {0x0 char}    {'None'}      {' '}        {1x1 struct}
```

Add Label and Attribute Using Lidar Label Definition Creator

Create an empty `labelDefinitionCreatorLidar` object.

```
ldc = labelDefinitionCreatorLidar
```

```
ldc =
labelDefinitionCreatorLidar
```

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

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

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

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

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 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 label `Vehicle` using the `info` object function.

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

```
          Name: "Vehicle"
          Type: {[Cuboid]}
    LabelColor: {''}
         Group: "Transport"
    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 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History
Introduced in R2020b

See Also

Objects
labelDefinitionCreatorLidar

Functions
addAttribute | editLabelDescription | removeLabel

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 Lidar Label Definition Creator

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
  _____    _____    _____    _____    _____

  {'Vehicle'}    {[Cuboid]}    {0x0 char}    {'None'}    {'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. 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.

Version History

Introduced in R2020b

See Also

Objects
`labelDefinitionCreatorLidar`

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

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 Lidar Label Definition Creator

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 RGB format - [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`.

Version History

Introduced in R2020b

See Also

Objects
labelDefinitionCreatorLidar

Functions
editLabelDescription

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 Lidar Label Definition Creator

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.

Version History
Introduced in R2020b

See Also

Objects
`labelDefinitionCreatorLidar`

Functions
`editLabelDescription` | `editLabelGroup`

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 Lidar Label Definition Creator

Create an empty labelDefinitionCreatorLidar object.

```
ldc = labelDefinitionCreatorLidar;
```

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

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

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.

Version History
Introduced in R2020b

See Also

Objects
`labelDefinitionCreatorLidar`

Functions
`editAttributeDescription`

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

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.

Version History
Introduced in R2020b

See Also

Objects
labelDefinitionCreatorLidar

Functions
editGroupName | editLabelDescription

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 Lidar Label Definition Creator

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 bu
```

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).

Version History
Introduced in R2020b

See Also

Objects
labelDefinitionCreatorLidar

Functions
addLabel | create

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 attributeNamefrom the label labelName in the labelDefinitionCreatorLidar object ldc.

Examples

Remove Attribute from Label in Lidar Label Definition Creator

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.

Version History
Introduced in R2020b

See Also

Objects
labelDefinitionCreatorLidar

Functions
addAttribute | addLabel | removeLabel

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 Lidar Label Definition Creator

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.

Version History

Introduced in R2020b

See Also

Objects
labelDefinitionCreatorLidar

Functions
addLabel | addAttribute | removeAttribute

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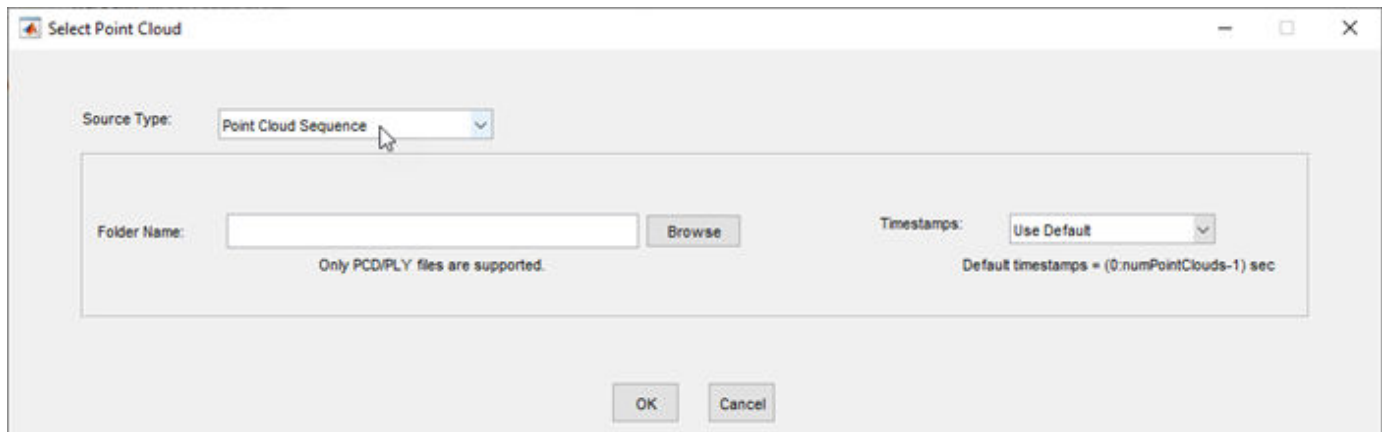
The class also provides an interface to read frames from loaded signals. The app renders these frames for labeling.

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
- `lidar.labeler.loading.CustomPointCloudSource` — Custom source 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.
	<table><tr><td>Abstract</td><td>true</td></tr></table>

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.
	<table><tr><td>Abstract</td><td>true</td></tr></table>
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.
	<table><tr><td>Abstract</td><td>true</td></tr></table>
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.
	<table><tr><td>Abstract</td><td>true</td></tr></table>
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.
	<table><tr><td>Static</td><td>true</td></tr></table>

Version History
Introduced in R2020b

See Also

Apps
Lidar Labeler

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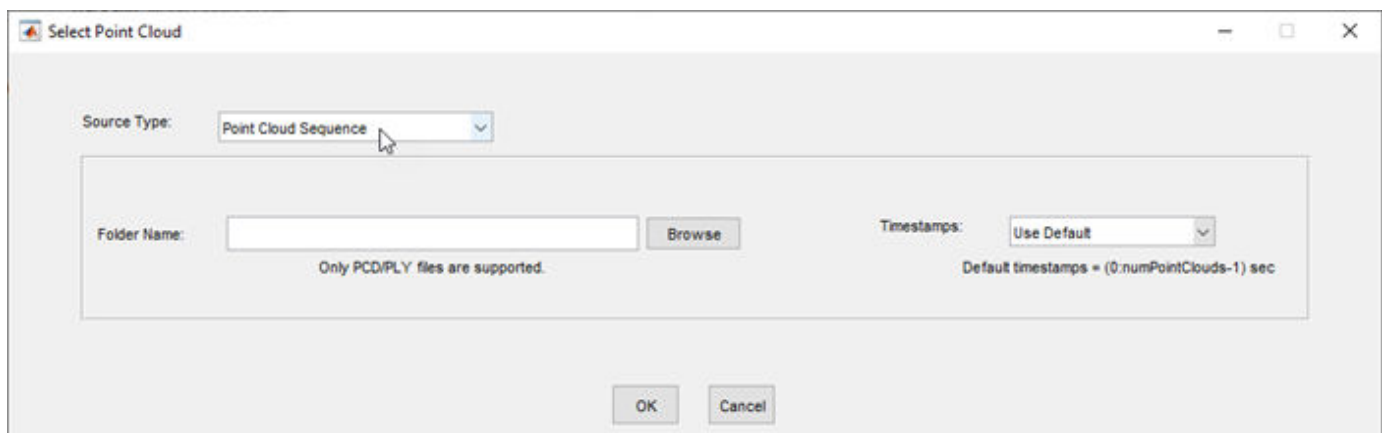
To access this dialog box, in the app, select **Import** > **Add Point Cloud**.

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)
```
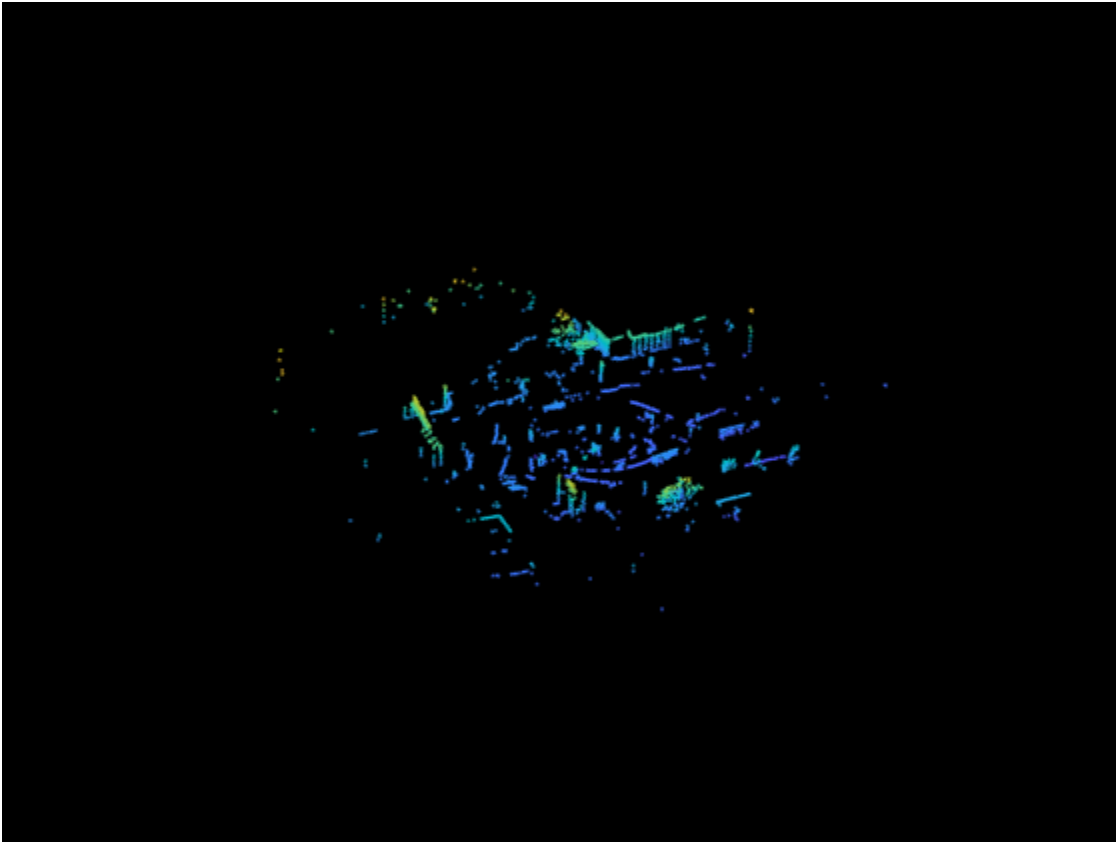
Version History
Introduced in R2020b

See Also

Apps
Lidar Labeler

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

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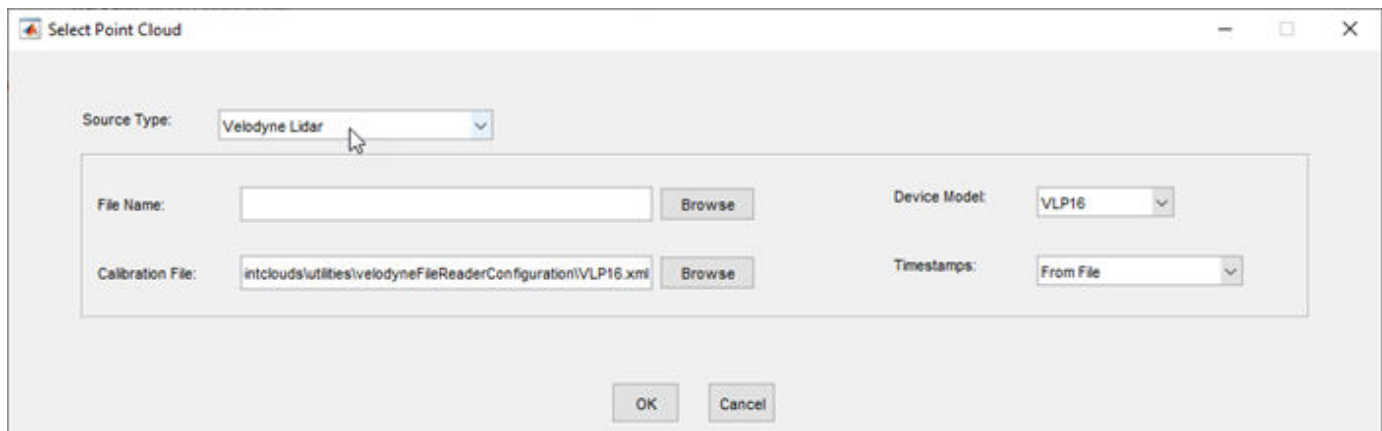
To access this dialog box, in the app, select **Import > Add Point Cloud**.

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. • `'VLP16'` (default) — VLP-16 device model • `'PuckLITE'` — Puck LITE device model • `'PuckHiRes'` — Puck Hi-Res device model • `'VLP32C'` — VLP-32C device model • `'HDL32E'` — HDL-32E device model • `'HDL64E'` — HDL-64E device model 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.

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.
	<table><tr><td>Static</td><td>true</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)
```
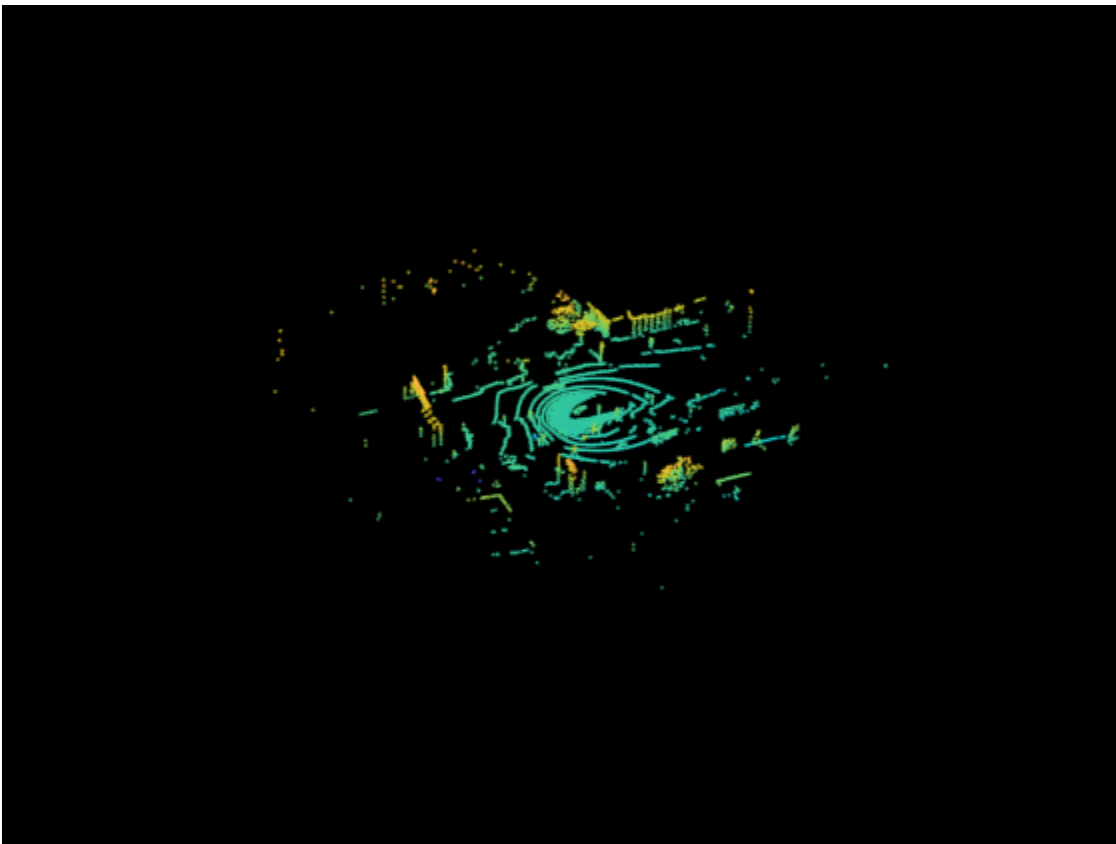
Version History
Introduced in R2020b

See Also

Apps
Lidar Labeler

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

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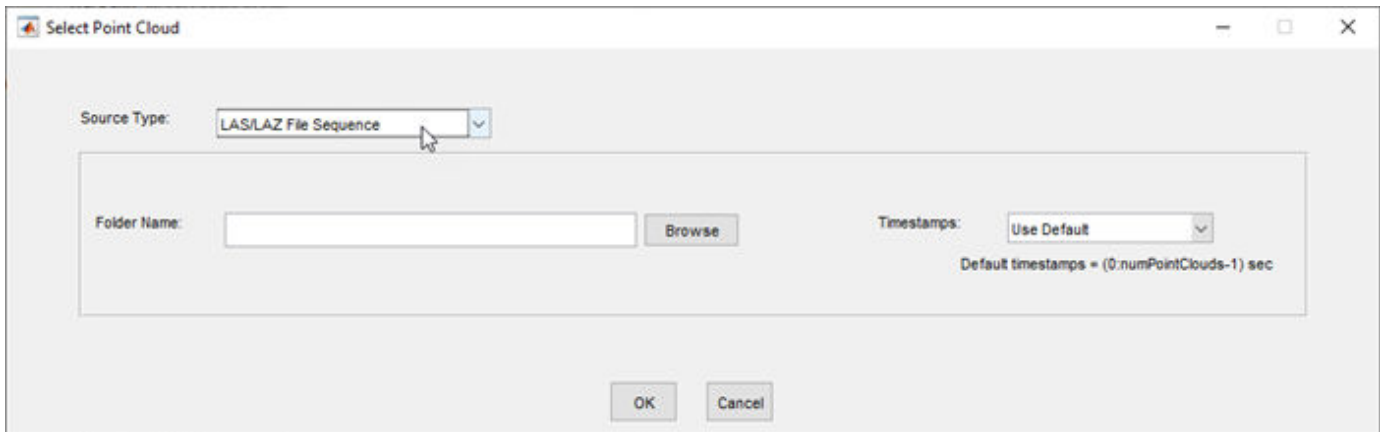
To access this dialog box, in the app, select **Import > Add Point Cloud**.

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)
```
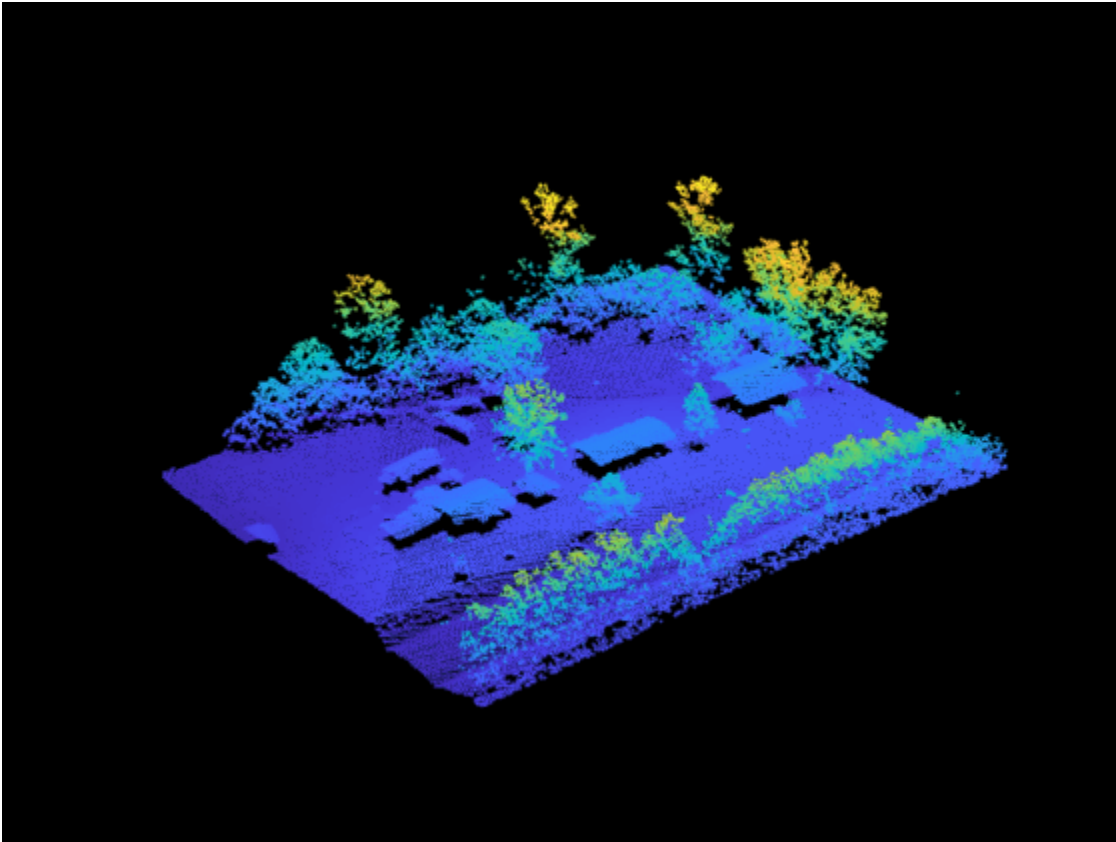
Version History
Introduced in R2020b

See Also

Apps
Lidar Labeler

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

lidar.labeler.loading.RosbagSource class

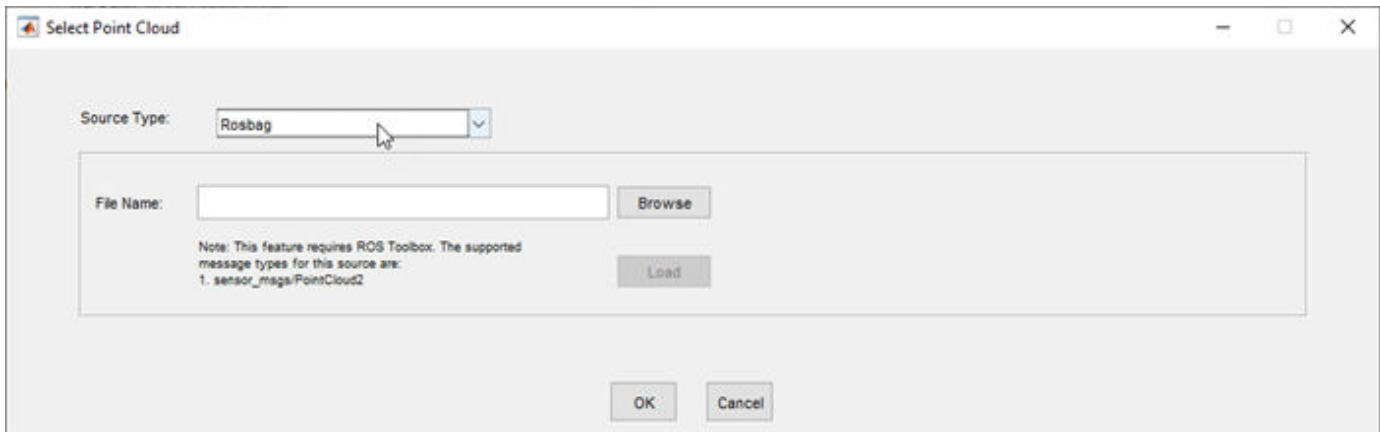**Package:** `lidar.labeler.loading lidar.labeler.loading lidar.labeler.loading lidar.labeler.loading lidar.labeler.loading lidar.labeler.loading`
Superclasses: `vision.labeler.loading.MultiSignalSource`

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 **Import > 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.

2-255

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

Version History
Introduced in R2020b

See Also

Apps
Lidar Labeler

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

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 name of the lidar data to load into the app.

```
sourceName = fullfile("lidarSequence");
```

Connect the video display to the app and display synchronized data.

```
lidarLabeler(sourceName,"SyncImageViewerTargetHandle",@helperSyncImageDisplay);
```
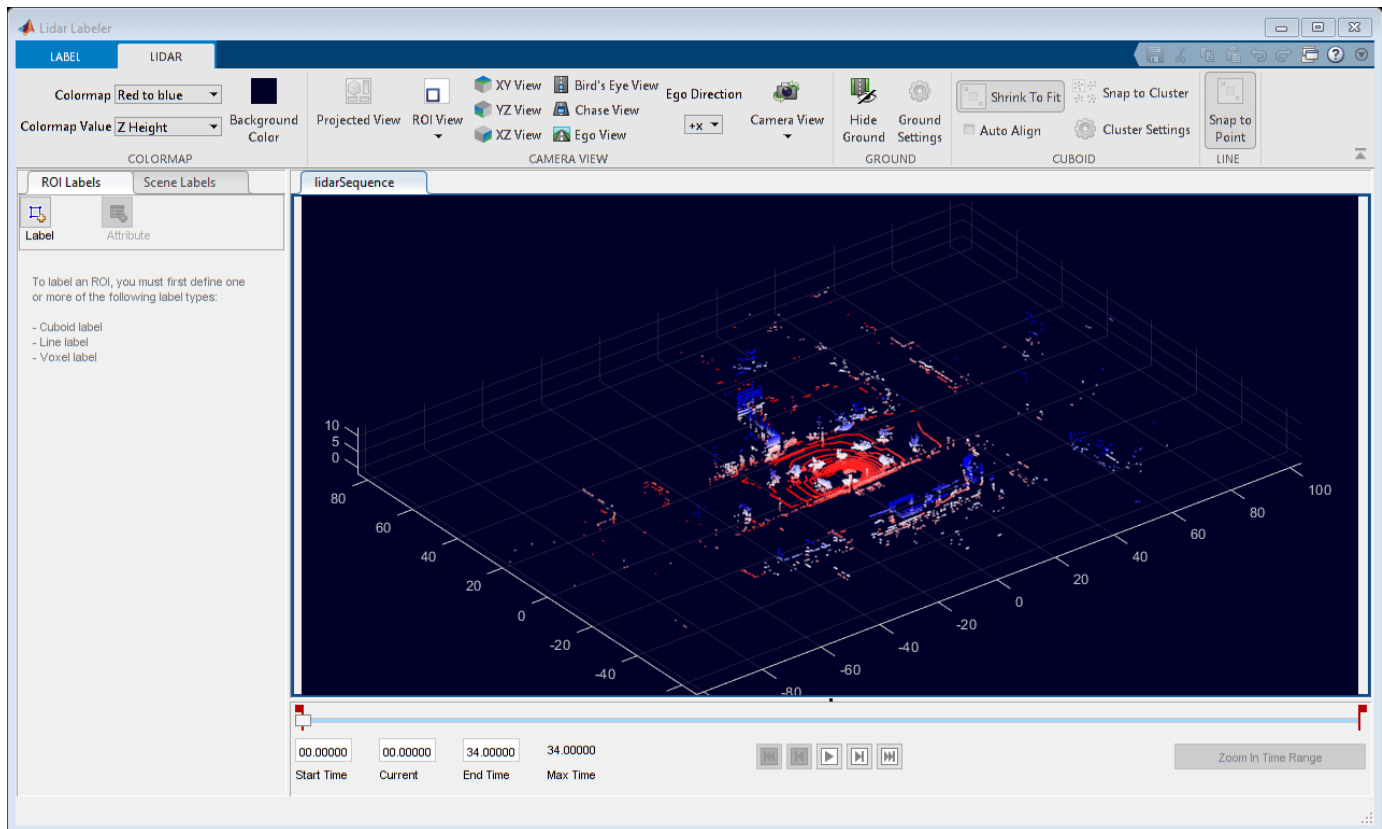
Tips

- 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

Version History
Introduced in R2020b

See Also

Apps
Lidar Labeler

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.

Version History
Introduced in R2020b

See Also
`lidar.syncImageViewer.SyncImageViewer` | Lidar Labeler

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.

Version History
Introduced in R2020b

See Also
`lidar.syncImageViewer.SyncImageViewer` | Lidar Labeler

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.

Version History
Introduced in R2020b

See Also
`lidar.syncImageViewer.SyncImageViewer` | Lidar Labeler

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.

Version History
Introduced in R2020b

See Also
`lidar.syncImageViewer.SyncImageViewer` | Lidar Labeler

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.

Version History
Introduced in R2020b

See Also
`lidar.syncImageViewer.SyncImageViewer` | Lidar Labeler

lasFileReader

LAS or LAZ file reader

Description

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. The `lasFileReader` object supports up to the LAS 1.4 specification.

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.

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.

Creation

Syntax

lasReader = lasFileReader(fileName)

Description

lasReader = lasFileReader(fileName) creates a `lasFileReader` object with properties set by reading the metadata present in the LAS or LAZ file `fileName`. 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 real-valued row vector

This property is read-only.

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

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

This property is read-only.

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

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

This property is read-only.

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

SystemIdentifier — Name of hardware sensor system identifier
string scalar

This property is read-only.

Name of the hardware sensor system identifier that generated the LAS files, specified as a string scalar.

GeneratingSoftware — Name of generating software
string scalar

This property is read-only.

Name of the generating software, specified as a string scalar. This property specifies the generating software package used when the LAS file was created.

FileCreationDate — Date of file creation
datetime object

This property is read-only.

Date of file creation, specified as a `datetime` object.

FileSourceID — LAS file source identifier
nonnegative integer

This property is read-only.

LAS file source identifier, specified as a nonnegative integer. Values are in the range 0 to 65535. This defines the flight line number if this file was created from an original flight line. A value 0 specifies that no ID has been assigned. Use the `ProjectID` and `FileSourceID` properties to uniquely identify each point in a LAS file.

ProjectID — Project ID
string scalar

This property is read-only.

Project ID, specified as a string scalar. This value is a globally unique identifier (GUID). Use the `ProjectID` and `FileSourceID` properties to uniquely identify each point in a LAS file.

PointDataFormat — Point data record format ID
nonnegative integer

This property is read-only.

Point data record format ID, specified as a nonnegative integer. Values are in the range 0 to 10. For more information, see "Point Data Record Format" on page 2-272.

ClassificationInfo — Classification information
table

This property is read-only.

Classification information, specified as a table. Each row of the table contains this information describing a point class:

- `Classification Value` — Unique classification ID number for the class, specified as a positive integer.
- `Class Name` — Label associated with the class, specified as a string scalar.
- `Number of Points by Class` — Number of points in the class, specified as a positive integer.

LaserReturnInfo — Laser return information
table

This property is read-only.

Laser return information, specified as a table. Each row of the table contains this information describing a laser return:

- `Laser Return Number` — Laser return number, specified as a positive integer.
- `Number of Points by Return` — Number of points per laser return, specified as a positive integer.

VariableLengthRecords — Variable length record information
table

This property is read-only.

Variable length record (VLR) or extended VLR information, specified as a table. Each row of the table contains this information describing a record:

- `Record ID` — Record identification number, specified as a positive integer.
- `User ID` — User identification associated with record ID, specified as a string scalar.
- `Description` — Description of record, specified as a string scalar.

Object Functions

readPointCloud	Read point cloud data from LAS or LAZ file
readCRS	Read coordinate reference system data from LAS or LAZ file
readVLR	Read variable length record from LAS or LAZ file
hasCRSData	Check if LAS or LAZ file has CRS data
hasGPSData	Check if LAS or LAZ file has GPS data
hasWaveformData	Check if LAS or LAZ file has waveform data
hasNearIRData	Check if LAS or LAZ file has near IR data

Examples

Read Point Cloud Data from LAZ File

This example shows how to read and visualize point cloud data from a LAS / 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.

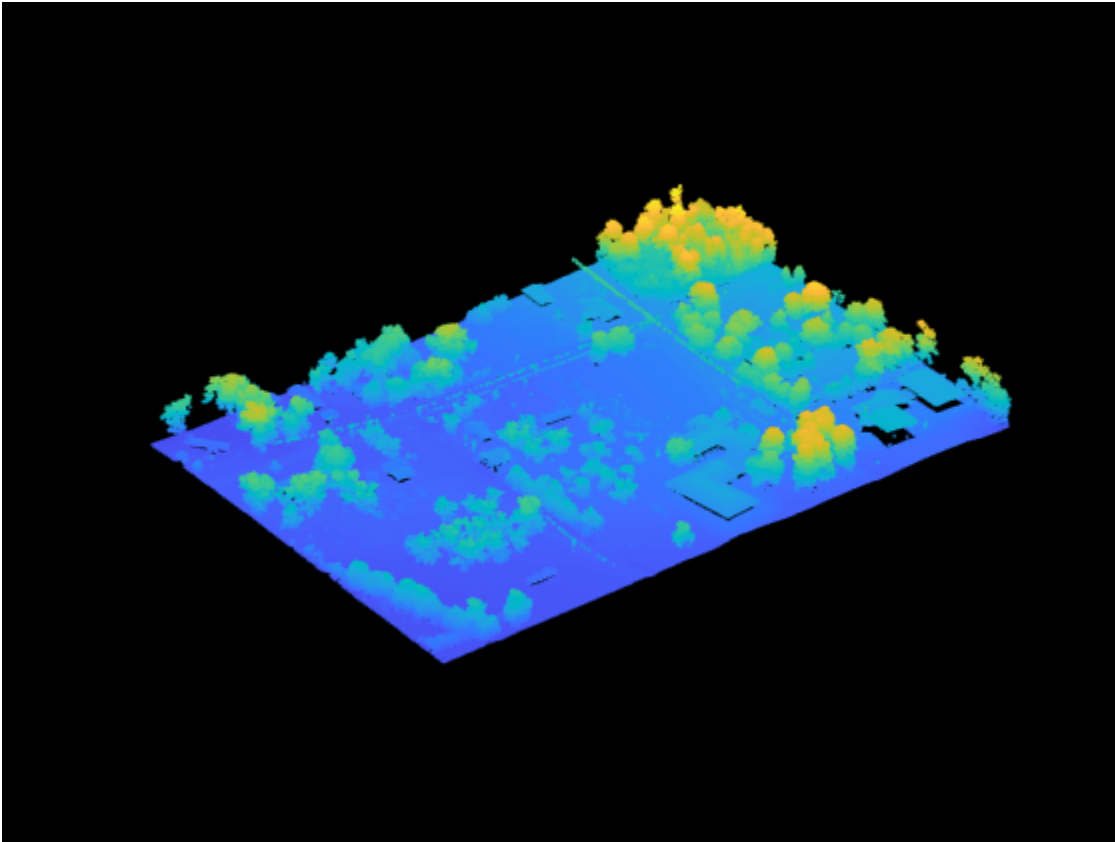Create a `lasFileReader` object to access the LAZ file data.

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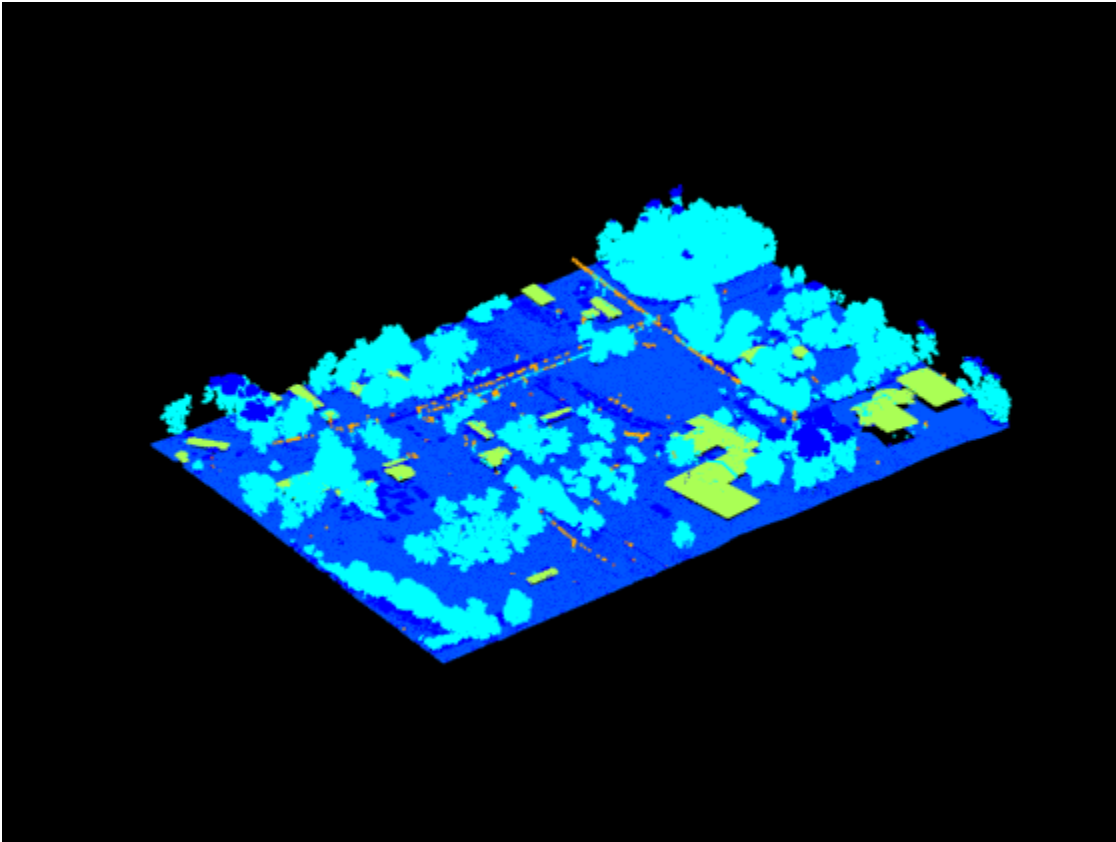
```
[ptCloud,pointAttributes] = readPointCloud(lasReader,"Attributes","Classification");
```

Color the points based on their classification attributes. Reshape the label image into the shape of the point cloud.

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

Visualize the color-coded point cloud.

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

2-271

More About

Point Data Record Format

A LAS file contains point cloud data as a collection of point data records, as well as a public header block and optional metadata information about the records. The public header block indicates the point numbers and point data bounds of the LAS file, as well as the point data record format.

As of the LAS 1.4 specification, there are 11 point data record formats, ranging from Point Data Record Format 0 to Point Data Record Format 10, divided into two major groups.

- Point Data Record Format 0 to Point Data Record Format 5 — These formats contain 20 core bytes, with formats 1–5 adding optional information about GPS time, color channels, or wave packets.

- Point Data Record Format 6 to Point Data Record Format 10 — These formats contain 30 core bytes, with formats 7–10 adding optional information about color channels, near IR, or wave packets.

For more information, including which point data record formats are available for each version of the LAS file specification, see the ASPRS LASER (LAS) File Format Exchange Activities page.

Version History
Introduced in R2020b

Object has additional properties

The lasFileReader object includes these additional properties:

- SystemIdentifier
- GeneratingSoftware
- FileCreationDate
- FileSourceID
- ProjectID
- PointDataFormat
- ClassificationInfo
- LaserReturnInfo
- VariableLengthRecords

See Also

Functions
pcread | pcshow

Objects
lasFileWriter | pointCloud | lidarPointAttributes | ibeoLidarReader |
velodyneFileReader

readPointCloud

Read point cloud data from 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 the specified point attributes `ptAtt` from a LAS or LAZ file. In addition to the point cloud, the function returns 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

This example shows how to read and visualize point cloud data from a LAS / 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.

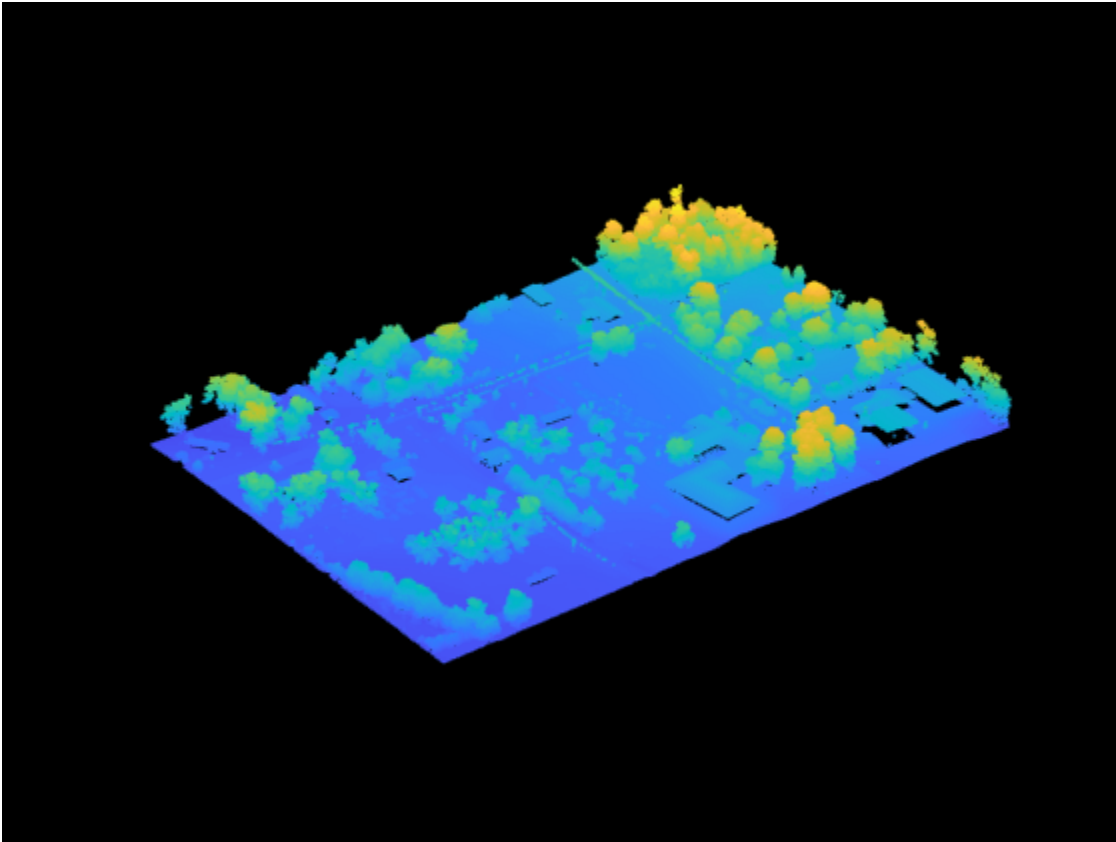Create a `lasFileReader` object to access the LAZ file data.

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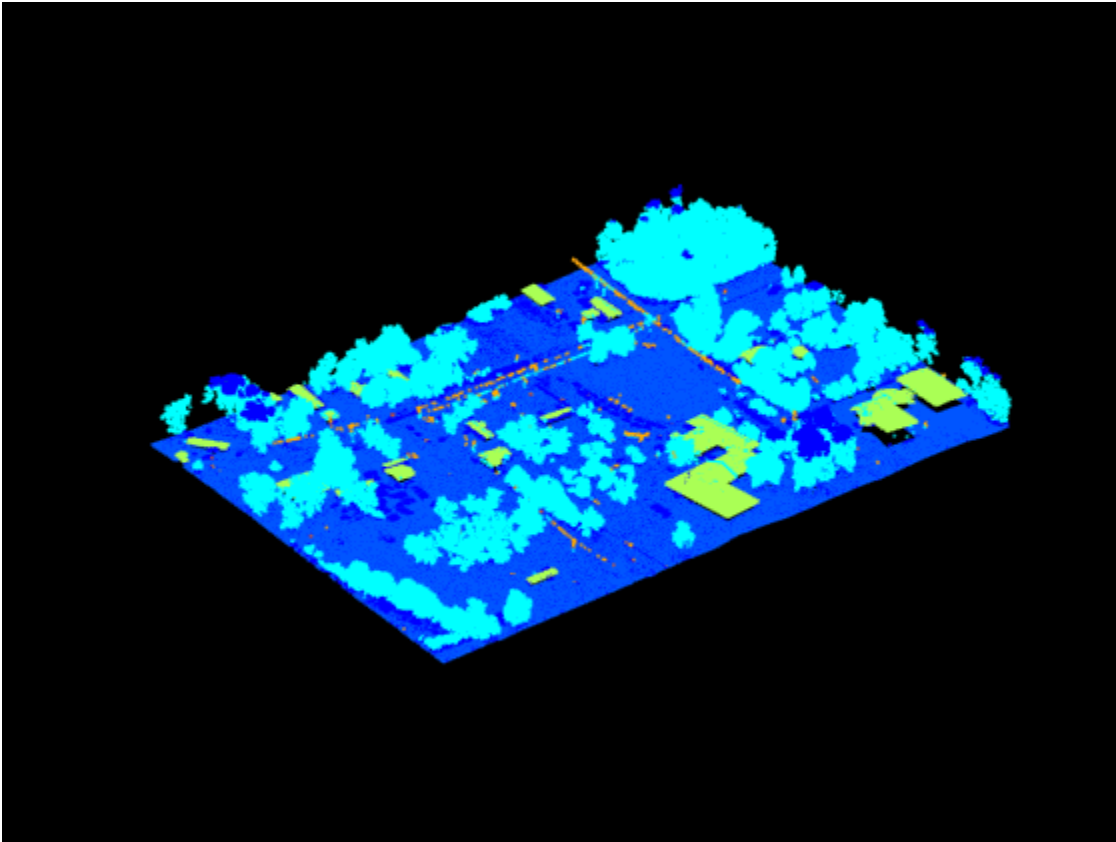
```
[ptCloud,pointAttributes] = readPointCloud(lasReader,"Attributes","Classification");
```

Color the points based on their classification attributes. Reshape the label image into the shape of the point cloud.

```
labels = label2rgb(pointAttributes.Classification);
colorData = reshape(labels,[],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 | vector of strings

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

- "Classification"
- "GPSTimeStamp"
- "LaserReturn"
- "NumReturns"
- "NearIR"
- "ScanAngle"
- "UserData"
- "PointSourceID"

- "ScannerChannel"
- "ScanDirectionFlag"
- "EdgeOfFlightLineFlag"
- "WaveformData"

The function returns the specified attributes of each point in a `lidarPointAttributes` object, `ptAttributes`. The unspecified attributes are returned empty.

Data Types: `char` | `string` | `cell`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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

Classification Number	Classification Type
5	High vegetation
6	Building
7	Low point (noise)
8	Reserved
9	Water
10	Rail
11	Road surface
12	Reserved
13	Wire guard (shield)
14	Wire conductor (phase)
15	Transmission tower
16	Wire-structure connector (insulator)
17	Bridge deck
18	High noise
19	Overhead structure
20	Ignored ground
21	Snow
22	Temporal exclusion
23- 63	Reserved
64 - 255	User-defined

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

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

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

Number of points segregated by their return numbers, specified as the comma-separated pair consisting of `'LaserReturn'` 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 attribute data
lidarPointAttributes object

Point attribute data, returned as a lidarPointAttributes object. The object contains data for the specified attributes ptAtt of each point in the ptCloud output.

Version History
Introduced in R2020b

Additional point attributes

You can now specify these additional point attributes to the ptAtt argument:

- "UserData"
- "PointSourceID"
- "ScannerChannel"
- "ScanDirectionFlag"
- "EdgeOfFlightLine"
- "WaveformData"

LaserReturns renamed to LaserReturn
Behavior changed in R2022a

The LaserReturns is now called LaserReturn. To update your code, replace all instances of the LaserReturns argument with LaserReturn.

Point attributes returned as object
Behavior changed in R2022a

The readPointCloud function returns the ptAttributes argument as a lidarPointAttributes object instead of as a structure.

See Also

Functions
pcread | pcshow

Objects
pointCloud | ibeoLidarReader | lasFileReader | velodyneFileReader

readCRS

Read coordinate reference system data from LAS or LAZ file

Syntax

```
crs = readCRS(lasReader)
```

Description

`crs = readCRS(lasReader)` reads the coordinate reference system data from the LAS or LAZ file indicated by the input `lasFileReader` object `lasReader`.

This function requires the Mapping Toolbox™.

Examples

Read CRS Data from LAZ File

Create a `lasFileReader` object for a LAZ file. Then, use the `readCRS` function to read coordinate reference system (CRS) data from the LAZ file and generate a `projcrs` object.

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

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

Check if the LAZ file contains CRS data using the `hasCRSData` function. If it does, read the CRS data from the LAZ file using the `readCRS` function.

```
if hasCRSData(lasReader)
    crs = readCRS(lasReader);
    disp(crs);
else
    disp("No CRS data available.");
end

  projcrs with properties:

                    Name: "NAD83 / UTM zone 16N"
            GeographicCRS: [1x1 geocrs]
        ProjectionMethod: "Transverse Mercator"
               LengthUnit: "meter"
    ProjectionParameters: [1x1 map.crs.ProjectionParameters]
```

Input Arguments

lasReader — LAS or LAZ file reader
lasFileReader object

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

Output Arguments

`crs` — Coordinate reference system
geocrs object | projcrs object

Coordinate reference system (CRS), returned as one of these objects:

- `geocrs` — Returned by a file that contains geographic CRS data.
- `projcrs` — Returned by a file that contains projected CRS data.

Version History
Introduced in R2022a

See Also

Functions
pcread | pcshow | readVLR | hasCRSData

Objects
pointCloud | lidarPointAttributes | ibeoLidarReader | lasFileReader | velodyneFileReader | geocrs | projcrs

Topics
"Create, Process, and Export Digital Surface Model from Lidar Data" (Mapping Toolbox)

readVLR

Read variable length record from LAS or LAZ file

Syntax

```
vlr = readVLR(lasReader,recordID)
vlr = readVLR(lasReader,recordID,userID)
```

Description

`vlr = readVLR(lasReader,recordID)` reads the variable length record `vlr` from the specified record `recordID` of the LAS or LAZ file reader `lasReader`.

`vlr = readVLR(lasReader,recordID,userID)` specifies the user ID of the variable length record.

Examples

Read VLR Data from LAZ File

Create a `lasFileReader` object for a LAZ file. Then, use the `readVLR` function to read variable length record (VLR) data from the LAZ file.

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

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

Read VLR data from the LAZ file using the `readVLR` function.

```
vlr = readVLR(lasReader,34737);
```

Display the VLR data.

```
disp(vlr)

      RecordID: 34737
        UserID: 'LASF_Projection'
   Description: 'GeoTIFF GeoAsciiParamsTag'
   RawByteData: [78 65 68 56 51 32 47 32 85 84 77 32 122 111 110 101 32 ... ]
          Data: 'NAD83 / UTM zone 16N + VERT_CS|NAD83|NAVD88 height'
```

Input Arguments

lasReader — LAS or LAZ file reader
`lasFileReader` object

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

recordID — Record ID
positive integer

Record ID, specified as a positive integer.

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

userID — User ID for VLR data
character vector | string scalar

User ID for the VLR data, specified as a character vector or string scalar. This value identifies the user that created the VLR data.

Data Types: `char` | `string`

Output Arguments

vlr — Variable length record
structure | array of structures

Variable length record, returned as a structure or an array of structures. Each structure contains these fields:

- `RecordID` — Record ID of the VLR, returned as a positive integer.
- `UserID` — User ID that created the VLR, returned as a string scalar.
- `Description` — Text description of the VLR, returned as a string scalar.
- `RawByteData` — Raw bytes of data stored in the VLR, returned as an integer vector. The length of the vector is equal to the `Record Length After Header` value in the corresponding VLR header.
- `Data` — Parsed data for standard VLR records. The value of this field depends on the contents of the VLR corresponding to its record ID and user ID. This table lists some standard record ID and user ID combinations.

Record ID	User ID	Description
0	LASF_Spec	Data of the Classification Lookup record, returned as a character vector or string scalar.
3	LASF_Spec	ASCII data of the Text Area Description record, returned as a character vector or string scalar.

Record ID	User ID	Description
100-354	LASF_Spec	Data of the Waveform Packet Descriptor record, returned as a structure containing these fields: • `BitsPerSample` — Number of bits for each sample in the range 2 to 32 bits. • `CompressionType` — Compression algorithm for waveform packets. Value 0 represents no compression. Reserved for future use. • `NumberOfSamples` — Number of samples in decompressed waveform packet. • `TemporalSpacing` — Temporal sample spacing in picoseconds. • `DigitizerGain` — Digitizer gain to use to convert raw digitized value to an absolute digitizer voltage. • `DigitizerOffset` — Digitizer offset to use to convert raw digitized value to an absolute digitizer voltage.
2111	LASF_Projection	ASCII data of the OGS Math Transform WKT record, returned as a character vector or string scalar.
2112	LASF_Projection	ASCII data of the OGS Coordinate System WKT record, returned as a character vector or string scalar.

Record ID	User ID	Description

34735	LASF_Projection	GeoTiff key values of the GeoKeyDirectoryTag record, returned as a structure containing these fields:
		
		• `KeyDirectoryVersion` — Key directory version number, returned as 1.
		• `KeyRevision` — Key revision number, returned as 1.
		• `MinorRevision` — Minor revision number, returned as 0.
		• `NumberOfKeys` — Number of keys, returned as a scalar.
		• `KeyEntries` — Structure for each key containing these fields:
		
		  • `KeyID` — Key ID for each GeoTIFF data.
		  • `TIFFTagLocation` — Location of the data for the specified key ID.
		  • `Count` — Number of characters in GeoAsciiParamsTag string value. Otherwise it returns 1.
		  • `ValueOffset` — Value depends on the `TIFFTagLocation` field.

34736	LASF_Projection	Data of the GeoDoubleParamsTag record, returned as a numeric vector of type `double`.
34737	LASF_Projection	ASCII data of the GeoAsciiParamsTag record, returned as a character vector or string scalar.

Note When you specify a valid combination of record ID and user ID, the `Data` field is nonempty. Otherwise, the function interprets the binary contents of the VLR and returns only `RawByteData`, leaving the `Data` field empty.

For more information on the VLR header, or the various types of records, see the ASPRS LASER (LAS) File Format Exchange Activities page.

Data Types: `struct`

Version History
Introduced in R2022a

Specify user ID to read VLR data

You can additionally specify the user ID along with the record ID to read VLR data from a LAS or LAZ file.

Returns both raw byte data and VLR data
Behavior change in future release

The output `vlr` structure contains both `RawByteData` and `Data` fields. Prior to R2022b, the function returned only one of these two fields.

See Also

Functions
pcread | pcshow | readCRS

Objects
pointCloud | lidarPointAttributes | ibeoLidarReader | lasFileReader | velodyneFileReader

hasCRSData

Check if LAS or LAZ file has CRS data

Syntax

```
flag = hasCRSData(lasReader)
```

Description

`flag = hasCRSData(lasReader)` returns a logical `1` (`true`) if the specified LAS or LAZ file `lasReader` contains a coordinate reference system (CRS) data. Otherwise, it returns a logical `0` (`false`).

Examples

Check for CRS Data in LAZ File

Create a `lasFileReader` object for a LAZ file. Then, use the `hasCRSData` function to check if the LAZ file contains CRS data to read.

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

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

Check for CRS data in the LAZ file by using the `hasCRSData` function.

```
flag = hasCRSData(lasReader);
disp(flag)

    1
```

Input Arguments

lasReader — LAS or LAZ file reader
`lasFileReader` object

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

Version History
Introduced in R2022a

See Also

Functions
readCRS | pcread | pcshow | hasGPSData | hasWaveformData | hasNearIRData

Objects
pointCloud | lidarPointAttributes | ibeoLidarReader | lasFileReader |
velodyneFileReader

hasGPSData

Check if LAS or LAZ file has GPS data

Syntax

```
flag = hasGPSData(lasReader)
```

Description

`flag = hasGPSData(lasReader)` returns a logical `1` (`true`) if the specified LAS or LAZ file `lasReader` contains a GPS data. Otherwise, it returns a logical `0` (`false`).

Examples

Check for GPS Data in LAZ File

Create a `lasFileReader` object for a LAZ file. Then, use the `hasGPSData` function to check if the LAZ file contains GPS data to read.

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

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

Check for GPS data in the LAZ file by using the `hasGPSData` function.

```
flag = hasGPSData(lasReader);
disp(flag)

    1
```

Input Arguments

lasReader — LAS or LAZ file reader
lasFileReader object

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

Version History
Introduced in R2022a

See Also

Functions
pcread | pcshow | hasWaveformData | hasNearIRData | hasCRSData

Objects
pointCloud | lidarPointAttributes | ibeoLidarReader | lasFileReader |
velodyneFileReader

hasNearIRData

Check if LAS or LAZ file has near IR data

Syntax

```
flag = hasNearIRData(lasReader)
```

Description

`flag = hasNearIRData(lasReader)` returns a logical `1` (`true`) if the specified LAS or LAZ file `lasReader` contains near IR data. Otherwise, it returns a logical `0` (`false`).

Examples

Check for Near IR Data in LAZ File

Create a `lasFileReader` object for a LAZ file. Then, use the `hasNearIRData` function to check if the LAZ file contains near IR data to read.

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

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

Check for near IR data in the LAZ file by using the `hasNearIRData` function.

```
flag = hasNearIRData(lasReader);
disp(flag)

    0
```

Input Arguments

lasReader — LAS or LAZ file reader
lasFileReader object

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

Version History
Introduced in R2022a

See Also

Functions
pcread | pcshow | hasGPSData | hasWaveformData | hasCRSData

Objects
pointCloud | lidarPointAttributes | ibeoLidarReader | lasFileReader | velodyneFileReader

hasWaveformData

Check if LAS or LAZ file has waveform data

Syntax

flag = hasWaveformData(lasReader)

Description

flag = hasWaveformData(lasReader) returns a logical 1 (true) if the specified LAS or LAZ file lasReader contains waveform data. Otherwise, it returns a logical 0 (false).

Examples

Check for Waveform Data in LAZ File

Create a lasFileReader object for a LAZ file. Then, use the hasWaveformData function to check the LAZ file for waveform data to read.

Create a lasFileReader object to access the LAZ file data.

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

Check waveform data in the LAZ file by using the hasWaveformData function.

```
flag = hasWaveformData(lasReader);
disp(flag)

    0
```

Input Arguments

lasReader — LAS or LAZ file reader
lasFileReader object

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

Version History
Introduced in R2022a

See Also

Functions
pcread | pcshow | hasGPSData | hasNearIRData | hasCRSData

Objects
pointCloud | lidarPointAttributes | ibeoLidarReader | lasFileReader |
velodyneFileReader

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 in meters
vector

Range readings from lidar, specified as a vector in meters. 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 in radians
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 in meters
[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.
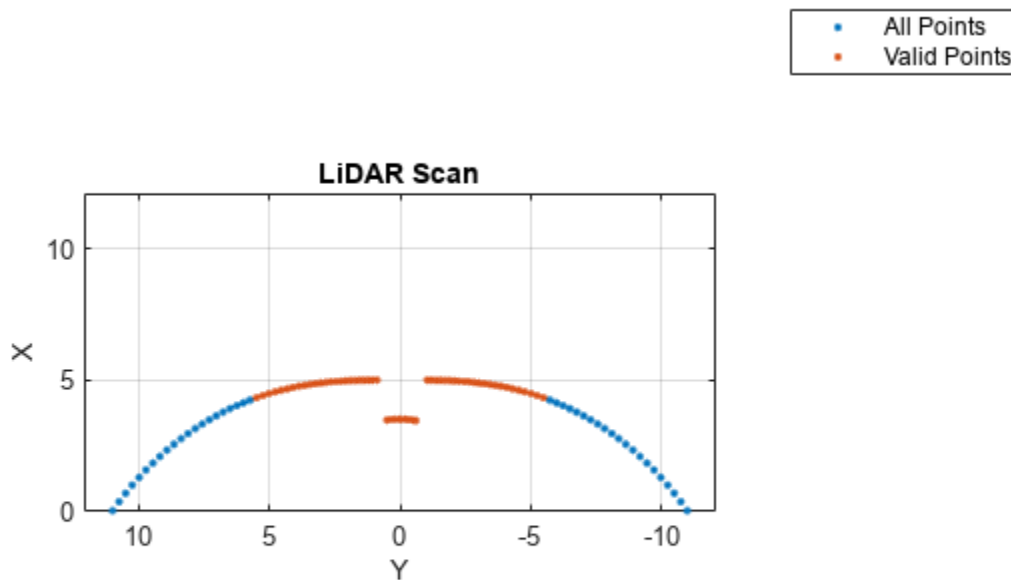
```
x = linspace(-2,2);
ranges = abs((1.5).*x.^2 + 5);
ranges(45:55) = 3.5;
angles = linspace(-pi/2,pi/2,numel(ranges));
```

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
```
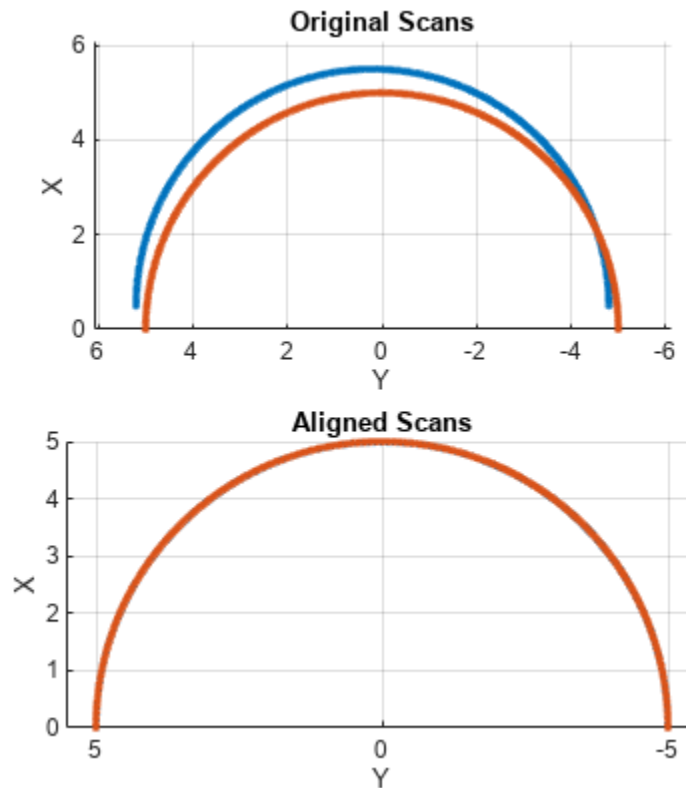

Version History

Introduced in R2020b

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

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')
```
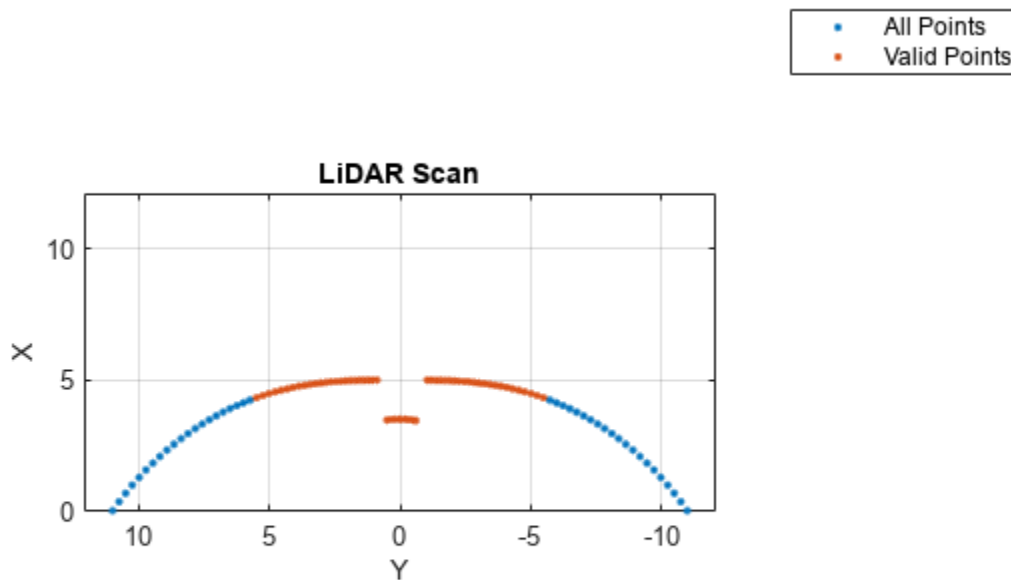
Input Arguments

scanObj — Lidar scan readings
`lidarScan` object

Lidar scan readings, specified as a `lidarScan` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History
Introduced in R2020b

See Also
`matchScans`

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.
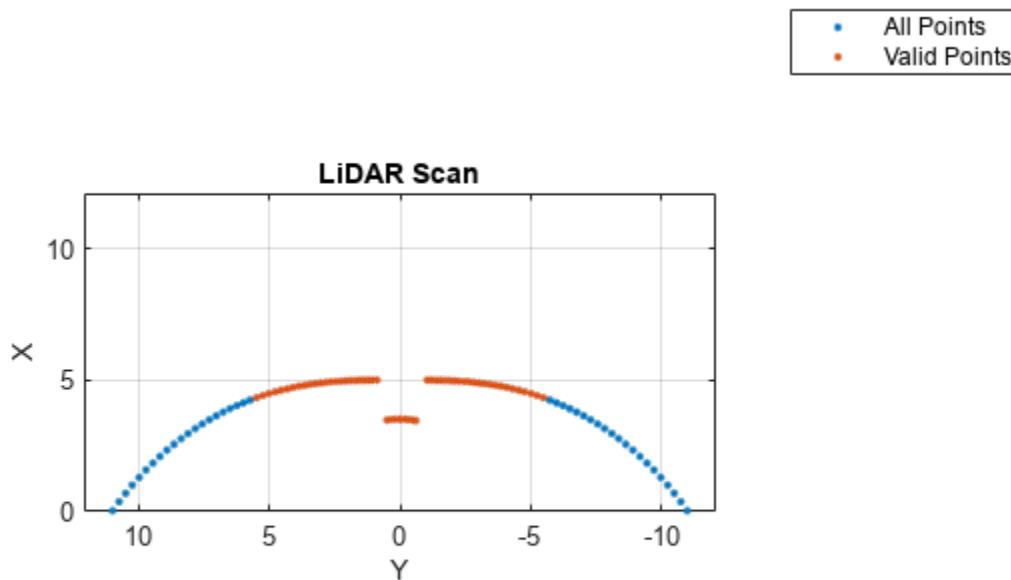
```
x = linspace(-2,2);
ranges = abs((1.5).*x.^2 + 5);
ranges(45:55) = 3.5;
angles = linspace(-pi/2,pi/2,numel(ranges));
```

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 pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

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.

Version History
Introduced in R2020b

See Also
`matchScans`

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;
```

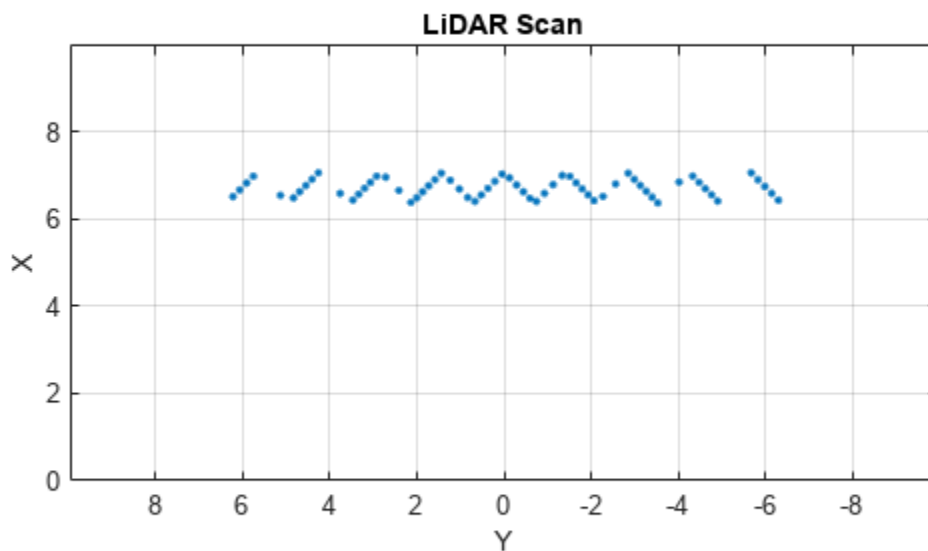Specify the pose of the sensor and the ground-truth map.

```
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)
```

Version History
Introduced in R2020b

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
occupancyMap | binaryOccupancyMap | lidarScan
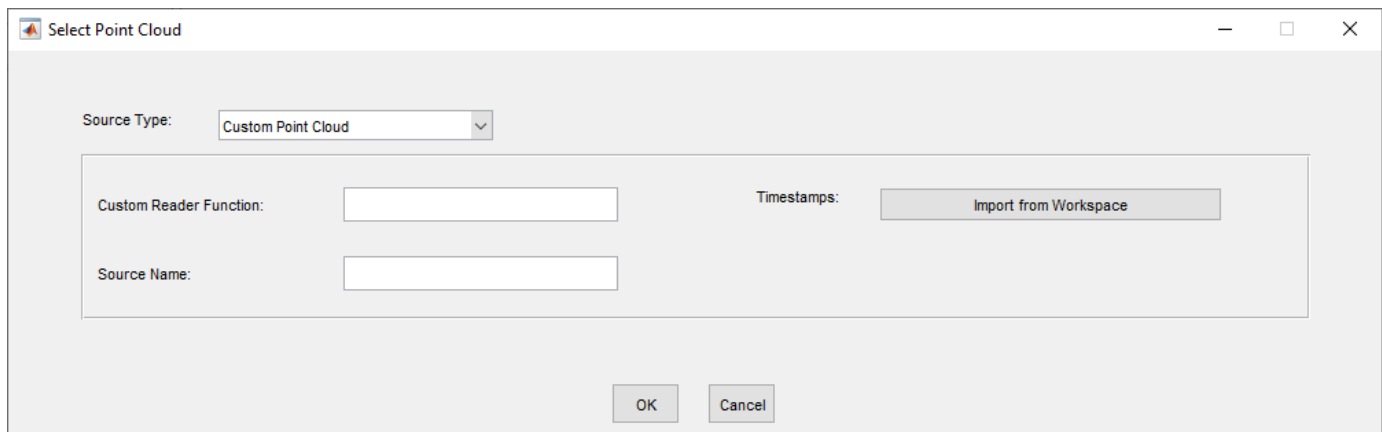
lidar.labeler.loading.CustomPointCloudSource class

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

Load point cloud data from custom sources into Lidar Labeler app

Description

The `lidar.labeler.loading.CustomPointCloudSource` class creates an interface for loading
point cloud data from a custom source into the **Lidar Labeler** app. This class controls the parameters
in the Select Point Cloud dialog box of the app when you set **Source Type** to `Custom Point Cloud`.

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

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

Creation

To create a `CustomPointCloudSource` object, write a custom reader function to read point cloud
data from the data source. Save the file to any folder on the MATLAB path. Alternatively, add the
folder containing the file to the MATLAB path. Then, use the
`lidar.labeler.loading.CustomPointCloudSource` function.

Syntax

`customptCloudSource = lidar.labeler.loading.CustomPointCloudSource`

Description

customptCloudSource = lidar.labeler.loading.CustomPointCloudSource creates a CustomPointCloudSource object for loading a signal from custom source. To specify the data source and the parameters required to load the source, use the loadSource method.

Properties

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

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

Attributes:

GetAccess	public
Constant	true
NonCopyable	true

Description — Description of class functionality
"A custom point cloud source reader" (default) | string scalar

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

Attributes:

GetAccess	public
Constant	true
NonCopyable	true

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

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

Attributes:

GetAccess	public
SetAccess	protected

SourceParams — Parameters for loading point cloud data from a custom source
[] (default) | structure

Parameters for loading point cloud data from a custom 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 custom source, specified as a cell array containing a single `duration` vector of timestamps. In the Select Point Cloud dialog box of the app, you can import the **Timestamps** parameter from a variable in the MATLAB workspace. The `SourceParams` property stores these timestamps in the `Timestamps` field.	Optional

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.

Version History
Introduced in R2021a

See Also

Apps
Lidar Labeler

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

Topics
"Use Custom Point Cloud Source Reader for Labeling"

lidarParameters

Lidar sensor parameters

Description

A `lidarParameters` object stores the parameters of a lidar sensor. To convert unorganized point clouds into organized point clouds using the `pcorganize` function, you must specify these sensor parameters. For more information, see "Lidar Sensor Parameters".

Creation

Syntax

```
params = lidarParameters(sensorName,horizontalResolution)
params = lidarParameters(verticalResolution,verticalFoV,horizontalResolution)
params = lidarParameters(verticalBeamAngles,horizontalResolution)
params = lidarParameters( ___ ,HorizontalFoV=horizontalFoV)
```

Description

`params = lidarParameters(sensorName,horizontalResolution)` returns the sensor parameters of the specified sensor `sensorName` as a `lidarParameters` object. `horizontalResolution` specifies the `HorizontalResolution` property. Use this syntax to load the parameters of a supported sensor. See "Supported Sensors".

`params = lidarParameters(verticalResolution,verticalFoV,horizontalResolution)` stores parameters for a uniform beam configuration lidar sensor. The `verticalResolution`, `verticalFoV`, and `horizontalResolution` arguments set the `VerticalResolution`, `VerticalFoV`, and `HorizontalResolution` properties, respectively.

`params = lidarParameters(verticalBeamAngles,horizontalResolution)` stores parameters for a gradient beam configuration lidar sensor. The `verticalBeamAngles` and `horizontalResolution` arguments set the `VerticalBeamAngles` and `HorizontalResolution` properties, respectively.

`params = lidarParameters( ___ ,HorizontalFoV=horizontalFoV)` specifies the horizontal field-of-view `HorizontalFoV` covered by the sensor in addition to any combination of input arguments from previous syntaxes.

Input Arguments

sensorName — Name of a supported sensor
character vector | string scalar

Name of a supported sensor, specified as a character vector or string scalar. Use this argument to load the parameters of a supported sensor. See "Supported Sensors".

horizontalResolution — Number of channels in horizontal direction
positive integer

Number of channels in the horizontal direction, specified as a positive integer. Typical values include 512 and 1024.

verticalResolution — Number of channels in vertical direction
positive integer

Number of channels in the vertical direction, specified as a positive integer. Typical values include 32 and 64.

verticalFoV — Vertical field-of-view of lidar sensor
two-element vector

Vertical field-of-view of the lidar sensor, specified as a two-element vector.

verticalBeamAngles — Angular position of each vertical channel
N-element vector

Angular position of each vertical channel, specified as an *N*-element vector, in degrees. *N* is the `verticalResolution` of the sensor.

Properties

HorizontalResolution — Number of channels in horizontal direction
positive integer

This property is read-only.

Number of channels in the horizontal direction, stored as a positive integer.

HorizontalFoV — Horizontal field-of-view of lidar sensor
positive scalar

This property is read-only.

Horizontal field-of-view of the lidar sensor, stored as a positive scalar, in degrees.

VerticalResolution — Number of channels in vertical direction
positive integer

Number of channels in the vertical direction, stored as a positive integer.

VerticalFoV — Vertical field-of-view of lidar sensor
two-element vector

Vertical field-of-view of the lidar sensor, stored as a two-element vector, in degrees.

HorizontalAngResolution — Horizontal angular resolution of lidar sensor
positive scalar

Horizontal angular resolution of the lidar sensor, stored as a positive scalar, in degrees.

`HorizontalBeamAngles` — Angular position of each horizontal channel
M-element vector

Angular position of each horizontal channel, stored as an *M*-element vector, in degrees. *M* is the `HorizontalResolution` of the sensor.

`VerticalBeamAngles` — Angular position of each vertical channel
N-element vector

Angular position of each vertical channel, stored as an *N*-element vector, in degrees. *N* is the `VerticalResolution` of the sensor.

Examples

Convert HDL-64E Unorganized Point Cloud into Organized Point Cloud

Load point cloud data into the workspace.

```
fileName = fullfile(toolboxdir("lidar"),"lidardata","lcc","HDL64", ...
                    "pointCloud","0001.pcd");
ptCloudUnorg = pcread(fileName);
```

Specify the horizontal resolution of the lidar sensor.

```
horizontalResolution = 1024;
```

Create a `lidarParameters` object that represents an `HDL64E` sensor with the specified `horizontalResolution`.

```
params = lidarParameters('HDL64E',horizontalResolution);
```

Convert the unorganized point cloud into an organized point cloud.

```
ptCloudOrg = pcorganize(ptCloudUnorg,params);
```

Display the dimensions of the input point cloud.

```
size(ptCloudUnorg.Location)
```

ans = *1×2*

```
    37879            3
```

Display the size of the converted point cloud. `pointCloud` objects store organized point clouds as *M*-by-*N*-by-3 arrays, whereas they store unorganized point clouds as *M*-by-3 matrices

```
size(ptCloudOrg.Location)
```

ans = *1×3*

```
      64        1024            3
```

Create a Lidar Parameters Object

Define lidar sensor parameters.

```
verticalFoV = [2 -24.69];
verticalResolution = 32;
horizontalResolution = 512;
```

Define a `lidarParmaters` object.

```
params = lidarParameters(verticalResolution,verticalFoV,...
                         horizontalResolution)

params =
  lidarParameters with properties:

        HorizontalResolution: 512
          VerticalResolution: 32
                   VerticalFoV: [2 -24.6900]
          VerticalBeamAngles: [2 1.1390 0.2781 -0.5829 -1.4439 -2.3048 ... ]
                 HorizontalFoV: 360
    HorizontalAngResolution: 0.7045
        HorizontalBeamAngles: [0 0.7045 1.4090 2.1135 2.8180 3.5225 ... ]
```

Create Lidar Parameters Object for Gradient Lidar Sensor

Define vertical beam angles of the sensor. Refer the data handbook of the sensor to find the beam angles. To learn more about beam configuration, see "Lidar Sensor Parameters".

```
verticalBeamAngles = [15.0000 3.0000 1.5000 0.8333 0.1667 -0.5000 ...
                      -1.1667 -1.8333 -2.5000 -3.1667 -3.8333 -4.5000 ...
                      -5.1667 -5.8333 -9.0000 -13.0000];
```

Define horizontal resolution of the sensor.

```
horizontalResolution = 512;
```

Define a `lidarParmaters` object.

```
params = lidarParameters(verticalBeamAngles,horizontalResolution)

params =
  lidarParameters with properties:

        HorizontalResolution: 512
          VerticalResolution: 16
                   VerticalFoV: [15 -13]
          VerticalBeamAngles: [15 3 1.5000 0.8333 0.1667 -0.5000 -1.1667 ... ]
                 HorizontalFoV: 360
    HorizontalAngResolution: 0.7045
        HorizontalBeamAngles: [0 0.7045 1.4090 2.1135 2.8180 3.5225 ... ]
```

Version History
Introduced in R2021b

See Also
`pcorganize`

Topics
"Unorganized to Organized Conversion of Point Clouds Using Spherical Projection"
"What are Organized and Unorganized Point Clouds?"

pointPillarsObjectDetector

PointPillars object detector

Description

The `pointPillarsObjectDetector` object defines a PointPillars object detector. To detect objects in a point cloud, pass the trained PointPillars object detector to the `detect` object function.

If you have a pretrained PointPillars deep learning network, you can use the `pointPillarsObjectDetector` function to create the `pointPillarsObjectDetector` object.

If you have training data, you can create an untrained `pointPillarsObjectDetector` object and use the `trainPointPillarsObjectDetector` function to train the model.

Creation

Syntax

```
detector = pointPillarsObjectDetector(pcRange,class,anchorBox)
detector = pointPillarsObjectDetector(net,pcRange,class,anchorBox)
detector = pointPillarsObjectDetector( ___ ,Name=Value)
```

Description

`detector = pointPillarsObjectDetector(pcRange,class,anchorBox)` creates an untrained PointPillars object detector with the specified attributes. To train the object detector, you must specify it as an input to the `trainPointPillarsObjectDetector` function.

`detector = pointPillarsObjectDetector(net,pcRange,class,anchorBox)` creates a PointPillars object detector by using the specified pretrained network `net`.

`detector = pointPillarsObjectDetector( ___ ,Name=Value)` sets the `ModelName`, `VoxelSize`, `NumPillars` and `NumPointsPerPillar` properties by using name-value arguments in addition to any combination of input arguments from previous syntaxes. For example, `pointPillarsObjectDetector(pcRange,class,anchorBox,ModelName="customDetector")` creates a PointPillars object detector with the name `"customDetector"`.

Properties

ModelName — Name of object detector
`' '` (default) | character vector | string scalar

Name of the object detector, specified as a character vector or string scalar.

To set this property, specify it as a name-value argument at object creation. For example, `pointPillarsObjectDetector(net,pcRange,class,anchorBox,ModelName="customDetector")` sets the name for the object detector to `"customDetector"`.

Network — PointPillars object detection network
dlnetwork object

This property is read-only.

PointPillars deep learning network to use for object detection, specified as a dlnetwork object. You can set this property at object creation by using the input argument net.

PointCloudRange — Range of input point cloud
six-element real-valued vector

This property is read-only.

Range of the input point cloud, specified as a six-element vector of the form [*xmin xmax ymin ymax zmin zmax*].

- *xmin* and *xmax* are the minimum and the maximum limits along the *x*-axis, respectively.
- *ymin* and *ymax* are the minimum and the maximum limits along the *y*-axis, respectively.
- *zmin* and *zmax* are the minimum and the maximum limits along the *z*-axis, respectively.

Set this property at object creation by using the input argument pcRange.

ClassNames — Names of object classes
categorical vector | vector of strings | cell array of character vectors

This property is read-only.

Names of the object classes, specified as a categorical vector, a vector of strings or a cell array of character vectors. Set this property at object creation by using the input argument class.

Data Types: char | string | categorical | cell

AnchorBoxes — Anchor boxes
N-by-1 cell array

This property is read-only.

Anchor boxes, specified as an *N*-by-1 cell array. *N* is the number of object classes in the PointPillars deep learning network. Each cell defines an anchor box as a vector of the form [*length width height center angle*].

- *length*, *width*, *height* — represent the length, width, and height of the anchor box, respectively. Specify each value as a positive real number, in meters.
- *center* — represents the center of the anchor box along *z* axis.
- *angle* — represents the orientation of the anchor box along *z* axis in radians, which is the *yaw* angle of the lidar sensor.

Set this property at object creation by using the input argument anchorBox.

Data Types: cell

VoxelSize — Size of pillars
[0.16 0.16] (default) | two-element real-valued vector

This property is read-only.

Size of the pillars, specified as a two-element vector of the form [*length width*], representing the length and width of the voxel in meters.

To set this property, specify it as a name-value argument at object creation. For example, `pointPillarsObjectDetector(net,pcRange,class,anchorBox,VoxelSize=[0.16 0.16])` sets the size of the voxel to [`0.16 0.16`].

NumPillars — Number of prominent pillars
12000 (default) | positive scalar

This property is read-only.

Number of prominent pillars, specified as a positive scalar.

To set this property, specify it as a name-value argument at object creation. For example, `pointPillarsObjectDetector(net,pcRange,class,anchorBox,NumPillars=1000)` sets the number of pillars to `1000`.

NumPointsPerPillar — Minimum number of points per pillar
100 (default) | positive scalar

This property is read-only.

Minimum number of points per pillar, specified as a positive scalar.

To set this property, specify it as a name-value argument at object creation. For example, `pointPillarsObjectDetector(net,pcRange,class,anchorBox,NumPointsPerPillar=100)` sets the minimum number of points per pillar to `100`.

Object Functions
detect   Detect objects using PointPillars object detector

Version History
Introduced in R2021b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.
Usage notes and limitations:

For code generation,

- Only the `detect` method of the `pointPillarsObjectDetector` is supported for code generation.
- Only the `Threshold`, `SelectStrongest`, and `MiniBatchSize` Name-Value pairs of the `detect` method are supported.
- To create a `pointPillarsObjectDetector` object for code generation, see "Load Pretrained Networks for Code Generation" (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For code generation,

- Only the `detect` method of the `pointPillarsObjectDetector` is supported for code generation.
- Only the `Threshold`, `SelectStrongest`, and `MiniBatchSize` Name-Value pairs of the `detect` method are supported.
- To create a `pointPillarsObjectDetector` object for code generation, see "Load Pretrained Networks for Code Generation" (GPU Coder).

See Also

Apps
Lidar Labeler | **Lidar Viewer**

Functions
`trainPointPillarsObjectDetector` | `pcorganize` | `pointnetplusLayers`

Topics
"Lidar 3-D Object Detection Using PointPillars Deep Learning"
"Code Generation For Lidar Object Detection Using PointPillars Deep Learning"
"Unorganized to Organized Conversion of Point Clouds Using Spherical Projection"
"Getting Started with PointPillars"
"Getting Started with Point Clouds Using Deep Learning"
"Datastores for Deep Learning" (Deep Learning Toolbox)

detect

Detect objects using PointPillars object detector

Syntax

```
bboxes = detect(detector,ptCloud)
[bboxes,scores] = detect(detector,ptCloud)
[ ___ ,labels] = detect(detector,ptCloud)

detectionResults = detect(detector,DS)

[ ___ ] = detect( ___ ,Name=Value)
```

Description

`bboxes = detect(detector,ptCloud)` detects objects within the input point cloud, `ptCloud`. The function returns the locations of detected objects as a set of bounding boxes.

`[bboxes,scores] = detect(detector,ptCloud)` returns the class-specific confidence score for each bounding box.

`[ ___ ,labels] = detect(detector,ptCloud)` returns the label assigned to each bounding box. The labels used for object classes are defined during training by using the `trainPointPillarsObjectDetector` function.

`detectionResults = detect(detector,DS)` detects objects within the series of point clouds in the datastore `DS`.

`[ ___ ] = detect( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to any combination of arguments from previous syntaxes. For example, `detect(detector,ptCloud,Threshold=0.5)` detects objects within the input point cloud with a detection threshold of `0.5`.

Examples

Detect Vehicles Using PointPillars Network

Load a pretrained PointPillars object detector into the workspace.

```
pretrainedDetector = load("pretrainedPointPillarsDetector.mat","detector");
detector = pretrainedDetector.detector;
```

Read the input point cloud using the `pcread` function.

```
ptCloud = pcread("PandasetLidarData.pcd");
```

Run the pretrained object detector on the point cloud.

```
[bboxes,scores,labels] = detect(detector,ptCloud);
bboxCar=bboxes(labels'=="Car",:);
bboxTruck=bboxes(labels'=="Truck",:);
```

Visualize the results using the `pcshow` function. For better visualization, select a region of interest, `roi`, from the point cloud data. Display the bounding boxes for cars, trucks using the `showShape` function.

```
roi = [0.0 89.12 -49.68 49.68 -5.0 5.0];
indices = findPointsInROI(ptCloud,roi);
figure
ax = pcshow(select(ptCloud,indices).Location);
zoom(ax,1.5)
showShape("cuboid",bboxCar,Color="green",Parent=ax,Opacity=0.3,LineWidth=1)
hold on;
showShape("cuboid",bboxTruck,Color="red",Parent=ax,Opacity=0.3,LineWidth=1)
```
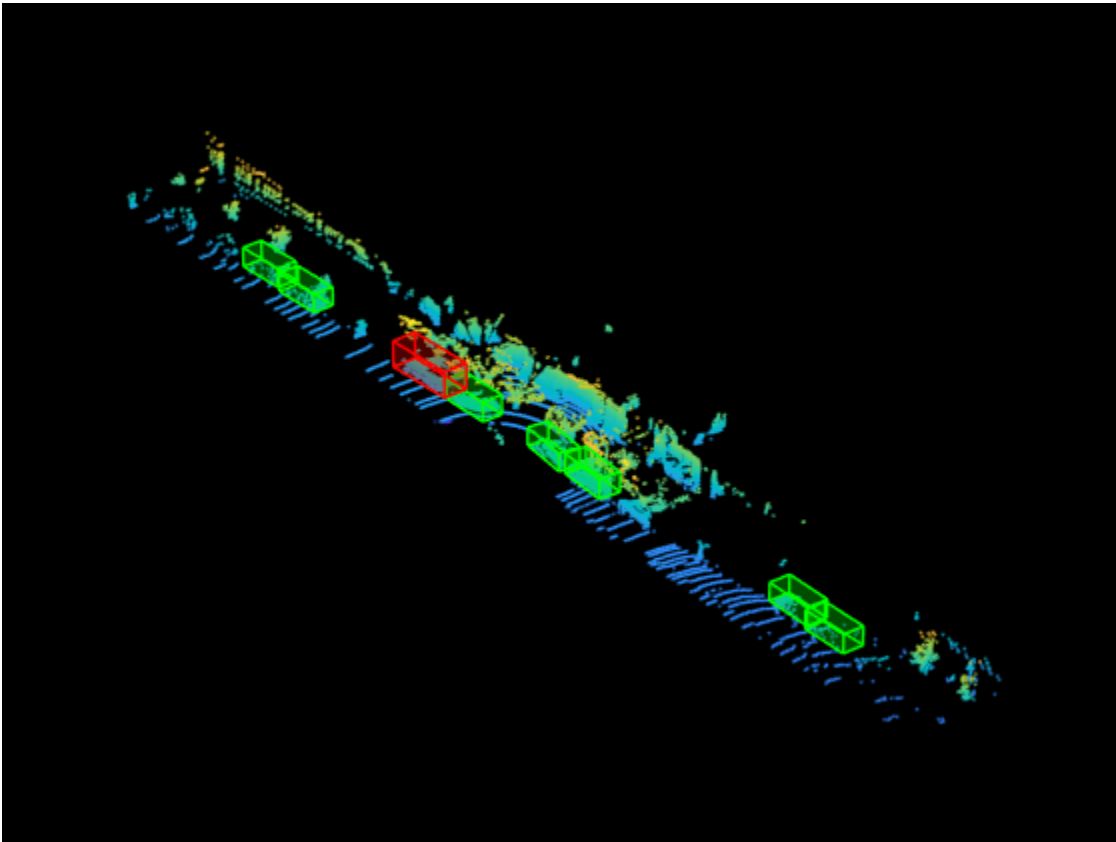

Input Arguments

detector — PointPillars object detector
`pointPillarsObjectDetector` object

PointPillars object detector, specified as a `pointPillarsObjectDetector` object.

ptCloud — Input point cloud
`pointCloud` object

Input point cloud, specified as a `pointCloud` object. This object must contain the locations, intensities, and RGB colors necessary to render the point cloud.

DS — Datastore
valid datastore object

Datastore, specified as a valid datastore object, which is a collection of point clouds. This datastore must be set up such that using the `read` function on the datastore object returns a cell array or table, the first column of which contains point clouds. For more information on creating datastore objects, see the `datastore` function.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `detect(detector,ptCloud,Threshold=0.5)`

Threshold — Detection threshold
`0.5` (default) | scalar in the range [0, 1]

Detection threshold, specified as a scalar in the range [0, 1]. The function removes detections that have scores lower than this threshold value. To reduce false positives, increase this value.

SelectStrongest — Strongest bounding box selection
`true` or `1` (default) | `false` or `0`

Strongest bounding box selection for each detected object, specified as a logical `1` (`true`) or `0` (`false`).

- `true` — The function returns the strongest bounding box per object. The function uses the `selectStrongestBboxMulticlass` function, which uses nonmaximal suppression to eliminate overlapping bounding boxes based on their confidence scores.

  By default, `detect` uses this code for the `selectStrongestBboxMulticlass` function:

  ```
  selectStrongestBboxMulticlass(bbox,scores,RatioType="Union", ...
                                OverlapThreshold=0.1);
  ```

- `false` — The function returns all the detected bounding boxes. You can then use a custom process to eliminate overlapping bounding boxes.

MiniBatchSize — Size of mini-batch
`8` (default) | positive scalar

Size of mini-batch, specified as a positive scalar. Use the `MiniBatchSize` argument to process a large collection of point clouds. Using this argument, the function groups point clouds into mini-batches and processes them as a batch to improve computational efficiency. Increase the mini-batch size to decrease processing time. Decrease the size to use less memory.

Acceleration — Performance optimization
`"auto"` (default) | `"none"`

Performance optimization, specified as `"auto"` or `"none"`.

- `"auto"` — Automatically selects the optimizations suitable for the network and environment of the detector. These optimizations improve performance at the expense of some overhead on the first call and possible additional memory usage.

- "none" — Disables all acceleration.

Output Arguments

bboxes — Locations of objects detected
M-by-9 matrix

Locations of the objects detected within the point cloud, returned as an *M*-by-9 matrix. Each row in the matrix is of the form [*x y z length width height roll pitch yaw*], representing the dimension and location of the bounding box. *M* is the number of bounding boxes.

scores — Detection confidence scores
M element column vector

Detection confidence scores for the bounding boxes, returned as an *M* element column vector. *M* is the number of bounding boxes. The score for each detection is the product of its objectness prediction and classification scores. Each score is in the range [0, 1]. A higher score indicates higher confidence in the detection.

labels — Labels for bounding boxes
M-by-1 categorical array

Labels for bounding boxes, returned as an *M*-by-1 categorical array. *M* is the number bounding boxes in the point cloud. Define the class names used to label the objects when you train the object detector.

detectionResults — Detection results
table

Detection results, returned as a table with columns, `Boxes`, `Scores`, and `Labels`. Each row of the table corresponds to a point cloud from the input datastore.

Column Name	Value	Description
Boxes	*M*-by-9 matrix, where *M* is the number of bounding boxes.	Bounding boxes for objects found in the corresponding point cloud.
Scores	*M* element column vector	Detection scores for the bounding boxes.
Labels	*M*-by-1 categorical array	Labels for the bounding boxes.

To evaluate the detection results, use the `evaluateDetectionAOS` function.

```
metrics = evaluateDetectionAOS(detectionResults,testLabels);
```

Version History
Introduced in R2021b

Extended Capabilities

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

Usage notes and limitations:

For code generation, only the `Threshold`, `SelectStrongest`, and `MiniBatchSize` Name-Value pairs of the `detect` method are supported.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For code generation, only the `Threshold`, `SelectStrongest`, and `MiniBatchSize` Name-Value pairs of the `detect` method are supported.

See Also

Apps
Lidar Labeler | **Lidar Viewer**

Functions
`trainPointPillarsObjectDetector`

Objects
`pointPillarsObjectDetector` | `yolov2ObjectDetector`

Topics
"Lidar 3-D Object Detection Using PointPillars Deep Learning"
"Code Generation For Lidar Object Detection Using PointPillars Deep Learning"
"Unorganized to Organized Conversion of Point Clouds Using Spherical Projection"
"Getting Started with PointPillars"
"Getting Started with Point Clouds Using Deep Learning"
"Datastores for Deep Learning" (Deep Learning Toolbox)

lidarPointAttributes

Object for storing lidar point attributes

Description

The `lidarPointAttributes` object stores additional point attributes that are not stored in a `pointCloud` object. These attributes are associated to lidar data.

Creation

Use either `lidarPointAttributes` function or the `readPointCloud` function to create a `lidarPointAttributes` object. The `readPointCloud` function also creates a `pointCloud` object.

To create a `lidarPointAttributes` object using `readPointCloud` function, see the "Read Attributes from LAS File" on page 2-336 example.

Syntax

`attr = lidarPointAttributes(Name=Value)`

Description

`attr = lidarPointAttributes(Name=Value)` creates a `lidarPointAttributes` object with properties set using one or more name-value arguments.

Properties

Count — Number of available point records in file
nonnegative integer

This property is read-only.

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

Classification — Classification numbers
[ ] (default) | *M*-by-1 vector of nonnegative integers

Classification numbers of each point, specified as an *M*-by-1 vector of nonnegative integers. *M* is equal to the number of available point records.

For LAS file point data record formats 0 to 5, the classification number ranges from 0 to 31.

Classification Number	Classification Type
0	Created, never classified
1	Unclassified
2	Ground

Classification Number	Classification Type
3	Low vegetation
4	Medium vegetation
5	High vegetation
6	Building
7	Low point (noise)
8	Model key-point
9	Water
10	Reserved
11	Reserved
12	Overlap points
13 - 31	Reserved

For LAS file point data record formats 6 to 10, the classification number ranges 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
13	Wire guard (shield)
14	Wire conductor (phase)
15	Transmission tower
16	Wire-structure connector (insulator)
17	Bridge deck
18	High noise
19	Overhead structure
20	Ignored ground
21	Snow
22	Temporal exclusion

Classification Number	Classification Type
23- 63	Reserved
64 - 255	User-defined

These are standard class names and class-object mappings. The class definition and mapping might differ depending on the application that created the LAS or LAZ file.

Example: `Classification=[0 255 128]'` specifies the classification numbers for three points as 0, 255, and 128.

Data Types: `uint8`

LaserReturn — Laser pulse return numbers
`[]` (default) | *M*-by-1 vector of positive integers

Laser pulse return numbers of each point, specified as an *M*-by-1 vector of positive integers. *M* is equal to the number of available point records.

For LAS file point data record formats 0 to 5, the values are in the range 1 to 5, and for point data record formats 6 to 10, the values are in the range 1 to 15.

Example: `LaserReturn=[10 15 1]'` specifies the laser pulse return numbers for three points as 10, 15, and 1.

Data Types: `uint8`

NumReturns — Total number of returns
`[]` (default) | *M*-by-1 vector of positive integers

Total number of returns for a pulse, specified as an *M*-by-1 vector of positive integers. *M* is equal to the number of available point records.

For LAS file point data record formats 0 to 5, the values are in the range 1 to 5, and for point data record formats 6 to 10, the values are in the range 1 to 15.

Example: `NumReturns=[1 10 15]'` specifies the total number of returns for three points as 1, 10, and 15.

Data Types: `uint8`

GPSTimeStamp — GPS time stamps
`[]` (default) | *M*-by-1 `duration` vector

GPS time stamps of each point, specified as an *M*-by-1 `duration` vector in seconds. *M* is equal to the number of available point records.

Example: `GPSTimeStamp=seconds(1:3)'` specifies the GPS time stamps for three points as 1, 2, and 3 seconds.

NearIR — Near infrared channel value
`[]` (default) | *M*-by-1 vector of positive integers

Near infrared channel value of each point, specified as an *M*-by-1 vector of positive integers. *M* is equal to the number of available point records. Values must be in the range [0, 65535].

Example: `NearIR=ones(3,1)` specifies the near infrared channel values for three points as 1.

Data Types: `uint16`

ScanAngle — Scan angle
[] (default) | *M*-by-1 vector

Scan angle of each point, specified as an *M*-by-1 vector. Each value represents the rotational angle at which the point is captured in the laser system. The angle is negative to the left of the front of the sensor, positive to the right, and 0 degrees directly in front. *M* is equal to the number of available point records.

For LAS file point data record formats 0 to 5, the values are in the range –90 to 90, and for point data record formats 6 to 10, the value ranges from –180 to 180.

Example: `ScanAngle=[0 -180 105]'` specifies the scan angles for three points as directly ahead, 180 degrees to the left, and 105 degrees to the right.

Data Types: `single`

PointSourceID — Point source ID
[] (default) | *M*-by-1 vector of nonnegative integers

Point source ID of each point, specified as an *M*-by-1 vector of nonnegative integers. Each element defines the source from which a point originates. *M* is equal to the number of available point records.

A source is a grouping of temporally consistent data, such as a flight line for aerial systems. The values are in the range [0, 65535].

Example: `PointSourceID=[0 1 65535]'` specifies the point source IDs for three points as 0, 1, and 65535.

Data Types: `uint16`

ScannerChannel — Scanner channel
[] (default) | *M*-by-1 vector of nonnegative integers

Scanner channel of each point, specified as an *M*-by-1 vector of nonnegative integers. *M* is equal to the number of available point records.

For single channel systems the value is `0`. The values are in the range [0, 3].

Example: `ScannerChannel=[0 0 0]'` specifies the scanner channels for three points as `0`.

Data Types: `uint8`

ScanDirectionFlag — Scan direction flag
[] (default) | *M*-by-1 logical vector

Scan direction flag of each point, specified as an *M*-by-1 logical vector. *M* is equal to the number of available point records.

The value specifies the direction of scanner mirror motion during the capture of the corresponding point. Left-to-right motion is defined as positive, while right-to-left motion is defined as negative. A logical `1` (`true`) denotes a positive scan direction for the point, and a logical `0` (`false`) denotes a negative scan direction.

Example: `ScanDirectionFlag=false(3,1)` specifies the scan direction flags for three points as right-to-left.

Data Types: `logical`

EdgeOfFlightLineFlag — Edge of flight line flag
[ ] (default) | *M*-by-1 logical vector

Edge of flight line flag of each point, specified as an *M*-by-1 logical vector. *M* is equal to the number of available point records.

Logical `1` (`true`) indicates that the point lies on the edge of the flight line. Otherwise, the corresponding element is a logical `0` (`false`).

Example: `EdgeOfFlightLineFlag=true(3,1)` specifies that three points lie on the edge of the flight line.

Data Types: `logical`

ClassificationFlags — Classification flags
empty structure (default) | structure

Classification flags, specified as a structure. The structure contains these fields:

- `Synthetic` — An *M*-by-1 logical vector, where logical `1` (`true`) indicates that the point was generated synthetically. Otherwise, the point is specified as logical `0` (`false`).
- `Keypoint` — An *M*-by-1 logical vector, where `true` indicates that the point is a model key-point. Otherwise, the point is specified as `false`.
- `Withheld` — An *M*-by-1 logical vector, where `true` indicates that the point is withheld from the processing algorithm. Otherwise, the point is specified as `false`.
- `Overlap` — An *M*-by-1 logical vector, where `true` indicates that the point lies within the overlap region of two or more swaths. Otherwise, the point is specified as `false`.

Example:
`ClassificationFlags=struct(Synthetic=true(3,1),Keypoint=false(3,1),Withheld=true(3,1),Overlap=false(3,1))` specifies the classification flags for three points as a structure.

WaveformData — Waveform data
empty structure (default) | structure

Waveform data, specified as a structure. The waveform data is stored in point records and the corresponding variable length records of a LAS or LAZ file. The structure contains these fields:

Field	Value	Description
WavePacketDescriptorIndex	*M*-by-1 vector of type `double`	If this value is greater than zero, then the corresponding point has waveform information in the VLR record indicated by the value of this field + 99. When this value is zero, the point has no waveform data.

Field	Value	Description
ByteOffsetToWaveformPacketData	*M*-by-1 vector of type `double`	Defines the location of the waveform packet of a particular point within the stored waveform data. This data is present in an extended variable length record (EVLR) or an auxiliary WPD file.
WaveformPacketSize	*M*-by-1 vector of type `double`	Defines the size of waveform packet in bytes. This value is present in an extended variable length record (EVLR) or an auxiliary WPD file.
ReturnPointLocation	*M*-by-1 vector of type `double`	Each element represents the offset from the first digitized value to the location within the waveform packet at which the associated return pulse was detected. These values are represented in picoseconds.
Xt	*M*-by-1 vector of type `double`	Defines the X component of parametric line equation. For more information, see "Parametric line equation" on page 2-338.
Yt	*M*-by-1 vector of type `double`	Defines the Y component of parametric line equation. For more information, see "Parametric line equation" on page 2-338.
Zt	*M*-by-1 vector of type `double`	Defines the Z component of parametric line equation. For more information, see "Parametric line equation" on page 2-338.
BitsPerSample	*M*-by-1 vector of type `double`	Number of bits for each sample in the range 2 to 32 bits.
CompressionType	*M*-by-1 vector of type `double`	Compression algorithm for waveform packets. Value 0 represents no compression. Reserved for future use.

Field	Value	Description
NumberOfSamples	*M*-by-1 vector of type `double`	Defines the number of samples in decompressed waveform packet.
TemporalSpacing	*M*-by-1 vector of type `double`	Defines the temporal sample spacing in picoseconds.
DigitizerGain	*M*-by-1 vector of type `double`	Defines the digitizer gain to use to convert raw digitized value to an absolute digitizer voltage.
DigitizerOffset	*M*-by-1 vector of type `double`	Defines the digitizer offset to use to convert raw digitized value to an absolute digitizer voltage.

This example specifies the waveform data for three points as a structure.

```
WaveformData = struct(WavePacketDescriptorIndex=[2011 2200 3215]', ...
ByteOffsetToWaveformPacketData=[51 65 78]', ...
WaveformPacketSize=[12 28 25]', ...
ReturnPointLocation=[1 2 3]', ...
Xt=[1 2 3]',Yt=[3 4 5]',Zt=[5 6 7]', ...
BitsPerSample=[1 2 3]', ...
CompressionType=[1 0 1]', ...
NumberOfSamples=[10 25 32]', ...
TemporalSpacing=[11 21 31]', ...
DigitizerGain=[11 12 13]', ...
DigitizerOffset=[21 22 23]')
```

UserData — User data
[] (default) | *M*-by-1 vector of integers

User data, specified as an *M*-by-1 vector of integers. *M* is equal to the number of available point records. This value corresponds to the user data field of the point record data in the LAS file. Use this field at your discretion.

Example: `UserData=[1 2 4]'` specifies the user data for three points.

Examples

Read Attributes from LAS File

Create a `lasFileReader` object for a LAS file. Then, use the `readPointCloud` function to read attributes from the LAS file and generate a `lidarPointAttributes` object.

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

```
path = fullfile(toolboxdir("lidar"),"lidardata", ...
    "las","aerialLidarData2.las");
lasReader = lasFileReader(path);
```

Read all points and point attributes from the LAS file to a `pointCloud` object and `lidarPointAttributes` object, respectively, by using the `readPointCloud` function.

```
[ptCloud,pointAttributes] = readPointCloud(lasReader,Attributes= ...
        ["Classification","LaserReturn","NumReturns", ...
        "EdgeOfFlightLine","ScanAngle"]);
```

Display the point attributes.

```
disp(pointAttributes)
```

```
  lidarPointAttributes with properties:

                   Count: 78970
          Classification: [78970x1 uint8]
              LaserReturn: [78970x1 uint8]
               NumReturns: [78970x1 uint8]
            GPSTimeStamp: [0x0 duration]
                   NearIR: []
                ScanAngle: [78970x1 single]
            PointSourceID: []
           ScannerChannel: []
        ScanDirectionFlag: []
       EdgeOfFlightLineFlag: [78970x1 logical]
       ClassificationFlags: [1x1 struct]
             WaveformData: [1x1 struct]
                 UserData: []
```

Create `lidarPointAttributes` Object

Specify attributes of the `lidarPointAttributes` object for three points.

```
classificationValues=[0 255 128]';
laserReturns=[10 15 1]';
numReturns=[1 10 15]';
```

Create a `lidarPointAttributes` object.

```
attr=lidarPointAttributes(Classification=classificationValues, ...
    LaserReturn=laserReturns,NumReturns=numReturns);
```

Display the point attributes.

```
disp(attr)
```

```
  lidarPointAttributes with properties:

                   Count: 3
          Classification: [3x1 uint8]
              LaserReturn: [3x1 uint8]
               NumReturns: [3x1 uint8]
            GPSTimeStamp: [0x0 duration]
                   NearIR: []
                ScanAngle: []
            PointSourceID: []
           ScannerChannel: []
        ScanDirectionFlag: []
       EdgeOfFlightLineFlag: []
       ClassificationFlags: [1x1 struct]
```

```
WaveformData: [1x1 struct]
    UserData: []
```

Algorithms

The values of `Xt`, `Yt`, and `Zt` fields in `WaveformData` property define a parametric line equation for extrapolating points along the associated waveform. The position along the wave is given by:

$$X = X_0 + X_t$$
$$Y = Y_0 + Y_t$$
$$Z = Z_0 + Z_t$$

where X, Y, and Z represent the spatial position of the derived point, X_0, Y_0, and Z_0 define the position of the anchor point, X_t, Y_t, and Z_t define the position of the point at a distance of time t, in picoseconds, relative to the anchor point. The X, Y, and Z units are identical to the units of the coordinate system of the LAS data. If the coordinate system is geographic, the horizontal units are decimal degrees and the vertical units are in meters.

The fields `Xt`, `Yt`, and `Zt` have been renamed to `dx`, `dy`, and `dz` in latest LAS file specification. For more information, see the ASPRS LASER (LAS) File Format Exchange Activities page.

Version History
Introduced in R2022a

WaveformData property has additional fields

The structure for the `WaveformData` property has these additional fields.

- `WavePacketDescriptorIndex`
- `ByteOffsetToWaveformPacketData`
- `WaveformPacketSize`

WaveformData structure fields are updated
Behavior change in future release

The order of fields in the `WaveformData` structure has been updated to:

```
        WavePacketDescriptorIndex: []
   ByteOffsetToWaveformPacketData: []
               WaveformPacketSize: []
               ReturnPointLocation: []
                              Xt: []
                              Yt: []
                              Zt: []
                    BitsPerSample: []
                  CompressionType: []
                  NumberOfSamples: []
                  TemporalSpacing: []
                    DigitizerGain: []
                  DigitizerOffset: []
```

Prior to R2022b, the order was:

```
                              Xt: []
                              Yt: []
                              Zt: []
             ReturnPointLocation: []
                   BitsPerSample: []
                 CompressionType: []
                 NumberOfSamples: []
                 TemporalSpacing: []
                   DigitizerGain: []
                 DigitizerOffset: []
```

See Also

Functions
pcread | pcshow | readPointCloud | writePointCloud

Objects
pointCloud | lasFileReader | lasFileWriter | ibeoLidarReader | velodyneFileReader

blockedPointCloud

Point cloud made from discrete blocks

Description

A `blockedPointCloud` object is a point cloud made from discrete blocks. Use blocked point clouds when a point cloud is too large to fit into memory. With a blocked point cloud, you can perform processing without running out of memory.

Creation

Syntax

```
bpc = blockedPointCloud(source,blockSize)
bpcs = blockedPointCloud(sources,blockSize)
bpc = blockedPointCloud( ___ ,Name=Value)
```

Description

`bpc = blockedPointCloud(source,blockSize)` creates a read-only `blockedPointCloud` object from the specified source `source` with the specified block size `blockSize`. The source can be a `pointCloud` object or the name of a file or folder that contains point cloud data.

`bpcs = blockedPointCloud(sources,blockSize)` creates an array of `blockedPointCloud` objects from multiple sources `sources` with the specified block size `blockSize`. The length of `bpcs` is equal to the number of sources in `sources`.

`bpc = blockedPointCloud( ___ ,Name=Value)` specifies the `Adapter` and `AlternateFileSystemRoots` properties using one or more name-value arguments.

Input Arguments

source — Source of point cloud data
`pointCloud` object | character vector | string scalar

Source of the point cloud data, specified as a `pointCloud` object, or as a character vector or string scalar specifying the name of a file or folder.

The `blockedPointCloud` function supports these `source` formats:

- Single LAS or LAZ file.
- `pointCloud` object.
- Name of file or folder that contains point cloud data.

sources — Sources of point cloud data
cell array of character vectors | string array | `Fileset` object

Sources of the point cloud data, specified as a cell array of character vectors, string array, or `FileSet` object. `blockedPointCloud` function creates an array of `blockedPointCloud` objects.

blockSize — Size of blocks
scalar | two-element row vector | three-element row vector

Size of the blocks, specified as a scalar, a two-element row vector, or a three-element row vector. The value you specify determines which dimensions the function blocks the point cloud along.

- Scalar — The function blocks the point cloud along the *X*-axis.
- Two-element row vector — The function blocks the point cloud along the *X*- and *Y*-axes.
- Three-element row vector — The function blocks the point cloud along the *X*-, *Y*- and *Z*-axes.

Properties

Adapter — Read and write interface for blocked point cloud object
`LAS` object | `InMemory` object | `LASBlocks` object | `MATBlocks` object

Read and write interface for the blocked point cloud object, specified as one of these adapter objects.

Adapter	Description
LAS	Store blocks in a single LAS file
InMemory	Store blocks in a variable in main memory
LASBlocks	Stores each block as a LAS file in a folder
MATBlocks	Stores each block as a MAT file in a folder

You can also create your own adapter using the `lidar.blocked.Adapter` class.

To set this property, you must specify it at object creation.

Example: `Adapter=lidar.blocked.LAS`

AlternateFileSystemRoots — Alternate file system path
string array | character vector | cell array of character vectors

Alternate file system path for the files specified in the source, specified as a string array, character vector, or cell array of character vectors containing one or more rows. Each row specifies a set of equivalent root paths.

Example: `AlternateFileSystemRoots=["Z:\datasets", "/mynetwork/datasets"]`

Data Types: `char` | `string` | `cell`

BlockSize — Size of blocks
scalar | two-element row vector | three-element row vector

Size of the blocks, specified as a scalar, a two-element row vector, or a three-element row vector. The value you specify determines which dimensions the function blocks the point cloud along.

- Scalar — The function blocks the point cloud along the *X*-axis.
- Two-element row vector — The function blocks the point cloud along the *X*- and *Y*-axes.
- Three-element row vector — The function blocks the point cloud along the *X*-, *Y*- and *Z*-axes.

Data Types: `double`

SizeInBlocks — Size expressed as number of blocks
three-element row vector of positive integers

This property is read-only.

Size expressed as the number of blocks, specified as a three-element row vector of positive integers. The element of the vector specify the number of blocks in the *X*-, *Y*- and *Z*-axes respectively. This property is dependent on the `BlockSize` property. The value includes partial blocks.

Data Types: `integer`

Source — Source of point cloud data
`pointCloud` object | character vector | string scalar

This property is read-only.

Source of the point cloud data, specified as a `pointCloud` object, or as a character vector or string scalar specifying the name of a file or folder.

Data Types: `char` | `string`

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. The first and second elements represent the minimum and maximum values of point cloud coordinates along the *X*-axis, respectively.

Data Types: `double`

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. The first and second elements represent the minimum and maximum values of point cloud coordinates along the *Y*-axis, respectively.

Data Types: `double`

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. The first and second elements represent the minimum and maximum values of point cloud coordinates along the *Z*-axis, respectively.

Data Types: `double`

Object Functions

apply	Process blocks of blocked point cloud
blocksub2roi	Convert block subscripts to ROI limits
gather	Collect blocks of blocked point cloud into workspace
getblock	Read specific block of blocked point cloud
getRegion	Read arbitrary region of blocked point cloud
roi2blocksub	Convert ROI to block subscripts
write	Write blocked point cloud data to new destination

Examples

Create Blocked Point Cloud

Create a blocked point cloud from a LAZ file.

```
pcfile = fullfile(toolboxdir("lidar"),"lidardata", ...
                    "las","aerialLidarData.laz");
```

```
bpc = blockedPointCloud(pcfile,[50 50]);
```

Display the details of the blocked point cloud.

```
disp(bpc)
```

```
  blockedPointCloud with properties:

   Read-only properties.
            Source: "B:\matlab\toolbox\lidar\lidardata\las\aerialLidarData.laz"
           Adapter: [1x1 lidar.blocked.LAS]
       SizeInBlocks: [9 6 1]
            XLimits: [4.2975e+05 4.3015e+05]
            YLimits: [3.6798e+06 3.6801e+06]
            ZLimits: [72.7900 125.8200]
    ClassUnderlying: "pointCloud"

   Settable properties
         BlockSize: [50 50 53.0300]
```

Version History

Introduced in R2022a

See Also

blockedPointCloudDatastore

apply

Process blocks of blocked point cloud

Syntax

```
bres = apply(bpc,fcn)
[bres1,bres2,...] = apply(bpc,fcn)
[bres1s,bres2s,...] = apply(bpcs,fcn)
[ ___ ] = apply( ___ ,Name=Value)
```

Description

`bres = apply(bpc,fcn)` processes the entire `blockedPointCloud` object `bpc` by applying the function handle `fcn` to each block. Returns `bres`, a new blocked point cloud containing the processed data.

`[bres1,bres2,...] = apply(bpc,fcn)` returns multiple output arguments. The specified function handle `fcn` must point to a user function that returns the same number of arguments.

`[bres1s,bres2s,...] = apply(bpcs,fcn)` processes the array of blocked point cloud `bpcs` by applying the function handle `fcn` to each block of each blocked point cloud. Returns an array of blocked point clouds containing the processed data.

`[ ___ ] = apply( ___ ,Name=Value)` modifies aspects of block processing using name-value arguments.

Examples

Downsample Point Cloud Data

Create a full file path for a LAZ file that contains aerial lidar data.

```
pcfile = fullfile(toolboxdir("lidar"),"lidardata", ...
                  "las","aerialLidarData.laz");
```

Create a `pcdownsample` function handle.

```
fun = @(block)pcdownsample(block.Data,random=0.1);
```

Create a `blockedPointCloud` object using the LAZ file.

```
bpc = blockedPointCloud(pcfile,[50 50]);
```

Process each block of the aerial point cloud data using the specified `pcdownsample` function handle.

```
pcdown = apply(bpc,fun);
```

Extract Normals from Point Cloud

Create a full file path for a LAZ file that contains aerial lidar data.

```
pcfile = fullfile(toolboxdir("lidar"),"lidardata", ...
                    "las","aerialLidarData.laz");
```

Create a `pcnormals` function handle.

```
fun = @(block)pcnormals(block.Data);
```

Create a `blockedPointCloud` object using the LAZ file.

```
bpc = blockedPointCloud(pcfile,[50 50]);
```

Create a MAT adapter for writing the files from the block processing operation as MAT files.

```
matad = lidar.blocked.MATBlocks;
outfile = fullfile(tempdir,"pcnormout");
```

Perform a block processing operation on the point cloud file to extract the normals.

```
pcnorms = apply(bpc,fun,Adapter=matad,OutputLocation=outfile);
```

Input Arguments

bpc — Blocked point cloud
blockedPointCloud object

Blocked point cloud, specified as a `blockedPointCloud` object.

bpcs — Blocked point clouds
array of blockedPointCloud objects

Blocked point clouds, specified as an array of `blockedPointCloud` objects.

fcn — Processing function
function handle

Processing function, specified as a function handle. For more information, see "Create Function Handle". The processing function `fcn` must accept a `bstruct` as input. To pass additional arguments, specify `fcn` as an anonymous function. For more information, see "Anonymous Functions".

`bstruct` is a structure with these fields:

Field	Description
Data	Block of data from `bpc`
BlockSize	Value of the `BlockSize` parameter.
ROI	ROI of the block, specified as a six-element numeric row vector in the order [*xmin xmax ymin ymax zmin zmax*].

Field	Description
PointAttributes	Attributes for each point, specified as a lidarPointAttributes object.
PCNumber	Index into the bpc array from the current point cloud.

The function fcn typically returns the results for one block. The results can be a pointCloud object or a structure.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: apply(bpc,fun,DisplayWaitbar=true) displays a wait bar for operations with long run times.

Adapter — Adapter used for writing blocked point cloud data
adapter object

Adapter used for writing blocked point cloud data, specified as an adapter object. To specify different adapters for different outputs, use a cell array. If you specify a scalar object, the function uses the specified adapter for every output.

This table lists the adapters included with the toolbox.

Adapter	Description
LAS	Store blocks in a single LAS file
InMemory	Store blocks in a variable in main memory
LASBlocks	Stores each block as a LAS file in a folder
MATBlocks	Stores each block as a MAT file in a folder

If you do not specify the OutputLocation argument, then the default value of Adapter is InMemory. If OutputLocation argument, then the value of Adapter is MATBlocks for non-point cloud output and LASBlocks for point cloud output.

DisplayWaitbar — Wait bar visualization
true or 1 (default) | false or 0

Wait bar visualization, specified as a logical 1 (true) or 0 (false). When set to true, the apply object function displays a wait bar operations with long run times. If you cancel the wait bar, the apply object function returns partial output, if available.

Data Types: logical

OutputLocation — Location of output folder
character vector | string scalar

Location of the output folder, specified as a string scalar or character vector.

If there is a single output, the apply object function writes it directly to this location.

For multiple outputs, the `apply` object function creates subfolders of the format `outputN`, where `N` is the sequential number of the output. If the input is an array, the `apply` object function derives the output name of each element from its `Source` property. If the input is in-memory, the `apply` function reports an error.

If the `UseParallel` property is `true`, `OutputLocation` must be a valid path on the client session. Use the `AlternateFileSystemRoots` property of the input to specify the required mapping for worker sessions. All outputs inherit this value.

UseParallel — Use parallel processing
`false` or `0` (default) | `true` or `1`

Use parallel processing, specified as a logical `0` (`false`) or `1` (`true`). The function first determines whether to use a new or an existing parallel pool. If no parallel pool is active, the function opens a new pool based on the default parallel settings. All adapters specified by the `Adapter` property must support parallel processing. You must specify a valid `OutputLocation`.

This argument requires Parallel Computing Toolbox™.

Data Types: `logical`

Output Arguments

bres — New blocked point cloud
`blockedPointCloud` object

New blocked point cloud, returned as a `blockedPointCloud` object.

Version History
Introduced in R2022a

See Also
`blockedPointCloud` | `selectBlockLocations`

blocksub2roi

Convert block subscripts to ROI limits

Syntax

```
roi = blocksub2roi(bpc,blocksub)
```

Description

`roi = blocksub2roi(bpc,blocksub)` converts the subscripts of a block to the ROI limits of that block in the `blockedPointCloud` object `bpc`.

Examples

Get ROI from Block Subscripts

Create a blocked point cloud from a LAZ file.

```
pcfile = fullfile(toolboxdir("lidar"),"lidardata", ...
  "las","aerialLidarData.laz");
bpc = blockedPointCloud(pcfile,[50 50]);
```

Convert the subscripts of a block to the ROI limits of that block.

```
roi = blocksub2roi(bpc,[2 2 1]);
```

Display the ROI limits of the block.

```
disp(roi)

   1.0e+06 *

    0.4298    0.4298    3.6799    3.6799    0.0001    0.0001
```

Input Arguments

bpc — Blocked point cloud
blockedPointCloud object

Blocked point cloud, specified as a `blockedPointCloud` object.

blocksub — Block subscripts
three-element row vector of positive integers

Block subscripts, specified as a three-element row vector of positive integers.

Example: [2 3 1]

Output Arguments

`roi` — ROI limits of the block
six-element row vector

ROI limits of the block, returned as a six-element row vector of form [*xmin xmax ymin ymax zmin zmax*], defining the range of the block.

Version History
Introduced in R2022a

See Also
`blockedPointCoud`

roi2blocksub

Convert ROI to block subscripts

Syntax

```
blocksubs = roi2blocksub(bpc,roi)
```

Description

`blocksubs = roi2blocksub(bpc,roi)` converts the specified ROI `roi` to the block subscripts `blocksubs` of the blocks of the blocked point cloud data `bpc` contained within the ROI.

Examples

### Get Block Subscripts from ROI

Create a blocked point cloud from a LAZ file.

```
pcfile = fullfile(toolboxdir("lidar"),"lidardata", ...
    "las","aerialLidarData.laz");
bpc = blockedPointCloud(pcfile,[50 50]);
```

Convert ROI limits to the block subscripts of the block contained within the ROI.

```
blocksubs = roi2blocksub(bpc,[429745.02 429775.02 ...
            3679830.75 3679860.75 72.79 125.82]);
```

Display the block subscripts.

```
disp(blocksubs)

    1    1    1
```

Input Arguments

bpc — Blocked point cloud
`blockedPointCloud` object

Blocked point cloud, specified as a `blockedPointCloud` object.

roi — ROI limits of the block
six-element row vector

ROI limits of the block, specified as a six-element row vector of form [*xmin xmax ymin ymax zmin zmax*], defining the range of the block.

Output Arguments

blocksubs — Subscripts of blocks
K-by-3 integer-valued matrix

Subscripts of the blocks, returned as a *K*-by-3 integer-valued matrix. *K* is the number of blocks in the specified ROI. The elements of each row correspond to the *X*-, *Y*- and *Z*-axes respectively.

Version History
Introduced in R2022a

See Also
blockedPointCoud

write

Write blocked point cloud data to new destination

Syntax

```
write(bpc,destination)
write(bpc,destination,Name=Value)
```

Description

write(bpc,destination) writes the blocked point cloud data bpc to the location specified by destination.

write(bpc,destination,Name=Value) specifies additional options for writing the blocked point cloud data using name-value arguments.

Examples

Downsample Point Cloud Data and Write

Create a full file path for a LAZ file that contains aerial lidar data.

```
pcfile = fullfile(toolboxdir("lidar"),"lidardata", ...
                "las","aerialLidarData.laz");
```

Create pcdownsample function handle.

```
fun = @(block)pcdownsample(block.Data,random=0.1);
```

Create blockedPointCloud object using the LAZ file.

```
bpc = blockedPointCloud(pcfile,[50 50]);
```

Process each block of the aerial point cloud data using the specified pcdownsample function handle.

```
pcdown = apply(bpc,fun);
```

Write the downsampled data to a specified destination.

```
write(pcdown,"aerial")
```

Create a new blockedPointCloud from the output and display its properties.

```
bpc2 = blockedPointCloud("aerial/aerial/");
disp(bpc2)

  blockedPointCloud with properties:

   Read-only properties.
            Source: "C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\25\tp65db348f\lidar-ex50165705\aeria
            Adapter: [1x1 lidar.blocked.LASBlocks]
        SizeInBlocks: [9 6 1]
```

```
         XLimits: [4.2975e+05 4.3015e+05]
         YLimits: [3.6798e+06 3.6801e+06]
         ZLimits: [79.4100 124.6200]
   ClassUnderlying: "pointCloud"

  Settable properties
        BlockSize: [50 50 53.0300]
```

Input Arguments

bpc — Blocked point cloud
blockedPointCloud object

Blocked point cloud, specified as a blockedPointCloud object.

destination — Location to write data
character vector | string scalar

Location to write data, specified as a character vector or string scalar. If you do not specify a full file or folder path, this function creates a folder specified by destination in the present working directory and writes the data in the created folder.

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: write(bpc,destination,DisplayWaitbar=true) displays a wait bar for operations with long run times.

Adapter — Adapter used for writing blocked point cloud data
adapter object

Adapter used for writing blocked point cloud data, specified as an adapter object.

This table lists the adapters included with the toolbox.

Adapter	Description
LAS	Store blocks in a single LAS file
LASBlocks	Stores each block as a LAS file in a folder
MATBlocks	Stores each block as a MAT file in a folder

DisplayWaitbar — Wait bar visualization
true or 1 (default) | false or 0

Wait bar visualization, specified as a logical 1 (true) or 0 (false). When set to true, the apply object function displays a wait bar operations with long run times. If you cancel the wait bar, the apply object function returns partial output, if available.

Data Types: logical

Version History
Introduced in R2022a

See Also
blockedPointCoud | `images.blocked.Adapter`

getRegion

Read arbitrary region of blocked point cloud

Syntax

```
ptCloud = getRegion(bpc,roi)
[ptCloud,pointAttributes] = getRegion( ___ )
```

Description

`ptCloud = getRegion(bpc,roi)` returns all points in the blocked point cloud `bpc` in specified region `roi`.

`[ptCloud,pointAttributes] = getRegion( ___ )` returns additional point attribute information `pointAttributes`, using all input arguments from the previous syntax, if the input is a LAZ or LAS file.

Examples

Read and Plot Sub-region of Blocked Point Cloud

Create a blocked point cloud from a LAZ file.

```
pcfile = fullfile(toolboxdir("lidar"),"lidardata", ...
  "las","aerialLidarData.laz");
bpc = blockedPointCloud(pcfile,[50 50]);
```

Specify a region in the point cloud and retrieve the data.

```
pcRegion = getRegion(bpc,[429745.02 429775.02 ...
              3679830.75 3679860.75 72.79 125.82]);
```

Plot the point cloud data from the specified region.

```
pcshow(pcRegion)
```
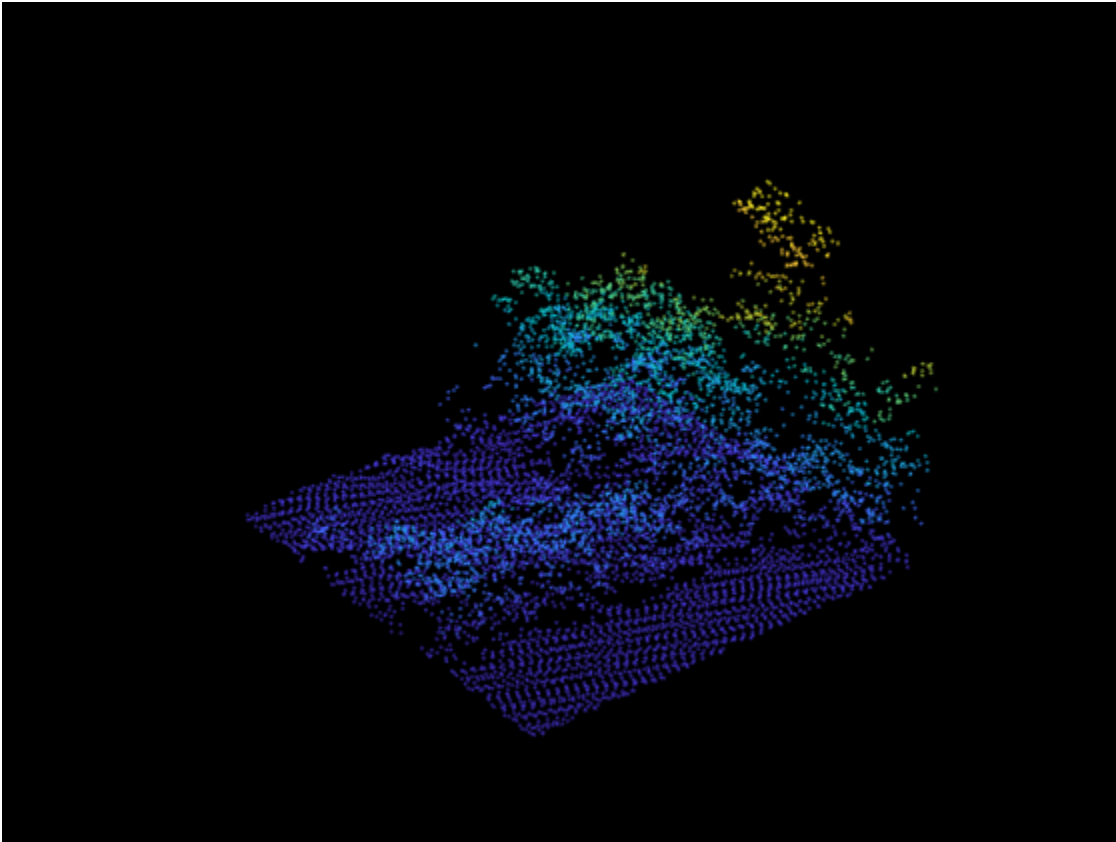
Input Arguments

bpc — Blocked point cloud
blockedPointCloud object

Blocked point cloud, specified as a blockedPointCloud object.

roi — ROI limits of the block
six-element row vector

ROI limits of the block, specified as a six-element row vector of form [*xmin xmax ymin ymax zmin zmax*], defining the range of the block.

Output Arguments

ptCloud — Point cloud data from specified region
pointCloud object

Point cloud data from specified region, returned as a pointCloud object.

pointAttributes — Point attributes
lidarPointAttributes object

Point attributes, returned as a lidarPointAttributes object.

Version History
Introduced in R2022a

See Also
blockedPointCoud | getBlock

getBlock

Read specific block of blocked point cloud

Syntax

```
ptCloud = getBlock(bpc,blocksub)
[ptCloud,pointAttributes] = getBlock( ___ )
```

Description

`ptCloud = getBlock(bpc,blocksub)` returns the block specified by the subscript location `blocksub`.

`[ptCloud,pointAttributes] = getBlock( ___ )` returns additional point attribute information `pointAttributes` using the input arguments from the previous syntax, if the input is a LAZ or LAS file.

Examples

Read Block from Blocked Point Cloud

Create a blocked point cloud from a LAZ file.

```
pcfile = fullfile(toolboxdir("lidar"),"lidardata", ...
  "las","aerialLidarData.laz");
bpc = blockedPointCloud(pcfile,[50 50]);
```

Display the size, in blocks, of the blocked point cloud.

```
disp(bpc.SizeInBlocks)
```

```
     9     6     1
```

Read a specific block of the point cloud.

```
block = getBlock(bpc,[2,2,1]);
```

Plot the point cloud data of the specified block.

```
pcshow(block)
```
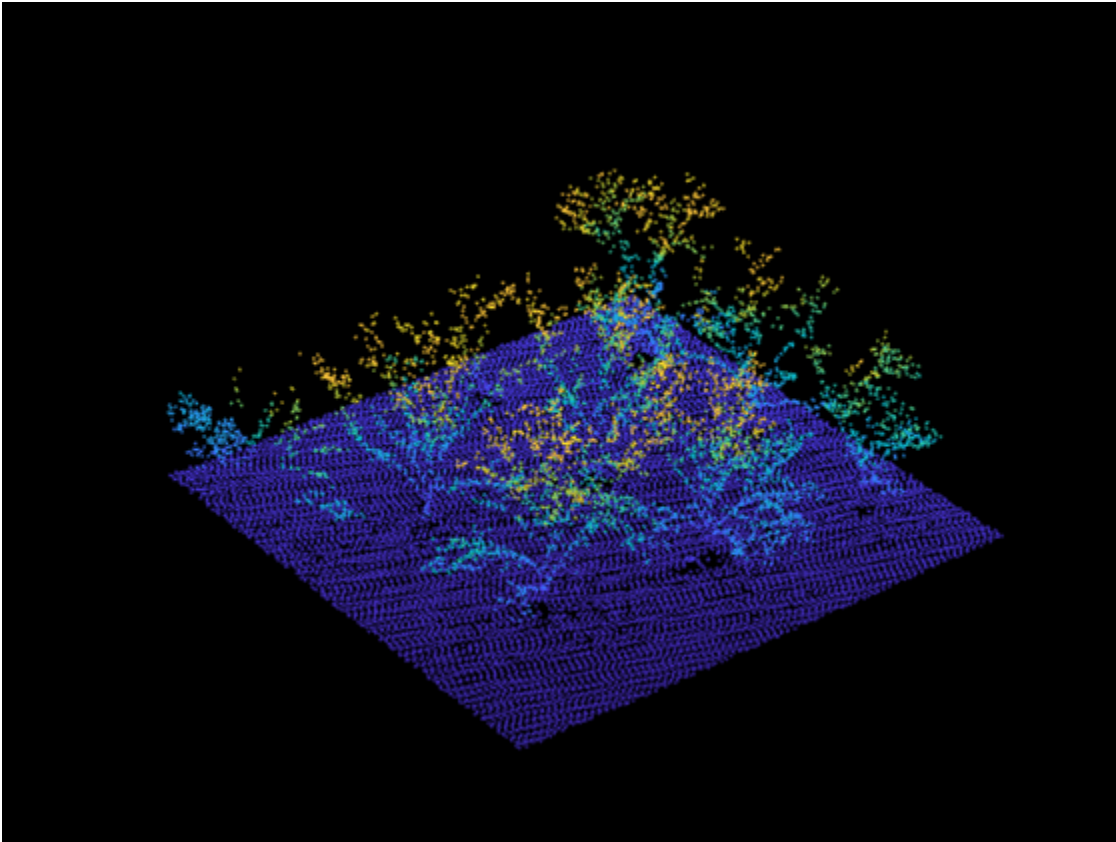
Input Arguments

bpc — Blocked point cloud
blockedPointCloud object

Blocked point cloud, specified as a blockedPointCloud object.

blocksub — Block subscripts
three-element integer-valued row vector

Block subscripts, specified as a three-element integer-valued row vector. Valid values for each element range from 1 to the value of the corresponding element in the SizeInBlocks property of bpc.

Example: [3 2 1]

Output Arguments

ptCloud — Block of point cloud data
pointCloud object

Block of point cloud data, returned as a pointCloud object.

pointAttributes — Point attributes
lidarPointAttributes object

Point attributes, returned as a `lidarPointAttributes` object.

Version History
Introduced in R2022a

See Also
`blockedPointCoud` | `getRegion` | `lidarPointAttributes`

gather

Collect blocks of blocked point cloud into workspace

Syntax

```
ptCloud = gather(bpc)
```

Description

`ptCloud = gather(bpc)` collects all the blocks of the `blockedPointCloud` object `bpc`, assembles them, and returns a single `pointCloud` object, `ptCloud`.

Examples

Collect Blocks from Blocked Point Cloud

Create a blocked point cloud from a LAZ file.

```
pcfile = fullfile(toolboxdir("lidar"),"lidardata", ...
  "las","aerialLidarData.laz");
bpc = blockedPointCloud(pcfile,[50 50]);
```

Collect the blocks of the blocked point cloud, and assemble them into a single point cloud in the workspace.

```
ptCloud = gather(bpc);
```

Plot the assembled point cloud.

```
pcshow(ptCloud.Location)
```
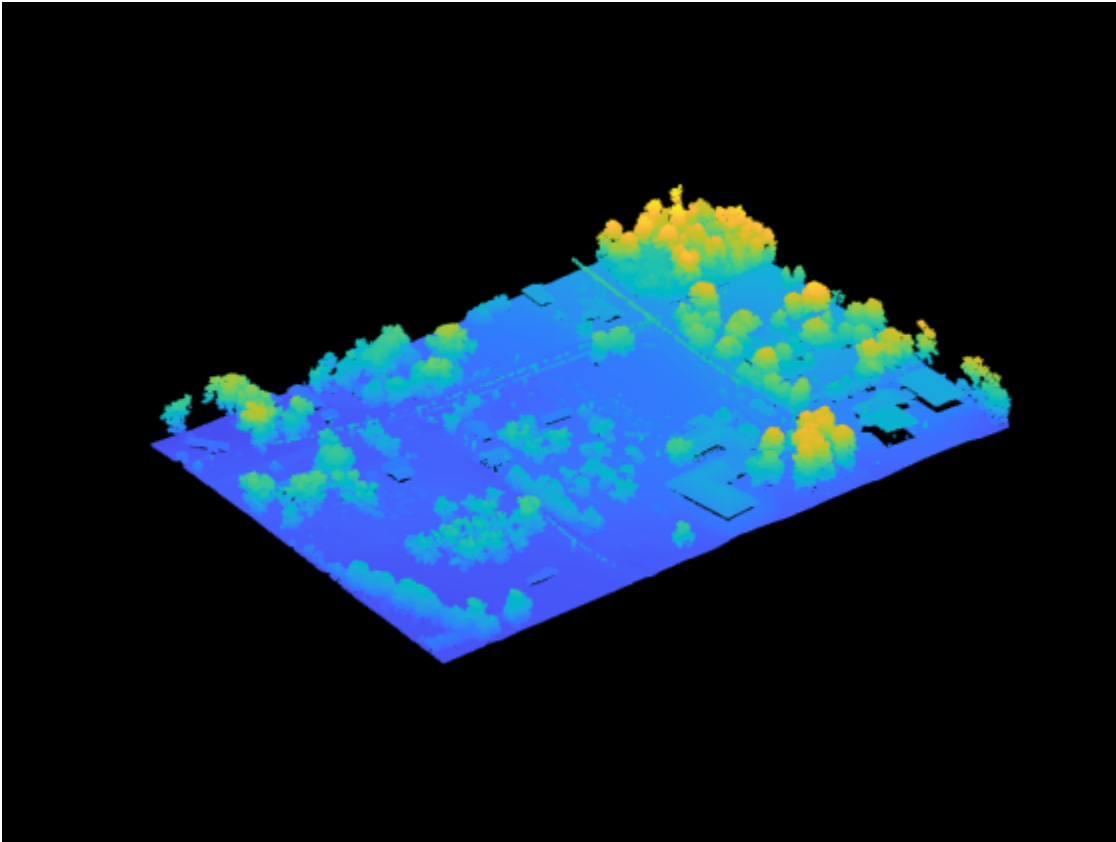
Input Arguments

bpc — Blocked point cloud
blockedPointCloud object

Blocked point cloud, specified as a blockedPointCloud object.

Output Arguments

ptCloud — Assembled point cloud
pointCloud object

Assembled point cloud, returned as a pointCloud object.

Version History
Introduced in R2022a

See Also
blockedPointCoud

blockedPointCloudDatastore

Datastore for use with blocks from `blockedPointCloud` objects

Description

A `blockedPointCloudDatastore` object manages a collection of point cloud blocks that belong to one or more `blockedPointCloud` objects.

Creation

Syntax

```
bpcds = blockedPointCloudDatastore(bpcs)
bpcds = blockedPointCloudDatastore(bpcs,Name=Value)
```

Description

`bpcds = blockedPointCloudDatastore(bpcs)` creates a `blockedPointCloudDatastore` object that manages a collection of point cloud blocks of one or more `blockedPointCloud` objects, `bpcs`.

The `BlockSize` property of the first element in `bpcs` is the default datastore block size.

`bpcds = blockedPointCloudDatastore(bpcs,Name=Value)` specifies the `BlockSize`, `BlockLocationSet` and `ReadSize` properties of `blockedPointCloudDatastore` object by using one or more name-value arguments.

Input Arguments

bpcs — Blocked point clouds
array of `blockedPointCloud` objects

Blocked point clouds, specified as an array of `blockedPointCloud` objects.

Properties

BlockLocationSet — Blocks to include in datastore
`blockLocationSet` object

Blocks to include in the datastore, specified as a `blockLocationSet` object. The object specifies which blocks to include from the blocked point cloud `bpcs`. You can repeat or omit individual blocks. To obtain the default value, `blockedPointCloudDatastore` calls the `selectBlockLocations` function.

You cannot change the `BlockLocationSet` property after creating the `blockedPointCloudDatastore`.

BlockSize — Block size
three-element numeric row vector

Block size, specified as a three-element numeric row vector. The elements specify the size of each block in the *X*-, *Y*- and *Z*- dimensions, respectively.

The default value is the block size of the first `blockedPointCloud` in `bpcs`.

You cannot change the `BlockSize` property after creating the `blockedPointCloudDatastore`.

Example: `BlockSize=[50 30 40]`

PointClouds — Blocked point clouds
array of `blockedPointCloud` objects

Blocked point clouds that supply blocks for the `blockedPointCloudDatastore`, specified as an array of `blockedPointCloud` objects. All elements of `pointClouds` must have the same number of dimensions and be of the same type.

You cannot change the `PointClouds` property after creating the `blockedPointCloudDatastore`.

ReadSize — Number of blocks to return in each call to `read` function
1 (default) | positive integer

Number of blocks to return in each call to the `read` function, specified as a positive integer. Each call to the `read` function reads at most `ReadSize` blocks.

TotalNumBlocks — Total number of blocks available
numeric scalar

This property is read-only.

Total number of blocks available, specified as a numeric scalar.

Object Functions

combine	Combine data from multiple datastores
hasdata	Returns true if more data is available in blockedPointCloudDatastore
numpartitions	Number of datastore partitions
partition	Partition blockedPointCloudDatastore
preview	Preview subset of data in datastore
read	Read data and metadata from blockedPointCloudDatastore
readall	Read all data from blockedPointCloudDatastore
reset	Reset datastore to initial state
shuffle	Shuffle data in datastore
subset	Create subset of datastore or FileSet
transform	Transform datastore

Examples

Create Point Cloud Datastore

Create a blocked point cloud from a LAZ file, specifying the block size.

```
pcfile = fullfile(toolboxdir("lidar"),"lidardata", ...
                "las","aerialLidarData.laz");
bpc = blockedPointCloud(pcfile,[50 50]);
```

Create a blocked point cloud datastore that contains the blocked point cloud.

```
bpcds = blockedPointCloudDatastore(bpc);
```

Read four blocks from the datastore.

```
bpcds.ReadSize = 4;
blocks = read(bpcds);
```

Display the details of the four blocks.

```
disp(blocks)

    {1x1 pointCloud}
    {1x1 pointCloud}
    {1x1 pointCloud}
    {1x1 pointCloud}
```

Create blockedPointCloudDatastore from Multiple Files

Create a FileSet object containing multiple LAS files.

```
fs = matlab.io.datastore.FileSet(...
    fullfile(toolboxdir("lidar"),"lidardata", ...
                "las"),"FileExtensions",".las");
```

Create an array of blockedPointCloud objects from the file set, and specify an adapter. Specifying an adapter means the blockedPointCloud function does not have to inspect each file to pick a suitable adapter.

```
readAdapter = lidar.blocked.LAS();
bpcs = blockedPointCloud(fs,[100 100],Adapter=readAdapter);
```

Create a blocked point cloud datastore from the blockedPointCloud array.

```
bpcds = blockedPointCloudDatastore(bpcs);
```

Read all data from the blockedPointCloudDatastore.

```
blocks = readall(bpcds);
```

Version History
Introduced in R2022a

See Also
blockedPointCloud | blockLocationSet | selectBlockLocations

read

Read data and metadata from `blockedPointCloudDatastore`

Syntax

```
b = read(bpcds)
[b,info] = read(bpcds)
```

Description

`b = read(bpcds)` returns the data extracted from the `blockedPointCloudDatastore` object `bpcds`.

`[b,info] = read(bpcds)` also returns `info`, a structure containing information about where in the `blockedPointCloudDatastore` the data is extracted from.

Examples

Read Data and Metadata from blockedPointCloudDatastore

Create a blocked point cloud.

```
pcfile = fullfile(toolboxdir("lidar"),"lidardata", ...
                      "las","aerialLidarData.laz");
bpc = blockedPointCloud(pcfile,[300 300]);
```

Create a `blockedPointCloudDatastore` from the blocked point cloud.

```
bpcds = blockedPointCloudDatastore(bpc);
```

Read data and metadata from the `blockedPointCloudDatastore`. Display the metadata.

```
while hasdata(bpcds)
    [data,info] = read(bpcds);
    disp(info)
end
```

```
              ROI: [4.2975e+05 4.3005e+05 3.6798e+06 3.6801e+06 ... ]
         PCNumber: 1
  PointAttributes: [1x1 lidarPointAttributes]
        BlockSize: [300 300 53.0300]

              ROI: [4.3005e+05 4.3035e+05 3.6798e+06 3.6801e+06 ... ]
         PCNumber: 1
  PointAttributes: [1x1 lidarPointAttributes]
        BlockSize: [300 300 53.0300]
```

Input Arguments

bpcds — Blocked point cloud datastore
blockedPointCloudDatastore object

Blocked point cloud datastore, specified as a blockedPointCloudDatastore object.

Output Arguments

b — Data from blockedPointCloudDatastore
cell array

Data from the blockedPointCloudDatastore, returned as a cell array of block data. The length of the cell array is equal to the value of the ReadSize property of the blockedPointCloudDatastore object.

info — Metadata from blockedPointCloudDatastore
structure

Metadata from the blockedPointCloudDatastore, returned as a structure with these fields. If the value of the ReadSize property of the blockedPointCloudDatastore object is greater than 1, these fields are arrays.

Field	Description
ROI	ROI of the block, specified as a six-element numeric row vector in the order [*xmin xmax ymin ymax zmin zmax*].
PCNumber	Index into the bpcds.PointClouds array corresponding to the blockedPointCloud from which this block is read.
PointAttributes	Attributes for each point, specified as a lidarPointAttributes object.
BlockSize	Value of the BlockSize parameter.

Version History
Introduced in R2022a

See Also
blockedPointCloud | blockedPointCloudDatastore

readall

Read all data from `blockedPointCloudDatastore`

Syntax

```
b = readall(bpcds)
```

Description

`b = readall(bpcds)` read all the data from the `blockedPointCloudDatastore` object `bpcds`.

Examples

Read All Blocks from `blockedPointCloudDatastore`

Create a blocked point cloud.

```
pcfile = fullfile(toolboxdir("lidar"),"lidardata", ...
                        "las","aerialLidarData.laz");
bpc = blockedPointCloud(pcfile,[200 200]);
```

Create a `blockedPointCloudDatastore` from the blocked point cloud.

```
bpcds = blockedPointCloudDatastore(bpc);
```

Read all the blocks from the `blockedPointCloudDatastore`. The `readall` object function returns a cell array containing the six blocks.

```
b = readall(bpcds)
```

```
b=6×1 cell array
    {1x1 pointCloud}
    {1x1 pointCloud}
    {1x1 pointCloud}
    {1x1 pointCloud}
    {1x1 pointCloud}
    {1x1 pointCloud}
```

Input Arguments

bpcds — Blocked point cloud datastore
`blockedPointCloudDatastore` object

Blocked point cloud datastore, specified as a `blockedPointCloudDatastore` object.

Output Arguments

b — Data from `blockedPointCloudDatastore`
cell array

Data from the `blockedPointCloudDatastore`, returned as a cell array of block data. The length of the cell array is equal to the value of the `ReadSize` property of the `blockedPointCloudDatastore` object. Each element of `b` contains the data for a single block of `bpcds`. The `readall` function returns the data from each individual read operation such that the data can be concatenated vertically.

The data type of this output is the same as the data type of the output of the `read` function.

Version History
Introduced in R2022a

See Also
`blockedPointCloud` | `blockedPointCloudDatastore`

hasdata

Returns `true` if more data is available in `blockedPointCloudDatastore`

Syntax

```
tf = hasdata(bpcds)
```

Description

`tf = hasdata(bpcds)` returns a logical scalar, `true` or `false`, indicating the availability of data in the `blockedPointCloudDatastore` object `bpcds`. Use `hasdata` in conjunction with the `read` function to read all the data within the datastore. Call `hasdata` before calling `read`.

Examples

Read Until All Data in `blockedPointCloudDatastore` Has Been Read

Create a blocked point cloud.

```
pcfile = fullfile(toolboxdir("lidar"),"lidardata", ...
                     "las","aerialLidarData.laz");
bpc = blockedPointCloud(pcfile,[300 300]);
```

Create a `blockedPointCloudDatastore` from the blocked point cloud.

```
bpcds = blockedPointCloudDatastore(bpc);
```

Read data and metadata from the `blockedPointCloudDatastore`. Display the metadata.

```
while hasdata(bpcds)
      [data,info] = read(bpcds);
      disp(info)
end
```

```
            ROI: [4.2975e+05 4.3005e+05 3.6798e+06 3.6801e+06 ... ]
        PCNumber: 1
  PointAttributes: [1x1 lidarPointAttributes]
       BlockSize: [300 300 53.0300]

            ROI: [4.3005e+05 4.3035e+05 3.6798e+06 3.6801e+06 ... ]
        PCNumber: 1
  PointAttributes: [1x1 lidarPointAttributes]
       BlockSize: [300 300 53.0300]
```

Input Arguments

bpcds — Blocked point cloud datastore
blockedPointCloudDatastore object

Blocked point cloud datastore, specified as a `blockedPointCloudDatastore` object.

Output Arguments

tf — Data availability
true or 1 | false or 0

Data availability, returned as a logical 1 (true) or 0 (false).

Version History
Introduced in R2022a

See Also
blockedPointCloudDatastore

partition

Partition `blockedPointCloudDatastore`

Syntax

```
subbpcds = partition(bpcds,n,index)
```

Description

`subbpcds = partition(bpcds,n,index)` partitions the blocked point cloud datastore `bpcds` into the specified number of parts `n`, and returns the partition corresponding to the specified index `index`.

Examples

Partition blockedPointCloudDatastore and Read First Partition

Create a blocked point cloud.

```
pcfile = fullfile(toolboxdir("lidar"),"lidardata", ...
                        "las","aerialLidarData.laz");
bpc = blockedPointCloud(pcfile,[300 300]);
```

Create a `blockedPointCloudDatastore` from the blocked point cloud.

```
bpcds = blockedPointCloudDatastore(bpc);
```

Partition the blocked point cloud datastore into two partitions, and create a new `blockedPointCloudDatastore` object from the data in the first partition.

```
bpcdsp1 = partition(bpcds,2,1);
```

Read data and metadata from the first partition. Display the metadata.

```
while hasdata(bpcdsp1)
      [data,info] = read(bpcdsp1);
      disp(info)
end
```

```
            ROI: [4.2975e+05 4.3005e+05 3.6798e+06 3.6801e+06 ... ]
        PCNumber: 1
    PointAttributes: [1x1 lidarPointAttributes]
        BlockSize: [300 300 53.0300]
```

Input Arguments

bpcds — Blocked point cloud datastore
`blockedPointCloudDatastore` object

Blocked point cloud datastore, specified as a `blockedPointCloudDatastore` object.

n — Number of partitions
numeric scalar

Number of partitions, specified as a numeric scalar. To estimate a reasonable value for N, use the `numpartitions` function.

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

index — Partition to read
numeric scalar

Partition to read, specified as a numeric scalar.

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

Output Arguments

subbpcds — Partitioned subset of datastore
`blockedPointCloudDatastore` object

Partitioned subset of the datastore, returned as a `blockedPointCloudDatastore` object.

Version History
Introduced in R2022a

See Also
`blockedPointCloud` | `blockedPointCloudDatastore`

lidar.blocked.Adapter class

Package: `lidar.blocked`

Adapter interface for `blockedPointCloud` objects

Description

The `lidar.blocked.Adapter` class specifies the interface for block-based reading and writing of data. Classes that inherit from this interface can be used with `blockedPointCloud` objects, enabling block-based stream processing of data.

To implement this class, you must:

1   Inherit from the `lidar.blocked.Adapter` class. Your class definition must have this format, where *MyAdapter* is the name of your custom adapter class.

    ```
    classdef MyAdapter < lidar.blocked.Adapter
        ...
    end
    ```

2   Define the three required methods for reading point cloud data from disk: `openToRead`, `getInfo`, and `getRegion`.

3   Optionally, define methods that enable additional reading and writing capabilities.

4   Optionally, for single-file destinations, define an `Extension` property that specifies the file extension to use when automatically creating a destination location. The property value must be a string, such as `"las"`. For adapters that store data in a folder, do not add this property, or specify the value of the property as empty (`[]`).

The `lidar.blocked.Adapter` class is a `handle` class.

Class Attributes

Abstract	true

For information on class attributes, see "Class Attributes".

Methods

Public Methods

Capability	Methods to Implement
Read data (Required)	`openToRead` — Open source for reading
	`getInfo` — Gather information about the source
	`getRegion` — Get specified region
Write data (Optional)	`openToWrite` — Create and open destination for writing
	`setRegion` — Set specified region

Capability	Methods to Implement
Perform clean up tasks (Optional)	`close` — Perform clean up tasks such as closing file handles
Enable parallel block processing (Optional)	`openInParallelToAppend` — Use the adapter in parallel mode with the `apply` object function

Tips

The toolbox includes several built-in adapters that subclass from the `Adapter` class. All these adapters support both read and write operations.

Adapter	Description
`LAS`	Store blocks in a single LAS file
`InMemory`	Store blocks in a variable in main memory
`LASBlocks`	Stores each block as a LAS file in a folder
`MATBlocks`	Stores each block as a MAT file in a folder

Version History
Introduced in R2022a

See Also
`blockedPointCloud`

lidar.blocked.InMemory

Read and write blocked point cloud data as workspace variable

Description

The `InMemory` object is an adapter that reads and writes single-resolution blocked point cloud data as a variable in the workspace.

The table lists the support that the `InMemory` object has for various `blockedPointCloud` capabilities.

Capability	Support
Data types	`struct` and `pointCloud` object
Process blocks in parallel using the `apply` function	No

Creation

Syntax

adapter = lidar.blocked.InMemory

Description

adapter = lidar.blocked.InMemory creates an `InMemory` object that reads and writes blocked point cloud data to a variable in the workspace.

Version History
Introduced in R2022a

See Also
blockedPointCloud | LASBlocks | MATBlocks | LAS

lidar.blocked.LAS

Read and write blocked point cloud data as single LAS file

Description

The LAS object is an adapter that reads and writes a point cloud as a single block in a single LAS file.

The table lists the support that the LAS object has for various `blockedPointCloud` capabilities.

Capability	Support
Data types	`struct` and `pointCloud` object
Process blocks in parallel using the `apply` function	No

Creation

Syntax

```
adapter = lidar.blocked.LAS
```

Description

`adapter = lidar.blocked.LAS` creates a LAS object that reads and writes blocked point cloud data as a single block in a single LAS file.

Properties

Extension — Preferred file extension
"las" (default) | "laz"

Preferred file extension, specified as `"las"` or `"laz"`.

Data Types: `char` | `string`

LasVersion — LAS version
"1.2" (default) | "1.0" | "1.1" | "1.3" | "1.4"

LAS version, specified as `"1.0"`, `"1.1"`, `"1.2"`, `"1.3"`, or `"1.4"`.

Data Types: `string` | `char`

Version History
Introduced in R2022a

See Also

blockedPointCloud | InMemory | LASBlocks | MATBlocks

lidar.blocked.LASBlocks

Read and write each block of blocked point cloud data as LAS file

Description

The `LASBlocks` object is an adapter that writes blocked point cloud data in LAS format.

When writing to disk, the object creates an individual LAS file for each block and saves the point cloud files in a folder. The object also creates and saves a MAT file with information about the blocked point cloud.

The table lists the support that the `LASBlocks` object has for various `blockedPointCloud` capabilities.

Capability	Support
Data types	`struct` and `pointCloud` object
Process blocks in parallel using the `apply` function	Yes

Creation

Syntax

```
adapter = lidar.blocked.LASBlocks
```

Description

`adapter = lidar.blocked.LASBlocks` creates a `LASBlocks` object that reads and writes blocked point cloud data as LAS files, with one LAS file for each block.

Properties

BlockFormat — Point cloud file format
"las" (default) | "laz"

Point cloud file format, specified as `"las"` or `"laz"`.

Data Types: `char` | `string`

LasVersion — LAS version
"1.2" (default) | "1.0" | "1.1" | "1.3" | "1.4"

LAS version, specified as `"1.0"`, `"1.1"`, `"1.2"`, `"1.3"`, or `"1.4"`.

Data Types: `string` | `char`

Version History
Introduced in R2022a

See Also
blockedPointCloud | MATBlocks | InMemory | LAS

lidar.blocked.MATBlocks

Read and write blocks of blocked point cloud data as MAT files

Description

The `MATBlocks` object is an adapter that reads and writes blocked point cloud data as MAT files, with one MAT file for each block.

The object saves the point cloud data files in a folder.

The table lists the support that the `MATBlocks` object has for various `blockedPointCloud` capabilities.

Capability	Support
Data types	`struct` and `pointCloud` object
Process blocks in parallel using the `apply` function	Yes

Creation

Syntax

adapter = lidar.blocked.MATBlocks

Description

adapter = lidar.blocked.MATBlocks creates a `MATBlocks` object that reads and writes blocked point cloud data as MAT files, with one MAT file for each block.

Version History
Introduced in R2022a

See Also
blockedPointCloud | LAS | InMemory | LASBlocks

close

Class: lidar.blocked.Adapter
Package: lidar.blocked

Close adapter

Syntax

```
close(obj)
```

Description

close(obj) closes and releases resources acquired from using the openToRead, openToWrite, and openInParallelToAppend methods. Use this method to flush data, close file handles, and perform other clean up actions.

Input Arguments

obj — Adapter object
lidar.blocked.Adapter object

Adapter object, specified as an instance of an adapter class that is subclassed from the lidar.blocked.Adapter class.

Version History
Introduced in R2022a

See Also
lidar.blocked.Adapter

getInfo

Class: `lidar.blocked.Adapter`
Package: `lidar.blocked`

Gather information about source

Syntax

```
info = getInfo(obj)
```

Description

`info = getInfo(obj)` gathers and returns `info`, a structure containing information about the source.

Input Arguments

obj — Adapter object
`lidar.blocked.Adapter` object

Adapter object, specified as an instance of an adapter class that is subclassed from the `lidar.blocked.Adapter` class.

Output Arguments

info — Information about source
structure

Information about the source, returned as a structure with additional metadata of the input point cloud. The structure contains these fields:

- `Size` — Size of the point cloud, returned as a three-element numeric vector.
- `Datatype` — Data type of the point cloud, returned as a `pointCloud` object or structure.
- `BlockSize` — Block size of the point cloud, returned as a three-element numeric vector.
- `MinLimits` — Minimum coordinate limits, returned as a three-element numeric vector of the form [*xmin ymin zmin*].
- `MaxLimits` — Maximum coordinate limits, returned as a three-element numeric vector of the form [*xmax ymax zmax*].

Version History
Introduced in R2022a

See Also
`lidar.blocked.Adapter`

getRegion

Class: `lidar.blocked.Adapter`
Package: `lidar.blocked`

Read arbitrary region of blocked point cloud

Syntax

`[ptCloud,pointAttributes] = getRegion(obj,roi)`

Description

`[ptCloud,pointAttributes] = getRegion(obj,roi)` returns all points of the point cloud `ptCloud` in the region specified by `roi`. The method also returns additional point attribute information, `pointAttributes`, if the input is a LAZ or LAS file.

Input Arguments

obj — Adapter object
`lidar.blocked.Adapter` object

Adapter object, specified as an instance of an adapter class that is subclassed from the `lidar.blocked.Adapter` class.

roi — ROI limits of block
six-element row vector

ROI limits of the block, specified as a six-element row vector in the order [*xmin xmax ymin ymax zmin zmax*], defining the range of the block.

Output Arguments

ptCloud — Point cloud
`pointCloud` object

Point cloud, returned as a `pointCloud` object.

pointAttributes — Point attributes
`lidarPointAttributes` object

Point attributes, returned as a `lidarPointAttributes` object.

Version History
Introduced in R2022a

See Also

`lidar.blocked.Adapter` | `setRegion` | `lidarPointAttributes` | `pointCloud`

openInParallelToAppend

Class: lidar.blocked.Adapter
Package: lidar.blocked

Open destination on parallel worker to append blocks

Syntax

openInParallelToAppend(obj,destination)

Description

openInParallelToAppend(obj,destination) opens the location specified by destination on a parallel worker in preparation for appending blocks.

Input Arguments

obj — Adapter object
lidar.blocked.Adapter object

Adapter object, specified as an instance of an adapter class that is subclassed from the lidar.blocked.Adapter class.

destination — Location
string scalar | character vector

Location, specified as a string scalar or character vector.

Data Types: char | string

Version History
Introduced in R2022a

See Also
lidar.blocked.Adapter

openToRead

Class: `lidar.blocked.Adapter`
Package: `lidar.blocked`

Open source for read access

Syntax

`openToRead(obj,source)`

Description

`openToRead(obj,source)` opens the specified location on disk `source` for read access. This method issues an error if the adapter does not support `source`.

Input Arguments

`obj` — Adapter object
`lidar.blocked.Adapter` object

Adapter object, specified as an instance of an adapter class that is subclassed from the `lidar.blocked.Adapter` class.

`source` — Location to read from
string scalar | character vector

Location to read from, specified as a string scalar or character vector.

Data Types: `char` | `string`

Version History
Introduced in R2022a

See Also
`lidar.blocked.Adapter` | `openToWrite`

openToWrite

Class: `lidar.blocked.Adapter`
Package: `lidar.blocked`

Create and open destination for writing

Syntax

`openToWrite(obj,destination,info)`

Description

`openToWrite(obj,destination,info)` opens the location specified by the `destination` for writing.

Use this method to prepare the destination for writing. For example, open a file handle, create a destination folder, or write a file header or metadata to the destination.

Input Arguments

`obj` — Adapter object
`lidar.blocked.Adapter` object

Adapter object, specified as an instance of an adapter class that is subclassed from the `lidar.blocked.Adapter` class.

`destination` — Location to write to
string scalar | character vector

Location to write to, specified as a string scalar or character vector.

Data Types: `char` | `string`

`info` — Information about source
structure

Information about the source, specified as a structure that contains additional metadata of the input point cloud. The structure can be empty.

Version History
Introduced in R2022a

See Also
`lidar.blocked.Adapter` | `openToRead`

setRegion

Class: `lidar.blocked.Adapter`
Package: `lidar.blocked`

Write specified region of blocked point cloud

Syntax

setRegion(obj,roi,data,info)

Description

setRegion(obj,roi,data,info) writes the `data` of a `pointCloud` object to the specified `roi`. You must specify additional information, `info`, to set the region, and to get information related to the point cloud. If no additional information is necessary, you must specify an empty array (`[]`).

Input Arguments

`obj` — Adapter object
`lidar.blocked.Adapter` object

Adapter object, specified as an instance of an adapter class that is subclassed from the `lidar.blocked.Adapter` class.

`roi` — ROI limits of block
six-element row vector

ROI limits of the block, specified as a six-element row vector in the order [*xmin xmax ymin ymax zmin zmax*], defining the range of the block.

`data` — Point cloud data
`pointCloud` object

Point cloud data, specified as a `pointCloud` object.

`info` — Information about source
array

Information about the source, specified as an array containing additional metadata of the input point cloud. It can also contain information necessary to set the region. If no additional information is necessary, you must specify it as an empty array.

Version History
Introduced in R2022a

See Also
lidar.blocked.Adapter | getRegion | lidarPointAttributes | pointCloud

lidar.labeler.AutomationAlgorithm class

Package: `lidar.labeler`

Interface for algorithm automation in ground truth labeling

Description

The `lidar.labeler.AutomationAlgorithm` class specifies the interface for defining custom automation algorithms to run in the **Lidar Labeler** app. Classes that inherit from the `AutomationAlgorithm` interface can be used with the automation workflow of the **Lidar Labeler** app to generate ground truth labels.

The `lidar.labeler.AutomationAlgorithm` class is a `handle` class.

Class Attributes

Abstract	true

For information on class attributes, see "Class Attributes".

Properties

The `AutomationAlgorithm` class predefines this set of properties.

GroundTruth — Ground truth data
groundTruthLidar object

Ground truth data, specified as a `groundTruthLidar`. This property holds all the labels in the **Lidar Labeler** app prior to automation.

Attributes:

GetAccess	public
SetAccess	private

SelectedLabelDefinitions — Selected label definitions
structure

Label definitions selected for automation in the app, specified as a structure. The app support selection of only one labeling definition per automation session. In the app, the selected label definition is highlighted in yellow in either the **ROI Labels** or **Scene Labels** pane on the left.

The structure contains these fields.

Field	Description
Type	`labelType` enumeration that contains the type of label definition. Valid label types are: • `labelType.Cuboid` • `labelType.Line` • `labelType.Scene` `lidarLabelType` object contains the label type for voxel labeled ROI. You define the label using `lidarLabelType.Voxel`.
Name	Character vector that contains the name of the label definition.
Attributes (optional)	Structure array that contains one structure for each attribute in the label definition. If the label definition does not contain attributes, then this property does not include the `Attributes` field. The first field of each attribute structure in this structure array contains the attribute name. The second field contains a structure of values that are associated with that name. If you are defining a `List` attribute, you must also define the list of values for that attribute. Values for `Numeric Value`, `String`, or `Logical` attributes are optional. Descriptions for the attributes are optional for all cases.
VoxelLabelID (optional)	Positive integer that contains the voxel label ID for the label definition. This `VoxelLabelID` field applies only for label definitions of type `lidarLabelType.Voxel`.

To view a sample `SelectedLabelDefinitions` structure that contains an attribute, enter this code at the MATLAB command prompt.

```
selectedLabelDefs.Type = labelType.Cuboid;
selectedLabelDefs.Name = 'Car';
selectedLabelDefs.Attributes = struct('distance', ...
    struct('DefaultValue',0,'Description','Sensor distance'))
```

To view a sample `SelectedLabelDefinitions` structure that contains voxel label definitions, enter this code at the MATLAB command prompt.

```
selectedLabelDefs.Type = lidarLabelType.Voxel;
selectedLabelDefs.Name = 'Tree';
selectedLabelDefs.Attributes = struct('distance', ...
    struct('DefaultValue',0,'Description','Sensor distance'))
```

Attributes:

```
GetAccess                                    public
SetAccess                                    private
```

ValidLabelDefinitions — Valid label definitions
structure array

Valid label definitions that the algorithm can automate, specified as a structure array. Each structure in the array corresponds to a valid label definition. To determine which label definitions are valid, the app uses the `checkLabelDefinition` method. This table describes the fields for each valid label definition structure.

Field	Description
Type	`labelType` enumeration that contains the type of label definition. Valid label types are: • `labelType.Cuboid` • `labelType.Line` • `labelType.Scene` `lidarLabelType` object contains the label type for voxel labeled ROI. You define the label using `lidarLabelType.Voxel`.
Name	Character vector that contains the name of the label definition.
Attributes (optional)	Structure array that contains one structure for each attribute in the label definition. If the label definition does not contain attributes, then this property does not include the `Attributes` field. The first field of each attribute structure in this structure array contains the attribute name. The second field contains a structure of values that are associated with that name. If you are defining a `List` attribute, you must also define the list of values for that attribute. Values for `Numeric Value`, `String`, or `Logical` attributes are optional. Descriptions for the attributes are optional for all cases.
VoxelLabelID (optional)	Positive integer that contains the voxel label ID for the label definition. This `VoxelLabelID` field applies only for label definitions of type `lidarLabelType.Voxel`.

To view a sample `ValidLabelDefinitions` structure that contains an attribute, enter this code at the MATLAB command prompt.

```
validLabelDefs(1).Type = labelType.Cuboid;
validLabelDefs(1).Name = 'Car';
validLabelDefs(2).Type = labelType.Line;
validLabelDefs(2).Name = 'LaneMarker';
validLabelDefs(3).Type = lidarLabelType.Voxel;
validLabelDefs(3).Name = 'Tree';
```

Attributes:

```
GetAccess                                    public
SetAccess                                    private
```

Clients of the `AutomationAlgorithm` class are required to define this set of properties. These properties set up the name, description, and user instructions for your automated algorithm.

Name — Automation algorithm name
character vector

Automation algorithm name, specified as a character vector.

Attributes:

GetAccess	public
Abstract	true
Constant	true
NonCopyable	true

Description — Automation algorithm description
character vector

Algorithm description, specified as a character vector.

Attributes:

GetAccess	public
Abstract	true
Constant	true
NonCopyable	true

UserDirections — Algorithm directions displayed in app
cell array

Algorithm directions displayed in app, specified as a cell array. `UserDirections` are specified as a `cellstr`, with each string representing a separate direction. Use the `checkSetup` method to verify that the directions have been adhered to.

Attributes:

GetAccess	public
Abstract	true
Constant	true
NonCopyable	true

Methods

Public Methods

Clients of an `AutomationAlgorithm` implement these user-defined functions to define execution of the algorithm. For more information on creating your own automation algorithm, see "Create Automation Algorithm for Labeling".

checkLabelDefinition	Validate label definition
checkSignalType	Validate signal type
checkSetup	Set up validation (optional)
initialize	Initialize state for algorithm execution (optional)
run	Run label automation on every frame in interval
terminate	Terminate automated algorithm (optional)
settingsDialog	Display algorithm settings (optional)

Version History
Introduced in R2022a

See Also

Apps
Lidar Labeler | **Image Labeler** | **Ground Truth Labeler** | **Video Labeler**

Functions
`groundTruthLidar` | `labelType` | `lidarLabelType` | `lidar.labeler.mixin.Temporal`

Topics
"Create Automation Algorithm for Labeling"
"Temporal Automation Algorithms"
"Automate Ground Truth Labeling For Vehicle Detection Using PointPillars"
"Automate Ground Truth Labeling for Lidar Point Cloud Semantic Segmentation Using Lidar Labeler"
"Automate Attributes of Labeled Objects" (Automated Driving Toolbox)

checkLabelDefinition

Class: `lidar.labeler.AutomationAlgorithm`
Package: `lidar.labeler`

Validate label definition

Syntax

`isValid = checkLabelDefinition(algObj,labelDef)`

Description

In the **Lidar Labeler** app, the `checkLabelDefinition` method checks whether each label defined in the **ROI Labels** and **Scene Labels** panes is valid. The method restricts an automation algorithm to use only relevant labels. For example, a label definition of type `Cuboid` cannot be used to mark a lane boundary.

Clients of `AutomationAlgorithm` must implement this method.

`isValid = checkLabelDefinition(algObj,labelDef)` returns `true` for valid label definitions and `false` for invalid definitions for the automation algorithm provided by `algObj`. `labelDef` is a structure containing all the label definitions in the **ROI Labels** and **Scene Labels** panes. Definitions that return `false` are disabled during automation.

Examples

Restrict Automation to ROI Labels of Any Type

This implementation of the `checkLabelDefinition` method designates ROI labels such as `Cuboid` and `Line` as valid and all other labels as invalid.

```
function isValid = checkLabelDefinition(algObj,labelDef)
    isValid = isROI(labelDef.Type);
end
```

Restrict Automation to Cuboid Labels of Any Type

This implementation of the `checkLabelDefinition` method designates `Cuboid` labels as valid and all other labels as invalid.

```
function isValid = checkLabelDefinition(algObj,labelDef)
    isValid = (labelDef.Type == labelType.Cuboid);
end
```

Input Arguments

algObj — Automation algorithm
`lidar.labeler.AutomationAlgorithm` object

Automation algorithm, specified as a `lidar.labeler.AutomationAlgorithm` object.

`labelDef` — Label definition
structure

Label definition, specified as a structure containing `Type` and `Name` fields.

Field	Description
Type	`labelType` enumeration that contains the type of label definition. Valid label types are: • `labelType.Cuboid` • `labelType.Line` • `labelType.Scene` `lidarLabelType` object contains the label type for voxel labeled ROI. You define the label using `lidarLabelType.Voxel`.
Name	Character vector that contains the name of the label definition.

To view a sample `labelDef` structure that contains a cuboid label definition, enter this code at the MATLAB command prompt.

```
labelDef(1).Type = labelType.Cuboid;
labelDef(1).Name = 'Car';
```

Output Arguments

`isValid` — True of false result of label definition validity check
1 | 0

True or false result of the label definition validity check, returned as a `1` or `0` of data type `logical`.

Tips

• To access the selected label definitions, use the SelectedLabelDefinitions property of the automation algorithm. In the **Lidar Labeler** app, the selected label definitions are highlighted in yellow in the **ROI Labels** and **Scene Labels** panes on the left.

Version History
Introduced in R2022a

See Also
`labelType` | `lidarLabelType` | `lidar.labeler.AutomationAlgorithm` | `checkSignalType` | `checkSetup`

checkSetup

Class: `lidar.labeler.AutomationAlgorithm`
Package: `lidar.labeler`

Set up validation (optional)

Syntax

```
isReady = checkSetup(algObj)
isReady = checkSetup(algObj,labelsToAutomate)
```

Description

In the **Lidar Labeler** app, the `checkSetup` method checks the validity of the setup when you click **Run** in an automation session. If `checkSetup` returns `true`, then the setup is valid and the app proceeds to run the automation algorithm by using the `initialize`, `run`, and `terminate` methods.

Clients of `AutomationAlgorithm` can optionally implement this method.

`isReady = checkSetup(algObj)` returns `true` if you completed set up correctly and the automation algorithm `algObj` can begin execution. Otherwise, `checkSetup` returns `false`.

`isReady = checkSetup(algObj,labelsToAutomate)` additionally provides a table, `labelsToAutomate`, that contains labels selected for the automation algorithm to use for labeling. This syntax is available only for time-dependent (temporal) automation algorithms.

Examples

Check Setup for ROI Labels

This implementation of the `checkSetup` method checks the setup for a temporal automation algorithm. This method determines that an automation algorithm is ready to run if at least one ROI label exists.

```
function isReady = checkSetup(algObj,labelsToAutomate)

    notEmpty = ~isempty(labelsToAutomate);
    hasROILabels = any(labelsToAutomate.Type == labelType.Cuboid ...
                | labelsToAutomate.Type == labelType.Line ...
                | labelsToAutomate.Type == lidarLabelType.Voxel);
    isReady = (notEmpty && hasROILabels)

end
```

Input Arguments

algObj — Automation algorithm
`lidar.labeler.AutomationAlgorithm` object
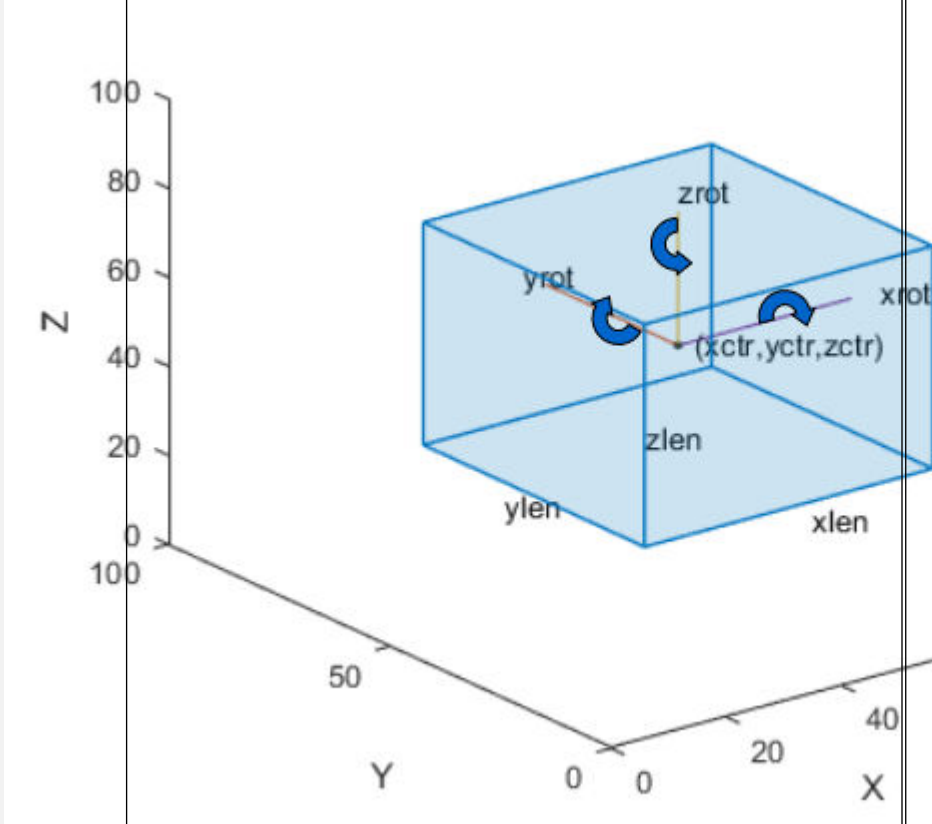
Automation algorithm, specified as a `lidar.labeler.AutomationAlgorithm` object.

labelsToAutomate — Labels selected for automation
table

Labels selected for automation, specified as a table with these columns.

Column Name	Description
Type	`labelType` enumeration that contains the type of label definition. Valid label types are: • `labelType.Cuboid` • `labelType.Line` • `labelType.Scene` `lidarLabelType.Voxel` is not supported.
Name	Character vector that contains the name of the label.
Time	Scalar of type `double` that specifies the time, in seconds, when the label was marked.

Column Name	Description
Position	Location of the label in the frame. The format of this vector depends on the label type.

Label Type	Position Format
Cuboid — Cuboid ROI labels	M-by-9 numeric vector of the form [xctr, yctr, zctr, xlen, ylen, zlen, xrot, yrot, zrot], where: • M is the number of labels in the frame. • xctr, yctr, and zctr specify the center of the cuboid. • xlen, ylen, and zlen specify the length of the cuboid along the *x*-axis, *y*-axis, and *z*-axis, respectively. • xrot, yrot, and zrot specify the rotation angles for the cuboid along the *x*-axis, *y*-axis, and *z*-axis, respectively. These angles are clockwise-positive when looking in the forward direction of their corresponding axes. This figure shows how these values specify the position of a cuboid.
Line — Polyline ROI labels	M-by-1 vector of cell arrays, where M is the number of labels in the frame. Each cell array contains an N-by-2 numeric matrix

Column Name	Description	
	Label Type	**Position Format**
		of the form [x1 y1; x2 y2; ... ; xN yN] for N points in the polyline.
	Scene — Scene labels	Logical value of 1 if the label is present in the frame and 0 otherwise.

Each row of the table corresponds to a label selected for automation. This `labelsToAutomate` table contains a line label with five points, and a cuboid label.

```
    Type            Name            Time          Position
  _____      _____      _____      _____

   Line          'LaneMarker'      0.066667      [5x2 double]
   Cuboid        'Truck'           0.099999      [1x9 double]
```

Output Arguments

isReady — True or false result of setup check
1 | 0

True or false result of the setup check, returned as a 1 or 0 of data type `logical`.

Version History
Introduced in R2022a

See Also
labelType | lidarLabelType | lidar.labeler.AutomationAlgorithm | checkLabelDefinition | checkSignalType

checkSignalType

Class: `lidar.labeler.AutomationAlgorithm`
Package: `lidar.labeler`

Validate signal type

Syntax

`isValid = checkSignalType(signalType)`

Description

In the **Lidar Labeler** app, the `checkSignalType` method validates whether each signal selected for automation is of the type `PointCloud`.

`isValid = checkSignalType(signalType)` returns logical `1` (true) when the specified signal type is valid.

Examples

Restrict Automation to Point Cloud Signals

Implement the `checkSignalType` method to designate `PointCloud` signals as valid and all other signals as invalid.

```
function isValid = checkSignalType(signalType)
    isValid = (signalType == vision.labeler.loading.SignalType.PointCloud);
end
```

Input Arguments

signalType — Signal type
`vision.labeler.loading.SignalType` enumeration

Signal type, specified as a `vision.labeler.loading.SignalType` enumeration.

Example: `vision.labeler.loading.SignalType.PointCloud`

Output Arguments

isValid — Result of signal type validity check
`1` | `0`

Result of the signal type validity check, returned as logical `1` (true) or logical `0` (false).

Attributes

```
Static                                    true
```

To learn about attributes of methods, see Method Attributes.

Version History
Introduced in R2022a

See Also
vision.labeler.loading.SignalType | lidar.labeler.AutomationAlgorithm | checkLabelDefinition | checkSetup

initialize

Class: lidar.labeler.AutomationAlgorithm
Package: lidar.labeler

Initialize state for algorithm execution (optional)

Syntax

```
initialize(algObj,frame)
initialize(algObj,frame,labelsToAutomate)
```

Description

The `initialize` method initializes the state of the automation algorithm before the automation algorithm runs.

Clients of `AutomationAlgorithm` can optionally implement this method.

`initialize(algObj,frame)` initializes the state of the `algObj` automation algorithm using the first frame in the time range of the data being labeled.

Clients of `AutomationAlgorithm` must implement this user-defined method.

`initialize(algObj,frame,labelsToAutomate)` additionally provides a table, `labelsToAutomate`, that contains labels selected for the automation algorithm to use for labeling. This syntax does not support voxel label automation. In addition, this syntax is available only for time-dependent (temporal) automation algorithms.

Input Arguments

algObj — Automation algorithm
lidar.labeler.AutomationAlgorithm object

Automation algorithm, specified as a `lidar.labeler.AutomationAlgorithm` object.

frame — Frame corresponding to start of time range
pointCloud object
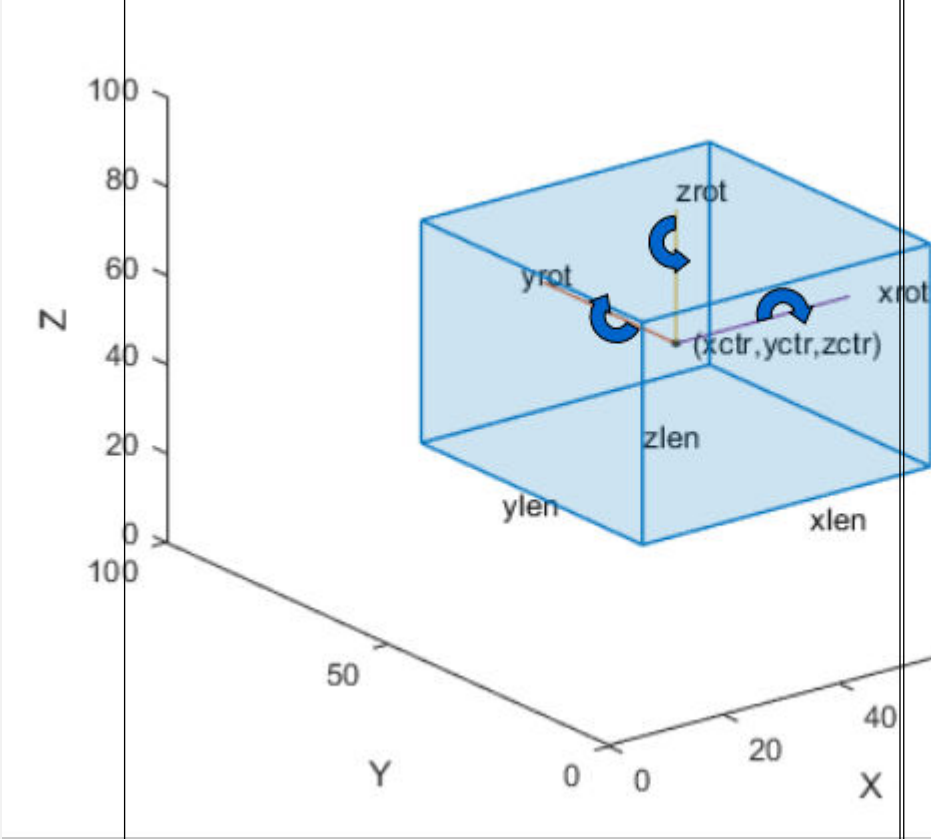
Frame corresponding to the start of time range, specified as a `pointCloud` object.

labelsToAutomate — Labels selected for automation
table

Labels selected for automation, specified as a table with these columns.

2-403

Column Name	Description
Type	`labelType` enumeration that contains the type of label definition. Valid label types are: • `labelType.Cuboid` • `labelType.Line` • `labelType.Scene` `lidarLabelType.Voxel` is not supported.
Name	Character vector that contains the name of the label.
Time	Scalar of type `double` that specifies the time, in seconds, when the label was marked.

Column Name	Description
Position	Location of the label in the frame. The format of this vector depends on the label type.

Label Type	Position Format
Cuboid — Cuboid ROI labels	M-by-9 numeric vector of the form [xctr, yctr, zctr, xlen, ylen, zlen, xrot, yrot, zrot], where: • M is the number of labels in the frame. • xctr, yctr, and zctr specify the center of the cuboid. • xlen, ylen, and zlen specify the length of the cuboid along the *x*-axis, *y*-axis, and *z*-axis, respectively. • xrot, yrot, and zrot specify the rotation angles for the cuboid along the *x*-axis, *y*-axis, and *z*-axis, respectively. These angles are clockwise-positive when looking in the forward direction of their corresponding axes. This figure shows how these values specify the position of a cuboid.
Line — Polyline ROI labels	M-by-1 vector of cell arrays, where M is the number of labels in the frame. Each cell array contains an N-by-2 numeric matrix

Column Name	Description	
	Label Type	**Position Format**
		of the form [x1 y1; x2 y2; ... ; xN yN] for N points in the polyline.
	Scene — Scene labels	Logical value of 1 if the label is present in the frame and 0 otherwise.

Each row of the table corresponds to a label selected for automation. This `labelsToAutomate` table contains a line label with five points, and a cuboid label.

```
     Type            Name            Time           Position
   _____    _____    _____     _____

    Line         'LaneMarker'       0.066667      [5x2 double]
    Cuboid       'Truck'            0.099999      [1x9 double]
```

Version History
Introduced in R2022a

See Also
lidar.labeler.AutomationAlgorithm | labelType | lidarLabelType | checkSetup | run | terminate | initialize

run

Class: lidar.labeler.AutomationAlgorithm
Package: lidar.labeler

Run label automation on every frame in interval

Syntax

autoLabels = run(algObj,frame)

Description

The run method computes the automated labels for a single frame by executing the automation algorithm. During automation, the **Lidar Labeler** app run this method in a loop to compute the automated labels for each frame in the selection being automated.

Clients of AutomationAlgorithm must implement this method.

autoLabels = run(algObj,frame) processes a single frame, frame, using the algObj automation algorithm, and returns the automated labels, autoLabels.

Input Arguments

algObj — Automation algorithm
lidar.labeler.AutomationAlgorithm object

Automation algorithm, specified as a lidar.labeler.AutomationAlgorithm object.

frame — Frame
pointCloud object

Frame whose labels are being computed, specified as a pointCloud object.

Output Arguments

autoLabels — Labels produced by automation

categorical matrix | structure array | table

Labels produced by the automation algorithm, returned as a categorical matrix, structure array, or table.
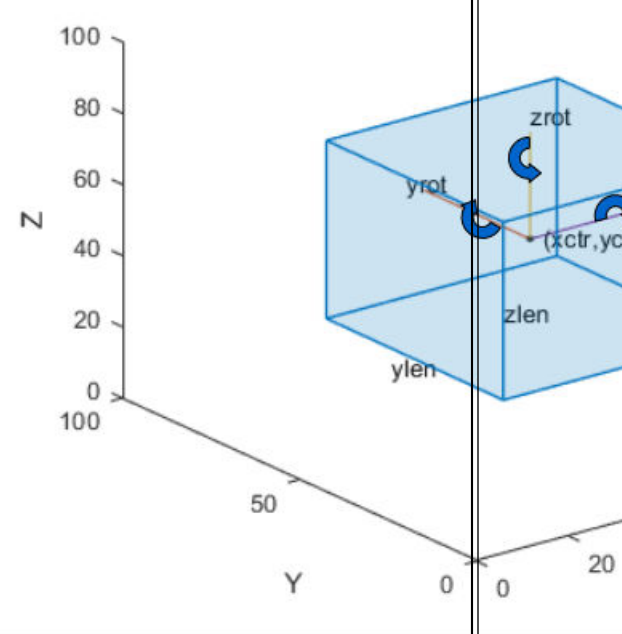
For algorithms that automate voxel labeling, implement the run method to return autoLabels as a categorical label matrix, where each category represents a voxel label.

For algorithms that automate nonvoxel labels, implement the run method to return a structure array. Each structure in the array contains the labels of a specific name and type. The method combines labels of the same name and type into a single structure in the array.

This table describes the columns of the autoLabels table or fields of each autoLabels structure.

Field Name	Description
Type	`labelType` enumeration that contains the type of label definition. Valid label types are: • `labelType.Cuboid` • `labelType.Line` • `labelType.Scene`
Name	Character vector containing the name of the label.

Field Name	Description
Position	Position of labels of the specified Name and Type. The format of Position depends on the label type.

Label Type	Position Format
Cuboid — Cuboid ROI labels	M-by-9 numeric matrix with rows of the form [xctr, yctr, zctr, xlen, ylen, zlen, xrot, yrot, zrot], where: • M is the number of labels in the frame. • xctr, yctr, and zctr specify the center of the cuboid. • xlen, ylen, and zlen specify the length of the cuboid along the *x*-axis, *y*-axis, and *z*-axis, respectively, before rotation has been applied. • xrot, yrot, and zrot specify the rotation angles for the cuboid along the *x*-axis, *y*-axis, and *z*-axis, respectively. These angles are clockwise-positive when looking in the forward direction of their corresponding axes. The figure shows how these values determine the position of a cuboid.

Field Name	Description		
	Label Type	**Position Format**	
	`Line` — Polyline ROI labels	M-by-1 vector of cell arrays, where M is the number of labels in the frame. Each cell array contains an N-by-2 numeric matrix of the form `[x1 y1; x2 y2; ... ; xN yN]` for N points in the polyline.	
	`Scene` — Scene labels	Logical value of `1` if the algorithm determines that the label is present in the frame and `0` otherwise.	
`Attributes` (optional)	Structure array that contains one structure for each attribute in the label. If the label definition does not contain attributes, then the `autoLabels` output does not include this field. For each structure in the `Attributes` structure array, the name of that structure is the name of the corresponding attribute. The value of the structure is the value of the corresponding attribute.		

To view a sample `autoLabels` structure array, enter this code at the MATLAB command prompt.

```
autoLabels(1).Name      = 'Car';
autoLabels(1).Type      = labelType.Cuboid;
autoLabels(1).Position  = [20 20 5 50 20 10 0 0 0];

autoLabels(2).Name      = 'Truck';
autoLabels(2).Type      = labelType.Cuboid;
autoLabels(2).Position  = [70 50 10 70 40 20 0 0 0];
```

You can also use the `run` method to return `autoLabels` as a table. The table rows are equivalent to the structures in a structure array. The table columns are equivalent to the structure fields. This table is equivalent to the sample `autoLabels` structure array previously specified.

```
    Name         Type        Position
 _____    _____    _____

  'Car'         Cuboid      [1x9 double]
  'Truck'       Cuboid      [1x9 double]
```

Version History
Introduced in R2022a

See Also
`lidar.labeler.AutomationAlgorithm` | `checkSetup` | `initialize` | `terminate` | `labelType` | `lidarLabelType`

settingsDialog

Class: lidar.labeler.AutomationAlgorithm
Package: lidar.labeler

Display algorithm settings (optional)

Syntax

settingsDialog(algObj)

Description

The settingsDialog method runs when the user clicks Settings in the labeling app. Use this method to provide a dialog figure with controls for user settings required for the algorithm. Use a modal dialog, created using functions like dialog, inputdlg or listdlg.

Clients of AutomationAlgorithm can optionally implement this method.

settingsDialog(algObj) displays automated algorithm settings in a dialog.

Input Arguments

algObj — Automation algorithm
lidar.labeler.AutomationAlgorithm object

Automation algorithm, specified as a lidar.labeler.AutomationAlgorithm object.

Version History
Introduced in R2022a

supportsMultisignalAutomation

Package: `lidar.labeler`

Set multisignal algorithm automation flag

Syntax

`success = supportsMultisignalAutomation(algObj)`

Description

`success = supportsMultisignalAutomation(algObj)` indicates whether the automation algorithm `algObj` supports the automation of multiple signals in a single automation session. Implement this method in automation algorithms developed for the **Ground Truth Labeler** app, which supports the labeling and automation of multiple signals. If the algorithm supports multisignal automation, then this method returns `success` as `true`.

In automation algorithms developed for the **Image Labeler**, **Video Labeler**, and **Lidar Labeler** apps, which support the labeling and automation of only one signal at a time, you can either delete this method or leave it unchanged. The default implementation of this method indicates that the automation algorithm does not support multisignal automation (`success = false`).

Examples

Indicate Algorithm Supports Multisignal Automation

Implement the `supportsMultisignalAutomation` method to indicate that the automation algorithm supports multisignal automation. This method is static and does not use the input automation algorithm, `algObj`. Therefore, you can specify the input argument as unused by using the tilde (~) operator.

```
function success = supportsMultisignalAutomation(~)
    success = true;
end
```

Input Arguments

algObj — Automation algorithm
`lidar.labeler.AutomationAlgorithm` object

Automation algorithm, specified as a `lidar.labeler.AutomationAlgorithm` object.

Version History
Introduced in R2022a

2-413

See Also

Apps
Ground Truth Labeler

Objects
`vision.labeler.AutomationAlgorithm`

terminate

Class: lidar.labeler.AutomationAlgorithm
Package: lidar.labeler

Terminate automated algorithm (optional)

Syntax

terminate(algObj)

Description

The terminate method cleans up the state of the automation algorithm after run processes the last frame in the specified interval or when you stop the automation algorithm.

Clients of AutomationAlgorithm can optionally implement this method.

terminate(algObj) cleans up the state of the automation algorithm.

Input Arguments

algObj — Automation algorithm
lidar.labeler.AutomationAlgorithm object

Automation algorithm, specified as a lidar.labeler.AutomationAlgorithm object.

Version History
Introduced in R2022a

See Also
checkSetup | initialize | run | lidar.labeler.AutomationAlgorithm

lidar.labeler.mixin.Temporal class

Package: `lidar.labeler.mixin`

Mixin interface for adding temporal context to automation algorithms

Description

The `lidar.labeler.mixin.Temporal` class provides an interface for attaching temporal properties to an automation algorithm used by the app.

The `lidar.labeler.mixin.Temporal` class is a `handle` class.

Class Attributes

Abstract	true

For information on class attributes, see "Class Attributes".

Properties

`StartTime` — Timestamp of first frame
scalar

Timestamp of the first frame of the algorithm interval, specified as a scalar.

Attributes:

GetAccess	public
SetAccess	private
Dependent	true
NonCopyable	true

`CurrentTime` — Timestamp of current executing frame
scalar

Timestamp of the current executing frame, specified as a scalar. This value updates during the execution of the algorithm.

Attributes:

GetAccess	public
SetAccess	private

`EndTime` — Timestamp of last frame
scalar

Timestamp of the last frame of the algorithm interval, specified as a scalar.

Attributes:

GetAccess	public
SetAccess	private
Dependent	true
NonCopyable	true

StartFrameIndex — Index of first frame
integer

Index of the first frame of the algorithm interval, specified as an integer.

Attributes:

GetAccess	public
SetAccess	private
Dependent	true
NonCopyable	true

EndFrameIndex — Index of last frame
integer

Index of the last frame of the algorithm interval, specified as an integer.

Attributes:

GetAccess	public
SetAccess	private
Dependent	true
NonCopyable	true

AutomationDirection — Direction in which to run automated algorithm
`'Forward'` | `'Reverse'`

Direction in which to run the automated algorithm, specified as `'Forward'` or `'Reverse'`.

Attributes:

GetAccess	public
SetAccess	private

Version History
Introduced in R2022a

See Also

Apps
Lidar Labeler | **Ground Truth Labeler** | **Video Labeler**

Objects
`lidar.labeler.AutomationAlgorithm`

Topics
"Create Automation Algorithm for Labeling"

"Temporal Automation Algorithms"
"Automate Ground Truth Labeling For Vehicle Detection Using PointPillars"
"Automate Ground Truth Labeling for Lidar Point Cloud Semantic Segmentation Using Lidar Labeler"

supportsReverseAutomation

Set reverse algorithm automation flag

Syntax

```
flag = supportsReverseAutomation(algObj)
```

Description

`flag = supportsReverseAutomation(algObj)` indicates whether the temporal automation algorithm, `algObj`, supports automation in the reverse direction. A `true` value enables the **Ground Truth Labeler** or **Video Labeler** to open the algorithm in reverse mode.

Examples

Set Algorithm Automation Direction Flag

```
function flag = supportsReverseAutomation(algObj)
      flag = true;
end
```

Input Arguments

algObj — Temporal automation algorithm
`lidar.labeler.mixin.Temporal` object

Temporal automation algorithm, specified as a `lidar.labeler.mixin.Temporal` object.

Output Arguments

flag — Reverse automation indicator
`true | false`

Reverse automation indicator, returned as `true` or `false`.

Version History
Introduced in R2022a

See Also

Apps
Lidar Labeler | **Ground Truth Labeler** | **Video Labeler**

Objects

lidarLabelType

Lidar label type enumerations for labeling

Description

The `lidarLabelType` enumeration enables you to specify the voxel labels used in the **Lidar Labeler** app. When creating label definitions by using a `labelDefinitionCreatorLidar` object, use this enumerations to create label definitions of voxel type. When selecting labels from a `groundTruthLidar` object, use these enumerations to select labels of voxel type.

Creation

Syntax

```
lidarLabelType('Voxel')
```

Description

`lidarLabelType('Voxel')` creates a voxel region of interest (ROI) label type for labeling point cloud data. You can also use the programmatic format, `lidarLabelType.Voxel`.

Object Functions
isROI   Determine if label types are ROI labels
isScene   Determine if label types are Scene labels

Examples

Specify Label Types Using Lidar Label Definition Creator

Create a lidar label definition creator object.

```
ldc = labelDefinitionCreatorLidar();
```

Add labels named "Road", "Bike" with the label types specified as line, cuboid, respectively.

```
addLabel(ldc,'Road',labelType.Line);
addLabel(ldc,'Bike',labelType.Cuboid);
```

Create voxel labels named as "Car" and "Tree".

```
addLabel(ldc,'Car',lidarLabelType.Voxel);
addLabel(ldc,'Tree',lidarLabelType.Voxel);
```

Display the details of the definitions stored in the lidar label definition creator object.

```
ldc
```

```
ldc =
labelDefinitionCreatorLidar contains the following labels:

    Road with 0 attributes and belongs to None group.    (info)
    Bike with 0 attributes and belongs to None group.    (info)
    Car with 0 attributes and belongs to None group.    (info)
    Tree with 0 attributes and belongs to None group.    (info)

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

Version History

Introduced in R2022a

See Also

Apps
Lidar Labeler | **Image Labeler** | **Ground Truth Labeler** | **Video Labeler**

Objects
labelDefinitionCreatorLidar | groundTruthLidar | groundTruthMultisignal | attributeType

Topics
"Get Started with the Lidar Labeler"
"Choose an App to Label Ground Truth Data"

lasFileWriter

LAS or LAZ file writer

Description

A `lasFileWriter` object stores metadata for a LAS or LAZ file as read-only properties. The `writePointCloud` object function uses these properties to write point cloud data as a LAS or LAZ file. The `lasFileWriter` object supports up to the LAS 1.4 specification.

`lasFileWriter` supports writing only unorganized `pointCloud` objects. The created LAS file contains a public header, which contains LAS file metadata, followed by the lidar point records.

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.

Creation

Syntax

```
lasWriter = lasFileWriter(fileName)
lasWriter = lasFileWriter(fileName,Name=Value)
```

Description

`lasWriter = lasFileWriter(fileName)` creates a `lasFileWriter` object with default properties to write lidar point cloud data into a LAS or LAZ file with the specified name `fileName`. The `fileName` input sets the `FileName` property.

`lasWriter = lasFileWriter(fileName,Name=Value)` specifies the properties of the `lasFileWriter` object by using one or more name-value arguments.

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. You can specify the extensions `.las` or `.laz`. If you do not specify a file extension, the default extension is `.laz`.

LasVersion — LAS or LAZ file version
"1.2" (default) | "1.0" | "1.1" | "1.3" | "1.4"

This property is read-only.

LAS or LAZ file version, specified as "1.0", "1.1", "1.2", "1.3", or "1.4".

Example: `LasVersion="1.4"` specifies the LAS version as 1.4.

PointDataFormat — Point data record format ID
3 (default) | 0 | 1 | 2 | 6 | 7 | 8

This property is read-only.

Point data record format ID, specified as 0, 1, 2, 3, 6, 7, or 8. Which point data record formats you can specify dependents on the specified `LasVersion` property.

LAS or LAZ Version	Supported Point Data Record Formats
1.0	Point data record formats 0 and 1
1.1	Point data record formats 0 and 1
1.2	Point data record formats 0 to 3
1.3	Point data record formats 0 to 3
1.4	Point data record formats 0 to 3 and 6 to 8

For more information, see "Point Data Record Format" on page 2-272.

Example: `PointDataFormat=2` specifies the point data record format as 2.

XYZScale — Scale of coordinates
`"auto"` (default) | three-element real-valued row vector

This property is read-only.

Scale of the coordinates, specified as `"auto"` or a three-element real-valued row vector of the form [*Xscale Yscale Zscale*]. When you call the `writePointCloud` function, the default value `"auto"` calculates the XYZScale value using the XLimits, YLimits, and ZLimits properties of the input `pointCloud` object. For more information, see "Point Cloud Data Representation" on page 2-424.

Example: `XYZScale=[10 20 30]` specifies the scale factors of the *X*-, *Y*-, and *Z*-coordinates as 10, 20, and 30 respectively.

XYZOffset — Offset of coordinates
`"auto"` (default) | three-element real-valued row vector

This property is read-only.

Offset of the coordinates, specified as `"auto"` or a three-element real-valued row vector of the form [*Xoffset Yoffset Zoffset*]. When you call the `writePointCloud` function, the default value `"auto"` calculates the XYZOffset value using the XLimits, YLimits, and ZLimits properties of the input `pointCloud` object. For more information, see "Point Cloud Data Representation" on page 2-424.

Example: `XYZOffset=[5 3 2]` specifies the offset values of the *X*-, *Y*-, and *Z*-coordinates as 5, 3, and 2 respectively.

Object Functions
writePointCloud   Write point cloud data to LAS or LAZ file
addVLR   Add VLR data to lasFileWriter object

Examples

Write Point Cloud Data to LAS File

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

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

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

```
ptCloud = readPointCloud(lasReader,Classification=2);
```

Create a `lasFileWriter` object to store the point cloud data in a LAS file.

```
lasWriter = lasFileWriter("ground.las");
```

Write the point cloud data to the LAS file by using the `writePointCloud` function.

```
writePointCloud(lasWriter,ptCloud);
```

Algorithms

The point cloud coordinate values are calculated as:

$$x = X * Xscale + Xoffset$$
$$y = Y * Yscale + Yoffset$$
$$z = Z * Zscale + Zoffset$$

where, *Xscale*, *Yscale*, and *Zscale* are set by the `XYZScale` property, and *Xoffset*, *Yoffset*, and *Zoffset* are set by the `XYZOffset` property. *X*, *Y*, and *Z* are the raw coordinate values of the point cloud data. You must specify the [*X Y Z*] values for each point in a `pointCloud` object when writing them to a LAS or LAZ file using the `writePointCloud` function.

Version History
Introduced in R2022a

Support for VLR Data

You can write VLR data into the LAS or LAZ file by using the `addVLR` function.

See Also

Functions
pcwrite | pcshow | writePointCloud | addVLR

Objects
pointCloud | lidarPointAttributes | lasFileReader

writePointCloud

Write point cloud data to LAS or LAZ file

Syntax

```
writePointCloud(lasWriter,ptCloud)
writePointCloud(lasWriter,ptCloud,ptAttributes)
```

Description

writePointCloud(lasWriter,ptCloud) writes point cloud data from a non-empty, unorganized pointCloud object, ptCloud, to a LAS or LAZ file using the lasFileWriter object lasWriter.

writePointCloud(lasWriter,ptCloud,ptAttributes) additionally writes the point attributes specified by ptAttributes to the LAS or LAZ file.

Examples

Write Point Cloud Data and Point Attributes to LAZ File

Create a lasFileReader object to access LAZ file data.

```
fileName = fullfile(toolboxdir("lidar"),"lidardata", ...
    "las","aerialLidarData2.las");
lasReader = lasFileReader(fileName);
```

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

```
[ptCloud,pointAttributes] = readPointCloud(lasReader, ...
    Classification=3:6, ...
    Attributes=["GPSTimeStamp","ScanAngle"]);
```

Create a lasFileWriter object to store the point cloud data in a LAS file.

```
lasWriter = lasFileWriter("points",PointDataFormat=1);
```

Write points related to vegetation and building in LAZ file.

```
writePointCloud(lasWriter,ptCloud,pointAttributes);
```

Input Arguments

lasWriter — LAS or LAZ file writer
lasFileWriter object

LAS or LAZ file writer, specified as a lasFileWriter object.

ptCloud — Point cloud
pointCloud object

Point cloud, specified as an unorganized `pointCloud` object.

Use the `removeInvalidPoints` function to remove invalid points from the point cloud and to convert an organized point cloud to an unorganized point cloud.

ptAttributes — Point attributes
`lidarPointAttributes` object

Point attributes, specified as a `lidarPointAttributes` object. Unspecified fields of the `lidarPointAttributes` object are set to their default values. The default values for the `LaserReturn` and `NumReturns` fields are `1`, while all other point properties defined by the `ptCloud` and `PtAttributes` objects default to `0`.

Version History
Introduced in R2022a

See Also

Functions
pcwrite | pcshow

Objects
pointCloud | lasFileReader | lidarPointAttributes

addVLR

Add VLR data to `lasFileWriter` object

Syntax

```
addVLR(lasWriter,recordID,userID,vlrData)
addVLR( ___ ,description)
```

Description

`addVLR(lasWriter,recordID,userID,vlrData)` adds the variable length record (VLR) data with the specified record ID and the user ID to the `lasFileWriter` object `lasWriter`.

`addVLR( ___ ,description)` adds a description for the VLR data.

Examples

Write VLR Data to LAS File

Create a `lasFileReader` object to read LAS/LAZ file data into the workspace.

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

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

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

Create a `lasFileWriter` object to store the point cloud data read from the file. Specify the filename and the LAS version.

```
lasWriter = lasFileWriter("points.las",LasVersion="1.4");
```

Add VLR data to the `lasFileWriter` object using the `addVLR` function. You must specify the record ID, user ID, and the VLR data.

```
addVLR(lasWriter,3,"LASF_Spec","Sample VLR data")
```

Write the data into the LAS file using the `writePointCloud` function.

```
writePointCloud(lasWriter,ptCloud)
```

Write CRS Data in GeoTIFF Format to LAS File

Create a `lasFileReader` object to read LAS/LAZ file data into the workspace.

```
filename = fullfile(toolboxdir("lidar"),"lidardata","las", ...
    "aerialLidarData2.las");
lasReader = lasFileReader(filename);
```

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

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

Read CRS data from the VLRs in GeoTIFF format using the `readVLR` function.

```
geoKeyVLR = readVLR(lasReader,34735);
geoAsciiParamsVLR = readVLR(lasReader,34737);
```

Create a `lasFileWriter` object to store the point cloud data read from the file. Specify the filename and the LAS version.

```
lasWriter = lasFileWriter("pointsWithCRS.las",LasVersion="1.4");
```

Add geo-keys to the `lasFileWriter` object.

```
addVLR(lasWriter,34735,"LASF_Projection",geoKeyVLR.Data.KeyEntries)
```

Add geo-ASCII parameters to the `lasFileWriter` object.

```
addVLR(lasWriter,34737,"LASF_Projection",geoAsciiParamsVLR.Data)
```

Write the data into the LAS file using the `writePointCloud` function.

```
writePointCloud(lasWriter,ptCloud)
```

Input Arguments

`lasWriter` — LAS or LAZ file writer
`lasFileWriter` object

LAS or LAZ file writer, specified as a `lasFileWriter` object.

`recordID` — Record ID for VLR data
nonnegative integer

Record ID for the VLR data, specified as a nonnegative integer in the range `[0, 65535]`. For some standard record IDs, see "VLR and CRS Data" on page 2-429.

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

`userID` — User ID for VLR data
character vector | string scalar

User ID for the VLR data, specified as a character vector or string scalar. This value identifies the user that created the VLR data. For some standard user IDs, see "VLR and CRS Data" on page 2-429.

Data Types: `char` | `string`

`vlrData` — VLR Data
character vector | string scalar | structure | numeric array

Variable length record data, specified as a character vector, a string scalar, a structure, or a numeric array. For more information on VLR data for standard record ID and user ID combinations, see "VLR and CRS Data" on page 2-429.

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

description — Description for VLR data
character vector | string scalar

Description for the VLR data, specified as a character vector or a string scalar.

Data Types: `char` | `string`

Algorithms

A LAS or LAZ file contains a public header block, variable length records (VLR), point data records, and extended variable length records (EVLR). VLRs and EVLRs are optional.

VLRs contain data about the projection information, metadata, waveform, and user information. This table gives the list of some standard VLRs defined in the LAS specification.

Record ID	User ID	VLR Data
0	LASF_Spec	A character vector or a string representing the data from the `Classification Lookup` record.
3	LASF_Spec	A character vector or a string representing the ASCII data from the `Text Area Description` record.
2111	LASF_Projection	A character vector or a string representing the ASCII data from the `OGC Math Transform WKT` record.
2112	LASF_Projection	A character vector or a string representing the ASCII data from the `OGC Coordinate System WKT` record.

34735	LASF_Projection	An array of structures representing the data from the GeoTiff key entries of the `GeoKeyDirectoryTag` record. The structure has these fields: • `KeyID` — The ID for each key of the GeoTIFF data. • `TiffTagLocation` — Indicates the data location for the corresponding key. It can take one of these three values: `0`, `34736`, or `34737`. • `Count` — Indicates the number of characters in the `GeoAsciiParamsTag` string value. Otherwise, this value is 1. • `ValueOffset` — The offset value for the data location. This value depends on the value of `TiffTagLocation`.
34736	LASF_Projection	A `double` vector representing the numeric values from the `GeoDoubleParamsTag` record.
34737	LASF_Projection	A character vector or a string representing the ASCII data from the `GeoDoubleParamsTag` record.

Coordinate reference system (CRS) data in a LAS file is represented in the well know text (WKT) or GeoTIFF format.

- VLRs with record IDs 2111 and 2112, with the user ID `LASF_Projection`, represent the geo-referencing information for the CRS data using well known text (WKT), in LAS files with point data format, 6 to 10.
- VLRs with record IDs 34735, 34736, and 34737, with the user ID `LASF_Projection`, represent the CRS data using GeoTIFF, in LAS files with point data format, 1 to 5.

For more information on the LAS file format, see the ASPRS LASER (LAS) File Format Exchange Activities page.

Version History
Introduced in R2022b

See Also
lasFileWriter | writePointCloud | lasFileReader | readVLR | readCRS

ousterFileReader

Read point cloud data from Ouster PCAP file

Description

The `ousterFileReader` object reads point cloud data from an Ouster® packet capture (PCAP) file.

Creation

Syntax

```
ousterReader = ousterFileReader(fileName,calibrationFile)
ousterReader = ousterFileReader(fileName,Name=Value)
```

Description

`ousterReader = ousterFileReader(fileName,calibrationFile)` creates an `ousterFileReader` object that reads point cloud data from an Ouster PCAP file. Specify the PCAP file `fileName` and the calibration file `calibrationFile`. The inputs set the `FileName` and `CalibrationFile` properties, respectively.

`ousterReader = ousterFileReader(fileName,Name=Value)` specify `SkipPartialFrames` and `CoordinateFrame` properties using one or more name-value arguments. For example, `ousterFileReader(fileName,calibrationFile,SkipPartialFrames=0)` creates an Ouster file reader that does not skip partial frames.

Properties

`FileName` — Name of Ouster PCAP file
character vector | string scalar

This property is read-only.

Name of the Ouster PCAP file, specified as a character vector or string scalar.

`CalibrationFile` — Name of Ouster calibration JSON file
character vector | string scalar

This property is read-only.

Name of the Ouster calibration JSON file, specified as a character vector or string scalar.

Note Specifying the incorrect calibration file returns no frames or an improperly calibrated point cloud.

SkipPartialFrames — Partial frame processing
true or 1 (default) | false or 0

This property is read-only.

Partial frame processing, specified as a logical 1 (true) or 0 (false). To skip partial frames, defined as frames with a horizontal field of view less than the mean horizontal field of view of all frames in the PCAP file, specify this property as true. Otherwise, specify it as false.

To set this property, you must specify it at object creation.

Example: SkipPartialFrames=true skips partial frames in the PCAP file.

CoordinateFrame — Coordinate frame for point cloud data
"center" (default) | "base"

This property is read-only.

Coordinate frame for point cloud data, specified as one of these options.

- "center" — Origin of the coordinate frame is at the center of the sensor.
- "base" — Origin of the coordinate frame is at the base of the sensor.

To set this property, you must specify it at object creation.

Example: CoordinateFrame="center" sets the origin of the coordinate frame at the center of the sensor.

Data Types: char | string

DeviceModel — Name of device model
character vector

This property is read-only.

Name of the device model, specified as a character vector.

LidarMode — Mode of lidar sensor
character vector

This property is read-only.

Mode of the lidar sensor, specified as a character vector. The mode defines the horizontal resolution and rotation frequency of the lidar sensor.

NumberOfFrames — Total number of point cloud frames in file
positive integer

This property is read-only.

Total number of point cloud frames in the file, specified as a positive integer.

Duration — Total duration of file
duration scalar

This property is read-only.

Total duration of the file, specified as a `duration` scalar in seconds.

StartTime — Time of first point cloud reading
`duration` scalar

This property is read-only.

Time of the first point cloud reading, specified as a `duration` scalar in seconds.

EndTime — Time of final point cloud reading
`duration` scalar

This property is read-only.

Time of the final point cloud reading, specified as a `duration` scalar in seconds.

CurrentTime — Time of current point cloud reading
`duration` scalar

Time of the current point cloud reading, specified as a `duration` scalar in seconds. As you read point clouds using `readFrame`, this property updates with the most recent point cloud reading time. You can use `reset` to reset the value of this property to the default value. The default value matches the `StartTime` property.

Timestamps — Start time for each point cloud frame
`duration` vector

This property is read-only.

Start time for each point cloud frame, specified as a `duration` vector with values in seconds. The length of the vector is equal to the value of the `NumberOfFrames` property. The value of the first element in the vector is same as the value of the `StartTime` property. You can use this property to read point cloud frames captured at different times.

Object Functions

hasFrame   Determine if another Ouster point cloud is available
readFrame   Read Ouster point cloud from file
reset   Reset ousterFileReader object to first frame

Examples

Read and Visualize Point Clouds from Ouster PCAP File

Download a ZIP file containing an Ouster packet capture (PCAP) file and the corresponding calibration file, and then unzip the file.

```
zipFile = matlab.internal.examples.downloadSupportFile("lidar","data/ouster_RoadIntersection.zip"
saveFolder = fileparts(zipFile);
pcapFileName = [saveFolder filesep 'ouster_RoadIntersection' filesep 'ouster_RoadIntersection.pca
calibFileName = [saveFolder filesep 'ouster_RoadIntersection' filesep 'OS1-128U.json'];
if ~(exist(pcapFileName,"file") && exist(calibFileName,"file"))
    unzip(zipFile,saveFolder);
end
```

Create an `ousterFileReader` object.

```
ousterReader = ousterFileReader(pcapFileName,calibFileName);
```

Define X-, Y-, and Z-axes limits for `pcplayer`, in meters.

```
xlimits = [-60 60];
ylimits = [-60 60];
zlimits = [-20 20];
```

Create a point cloud player.

```
player = pcplayer(xlimits,ylimits,zlimits);
```

Set labels for the `pcplayer` axes.

```
xlabel(player.Axes,"X (m)");
ylabel(player.Axes,"Y (m)");
zlabel(player.Axes,"Z (m)");
```

Specify the `CurrentTime` of the Ouster file reader so that it starts reading from 0.3 seconds after the start time.

```
ousterReader.CurrentTime = ousterReader.StartTime + seconds(0.3);
```

Display the stream of point clouds from `CurrentTime` to the final point cloud.

```
while(hasFrame(ousterReader) && player.isOpen())
    ptCloud = readFrame(ousterReader);
    view(player,ptCloud);
end
```
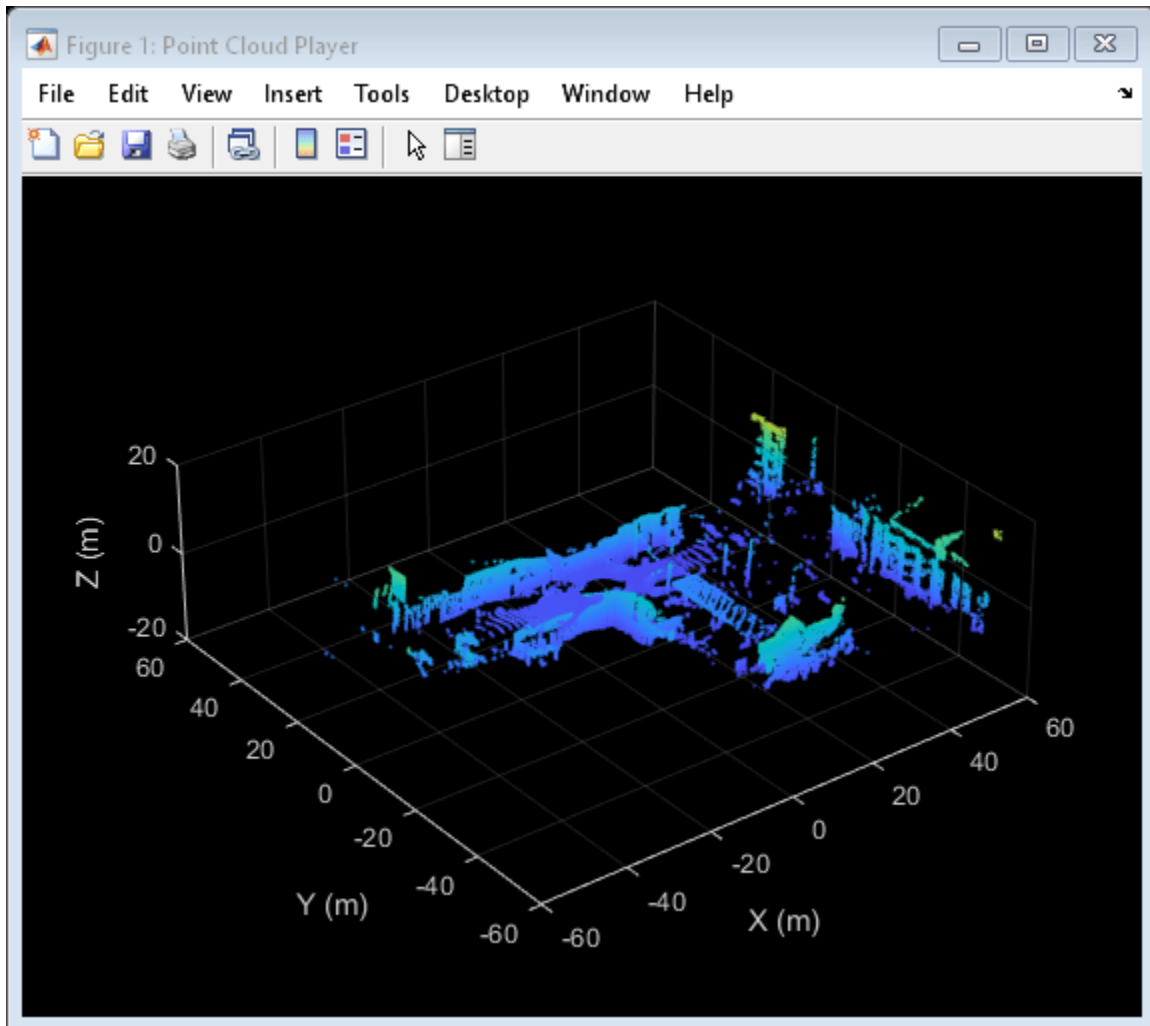
Version History

Introduced in R2022a

See Also

Functions
readFrame | hasFrame | pcplayer

Objects
hesaiFileReader | pointCloud | velodyneFileReader

External Websites
Ouster Product Documentation

hasFrame

Determine if another Ouster point cloud is available

Syntax

```
isAvailable = hasFrame(ousterReader)
```

Description

`isAvailable = hasFrame(ousterReader)` determines if another point cloud is available in the packet capture (PCAP) file of the input Ouster file reader. As you read point clouds using `readFrame`, the point clouds are read sequentially until this function returns `false`.

Examples

Check for Next Point Cloud in Ouster PCAP File

Download a ZIP file containing an Ouster packet capture (PCAP) file and the corresponding calibration file, and then unzip the file.

```
zipFile = matlab.internal.examples.downloadSupportFile("lidar","data/ouster_RoadIntersection.zip"
saveFolder = fileparts(zipFile);
pcapFileName = [saveFolder filesep 'ouster_RoadIntersection' filesep 'ouster_RoadIntersection.pc
calibFileName = [saveFolder filesep 'ouster_RoadIntersection' filesep 'OS1-128U.json'];
if ~(exist(pcapFileName,"file") && exist(calibFileName,"file"))
    unzip(zipFile,saveFolder);
end
```

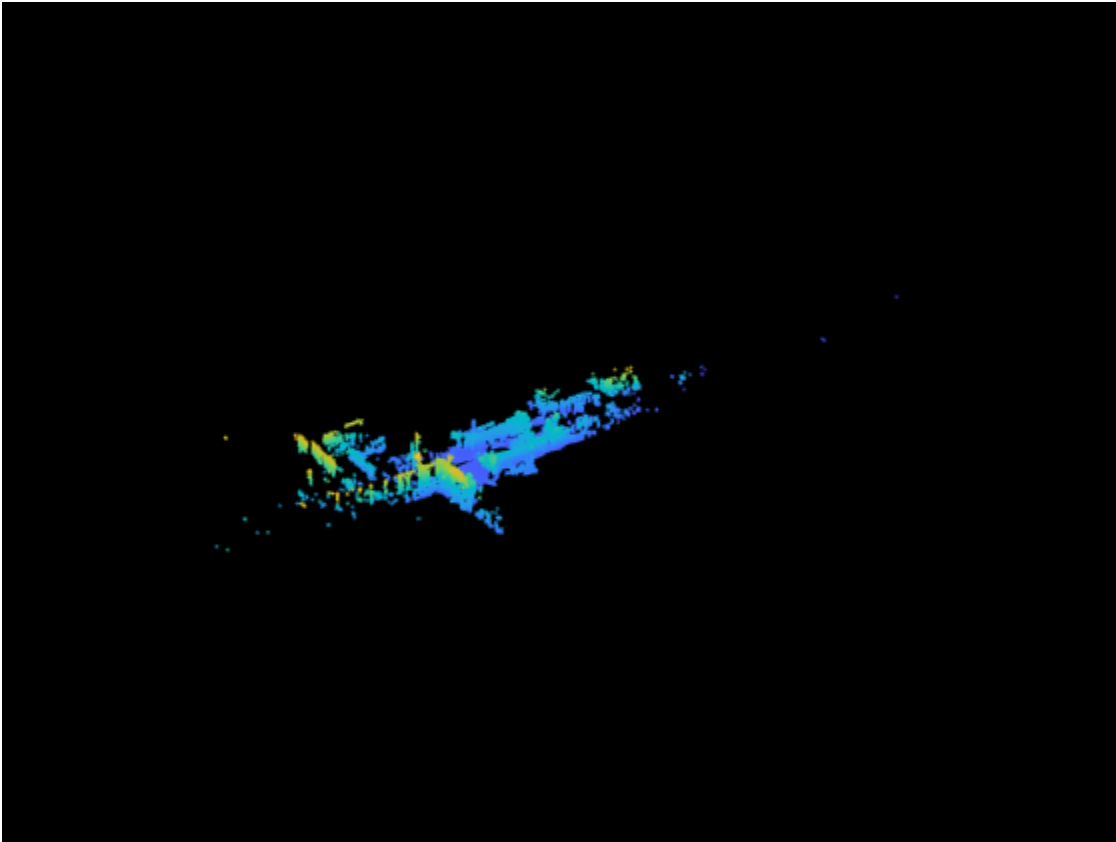Create an `ousterFileReader` object.

```
ousterReader = ousterFileReader(pcapFileName,calibFileName);
```

Check if the file reader has a point cloud to read using `hasFrame`.

```
disp(hasFrame(ousterReader))
```

```
1
```

Read the last point cloud frame of the file.

```
ptCloud = readFrame(ousterReader,ousterReader.NumberOfFrames);
```

Check if `ousterReader` has a next point cloud available to read.

```
disp(hasFrame(ousterReader))
```

```
0
```

Input Arguments

ousterReader — Ouster file reader
ousterFileReader object

Ouster file reader, specified as ousterFileReader object.

Output Arguments

isAvailable — Frame is available
true or 1 (default) | false or 0

Frame is available, returned as 1 (true) or 0 (false). This argument returns true if the Ouster file reader contains one or more point cloud frames to read after the current frame. Otherwise, it returns false.

Version History
Introduced in R2022a

See Also
ousterFileReader | readFrame | reset | velodyneFileReader | hasFrame | hesaiFileReader | hasFrame | pointCloud

External Websites
Ouster Product Documentation

readFrame

Read Ouster point cloud from file

Syntax

```
ptCloud = readFrame(ousterReader)
ptCloud = readFrame(ousterReader,frameNumber)
ptCloud = readFrame(ousterReader,frameTime)
[ptCloud,pcAttributes] = readFrame( ___ )
```

Description

`ptCloud = readFrame(ousterReader)` reads the next point cloud in sequence from the Ouster PCAP file and returns a `pointCloud` object.

`ptCloud = readFrame(ousterReader,frameNumber)` reads the point cloud with the specified frame number from the file.

`ptCloud = readFrame(ousterReader,frameTime)` reads the first point cloud recorded at or after the given `frameTime`.

`[ptCloud,pcAttributes] = readFrame( ___ )` returns a structure, `pcAttributes`, containing attributes for each point using any combination of input arguments from previous syntaxes.

Examples

Read Ouster PCAP Point Cloud Using Frame Number

Download a ZIP file containing an Ouster packet capture (PCAP) file and the corresponding calibration file, and then unzip the file.

```
zipFile = matlab.internal.examples.downloadSupportFile("lidar","data/ouster_RoadIntersection.zip"
saveFolder = fileparts(zipFile);
pcapFileName = [saveFolder filesep 'ouster_RoadIntersection' filesep 'ouster_RoadIntersection.pca
calibFileName = [saveFolder filesep 'ouster_RoadIntersection' filesep 'OS1-128U.json'];
if ~(exist(pcapFileName,"file") && exist(calibFileName,"file"))
    unzip(zipFile,saveFolder);
end
```
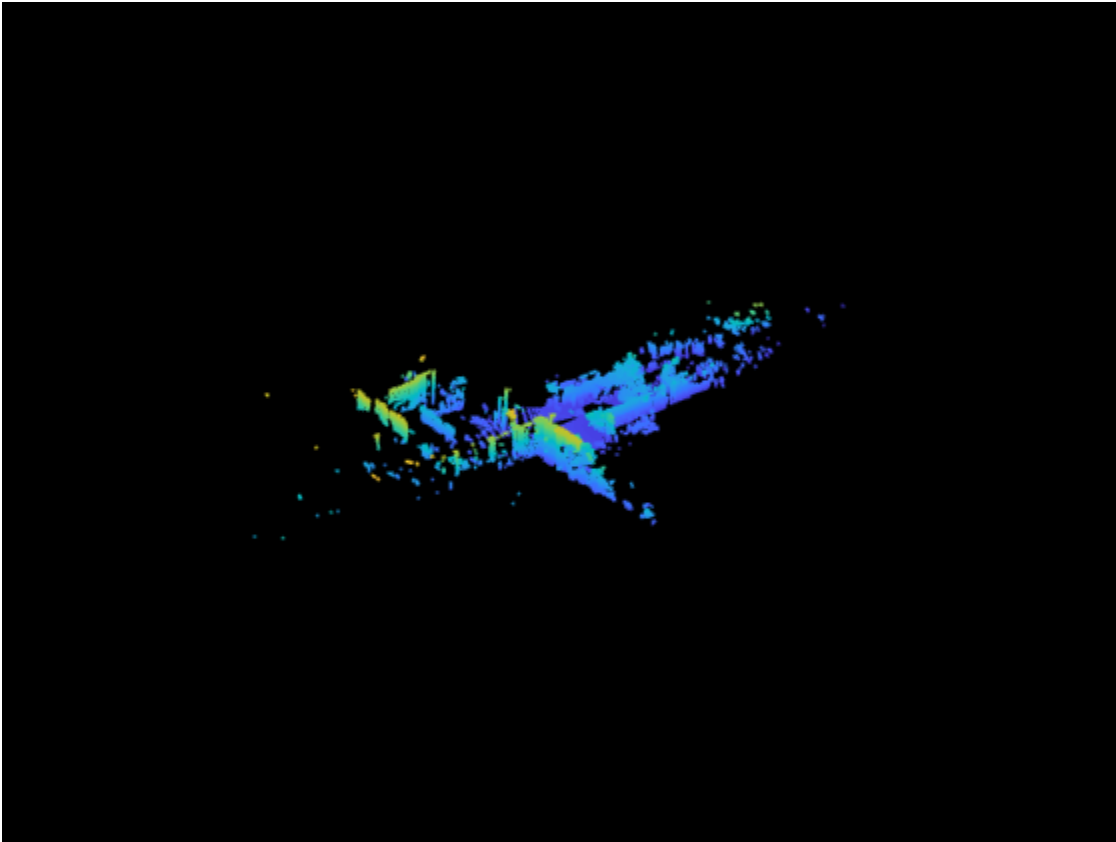
Create an `ousterFileReader` object.

```
ousterReader = ousterFileReader(pcapFileName,calibFileName);
```

Read the fifth frame of the Ouster PCAP point cloud data.

```
frameNumber = 5;
ptCloud = readFrame(ousterReader,frameNumber);
```

Display the point cloud.

```
pcshow(ptCloud)
```

Read Ouster PCAP Point Cloud Using Time Duration

Download a ZIP file containing an Ouster packet capture (PCAP) file and the corresponding calibration file, and then unzip the file.

```
zipFile = matlab.internal.examples.downloadSupportFile("lidar","data/ouster_RoadIntersection.zip'
saveFolder = fileparts(zipFile);
pcapFileName = [saveFolder filesep 'ouster_RoadIntersection' filesep 'ouster_RoadIntersection.pca
calibFileName = [saveFolder filesep 'ouster_RoadIntersection' filesep 'OS1-128U.json'];
if ~(exist(pcapFileName,"file") && exist(calibFileName,"file"))
    unzip(zipFile,saveFolder);
end
```

Create an ousterFileReader object.

```
ousterReader = ousterFileReader(pcapFileName,calibFileName);
```

Read the first Ouster PCAP point cloud frame from 3 seconds after start time.

```
frameTime = ousterReader.StartTime + seconds(3);
[ptCloud,pcatt] = readFrame(ousterReader,frameTime);
```

Display the point cloud.

```
pcshow(ptCloud)
```

Input Arguments

ousterReader — Ouster file reader
ousterFileReader object

Ouster file reader, specified as ousterFileReader object.

frameNumber — Frame number of desired point cloud in file
positive integer

Frame number of the desired point cloud in the file, specified as a positive integer. Frame numbers are sequential.

frameTime — Frame time of desired point cloud in file
duration scalar

Frame time of the desired point cloud in the file, specified as a duration scalar in seconds. The function returns the first frame available at or after the specified frameTime.

Output Arguments

ptCloud — Point cloud
pointCloud object

Point cloud, returned as a `pointCloud` object.

`pcAttributes` — Point cloud attributes
structure

Point cloud attributes for each point, returned as a structure that contains these fields:

- `Range` — Distance from the origin, specified as an *M*-by-*N* matrix, same as the size of `Location` property of `pointCloud` object `ptCloud`, in meters.

- `SignalPhoton` — Signal intensity of photons in the signal return measurement, specified as an *M*-by-*N* matrix, same as the size of `Location` property of `pointCloud` object `ptCloud`.

- `NearInfrared` — Near infrared (NIR) photons related to natural environmental illumination, specified as an *M*-by-*N* matrix, same as the size of `Location` property of `pointCloud` object `ptCloud`.

Version History
Introduced in R2022a

See Also
ousterFileReader | hasFrame | reset | velodyneFileReader | readFrame | hesaiFileReader | readFrame | pointCloud | pcshow

External Websites
Ouster Product Documentation

reset

Reset `ousterFileReader` object to first frame

Syntax

```
reset(ousterReader)
```

Description

`reset(ousterReader)` resets the Ouster file reader object `ousterReader` to the first frame by resetting its `CurrentTime` property to the default value. The default value is the value of the `StartTime` property of `ousterReader`.

Examples

Reset `ousterFileReader` Object

Download a ZIP file containing an Ouster packet capture (PCAP) file and the corresponding calibration file, and then unzip the file.

```
zipFile = matlab.internal.examples.downloadSupportFile("lidar", "data/ouster_RoadIntersection.zip
saveFolder = fileparts(zipFile);
pcapFileName = [saveFolder filesep 'ouster_RoadIntersection' filesep 'ouster_RoadIntersection.pca
calibFileName = [saveFolder filesep 'ouster_RoadIntersection' filesep 'OS1-128U.json'];
if ~(exist(pcapFileName,"file") && exist(calibFileName,"file"))
    unzip(zipFile,saveFolder);
end
```

Create an `ousterFileReader` object.

```
ousterReader = ousterFileReader(pcapFileName,calibFileName);
```

Read the 100th point cloud from the Ouster PCAP file.

```
ptCloud = readFrame(ousterReader,100);
```

Check the difference between the values of `CurrentTime` and `StartTime`.

```
disp(ousterReader.CurrentTime - ousterReader.StartTime)
```

```
    9.9977 sec
```

Reset the `ousterFileReader` object.

```
reset(ousterReader);
```

Display the difference between the values of `CurrentTime` and `StartTime`.

```
disp(ousterReader.CurrentTime - ousterReader.StartTime)
```

```
    0 sec
```

Input Arguments

ousterReader — Ouster file reader
ousterFileReader object

Ouster file reader, specified as an ousterFileReader object.

Version History
Introduced in R2022a

See Also
ousterFileReader | hasFrame | readFrame | velodyneFileReader | reset | hesaiFileReader | reset

External Websites
Ouster Product Documentation

hesaiFileReader

Read point cloud data from Hesai PCAP file

Description

The `hesaiFileReader` object reads point cloud data from a Hesai® packet capture (PCAP) file.

Creation

Syntax

```
hesaiReader = hesaiFileReader(fileName,deviceModel)
hesaiReader = hesaiFileReader( ___ ,Name=Value)
```

Description

`hesaiReader = hesaiFileReader(fileName,deviceModel)` creates a `hesaiFileReader` object that reads point cloud data from a Hesai PCAP file. Specify the PCAP file `fileName` and the device model `deviceModel`. The inputs set the `FileName` and `DeviceModel` properties, respectively. The reader supports the Pandar128E3X, Pandar64, PandarQT, and PandarXT32 device models.

`hesaiReader = hesaiFileReader( ___ ,Name=Value)` specify `CalibrationFile` and `SkipPartialFrames` properties using one or more name-value arguments. For example, `hesaiFileReader(fileName,deviceModel,SkipPartialFrames=0)` creates an Hesai file reader that does not skip partial frames.

Properties

FileName — Name of Hesai PCAP file
character vector | string scalar

This property is read-only.

Name of the Hesai PCAP file, specified as a character vector or string scalar.

DeviceModel — Name of Hesai device model
"Pandar128E3X" | "Pandar64" | "PandarQT" | "PandarXT32"

This property is read-only.

Name of the Hesai device model, specified as "Pandar128E3X", "Pandar64", "PandarQT", or "PandarXT32".

Note Specifying the incorrect device model returns no frames or an improperly calibrated point cloud.

CalibrationFile — Name of Hesai calibration CSV file
`''` (default) | character vector | string scalar

This property is read-only.

Name of the Hesai calibration CSV file, specified as a character vector or string scalar. This calibration file is included with every sensor.

To set this property, you must specify it at object creation.

Example: `CalibrationFile='CalibrationFileName'` specifies the Hesai calibration CSV file.

SkipPartialFrames — Partial frame processing
`true` or `1` (default) | `false` or `0`

This property is read-only.

Partial frame processing, specified as a logical `1` (`true`) or `0` (`false`). To skip partial frames, defined as frames with a horizontal field of view less than the mean horizontal field of view of all frames in the PCAP file, specify this property as `true`. Otherwise, specify it as `false`.

To set this property, you must specify it at object creation.

Example: `SkipPartialFrames=true` skips partial frames in the PCAP file.

ReturnMode — Return mode of Hesai PCAP file
character vector

This property is read-only.

Return mode of the Hesai PCAP file, specified as a character vector.

NumberOfFrames — Total number of point cloud frames in file
positive integer

This property is read-only.

Total number of point cloud frames in the file, specified as a positive integer.

Duration — Total duration of file
duration scalar

This property is read-only.

Total duration of the file, specified as a `duration` scalar in seconds.

StartTime — Time of first point cloud reading
duration scalar

This property is read-only.

Time of the first point cloud reading, specified as a `duration` scalar in seconds.

The Hesai sensor sets the start time value relative to the most recent second. For instance, if the file is recorded for 7 minutes from 1:58:00.5 p.m. to 2:05:00.5 p.m., then:

- StartTime = 0.5 s
- EndTime = StartTime + 7 min × 60 s = 420.5 s

EndTime — Time of last point cloud reading
duration scalar

This property is read-only.

Time of the last point cloud reading, specified as a duration scalar in seconds.

The Hesai sensor sets the start time value relative to the most recent second. For instance, if the file is recorded for 7 minutes from 1:58:00.5 p.m. to 2:05:00.5 p.m., then:

- StartTime = 0.5 s
- EndTime = StartTime + 7 min × 60 s = 420.5 s

CurrentTime — Time of current point cloud reading
duration scalar

Time of the current point cloud reading, specified as a duration scalar in seconds. As you read point clouds using readFrame, this property updates with the most recent point cloud reading time. You can use reset to reset the value of this property to the default value. The default value matches the StartTime property.

Timestamps — Start time for each point cloud frame
duration vector

This property is read-only.

Start time for each point cloud frame, specified as a duration vector with values in seconds. The length of the vector is equal to the value of the NumberOfFrames property. The value of the first element in the vector is same as the value of the StartTime property. You can use this property to read point cloud frames captured at different times.

Object Functions

hasFrame   Determine if another Hesai point cloud is available
readFrame   Read Hesai point cloud from file
reset   Reset hesaiFileReader object to first frame

Examples

Read and Visualize Point Clouds from Hesai PCAP File

Download a ZIP file containing a Hesai packet capture (PCAP) file and then unzip the file.

```
zipFile = matlab.internal.examples.downloadSupportFile("lidar","data/hesai_BusyRoad.zip");
saveFolder = fileparts(zipFile);
pcapFileName = [saveFolder filesep 'hesai_BusyRoad.pcap'];
if ~exist(pcapFileName,"file")
    unzip(zipFile,saveFolder);
end
```

Create a hesaiFileReader object.

```
hesaiReader = hesaiFileReader(pcapFileName,"Pandar128E3X");
```

Define X-, Y-, and Z-axes limits for `pcplayer`, in meters.

```
xlimits = [-60 60];
ylimits = [-60 60];
zlimits = [-20 20];
```

Create a point cloud player.

```
player = pcplayer(xlimits,ylimits,zlimits);
```

Set labels for the `pcplayer` axes.

```
xlabel(player.Axes,"X (m)");
ylabel(player.Axes,"Y (m)");
zlabel(player.Axes,"Z (m)");
```

Specify the `CurrentTime` of the Hesai file reader so that it starts reading from 0.3 seconds after the start time.

```
hesaiReader.CurrentTime = hesaiReader.StartTime + seconds(0.3);
```

Display the stream of point clouds from CurrentTime to the final point cloud.

```
while(hasFrame(hesaiReader) && player.isOpen())
    ptCloud = readFrame(hesaiReader);
    view(player,ptCloud(1));
end
```
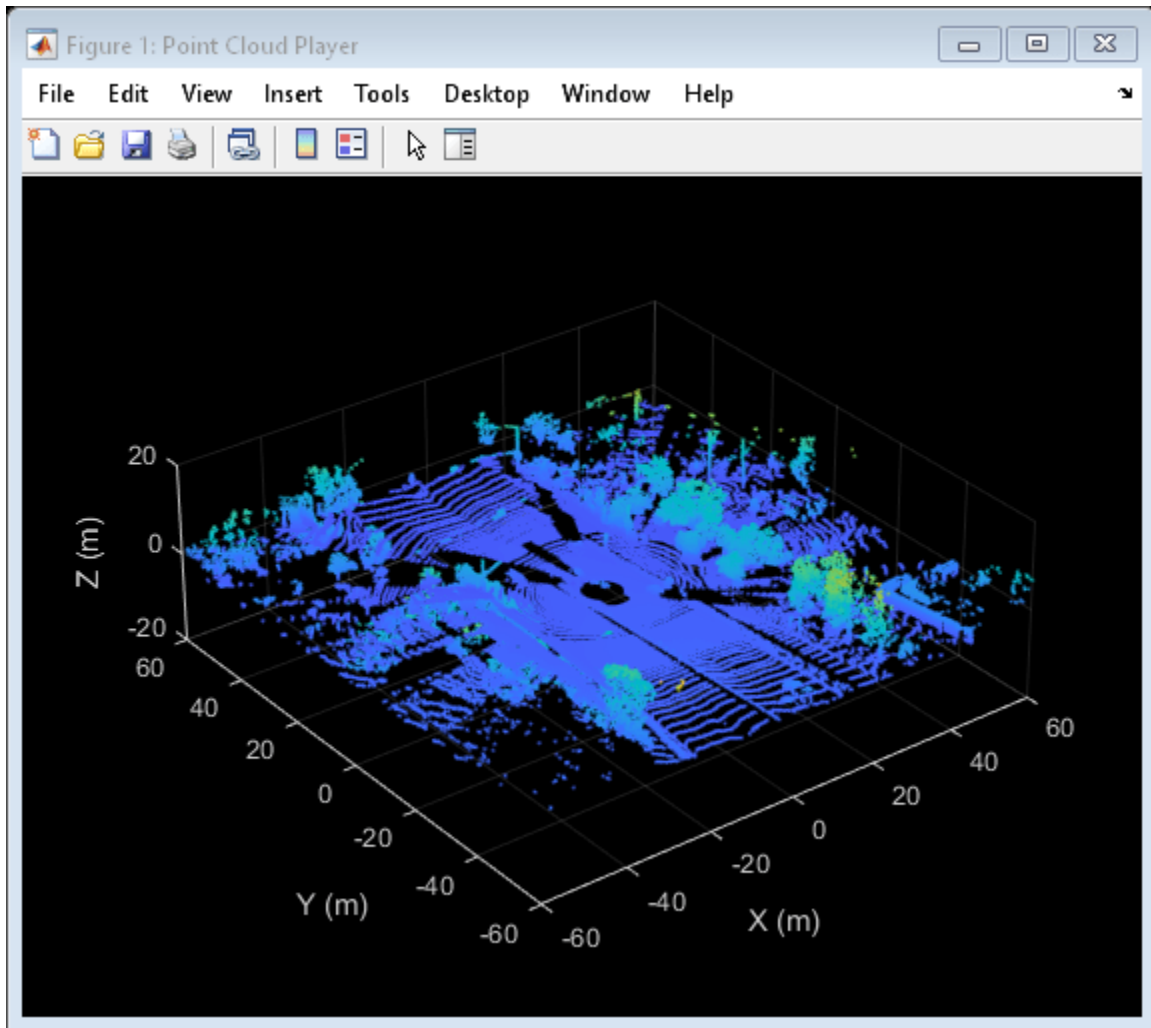
Version History

Introduced in R2022a

See Also

Functions
readFrame | hasFrame | pcshow | pcplayer

Objects
pointCloud | ousterFileReader | velodyneFileReader

External Websites
Hesai Product Documentation

hasFrame

Determine if another Hesai point cloud is available

Syntax

```
isAvailable = hasFrame(hesaiReader)
```

Description

`isAvailable = hasFrame(hesaiReader)` determines if another point cloud is available in the packet capture (PCAP) file of the input Hesai file reader. As you read point clouds using `readFrame`, the point clouds are read sequentially until this function returns `false`.

Examples

Check for Next Point Cloud in Hesai PCAP File

Download a ZIP file containing a Hesai packet capture (PCAP) file and then unzip the file.

```
zipFile = matlab.internal.examples.downloadSupportFile("lidar","data/hesai_BusyRoad.zip");
saveFolder = fileparts(zipFile);
pcapFileName = [saveFolder filesep 'hesai_BusyRoad.pcap'];
if ~exist(pcapFileName,"file")
    unzip(zipFile,saveFolder);
end
```
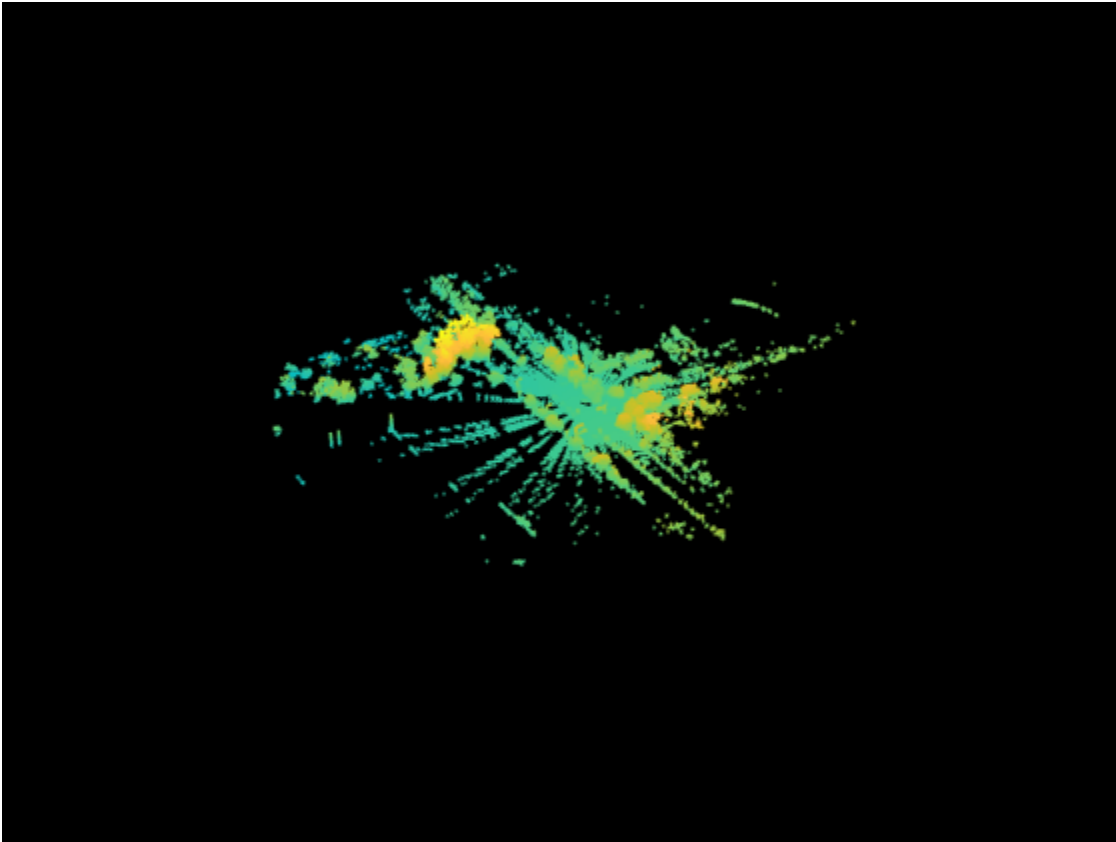
Create a `hesaiFileReader` object.

```
hesaiReader = hesaiFileReader(pcapFileName,"Pandar128E3X");
```

Check if `hesaiReader` has a next point cloud to read.

```
disp(hasFrame(hesaiReader))
```

```
    1
```

Read the last frame in the file.

```
ptCloud = readFrame(hesaiReader,hesaiReader.NumberOfFrames);
```

Check if `hesaiReader` has a next point cloud frame available to read.

```
disp(hasFrame(hesaiReader))
```

```
    0
```

Input Arguments

hesaiReader — Hesai file reader
`hesaiFileReader` object

Hesai file reader, specified as `hesaiFileReader` object.

Output Arguments

isAvailable — Frame is available
true or 1 (default) | false or 0

Frame is available, returned as 1 (`true`) or 0 (`false`). This argument returns `true` if the Hesai file reader contains one or more point cloud frames to read after the current frame. Otherwise, it returns `false`.

Version History
Introduced in R2022a

See Also
hesaiFileReader | readFrame | reset | velodyneFileReader | hasFrame | ousterFileReader | hasFrame | pointCloud

External Websites
Hesai Product Documentation

readFrame

Read Hesai point cloud from file

Syntax

```
ptCloud = readFrame(hesaiReader)
ptCloud = readFrame(hesaiReader,frameNumber)
ptCloud = readFrame(hesaiReader,frameTime)
ptCloud = readFrame( ___ ,ReadMode=readMode)
```

Description

`ptCloud = readFrame(hesaiReader)` reads the next point cloud in sequence from the Hesai PCAP file and returns a `pointCloud` object.

`ptCloud = readFrame(hesaiReader,frameNumber)` reads the point cloud with the specified frame number from the file.

`ptCloud = readFrame(hesaiReader,frameTime)` reads the first point cloud recorded at or after the given `frameTime`.

`ptCloud = readFrame( ___ ,ReadMode=readMode)` reads the points that belong to the specified return mode `readMode`.

Examples

Read Hesai PCAP Point Cloud Using Frame Number

Download a ZIP file containing a Hesai packet capture (PCAP) file and then unzip the file.

```
zipFile = matlab.internal.examples.downloadSupportFile("lidar","data/hesai_BusyRoad.zip");
saveFolder = fileparts(zipFile);
pcapFileName = [saveFolder filesep 'hesai_BusyRoad.pcap'];
if ~exist(pcapFileName,"file")
    unzip(zipFile,saveFolder);
end
```

Create a `hesaiFileReader` object.

```
hesaiReader = hesaiFileReader(pcapFileName,"Pandar128E3X");
```

Read the fifth frame of the Hesai PCAP point cloud data.

```
frameNumber = 5;
ptCloud = readFrame(hesaiReader,frameNumber);
```

Display the point cloud.

```
pcshow(ptCloud(1))
```

Read Hesai PCAP Point Cloud Using Time Duration

Download a ZIP file containing a Hesai packet capture (PCAP) file and then unzip the file.

```
zipFile = matlab.internal.examples.downloadSupportFile("lidar","data/hesai_BusyRoad.zip");
saveFolder = fileparts(zipFile);
pcapFileName = [saveFolder filesep 'hesai_BusyRoad.pcap'];
if ~exist(pcapFileName,"file")
    unzip(zipFile,saveFolder);
end
```

Create a `hesaiFileReader` object.

```
hesaiReader = hesaiFileReader(pcapFileName,"Pandar128E3X");
```

Read the first Hesai PCAP point cloud frame from 3 seconds after the `StartTime`.

```
frameTime = hesaiReader.StartTime + seconds(3);
ptCloud = readFrame(hesaiReader,frameTime);
```
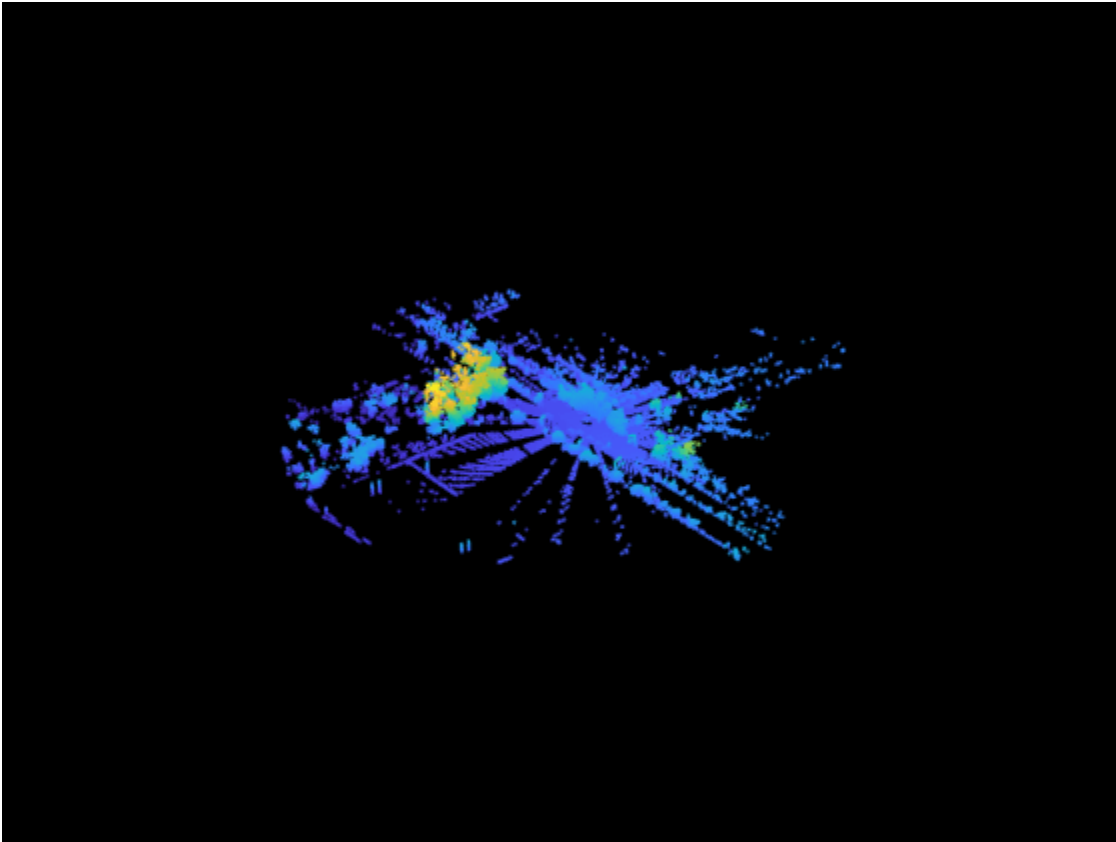
Display the point cloud.

```
pcshow(ptCloud(1))
```

Input Arguments

hesaiReader — Hesai file reader
hesaiFileReader object

Hesai file reader, specified as a hesaiFileReader object.

frameNumber — Frame number of desired point cloud in file
positive integer

Frame number of the desired point cloud in the file, specified as a positive integer. Frame numbers are sequential.

frameTime — Frame time of desired point cloud in file
duration scalar

Frame time of the desired point cloud in the file, specified as a duration scalar in seconds. The function returns the first frame available at or after the specified frameTime.

readMode — Mode for reading data from Hesai PCAP file
"hesaiReader.ReturnMode" (default) | character vector | cell array of character vectors | string scalar | string array

Mode for reading data from the Hesai PCAP file, specified as a character vector, cell array of character vectors, string scalar, or a string array. This value must be a subset of the `ReturnMode` property of the `hesaiFileReader` object.

If you specify this argument as a cell array of character vectors or string array, then this function returns a point cloud array containing point clouds that belong to the specified modes in the same order.

Example: ReadMode="hesaiReader.ReturnMode" specifies the return mode to read point cloud data.

Output Arguments

`ptCloud` — Point cloud
`pointCloud` object | array of `pointCloud` objects

Point cloud, returned as a `pointCloud` object or array of `pointCloud` objects. The function returns an array of `pointCloud` objects only when you specify `readMode` as a cell array of character vectors or string array. The order of the `pointCloud` objects in the array corresponds to the order of the specified read modes in the `readMode` argument.

Version History
Introduced in R2022a

See Also
`hesaiFileReader` | `hasFrame` | `reset` | `velodyneFileReader` | `readFrame` | `ousterFileReader` | `readFrame` | `pointCloud` | `pcshow`

External Websites
Hesai Product Documentation

reset

Reset `hesaiFileReader` object to first frame

Syntax

```
reset(hesaiReader)
```

Description

`reset(hesaiReader)` resets Hesai file reader object `hesaiReader` to the first frame by resetting its `CurrentTime` property to the default value. The default value is the value of the `StartTime` property of `hesaiReader`.

Examples

Reset `hesaiFileReader` Object

Download a ZIP file containing a Hesai packet capture (PCAP) file and then unzip the file.

```
zipFile = matlab.internal.examples.downloadSupportFile("lidar","data/hesai_BusyRoad.zip");
saveFolder = fileparts(zipFile);
pcapFileName = [saveFolder filesep 'hesai_BusyRoad.pcap'];
if ~exist(pcapFileName,"file")
    unzip(zipFile,saveFolder);
end
```

Create a `hesaiFileReader` object.

```
hesaiReader = hesaiFileReader(pcapFileName,"Pandar128E3X");
```

Read the 100th point cloud from the Hesai PCAP file.

```
ptCloud = readFrame(hesaiReader,100);
```

Check the difference between the values of `CurrentTime` and `StartTime`.

```
disp(hesaiReader.CurrentTime - hesaiReader.StartTime)
```

```
   10 sec
```

Reset the `hesaiFileReader` object.

```
reset(hesaiReader);
```

Display the difference between the values of `CurrentTime` and `StartTime`.

```
disp(hesaiReader.CurrentTime - hesaiReader.StartTime)
```

```
   0 sec
```

Input Arguments

hesaiReader — Hesai file reader
hesaiFileReader object

Hesai file reader, specified as a hesaiFileReader object.

Version History
Introduced in R2022a

See Also
hesaiFileReader | readFrame | hasFrame | velodyneFileReader | reset | ousterFileReader | reset

External Websites
Hesai Product Documentation

Functions

pcsemanticseg

Point cloud semantic segmentation using deep learning

Syntax

```
C = pcsemanticseg(pc,network)
[C,score,allScores] = pcsemanticseg(pc,network)

plds = pcsemanticseg(ds,network)

[ ___ ] = pcsemanticseg( ___ ,Name=Value)
```

Description

`C = pcsemanticseg(pc,network)` performs the semantic segmentation results on the input point cloud using deep learning and returns the results.

`[C,score,allScores] = pcsemanticseg(pc,network)` additionally returns the classification score `score` for each predicted label in `C` and the scores of all the label categories that the network can classify `allScores`.

`plds = pcsemanticseg(ds,network)` returns the semantic segmentation results for a collection of point clouds in a datastore object `ds`.

The function supports parallel computing using multiple MATLAB workers. You can enable parallel computing using the "Computer Vision Toolbox Preferences" dialog box.

`[ ___ ] = pcsemanticseg( ___ ,Name=Value)` specifies options using one or more name-value arguments. For example, `OutputType="double"` returns the segmentation results as numeric array of data type `double`.

Examples

Semantically Segmentat Organized Point Cloud Using Deep Learning

Load a pretrained network into the workspace. This network segments vehicles from the input point cloud.

```
data = load("pointCloudVehicleSegmentationNetwork.mat");
net = data.net

net =
  DAGNetwork with properties:

        Layers: [80×1 nnet.cnn.layer.Layer]
    Connections: [87×2 table]
     InputNames: {'Organized-Point-Cloud-Input'}
    OutputNames: {'Loss'}
```

Load the test point cloud data.

```
dataDir = fullfile(toolboxdir("lidar"),"lidardata", ...
        "sampleWPIPointClouds","pointClouds","010.pcd");
```

Read and display the test point cloud.

```
ptCloud = pcread(dataDir);
figure
pcshow(ptCloud.Location)
```
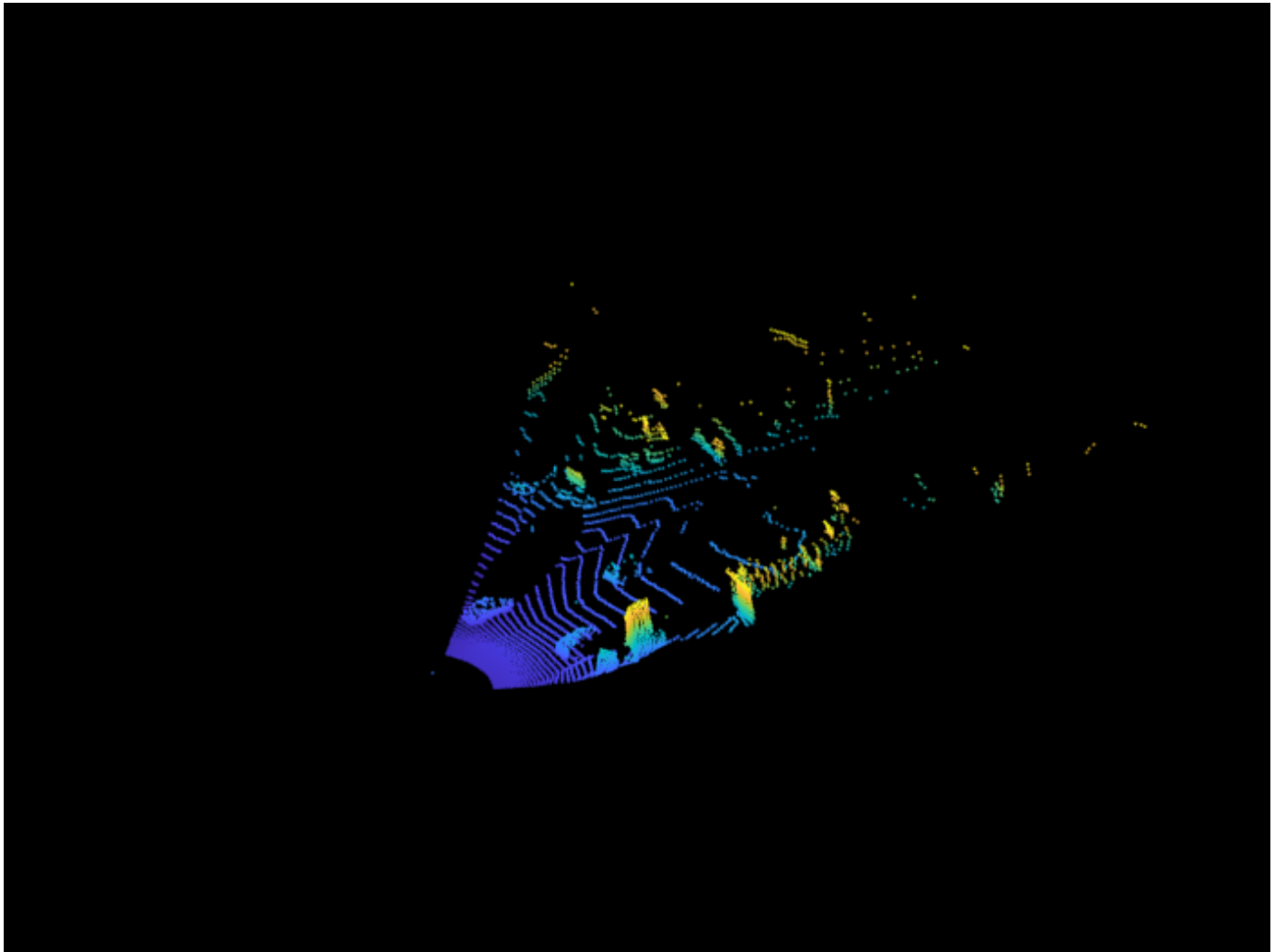

Display the intensity channel of the point cloud.
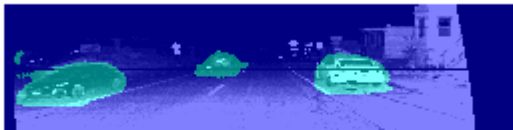
```
figure
imshow(uint8(ptCloud.Intensity))
```

Preprocess the point cloud data and perform semantic segmentation.

```
pc = cat(3,ptCloud.Location,ptCloud.Intensity);
[C,scores] = pcsemanticseg(pc,net);
```

Overlay the segmentation results on the intensity channel and display the results.

```
B = labeloverlay(uint8(ptCloud.Intensity),C);
figure
imshow(B)
```


Evaluate Semantic Segmentation of Point Cloud Test Set

Load a pretrained network into the workspace. This network segments vehicles from the input point cloud.

```
data = load("pointCloudVehicleSegmentationNetwork.mat");
net = data.net;
```

Load the test point cloud data, and create a file datastore.

```
dataDir = fullfile(toolboxdir("lidar"),"lidardata","sampleWPIPointClouds");
testDataDir = fullfile(dataDir,"pointClouds");
ptCloudDs = fileDatastore(testDataDir,ReadFcn=@pcread);
```

Concatenate the intensity channel values to the datastore.

```
pcds = transform(ptCloudDs,@(ptCloud)cat(3,ptCloud.Location,ptCloud.Intensity));
```

Load the ground truth labels.

```
testLabelDir = fullfile(dataDir,"segmentationLabels");
pldsTruth = fileDatastore(testLabelDir,ReadFcn=@(x)load(x).labels);
```

Perform semantic segmentation on all the test point clouds.

```
pldsResults = pcsemanticseg(pcds,net,WriteLocation=tempdir);

Running semantic segmentation network
-------------------------------------
* Processed 10 point clouds.
```

Compare the results against the ground truth labels.

```
metrics = evaluateSemanticSegmentation(pldsResults,pldsTruth)

Evaluating semantic segmentation results
----------------------------------------
* Selected metrics: global accuracy, class accuracy, IoU, weighted IoU, BF score.
* Processed 10 images.
* Finalizing... Done.
* Data set metrics:

    GlobalAccuracy    MeanAccuracy    MeanIoU    WeightedIoU    MeanBFScore
    _____    _____    _____    _____    _____

       0.94751           0.9723        0.67226     0.92655         0.47675

metrics =
  semanticSegmentationMetrics with properties:

              ConfusionMatrix: [2×2 table]
    NormalizedConfusionMatrix: [2×2 table]
                DataSetMetrics: [1×5 table]
                 ClassMetrics: [2×3 table]
                 ImageMetrics: [10×5 table]
```

Input Arguments

pc — Input point cloud
numeric matrix | 3-D numeric array | 4-D numeric array

Input point cloud, specified as one of these options.

Point Cloud Type	Data Format
Unorganized point cloud	Numeric matrix of size M-by-K, where M is the number of points in the point cloud, and K is the number of channels, such as intensity and color.
Array of unorganized point clouds	3-D numeric array of size M-by-K-by-P, where P is the number of point clouds.
Organized point cloud	3-D numeric array of size M-by-N-by-K, where M and N are the rows and columns in the point cloud, respectively, and K is the number of channels, such as intensity and color.

Point Cloud Type	Data Format
Array of organized point clouds	4-D numeric array of size *M*-by-*N*-by-*K*-by-*P*, where *P* is the number of point clouds.

The input point cloud can also be a `gpuArray` that contains one of the point cloud types listed in the table (requires Parallel Computing Toolbox).

Data Types: `double` | `single`

`network` — Network
SeriesNetwork object | DAGNetwork object | dlnetwork object

Network, specified as a `SeriesNetwork`, `DAGNetwork`, or `dlnetwork` object.

`ds` — Collection of point clouds
datastore object

Collection of point clouds, specified as a datastore object. The `read` function of the datastore must return a numeric array or cell array. For a cell array, the data in each cell must be a numeric array. For cell arrays with multiple columns, the function processes only the first column.

For more information, see "Datastores for Deep Learning" (Deep Learning Toolbox).

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `pcsemanticseg(pc,network,OutputType="double")` returns the segmentation results as numeric array of data type `double`.

`OutputType` — Returned segmentation type
"categorical" (default) | "double" | "uint8"

Returned segmentation type, specified as `"categorical"`, `"double"`, or `"uint8"`.

When you specify this value as `"categorical"`, the function returns the segmentation labels as a categorical array.

When you specify this value as `"double"` or `"uint8"`, the function returns the segmentation results as a numeric array of the corresponding data type containing label IDs. The IDs are integer values corresponding to the class names defined in the classification layer of the input network.

Note You cannot use the `OutputType` argument with a datastore input.

Data Types: `char` | `string`

`MiniBatchSize` — Size of point cloud groups
32 (default) | positive integer

Size of the point cloud groups, specified as an integer. For a large collection of point clouds, the function groups and processes the point clouds together as a batch. Increasing the `MiniBatchSize` value improves the computational efficiency, but requires more memory.

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

ExecutionEnvironment — Hardware resource
`"auto"` (default) | `"gpu"` | `"cpu"`

Hardware resource for processing the point clouds, specified as `"auto"`, `"gpu"`, or `"cpu"`.

Execution Environment	Description
`"auto"`	Use a GPU, if available. Otherwise, use the CPU. Using a GPU requires Parallel Computing Toolbox and a CUDA®-enabled NVIDIA® GPU. For information about the supported capabilities, see "GPU Computing Requirements" (Parallel Computing Toolbox).
`"gpu"`	Use the GPU. If a suitable GPU is not available, the function returns an error message.
`"cpu"`	Use the CPU.

Data Types: `char` | `string`

Acceleration — Performance optimization
`"auto"` (default) | `"mex"` | `"none"`

Performance optimization, specified as `"auto"`, `"mex"`, or `"none"`.

Acceleration	Description
`"auto"`	Automatically apply a number of optimizations suitable for the input network and hardware resource. This is the default option. MATLAB does not generate a MEX function with this option.
`"none"`	Disable all acceleration.

`"auto"` option can offer performance benefits, but at the expense of an increased initial run time. Subsequent calls with compatible parameters are faster. Use performance optimization when you plan to call the function multiple times using new input data.

Data Types: `char` | `string`

Classes — Classes into which points are classified
`"auto"` (default) | cell array of character vectors | vector of strings | categorical vector

Classes into which points are classified, specified as `"auto"`, a cell array of character vectors, a vector of strings, or a categorical vector. If the value is a categorical vector, Y, then the function sorts and orders the elements of the vector according to `categories(Y)`.

If the network is a `dlnetwork` object, then the number of classes specified by `Classes` must match the number of channels in the output of the network predictions. By default, when the value of `Classes` is `"auto"`, the function numbers the classes from 1 through *K*, where *K* is the number of channels in the output layer of the network.

If the network is a `SeriesNetwork` or `DAGNetwork` object, then the number of classes specified by `Classes` must match the number of classes in the classification output layer. By default, when the

value of `Classes` is `"auto"`, the function sets the classes automatically using the classification output layer.

WriteLocation — Folder to write to
pwd (default) | string scalar | character vector

Folder to write to, specified as `pwd`, a string scalar, or a character vector. The specified folder must exist and have write permissions.

Note This argument is applicable only when you specify the input point clouds using a datastore.

Data Types: `char` | `string`

NamePrefix — Prefix for output file names
"pointLabel" (default) | string scalar | character vector

Prefix for the output file names, specified as a string scalar or character vector. The function returns the point cloud files as:

- `NamePrefix_N.png`, where *N* is the index of the corresponding point cloud file in `pointCloudDs.Files`.

Note This argument is applicable only when you specify the input point clouds using a datastore.

Data Types: `char` | `string`

Verbose — Display progress information
true or 1 (default) | false or 0

Display progress information, specified as logical `1` (`true`) or `0` (`false`).

Note This argument is applicable only when you specify the input point clouds using a datastore.

Data Types: `logical`

UseParallel — Run parallel computations
false or 0 (default) | true or 1

Run parallel computations, specified as a logical `1` (`true`) or `0` (`false`).

To run in parallel, set `UseParallel` to `true`, or enable this by default using the Computer Vision Toolbox™ preferences.

For more information, see "Parallel Computing Toolbox Support".

Note This argument is applicable only when you specify the input point clouds using a datastore.

Data Types: `logical`

Output Arguments

C — Segmentation labels
categorical vector | categorical matrix | categorical array

Segmentation labels, returned as a categorical array. The argument contains labels for all points in the input point cloud.

Input Point Cloud	Semantic Labels
Unorganized point cloud	Numeric vector with M elements. Element C(i) is the categorical label assigned to the point pc(i) in the input point cloud.
Array of unorganized point clouds	Numeric matrix of size M-by-P. Element C(i,k) is the categorical label assigned to the ith point in the kth point cloud of the point cloud array.
Organized point cloud	Numeric matrix of size M-by-N. Element C(i,j) is the categorical label assigned to the point pc(i,j) in the input point cloud.
Array of organized point clouds	3-D numeric array of size M-by-N-by-P. Element C(i,j,k) is the categorical label assigned to the point pc(i,j) in the kth point cloud of the input point cloud array.

score — Confidence scores
numeric vector | numeric matrix | numeric array

Confidence scores for each categorical label in C, returned as a vector, matrix, or array of values between 0 and 1. The scores represents the confidence in the corresponding predicted label in C. Higher score values indicate a higher confidence in the predicted label.

Input Point Cloud	Semantic Labels
Unorganized point cloud	Numeric vector with M elements. Element score(i) is the score assigned to the point pc(i) in the input point cloud.
Array of unorganized point clouds	Numeric matrix of size M-by-P. Element score(i,k) is the score assigned to the ith point in the kth point cloud of the point cloud array.
Organized point cloud	Numeric matrix of size M-by-N. Element score(i,j) is the score assigned to the point pc(i,j) in the input point cloud.
Array of organized point clouds	3-D numeric array of size M-by-N-by-P. Element score(i,j,k) is the score assigned to the point pc(i,j) in the kth point cloud of the input point cloud array.

allScores — Scores for all label categories
numeric matrix | 3-D numeric array | 4-D numeric array

Scores for all label categories the input network can classify, returned as a numeric array.

This table shows the format of this output for each type of point cloud input. L is the total number of label categories.

Input Point Cloud	Semantic Labels
Unorganized point cloud	Numeric matrix of size M-by-L. Element `allScores`(i,q) is the score of the qth label at the point `pc`(i) in the input point cloud.
Array of unorganized point clouds	3-D numeric array of size M-by-L-by-P. Element `allScores`(i,q,k) is the score of the qth label at the point `pc`(i) in the kth point cloud of the point cloud array.
Organized point cloud	3-D numeric array of size M-by-N-by-L. Element `allScores`(i,j,q) is the score of the qth label at the point `pc`(i,j) in the input point cloud.
Array of organized point clouds	4-D numeric array of size M-by-N-by-L-by-P. Element `allScores`(i,j,q,k) is the score of the qth label at the point `pc`(i, j) in the kth point cloud of the input point cloud array.

`plds` — Semantic segmentation results
`fileDatastore` object

Semantic segmentation results, returned as a `fileDatastore` object. The function saves the segmentation result of each point cloud as a MAT file. You can use the `read` function on this output to obtain the categorical labels for the point clouds in `ds`.

Version History
Introduced in R2022b

See Also
semanticseg | segmentAerialLidarVegetation | segmentAerialLidarBuildings | pcsemanticseg | evaluateSemanticSegmentation | squeezesegv2Layers | pointnetplusLayers | pointCloudInputLayer

Topics
"Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning"
"Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network"
"Getting Started with Point Clouds Using Deep Learning"
"List of Deep Learning Layers" (Deep Learning Toolbox)
"Datastores for Deep Learning" (Deep Learning Toolbox)

segmentAerialLidarBuildings

Segment building points from aerial lidar data

Syntax

```
buildingPtsIdx = segmentAerialLidarBuildings(ptCloud)
[buildingPtsIdx,nonBuildingPtCloud,buildingPtCloud] =
segmentAerialLidarBuildings(ptCloud)
[ ___ ] = segmentAerialLidarBuildings( ___ ,ExecutionEnvironment=env)
```

Description

`buildingPtsIdx = segmentAerialLidarBuildings(ptCloud)` segments the building points and non-building points from the input unorganized point cloud `ptCloud` using a pretrained PointNet ++ model and returns the building point indices..

Note This function requires Deep Learning Toolbox™.

`[buildingPtsIdx,nonBuildingPtCloud,buildingPtCloud] = segmentAerialLidarBuildings(ptCloud)` additionally returns the building points and non-building points as individual `pointCloud` objects.

`[ ___ ] = segmentAerialLidarBuildings( ___ ,ExecutionEnvironment=env)` specifies the execution environment for the function in addition to any combination of arguments from previous syntaxes.

Examples

Segment Buildings from Aerial Lidar Data

Specify a LAZ file that contains aerial lidar data.

```
fileName = fullfile(toolboxdir("lidar"),"lidardata","las", ...
            "aerialLidarData.laz");
```

Read point cloud data from the LAZ file into the workspace.

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

Segment the building points from the point cloud.

```
[~,nonBuildingPtCloud,buildingPtCloud] = segmentAerialLidarBuildings(ptCloud);
```

Visualize the building and non-building points.

```
figure
pcshowpair(buildingPtCloud,nonBuildingPtCloud)
```
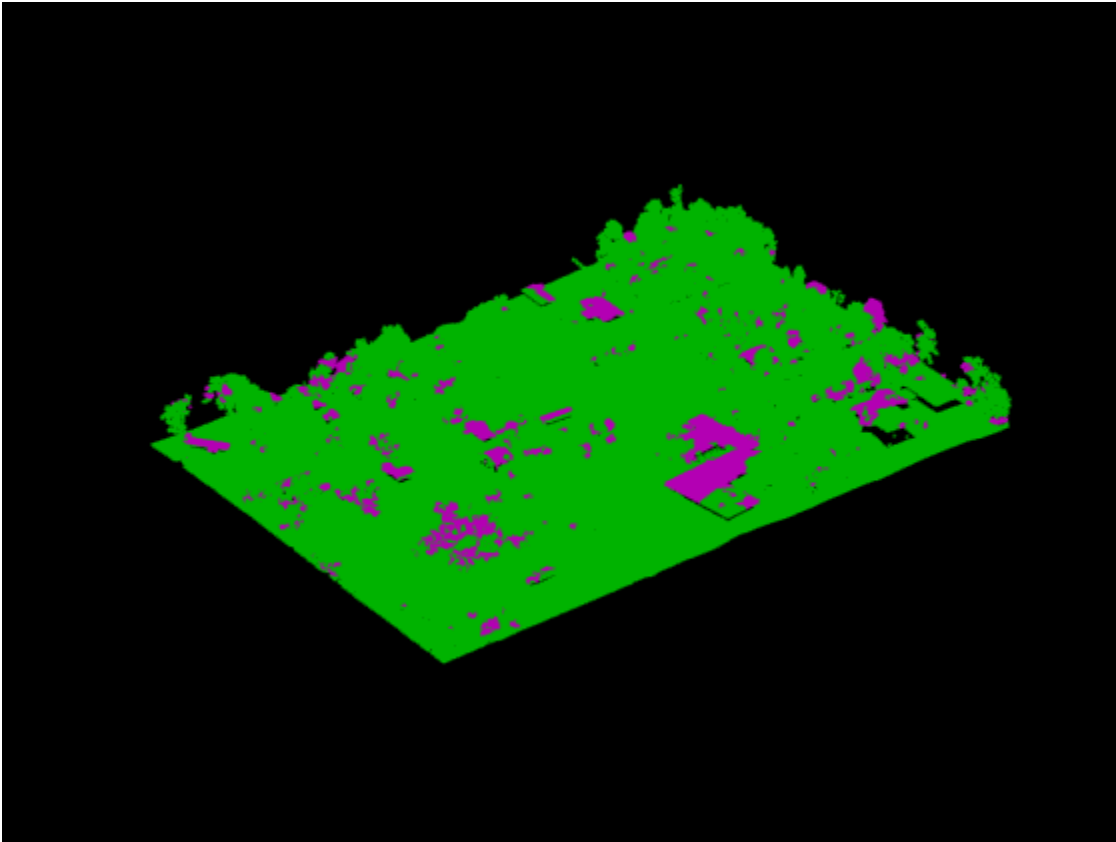
Input Arguments

`ptCloud` — Unorganized point cloud
`pointCloud` object

Unorganized point cloud data, specified as a `pointCloud` object.

`env` — Hardware resource
`"auto"` (default) | `"gpu"` | `"cpu"`

Hardware resource to use to process the point cloud, specified as one of these options.

- `"auto"` — Use a GPU if available. Otherwise, use the CPU. The use of a GPU requires Parallel Computing Toolbox and a CUDA-enabled NVIDIA GPU. For information about the supported compute capabilities, see "GPU Computing Requirements" (Parallel Computing Toolbox).
- `"gpu"` — The function runs on a GPU. If a suitable GPU is not available, the function returns an error message.
- `"cpu"` — The function runs on a CPU.

Data Types: `char` | `string`

Output Arguments

buildingPtsIdx — Binary map of segmented point cloud
M-element logical vector

Binary map of the segmented point cloud, returned as an *M*-element logical vector. Elements that correspond to building points in the point cloud are `true`, and non-building points are `false`.

nonBuildingPtCloud — Point cloud of non-building points
pointCloud object

Point cloud of non-building points, returned as a `pointCloud` object.

buildingPtCloud — Point cloud of building points
pointCloud object

Point cloud of building points, returned as a `pointCloud` object.

Version History

Introduced in R2022b

See Also

pcsemanticseg | segmentAerialLidarVegetation | segmentGroundSMRF | segmentLidarData

segmentAerialLidarVegetation

Segment vegetation points from aerial lidar data

Syntax

```
vegetationPtsIdx = segmentAerialLidarVegetation(ptCloud)
[vegetationPtsIdx,nonVegetationPtCloud,vegetationPtCloud] =
segmentAerialLidarVegetation(ptCloud)
[ ___ ] = segmentAerialLidarVegetation( ___ ,ExecutionEnvironment=env)
```

Description

`vegetationPtsIdx = segmentAerialLidarVegetation(ptCloud)` segments the vegetation points and non-vegetation points from the input unorganized point cloud `ptCloud` using a pretrained PointNet++ model and returns the building point indices.

Note This function requires Deep Learning Toolbox.

`[vegetationPtsIdx,nonVegetationPtCloud,vegetationPtCloud] = segmentAerialLidarVegetation(ptCloud)` additionally returns the vegetation points and non-vegetation points as individual `pointCloud` objects.

`[ ___ ] = segmentAerialLidarVegetation( ___ ,ExecutionEnvironment=env)` specifies the execution environment for the function in addition to any combination of arguments from previous syntaxes.

Examples

Segment Vegetation from Aerial Lidar Data

Specify a LAZ file that contains aerial lidar data.

```
fileName = fullfile(toolboxdir("lidar"),"lidardata","las", ...
            "aerialLidarData.laz");
```

Read point cloud data from the LAZ file into the workspace.

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

Segment the vegetation points from the point cloud.

```
[~,nonVegetationPtCloud,vegetationPtCloud] = segmentAerialLidarVegetation(ptCloud);
```

Visualize the vegetation and non-vegetation points.

```
figure
pcshowpair(vegetationPtCloud,nonVegetationPtCloud)
```
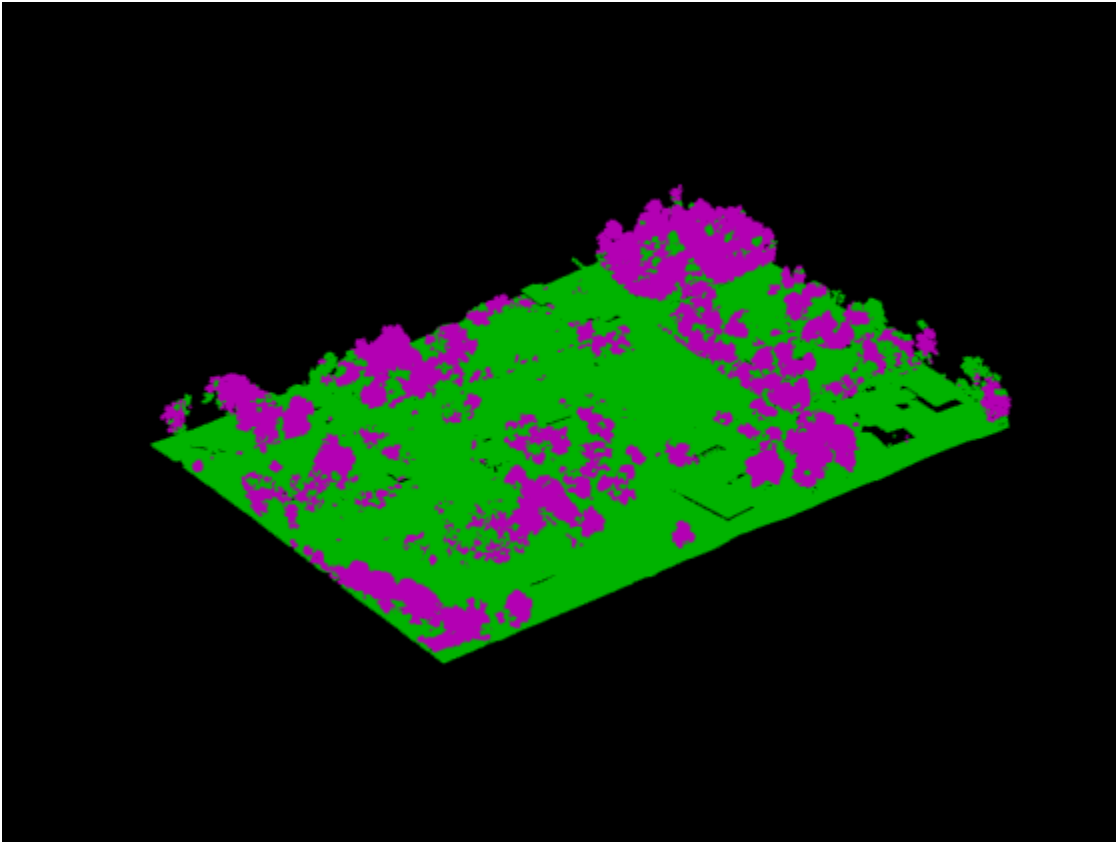
Input Arguments

`ptCloud` — Unorganized point cloud
`pointCloud` object

Unorganized point cloud data, specified as a `pointCloud` object.

`env` — Hardware resource
`"auto"` (default) | `"gpu"` | `"cpu"`

Hardware resource to use to process the point cloud, specified as one of these options.

- `"auto"` — Use a GPU if available. Otherwise, use the CPU. The use of a GPU requires Parallel Computing Toolbox and a CUDA-enabled NVIDIA GPU. For information about the supported compute capabilities, see "GPU Computing Requirements" (Parallel Computing Toolbox).
- `"gpu"` — The function uses a GPU. If a suitable GPU is not available, the function returns an error message.
- `"cpu"` — The function uses a CPU.

Data Types: `char` | `string`

Output Arguments

vegetationPtsIdx — Binary map of segmented point cloud
logical vector

Binary map of the segmented point cloud, returned as an *M*-element logical vector. Elements that correspond to vegetation points in the point cloud are `true`, and non-vegetation points are `false`.

nonVegetationPtCloud — Point cloud of non-vegetation points
`pointCloud` object

Point cloud of non-vegetation points, returned as a `pointCloud` object.

vegetationPtCloud — Point cloud of vegetation points
`pointCloud` object

Point cloud of vegetation points, returned as a `pointCloud` object.

Version History
Introduced in R2022b

See Also
pcsemanticseg | segmentAerialLidarBuildings | segmentGroundSMRF | segmentLidarData

pcregisterfgr

Register two point clouds using FGR algorithm

Syntax

```
tform = pcregisterfgr(moving,fixed,gridSize)
[tform,rmse] = pcregisterfgr(moving,fixed,gridSize)
[ ___ ] = pcregisterfgr( ___ ,MaxIterations=numIterations)
```

Description

`tform = pcregisterfgr(moving,fixed,gridSize)` returns a rigid transformation that registers a moving point cloud to a fixed point cloud.

The function registers points using a fast global registration (FGR) algorithm based on FPFH features.

`[tform,rmse] = pcregisterfgr(moving,fixed,gridSize)` additionally returns the root mean square error of the Euclidean distance between the inlier aligned points.

`[ ___ ] = pcregisterfgr( ___ ,MaxIterations=numIterations)` specifies the maximum number of iterations for the FGR algorithm, in addition to any combination of arguments from previous syntaxes.

Examples

Align Two Point Clouds Using FGR Algorithm

Load point cloud data into the workspace.

```
ld = load("livingRoom.mat");
fixed = ld.livingRoomData{1};
moving = ld.livingRoomData{2};
```

To improve the efficiency and the accuracy of the FGR registration algorithm, downsample the point clouds. Display the point clouds.

```
fixedDownsampled = pcdownsample(fixed,gridAverage=0.05);
movingDownsampled = pcdownsample(moving,gridAverage=0.05);
pcshowpair(moving,fixed,VerticalAxis="Y",VerticalAxisDir="Down")
```
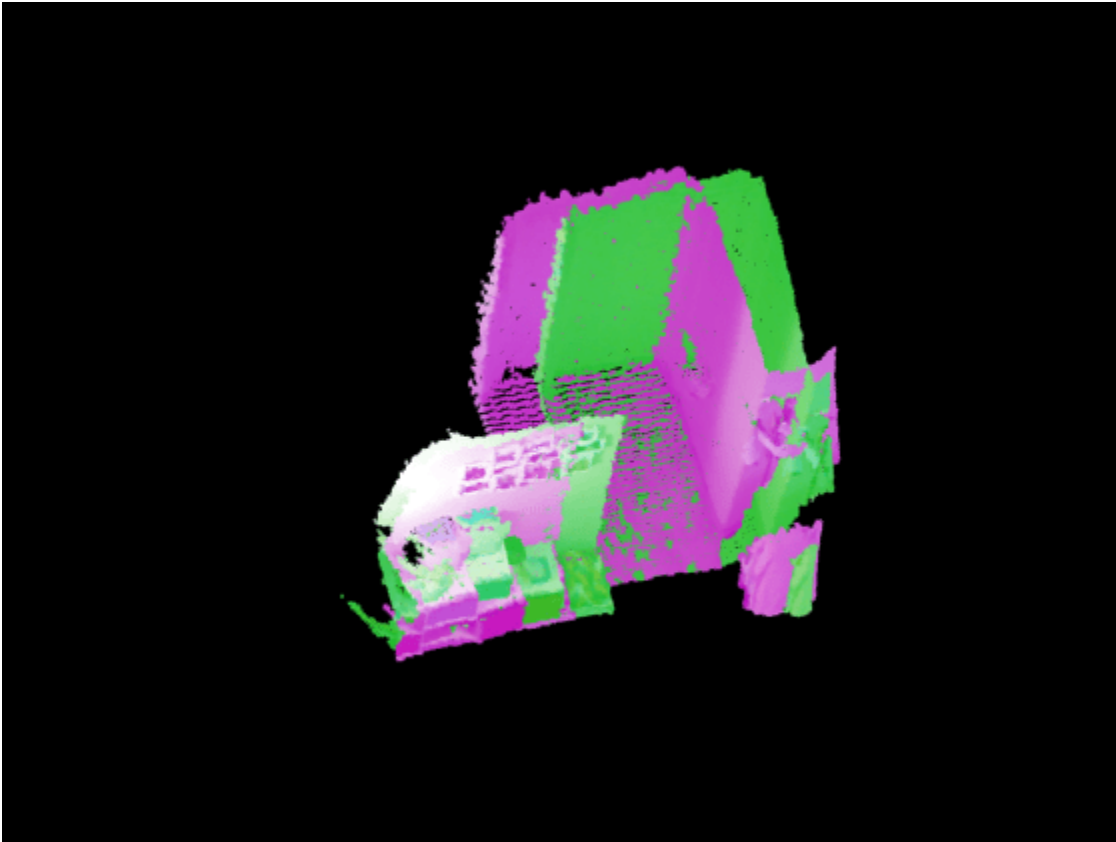
Perform registration using the FGR algorithm.

```
gridSize = 0.1;
[tform,rmse] = pcregisterfgr(movingDownsampled,fixedDownsampled, ...
                             gridSize,MaxIterations=100);
```

Visualize the alignment.

```
movingRegistered = pctransform(moving,tform);
pcshowpair(movingRegistered,fixed,VerticalAxis="Y",VerticalAxisDir="Down");
```
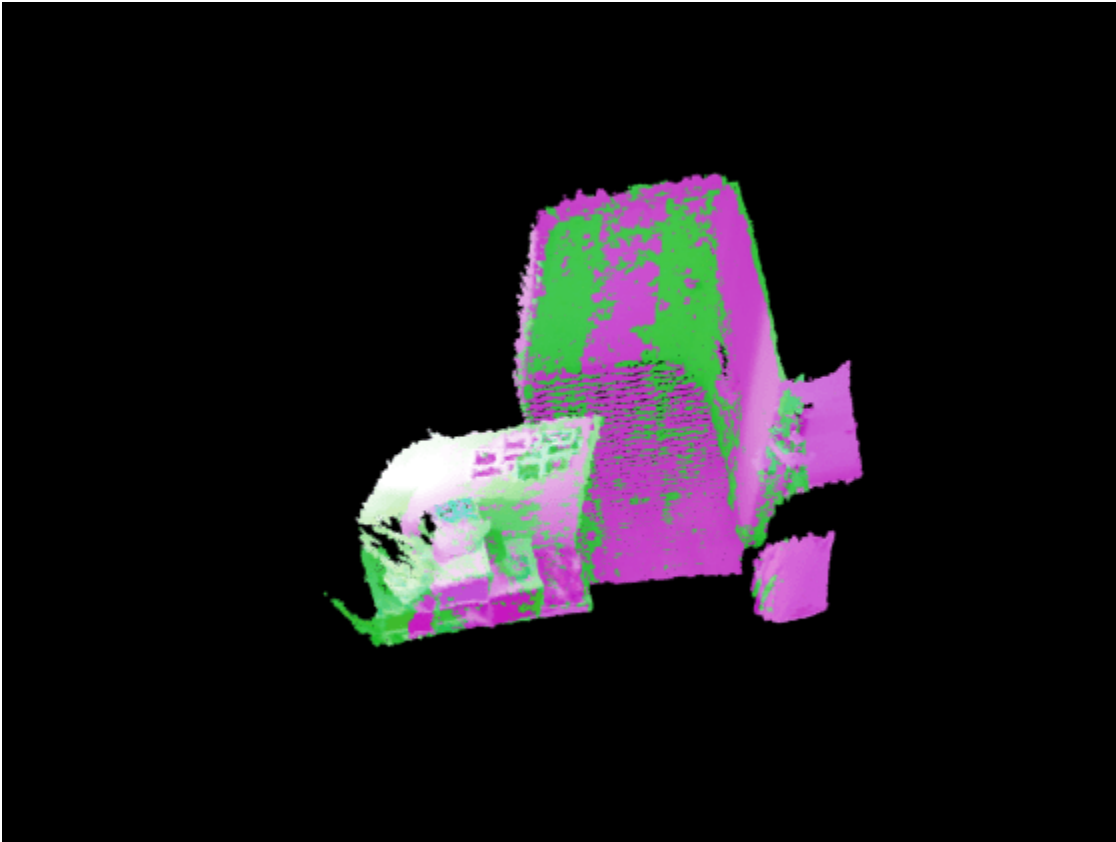
Merge the point clouds and visualize the merged result.

```
mergeSize = 0.01;
merged = pcmerge(movingRegistered,fixed,mergeSize);
figure(Name = "merged point cloud after Fast Global registration");
pcshow(merged,VerticalAxis="Y",VerticalAxisDir ="Down");
```
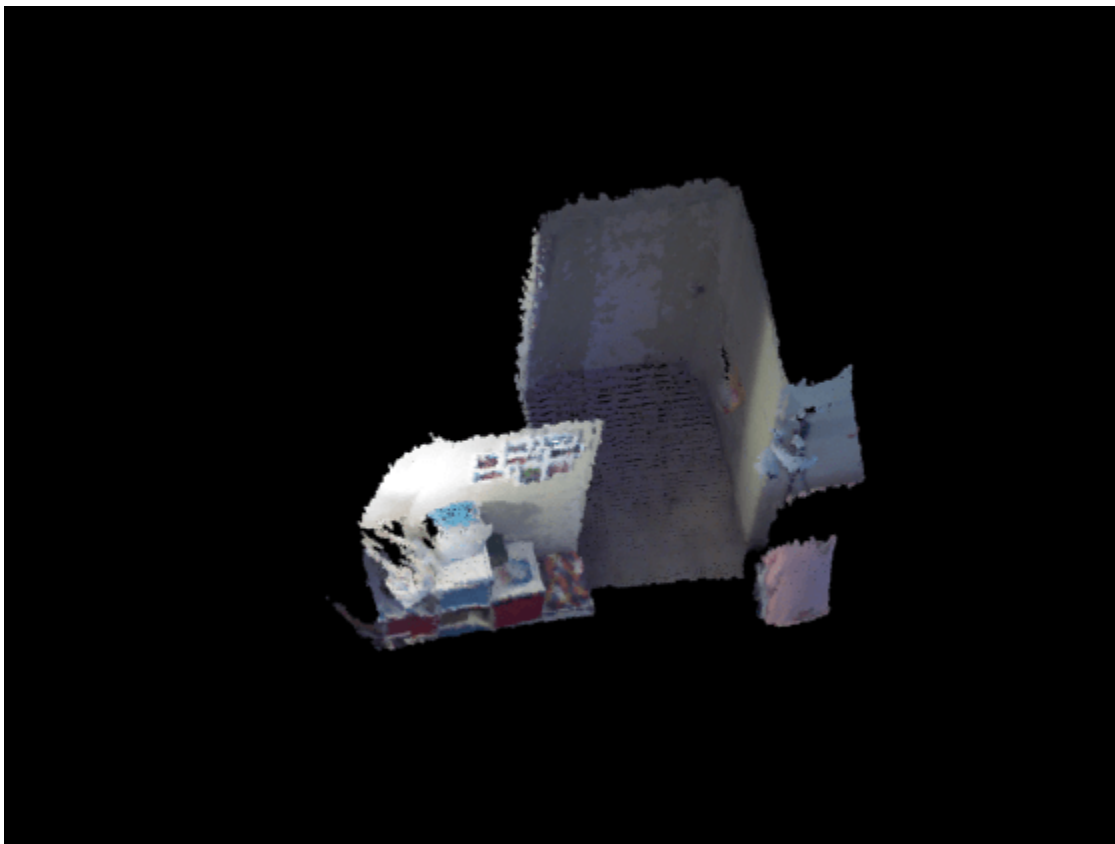
Input Arguments

moving — Moving point cloud
pointCloud object

Moving point cloud, specified as a pointCloud object.

fixed — Fixed point cloud
pointCloud object

Fixed point cloud, specified as a pointCloud object.

gridSize — Grid size to search for correspondence between point clouds
positive scalar

Grid size to search for correspondence between the point clouds, specified as a positive scalar.

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

numIterations — Maximum number of iterations
64 (default) | positive integer

Maximum number of iterations before the FGR algorithm stops, specified as a positive integer.

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

Output Arguments

`tform` — Rigid transformation
`rigidtform3d` object

Rigid transformation, returned as a `rigidtform3d` object. The rigid transformation registers a moving point cloud to a fixed point cloud. The `rigidtform3d` object describes the rigid 3-D transform. The FGR algorithm estimates the rigid transformation between the moving and fixed point clouds.

`rmse` — Root mean square error
positive scalar

Root mean square error, returned as a positive scalar that represents the Euclidean distance between the inlier aligned points. A lower error values indicates a better registration.

Tips

- To improve the accuracy and the efficiency of registration, downsample the point clouds using the `pcdownsample` function before using the `pcregisterfgr` function.
- For ground vehicle point clouds, you can improve performance and accuracy by removing the ground using `pcfitplane` or `segmentGroundFromLidarData` before registration. For details on how to do this, see the `helperProcessPointCloud` function in the "Build a Map from Lidar Data" (Automated Driving Toolbox) example.
- To merge more than two point clouds, you can use the `pccat` function instead of the `pcmerge` function.

Algorithms

This figure shows the workflow of point cloud registration using the FGR algorithm.
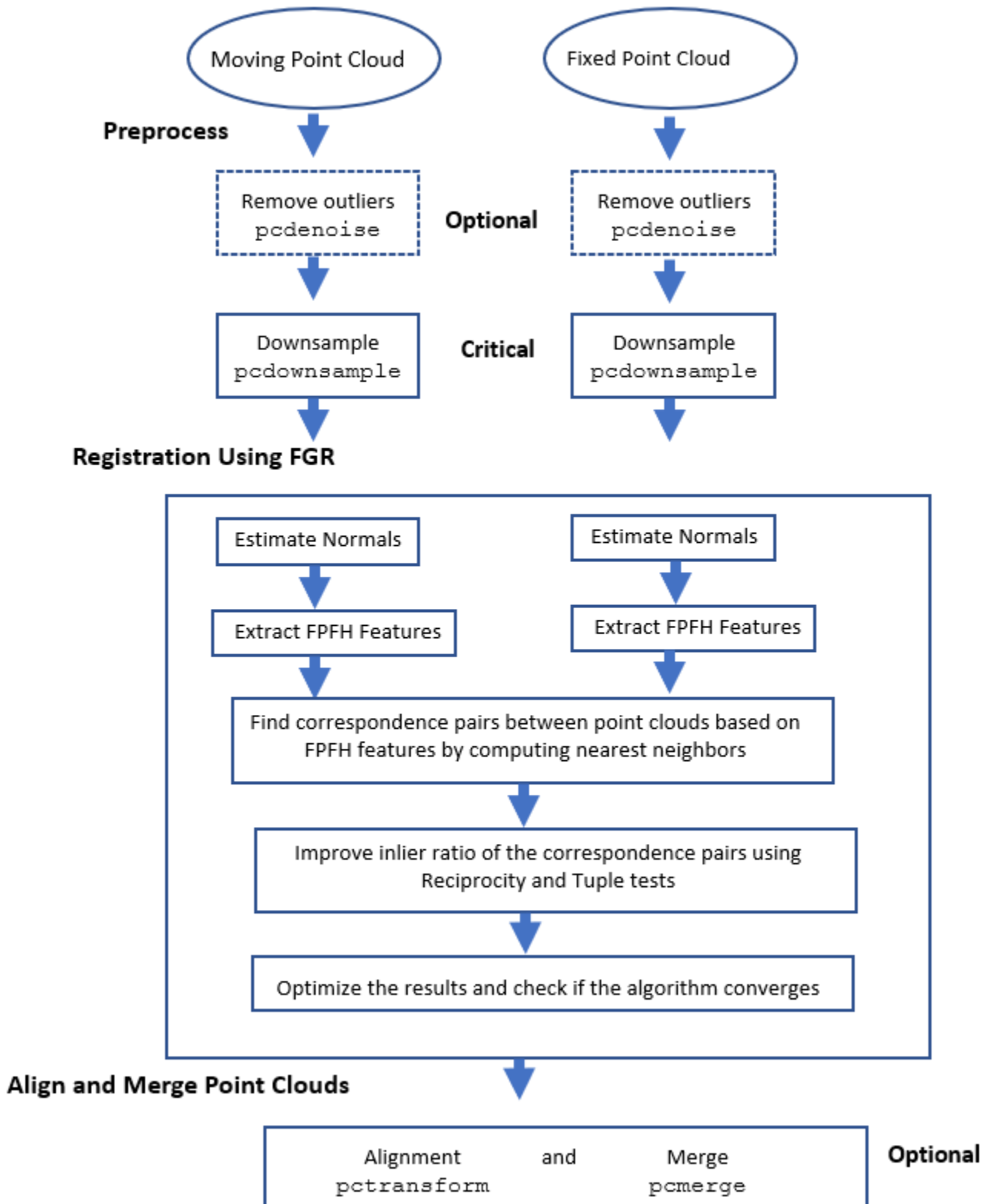
Version History
Introduced in R2022b

See Also

Functions
pcregistericp | pcregisterndt | pcregistercpd | pctransform | pcshow | pcdownsample | pcfitplane | pcdenoise | pcmerge

Objects
pointCloud | rigidtform3d

Topics
"3-D Point Cloud Registration and Stitching"
"Implement Point Cloud SLAM in MATLAB"

surfaceMeshShow

Display surface mesh

Syntax

```
surfaceMeshShow(surfaceMeshObj)
surfaceMeshShow(triangulationObj)
surfaceMeshShow(vertices,faces)
surfaceMeshShow( ___ ,Name=Value)
```

Description

surfaceMeshShow(surfaceMeshObj) displays the surface mesh specified by the surfaceMesh object surfaceMeshObj.

surfaceMeshShow(triangulationObj) displays the surface mesh specified by the triangulation object.

surfaceMeshShow(vertices,faces) displays the surface mesh defined by the input vertices and faces.

surfaceMeshShow(___ ,Name=Value) specifies options using one or more name-value arguments in addition to any combination of arguments from previous syntaxes. For example, Title="Cuboid" displays the surface mesh with the title "Cuboid".

Examples

Display Surface Mesh Using surfaceMesh Object

Define the mesh vertices and faces for a surface mesh.

```
vertices = [0 0 0; 0 0 1; 0 1 1; 0 0 2; 1 0.5 1];
faces = [1 2 3; 2 3 4; 2 3 5];
```

Create a surfaceMesh object using vertices and faces.

```
mesh = surfaceMesh(vertices,faces);
```

Display the surface mesh.

```
surfaceMeshShow(mesh,Title="Surface Mesh",ColorMap="hot",BackgroundColor="blue")
```

Display Surface Mesh Using triangulation Object

Create a triangulation object that represents a 3-D triangulation.

```
[x,y] = meshgrid(1:15,1:15);
tri = delaunay(x,y);
```

```
z = peaks(15);
triangulationObject = triangulation(tri,x(:),y(:),z(:));
```

Display the surface mesh defined by the triangulation.

```
surfaceMeshShow(triangulationObject,ColorMap="summer",Title="Triangulation Obj Mesh")
```

Input Arguments

surfaceMeshObj — Surface mesh data
surfaceMesh object

Surface mesh data, specified as a surfaceMesh object.

triangulationObj — Triangulation of surface mesh
triangulation object

Triangulation of surface mesh, specified as a triangulation object.

vertices — Mesh vertices
M-by-3 matrix

Mesh vertices, specified as an *M*-by-3 matrix. Each row of the matrix is of the form [x y z], specifying the coordinates of a vertex. Each vertex has a vertex ID equal to its row number in the matrix. *M* is the total number of vertices.

faces — Mesh triangular faces
N-by-3 matrix

Mesh triangular faces, specified as an *N*-by-3 matrix. Each row of the matrix is of the form [V_1 V_2 V_3], specifying the vertex IDs of the vertices that define the triangular face. *N* is the number of faces.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: surfaceMeshShow(mesh,Title="Cuboid") displays the surface mesh with the title "Cuboid".

ColorMap — Colormap for surface mesh
'parula' (default) | character vector | string scalar

Colormap for the surface mesh, specified as one of these options.

- parula
- turbo
- hsv
- hot
- cool
- spring

- summer
- autumn
- winter
- gray
- bone
- copper
- pink
- jet
- lines
- colorcube
- prism
- flag
- white

For more information, see `colormap`.

BackgroundColor — Background color
[0 0 0] (default) | RGB triplet | hexadecimal color code | color name | short color name

Background color for the surface mesh, specified as one of these options.

- **RGB Triplet** — 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].
- **Hexadecimal Color Code** — A character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.
- **Color Name or Short Name** — Specify the name of a color such as 'red' or 'green'. Short names specify a letter from a color name, such as 'r' or 'g'.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code
"red"	"r"	[1 0 0]	"#FF0000"
"green"	"g"	[0 1 0]	"#00FF00"
"blue"	"b"	[0 0 1]	"#0000FF"
"cyan"	"c"	[0 1 1]	"#00FFFF"
"magenta"	"m"	[1 0 1]	"#FF00FF"
"yellow"	"y"	[1 1 0]	"#FFFF00"
"black"	"k"	[0 0 0]	"#000000"
"white"	"w"	[1 1 1]	"#FFFFFF"

Alpha — Transparency of surface mesh
1 (default) | positive scalar in range [0, 1]

Transparency of the surface mesh, specified as a positive scalar in the range [0, 1]. A value of 1 is fully opaque, 0 is completely transparent, and values in between them are semitransparent.

Data Types: single | double

WireFrame — Display mesh surface as wireframe
false (default) | true

Display the mesh surface as a wireframe, specified as a logical true or false. When set to true, the function displays the mesh surfaces as a wireframe. Otherwise, the surface has a solid fill.

Data Types: logical

VerticesOnly — Display only mesh vertices
false (default) | true

Display only mesh vertices, specified as a logical true or false. When set to true, the function displays only the mesh vertices.

Data Types: logical

Title — Title for surface mesh display
" " (default) | character vector | string scalar

Title for the surface mesh display, specified as a character vector or string scalar. This value is empty by default.

Data Types: char | string

Limitations

The surfaceMeshShow function does not display mesh face colors specified by the FaceColors property of the input surfaceMesh object.

You cannot save the output by using the savefig function.

To use functions such as plot after the surfaceMeshShow function, you must create a new figure window. The surfaceMeshShow function cannot create figures.

Version History
Introduced in R2022b

See Also
surfaceMesh | readSurfaceMesh | writeSurfaceMesh | surfaceMeshShow

readSurfaceMesh

Read 3-D surface mesh data from STL or PLY file

Syntax

```
mesh = readSurfaceMesh(fileName)
```

Description

`mesh = readSurfaceMesh(fileName)` reads surface mesh data from an STL or a PLY file with the specified filename and returns it as a `surfaceMesh` object.

Examples

Read Surface Mesh from PLY File

Specify a PLY file from which to read surface mesh data.

```
fileName = fullfile(toolboxdir("lidar"),"lidardata", ...
"surfaceMesh","sphere.ply");
```

Read surface mesh data from the PLY file into the workspace.

```
mesh = readSurfaceMesh(fileName)

mesh =
  surfaceMesh with properties:

         Vertices: [482x3 double]
            Faces: [956x3 int32]
    VertexNormals: [482x3 double]
     VertexColors: []
      FaceNormals: []
       FaceColors: []
       NumVertices: 482
          NumFaces: 956
```

Dispaly the surface mesh.

```
surfaceMeshShow(mesh)
```

Read Surface Mesh from STL file

Specify an STL file from which to read surface mesh data.

```
fileName = fullfile(toolboxdir("lidar"),"lidardata", ...
"surfaceMesh","mobius.stl");
```

Read surface mesh data from the STL file into the workspace.

```
mesh = readSurfaceMesh(fileName)

mesh =
  surfaceMesh with properties:

         Vertices: [1050x3 double]
            Faces: [1960x3 int32]
    VertexNormals: []
     VertexColors: []
      FaceNormals: [1960x3 double]
       FaceColors: []
      NumVertices: 1050
         NumFaces: 1960
```

Dispaly the surface mesh.

```
surfaceMeshShow(mesh)
```

Input Arguments

fileName — Filename to read surface mesh data
character vector | string scalar

Filename to read surface mesh data, specified as a character vector or string scalar. You must specify the file extension .stl or .ply with the filename. If the file is not in your working folder you must specify the full file path.

Data Types: char | string

Output Arguments

mesh — Surface mesh data from STL or PLY file
surfaceMesh object

Surface mesh data from an STL or a PLY file, returned as a surfaceMesh object.

Limitations

You cannot read these attributes from an STL file.

• Vertex normals
• Vertex colors
• Face colors

You cannot read face colors from a PLY file.

Version History
Introduced in R2022b

See Also

surfaceMesh | pc2surfacemesh | readSurfaceMesh | writeSurfaceMesh | surfaceMeshShow

writeSurfaceMesh

Write 3-D surface mesh into STL or PLY file

Syntax

```
writeSurfaceMesh(mesh,fileName)
writeSurfaceMesh(mesh,fileName,Encoding=enc)
```

Description

`writeSurfaceMesh(mesh,fileName)` writes the surface mesh `mesh` into an STL or PLY file with the specified filename.

`writeSurfaceMesh(mesh,fileName,Encoding=enc)` additionally specifies the encoding type as `"ascii"` or `"binary"`.

Examples

Write Surface Mesh Data into STL and PLY File

Define mesh vertices and faces for a surface mesh.

```
vertices = [0 0 0; 0 0 1; 0 1 1; 0 0 2; 1 0.5 1];
faces = [1 2 3; 2 3 4; 2 3 5];
```

Create and display the surface mesh.

```
mesh = surfaceMesh(vertices,faces);
surfaceMeshShow(mesh)
```

Write the surface mesh data into an STL file.

```
writeSurfaceMesh(mesh,"ManifoldMesh.stl")
```

Write the surface mesh data into a PLY file.

```
writeSurfaceMesh(mesh,"ManifoldMesh.ply")
```

Input Arguments

mesh — Surface mesh to write
surfaceMesh object

Surface mesh to write, specified as a `surfaceMesh` object.

fileName — Filename to write surface mesh data to
character vector | string scalar

Filename to write surface mesh data to, specified as a character vector or string scalar. You must specify the `.stl` or `.ply` extension along with the filename. If a file of specified name already exists, it must have write permissions.

enc — Encoding type
`"ascii"` (default) | `"binary"`

Encoding type in which to write surface mesh data to the file, specified as `"ascii"` or `"binary"`.

Data Types: `char` | `string`

Limitations

You cannot write these attributes into an STL file.

- Vertex normals
- Vertex colors
- Face colors

You cannot write face normals and face colors into a PLY file.

Version History
Introduced in R2022b

See Also
`surfaceMesh` | `pc2surfacemesh` | `readSurfaceMesh` | `writeSurfaceMesh` | `surfaceMeshShow`

pc2surfacemesh

Construct surface mesh from 3-D point cloud

Syntax

```
[mesh,depth,perVertexDensity] = pc2surfacemesh(ptCloudIn,"poisson")
[mesh,depth,perVertexDensity] = pc2surfacemesh(ptCloudIn,'poisson',
inputDepth)

[mesh,radii] = pc2surfacemesh(ptCloudIn,"ball-pivot")
[mesh,radii] = pc2surfacemesh(ptCloudIn,'ball-pivot',inputRadii)
```

Description

`[mesh,depth,perVertexDensity] = pc2surfacemesh(ptCloudIn,"poisson")` creates a surface mesh from the input point cloud `ptCloudIn` using the Poisson reconstruction method. The function also returns the octree depth used in the reconstruction `depth` and the vertex density `perVertexDensity`.

`[mesh,depth,perVertexDensity] = pc2surfacemesh(ptCloudIn,'poisson', inputDepth)` additionally specifies the octree depth value for the Poisson reconstruction method.

`[mesh,radii] = pc2surfacemesh(ptCloudIn,"ball-pivot")` constructs a surface mesh from point cloud data using the ball-pivot method. The function also returns the radii used in the reconstruction.

`[mesh,radii] = pc2surfacemesh(ptCloudIn,'ball-pivot',inputRadii)` additionally specifies the radii for the ball-pivot reconstruction method.

Examples

Construct Surface Mesh from Point Cloud Using Poisson Method

Load point cloud data from a PLY file into the workspace.

```
ptCloud = pcread("teapot.ply");
```

Display the input point cloud.

```
pcshow(ptCloud)
```
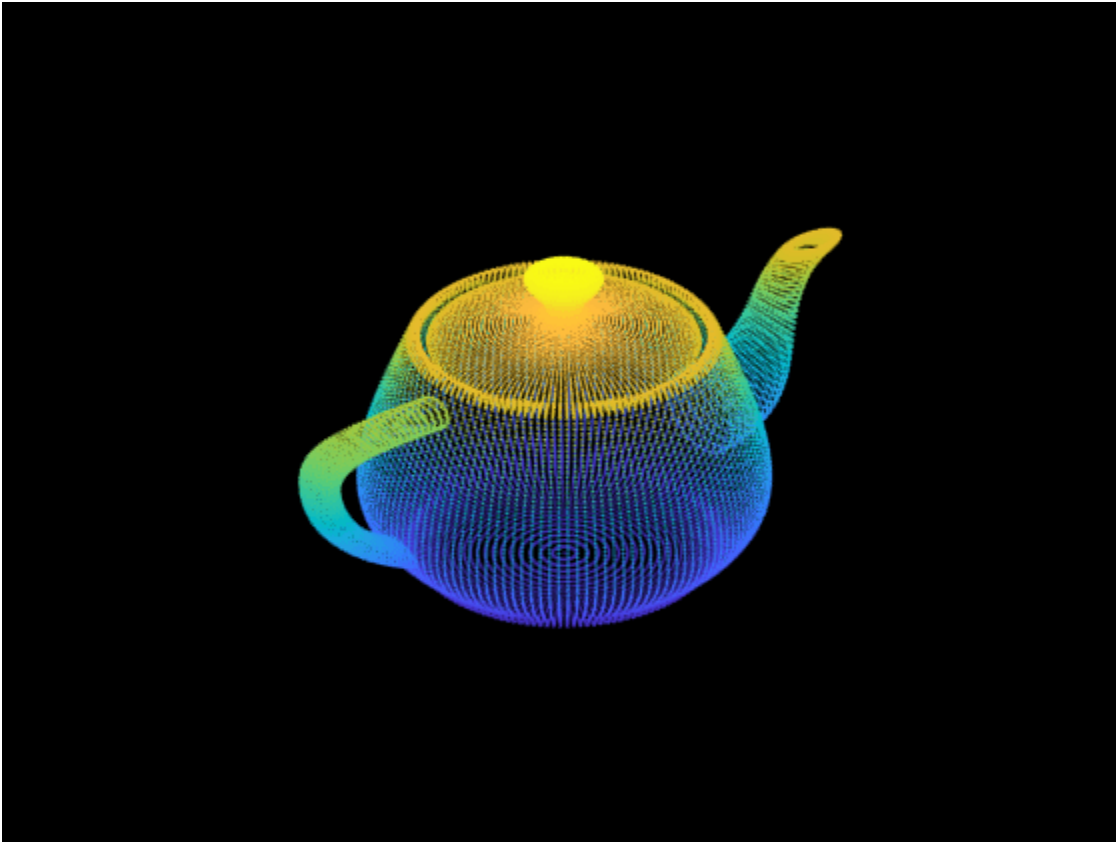
Downsample the point cloud.

```
gridstep = 0.05;
ptCloudDownSampled = pcdownsample(ptCloud,"gridAverage",gridstep);
```

Construct surface mesh from the point cloud data using the Poisson method, and display the surface mesh.

```
depth = 8;
mesh = pc2surfacemesh(ptCloudDownSampled,"poisson",depth);
surfaceMeshShow(mesh)
```

Construct Surface Mesh from Point Cloud Using Ball-Pivot Method

Load point cloud data from a PLY file into the workspace.

```
ptCloud = pcread("teapot.ply");
```

Display the input point cloud.

```
pcshow(ptCloud)
```
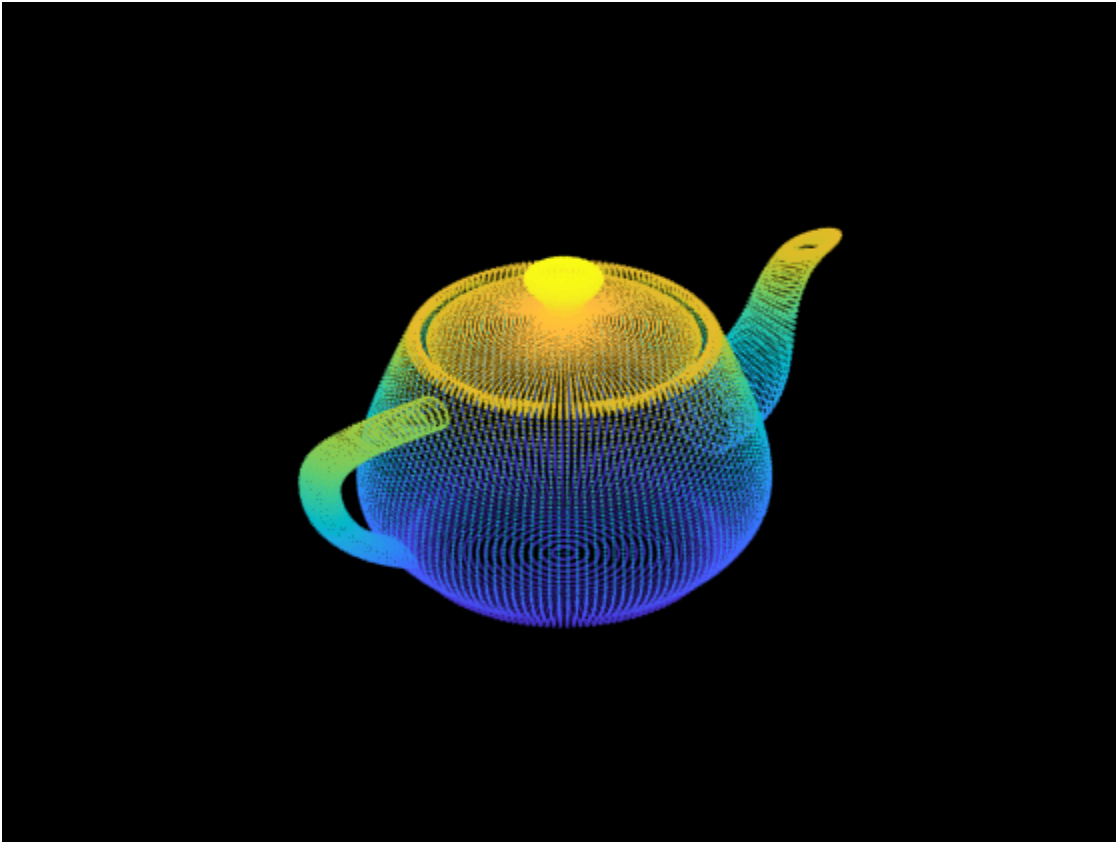
Downsample the point cloud.

```
gridstep = 0.05;
ptCloudDownSampled = pcdownsample(ptCloud,"gridAverage",gridstep);
```

Construct a surface mesh from the point cloud data using the ball-pivot method and display the surface mesh.

```
mesh = pc2surfacemesh(ptCloudDownSampled,"ball-pivot");
surfaceMeshShow(mesh)
```

Input Arguments

ptCloudIn — Input point cloud data
pointCloud object

Input point cloud data, specified as a pointCloud object.

inputDepth — Octree depth to use in Poisson reconstruction
8 (default) | positive integer

Octree depth to use in Poisson reconstruction, specified as a positive integer in the range [2, 12]. Increasing the octree depth of the Poisson reconstruction increase the detail of the surface mesh.

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

inputRadii — Radii values for ball-pivot reconstruction
M-element vector

Radii values for ball-pivot reconstruction, specified as an *M*-element vector. You must specify the values depending on the point cloud density. Values are in meters.

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

Output Arguments

mesh — Surface mesh constructed from point cloud
`surfaceMesh` object

Surface mesh constructed from the point cloud, returned as a `surfaceMesh` object.

depth — Octree depth used in Poisson reconstruction
positive integer

Octree depth used in the Poisson reconstruction, returned as a positive integer.

radii — Radii values for ball-pivot reconstruction
three-element vector

Radii values for the ball-pivot reconstruction, returned as a three-element vector. Units are in meters.

perVertexDensity — Density at mesh vertices
M-element vector

Density at mesh vertices, returned as an *M*-element vector. *M* is the number of mesh vertices in the output surface mesh. You can further refine the mesh by eliminating vertices with insignificant density.

Algorithms

Poisson Reconstruction

The *Poisson* reconstruction method consists of these steps.

1   Transform the point samples into a continuous vector field.
2   Solve a Poisson system, containing 3-D Laplacian equations, to find a function whose gradient best describes the point cloud.
3   Reconstruct the surface from the function equation.

Ball-Pivot Reconstruction

The *ball-pivot* method triangulates a set of points by rolling a ball, of radius *r*, on the point cloud. The algorithm consists of these steps.

1   Place the ball in contact with three sample points. These points form the *seed* triangle.
2   Keep the ball in contact with two of these initial points (an *edge* of the seed triangle) and pivot the ball until it touches another point. The edge and the new point define a new triangle.

3 Pivot the ball using the new triangle edge. Using an edge of the new triangle, repeat the process of pivoting and defining a new triangle with a touched point. The triangles formed through this process constitute the interpolating mesh.

4 Continue this process until all reachable edges are covered, and then start with another seed triangle.

5 Repeat the entire process with larger radii to reconstruct uneven surfaces.

Version History
Introduced in R2022b

See Also
surfaceMesh | readSurfaceMesh | writeSurfaceMesh | surfaceMeshShow

segmentCurbPoints

Segment curb points from point cloud

Syntax

```
curbPtsIdx = segmentCurbPoints(onRoadPointCloud)
curbPtsIdx = segmentCurbPoints(onRoadPointCloud,roadAngles)
curbPtsIdx = segmentCurbPoints(onRoadPointCloud,roadAngles,prevCurbPoints)
[curbPtsIdx,curbPtCloud] = segmentCurbPoints( ___ )
[ ___ ] = segmentCurbPoints( ___ ,Name=Value)
```

Description

`curbPtsIdx = segmentCurbPoints(onRoadPointCloud)` segments the indices of the feature curb points from an organized point cloud which contains on-road points. A *curb* usually defines the road boundary, and forms an edge for the sidewalk.

`curbPtsIdx = segmentCurbPoints(onRoadPointCloud,roadAngles)` specifies the road angles. The function further processes the feature curb points using these road angles and returns the candidate curb points.

`curbPtsIdx = segmentCurbPoints(onRoadPointCloud,roadAngles,prevCurbPoints)` specifies the curb points segmented from the previous point cloud frames. The function uses the previous curb points to improve robustness when the input point cloud has occlusions.

`[curbPtsIdx,curbPtCloud] = segmentCurbPoints( ___ )` returns the segmented curb points in the on-road point cloud as a `pointCloud` object, using any combination of input arguments from previous syntaxes.

`[ ___ ] = segmentCurbPoints( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to any combination of arguments from previous syntaxes. For example, `HeightLimits=[0.1 0.3]` specifies the minimum and the maximum height of the road curb as 0.1 and 0.3 meters, respectively.

Examples

Segment Curb Points from Point Cloud

Read point cloud data from a PCD file by using the `pcread` function.

```
ptCloud = pcread("HDL64LidarData.pcd");
```

Organize the point cloud data by using the `pcorganize` function.

```
ptCloud = pcorganize(ptCloud,lidarParameters("HDL64E",1024));
```

Extract a region of interest, which contains a road, from the point cloud data.

```
roi = [-25 25 -10 24 ptCloud.ZLimits];
indices = findPointsInROI(ptCloud,roi);
ptCloud = select(ptCloud,indices,OutputSize="full");
```

Segment the on-road and off-road points from the point cloud by using the segmentGroundSMRF function.

```
[~,offRoadPtCloud,onRoadPtCloud] = segmentGroundSMRF(ptCloud);
```
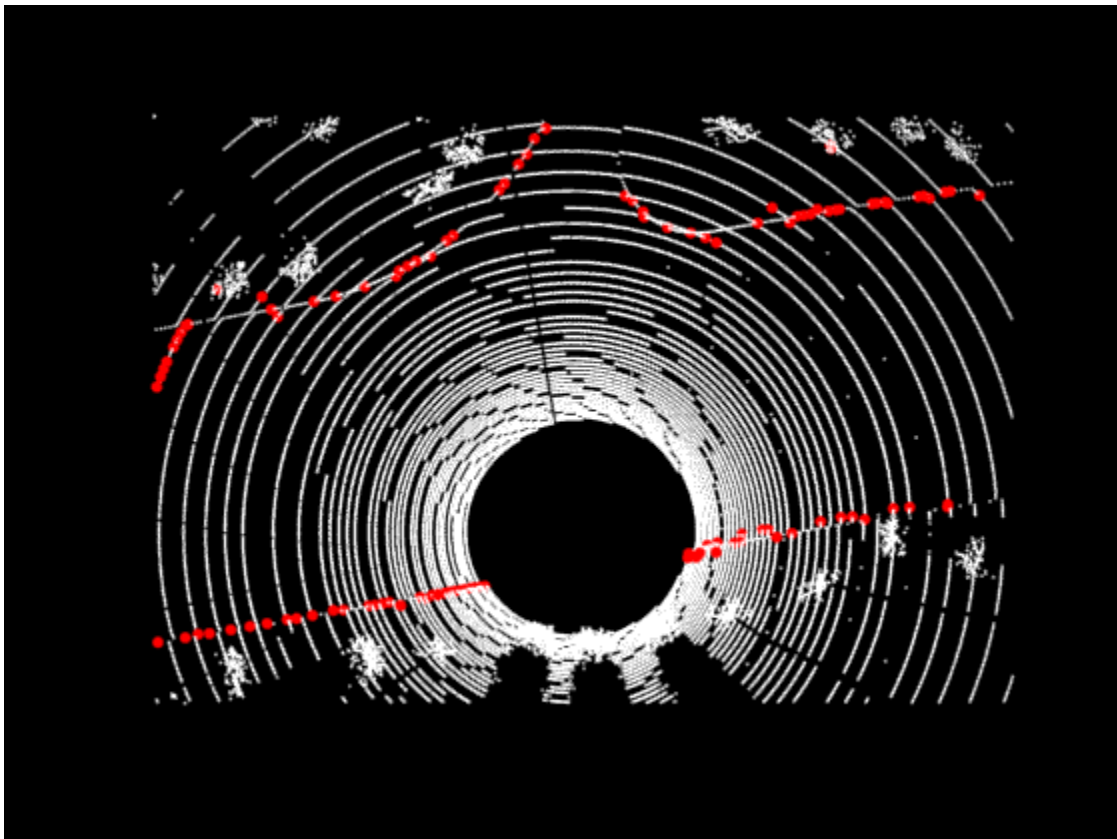
Detect road angles from the off-road points.

```
roadAngles = detectRoadAngles(offRoadPtCloud,MinSectorSize=10,SectorMergeThreshold=30);
```

Segment curb points from the on-road points of the point cloud.

```
[~,curbPtCloud] = segmentCurbPoints(onRoadPtCloud,roadAngles,NumScanNeighbors=10, ...
        HeightLimits=[0.001 0.5],HeightDeviationLimits=[0.001 0.5], ...
        SmoothnessThreshold=0.0001,HorizontalAngularResolution=0.33);
```

Visualize the segmented curb points.

```
figure
pcshow(ptCloud.Location,"w")
hold on;
pcshow(curbPtCloud.Location,"r",MarkerSize=200)
hold off
view(2)
title("Curb points")
```

Input Arguments

onRoadPointCloud — Organized point cloud with on-road points
pointCloud object

Organized point cloud with on-road points, specified as a `pointCloud` object. The on-road area consists of sidewalks, curb surfaces, and road surfaces. You can apply ground segmentation or plane-fitting algorithms to your point cloud data to extract on-road points. For more details, see "Extract On-Road and Off-Road Points from Point Cloud".

roadAngles — Road segmentation angles
M-element vector

Road segmentation angles, specified as an *M*-element vector. The value of *M* depends on the road type.

Road Type	*M* Value
Straight or curved road	2
T-shaped or Y-shaped road	3
Cross-road (+)	4
6-way junction	6

You can compute road angles by using the `detectRoadAngles` function or input them manually to the function.

When you specify the road angles, the function processes the feature curb points using RANSAC polynomial fitting to return the candidate curb points.

prevCurbPoints — Segmented curb points from previous point cloud frames
pointCloud object | array of pointCloud objects

Segmented curb points from the previous point cloud frames, specified as a `pointCloud` object or an array of `pointCloud` objects. These point clouds must be in the `onRoadPointCloud` frame of reference. Detections from the previous frames can improve the robustness of the function when the input point cloud has occlusions.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `segmentCurbPoints(onRoadPtCloud,HeightLimits=[0.1 0.3])` specifies the minimum and the maximum height of the road curb as 0.1 and 0.3 meters, respectively.

NumScanNeighbors — Number of neighbors for each point in a scan line
5 (default) | positive integer

Number of left and right neighbors for each point in a scan line, specified as a positive integer. Increasing this value can improve the accuracy of curb segmentation.

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

`HeightLimits` — Height difference limits for road curb
`[0.02 0.25]` (default) | two-element nonnegative vector

Height limits for the road curb, specified as a two-element nonnegative vector of the form [*min max*]. The *min*, *max* values specify the minimum and the maximum height of the road curb, respectively. The typical range of the minimum height is `[0.001, 0.05]`, and the typical range of the maximum height is `[0.15, 0.5]`. The values are in meters.

Data Types: `single` | `double`

`HeightDeviationLimits` — Minimum and maximum standard deviation in curb height
`[0.02 0.07]` (default) | two-element nonnegative vector

Minimum and maximum standard deviation in the curb height, specified as a two-element nonnegative vector of the form [*min max*]. The *min*, *max* values specify the minimum and the maximum standard deviation in the curb height, respectively.

Data Types: `single` | `double`

`SmoothnessThreshold` — Minimum smoothness value for curb points
`0.001` (default) | positive scalar

Minimum smoothness value for curb points, specified as a positive scalar. The typical range of this value is (`0`, `0.005`]. Increasing this value can decrease the number of curb points segmented.

Data Types: `single` | `double`

`HorizontalAngularResolution` — Horizontal angular resolution of lidar sensor
positive scalar

Horizontal angular resolution of the lidar sensor, specified as a positive scalar in degrees. When you do not specify this value, the function internally computes it as 360/
(`size(onRoadPointCloud.Location,2)`). The typical range for horizontal angular resolution is
`[0.1, 0.4]`. Increasing this value can decrease the number of curb points segmented.

Data Types: `single` | `double`

Output Arguments

`curbPtsIdx` — Indices of curb points
M-by-*N* logical matrix

Binary map of the point cloud with the indices of the curb points, returned as an *M*-by-*N* logical matrix. *M* and *N* are the number of rows and columns, respectively in the input `onRoadPointCloud`.

`curbPtCloud` — Point cloud of segmented curb points
`pointCloud` object

Point cloud of the segmented curb points, returned as a `pointCloud` object.

Algorithms

The function extracts curb points from `onRoadPointCloud` using these steps.

1   For each point in a scan line, the function computes these three features.

- *Height Difference Feature* — Computes the standard deviation and the height maximum difference around a point. The standard deviation and height difference of a curb point must be within the specified `HeightDeviationLimits` and `HeightLimits`, respectively.

- *Smoothness Feature* — Computes the smoothness of the area around a point. A higher smoothness value indicates that the point is an edge point, and a lower values indicates that the point is a plane point. The smoothness value for a curb point must be greater than the `SmoothnessThreshold` value.

- *Horizontal and Vertical Continuity Feature* — Computes the horizontal and vertical distance between a point and its immediate neighbors. These horizontal and vertical distance values must be less than the horizontal and the vertical continuity thresholds, respectively. The function computes the thresholds values from the `HorizontalAngularResolution` of the lidar sensor.

2   If a point satisfies all computed features, then it as a *feature curb point*.

3   If you specify road angles as an input, the function further fine tunes the feature curb points using RANSAC polynomial fitting, and returns the *candidate curb points*.

Version History
Introduced in R2022b

See Also
detectRoadAngles | segmentGroundSMRF | segmentLidarData | segmentGroundFromLidarData | pcfitplane

Topics
"Extract On-Road and Off-Road Points from Point Cloud"
"Curb Detection and Tracking in 3-D Lidar Point Cloud"

detectRoadAngles

Detect road angles in point cloud

Syntax

```
roadAngles = detectRoadAngles(offRoadPointCloud)
roadAngles = detectRoadAngles(offRoadPointCloud,Name=Value)
```

Description

`roadAngles = detectRoadAngles(offRoadPointCloud)` detects road angles in a point cloud. The point cloud must contain off-road points.

`roadAngles = detectRoadAngles(offRoadPointCloud,Name=Value)` specifies options using one or more name-value arguments. For example, `MinSectorSize=10` specifies the minimum sector size required to detect a road segment as 10 degrees.

Examples

Segment Curb Points from Point Cloud

Read point cloud data from a PCD file by using the `pcread` function.

```
ptCloud = pcread("HDL64LidarData.pcd");
```

Organize the point cloud data by using the `pcorganize` function.

```
ptCloud = pcorganize(ptCloud,lidarParameters("HDL64E",1024));
```

Extract a region of interest, which contains a road, from the point cloud data.

```
roi = [-25 25 -10 24 ptCloud.ZLimits];
indices = findPointsInROI(ptCloud,roi);
ptCloud = select(ptCloud,indices,OutputSize="full");
```

Segment the on-road and off-road points from the point cloud by using the `segmentGroundSMRF` function.

```
[~,offRoadPtCloud,onRoadPtCloud] = segmentGroundSMRF(ptCloud);
```

Detect road angles from the off-road points.

```
roadAngles = detectRoadAngles(offRoadPtCloud,MinSectorSize=10,SectorMergeThreshold=30);
```

Segment curb points from the on-road points of the point cloud.

```
[~,curbPtCloud] = segmentCurbPoints(onRoadPtCloud,roadAngles,NumScanNeighbors=10, ...
        HeightLimits=[0.001 0.5],HeightDeviationLimits=[0.001 0.5], ...
        SmoothnessThreshold=0.0001,HorizontalAngularResolution=0.33);
```

Visualize the segmented curb points.

```
figure
pcshow(ptCloud.Location,"w")
hold on;
pcshow(curbPtCloud.Location,"r",MarkerSize=200)
hold off
view(2)
title("Curb points")
```


Input Arguments

offRoadPointCloud — Point cloud with off-road points
pointCloud object

Point cloud with off-road points, specified as a `pointCloud` object. Off-road points usually consist of trees, buildings, and other objects. You can apply ground segmentation or plane-fitting algorithms to your point cloud data to extract off-road points. For more details, see "Extract On-Road and Off-Road Points from Point Cloud".

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `detectRoadAngles(offRoadPointCloud,MinSectorSize=10)` specifies the minimum sector size required to detect a road segment as 10 degrees.

MinSectorSize — Minimum sector size to detect road segment
5 (default) | scalar in the range [0, 360]

Minimum sector size to detect a road segment, specified as a scalar in the range [0, 360, in degrees. The function does not detect any sector smaller than this value as a road segment. Increasing this value can improve the accuracy of the road angle detection. MinSectorSize must be in the range [0,360].

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

SectorMergeThreshold — Minimum merge angle between two sectors
15 (default) | scalar in the range [0, 360]

Minimum merge angle between two sectors, specified as a scalar in the range [0, 360, in degrees. The function merges two sectors when the angle between them is lower than the SectorMergeThreshold value. Increasing this value can improve the accuracy of the road angle detection. SectorMergeThreshold must be in the range [0,360].

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

Output Arguments

roadAngles — Road segmentation angles
M-element vector

Road segmentation angles, returned as an *M*-element vector. *M* is the number of directions in which the egovehicle can travel, depending on the road type. The values are in degrees, with respect to the lidar sensor coordinate system.

Road Type	*M* Value
Straight or curved road	2
T-shaped or Y-shaped road	3
Cross-road (+)	4
6-way junction	6

Algorithms

The function uses a beam model, followed by a toe-finding algorithm, on the off-road points to detect the road angles.
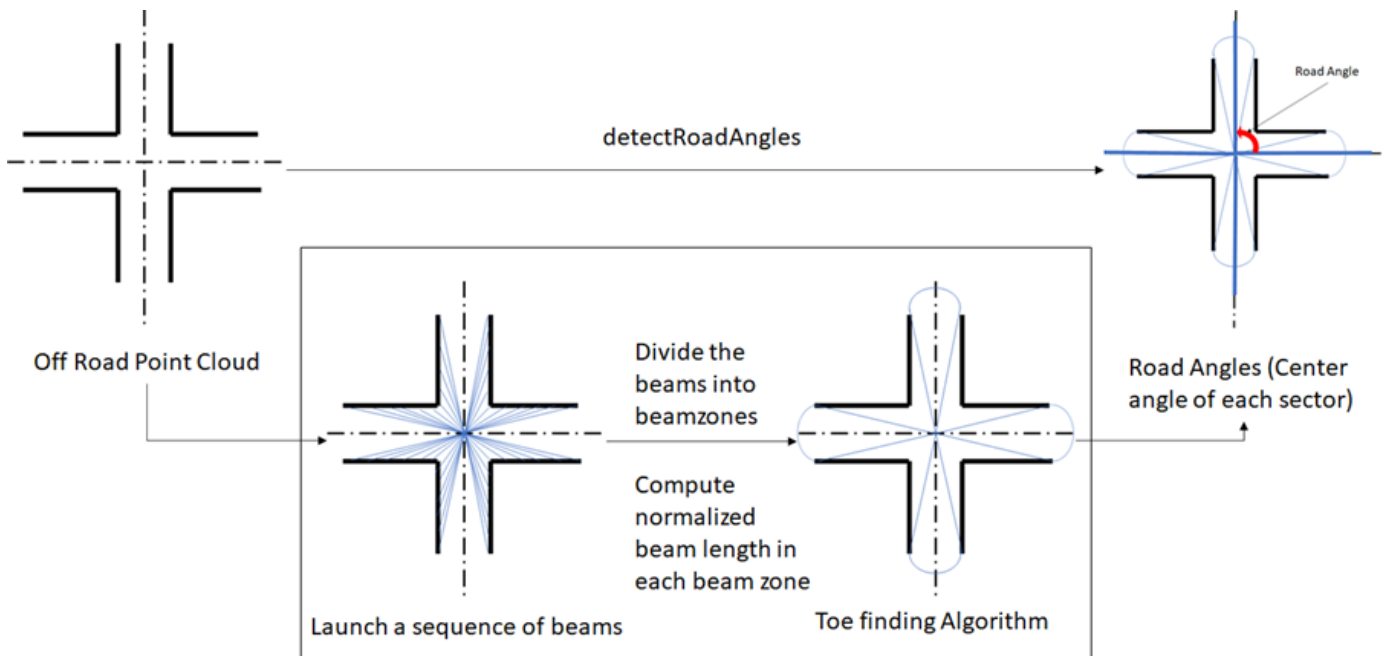
Beam Model

The beam model follows these steps.

1   Launch a sequence of beams from the lidar sensor mounted on the ego vehicle. The lidar sensor is the launching point.

2   Divide the beams into beam zones according to the angular resolution of the sensor.

3   Determine the beam angles and beam lengths with respect to the launching point.

4   For each beam zone, determine the distance between the point closest to the launching point and the point farthest from the launching point. Compute the normalized beam length as the ratio of the shortest distance to the longest distance.

Toe-Finding Algorithm

The toe-finding algorithm follows these steps.

1   Classify the beam zones into sectors based on their normalized beam lengths.

2   Update the sectors using the specified `MinSectorSize` and `SectorMergeThreshold` values.
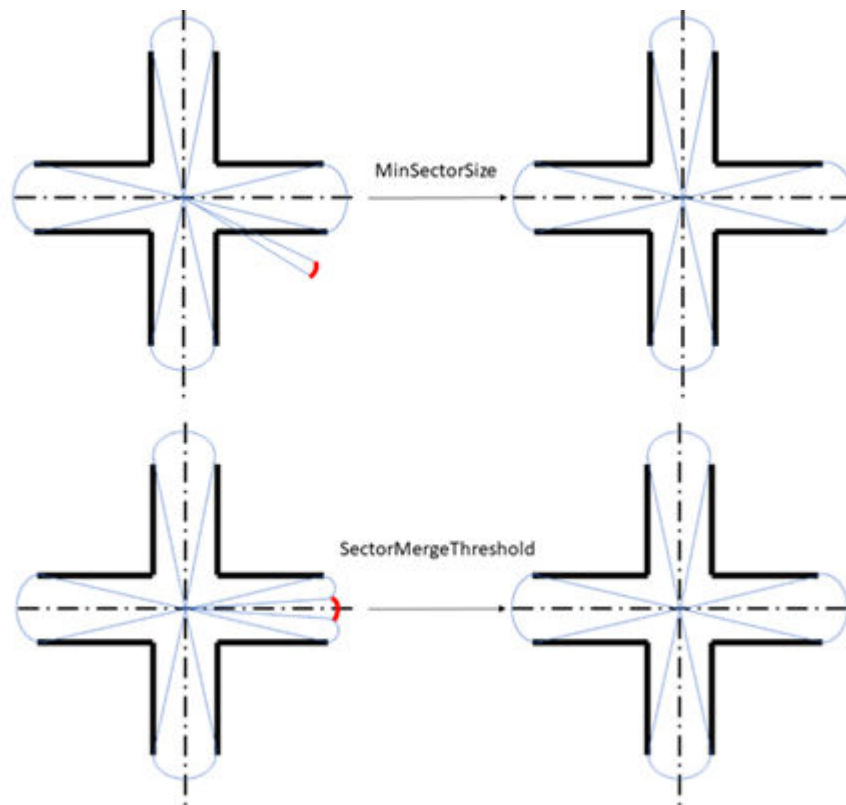
3   Return the center angle of each sector with respect to the positive *x*-axis as the road segmentation angle.

Version History

Introduced in R2022b

See Also

segmentCurbPoints | segmentGroundSMRF | segmentLidarData | segmentGroundFromLidarData | pcfitplane

Topics
"Extract On-Road and Off-Road Points from Point Cloud"
"Curb Detection and Tracking in 3-D Lidar Point Cloud"

show

Visualize LOAM map

Syntax

```
show(loamMap)
show(loamMap,Name=Value)
ax = show( ___ )
```

Description

show(loamMap) displays the edge and surface points contained in the input lidar odometry and mapping (LOAM) map. The function displays surface points in magenta and edge points in green.

show(loamMap,Name=Value) specifies options using one or more name-value argument. For example, MarkerSize=5 sets the diameter size of the marker to 5 points.

ax = show(___) returns the plot axes using any combination of input arguments from previous syntaxes.

Examples

Add Points to LOAM Map

Create a LOAM map to store LOAM feature points.

```
voxelSize = 0.5;
loamMap = pcmaploam(voxelSize);
```

Load point cloud data into the workspace.

```
ld = load("drivingLidarPoints.mat");
```

Detect LOAM feature points.

```
points = detectLOAMFeatures(ld.ptCloud);
```

Add the LOAM points to the map.

```
absPose = rigidtform3d;
addPoints(loamMap,points,absPose)
```

Visualize the points in the LOAM map.

```
show(loamMap)
```

Input Arguments

`loamMap` — LOAM map
`pcmaploam` object

LOAM map, specified as a `pcmaploam` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `show(loamMap,MarkerSize=5)` sets the diameter size of the marker to 5 points.

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

Diameter of each marker, specified as a positive scalar. The value specifies the approximate diameter of the markers used for each LOAM point in the display. MATLAB graphics defines the unit as points.

`Parent` — Axes on which to display the visualization
`Axes` graphics object

Axes on which to display the visualization, specified as an `Axes` object. To create an `Axes` object, use the `axes` function. To display the visualization in a new figure, leave `Parent` unspecified.

Output Arguments

ax — Plot axes
Axes graphics object

Plot axes, returned as an `axes` graphics object. You can set the default center of rotation for the viewer as around the axes center or a point. Set the default behavior from the "Computer Vision Toolbox Preferences".

Version History
Introduced in R2022b

See Also

Objects
pcmaploam

Functions
addPoints | findPose

findPose

Find absolute pose of points in map

Syntax

```
absPose = findPose(loamMap,points,relPose)
absPose = findPose( ___ ,localMapSize)
[absPose,optimizedRelPose] = findPose( ___ )
[absPose,optimizedRelPose,rmse] = findPose( ___ )
[ ___ ] = findPose( ___ ,Name=Value)
```

Description

`absPose = findPose(loamMap,points,relPose)` returns the optimized absolute pose that aligns the specified points to the points in the lidar odometry and mapping (LOAM) map `loamMap`.

`absPose = findPose( ___ ,localMapSize)` specifies the size of the local map used to refine the absolute pose, in addition to all input arguments from the previous syntax. When you do not specify `localMapSize`, the function uses a map size defined by the *x*, *y*, and *z* spatial extents of all the input points enlarged on all sides by a spatial radius, `SearchRadius`, centered around the estimated absolute pose `absPose`.

`[absPose,optimizedRelPose] = findPose( ___ )` returns the optimized relative pose.

`[absPose,optimizedRelPose,rmse] = findPose( ___ )` returns the root mean squared error of the Euclidean distance between the aligned points. Lower values indicate a more accurate alignment.

`[ ___ ] = findPose( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to any combination of arguments from previous syntaxes. For example, `SearchRadius=4` sets the search radius for point matching to 4.

Examples

Find Absolute Pose of Points in LOAM Map

Create a map to store LOAM feature points.

```
voxelSize = 0.1;
loamMap = pcmaploam(voxelSize);
```

Create a `velodyneFileReader` object to read point cloud daata.

```
veloReader = velodyneFileReader("lidarData_ConstructionRoad.pcap","HDL32E");
```

Read the first point cloud into the workspace.

```
ptCloud1 = readFrame(veloReader,1);
```

Detect LOAM feature points.

```
points1 = detectLOAMFeatures(ptCloud1);
```

Downsample the less planar surface points to improve registration speed.

```
gridStep = 1;
points1 = downsampleLessPlanar(points1,gridStep);
```

Add the LOAM points of the first point cloud to the map.

```
absPose = rigidtform3d;
addPoints(loamMap,points1,absPose)
```

Read the fifth point cloud, and detect the LOAM feature points in it.

```
ptCloud2 = readFrame(veloReader,5);
points2 = detectLOAMFeatures(ptCloud2);
```

Downsample the less planar surface points.

```
points2 = downsampleLessPlanar(points2,gridStep);
```

Get a relative pose estimate by using the `pcregisterloam` function.

```
relPose = pcregisterloam(points2,points1);
```

Find the absolute pose of the points from the fifth point cloud in the map.

```
absPose = findPose(loamMap,points2,relPose);
```

Add the points from the fifth point cloud to the map.

```
addPoints(loamMap,points2,absPose)
```

Visualize the map.

```
figure
show(loamMap,MarkerSize=20)
```
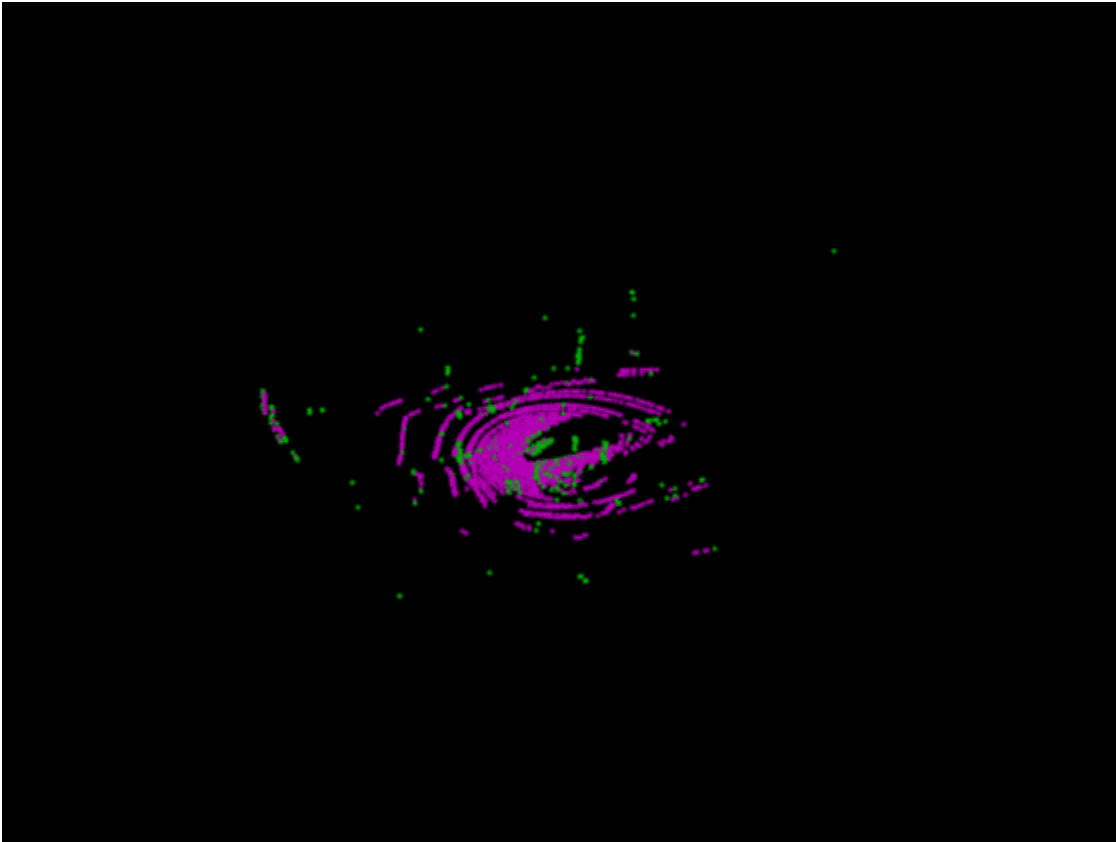
Input Arguments

`loamMap` — LOAM map
`pcmaploam` object

LOAM map, specified as a `pcmaploam` object.

`points` — LOAM points
`LOAMPoints` object

LOAM points, specified as a `LOAMPoints` object.

`relPose` — Relative pose
`rigidtform3d` object

Relative pose, specified as a `rigidtform3d` object. The `relPose` input contains the relative pose between the new input points and the last points added to the LOAM map `loamMap` in the sensor frame. You can use the `pcregisterloam` function to estimate the relative pose.

`localMapSize` — Local map size
three-element vector

Local map size, specified as a three-element vector of the form [dx dy dz]. When you do not specify `localMapSize`, the function uses a map size defined by the x, y, and z spatial extents of all the input

points enlarged on all sides by a spatial radius, `SearchRadius`, centered around the estimated absolute pose `absPose`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `findPose(loamMap,points,relPose,SearchRadius=4)` sets the search radius for point matching to 4.

SearchRadius — Search radius for point matching
3 (default) | positive integer

Search radius for point matching, specified as a positive integer. When matching, the function finds the closest edge and surface points within the specified radius. For best results, set this value based on the certainty of the relative pose, specified by the `relPose` input. You can increase the value of `SearchRadius` when there is greater uncertainty in the relative pose input, but this can also decrease the registration speed.

MaxIterations — Maximum iterations before LOAM registration stops
20 (default) | positive integer

Maximum iterations before LOAM registration stops, specified as a positive integer.

Tolerance — Tolerance between consecutive LOAM iterations
[0.01,0.5] (default) | two-element vector

Tolerance between consecutive LOAM iterations, specified as a two-element vector, with nonnegative values, of the form [*Tdiff Rdiff*].

- *Tdiff* — Tolerance for the estimated absolute difference in translation and rotation between consecutive LOAM iterations. Measures the Euclidean distance between two translation vectors.

- *Rdiff* — Tolerance for the estimated absolute difference of the angular rotation between consecutive iterations, measured in degrees.

The algorithm stops when the difference between the estimates of the rigid transformations in the most recent consecutive iteration falls below the specified tolerance value.

Verbose — Display progress information
false or 0 (default) | true or 1

Display progress information, specified as a numeric or logical 0 (`false`) or 1 (`true`). To display progress information, set `Verbose` to `true`.

Output Arguments

absPose — Absolute pose
rigidtform3d object

Absolute pose for aligning new points to the existing points in a LOAM map, returned as a `rigidtform3d` object. The `addPoints` function uses the absolute pose to align new points to the existing map.

optimizedRelPose — Optimized relative pose
`rigidtform3d` object

Optimized relative pose, returned as a `rigidtform3d` object.

rmse — Root mean squared error
positive scalar

Root mean squared error, returned as a positive scalar that described the Euclidean distance between the aligned points. Lower values indicate a more accurate alignment.

Version History
Introduced in R2022b

See Also

Objects
pcmaploam | LOAMPoints | rigidtform3d

Functions
addPoints | show

addPoints

Add LOAM points to map

Syntax

```
addPoints(loamMap,points,absPose)
```

Description

`addPoints(loamMap,points,absPose)` adds the specified LOAM points to the lidar odometry and mapping (LOAM) map. The function uses the absolute pose to align the new points to the existing map.

`addPoints` ensures that each voxel has no more than one edge point and one surface point. The voxel size is specified by the `VoxelSize` property of `loamMap`.

Examples

Add Points to LOAM Map

Create a LOAM map to store LOAM feature points.

```
voxelSize = 0.5;
loamMap = pcmaploam(voxelSize);
```

Load point cloud data into the workspace.

```
ld = load("drivingLidarPoints.mat");
```

Detect LOAM feature points.

```
points = detectLOAMFeatures(ld.ptCloud);
```

Add the LOAM points to the map.

```
absPose = rigidtform3d;
addPoints(loamMap,points,absPose)
```

Visualize the points in the LOAM map.

```
show(loamMap)
```

Input Arguments

loamMap — LOAM map
pcmaploam object

LOAM map, specified as a pcmaploam object.

points — LOAM points
LOAMPoints object

LOAM points, specified as a LOAMPoints object.

absPose — Absolute pose
rigidtform3d object

Absolute pose, specified as a rigidtform3d object. The object function uses the absolute pose to align the new points to the existing map.

When the map size, specified by the MapSize property of the pcmaploam object loamMap, limits the size of the LOAM map, the addPoints object function updates the XLimits, YLimits, and ZLimits properties of loamMap to reflect the LOAM points it contains. The function uses the center of the map, which is defined by absPose, to update the properties.

Version History
Introduced in R2022b

See Also

Objects
pcmaploam | LOAMPoints | rigidtform3d

Functions
show | findPose

pcregisterloam

Register two point clouds using LOAM algorithm

Syntax

```
tform = pcregisterloam(movingPtCloud,fixedPtCloud,gridStep)
tform = pcregisterloam(movingPoints,fixedPoints)
[tform,rmse] = pcregisterloam( ___ )
[ ___ ] = pcregisterloam( ___ ,Name=Value)
```

Description

`tform = pcregisterloam(movingPtCloud,fixedPtCloud,gridStep)` registers the moving point cloud `movingPoints` with the fixed point cloud `fixedPoints` using the lidar odometry and mapping (LOAM) algorithm. The function returns the rigid transformation `tform`, between the moving and fixed point clouds. `gridStep` specifies the size of a 3-D box used to downsample the LOAM points detected in the input point clouds.

`tform = pcregisterloam(movingPoints,fixedPoints)` registers the moving LOAM points `movingPoints` with the fixed LOAM points `fixedPoints` and returns the rigid transformation between them `tform`. Using this function to register LOAM points is faster and more accurate than using it to register point clouds.

You can obtain the LOAM points of the moving and fixed point clouds by using the `detectLOAMFeatures` function, which detects LOAM feature points from organized point clouds.

`[tform,rmse] = pcregisterloam( ___ )` returns the root mean squared error `rmse` of the Euclidean distance between the aligned points.

`[ ___ ] = pcregisterloam( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to any combination of arguments from previous syntaxes. For example, `MatchingMethod='one-to-one'` sets the matching method algorithm to `'one-to-one'`.

Examples

Align Two Point Clouds Using LOAM Registration Algorithm

Load and visualize point cloud data.

```
ld = load('livingRoom.mat');
movingPtCloud = ld.livingRoomData{1};
fixedPtCloud = ld.livingRoomData{2};
figure
pcshowpair(movingPtCloud,fixedPtCloud,'VerticalAxis','Y','VerticalAxisDir','Down')
```
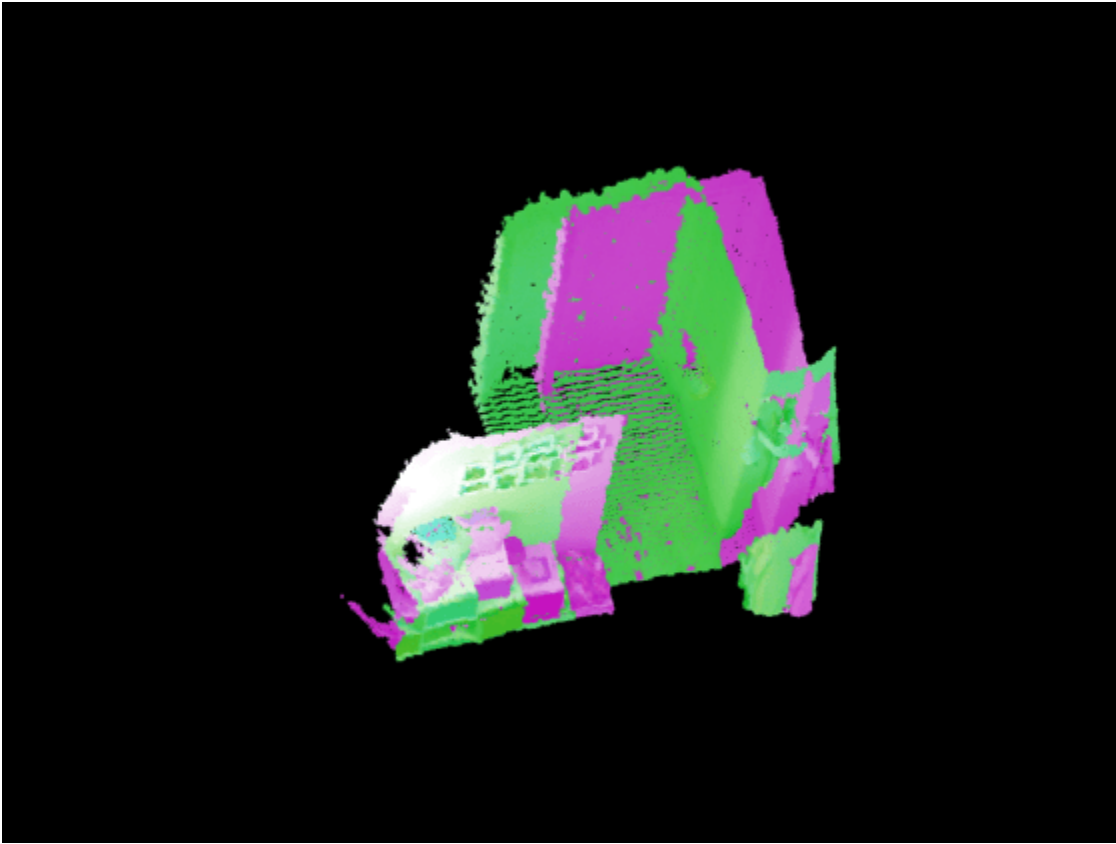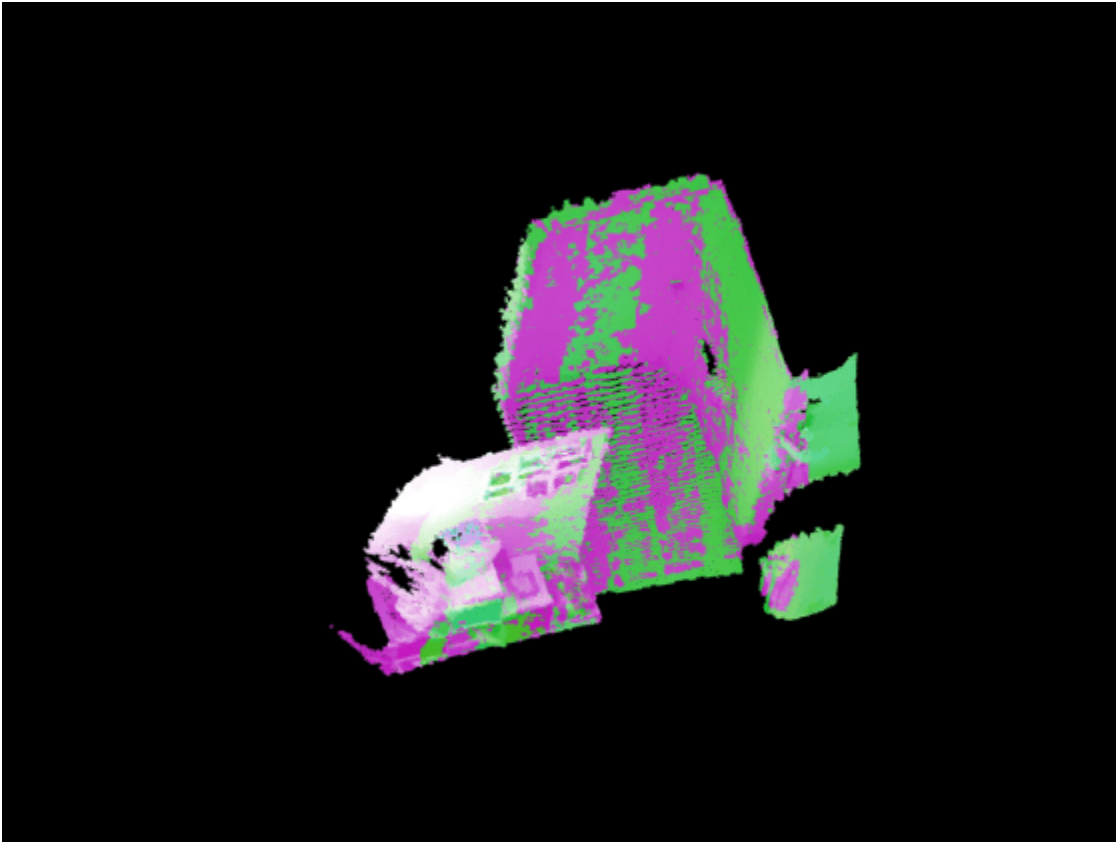
Register the point clouds.

```
gridStep = 0.5;
tform = pcregisterloam(movingPtCloud,fixedPtCloud,gridStep);
```

Align and visualize the point clouds.

```
alignedPtCloud = pctransform(movingPtCloud, tform);
figure
pcshowpair(alignedPtCloud,fixedPtCloud,'VerticalAxis','Y','VerticalAxisDir','Down')
```

Align Two Point Clouds Using LOAM Points

Read point cloud data from a Velodyne PCAP file into the workspace.

```
veloReader = velodyneFileReader("lidarData_ConstructionRoad.pcap","HDL32E");
```

Read the first point cloud from the data. Use this point cloud as the fixed point cloud.

```
fixedPtCloud = readFrame(veloReader,1);
```

Detect LOAM feature points in the fixed point cloud.

```
fixedPoints = detectLOAMFeatures(fixedPtCloud);
```

Downsample the less planar surface points from the fixed point cloud, to improve registration speed.

```
gridStep = 1;
fixedPoints = downsampleLessPlanar(fixedPoints,gridStep);
```

Read and detect LOAM feature points from the fifth point cloud in the data. Use this point cloud as the moving point cloud.

```
movingPtCloud = readFrame(veloReader,5);
movingPoints = detectLOAMFeatures(movingPtCloud);
```

Downsample the less planar surface points from the moving point cloud.

```
movingPoints = downsampleLessPlanar(movingPoints,gridStep);
```

Register the moving point cloud to the fixed point cloud.

```
tform = pcregisterloam(movingPoints,fixedPoints);
```

Transform the moving point cloud to align it to the fixed point cloud.

```
alignedPtCloud = pctransform(movingPtCloud,tform);
```

Visualize the aligned point clouds. Points from the fixed point cloud display as green, while points from the moving point cloud display as magenta.

```
figure
pcshowpair(alignedPtCloud,fixedPtCloud)
```


Input Arguments

movingPtCloud — Organized moving point cloud
pointCloud object

Organized moving point cloud, specified as a pointCloud object. The point cloud object must contain an organized point cloud with a Location property of size *M*-by-*N*-by-3 matrix, where *M* is the number of laser scans and *N* is the number of points per scan. The first page represents the *x*-coordinates, the second page represents the *y*-coordinates, and the third page represents the *z*-coordinates for each point.

fixedPtCloud — Organized fixed point cloud
pointCloud object

Organized fixed point cloud, specified as a `pointCloud` object. The point cloud object must contain an organized point cloud with a `Location` property of size *M*-by-*N*-by-3 matrix, where *M* is the number of laser scans and *N* is the number of points per scan. The first page represents the *x*-coordinates, the second page represents the *y*-coordinates, and the third page represents the *z*-coordinates for each point.

movingPoints — Moving LOAM points
LOAMPoints object

Moving LOAM points, specified as a `LOAMPoints` object. You can obtain the LOAM points from the moving point cloud by using the `detectLOAMFeatures` function, which detects LOAM feature points from organized point clouds.

fixedPoints — Fixed LOAM points
LOAMPoints object

Fixed LOAM points, specified as a `LOAMPoints` object. You can obtain the LOAM points from the fixed point cloud by using the `detectLOAMFeatures` function, which detects LOAM feature points from organized point clouds.

gridStep — Size of 3-D box for downsampling detected LOAM points
positive scalar

Size of the 3-D box for downsampling the detected LOAM points in the input point cloud, specified as a positive scalar.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `pcregisterloam(movingPoints,fixedPoints,MatchingMethod='one-to-one')` sets the registration matching method to `'one-to-one'`.

InitialTransform — Initial rigid transformation
rigidtform3d object

Initial rigid transformation, specified as a `rigidtform3d` object. Set the initial transformation to a coarse estimate. If you do not provide a coarse or initial estimate, the function uses a `rigidtform3d` object that contains a translation that moves the center of the moving points to the center of the fixed points.

MatchingMethod — Matching method
'one-to-one' (default) | 'one-to-many'

Matching method, specified as `'one-to-one'` or `'one-to-many'`.

- `'one-to-one'` — The algorithm selects the nearest neighbor as a match.
- `'one-to-many'` — The algorithm selects multiple nearest neighbors within the specified search radius as matches. Using the `'one-to-many'` method can increase registration accuracy, but at the cost of processing speed.

SearchRadius — Search radius for point matching
3 (default) | positive integer

Search radius for point matching, specified as a positive integer. When matching, the function finds the closest edge and surface points within the specified radius. For best results, set this value based on the certainty of the "InitialTransform" on page 3-0 . Increase the value of the SearchRadius when there is greater uncertainty in the initial transformation, but this can also decrease the registration speed.

MaxIterations — Maximum iterations before LOAM registration stops
20 (default) | positive integer

Maximum iterations before LOAM registration stops, specified as a positive integer.

Tolerance — Tolerance between consecutive LOAM iterations
[0.01,0.5] (default) | two-element vector

Tolerance between consecutive LOAM iterations, specified as a two-element vector with nonnegative values. The vector, [*Tdiff Rdiff*].

- *Tdiff* — Tolerance for the estimated absolute difference in translation and rotation estimated in consecutive LOAM iterations. Measures the Euclidean distance between two translation vectors.

- *Rdiff* — Tolerance for the estimated absolute difference of the angular rotation between consecutive iterations, measured in degrees.

The algorithm stops when the difference between the estimates of the rigid transformations in the most recent consecutive iterations falls below the specified tolerance value.

Verbose — Display progress information
false or 0 (default) | true or 1

Display progress information, specified as a numeric or logical 0 (false) or 1 (true). To display progress information, set Verbose to true.

Output Arguments

tform — Rigid 3-D transformation
rigidtform3d object

Rigid 3-D transformation, returned as a rigidtform3d object. tform describes the rigid 3-D transformation that registers the moving points to the fixed points.

rmse — Root mean squared error
positive scalar

Root mean squared error, returned as a positive scalar that represents the Euclidean distance between aligned points. A lower rmse value indicates a more accurate registration.

Version History
Introduced in R2022a

Supports rigidtform3d objects
Behavior changed in R2022b

You can now specify the `InitialTransform` name-value argument as a `rigidtform3d` object, which uses the premultiply convention. Although you can still specify `InitialTransform` as a `rigid3d` object, this object is not recommended because it uses the postmultiply convention. For more information, see "Migrate Geometric Transformations to Premultiply Convention".

When you specify `InitialTransform`, the `pcregisterloam` function returns `tform` as an object of the same type. When you do not specify `InitialTransform`, the `pcregisterloam` function returns `tform` as a `rigidtform3d` object. Before, the function returned `tform` as a `rigid3d` object.

References

[1] Zhang, Ji, and Sanjiv Singh. "LOAM: Lidar Odometry and Mapping in Real-Time." In *Robotics: Science and Systems X*. Robotics: Science and Systems Foundation, 2014. https://doi.org/10.15607/RSS.2014.X.007.

See Also

Objects
`rigidtform3d` | `pointCloud` | `LOAMPoints`

Functions
`detectLOAMFeatures` | `pcregistericp` | `pcregisterndt` | `pcregistercorr`

Topics
"Build a Map with Lidar Odometry and Mapping (LOAM) Using Unreal Engine Simulation"

detectLOAMFeatures

Detect LOAM feature points from 3-D lidar data

Syntax

```
points = detectLOAMFeatures(ptCloudOrg)
points = detectLOAMFeatures(ptCloudOrg,Name=Value)
```

Description

`points = detectLOAMFeatures(ptCloudOrg)` detects lidar odometry and mapping (LOAM) features in a point cloud based on curvature values. The function computes the curvature of each point using the closest neighbors of that point in the same laser scan. The curvature value of a feature point determines whether the function classifies it as a sharp edge, less sharp edge, planar surface, or less planar surface point.

`points = detectLOAMFeatures(ptCloudOrg,Name=Value)` specifies options using one or more name-value arguments. For example, `NumRegionsPerLaser=6` sets the number of regions to split each laser scan to `6`. Unspecified arguments have default values.

Examples

Detect and Visualize LOAM Feature Points

Load an organized lidar point cloud from a MAT file into the workspace.

```
ld = load("drivingLidarPoints.mat");
ptCloudOrg = ld.ptCloud;
```

Detect lidar odometry and mapping (LOAM) feature points.

```
points = detectLOAMFeatures(ptCloudOrg);
```

Visualize the LOAM points.

```
figure
show(points)
```

Align Two Point Clouds Using LOAM Points

Read point cloud data from a Velodyne PCAP file into the workspace.

```
veloReader = velodyneFileReader("lidarData_ConstructionRoad.pcap","HDL32E");
```

Read the first point cloud from the data. Use this point cloud as the fixed point cloud.

```
fixedPtCloud = readFrame(veloReader,1);
```

Detect LOAM feature points in the fixed point cloud.

```
fixedPoints = detectLOAMFeatures(fixedPtCloud);
```

Downsample the less planar surface points from the fixed point cloud, to improve registration speed.

```
gridStep = 1;
fixedPoints = downsampleLessPlanar(fixedPoints,gridStep);
```

Read and detect LOAM feature points from the fifth point cloud in the data. Use this point cloud as the moving point cloud.

```
movingPtCloud = readFrame(veloReader,5);
movingPoints = detectLOAMFeatures(movingPtCloud);
```

Downsample the less planar surface points from the moving point cloud.

```
movingPoints = downsampleLessPlanar(movingPoints,gridStep);
```

Register the moving point cloud to the fixed point cloud.

```
tform = pcregisterloam(movingPoints,fixedPoints);
```

Transform the moving point cloud to align it to the fixed point cloud.

```
alignedPtCloud = pctransform(movingPtCloud,tform);
```

Visualize the aligned point clouds. Points from the fixed point cloud display as green, while points from the moving point cloud display as magenta.

```
figure
pcshowpair(alignedPtCloud,fixedPtCloud)
```


Align Two Point Clouds Using ICP Algorithm With LOAM Points

Create a velodyneFileReader object.

```
veloReader = velodyneFileReader("lidarData_ConstructionRoad.pcap","HDL32E");
```

Read first and fifth point clouds.

```
fixedPtCloud = readFrame(veloReader,1);
movingPtCloud = readFrame(veloReader,5);
```

Detect LOAM feature points.

```
fixedPoints = detectLOAMFeatures(fixedPtCloud);
movingPoints = detectLOAMFeatures(movingPtCloud);
```

Create point cloud objects with the LOAM points.

```
fixedPtCloudLoam = pointCloud(fixedPoints.Location);
movingPtCloudLoam = pointCloud(movingPoints.Location);
```

Register the point clouds.

```
tform = pcregistericp(movingPtCloudLoam,fixedPtCloudLoam);
```

Transform the moving point cloud to align it to the fixed point cloud.

```
alignedPtCloud = pctransform(movingPtCloud,tform);
```

Visualize the aligned point clouds.

```
figure
pcshowpair(alignedPtCloud,fixedPtCloud)
```
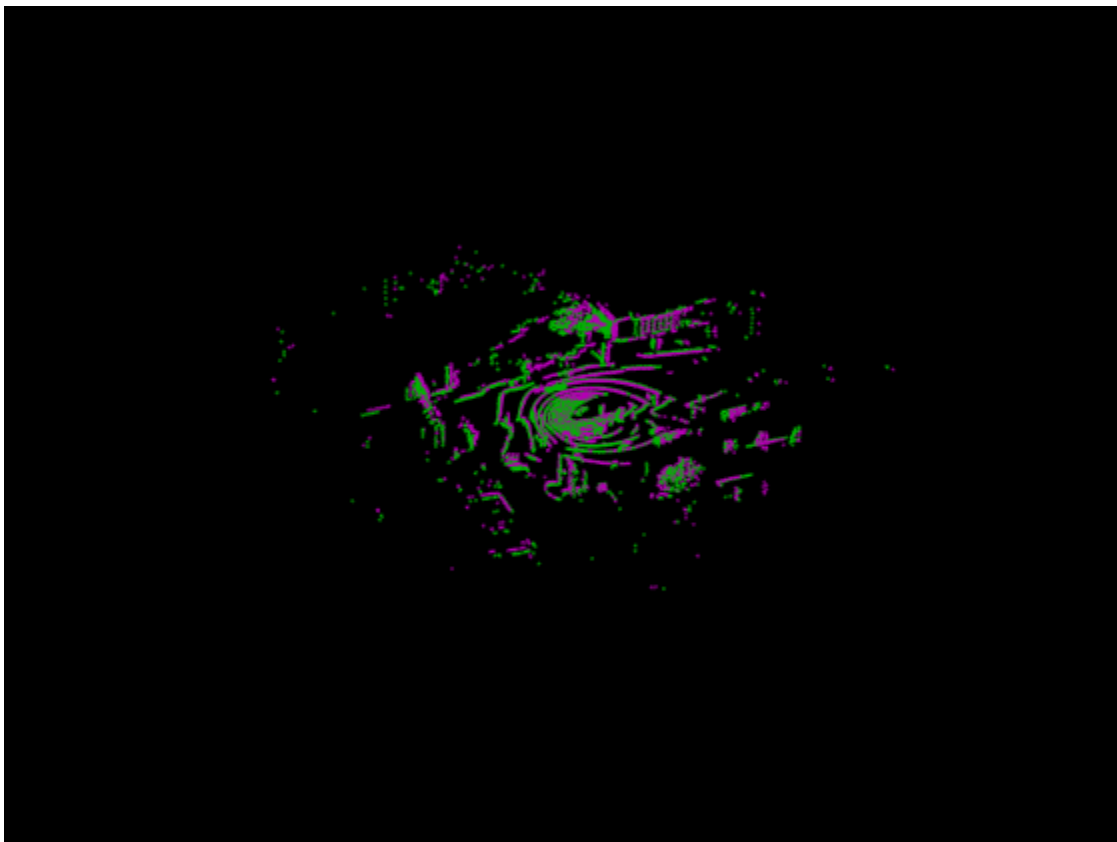

Input Arguments

ptCloudOrg — Organized point cloud
pointCloud object

Organized point cloud, specified as a `pointCloud` object. The point cloud object must contain an organized point cloud with a `Location` property of size *M*-by-*N*-by-3 matrix, where *M* is the number of laser scans and *N* is the number of points per scan. The first page represents the *x*-coordinates, the second page represents the *y*-coordinates, and the third page represents the *z*- coordinates for each point.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `detectLOAMFeatures(ptCloudOrg,NumRegionsPerLaser = 6)` sets the number of regions to split each laser scan into for feature point detection to 6.

NumRegionsPerLaser — Number of regions per laser scan
6 (default) | positive integer

Number of regions per laser scan for feature point detection, specified as a positive integer. The function splits each laser scan into regions that contain equal numbers of points. The algorithm detects up to the maximum number of points for each feature type in each region. You can specify the maximum number of sharp edge, less sharp edge, and planar surface points by using the `MaxSharpEdgePoints`, `MaxLessSharpEdgePoints`, and `MaxPlanarSurfacePoints` arguments, respectively. Increase the number of spatial regions to detect more edge points and planar surface points uniformly throughout the point cloud. Detecting more edge points and planar surface points can improve registration accuracy using LOAM points, but can decrease the processing speed of the function.

MaxSharpEdgePoints — Maximum number of sharp edge points per laser scan region
1 (default) | positive integer

Maximum number of sharp edge points per laser scan region, specified as a positive integer. The sharp edge points are the points with the highest curvatures.

MaxLessSharpEdgePoints — Maximum number of less sharp edge points per laser scan region
`MaxSharpEdgePoints*2` (default) | positive integer

Maximum number of less sharp edge points per laser scan region, specified as a positive integer. The less sharp edge points are the points with the highest curvature values after the sharp edge points.

MaxPlanarSurfacePoints — Maximum number of planar surface points per laser scan region
1 (default) | positive integer

Maximum number of planar surface points per laser scan region, specified as a positive integer. The planar surface points are the points with the lowest curvature values.

Output Arguments

points — LOAM feature points
LOAMPoints object

LOAM feature points, returned as a `LOAMPoints` object.

Tips

- Because LOAM feature point detection supports only organized point clouds, convert an unorganized point cloud into an organized point cloud by using the `pcorganize` function.

- The LOAM algorithm relies on the neighborhood of each point to compute its curvature and identify which points are on the boundaries of occluded regions. These points are considered unreliable points. Because of this unreliability, any preprocessing steps to the point clouds prior to feature point detection is not recommended.

- You can increase registration accuracy by increasing the maximum total number of feature points the function can detect. To increase the total number of feature points, increase the value of one or more of the `MaxSharpEdgePoints`, `MaxLessSharpEdgePoints`, and `MaxPlanarSurfacePoints` arguments. Note that this can also decrease the processing speed.

Algorithms

- The feature point detection algorithm supports VLP-16, HDL-32, and other spinning lidar sensors also known as surround sensors.

- The laser ID of each point corresponds to the laser that detects the point. For organized point clouds used with this algorithm, the `pointCloud Location` property stores the collected points as an *M*-by-*N*-by-3 matrix. Each row *M* represents a separate laser scan with *N* number of points, and 3 represents the *x*,*y*,*z* coordinates for each point.

- The algorithm uses the laser ID for point detection in `detectLOAMFeatures` and for point matching in `pcregisterloam`.

Version History

Introduced in R2022a

References

[1] Zhang, Ji, and Sanjiv Singh. "LOAM: Lidar Odometry and Mapping in Real-Time." In *Robotics: Science and Systems X*. Robotics: Science and Systems Foundation, 2014. https://doi.org/10.15607/RSS.2014.X.007.

See Also

Objects
LOAMPoints | pointCloud

Functions
pcregisterloam | pcorganize | pcregistericp | extractFPFHFeatures

Topics
"Build a Map with Lidar Odometry and Mapping (LOAM) Using Unreal Engine Simulation"

extractEigenFeatures

Extract eigenvalue-based features from point cloud segments

Syntax

```
features = extractEigenFeatures(ptCloud,labels)
features = extractEigenFeatures(segmentsIn)
[features,segmentsOut] = extractEigenFeatures( ___ )
[ ___ ] = extractEigenFeatures( ___ ,NormalizeEigenvalues=tf)
```

Description

`features = extractEigenFeatures(ptCloud,labels)` extracts eigenvalue-based features from a point cloud using labels, `labels`, that correspond to the segmented point cloud.

Eigenvalue-based features characterize geometrical features of point cloud segments. These features can be used in simultaneous localization and mapping (SLAM) applications for loop closure detection and localization in a target map.

`features = extractEigenFeatures(segmentsIn)` returns eigenvalue-based features from the point cloud segments `segmentsIn`. Use this syntax to facilitate the selection of specific segments in a point cloud scan for local feature extraction.

`[features,segmentsOut] = extractEigenFeatures( ___ )` additionally returns the segments extracted from the input point cloud using any combination of arguments from previous syntaxes. Use this syntax to facilitate visualization of the segments.

`[ ___ ] = extractEigenFeatures( ___ ,NormalizeEigenvalues=tf)` normalizes the eigenvalues prior to computing features, specified as `true` or `false`. Set `tf` to `true` when the next step is to use a classifier to assign a semantic label to a 3-D point. Set `tf` to `false` when the next step is to find matching features. The default value is `false`.

Examples

Compute Eigenvalue-Based Features from Normalized Eigenvalues

Load an organized lidar point cloud.

```
ld = load('drivingLidarPoints.mat');
ptCloud = ld.ptCloud;
```

Segment and remove the ground plane.

```
groundPtsIdx = segmentGroundFromLidarData(ptCloud,'ElevationAngleDelta',15);
ptCloud = select(ptCloud,~groundPtsIdx,'OutputSize','full');
```

Cluster the remaining points with a minimum of 50 points per cluster.

```
distThreshold = 0.5; % in meters
minPoints = 50;
[labels,numClusters] = segmentLidarData(ptCloud,distThreshold,'NumClusterPoints',minPoints);
```

Compute eigenvalue-based features.

```
features = extractEigenFeatures(ptCloud,labels,'NormalizeEigenvalues',true)
```

```
features=17×1 object
  16x1 eigenFeature array with properties:

    Feature
    Centroid
       ⋮
```

Match Eigenvalue-Based Features Between Point Clouds

Create a Velodyne PCAP file reader.

```
veloReader = velodyneFileReader('lidarData_ConstructionRoad.pcap','HDL32E');
```

Read the first and fourth scans from the file.

```
ptCloud1 = readFrame(veloReader,1);
ptCloud2 = readFrame(veloReader,4);
```

Remove the ground plane from the scans.

```
maxDistance = 1; % in meters
referenceVector = [0 0 1];
[~,~,selectIdx] = pcfitplane(ptCloud1,maxDistance,referenceVector);
ptCloud1 = select(ptCloud1,selectIdx,'OutputSize','full');
[~,~,selectIdx] = pcfitplane(ptCloud2,maxDistance,referenceVector);
ptCloud2 = select(ptCloud2,selectIdx,'OutputSize','full');
```

Cluster the point clouds with a minimum of 10 points per cluster.

```
minDistance = 2; % in meters
minPoints = 10;
labels1 = pcsegdist(ptCloud1,minDistance,'NumClusterPoints',minPoints);
labels2 = pcsegdist(ptCloud2,minDistance,'NumClusterPoints',minPoints);
```

Extract eigen-value features and the corresponding segments from each point cloud.

```
[eigFeatures1,segments1] = extractEigenFeatures(ptCloud1,labels1);
[eigFeatures2,segments2] = extractEigenFeatures(ptCloud2,labels2);
```

Create matrices of the features and centroids extracted from each point cloud, for matching.

```
features1 = vertcat(eigFeatures1.Feature);
features2 = vertcat(eigFeatures2.Feature);
centroids1 = vertcat(eigFeatures1.Centroid);
centroids2 = vertcat(eigFeatures2.Centroid);
```

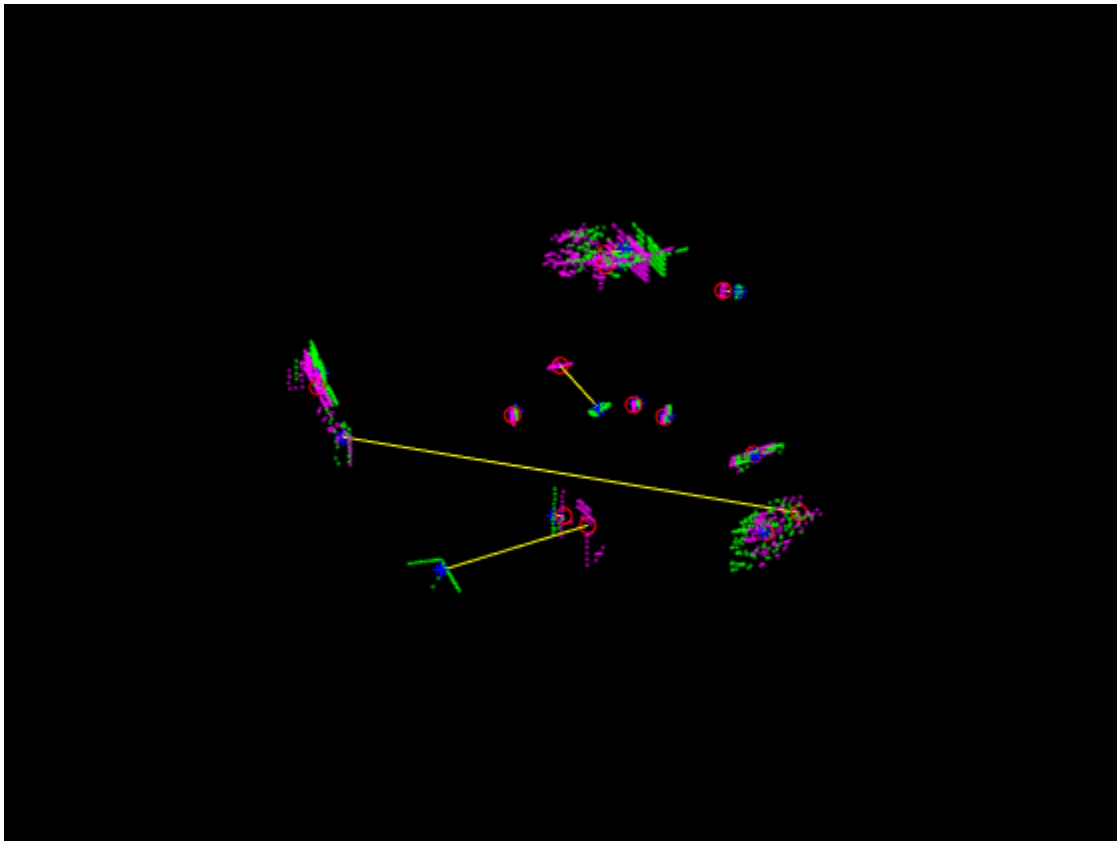Find putative feature matches.

```
indexPairs = pcmatchfeatures(features1,features2, ...
    pointCloud(centroids1),pointCloud(centroids2));
```

Get the matched segments and features for visualization.

```
matchedSegments1 = segments1(indexPairs(:,1));
matchedSegments2 = segments2(indexPairs(:,2));
matchedFeatures1 = eigFeatures1(indexPairs(:,1));
matchedFeatures2 = eigFeatures2(indexPairs(:,2));
```

Visualize the matches.

```
figure
pcshowMatchedFeatures(matchedSegments1,matchedSegments2,matchedFeatures1,matchedFeatures2)
title('Matched Segments')
```


Input Arguments

ptCloud — Point cloud
pointCloud object

Point cloud, specified as a pointCloud object.

labels — Cluster labels
M-element vector of numeric values | *M*-by-*N* matrix of numeric values

Cluster labels, specified as an *M*-element vector of numeric values for unorganized point clouds or an *M*-by-*N* matrix of numeric values for organized point clouds. The labels correspond to the results of segmenting the input point cloud. Each point in the point cloud has a cluster label, specified by the corresponding element in `labels`.

You can use the `pcsegdist` or the `segmentLidarData` function to return labels.

segmentsIn — Point cloud segments
vector of `pointCloud` objects

Point cloud segments, specified as a vector of `pointCloud` objects. Each point cloud segment in the input must have a minimum of two points for feature extraction. No features or segments are returned for input segments with only one point.

Output Arguments

features — Eigenvalue-based features
vector of `eigenFeature` objects

Eigenvalue-based features, returned as a vector of `eigenFeature` objects. When you extract features from a labeled point cloud input, each element in this vector contains the features extracted from the corresponding cluster of labeled points. When you extract features from a segments input, each element in this vector contains the features extracted from the corresponding element in the segments vector.

segmentsOut — Segments extracted from point cloud
vector of `pointCloud` objects

Segments extracted from the point cloud, specified as a vector of `pointCloud` objects. The length of the segments vector corresponds to the number of nonzero, unique labels.

Version History
Introduced in R2021a

References

[1] Weinmann, M., B. Jutzi, and C. Mallet. "Semantic 3D Scene Interpretation: A Framework Combining Optimal Neighborhood Size Selection with Relevant Features." *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences* II–3 (August 7, 2014): 181–88. https://doi.org/10.5194/isprsannals-II-3-181-2014.

See Also

Functions
scanContextDescriptor | pcmatchfeatures | pcshowMatchedFeatures | extractFPFHFeatures | segmentLidarData | pcsegdist

Objects
pointCloud | eigenFeature | pcmapsegmatch

Topics
"Build Map and Localize Using Segment Matching"
"Implement Point Cloud SLAM in MATLAB"

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',[25 75]` sets the angular range for the azimuth angles of the function.

Examples

Fit Cuboid Over Point Cloud Data

Fit cuboid 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 pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: 'AzimuthRange',[25 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.

Version History
Introduced in R2020b

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™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Functions
pcfitplane | pcfitcylinder | pcfitsphere

Objects
pointCloud | cuboidModel

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.

```
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
```
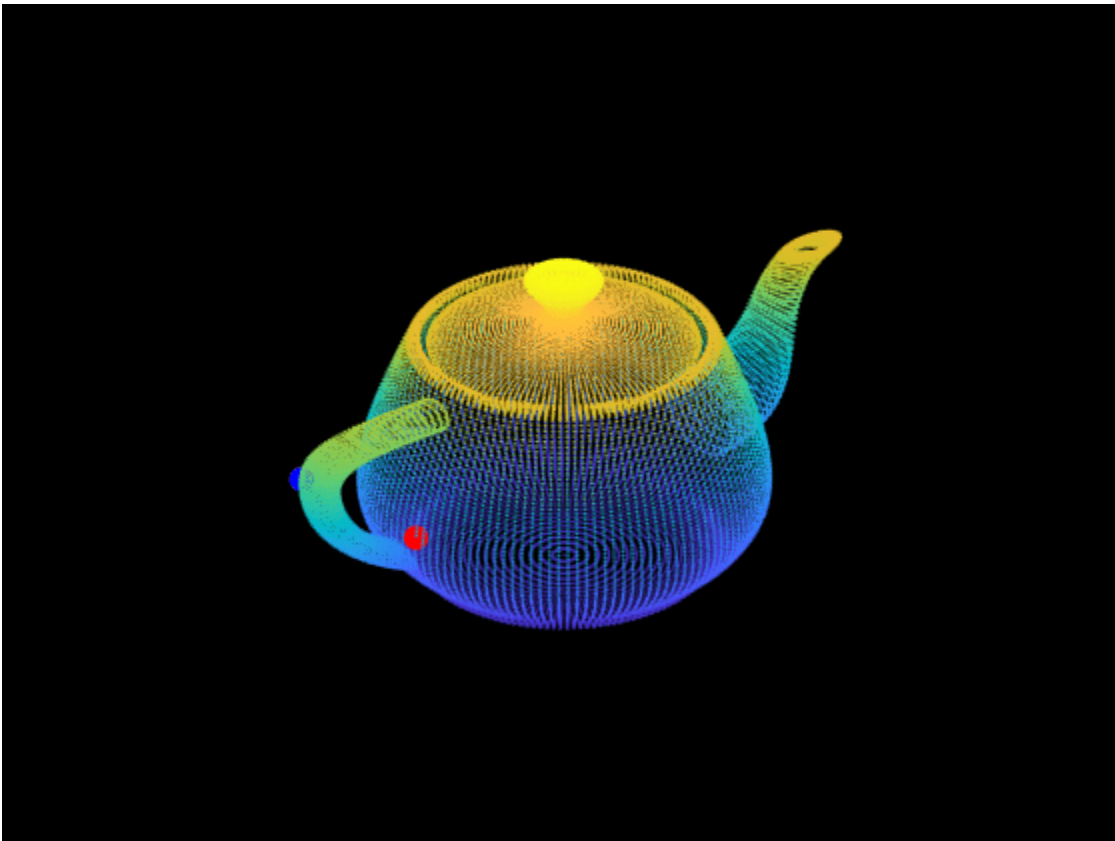

Display the extracted FPFH descriptors at key points.

```
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')
```
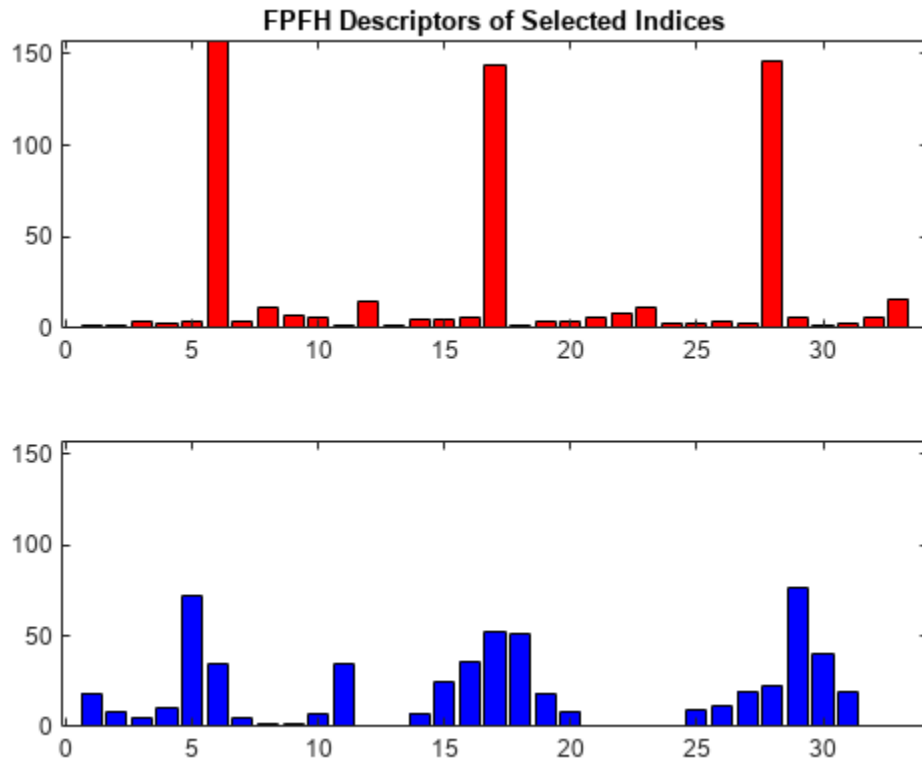
Input Arguments

`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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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

Version History
Introduced in R2020b

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
pcread | pcdownsample | pcnormals | pcshow

Objects
pointCloud

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')
```
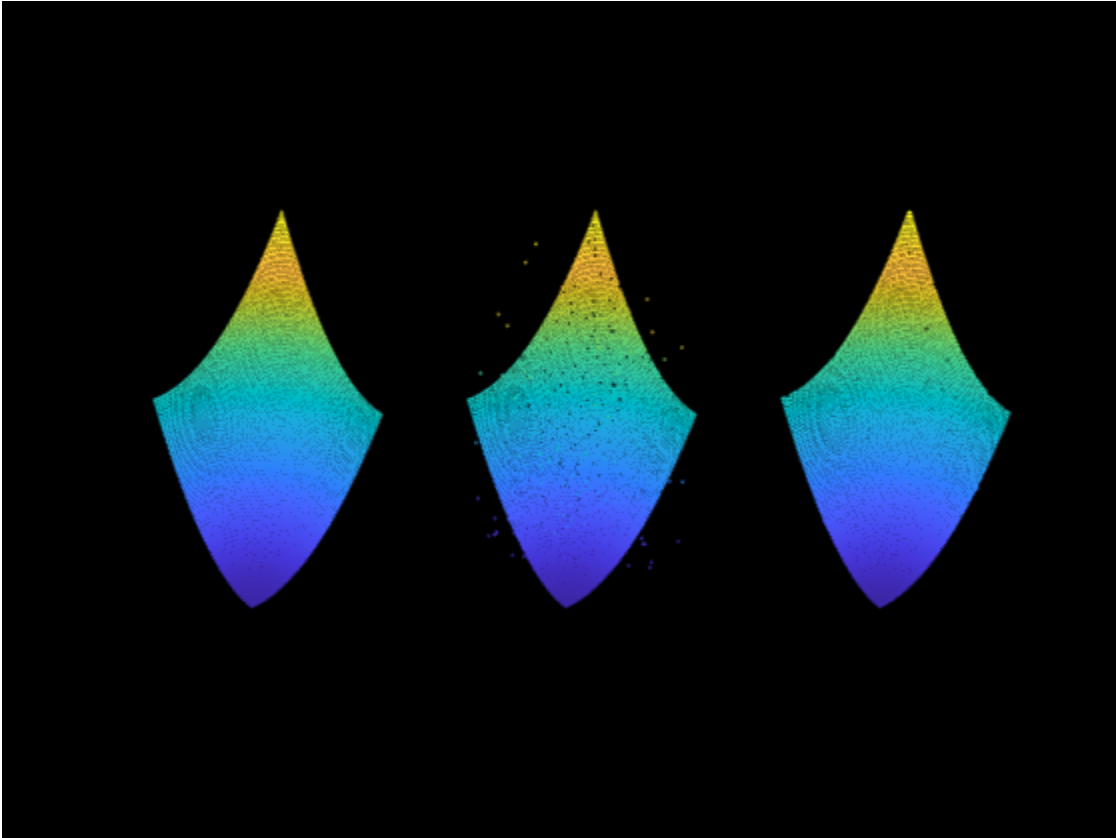

Apply Median Filter on Unorganized Point Cloud Data

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')
```
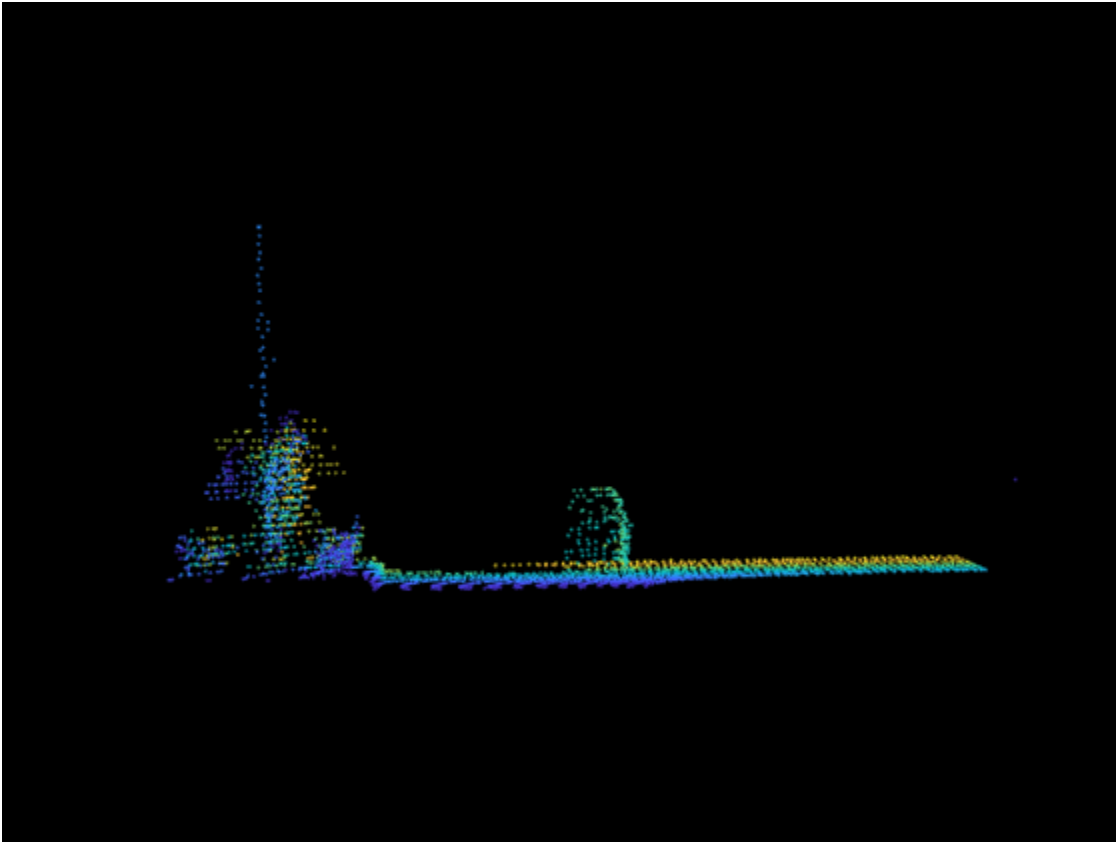
Add noise along the z-channel in the interval (a,b). Values of a and b are chosen to make the noise appear close to the ground.

```
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')
```
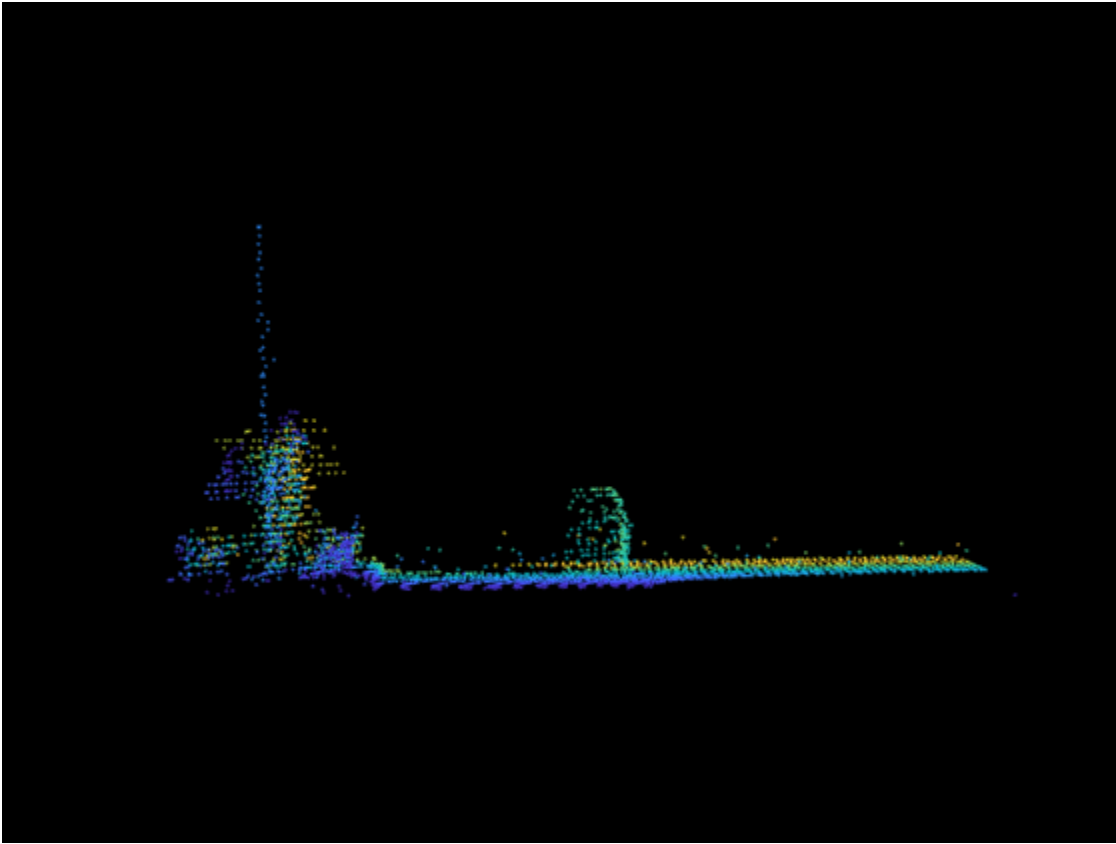
Apply median filter on the point cloud.

```
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')
```
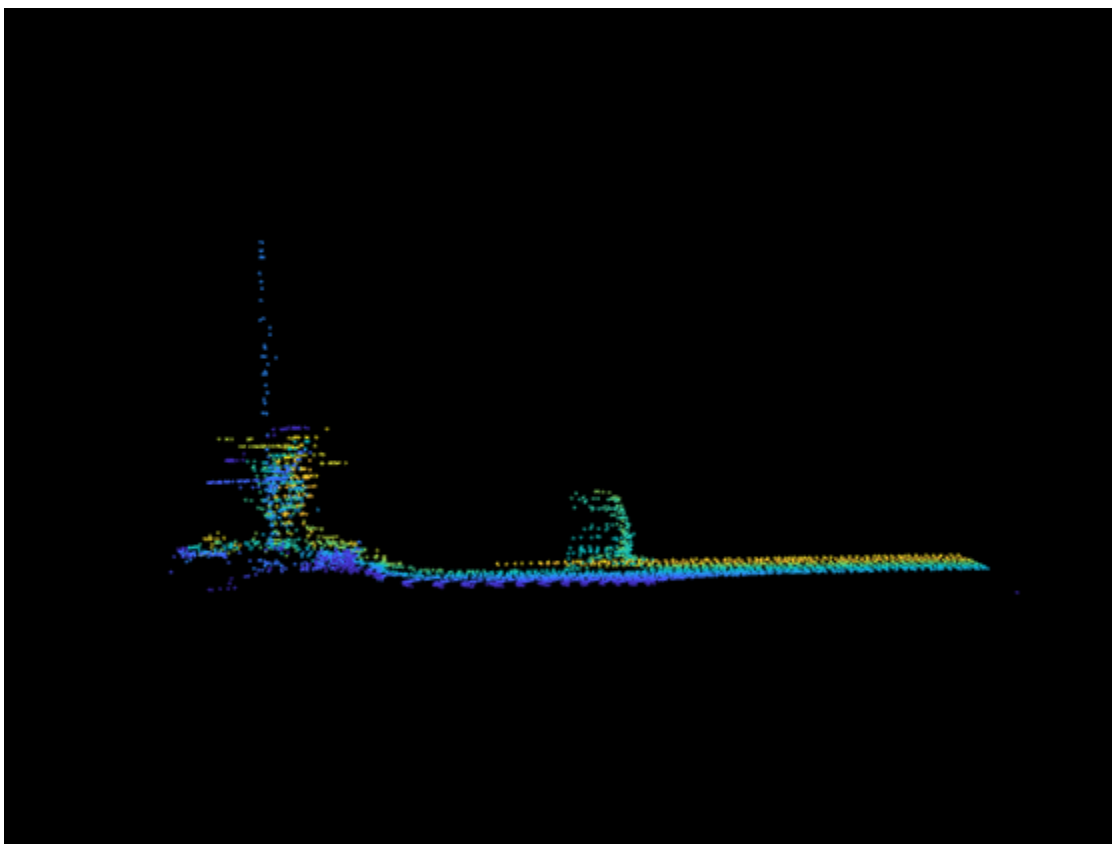
Input Arguments

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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. The computation time increases when there are a lot of points inside the specified radius. So, large radius values for dense point clouds can cause high computation time and impact performance.

Data Types: `single` | `double`

Output Arguments

ptCloudOut — Filtered point cloud
`pointCloud` object

Filtered point cloud, returned as a `pointCloud` object.

Version History
Introduced in R2020b

Extended Capabilities

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

See Also

Functions
pcdenoise | pcdownsample | pcmerge | pcshow | pctransform

Objects
pointCloud

estimateCheckerboardCorners3d

Estimate world frame coordinates of checkerboard corner points in image

Syntax

```
imageCorners3d = estimateCheckerboardCorners3d(I,cameraIntrinsic,checkerSize)
[imageCorners3d,boardDimensions] = estimateCheckerboardCorners3d(I,
cameraIntrinsic,checkerSize)
[imageCorners3d,boardDimensions,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`.

`[imageCorners3d,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 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,rigidtform3d);
J = undistortImage(Image,intrinsic.cameraParams);
imshow(J)
hold on
plot(imPts(:,1),imPts(:,2),".r","MarkerSize",30)
title("Detected Checkerboard Corners")
hold off
```

Detected Checkerboard Corners

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.
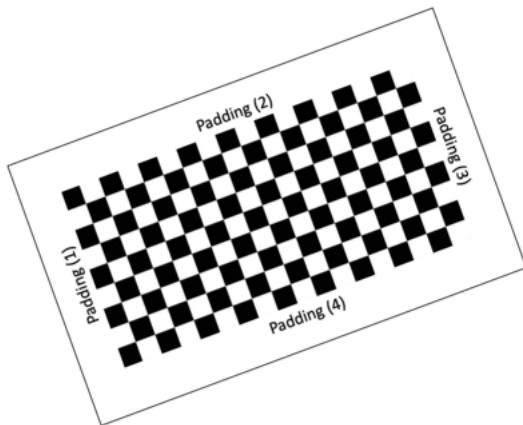
Checkerboard Padding

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.

3-97

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.

Limitations

* Partial detection of checkerboard is not supported.

Version History
Introduced in R2020b

See Also

Functions
detectRectangularPlanePoints | estimateLidarCameraTransform

Topics
"Lidar and Camera Calibration"
"Calibration Guidelines"
"What Is Lidar-Camera Calibration?"

detectRectangularPlanePoints

Detect rectangular plane of specified dimensions in point cloud

Syntax

```
ptCloudPlanes = detectRectangularPlanePoints(ptCloudIn,planeDimensions)
[ptCloudPlanes,ptCloudUsed] = detectRectangularPlanePoints(ptCloudArray,
planeDimensions)
[ ___ ] = detectRectangularPlanePoints(ptCloudFileNames,planeDimensions)
[ptCloudPlanes,ptCloudUsed,indicesCell] = detectRectangularPlanePoints( ___ )
[ ___ ] = 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 returns 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.

`[ptCloudPlanes,ptCloudUsed,indicesCell] = detectRectangularPlanePoints( ___ )` 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 input 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')
```
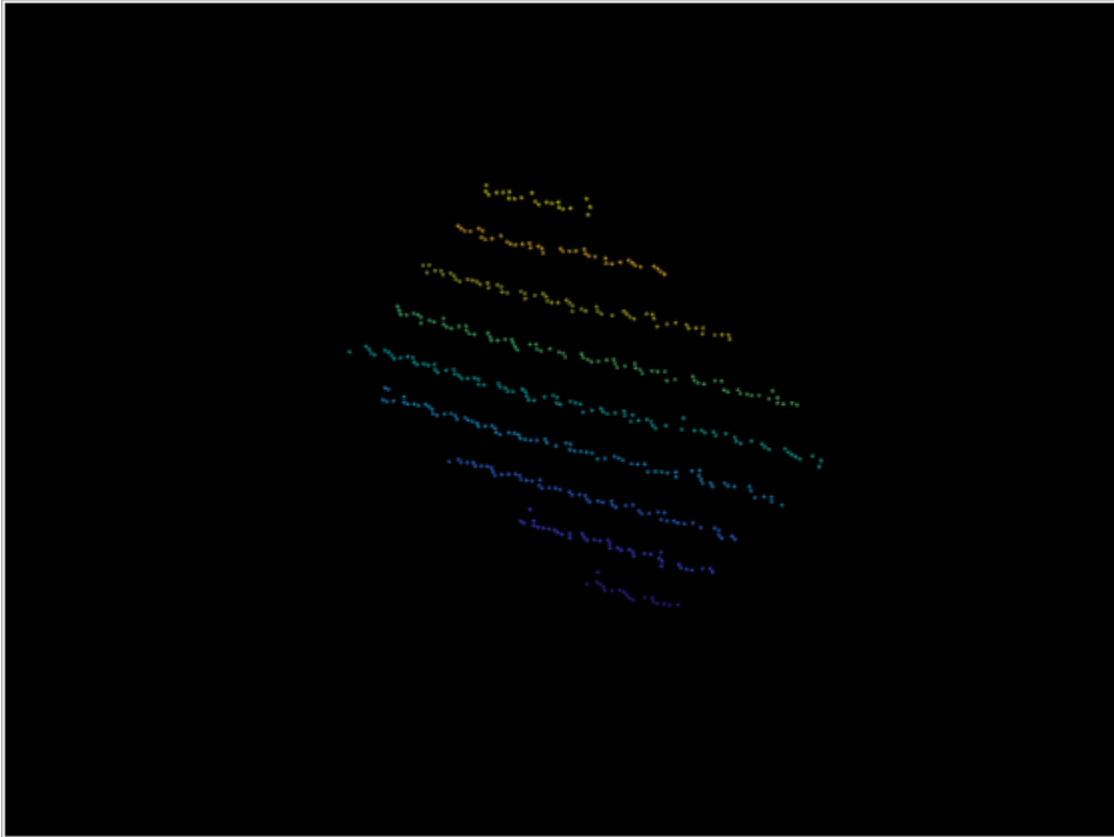
Visualize the detected rectangular plane on the input point cloud.

```
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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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. Low resolution lidars require higher `'MinDistance'` threshold and vice-versa.

Note This value should be greater than the minimum distance between two scan lines of the checkerboard. Too small value for `'MinDistance'` might result in incorrect detections.

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` or `0` (default) | `true` or `1`

Remove the ground plane from the point cloud, specified as the comma-separated pair consisting of `'RemoveGround'` and a logical `0` (`false`) or `1` (`true`).

The normal of the plane is assumed to be aligned with the positive direction of the *z*-axis with the reference vector [0 0 1].

Data Types: `logical`

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 `1` (`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.

Version History
Introduced in R2020b

See Also

Functions
estimateCheckerboardCorners3d | estimateLidarCameraTransform | projectLidarPointsOnImage

Topics
"Lidar and Camera Calibration"

estimateLidarCameraTransform

Estimate rigid transformation from lidar sensor to camera

Syntax

```
tform = estimateLidarCameraTransform(ptCloudPlanes,imageCorners)
tform = estimateLidarCameraTransform(ptCloudPlanes,imageCorners,intrinsics)
[tform,errors] = estimateLidarCameraTransform( ___ )
[ ___ ] = estimateLidarCameraTransform( ___ ,Name,Value)
```

Description

`tform = estimateLidarCameraTransform(ptCloudPlanes,imageCorners)` estimates the transformation between a lidar sensor and a camera using the checkerboard planes extracted from lidar sensor data and 3-D image corners of the checkerboard extracted from camera data, respectively.

`tform = estimateLidarCameraTransform(ptCloudPlanes,imageCorners,intrinsics)` uses the checkerboard planes extracted from a lidar sensor, 2-D or 3-D image corners of the checkerboard extracted from a camera, and the camera intrinsic parameters to estimate the transformation between the lidar sensor and the camera.

`[tform,errors] = estimateLidarCameraTransform( ___ )` returns the inaccuracy in estimating the transformation matrix `errors` using any combination of input arguments in previous syntaxes.

`[ ___ ] = estimateLidarCameraTransform( ___ ,Name,Value)` specifies options using one or more name-value 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.

```
imagFileNames = imageFileNames(imagesUsed);
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,intrinsic.cameraParams);
```

Display translation, rotation, and reprojection errors as bar graphs.

```
figure
bar(errors.TranslationError)
xlabel('Frame Number')
title('Translation Error (meters)')
```
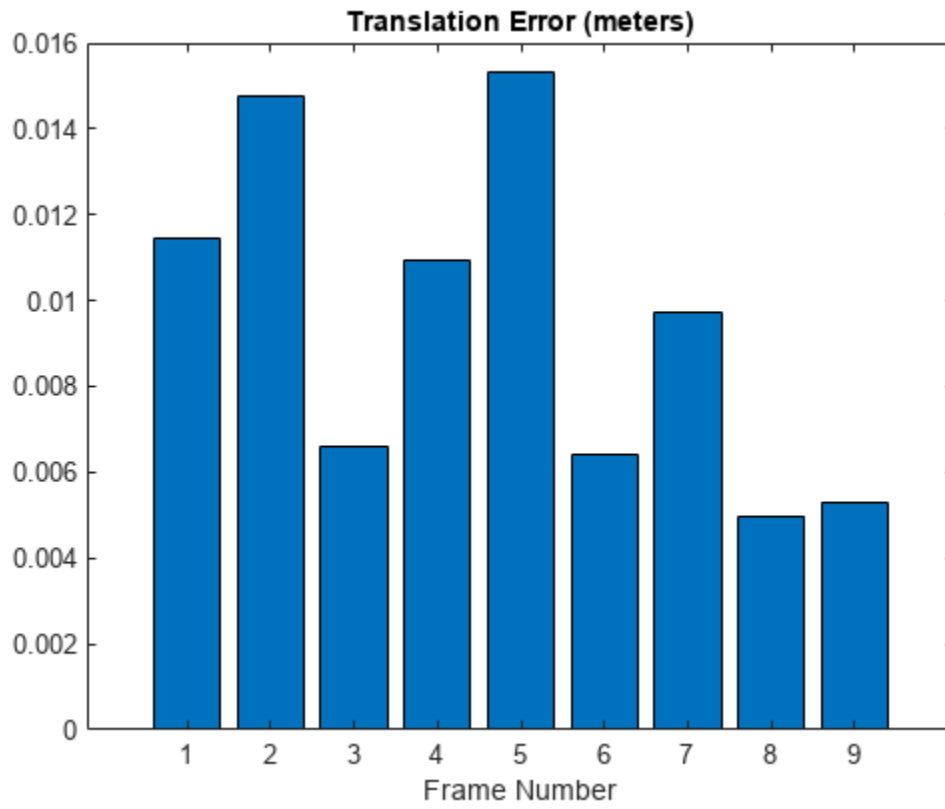
```
figure
bar(errors.RotationError)
xlabel('Frame Number')
title('Rotation Error (degrees)')
```
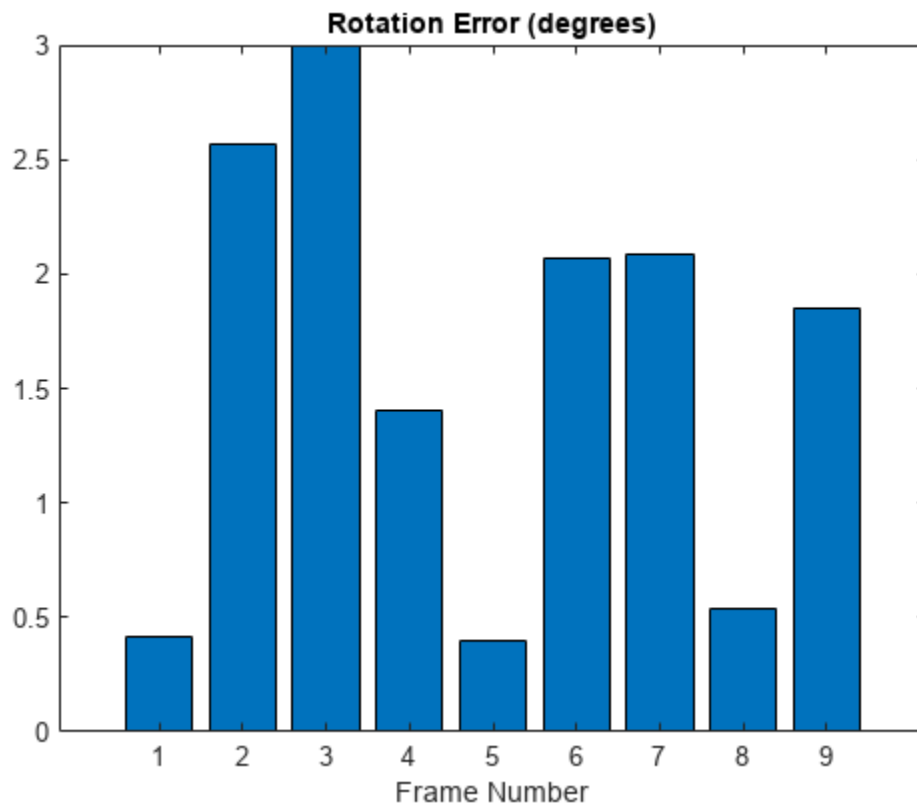
```
figure
bar(errors.ReprojectionError)
xlabel('Frame Number')
title('Reprojection Error (pixels)')
```

Reprojection Error (pixels)
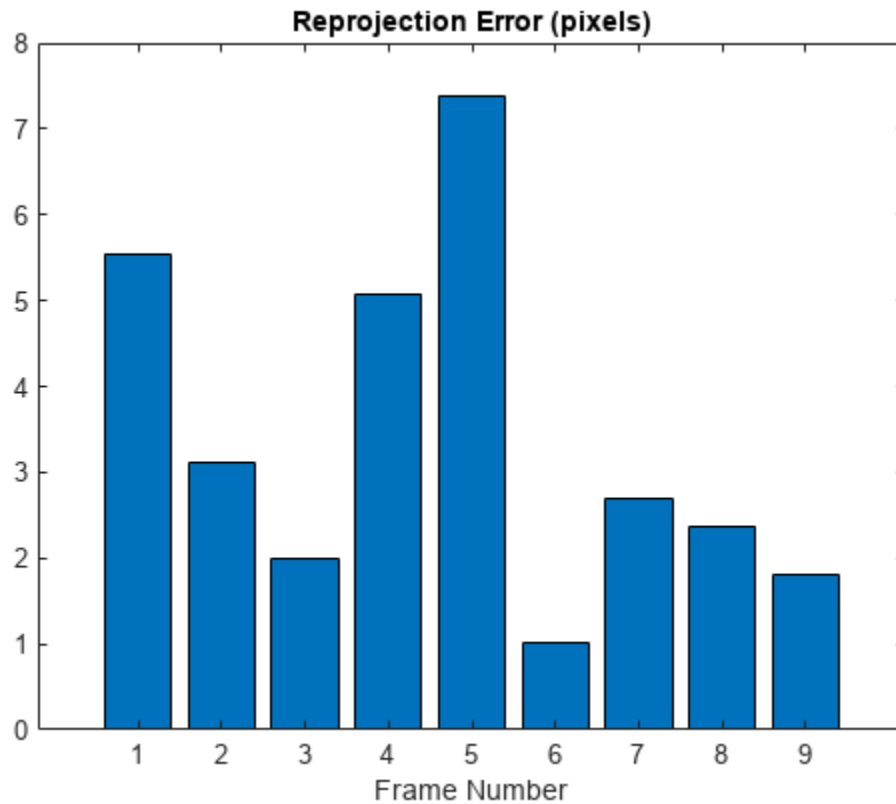
Input Arguments

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 `imageCorners` arguments. This means that number of point clouds and number of images used for detection must also be equal.

imageCorners — Checkerboard corners extracted from camera data
4-by-2-by-*P* array | 4-by-3-by-*P* array

Checkerboard corners extracted from camera data, specified in 2-D or 3-D coordinates.

- 2-D coordinates, specified as a 4-by-2-by-*P* array. Each row of a channel is of the form of [*x y*], of a checkerboard corner extracted from the corresponding camera image. The values are in pixel coordinates

- 3-D coordinates, specified as a 4-by-3-by-*P* array. Each row of a channel is of the form of [*x y z*], of a checkerboard corner extracted from the corresponding camera image. The values are in world coordinate system.

3-109

P represents the number of camera images used for detection. *P* must be equal for both the `ptCloudPlanes` and `imageCorners` arguments. This means that number of point clouds and number of images used for detection must also be equal.

Note When `imageCorners` value is in 2-D coordinates, you must specify the camera intrinsic parameters, `intrinsics`.

Data Types: `single` | `double`

`intrinsics` — Camera intrinsic parameters
`cameraIntrinsics` object

Camera intrinsics parameters, specified as a `cameraIntrinsics` object.

Note When `imageCorners` value is in 2-D coordinates, you must specify the camera intrinsic parameter, `intrinsics`. When the `imageCorners` are in 3-D coordinates, `intrinsics` is an optional input.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `Verbose=true` sets the function to display progress.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 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 `rigidtform3d` object (default) | `rigidtform3d` object

Initial rigid transformation, specified as a `rigidtform3d` 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.

`Verbose` — Display function progress
`false` or `0` (default) | `true` or `1`

Display function progress, specified as a logical 0 (false) or logical 1 (true).

Data Types: logical

Output Arguments

tform — Lidar to camera rigid transformation
rigidtform3d object

Lidar to camera rigid transformation, returned as a rigidtform3d object. The returned object registers the point cloud data from a lidar sensor to the coordinate frame of a camera.

errors — Inaccuracy in the transformation matrix estimation
structure | *P*-element numeric array

Inaccuracy of the transformation matrix estimation, returned as a structure or *P*-element numeric array.

- The function returns a structure when image corners are in 3-D coordinates. 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 intrinsics argument, 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.

- For 2-D image corners, the function only returns the reprojection error.

Version History
Introduced in R2020b

Supports rigidtform3d objects
Behavior changed in R2022b

You can now specify the InitialTransform name-value argument as a rigidtform3d object, which uses the premultiply convention. Although you can still specify InitialTransform as a rigid3d object, this object is not recommended because it uses the postmultiply convention. For more information, see "Migrate Geometric Transformations to Premultiply Convention".

When you specify InitialTransform, the estimateLidarCameraTransform function returns tform as an object of the same type. When you do not specify InitialTransform, the

estimateLidarCameraTransform function now returns `tform` as a `rigidtform3d` object. Before, the function returned `tform` as a `rigid3d` object.

Support for 2-D image corners input

You can specify the `imageCorners` input in 2-D coordinates. You must additionally specify the camera intrinsic parameters by using the `instrinsics` input to estimate the transformation matrix when image corners are 2-D.

See Also

Functions
detectRectangularPlanePoints | estimateCheckerboardCorners3d | projectLidarPointsOnImage | fuseCameraToLidar | bboxCameraToLidar

Topics
"Lidar and Camera Calibration"

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 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);
```
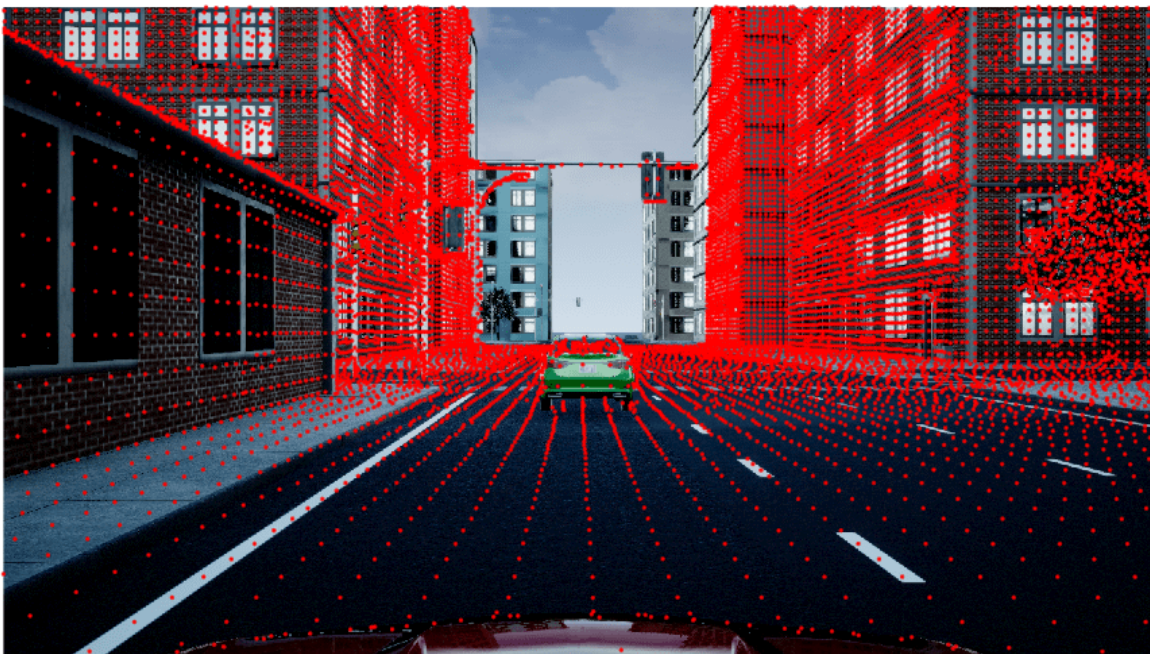
Downsample the point cloud data.

3-113

```
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
`rigidtform3d` object

Lidar to camera rigid transformation, specified as a `rigidtform3d` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `ImageSize=[250 400]` sets the size of the image on which to project the points to 250-by-400 pixels.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 a vector of positive integers.

Data Types: `single` | `double`

`ImageSize` — Size of image on which points are projected
`intrinsics.ImageSize` (default) | two-element row vector

Size of the image on which the points are projected, specified as 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 `ImageSize` property from the camera intrinsic parameters `intrinsics` to estimate the field of view of the camera.

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`

Version History
Introduced in R2020b

Supports `rigidtform3d` objects

You can now specify `tform` as a `rigidtform3d` object, which uses the premultiply convention. Although you can still specify `tform` as a `rigid3d` object, this object is not recommended because it uses the postmultiply convention. For more information, see "Migrate Geometric Transformations to Premultiply Convention".

Extended Capabilities

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

See Also

Functions
detectRectangularPlanePoints | estimateLidarCameraTransform | estimateCheckerboardCorners3d | fuseCameraToLidar | bboxCameraToLidar

Topics
"Lidar and Camera Calibration"

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( ___ )
[ptCloudOut,colormap,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.

`[ptCloudOut,colormap,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 output arguments from previous syntaxes.
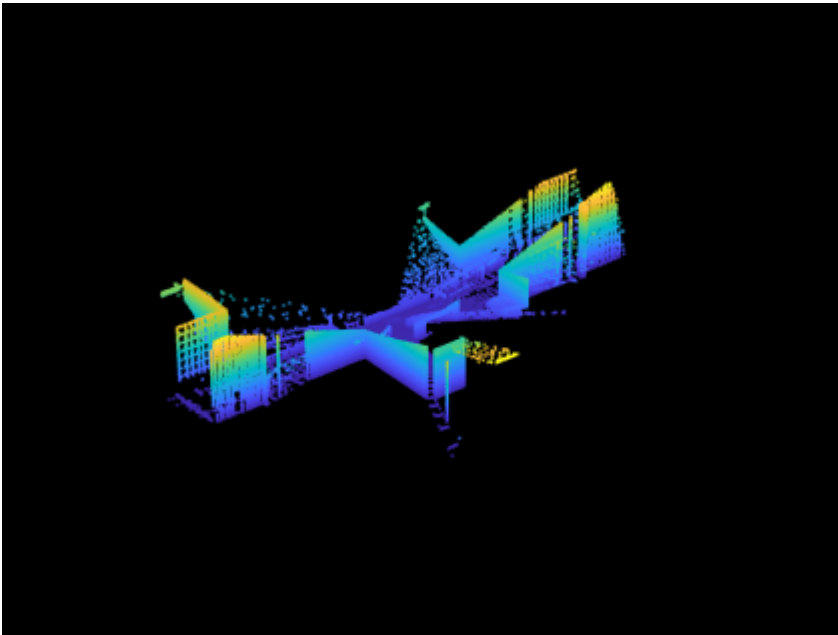
Examples

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.
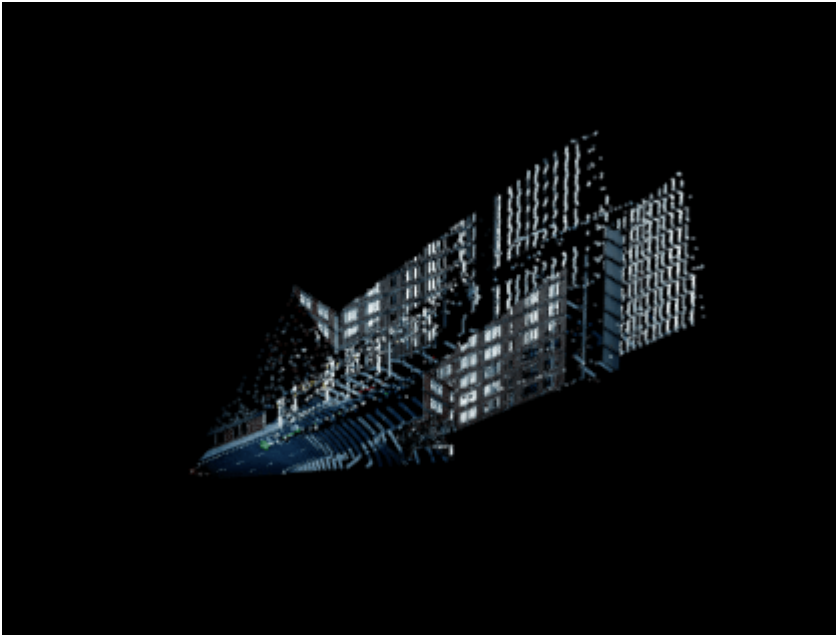
```
intrinsics = gt.camParams;
camToLidar = gt.tform;
```

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
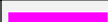`rigidtform3d` object

Camera to lidar rigid transformation, specified as a `rigidtform3d` 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

Version History
Introduced in R2020b

Supports `rigidtform3d` objects

You can now specify `tform` as a `rigidtform3d` object, which uses the premultiply convention. Although you can still specify `tform` as a `rigid3d` object, this object is not recommended because it uses the postmultiply convention. For more information, see "Migrate Geometric Transformations to Premultiply Convention".

Extended Capabilities

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

See Also

Functions
detectRectangularPlanePoints | estimateLidarCameraTransform | estimateCheckerboardCorners3d | projectLidarPointsOnImage | bboxCameraToLidar

Topics
"Lidar and Camera Calibration"

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 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)
```
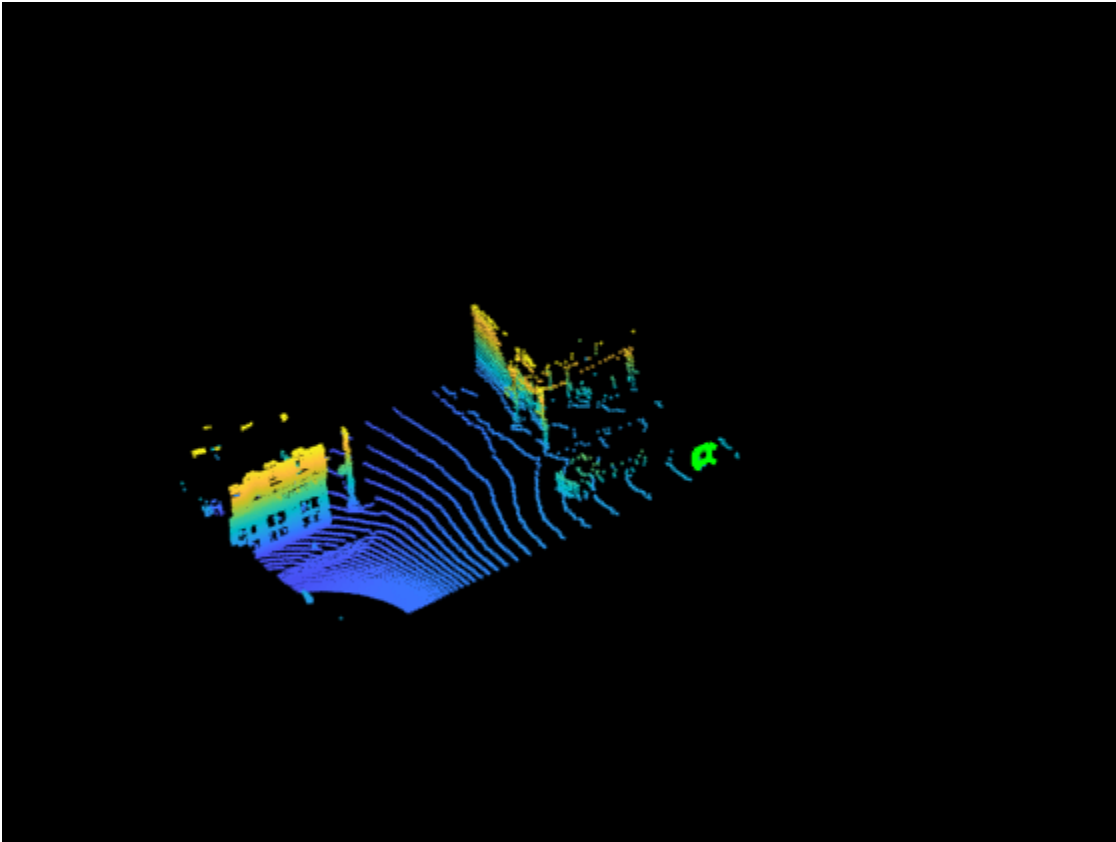

Estimate the bounding box in the point cloud.

```
[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
rigidtform3d object

Camera to lidar rigid transformation, specified as a `rigidtform3d` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `ClusterThreshold=0.5` sets the Euclidean distance threshold for differentiating point cloud clusters to 0.5 world units.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 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 a two-element vector of 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.

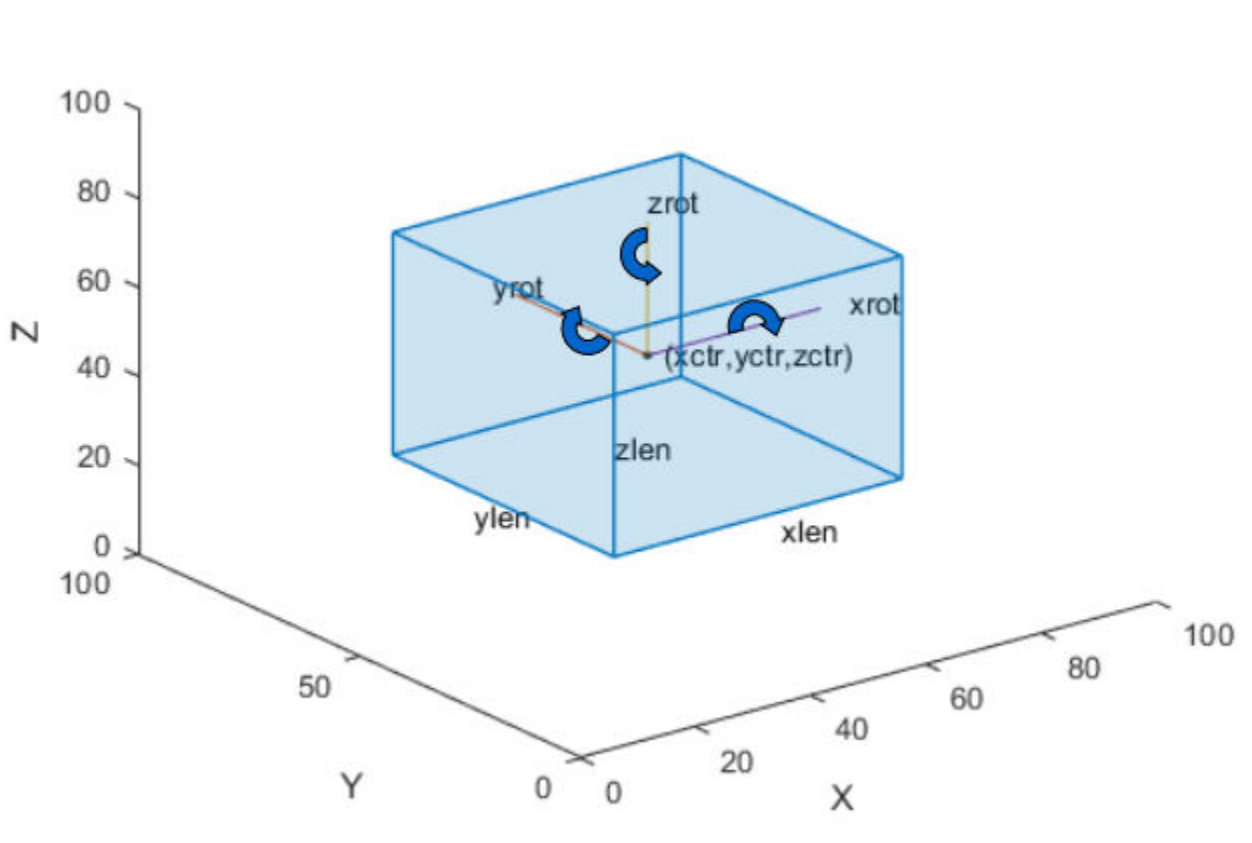Data Types: single | double

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 — Bounding box detection flag
M-element row vector of logicals

Bounding box 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`

Version History
Introduced in R2020b

Supports `rigidtform3d` objects

You can now specify `tform` as a `rigidtform3d` object, which uses the premultiply convention. Although you can still specify `tform` as a `rigid3d` object, this object is not recommended because it uses the postmultiply convention. For more information, see "Migrate Geometric Transformations to Premultiply Convention".

Extended Capabilities

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

See Also

Functions
bboxLidarToCamera | projectLidarPointsOnImage | fuseCameraToLidar

Topics
"Lidar and Camera Calibration"

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 Corresponding Features in Point Clouds

This example shows how to match corresponding point cloud features using the `pcmatchfeatures` function.

Preprocessing

Read point cloud data into the workspace.

```
ptCld = pcread("teapot.ply");
```

Downsample the point cloud.

```
ptCloud = pcdownsample(ptCld,"gridAverage",0.05);
```

Transform and create a new point cloud using the transformation matrix A.

```
A = [cos(pi/6) -sin(pi/6) 0 5; ...
     sin(pi/6)  cos(pi/6) 0 5; ...
             0          0  1 10; ...
             0          0  0 1];
tform = affinetform3d(A);
ptCloudTformed = pctransform(ptCloud,tform);
```

Visualize the two point clouds.

```
pcshowpair(ptCloud,ptCloudTformed);
legend("Original", "Transformed","TextColor",[1 1 0]);
```
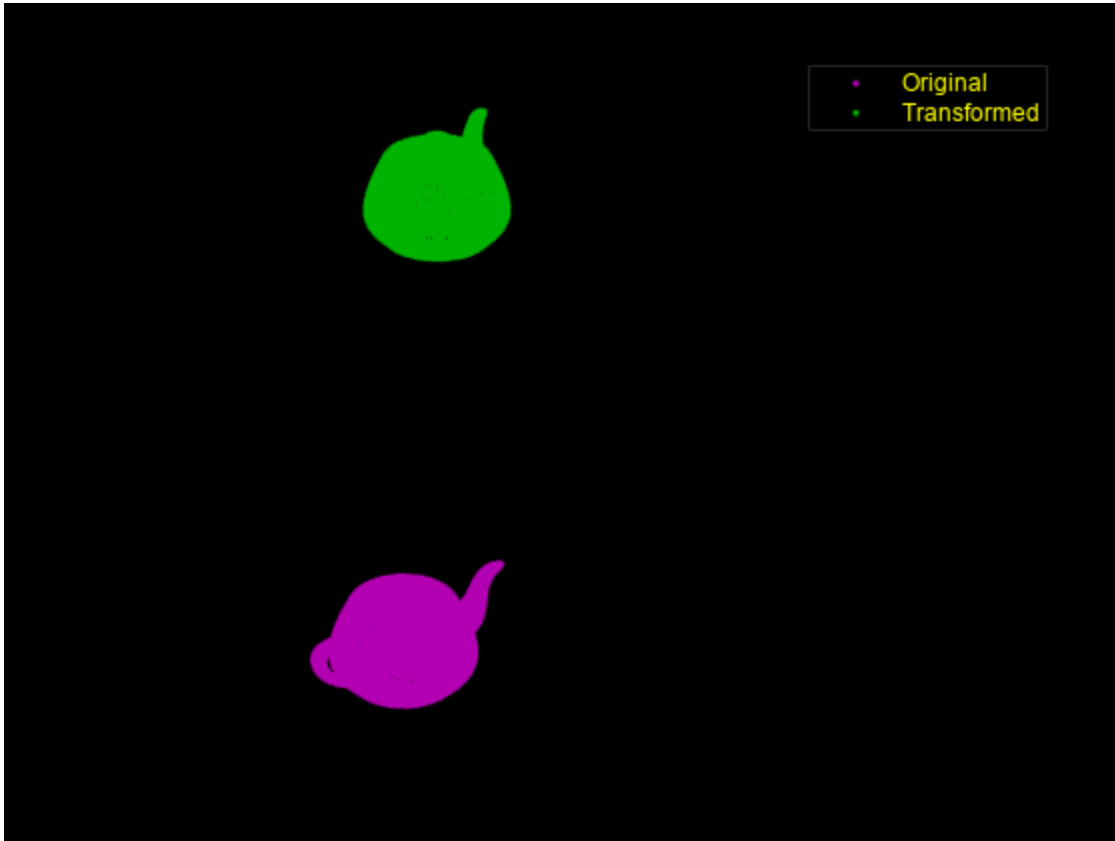

Match Corresponding Features

In the preprocessing section, we created a second point cloud by translating and rotating the original point cloud. In this section, we use the `pcmatchfeatures` function to find matching features between these point clouds.

Extract features from both the point clouds using the `extractFPFHFeatures` function.

```
fixedFeature = extractFPFHFeatures(ptCloud);
movingFeature = extractFPFHFeatures(ptCloudTformed);
length(movingFeature)
```

`ans = 16578`

Find matching features.

```
[matchingPairs,scores] = pcmatchfeatures(fixedFeature,movingFeature,ptCloud,ptCloudTformed);
length(matchingPairs)
```

`ans = 3397`

A score close to zero means that the algorithm is confident about a match and vice-versa. Calculate the mean score for all the matches using the `scores` vector.

```
mean(scores)
```

```
ans = 0.0017
```

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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. For more information about the algorithm, see [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.

Note At least three features must be matched between the feature matrices to consider the spatial relation.

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`

Version History
Introduced in R2020b

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
pcshowMatchedFeatures | extractFPFHFeatures

pcshowMatchedFeatures

Display point clouds with matched feature points

Syntax

```
pcshowMatchedFeatures(ptCloud1,ptCloud2,matchedPtCloud1,matchedPtCloud2)
pcshowMatchedFeatures(segments1,segments2,features1,features2)
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. The plot is color coded by point cloud and each
connected to the corresponding point in the other point cloud by a line.

pcshowMatchedFeatures(segments1,segments2,features1,features2) displays the point
cloud segments, segments1 and segments2, with their corresponding centroids in the "Centroid"
on page 2-0   property of features1 and features2. The plot is color coded and the
corresponding centroids are connected 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

Visualize Matching Features in Point Clouds

This example shows how to visualize matching point cloud features using the
pcshowMatchedFeatures function. The example uses features calculated using
extractFPFHFeatures function.

Load the required files into the workspace.

```
load("features1.mat");
load("features2.mat");
load("ptCloud1.mat");
load("ptCloud2.mat");
```

Match features between two point clouds.

```
indexPairs = pcmatchfeatures(features1,features2,ptCloud1,ptCloud2);
```

Create point clouds of only the points in each point cloud with matching features in the other point
cloud.

```
matchedPts1 = select(ptCloud1,indexPairs(:,1));
matchedPts2 = select(ptCloud2,indexPairs(:,2));
```

Visualize the matches.

```
pcshowMatchedFeatures(ptCloud1,ptCloud2,matchedPts1,matchedPts2, ...
    "Method","montage")
xlim([-40 210])
ylim([-50 50])
title("Matched Points")
```
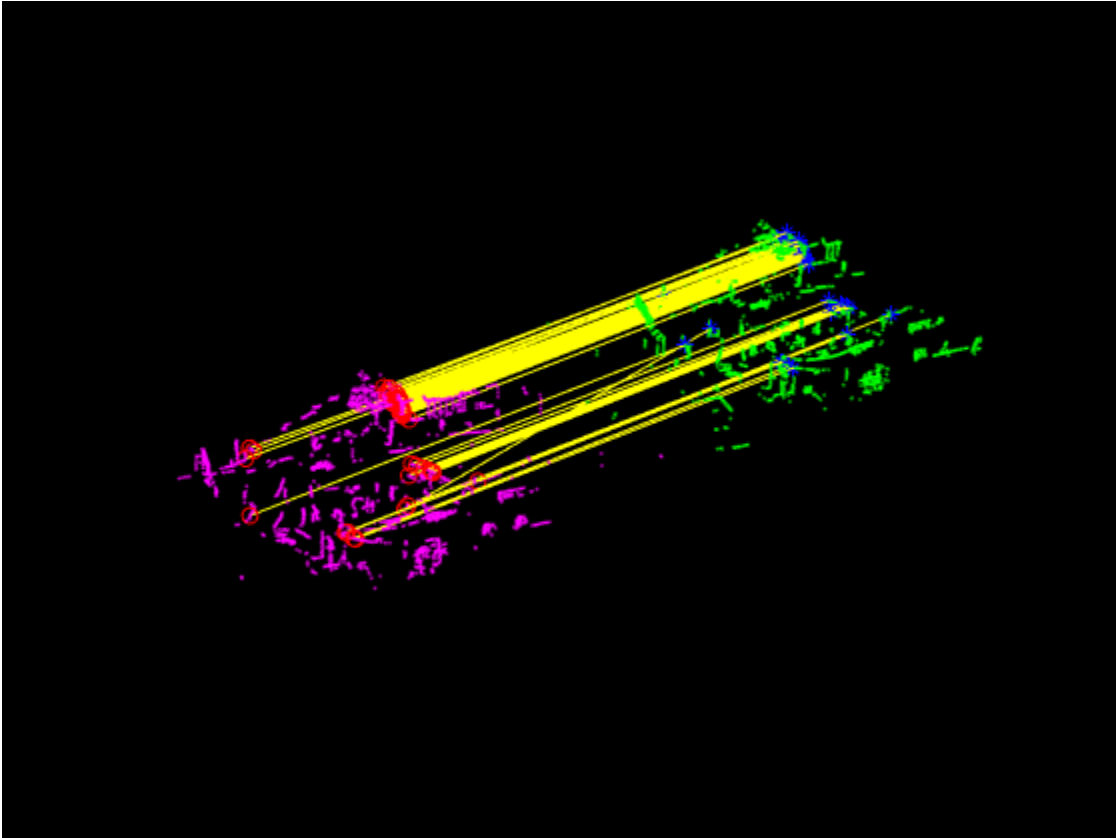

The matched features and point clouds are color coded to improve visualization:

- 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.

Match Eigenvalue-Based Features Between Point Clouds

Create a Velodyne PCAP file reader.

```
veloReader = velodyneFileReader('lidarData_ConstructionRoad.pcap','HDL32E');
```

Read the first and fourth scans from the file.

```
ptCloud1 = readFrame(veloReader,1);
ptCloud2 = readFrame(veloReader,4);
```

Remove the ground plane from the scans.

```
maxDistance = 1; % in meters
referenceVector = [0 0 1];
[~,~,selectIdx] = pcfitplane(ptCloud1,maxDistance,referenceVector);
ptCloud1 = select(ptCloud1,selectIdx,'OutputSize','full');
[~,~,selectIdx] = pcfitplane(ptCloud2,maxDistance,referenceVector);
ptCloud2 = select(ptCloud2,selectIdx,'OutputSize','full');
```

Cluster the point clouds with a minimum of 10 points per cluster.

```
minDistance = 2; % in meters
minPoints = 10;
labels1 = pcsegdist(ptCloud1,minDistance,'NumClusterPoints',minPoints);
labels2 = pcsegdist(ptCloud2,minDistance,'NumClusterPoints',minPoints);
```

Extract eigen-value features and the corresponding segments from each point cloud.

```
[eigFeatures1,segments1] = extractEigenFeatures(ptCloud1,labels1);
[eigFeatures2,segments2] = extractEigenFeatures(ptCloud2,labels2);
```

Create matrices of the features and centroids extracted from each point cloud, for matching.

```
features1 = vertcat(eigFeatures1.Feature);
features2 = vertcat(eigFeatures2.Feature);
centroids1 = vertcat(eigFeatures1.Centroid);
centroids2 = vertcat(eigFeatures2.Centroid);
```
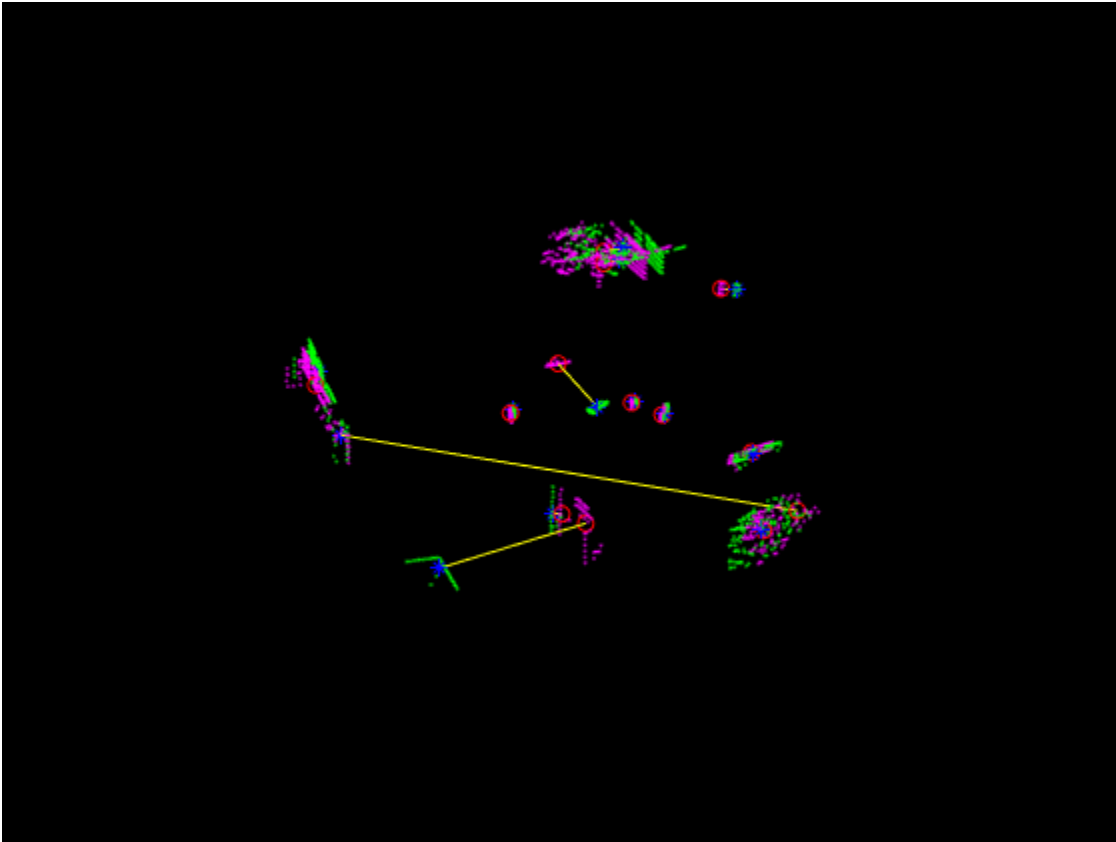
Find putative feature matches.

```
indexPairs = pcmatchfeatures(features1,features2, ...
    pointCloud(centroids1),pointCloud(centroids2));
```

Get the matched segments and features for visualization.

```
matchedSegments1 = segments1(indexPairs(:,1));
matchedSegments2 = segments2(indexPairs(:,2));
matchedFeatures1 = eigFeatures1(indexPairs(:,1));
matchedFeatures2 = eigFeatures2(indexPairs(:,2));
```

Visualize the matches.

```
figure
pcshowMatchedFeatures(matchedSegments1,matchedSegments2,matchedFeatures1,matchedFeatures2)
title('Matched Segments')
```

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.

segments1 — Point cloud segments
M-element vector of `pointCloud` objects

Point cloud segments, specified as a *M*-element vector of `pointCloud` objects.

segments2 — Point cloud segments
M-element vector of `pointCloud` objects

Point cloud segments, specified as a *M*-element vector of `pointCloud` objects.

features1 — Corresponding centroids in first segment features
M-element vector of `eigenFeature` objects

Corresponding centroids in the first segment features, specified as a *M*-element vector of `eigenFeature` objects. The "Centroid" on page 2-0   property of each feature in `features1` is plotted with a red circle by default.

features2 — Corresponding centroids in second segment features
M-element vector of `eigenFeature` objects

Corresponding centroids in the second segment features, specified as a *M*-element vector of `eigenFeature` objects. The "Centroid" on page 2-0   property of each feature in `features2` is plotted with a blue asterisk by default.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History
Introduced in R2020b

See Also

Functions
pcmatchfeatures | extractFPFHFeatures | extractEigenFeatures | pcmapsegmatch

Objects
eigenFeature | pointCloud

Topics
"Lidar Localization Using Segment Matching" on page 2-116
"Build Map and Localize Using Segment Matching"

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History
Introduced in R2020b

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
semanticseg | trainNetwork | evaluateSemanticSegmentation

Objects
focalLossLayer | pixelClassificationLayer | layerGraph | DAGNetwork

Topics
"Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network"

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.
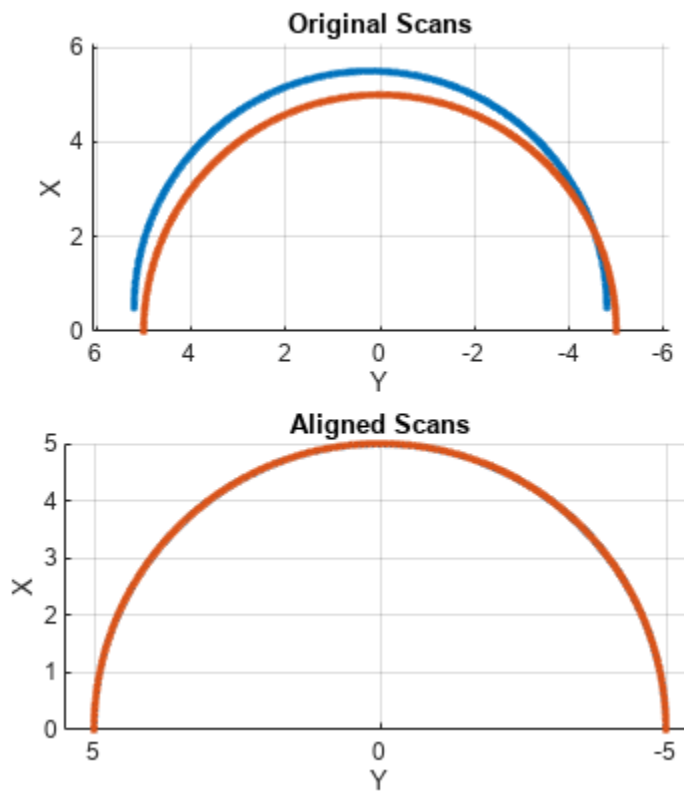
```
pose = matchScans(currScan,refScan);
```

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)
```

3-143

```
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 pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

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.

Version History
Introduced in R2020b

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
`matchScansGrid` | `matchScansLine` | `lidarScan`

Classes
`occupancyMap` | `monteCarloLocalization`

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
```
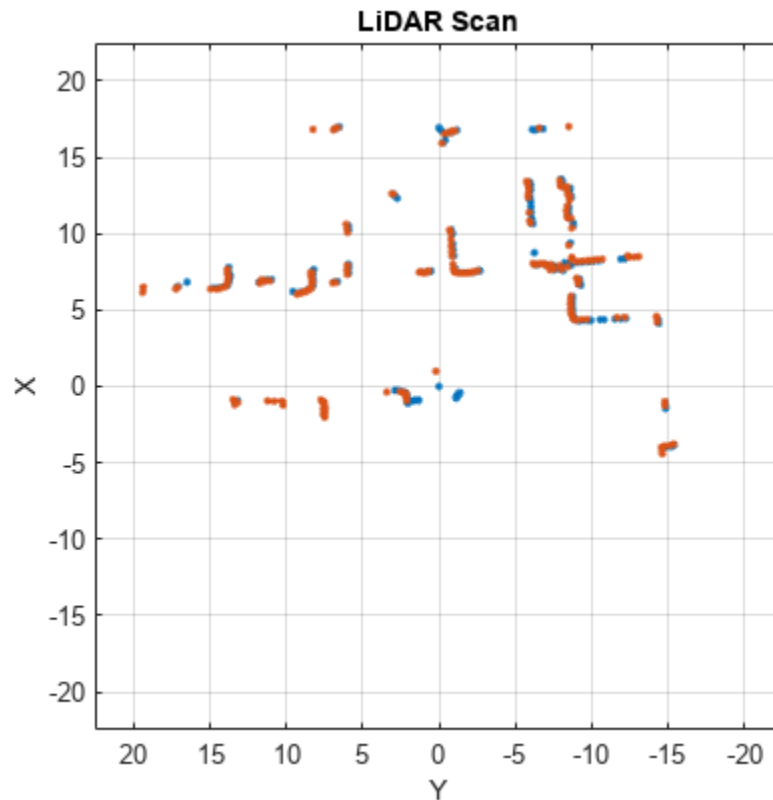
LiDAR Scan
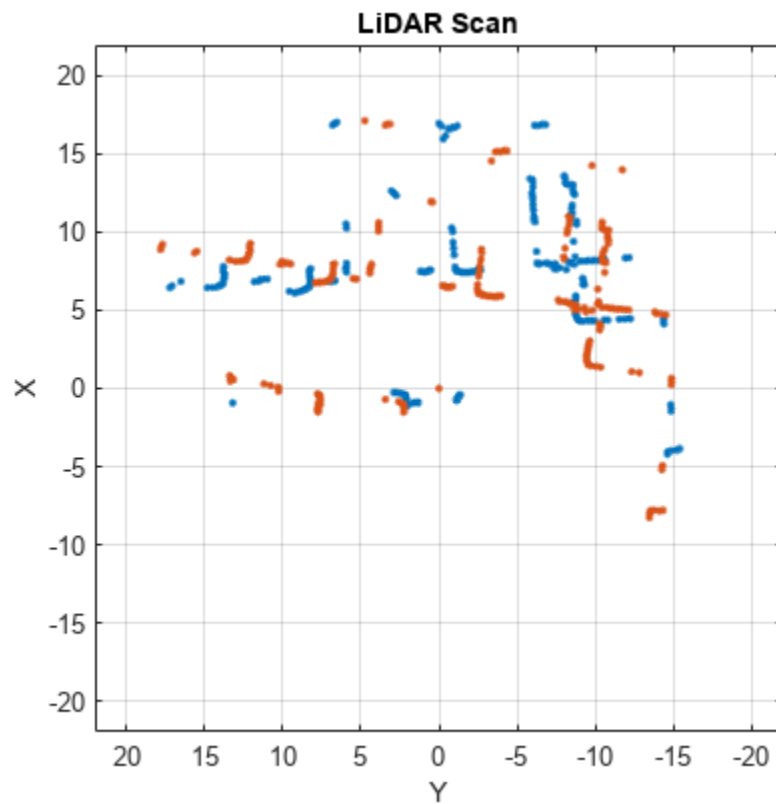
Use `matchScansGrid` to estimate the pose between the two scans.

```
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
```

LiDAR Scan

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 pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

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.

Version History

Introduced in R2020b

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
matchScans | matchScansLine | lidarScan

Classes
lidarSLAM

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
```

LiDAR Scan
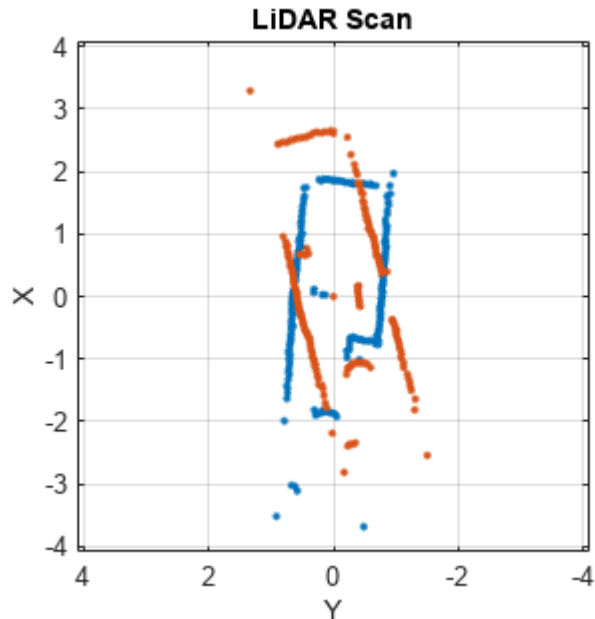
Set parameters for line feature extraction and association. The noise of the lidar data determines the smoothness threshold, which defines when a line break occurs for a specific line feature. Increase this value for more noisy lidar data. The compatibility scale determines when features are considered 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
```
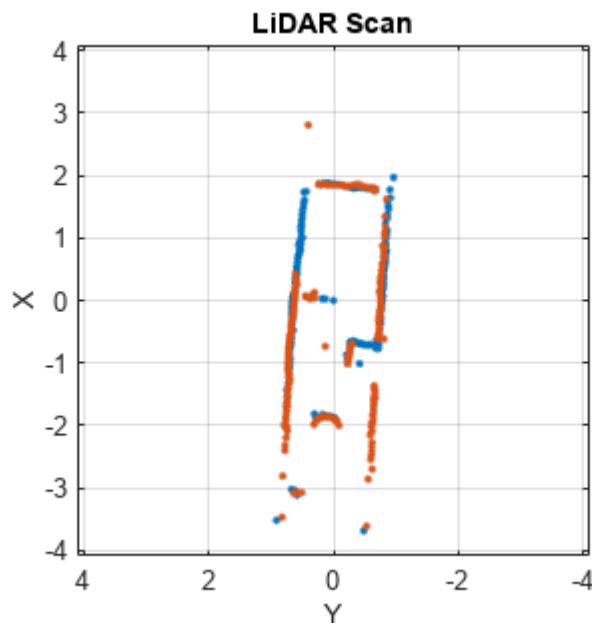

LiDAR Scan

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 ReferenceFeaturesand 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.

Version History

Introduced in R2020b

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`

bboxLidarToCamera

Estimate 2-D bounding box in camera frame using 3-D bounding box in lidar frame

Syntax

```
bboxesCamera = bboxLidarToCamera(bboxesLidar,intrinsics,tform)
bboxesCamera = bboxLidarToCamera(bboxesLidar,intrinsics,tform,L)
[bboxesCamera,boxesUsed] = bboxLidarToCamera( ___ )
[ ___ ] = bboxLidarToCamera( ___ ,'ProjectedCuboid',true)
```

Description

`bboxesCamera = bboxLidarToCamera(bboxesLidar,intrinsics,tform)` estimates 2-D bounding boxes in the camera frame from 3-D bounding boxes in the lidar frame `bboxesLidar`. The function uses the camera intrinsic parameters `intrinsics` and a lidar to camera transformation matrix `tform`.

`bboxesCamera = bboxLidarToCamera(bboxesLidar,intrinsics,tform,L)` further refines the 2-D bounding boxes to the edges of the object inside it using L. L is the corresponding labeled 2-D image of the 2-D bounding boxes, where the objects are labeled distinctively.

`[bboxesCamera,boxesUsed] = bboxLidarToCamera( ___ )` indicates for which of the specified 3-D bounding boxes the function detects a corresponding 2-D bounding box in the camera frame.

`[ ___ ] = bboxLidarToCamera( ___ ,'ProjectedCuboid',true)` returns 3-D projected cuboids instead of 2-D bounding boxes.

Examples

Transfer Bounding Box from Point Cloud to Image

Load ground truth data from a MAT file into the workspace. Extract the image, point cloud, 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 lidar to camera transformation matrix from the ground truth data.

```
tform = gt.camToLidar.invert;
```
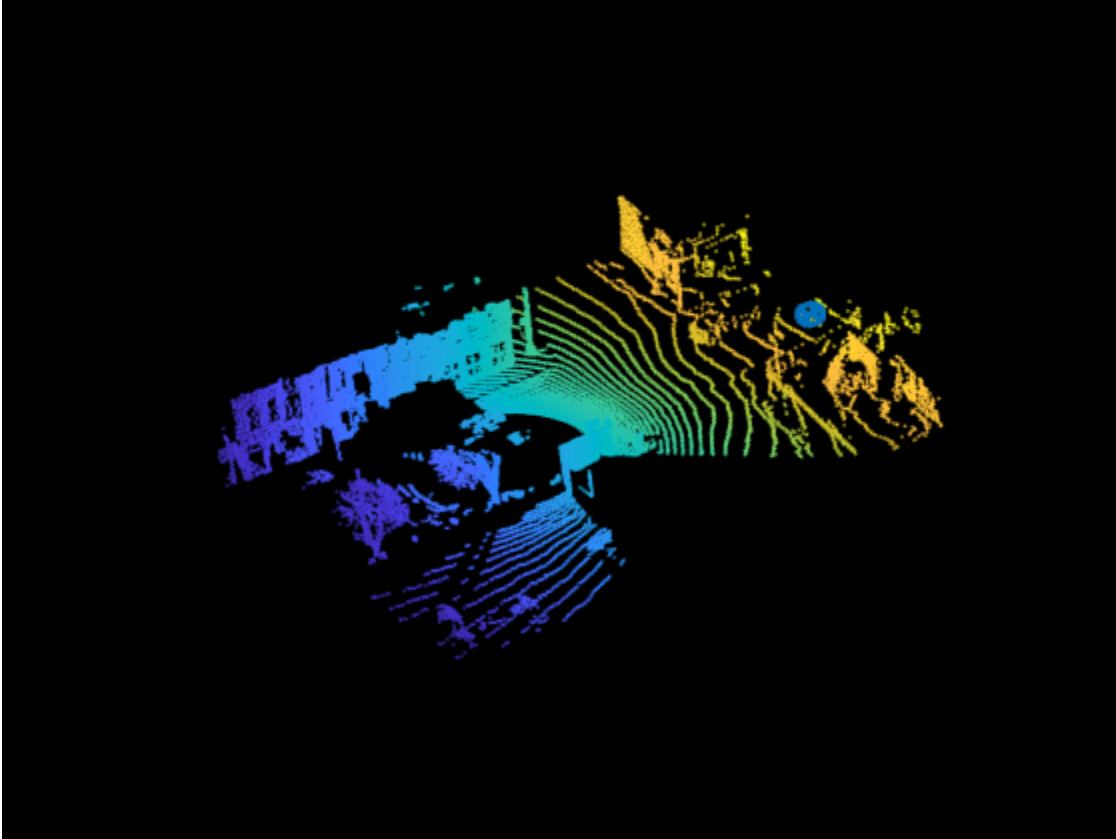
Extract the 3-D bounding box information.

```
bboxLidar = gt.cuboid1;
```

Estimate the 2-D bounding box on the image.

```
bboxesCamera = bboxLidarToCamera(bboxLidar,intrinsics,tform);
```

Display the 3-D bounding box overlaid on the point cloud.

```
pcshow(pc.Location,pc.Location(:,1))
showShape('cuboid',bboxLidar)
```


Display the 2-D bounding box overlaid on the image.

```
J = undistortImage(im,intrinsics);
annotatedImage = insertObjectAnnotation(J,'Rectangle',bboxesCamera,'Vehicle');
imshow(annotatedImage)
```

Project 3-D Bounding Box from Point Cloud to Image

Load ground truth data from a MAT file into the workspace. Extract the image, point cloud, 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 lidar to camera transformation matrix from the ground truth data.

```
tform = gt.camToLidar.invert;
```

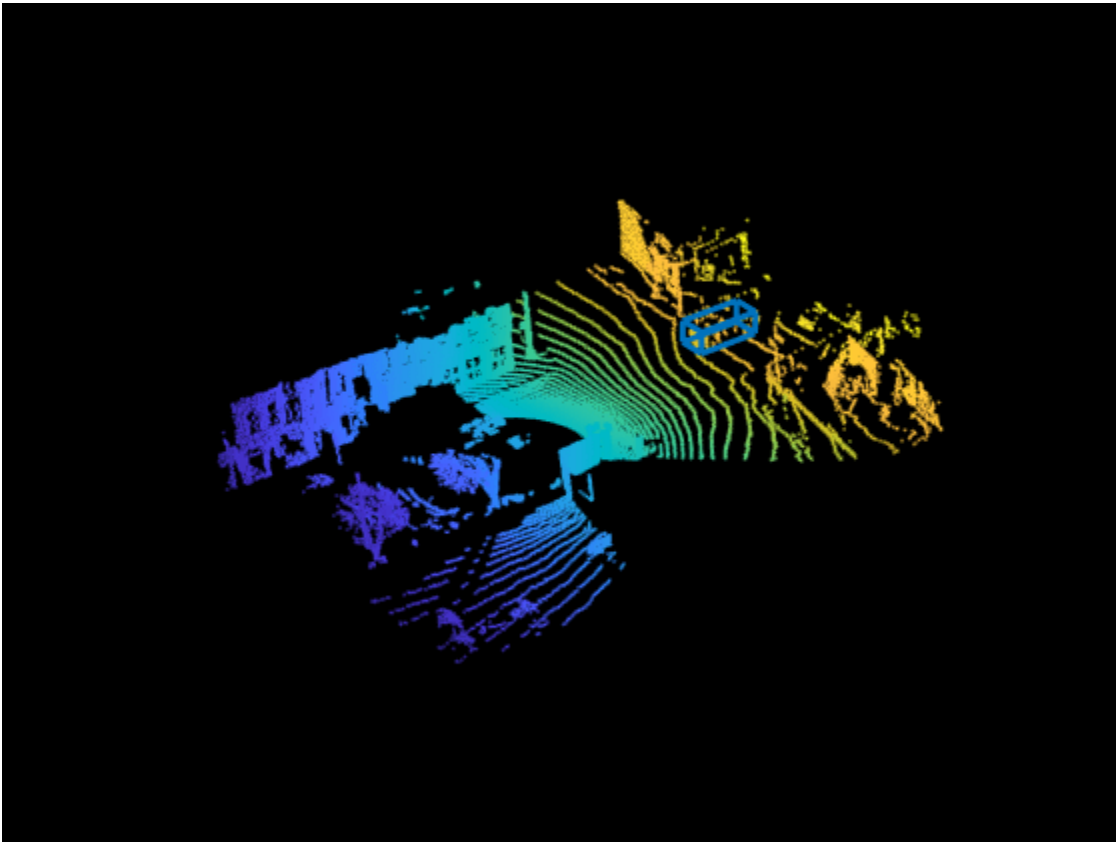Extract the 3-D bounding box information.

```
bboxLidar = gt.cuboid2;
```

Estimate the projected 3-D bounding box on the image.

```
bboxesCamera = bboxLidarToCamera(bboxLidar,intrinsics,tform,...
        'ProjectedCuboid',true);
```

Display the 3-D bounding box overlaid on the point cloud.

```
figure
pcshow(pc.Location,pc.Location(:,1))
showShape('cuboid',bboxLidar)
```


Display the 3-D projected bounding box overlaid on the image.

```
J = undistortImage(im,intrinsics);
h = imshow(J);
pcH = vision.roi.ProjectedCuboid;
pcH.Parent = h.Parent;
pcH.Position = bboxesCamera;
```

Input Arguments

bboxesLidar — 3-D bounding boxes in lidar frame
cuboidModel object | *N*-by-9 matrix of real values

3-D bounding boxes in the lidar frame, specified as a `cuboidModel` object or an *N*-by-9 matrix of real values. *N* is the number of 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 you look in the forward direction of their corresponding axes.

This figure shows how these values determine the position of a cuboid.
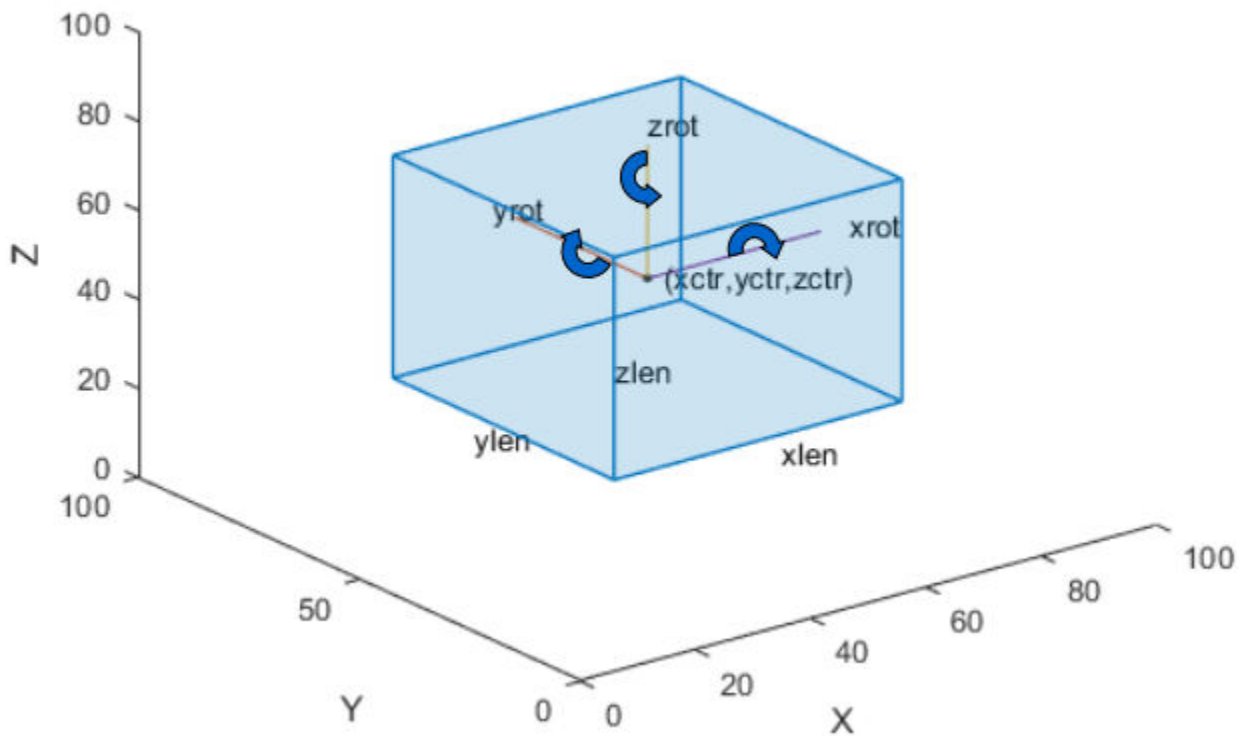
Note The function assumes that the point cloud data that corresponds to the 3-D bounding boxes and the image data are time synchronized.

Data Types: `single` | `double`

`intrinsics` — Camera intrinsic parameters
`cameraIntrinsics` object

Camera intrinsic parameters, specified as a `cameraIntrinsics` object.

`tform` — Camera to lidar rigid transformation
`rigidtform3d` object

Camera to lidar rigid transformation, specified as a `rigidtform3d` object.

L — Labeled 2-D image
matrix of real values

Labeled 2-D image, specified as a matrix of real values. The matrix size is the same as the `ImageSize` property of `intrinsics`.

Note Labeled images are assumed to be undistorted.

Data Types: `single` | `double` | `int8` | `int16` | `uint8` | `uint16`

Output Arguments

bboxesCamera — 2-D bounding boxes in camera frame
M-by-4 matrix of real values | *M*-by-8 matrix of real values

2-D bounding boxes in the camera frame, returned as an *M*-by-4 matrix of real values. *M* is the number of detected bounding boxes. 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.

If `'ProjectedCuboid'` is set to `true`, the 2-D bounding boxes are returned as an *M*-by-8 matrix of real values. The bounding boxes have a cuboid shape and enclose the object. Each row of the matrix contains the size and location of the cuboid bounding box in the form [*frontFace* *backFace*]. Both the faces are represented as 2-D bounding boxes.

Data Types: `single` | `double`

boxesUsed — Bounding box detection flag
N-element row vector of logicals

Bounding box detection flag, returned as an *N*-element row vector of logicals. *2* is the number of input 3-D bounding boxes. If the function detects a corresponding 2-D bounding box in the camera frame, then it returns a value of `true` for that input 3-D bounding box. If the function does not detect a corresponding 2-D bounding box, then it returns a value of `false`.

Data Types: `logical`

Version History
Introduced in R2021a

Supports `rigidtform3d` objects

You can now specify `tform` as a `rigidtform3d` object, which uses the premultiply convention. Although you can still specify `tform` as a `rigid3d` object, this object is not recommended because it uses the postmultiply convention. For more information, see "Migrate Geometric Transformations to Premultiply Convention".

See Also

Functions
`bboxCameraToLidar` | `projectLidarPointsOnImage` | `fuseCameraToLidar`

segmentGroundSMRF

Segment ground from lidar data using a SMRF algorithm

Syntax

```
groundPtsIdx = segmentGroundSMRF(ptCloud)
groundPtsIdx = segmentGroundSMRF(ptCloud,gridResolution)
[groundPtsIdx,nonGroundPtCloud,groundPtCloud] = segmentGroundSMRF( ___ )
[ ___ ] = segmentGroundSMRF( ___ ,Name=Value)
```

Description

`groundPtsIdx = segmentGroundSMRF(ptCloud)` segments the input point cloud `ptCloud` into ground and nonground points using a simple morphological filter (SMRF) algorithm. For more information on the SMRF algorithm, see "Simple Morphological Filter" on page 3-171.

`groundPtsIdx = segmentGroundSMRF(ptCloud,gridResolution)` additionally specifies the dimension of the grid elements.

`[groundPtsIdx,nonGroundPtCloud,groundPtCloud] = segmentGroundSMRF( ___ )` returns the ground points and nonground points as individual `pointCloud` objects, in addition to the ground point indices, using any of the input argument combinations from previous syntaxes.

`[ ___ ] = segmentGroundSMRF( ___ ,Name=Value)` specifies options using one or more name-value arguments. For example, `ElevationThreshold=0.4` sets the elevation threshold for identifying nonground points to `0.4`.

Examples

Segment Ground in Aerial Lidar Data

Segment the ground in an unorganized aerial point cloud.

Create a `lasFileReader` object to access the data of `aerialLidarData2.las`.

```
fileName = fullfile(toolboxdir("lidar"),"lidardata","las", ...
          "aerialLidarData2.las");
lasReader = lasFileReader(fileName);
```

Read the point cloud data from the LAS file using the `readPointCloud` function.
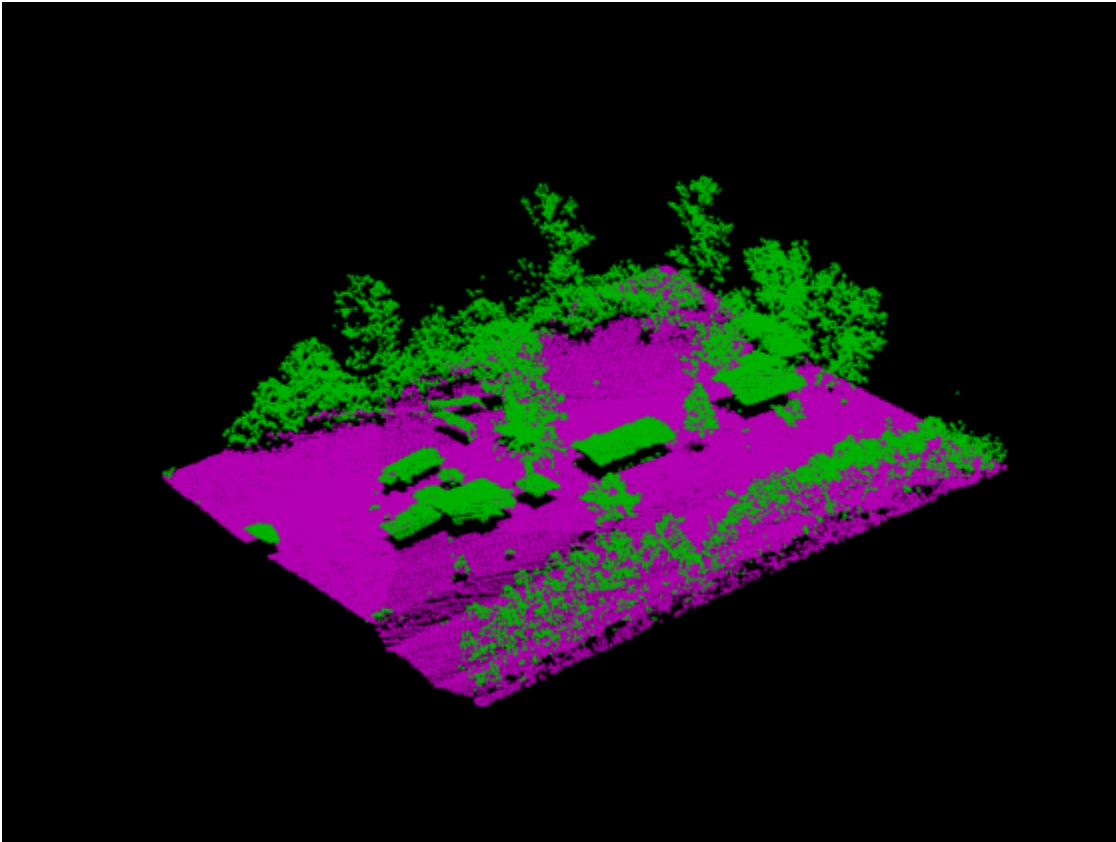
```
ptCloud = readPointCloud(lasReader);
```

Segment the ground data from the point cloud.

```
[groundPtsIdx,nonGroundPtCloud,groundPtCloud] = segmentGroundSMRF(ptCloud);
```

Visualize the ground and nonground points.

```
figure
pcshowpair(groundPtCloud,nonGroundPtCloud)
```

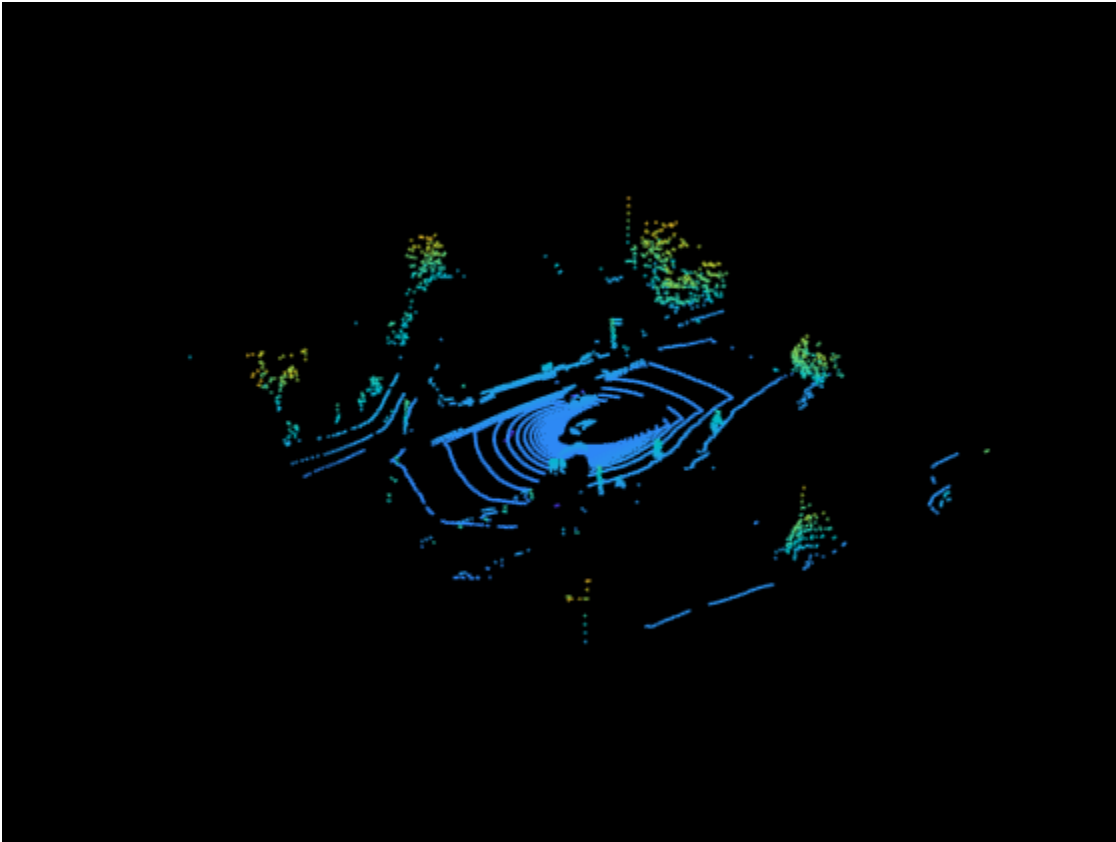**Segment and Remove Ground from Point Cloud Data**

Segment and remove the ground from an organized point cloud.

Load the point cloud data into the workspace. This point cloud was captured in a highway scenario.

```
ld = load("drivingLidarPoints.mat");
```

Display the input point cloud.

```
pcshow(ld.ptCloud)
xlim([-40 40])
ylim([-50 50])
```
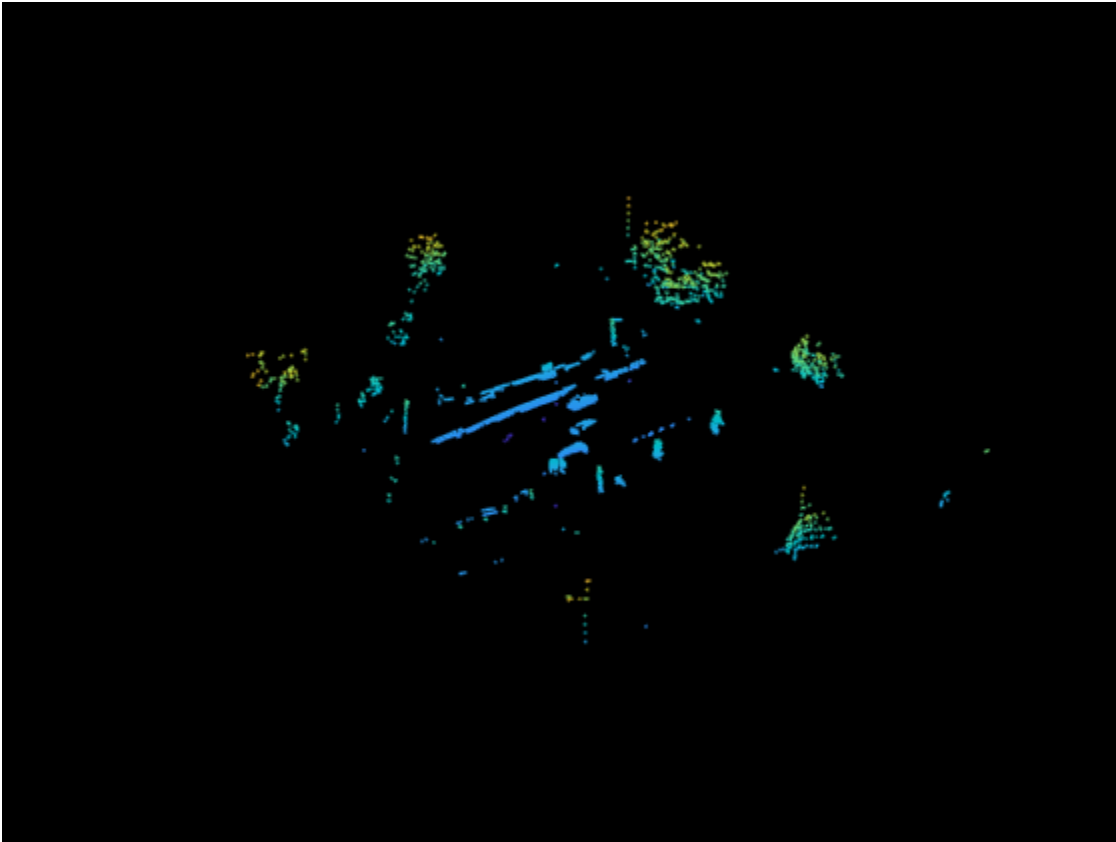
Segment the ground data from the point cloud.

```
[~,nonGroundPtCloud,groundPtCloud] = segmentGroundSMRF( ...
   ld.ptCloud,MaxWindowRadius=5,ElevationThreshold=0.1,ElevationScale=0.25);
```

Visualize the nonground points.

```
figure
pcshow(nonGroundPtCloud)
xlim([-40 40])
ylim([-50 50])
```

Input Arguments

ptCloud — Point cloud data
pointCloud object

Point cloud data, specified as a `pointCloud` object.

gridResolution — Dimension of each grid element
1 (default) | positive scalar

Dimension of each grid element, specified as a positive scalar. The function samples the input point cloud into grids along the *xy*-direction using the grid size to create a minimum surface map. Decreasing the value of grid resolution may return nonground points as ground.

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

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `segmentGroundSMRF(ptCloud,ElevationThreshold=0.4)` sets the elevation threshold for identifying nonground points to `0.4`.

MaxWindowRadius — Maximum radius for structuring element
18 (default) | positive integer

Maximum radius for the disc-shaped structuring element in the morphological opening operation, specified as a positive integer. You can segment large buildings as ground by specifying a smaller radius. Increasing this value can increase the computation time of the function.

Note The default value works effectively for aerial lidar data. For better performance on terrestrial data, set `MaxWindowRadius` to a smaller value, such as `8`.

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

SlopeThreshold — Slope threshold for identifying grid elements
0.15 (default) | nonnegative scalar

Slope threshold for identifying grid elements as ground or nonground in a minimum elevation surface map, specified as a nonnegative scalar. The function classifies a grid element as ground if its slope is less than `SlopeThreshold`. Increase this value to classify steeper slopes as ground.

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

ElevationThreshold — Elevation threshold for identifying points
0.5 (default) | nonnegative scalar

Elevation threshold for identifying points as ground or nonground in the estimated elevation model, specified as a nonnegative scalar. The function classifies a point as ground if the elevation difference between the point and estimated ground surface is less than `ElevationThreshold`. To include more points from bumpy ground, increase this value.

Note The default value works effectively for aerial lidar data. For best results on terrestrial data, set `ElevationThreshold` to a smaller value, such as `0.1`.

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

ElevationScale — Elevation threshold scaling factor
1.25 (default) | nonnegative scalar

Elevation threshold scaling factor, specified as a nonnegative scalar. You can identify ground points on steep slopes by increasing this value.

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

Output Arguments

groundPtsIdx — Binary map of segmented point cloud
logical matrix | logical vector

Binary map of the segmented point cloud, returned as a logical matrix or a logical vector.

- For an organized point cloud of the form *M*-by-*N*-by-3, `groundPtsIdx` is returned as an *M*-by-*N* logical matrix.

- For an unorganized point cloud of the form *M*-by-3, `groundPtsIdx` is returned as an *M* element logical vector.

Elements of this output that correspond to ground points in the point cloud are `true` and nonground points are `false`.

nonGroundPtCloud — Point cloud of nonground points
pointCloud object

Point cloud of nonground points, returned as a `pointCloud` object.

groundPtCloud — Point cloud of ground points
pointCloud object

Point cloud of ground points, returned as a `pointCloud` object.

Algorithms

A simple morphological filter (SMRF) algorithm [1] segments point cloud data into ground and nonground points. The algorithm consists of three stages:

1 Create a minimum elevation surface map from the point cloud data.
2 Segment the surface map into ground and nonground grid elements.
3 Segment the original point cloud data.

Create Minimum Elevation Surface Map

1 Divide the point cloud data into grids along the *xy*- plane (bird's-eye view). Specify the grid size using `gridResolution`.
2 Find the lowest elevation (Z_{min}) value for each grid element (pixel).
3 Combine all the Z_{min} values into a 2-D matrix (raster image) to create a minimum elevation surface map.

Segment Surface Map

1 Apply a morphological opening operation on the minimum surface map. This applies an erosion filter followed by a dilation filter. For more information on morphological opening, see "Types of Morphological Operations".
2 The shape of the structuring element and its window radius define the search neighbourhood for the morphological opening. Use a disc-shaped structuring element, and start with a window radius of 1 pixel. For more information, see "Structuring Elements".
3 Calculate the slope between the minimum surface and opened surface maps at each grid element. If the difference is greater than the elevation threshold, classify the pixel as nonground.
4 Execute steps 1 through 3 iteratively. Increase the window radius by 1 pixel in each iteration until it reaches the maximum radius specified by `MaxWindowRadius`.
5 The end result of the iteration process is a binary mask in which each pixel of the point cloud is classified as either ground or nonground.

Segment Original Point Cloud

1 Apply the binary mask to the original minimum surface map to remove nonground grids.

3-171

2   Fill the unfilled grids using image interpolation techniques to create an estimated elevation model.

3   Calculate the elevation difference between each point in the original point cloud and the corresponding point in the estimated elevation model. If the difference is greater than `ElevationThreshold`, classify the pixel as nonground.

4   Multiply the slope of the elevation model at the each point by `ElevationScale`, and add the result to the `ElevationThreshold` value to identify ground points on steep slopes.

Version History
Introduced in R2021a

References

[1] Pingel, Thomas J., Keith C. Clarke, and William A. McBride. "An Improved Simple Morphological Filter for the Terrain Classification of Airborne LIDAR Data." *ISPRS Journal of Photogrammetry and Remote Sensing* 77 (March 2013): 21–30.https://doi.org/10.1016/j.isprsjprs.2012.12.002.

Extended Capabilities

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

This function has a real-time performance limitation due to limitations in the `imopen`, `regionfill` functions.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Functions
pcsegdist | segmentLidarData | segmentGroundFromLidarData

Objects
lasFileReader

transformScan

Transform laser scan based on relative pose

Syntax

```
transScan = transformScan(scan,relPose)
```

```
[transRanges,transAngles] = transformScan(ranges,angles,relPose)
```

Description

`transScan = transformScan(scan,relPose)` transforms the laser scan specified in `scan` by using the specified relative pose, `relPose`.

`[transRanges,transAngles] = transformScan(ranges,angles,relPose)` transforms the laser scan specified in `ranges` and `angles` by using the specified relative pose, `relPose`.

Examples

Transform Laser Scans

Create a `lidarScan` object. Specify the ranges and angles as vectors.

```
refRanges = 5*ones(1,300);
refAngles = linspace(-pi/2,pi/2,300);
refScan = lidarScan(refRanges,refAngles);
```

Translate the laser scan by an `[x y]` offset of `(0.5,0.2)`.

```
transformedScan = transformScan(refScan,[0.5 0.2 0]);
```

Rotate the laser scan by 20 degrees.

```
rotateScan = transformScan(refScan,[0,0,deg2rad(20)]);
```

Input Arguments

scan — Lidar scan readings
lidarScan object

Lidar scan readings, specified as a `lidarScan` object.

ranges — Range values from scan data
vector

Range values from scan data, specified as a vector in meters. These range values are distances from a sensor at specified `angles`. The vector must be the same length as the corresponding `angles` vector.

angles — Angle values from scan data
vector

Angle values from scan data, specified as a vector in radians. These angle values are the specific angles of the specified `ranges`. The vector must be the same length as the corresponding `ranges` vector.

relPose — Relative pose of current scan
[x y theta]

Relative pose of current scan, specified as [x y theta], where [x y] is the translation in meters and `theta` is the rotation in radians.

Output Arguments

transScan — Transformed lidar scan readings
`lidarScan` object

Transformed lidar scan readings, specified as a `lidarScan` object.

transRanges — Range values of transformed scan
vector

Range values of transformed scan, returned as a vector in meters. These range values are distances from a sensor at specified `transAngles`. The vector is the same length as the corresponding `transAngles` vector.

transAngles — Angle values from scan data
vector

Angle values of transformed scan, returned as a vector in radians. These angle values are the specific angles of the specified `transRanges`. The vector is the same length as the corresponding `ranges` vector.

Version History
Introduced in R2021a

Extended Capabilities

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

See Also
`matchScans` | `lidarScan` | `matchScansGrid`

Topics
"Build Map from 2-D Lidar Scans Using SLAM"

pcorganize

Convert 3-D point cloud into organized point cloud

Syntax

```
ptCloudOut = pcorganize(ptCloudIn,params)
```

Description

`ptCloudOut = pcorganize(ptCloudIn,params)` converts a 3-D point cloud, `ptCloudIn`, into an organized point cloud, `ptCloutOut`, using the sensor parameters, `params`.

Examples

Convert HDL-64E Unorganized Point Cloud into Organized Point Cloud

Load point cloud data into the workspace.

```
fileName = fullfile(toolboxdir("lidar"),"lidardata","lcc","HDL64", ...
                    "pointCloud","0001.pcd");
ptCloudUnorg = pcread(fileName);
```

Specify the horizontal resolution of the lidar sensor.

```
horizontalResolution = 1024;
```

Create a `lidarParameters` object that represents an HDL64E sensor with the specified `horizontalResolution`.

```
params = lidarParameters('HDL64E',horizontalResolution);
```

Convert the unorganized point cloud into an organized point cloud.

```
ptCloudOrg = pcorganize(ptCloudUnorg,params);
```

Display the dimensions of the input point cloud.

```
size(ptCloudUnorg.Location)
```

ans = *1×2*

```
    37879            3
```

Display the size of the converted point cloud. `pointCloud` objects store organized point clouds as *M*-by-*N*-by-3 arrays, whereas they store unorganized point clouds as *M*-by-3 matrices

```
size(ptCloudOrg.Location)
```

ans = *1×3*

```
        64        1024        3
```

Input Arguments

ptCloudIn — Input point cloud data
pointCloud object

Input point cloud data, specified as a `pointCloud` object.

params — Lidar sensor parameters
lidarParameters object

Lidar sensor parameters, specified as a `lidarParameters` object. For more information about lidar sensor parameters, see "Lidar Sensor Parameters".

Output Arguments

ptCloudOut — Organized point cloud data
pointCloud object

Organized point cloud data, returned as a `pointCloud` object.

Version History
Introduced in R2021b

See Also
`lidarParameters` | `pointCloud`

Topics
"Unorganized to Organized Conversion of Point Clouds Using Spherical Projection"
"Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network"
"What are Organized and Unorganized Point Clouds?"

pc2dem

Create digital elevation model (DEM) of point cloud data

Syntax

```
elevModel = pc2dem(ptCloudIn)
elevModel = pc2dem(ptCloudIn,gridResolution)
[elevModel,xlimits,ylimits] = pc2dem( ___ )
[ ___ ] = pc2dem( ___ ,Name,Value)
```

Description

`elevModel = pc2dem(ptCloudIn)` creates and returns a digital elevation model (DEM) `elevModel` for the input point cloud. The output matrix contains generalized elevation information of the input point cloud. For more information, see "Algorithms" on page 3-183.

`elevModel = pc2dem(ptCloudIn,gridResolution)` additionally specifies the dimensions of the grid element.

`[elevModel,xlimits,ylimits] = pc2dem( ___ )` additionally returns the *x*- and *y*-limits of the DEM, using any combination of input arguments from previous syntaxes.

`[ ___ ] = pc2dem( ___ ,Name,Value)` specifies options using one or more name-value arguments. For example, `'CornerFillMethod','min'` specifies for the function to compute the generalized elevation values for the grid corners in the DEM as the minimum elevation in the default search radius of each grid corner.

Examples

Create Digital Terrain Model from Point Cloud

Create a `lasFileReader` object to read point cloud data stored in `aerialLidarData.laz` file.
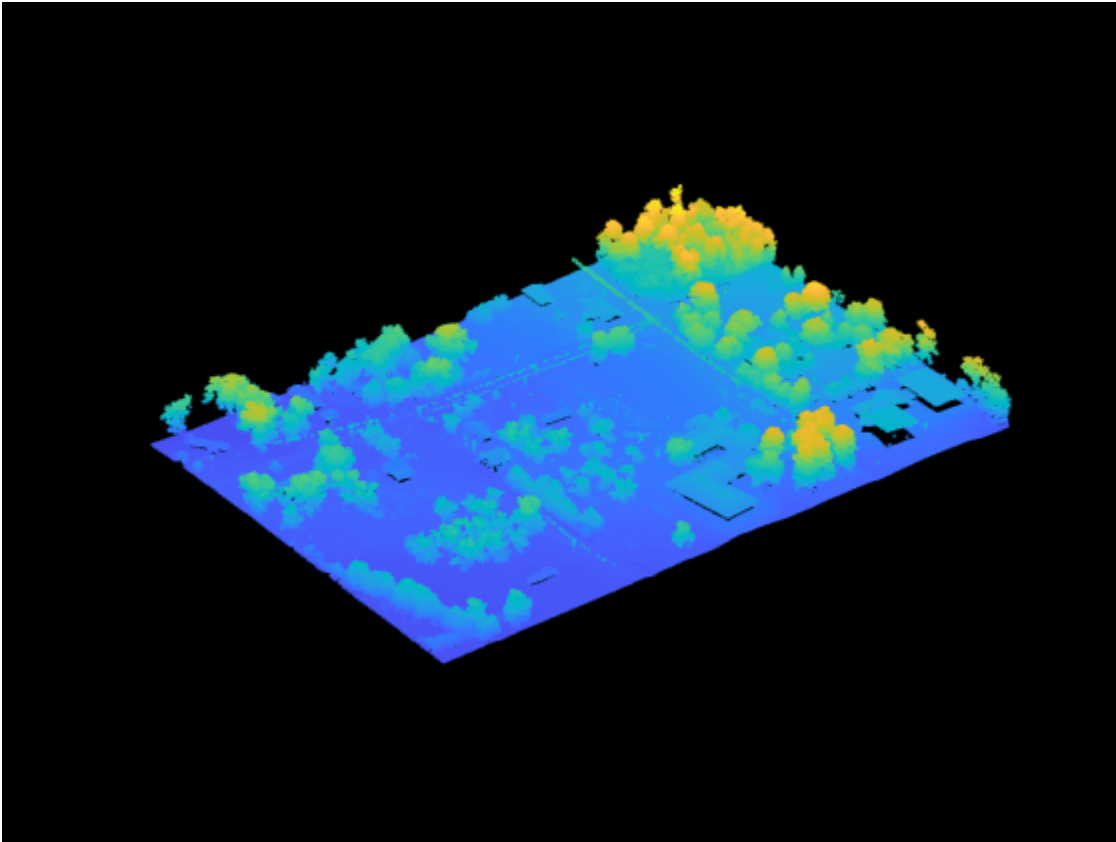
```
fileName = fullfile(toolboxdir("lidar"),"lidardata","las", ...
            "aerialLidarData.laz");
lasReader = lasFileReader(fileName);
```

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

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

Visualize the point cloud data.

```
figure
pcshow(ptCloud.Location)
title("Point Cloud")
```

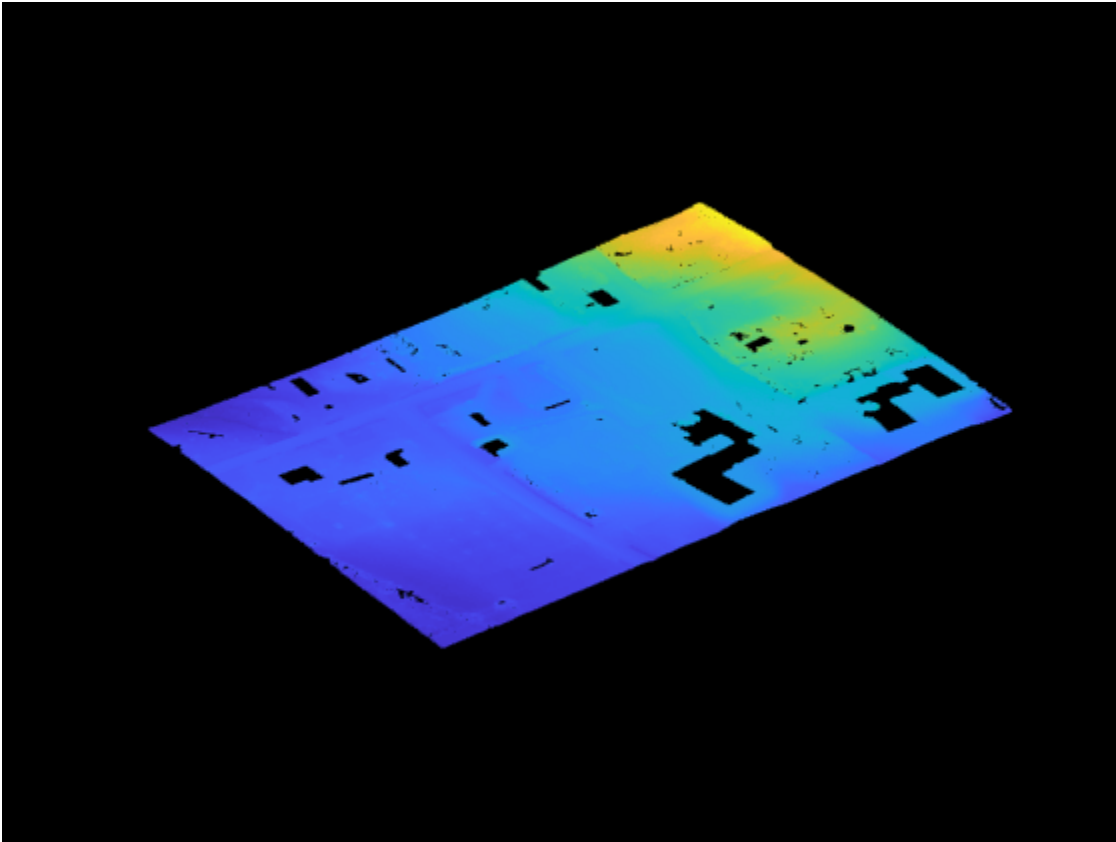Segment the ground points from the point cloud data.

```
groundPtsIdx = segmentGroundSMRF(ptCloud);
```

Extract the ground points.

```
ptCloudWithGround = select(ptCloud,groundPtsIdx);
```

Visualize the ground points.

```
figure
pcshow(ptCloudWithGround.Location)
title("Ground Points")
```
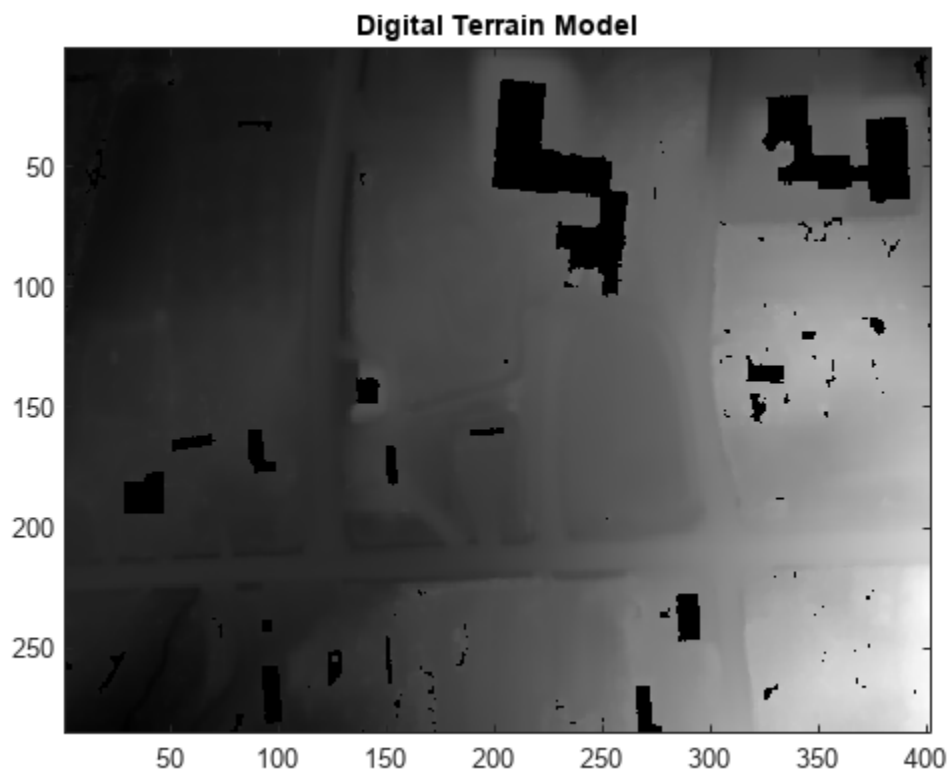
Create a digital terrain model (DTM) from the segmented ground points.

```
terrainModel = pc2dem(ptCloudWithGround);
```

Visualize the DTM.

```
figure
imagesc(terrainModel)
colormap(gray)
title("Digital Terrain Model")
```

Digital Terrain Model

Create Digital Surface Model from Point Cloud

Create a `lasFileReader` object to read aerial point cloud data from `"aerialLidarData.laz"`.

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

Read the point cloud data of the first return of the lidar sensor from the LAS file using the `readPointCloud` function.

```
ptCloud = readPointCloud(lasReader,"LaserReturn",1);
```

Create a digital surface model (DSM) of the point cloud with a grid element resolution of 1.1 meters.

```
gridRes = 1.1;
surfaceModel = pc2dem(ptCloud,gridRes,"CornerFillMethod","max");
```

Define the location of the illumination source.

```
azimuthAng = 135;
zenithAng = 45;
```

Compute the directional gradients of the DSM using the `imgradientxy` function.

```
[gx,gy] = imgradientxy(surfaceModel,"sobel");
```

Normalize the gradients using the grid element resolution.

```
gx = gx/(8*gridRes);
gy = gy/(8*gridRes);
```

Compute the slope and aspect of the DSM.

```
slopeAngle = atand(sqrt(gx.^2 + gy.^2));
aspectAngle = atan2d(gy,-gx);
aspectAngle(aspectAngle < 0) = aspectAngle(aspectAngle < 0) + 360;
```

Calculate the hillshade using the algorithm from Esri®. A hillshade is a 3-D grayscale representation of a surface, with the relative position of the illumination source taken into account when shading the image.

```
h = 255.0*((cosd(zenithAng).*cosd(slopeAngle)) ...
    + (sind(zenithAng).*sind(slopeAngle).*cosd(azimuthAng - aspectAngle)));
h(h < 0) = 0;
```

Visualize the hillshade of the DSM.

```
figure
imagesc(h)
colormap(gray)
```
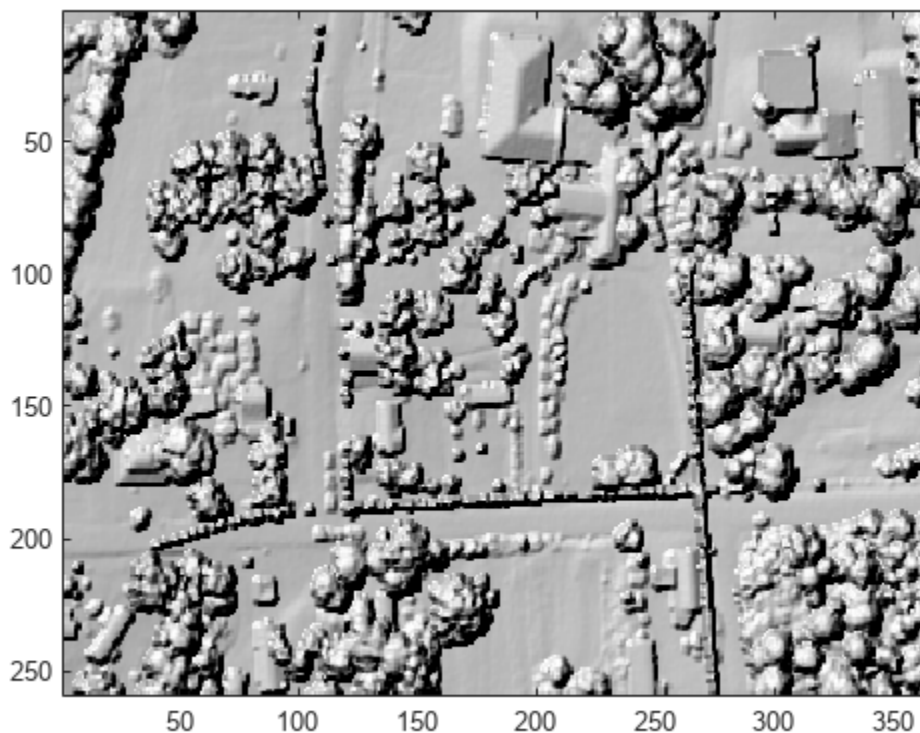
Input Arguments

ptCloudIn — Input point cloud
pointCloud object

Input point cloud, specified as a `pointCloud` object.

gridResolution — Resolution of grid element
[1 1] (default) | two-element vector | scalar

Resolution of the grid element along the *xy*-axes, specified as a two-element vector of the form [*x y*], or as a scalar for square elements. Values for this argument must be positive real numbers.

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

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'CornerFillMethod','min'` selects the type of method to compute the generalized elevation values for each grid corner in the DEM.

CornerFillMethod — Method for grid corner compute generalized elevation value calculation
`'mean'` (default) | character vector | string scalar

Method for grid corner the generalized elevation value calculation, specified as a character vector or a string scalar. The list of supported methods and how they must be specified is as follows:

- `'min'` — Minimum elevation of all the points in the search radius
- `'max'` — Maximum elevation of all the points in the search radius
- `'mean'` — Mean elevation of all the points in the search radius
- `'idw'` — Inverse distance weighted (IDW) mean elevation of all the points in the search radius

Data Types: `char` | `string`

SearchRadius — Radius of search region around each grid corner
`sqrt(2)*gridResolution(1)` (default) | positive scalar

Radius of search region around each grid corner, specified as a positive scalar.

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

FilterSize — Filter size for IDW interpolation
1 (default) | positive odd integer

Filter size for IDW interpolation to fill unfilled grid corners, specified as a positive odd integer.

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

Output Arguments

`elevModel` — Digital elevation model
M-by-*N* matrix of real values

Digital elevation model, returned as an *M*-by-*N* matrix of real values. The values of *M* and *N* are computed based on point cloud limits along the *xy*-axes and the `gridResolution`.

`xlimits` — *x*-axis limits of elevation model
two-element real-valued vector

x-axis limits of the elevation model, returned as a two-element real-valued vector.

`ylimits` — *y*-axis limits of elevation model
two-element real-valued vector

y-axis limits of the elevation model, returned as a two-element real-valued vector.

Algorithms

The function uses a local binning algorithm to create a digital elevation model (DEM) of the point cloud data. The algorithm assumes that the elevation of points is along the *z*-axis.

Local Binning Algorithm:

- Divide the point cloud into a grid along the *xy*-dimensions (bird's eye view). Specify the grid dimensions using the `gridResolution` argument.

- Utilize the elevation information of all points within a circular region around each grid corner to compute generalized grid values. You can specify the search radius and computation method using the `'SearchRadius'` and `'CornerFillMethod'` name-value arguments, respectively.

- If there are no points within the circular region, the algorithm does not compute a value and those grid corners remain unfilled. The function represents them as `NaN`. The algorithm uses inverse distance weighted (IDW) interpolation to fill the unfilled grid corners. To specify the filter size for the IDW interpolation method, use the `'FilterSize'` name-value argument.

Version History
Introduced in R2021b

Extended Capabilities

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

See Also
`pointCloud` | `velodyneFileReader` | `lasFileReader` | `segmentGroundSMRF` | `pcread`

trainPointPillarsObjectDetector

Train PointPillars object detector

Syntax

```
detector = trainPointPillarsObjectDetector(trainingData,detector,options)
[detector,info] = trainPointPillarsObjectDetector(trainingData,detector,
options)
___ = trainPointPillarsObjectDetector( ___ ,Name=Value)

detector = trainPointPillarsObjectDetector(trainingData,checkpoint,options)
```

Description

Train a Detector

`detector = trainPointPillarsObjectDetector(trainingData,detector,options)`
trains a PointPillars object detector using deep learning and the specified training options for the
detection network.

`[detector,info] = trainPointPillarsObjectDetector(trainingData,detector,
options)` returns information on the training progress of the object detector, such as the training
accuracy for each iteration.

`___ = trainPointPillarsObjectDetector( ___ ,Name=Value)` uses additional options
specified by one or more name-value arguments and any of the previous inputs.

Resume Training a Detector

`detector = trainPointPillarsObjectDetector(trainingData,checkpoint,options)`
resumes training from the saved detector checkpoint.

You can use this syntax to:

*   Add more training data and continue the training.
*   Improve training accuracy by increasing the maximum number of iterations.

Input Arguments

`trainingData` — Input training data
valid datastore object | table

Training data, specified as a valid datastore object or table.

*   If you use a datastore object, your data must be set up such that using the `read` function on the
    datastore object returns a cell array or table with three columns. Each row corresponds to a point
    cloud, and the columns must follow this format.

    *   First column — Organized or unorganized point cloud data, specified as a `pointCloud` object.

- Second column — Bounding boxes, specified as a cell array containing an *M*-by-9 matrix. Each row of the matrix is of the form [*x y z length width height roll pitch yaw*], representing the location and dimension of a bounding box. *M* is the number of bounding boxes.
- Third column — Labels, specified as a cell array containing an *M*-by-1 categorical vector with object class names. All categorical data returned by the datastore must use the same pool of categories.

You can use the `combine` function to combine two or more datastores. For more information on creating datastore objects, see the `datastore` function.

- If you use a table, the table must have two or more columns. The first column must contain point cloud file names. The point cloud files can be in any format supported by `pcread` function. Each of the remaining columns represent a single object class such as *Car*, or *Truck* containing cell vectors. Each cell contains an *M*-by-9 matrix, *M* is the number of bounding boxes. The columns of the each matrix are of the form [*x y z length width height roll pitch yaw*], specifying the location and dimensions of the bounding box in the corresponding point cloud.

You can generate the input training data from labeled ground truth samples by using the `lidarObjectDetectorTrainingData` function.

detector — PointPillars object detector
`pointPillarsObjectDetector` object

PointPillars object detector, specified as a `pointPillarsObjectDetector` object.

- You can train an untrained object detector using the training options.
- You can continue training a pretrained detector with additional training data, or perform more training iterations to improve detector accuracy.

options — Training options
`TrainingOptionsSGDM` object | `TrainingOptionsRMSProp` object | `TrainingOptionsADAM` object

Training options, specified as a `TrainingOptionsSGDM`, `TrainingOptionsRMSProp`, or `TrainingOptionsADAM` object returned by the `trainingOptions` function. To specify the solver name and other options for network training, use the `trainingOptions` function.

Note The `trainPointPillarsObjectDetector` function supports these training options.

Name-Value Arguments	Supported Options
ExecutionEnvironment	• "auto", "gpu", "cpu" • For "multi-gpu", "parallel" set the DispatchInBackground argument to false.
ResetInputNormalization	false
BatchNormalizationStatistics	"moving"
OutputNetwork	"last-iteration"

---

checkpoint — Saved detector checkpoint
`pointPillarsObjectDetector` object

Saved detector checkpoint, specified as a `pointPillarsObjectDetector` object. To periodically save a detector checkpoint during training, specify `CheckpointPath`. To control how frequently check points are saved see the `CheckPointFrequency` and `CheckPointFrequencyUnit` training options.

To load a checkpoint for a previously trained detector, first load the corresponding MAT file from the checkpoint path. Then extract the object detector from the loaded data. For example, if the `CheckpointPath` property of your `options` object is `'/checkpath'`, you can load a checkpoint MAT file by using this code.

```
data = load("/checkpath/pointpillars_checkpoint__216__2018_11_16__13_34_30.mat");
checkpoint = data.detector;
```

The name of the MAT file includes the iteration number and timestamp of when the detector checkpoint was saved. The MAT file saves the detector in the `detector` variable. Use the `trainPointPillarsObjectDetector` function to train the detector.

```
pointPillarsDetector = trainPointPillarsObjectDetector(trainingData,checkpoint,options);
```

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example:
`trainPointPillarsObjectDetector(data,detector,ExperimentManager=expMonitorObj)` specifies the `ExperimentManger` value by using an `experiments.Monitor` object, `expMonitorObj`.

ExperimentManager — Detector training experiment monitoring
`'none'` (default) | `experiments.Monitor` object

Detector training experiment monitoring, specified as an `experiments.Monitor` object for use with the **Experiment Manager** app. You can use this object to track the progress of training, update information fields in the training results table, record values of the metrics used by the training, and to produce training plots. For an example using this app, see "Train Object Detectors in Experiment Manager".

Information monitored during training:

- Training loss at each iteration.
- Learning rate at each iteration.

Validation information when the training `options` input contains validation data:

- Validation loss at each iteration.

Output Arguments

detector — Trained PointPillars object detector
`pointPillarsObjectDetector` object

Trained PointPillars object detector, returned as `pointPillarsObjectDetector` object.

info — Training progress information
structure array

Training progress information, returned as a structure array with these fields. Each field corresponds to a stage of training.

- `TrainingLoss` — Training loss at each iteration. The training loss is the mean squared error (MSE), calculated as the sum of the localization error, confidence loss, and classification loss.
- `ValidationLoss` — Validation loss at each iteration.

Each field is a numeric vector with one element per training iteration. If the function does not calculate a value at a specific iteration, it returns a value of `NaN` for that iteration. The structure contains `ValidationLoss` only when `options` specifies validation data.

Version History
Introduced in R2021b

Support for Experiment Manager App

You can now specify the `ExperimentManager` value by using an `experiments.Monitor` object as a name-value argument to the function. You can use the `experiments.Monitor` object with the **Experiment Manager** app.

For more information on using this app, see "Train Object Detectors in Experiment Manager".

See Also

Apps
Lidar Labeler | **Lidar Viewer**

Functions
`trainingOptions` | `trainRCNNObjectDetector` | `fileDatastore` | `combine` | `lidarObjectDetectorTrainingData`

Objects
`pointPillarsObjectDetector` | `yolov2ObjectDetector`

Topics
"Lidar 3-D Object Detection Using PointPillars Deep Learning"
"Code Generation For Lidar Object Detection Using PointPillars Deep Learning"
"Unorganized to Organized Conversion of Point Clouds Using Spherical Projection"
"Getting Started with PointPillars"
"Getting Started with Point Clouds Using Deep Learning"
"Datastores for Deep Learning" (Deep Learning Toolbox)

pointnetplusLayers

Create PointNet++ segmentation network

Syntax

```
lgraph = pointnetplusLayers(numPoints,pointsDim,numClasses)
lgraph = pointnetplusLayers( ___ ,Name=Value)
```

Description

PointNet++ is a neural network that predicts point-wise labels for an unorganized lidar point cloud. The network partitions the input points into a set of clusters and then extracts the features using a multi-layer perceptron (MLP) network. To use this network for semantic segmentation, train it using the `trainNetwork` function.

`lgraph = pointnetplusLayers(numPoints,pointsDim,numClasses)` creates a PointNet++ segmentation network and returns it as `lgraph`, a `layerGraph` object.

`lgraph = pointnetplusLayers( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in the preceding syntax. For example, `pointnetplusLayers(numPoints,pointsDim,numClasses,ClusterSize=32)` creates a PointNet++ network with 32 points in each cluster.

Examples

Create and Analyze Custom PointNet++ Network

Define the input parameters for a custom PointNet++ network.

```
numPoints = 10000;
pointsDim = 3;
numClasses = 8;
```

Create the custom PointNet++ network.

```
lgraph = pointnetplusLayers(numPoints,pointsDim,numClasses, ...
    NormalizationLayer="instance", ...
    NumSetAbstractionModules=3, ...
    NumClusters=2048, ...
    ClusterRadius=0.1, ...
    ClusterSize=32, ...
    PointNetLayerSize=32);
```

Analyze the network using the `analyzeNetwork` (Deep Learning Toolbox) function.

```
analyzeNetwork(lgraph)
```

You can train this network using the `trainNetwork` (Deep Learning Toolbox) function and use it for different applications. To learn more about training the PointNet++ network, see the "Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning" example.

Input Arguments

numPoints — Number of points in input point cloud
positive integer

Number of points in the input point cloud, specified as a positive integer.

pointsDim — Dimensions of input point cloud matrix
positive integer greater than or equal to 3

Dimensions of the input point cloud data matrix, specified as a positive integer greater than or equal to 3. The matrix must contain the *xyz* coordinates and any additional data such as range, mask, and intensity.

numClasses — Number of classes
positive integer greater than 1

Number of classes the network must be configured to classify, specified as a positive integer greater than 1.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `lgraph = pointnetplusLayers(numPoints,pointsDim,numClasses,NumSetAbstractionModules=3);`

NormalizationLayer — Type of normalization
`"batch"` (default) | `"instance"`

Type of normalization used in network, specified as `"batch"` or `"instance"`.

- `"batch"` — Use a `batchNormalizationLayer`
- `"instance"` — Use an `instanceNormalizationLayer`

Data Types: `string` | `char`

NumSetAbstractionModules — Number of set abstraction modules
4 (default) | positive integer

Number of set abstraction modules for the encoder subnetwork, specified as a positive integer. The decoder subnetwork contains the same number of feature propagation modules.

NumClusters — Number of clusters
1024 (default) | positive integer

Number of clusters to group the input points into, specified as a positive integer. The value of NumClusters must be a power of two in the range [4^N, `numPoints`], where *N* is the number of set abstraction modules.

This value specifies the number of clusters in the first set abstraction module. For subsequent set abstraction modules, the function automatically computes the number of clusters as *K/4*, where *K* is the number of clusters from the previous set abstraction module.

`ClusterRadius` — Cluster radius of input points
`0.1` (default) | positive scalar in range (0, 1]

Cluster radius of the input points, specified as a positive scalar in the range (0, 1].

This value specifies the cluster radius for the first set abstraction module. For subsequent set abstraction modules, the function automatically computes the cluster radius as twice the value from the previous set abstraction module.

`ClusterSize` — Number of points in each cluster
`32` (default) | positive integer

Number of points in each cluster, specified as a positive integer. For a given cluster radius in each set abstraction module, this value must be a power of two less than $K/4^{(N-2)}$. *K* is the number of clusters in the network and *N* is the number of set abstraction modules.

This value is constant across all set abstraction modules.

`PointNetLayerSize` — Size of first layer in MLP network
`32` (default) | positive integer

Size of first layer in the MLP network of the set abstraction module, specified as a positive integer. Each set abstraction module contains a mini PointNet with a shared MLP network implemented using 1-by-1 convolution. The sizes of first, second, and third layers in the shared MLP network are *S*, *S*, *2*S* which correspond to the number of filters in the first, second and third convolution layers, respectively.

This value specifies the size of first layer in the MLP network of the first set abstraction module. For each subsequent set abstraction modules, the value of *S* is twice the value of *S* from the previous set abstraction module.

Output Arguments

`lgraph` — Output PointNet++ network
`layerGraph` object

Output PointNet++ network, returned as a `layerGraph` object.

More About

PointNet++ Network

A PointNet++ network has an encoder subnetwork with set abstraction modules, followed by a corresponding decoder subnetwork with feature propagation modules.

- The set abstraction module identifies new cluster centers using farthest point sampling and groups the points into clusters using the ball query algorithm. The feature propagation module interpolates the points using inverse weighted distance based on the k-nearest neighbors algorithm.

- The function creates the network with single scale grouping (SSG) architecture.
- The function uses the narrow-normal weight initialization method to initialize the weights of each convolution layer in the network.
- The function initializes all bias terms to zero.

Version History
Introduced in R2021b

See Also

Apps
Lidar Labeler | **Lidar Viewer**

Functions
squeezesegv2Layers | semanticseg | pixelClassificationLayer | focalLossLayer

Topics
"Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning"
"Code Generation For Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning"
"Getting Started with PointNet++"
"Getting Started with Point Clouds Using Deep Learning"
"Datastores for Deep Learning" (Deep Learning Toolbox)

detectISSFeatures

Detect ISS feature points in point cloud

Syntax

```
points = detectISSFeatures(ptCloud)
[points,indices] = detectISSFeatures(ptCloud)
___ = detectISSFeatures(ptCloud,Name=Value)
```

Description

`points = detectISSFeatures(ptCloud)` detects intrinsic shape signature (ISS) feature points in the input point cloud `ptCloud`. ISS is a 3-D shape representation method for 3-D object recognition. ISS feature points are the points rich in 3-D structural variation in their neighbourhoods.

`[points,indices] = detectISSFeatures(ptCloud)` additionally returns the linear indices for the detected ISS feature points.

`___ = detectISSFeatures(ptCloud,Name=Value)` specifies options using one or more name-value arguments in addition to any combination of output arguments from previous syntaxes. For example, `detectISSFeatures(ptCloud,Radius=0.05)` computes the ISS saliency within a 0.05 m radius around each point when identifying the feature points.
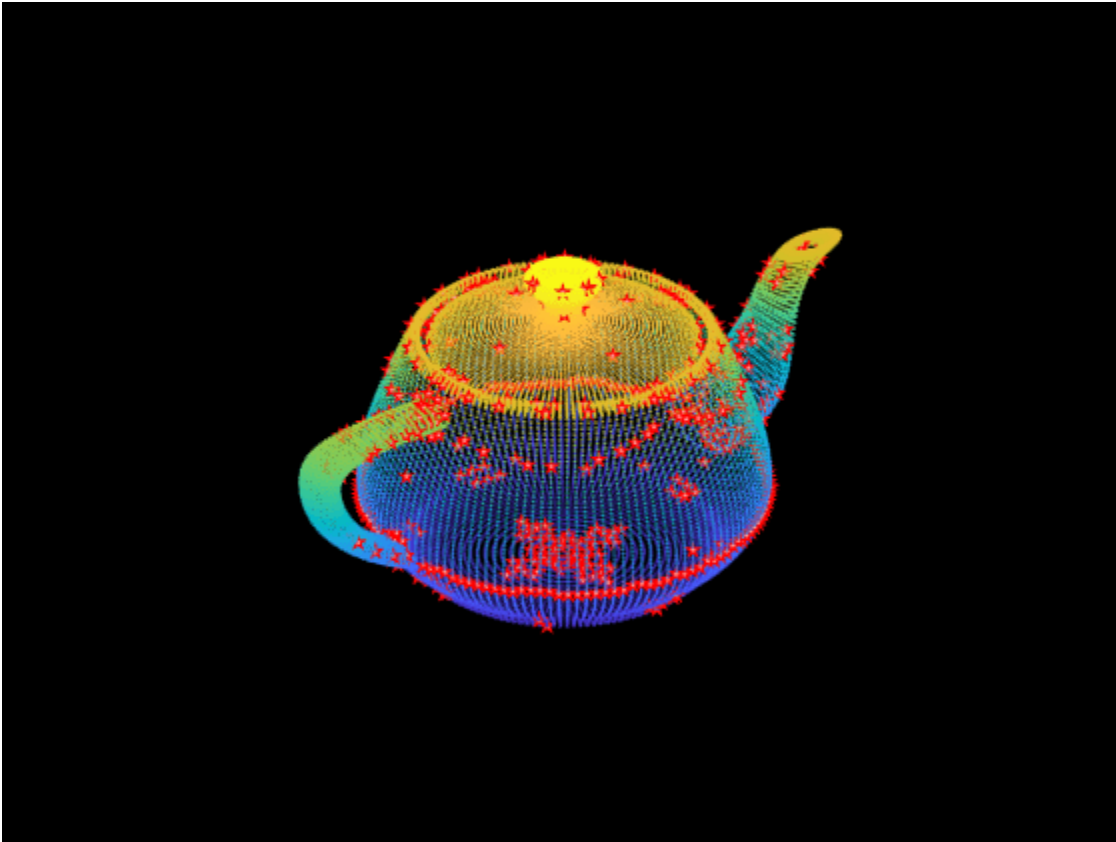
Examples

Detect ISS Feature Points in Point Cloud

Read a point cloud from a PLY file into the workspace.

```
ptCloud = pcread("teapot.ply");
```

Detect ISS feature points in the point cloud, and display the point cloud and ISS feature points.

```
points = detectISSFeatures(ptCloud);
figure(Name="Detected feature points")
pcshow(ptCloud)
hold on
plot3(points(:,1),points(:,2),points(:,3),"pentagram", ...
    MarkerSize=5,MarkerFaceColor=[1 0.6 0.6],Color="red")
```

Align Point Clouds Using Feature Extraction and Matching

Read a point cloud from a PCD file into the workspace.

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

Transform the input point cloud by applying both rotation and translation.

```
eulerAngles = [0 0 30];
trans = [5 5 0];
tform = rigidtform3d(eulerAngles,trans);
ptCloudTform = pctransform(ptCloud,tform);
```

Detect ISS feature points in the original, fixed, and transformed, moving point clouds.

```
[~,indicesFixed] = detectISSFeatures(ptCloud);
[~,indicesMoving] = detectISSFeatures(ptCloudTform);
```

Extract FPFH features for the detected feature points in each point cloud.

```
ptCloudFixed = select(ptCloud,indicesFixed);
fixedFeature = extractFPFHFeatures(ptCloudFixed);
ptCloudMoving = select(ptCloudTform,indicesMoving);
movingFeature = extractFPFHFeatures(ptCloudMoving);
```

Match the features between the fixed and moving point clouds.

```
[matchingPairs,scores] = pcmatchfeatures(fixedFeature,movingFeature, ...
                    ptCloudFixed,ptCloudMoving,Method="Exhaustive");
```

Estimate the transform using the matched features.

```
fixedPts = select(ptCloudFixed,matchingPairs(:,1));
matchingPts = select(ptCloudMoving,matchingPairs(:,2));
estimatedTform = estgeotform3d(fixedPts.Location, ...
          matchingPts.Location,"rigid");
```

Align the point clouds using the estimated transform.

```
ptCloudAligned = pctransform(ptCloudTform,invert(estimatedTform));
```

Visualize the aligned point clouds.

```
pcshowpair(ptCloud,ptCloudAligned)
title("Aligned Point Clouds")
```
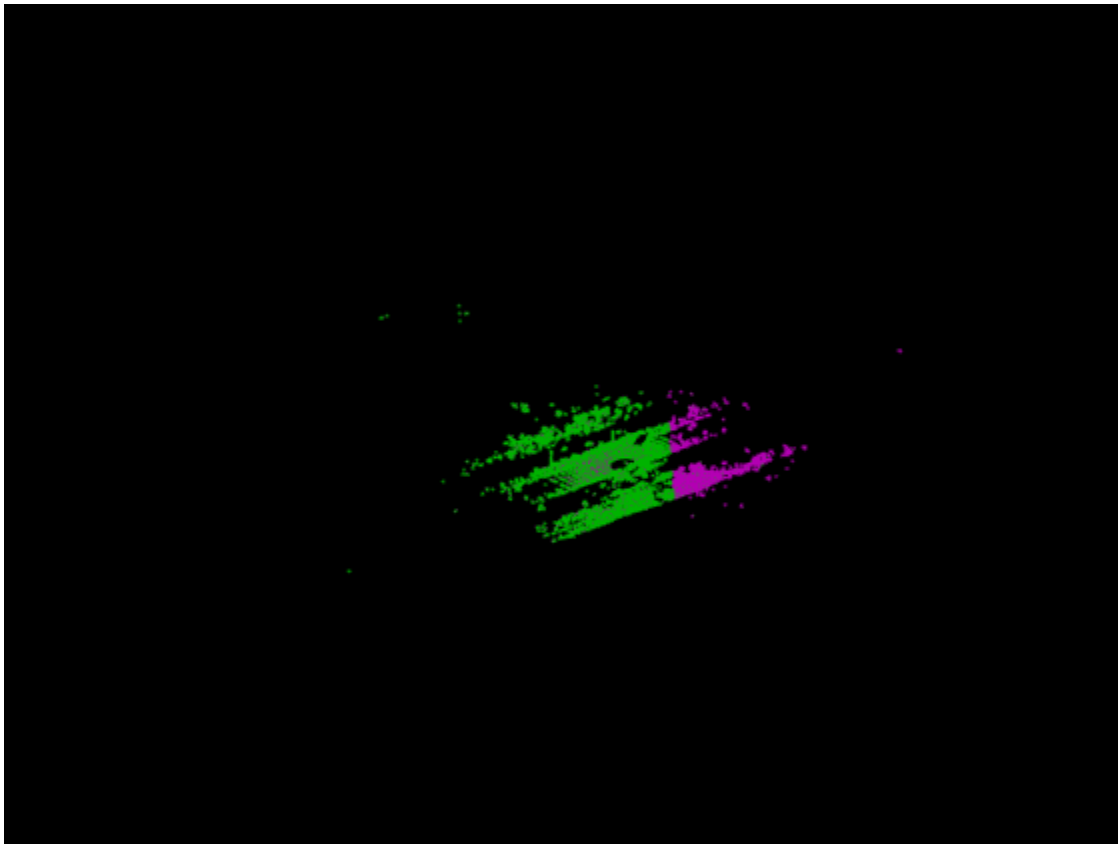

Register and Align Point Clouds Using ICP Algorithm with ISS Feature Points

Load a MAT file that contains point cloud data into the workspace. Extract the first two point clouds from the data.

```
inputData = load("livingRoom.mat");
movingPtCloud = inputData.livingRoomData{1};
fixedPtCloud = inputData.livingRoomData{2};
```

Visualize the extracted point clouds.

```
pcshowpair(movingPtCloud,fixedPtCloud,VerticalAxis="Y",VerticalAxisDir="Down")
title("Point Clouds Before Alignment")
```
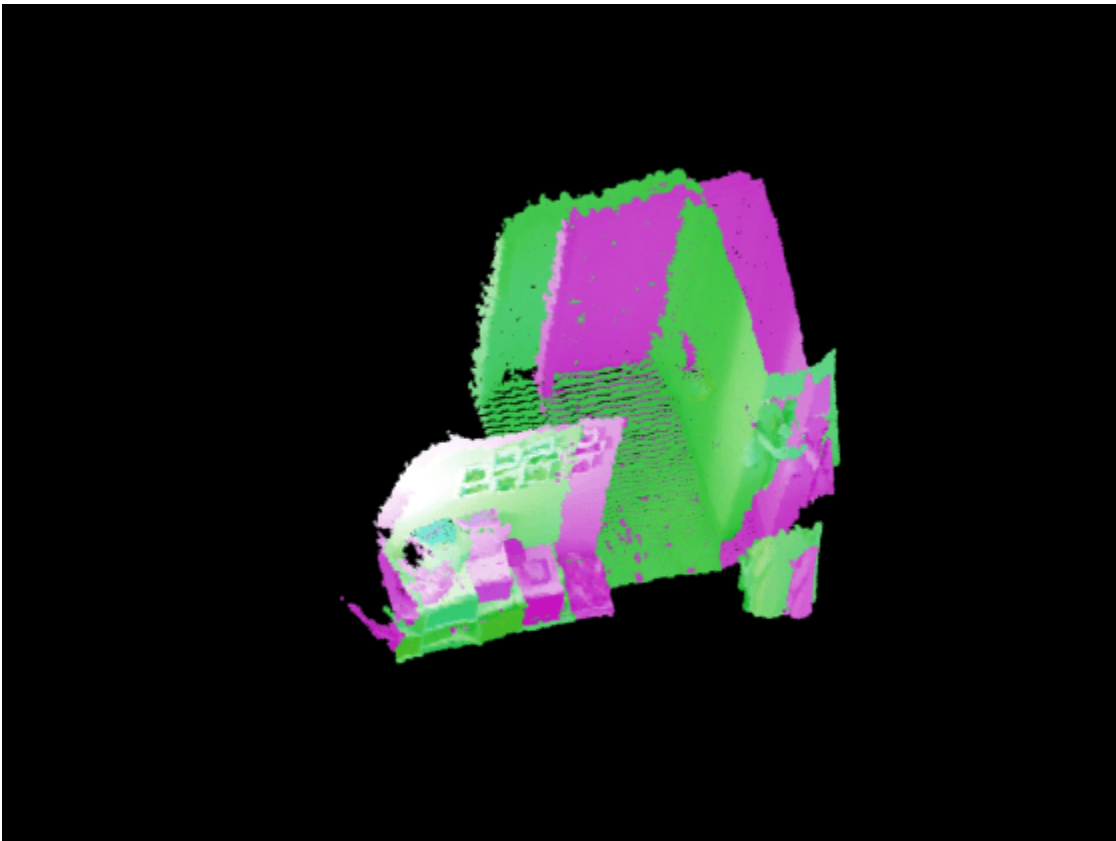

Detect ISS feature points in the point clouds.

```
[~,indicesMoving] = detectISSFeatures(movingPtCloud);
[~,indicesFixed] = detectISSFeatures(fixedPtCloud);
```

Extract the feature points from each point cloud, and register the point clouds to one another.

```
movingISSPtCloud = select(movingPtCloud,indicesMoving);
fixedISSPtCloud = select(fixedPtCloud,indicesFixed);
tform = pcregistericp(movingISSPtCloud,fixedISSPtCloud, ...
Extrapolate=true);
```

Align the point clouds using the registered transform, and visualize the results.

```
movingPtCloudAligned = pctransform(movingPtCloud,tform);
figure
pcshowpair(movingPtCloudAligned,fixedPtCloud,VerticalAxis="Y",VerticalAxisDir="Down")
title("Aligned Point Clouds")
```
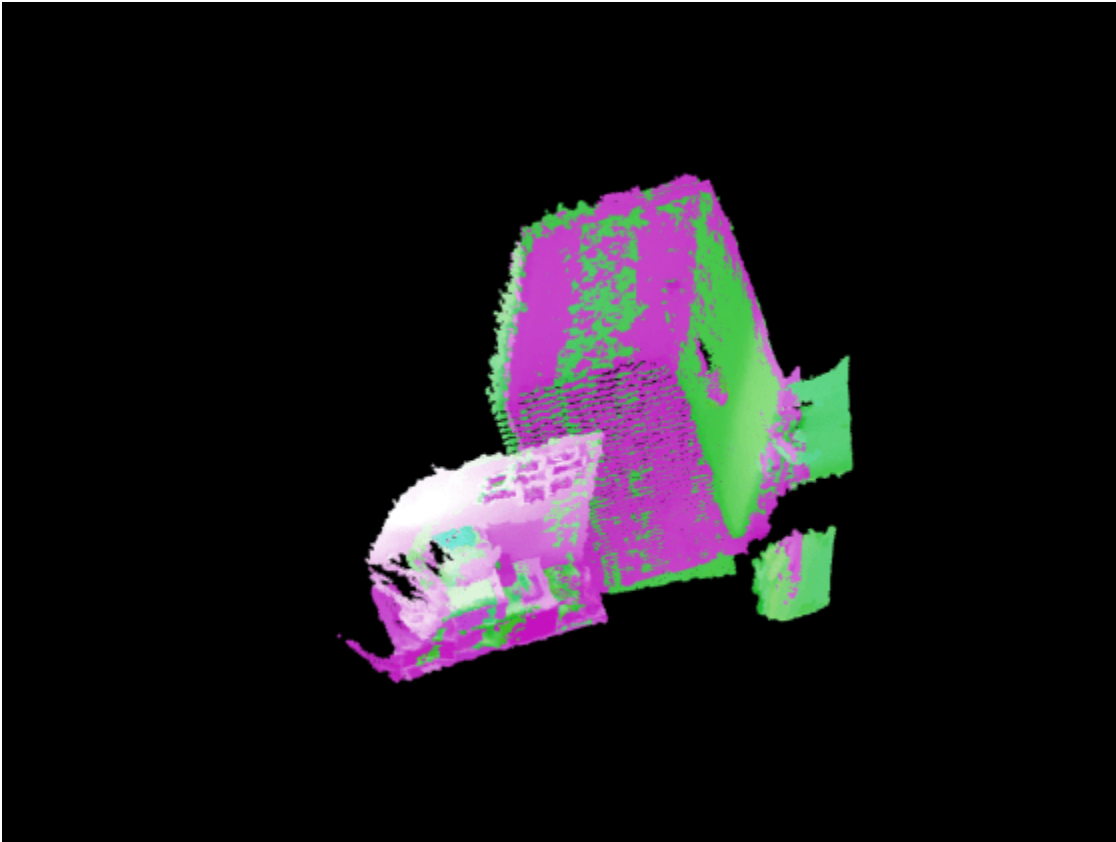
Input Arguments

`ptCloud` — Input point cloud
pointCloud object

Input point cloud, specified as a `pointCloud` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `detectISSFeatures(ptCloud,Radius=0.05)` computes the ISS saliency within a 0.05 m radius around each point when identifying the feature points.

`Radius` — Neighborhood radius for computing saliency
positive scalar

Neighborhood radius for computing ISS saliency, specified as positive scalar. The ISS saliency of a point is a measure of the richness of 3-D structural variation in its neighborhood, which determines whether or not the point is a feature point. The function computes a scatter matrix within the specified radius of each point to determine its ISS saliency and identify feature points. The default value is six times the average distance from each point in the input point cloud to its nearest neighbouring point. Units are in meters.

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

NonMaxRadius — Neighborhood radius in which to apply non-maxima suppression
positive scalar

Neighborhood radius in which to apply the non-maxima suppression algorithm, specified as positive scalar. The default value is four times the average distance from each point in the input point cloud to its nearest neighbouring point. Increasing this value can reduce the number of detected feature points. Units are in meters.

Note `NonMaxRadius` value must be less than or equal to the value of `Radius`. Otherwise, the function reduces the value of `NonMaxRadius` to the value of `Radius`.

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

MaxGamma21 — Ratio of second eigenvalue to first eigenvalue
`0.975` (default) | positive scalar in range `(0, 1]`

Ratio of the second eigenvalue to first eigenvalue in the scatter matrix, specified as a positive scalar in the range `(0, 1]`. The function uses this ratio to define the x-, y-, z-axes of the intrinsic reference frame. For a lower ratio value, the function excludes the points with similar 3-D features along the first and second principal axes.

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

MaxGamma32 — Ratio of third eigenvalue to second eigenvalue
`0.975` (default) | positive scalar in range `(0, 1]`

Ratio of the third eigenvalue to second eigenvalue in the scatter matrix, specified as a positive scalar in the range `(0, 1]`. The function uses this ratio to define the x-, y-, z-axes of the intrinsic reference frame. For a lower ratio value, the function excludes the points with similar 3-D features along the second and third principal axes.

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

MinNeighbors — Minimum neighbors required for ISS feature point
5 (default) | positive integer

Minimum number of neighbors required for an ISS feature point, specified as a positive integer. Increasing this value can reduce the total number of feature points detected.

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

Output Arguments

points — ISS feature points
M-by-3 matrix

ISS feature points in the input point cloud, returned as an *M*-by-3 matrix. *M* is the total number of feature points. Each row contains the [*x y z*] coordinates of a feature point.

indices — Indices for feature points
M-by-1 matrix

Linear indices for the detected ISS feature points, returned as an *M*-by-1 matrix.

Algorithms

Intrinsic shape signatures (ISS) are a method of 3-D shape representation. ISS feature points are rich in 3-D structural variations in their neighbourhood. This method has applications in modeling, visualization, and classification of 3-D point clouds.

To detect ISS feature points in a point cloud, the `detectISSFeatures` function follows these steps.

- Computes a point scatter matrix within the specified `Radius` around each point.
- Computes the eigenvalues λ_1, λ_2, and λ_3 in decreasing order of magnitude for the scatter matrix. These eigenvalues represent a direction in the 3-D space based on the number of point position variations.
- Using the eigenvalues, the function defines a view-independent intrinsic reference frame with the principal *x*-, *y*-, *z*-axes.
- Uses λ_2/λ_1, λ_3/λ_2 as criteria to avoid the points with similar spatial spread along the principal axes while detecting feature points. You can specify eigenvalue ratios for λ_2/λ_1 and λ_3/λ_2 using the `MaxGamma21` and `MaxGamma32` arguments, respectively.
- Computes the ISS saliency for each point using the smallest eigenvalue, λ_3. ISS feature point is the point with maximum ISS saliency within the specified radius around each point.
- You can further process these feature points to match point clouds, estimate pose transformations, and detect 3-D objects.

Version History
Introduced in R2022a

See Also

Apps
Lidar Labeler | **Lidar Viewer** | **Lidar Camera Calibrator**

Functions
extractFPFHFeatures | extractEigenFeatures | pcmatchfeatures | detectLOAMFeatures

pc2scan

Convert 3-D point cloud into 2-D lidar scan

Syntax

```
scan = pc2scan(ptCloudIn)
scan = pc2scan(ptCloudIn,tform)
scan = pc2scan(ptCloudIn,Name=Value)
```

Description

`scan = pc2scan(ptCloudIn)` converts an input point cloud into a 2-D lidar scan and returns it as a `lidarScan` object.

`scan = pc2scan(ptCloudIn,tform)` specifies the transformation `tform` of the input point cloud into the 2-D lidar sensor coordinate system, and then converts it into a 2-D lidar scan. `tform` represents the pose of the 2-D lidar sensor with respect to the point cloud origin.

`scan = pc2scan(ptCloudIn,Name=Value)` specifies options using one or more name-value arguments. For example, `pc2scan(ptCloudIn,ElevationAngleTolerance=5)` selects points with elevation angles in the range [-5, 5] degrees to generate the scan.

Examples

Convert Point Cloud to Lidar Scan

Create a velodyne PCAP file reader object.

```
veloReader = velodyneFileReader("lidarData_ConstructionRoad.pcap","HDL32E");
```

Read point cloud data from 0.3 seconds after the start time of the file by using the `readFrame` method. Display the point cloud.

```
veloReader.CurrentTime = veloReader.StartTime + seconds(0.3);
ptCloud = readFrame(veloReader);
pcshow(ptCloud)
```
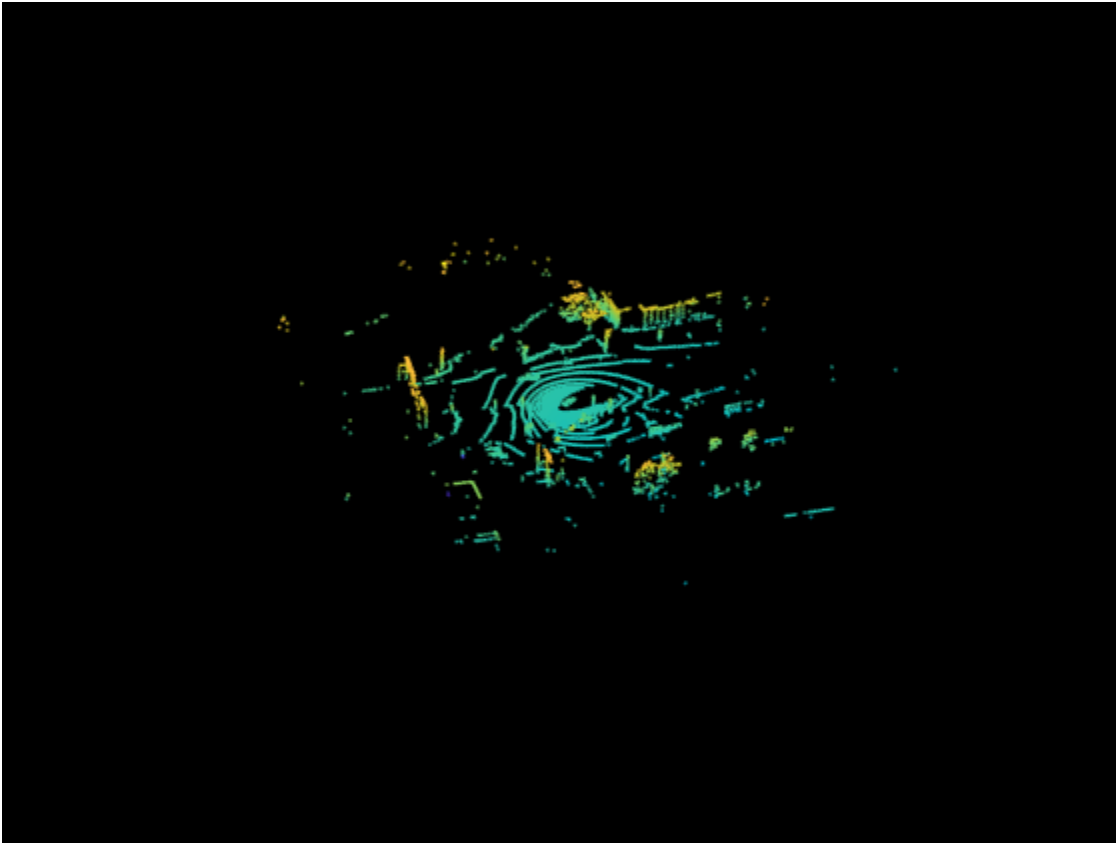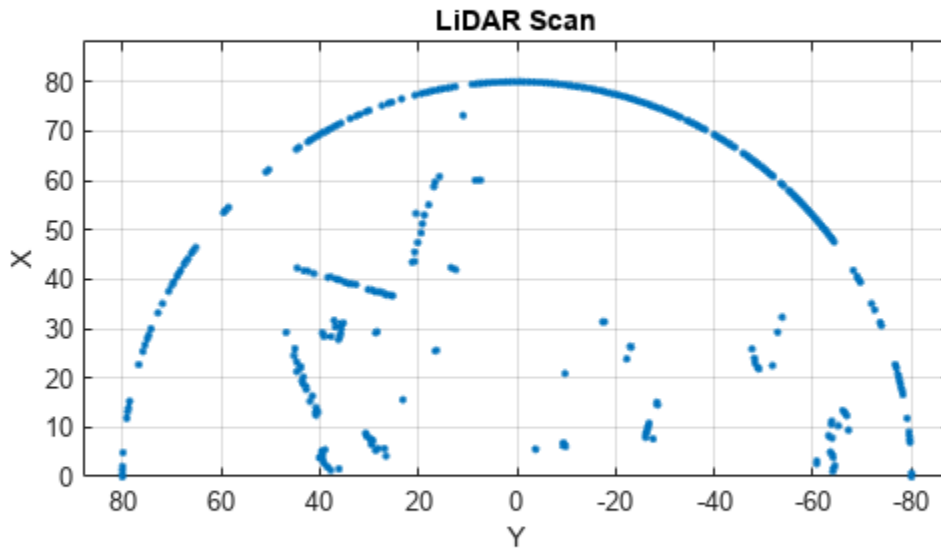
Segment the point cloud and remove ground points.

```
groundPtsIdx = segmentGroundFromLidarData(ptCloud);
nonGroundPtCloud= select(ptCloud,~groundPtsIdx,OutputSize="full");
```

Convert the point cloud into 2-D lidar scan and display the output.

```
scan = pc2scan(nonGroundPtCloud);
figure
plot(scan)
```

LiDAR Scan

Input Arguments

ptCloudIn — Input point cloud
`pointCloud` object

Input point cloud, specified as a `pointCloud` object.

By default, the function assumes the input point cloud is in the sensor coordinate system with the point cloud origin at the sensor center, and the sensor capturing the point cloud has no rotations about the coordinate axes.

You must specify the `Location` property of the `pointCloud` object in meters.

tform — Rigid transformation between lidar sensor and point cloud origin
`rigidtform3d` object

Rigid transformation between the 2-D lidar sensor and the point cloud origin, specified as a `rigidtform3d` object. The function uses `tform` to convert the input point cloud into the 2-D lidar sensor coordinate system.

Note By default, the function assumes the input point cloud is in the sensor coordinate system, and the sensor capturing the point cloud has no rotations about the coordinate axes if you do not specify `tform`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `pc2scan(ptCloudIn,ElevationAngleTolerance=5)` selects points with elevation angles in the range [-5, 5] degrees to generate the scan.

`ElevationAngleTolerance` — Elevation angle tolerance
`1` (default) | positive scalar

Elevation angle tolerance, specified as a positive scalar. When generating the 2-D lidar scan, the function selects only the points of the input point cloud with elevation angles in the range [-`ElevationAngleTolerance`, `ElevationAngleTolerance`]. Lower values of `ElevationAngleTolerance` can result in a more accurate output scan. Units are in degrees.

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

`ScanAngleResolution` — Angle between consecutive scan lines
`0.5` (default) | positive scalar

Angle between the consecutive scan lines of the 2-D lidar sensor, specified as a positive scalar. Lower values can result in a finer scan output. Units are in degrees.

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

`ScanRangeLimits` — Scanning range of lidar sensor
`[eps 80]` (default) | two-element vector

Scanning range of the 2-D lidar sensor, specified as a two-element vector of the form `[min max]`. Units are in meters.

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

`ScanAngleLimits` — Scanning angle limits of lidar sensor
`[-90 90]` (default) | two-element vector

Scanning angle limits of the 2-D lidar sensor, specified as a two-element vector of the form `[min max]`. This vector defines the horizontal field of view of the sensor. Units are in degrees.

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

Output Arguments

`scan` — Output 2-D lidar scan
`lidarScan` object

Output 2-D lidar scan, returned as a `lidarScan` object.

Algorithms

The function follows these steps to convert a point cloud into a 2-D lidar scan.

- Converts the input point cloud to the 2-D lidar sensor coordinate system using the specified transformation `tform`. If you do not specify `tform`, the function assumes the data is in the sensor coordinate system.
- Projects the points on the *xy*-plane. For each projected point, the function computes the angle and range. The angle is counter-clockwise positive along the *x*-axis, and the range is the distance from the origin.
- Selects the points whose angle and range are within the specified `ScanAngleLimits` and `ScanRangeLimits`, respectively.
- Selects the points whose elevation angle is within the specified `ElevationAngleTolerance`.
- Computes the scan line index for each point from the measured angle. The function assigns a scan angle to each scan line, starting from the first scan line and assigning it the value of the first element of `ScanAngleLimits`. The function then increments the angle in steps of the specified `ScanAngleResolution` to the remaining scan lines.
- By default, each scan line index has the maximum range. For multiple scan lines with same index, the function assigns the range value of the point closest to the origin.
- Generates the 2-D lidar scan using the scan angles and the range values.

Version History
Introduced in R2022a

Supports `rigidtform3d` objects

You can now specify `tform` as a `rigidtform3d` object, which uses the premultiply convention. Although you can still specify `tform` as a `rigid3d` object, this object is not recommended because it uses the postmultiply convention. For more information, see "Migrate Geometric Transformations to Premultiply Convention".

See Also

Apps
Lidar Labeler | **Lidar Viewer** | **Lidar Camera Calibrator**

Functions
`matchScansGrid` | `matchScans` | `lidarScan` | `transformScan` | `pc2dem`

sampleLidarData

Sample 3-D bounding boxes and corresponding points from training data

Syntax

```
[pcds,blds] = sampleLidarData(trainingData,classNames)
[pcds,blds] = sampleLidarData( ___ ,Name=Value)
```

Description

`[pcds,blds] = sampleLidarData(trainingData,classNames)` samples 3-D bounding boxes and the corresponding points from the specified training data and returns them as datastore objects.

`[pcds,blds] = sampleLidarData( ___ ,Name=Value)` specifies one or more name-value arguments in addition to all input arguments from the previous syntax. For example, `sampleLidarData(trainingData,classNames,MinPoints=20)` samples only boxes that have a minimum of 20 points inside them.

Examples

Perform Ground Truth Data Augmentation on Point Cloud

Load a point cloud and its class labels into the workspace.

```
unzip("sampleWPIPointClouds.zip",tempdir);
dataLocation = fullfile(tempdir,"sampleWPIPointClouds");
load("sampleWPILabels.mat","trainLabels");
```
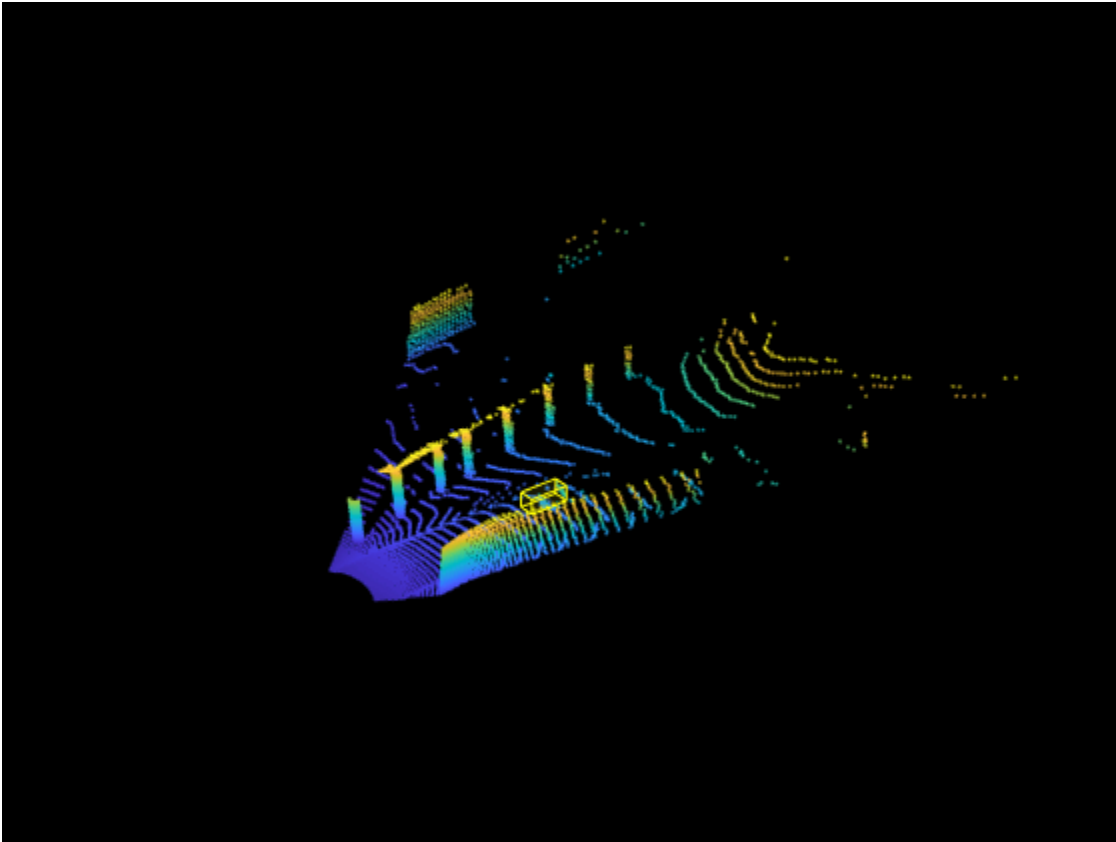
Create a datastore for training data.

```
pcds = fileDatastore(dataLocation,"ReadFcn",@(x) pcread(x));
blds = boxLabelDatastore(trainLabels);
trainingData = combine(pcds,blds);
```

Define the class names to sample from the input data. Use the `sampleLidarData` function to sample the corresponding bounding boxes.

```
classNames = {'car'};
[pcdsSampled,bldsSampled] = sampleLidarData(trainingData,classNames,Verbose=false);
cdsSampled = combine(pcdsSampled,bldsSampled);
```
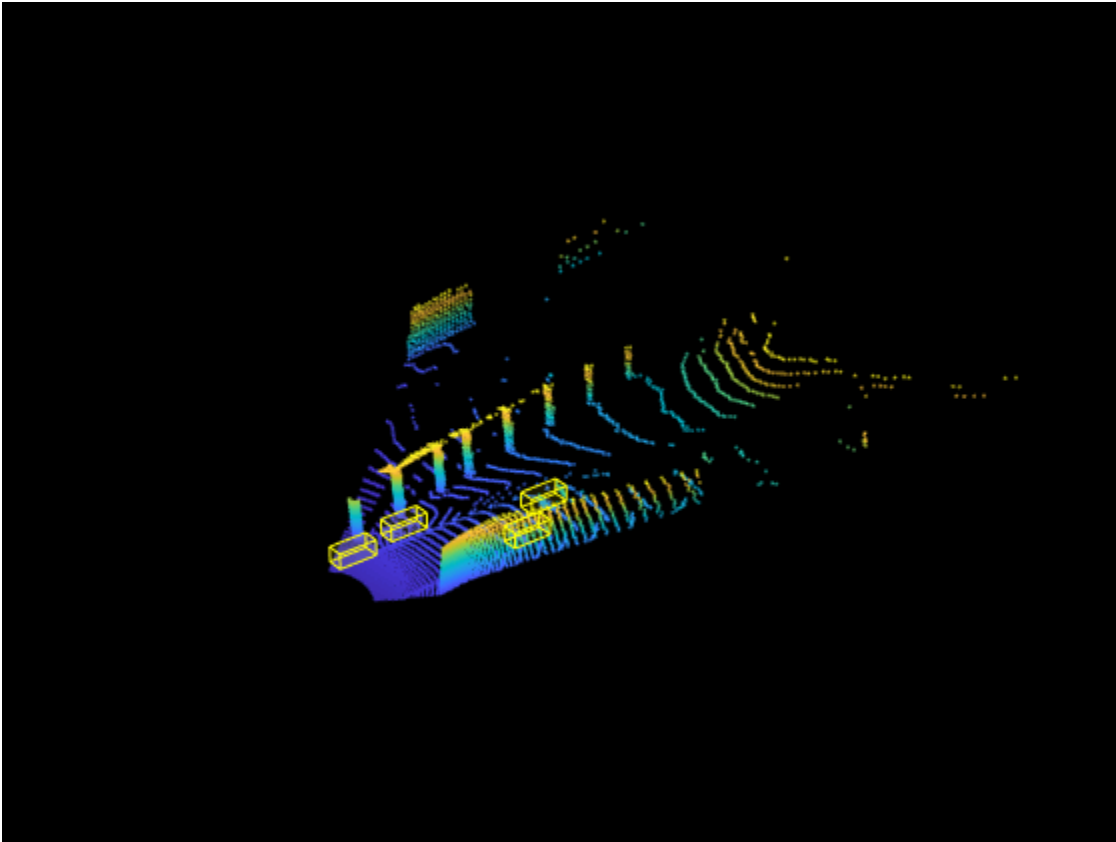
Read a point cloud from the training data.

```
pcBoxLabels = read(trainingData);
figure
pcshow(pcBoxLabels{1,1}.Location)
showShape(cuboid=pcBoxLabels{1,2},Opacity=0.1, ...
         Color="yellow",LineWidth=0.5);
title("Original Point Cloud")
```

Augment the point cloud data `pcBoxLabels` with points sampled from the datastore `cdsSampled` using the `pcBboxOversample` function.

```
totalObjects = 5;
augmentedPcBoxLabels = pcBboxOversample(pcBoxLabels,cdsSampled,classNames,totalObjects);
figure
pcshow(augmentedPcBoxLabels{1,1}.Location)
showShape(cuboid=augmentedPcBoxLabels{1,2},Opacity=0.1, ...
        Color="yellow",LineWidth=0.5);
title("Augmented Point Cloud")
```

Input Arguments

`trainingData` — Input point cloud data
valid datastore object | table

Input point cloud data, specified as a valid datastore object or a table.

- If you specify a datastore object, your data must be set up such that using the `read` function on the datastore object returns a cell array or table with three columns. Each row corresponds to a point cloud, and the columns must follow this format.

  - First column — Organized or unorganized point cloud data, specified as a `pointCloud` object.
  - Second column — Bounding boxes, specified as a cell array containing an M-by-9 matrix. Each row of the matrix is of the form [*x y z length width height roll pitch yaw*], representing the location and dimension of a bounding box. *M* is the number of bounding boxes.
  - Third column — Labels, specified as a cell array containing an *M*-by-1 categorical vector containing the object class names.

  You can use the `combine` function to combine two or more datastores. For more information on creating datastore objects, see the `datastore` function.

- If you specify a table, the table must have two or more columns. The first column must contain point cloud file names. The point cloud files can be in any format supported by the `pcread` function. Each of the remaining columns contains a cell array that represents a single object class,

such as `Car`, or `Truck`. Each cell contains an *M*-by-9 matrix. Each row of the matrix is of the form [*x y z length width height roll pitch yaw*], specifying the location and dimensions of the bounding box in the corresponding point cloud. *M* is the number of bounding boxes.

`classNames` — Names of object classes
character vector | string scalar | vector of strings | cell array of character vectors

Name of the object classes, specified as a character vector, string scalar, vector of strings, or a cell array of character vectors. The function samples these classes from the input training data. For example, 'car', 'truck', or 'pedestrian'.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `sampleLidarData(trainingData,classNames,MinPoints=20)` samples only objects that have a minimum of 20 points inside them.

`MinPoints` — Minimum points required to sample object
0 (default) | positive scalar | *M*-element vector

Minimum number of points required to sample an object, specified as a positive scalar or an *M*-element vector. *M* is the number of classes. If the value is a vector, each element corresponds to the respective class. Otherwise the function uses the same value for all the classes.

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

`WriteLocation` — Folder in which to write sampled data
pwd(working folder location) (default) | character vector | string scalar

Folder in which to write the sampled data, specified as a character vector or string scalar. The folder must exist in the location specified, and you must have write permissions. By default, the function writes this data into the current working folder. The data consists of the sampled points and their respective box labels.

Data Types: `char` | `string`

`Verbose` — Display of data writing progress
`true` (default) | `false`

Display of data writing progress, specified as a logical `true` or `false`.

Data Types: `logical`

Output Arguments

`pcds` — File locations of points sampled from training data
`fileDatastore` object

File locations of the points sampled from the training data, returned as a `fileDatastore` object.

`blds` — Sampled 3-D bounding boxes and labels
`boxLabelDatastore` object

Sampled 3-D bounding boxes and labels, returned as a `boxLabelDatastore` object.

Algorithms

Lidar object detection techniques directly predict 3-D bounding boxes around objects of interest. Data augmentation helps you improve prediction accuracy and avoid overfitting issues while training.

You can perform ground truth data augmentation on point clouds using these steps.

1   Sample 3-D bounding boxes and the corresponding points from input training data using the `sampleLidarData` function.
2   Augment a point cloud randomly with the sampled bounding boxes by using the `pcBboxOversample` function. The function performs a collision test on the sampled boxes and the ground truth boxes of the input point cloud to avoid overlap.

This technique alleviates the class imbalance problem in lidar object detection.

Version History
Introduced in R2022a

See Also

Apps
Lidar Labeler | **Lidar Viewer**

Functions
`pcBboxOversample` | `transform` | `pointPillarsObjectDetector` | `pointnetplusLayers` | `squeezesegv2Layers`

Topics
"Data Augmentations for Lidar Object Detection Using Deep Learning"
"Lidar 3-D Object Detection Using PointPillars Deep Learning"

pcBboxOversample

Randomly augment point cloud data using objects

Syntax

```
augmentedPcBoxLabels = pcBboxOversample(pcBoxLabels,sampleData,classNames,
totalObjects)
```

Description

`augmentedPcBoxLabels = pcBboxOversample(pcBoxLabels,sampleData,classNames,
totalObjects)` randomly augments the specified point cloud data `pcBoxLabels` using objects with
specified class names `classNames` from the training data datastore `sampleData`.

Examples

Perform Ground Truth Data Augmentation on Point Cloud

Load a point cloud and its class labels into the workspace.

```
unzip("sampleWPIPointClouds.zip",tempdir);
dataLocation = fullfile(tempdir,"sampleWPIPointClouds");
load("sampleWPILabels.mat","trainLabels");
```
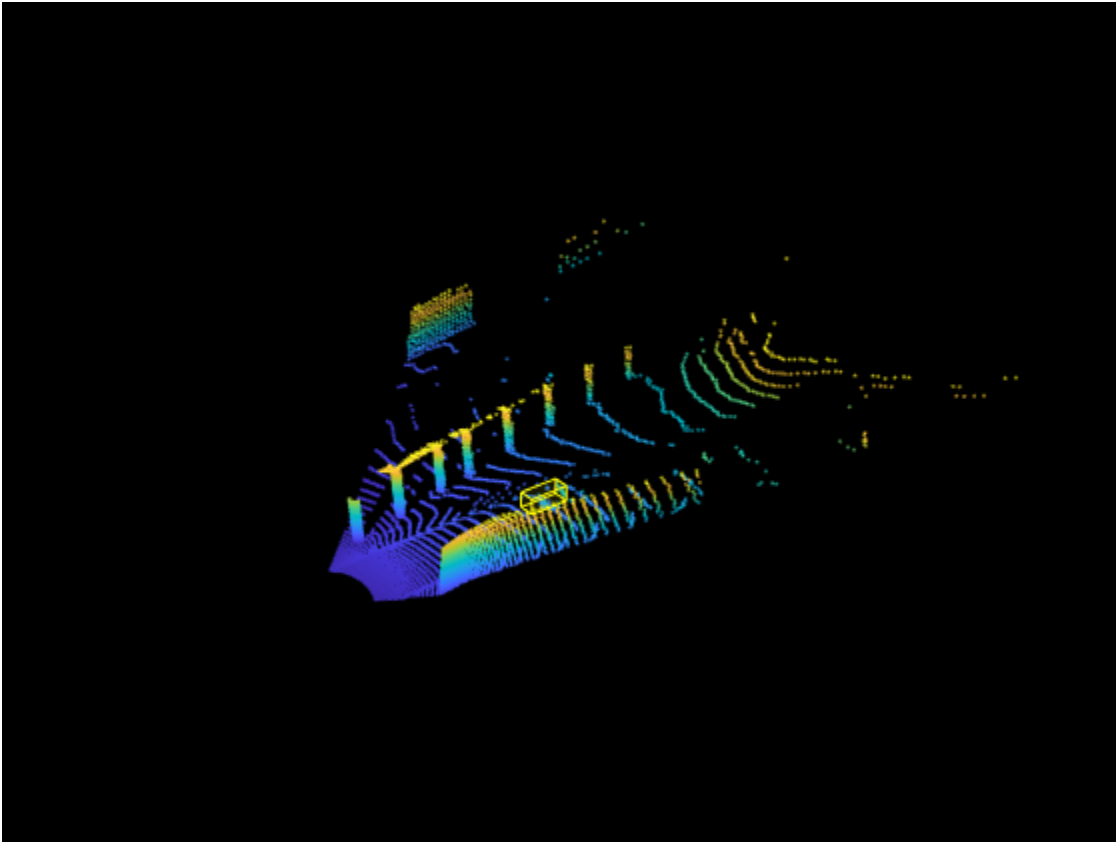
Create a datastore for training data.

```
pcds = fileDatastore(dataLocation,"ReadFcn",@(x) pcread(x));
blds = boxLabelDatastore(trainLabels);
trainingData = combine(pcds,blds);
```

Define the class names to sample from the input data. Use the `sampleLidarData` function to sample
the corresponding bounding boxes.

```
classNames = {'car'};
[pcdsSampled,bldsSampled] = sampleLidarData(trainingData,classNames,Verbose=false);
cdsSampled = combine(pcdsSampled,bldsSampled);
```
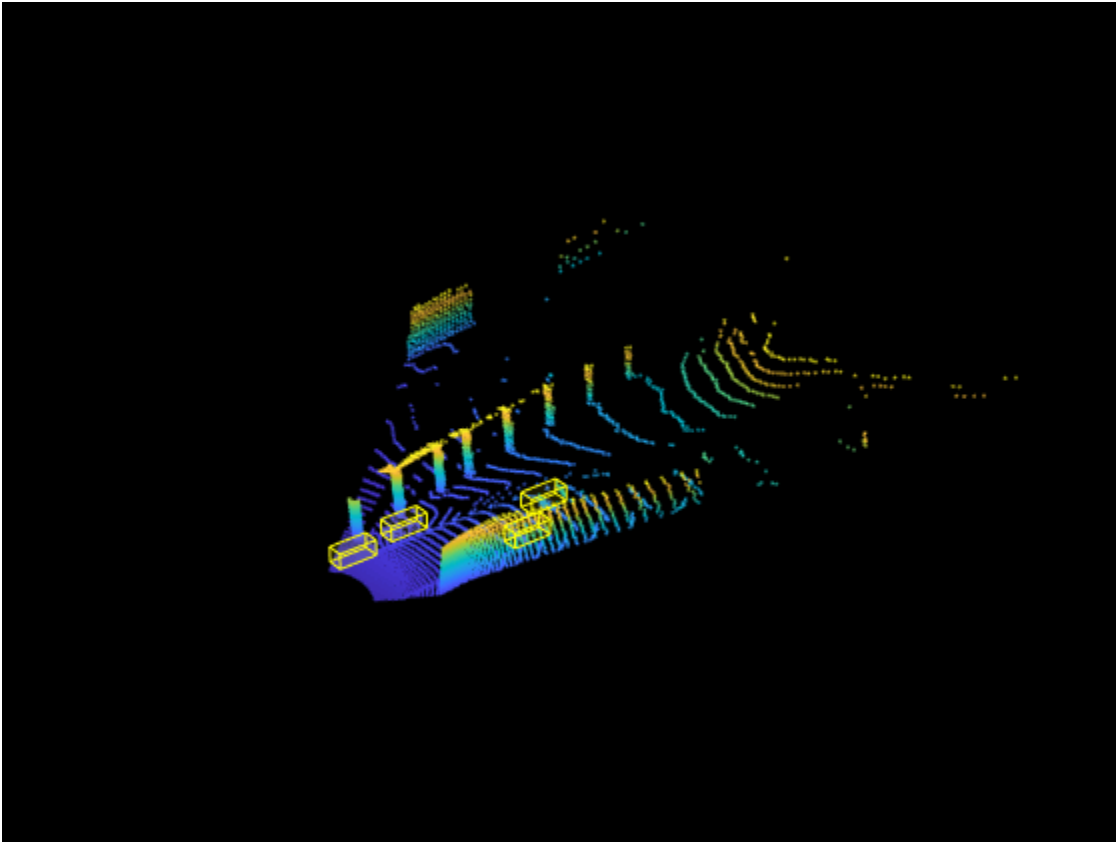
Read a point cloud from the training data.

```
pcBoxLabels = read(trainingData);
figure
pcshow(pcBoxLabels{1,1}.Location)
showShape(cuboid=pcBoxLabels{1,2},Opacity=0.1, ...
        Color="yellow",LineWidth=0.5);
title("Original Point Cloud")
```

Augment the point cloud data `pcBoxLabels` with points sampled from the datastore `cdsSampled` using the `pcBboxOversample` function.

```
totalObjects = 5;
augmentedPcBoxLabels = pcBboxOversample(pcBoxLabels,cdsSampled,classNames,totalObjects);
figure
pcshow(augmentedPcBoxLabels{1,1}.Location)
showShape(cuboid=augmentedPcBoxLabels{1,2},Opacity=0.1, ...
        Color="yellow",LineWidth=0.5);
title("Augmented Point Cloud")
```

Input Arguments

`pcBoxLabels` — Input point cloud data
1-by-3 cell array

Input point cloud data, specified as a 1-by-3 cell array. The cells contain the point cloud, the bounding box annotations and the bounding box categories, respectively.

`sampleData` — Training Data
valid datastore object | table

Training data, specified as a valid datastore object or table.

- If you specify a datastore object, your data must be set up such that using the read function on the datastore object returns a cell array or table with three columns. Each row corresponds to an object, and the columns must follow this format.

  - First column — Sampled points, specified as a cell array.
  - Second column — Bounding box, specified as a cell array containing a 9-element vector of the form [*x y z length width height roll pitch yaw*], representing the location and dimensions of the bounding box for the sampled points.
  - Third column — Label, specified as a cell array containing a categorical vector with the object class name.

You can use the `combine` function to combine two or more datastores. For more information on creating datastore objects, see the `datastore` function.

- If you use a table, the table must have two or more columns. The first column must contain point cloud file names with the file location. The point cloud files can be in any format supported by `pcread` function. Each of the remaining columns represent a single object class such as *Car*, or *Truck* containing a 9-element vector of the form [*x y z length width height roll pitch yaw*], specifying the location and dimensions of the bounding box corresponding to the sampled points in the point cloud.

`classNames` — Names of object classes
M-element vector of strings | *M*-element categorical vector | *M*-element cell array of character vectors

Names of object classes, specified as a *M*-element vector of strings, *M*-element categorical vector, *M*-element cell array of character vectors. *M* is the number of object classes specified.

`totalObjects` — Total objects in each class
scalar | *M*-element vector

Total number of objects in each class of the output point cloud, specified as a positive scalar or *M*-element vector. When this value is a scalar, the function uses the same value for all classes. When you specify an *M*-element vector, each element specifies the number of objects of the corresponding class in the `classNames` argument.

Output Arguments

`augmentedPcBoxLabels` — Augmented point cloud data
1-by-3 cell array

Augmented point cloud data, returned as a 1-by-3 cell array. The cells contain the augmented point cloud, the bounding box annotations, and the box categories, respectively.

Algorithms

Lidar object detection techniques directly predict 3-D bounding boxes around objects of interest. Data augmentation helps you improve prediction accuracy and avoid overfitting issues while training.

You can perform ground truth data augmentation on point clouds using these steps.

1 Sample 3-D bounding boxes and the corresponding points from input training data using the `sampleLidarData` function.
2 Augment a point cloud randomly with the sampled bounding boxes by using the `pcBboxOversample` function. The function performs a collision test on the sampled boxes and the ground truth boxes of the input point cloud to avoid overlap.

This technique alleviates the class imbalance problem in lidar object detection.

Version History
Introduced in R2022a

See Also

Apps
Lidar Labeler | **Lidar Viewer**

Functions
sampleLidarData | transform | pointPillarsObjectDetector | pointnetplusLayers |
squeezesegv2Layers

Topics
"Data Augmentations for Lidar Object Detection Using Deep Learning"
"Lidar 3-D Object Detection Using PointPillars Deep Learning"

lidarObjectDetectorTrainingData

Create training data for lidar object detection

Syntax

```
trainingData = lidarObjectDetectorTrainingData(gTruth)
[ptds,blds] = lidarObjectDetectorTrainingData(gTruth)
___ = lidarObjectDetectorTrainingData(gTruth,Name=Value)
```

Description

`trainingData = lidarObjectDetectorTrainingData(gTruth)` creates a table of training data from the specified ground truth label data. Use this training data to train the deep learning networks in Lidar Toolbox for lidar object detection.

`[ptds,blds] = lidarObjectDetectorTrainingData(gTruth)` creates a file datastore and a box label datastore training data from the specified ground truth label data. To create a datastore for training the network, combine the file and box label datastores by using `combine(ptds, blds)`. Use the combined datastore to train the deep learning networks in Lidar Toolbox for lidar object detection.

`___ = lidarObjectDetectorTrainingData(gTruth,Name=Value)` uses additional options specified by one or more name-value arguments.

Examples

Generate Training Data for Point Cloud Object Detection

This example shows how to generate training data to train a deep learning network for point cloud object detection.

Step 1: Create Ground Truth from Data Source

Specify the name of the file containing the point cloud data. The input file is a Velodyne® packet capture (PCAP) file.

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

Specify the parameters for loading the point cloud sequence from the data source.

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

Load the point cloud data from the specified source file by using the `vision.labeler.loading.VelodyneLidarSource` function.

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

Define class labels to specify the names of the objects in the input point cloud.

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

Define bounding boxes to specify the location of each object in the point cloud sequence, at each timestamp. Store information about bounding boxes and timestamp to a table.

```
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 ground truth object.

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

Step 2: Generate Training Data

Create point cloud and box label datastores from the labeled ground truth by using the `lidarObjectDetectorTrainingData` function.

```
[pcds,bxds] = lidarObjectDetectorTrainingData(gTruth);
```

```
Write point cloud extracted for training to folder:
    C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\2\tpbb30a1f5\lidar-ex45787688
```

```
Writing 1 point clouds extracted from dataSource1...Completed.
```

Generate training data by combining the point cloud and box label datastores.

```
trainingData = combine(pcds,bxds);
```

Step 3: Configure Object Detector

Specify the class names, anchor boxes, point cloud range, and the voxel size. Configure the PointPillars object detector for training and inference.

```
classNames = "Car";
anchorBoxes = {[1.9,4.5,1.7,-1.78,0; 1.9,4.5,1.7,-1.78,1.57]};
pcRange = [0,69.12,-39.68,39.68,-5,5];
voxSize = [0.16,0.16];
detector = pointPillarsObjectDetector(pcRange,classNames,anchorBoxes,...
    VoxelSize=voxSize);
```

Step 4: Train Object Detector

Specify training options.

```
options = trainingOptions("adam",...
    Plots="none",...
    MaxEpochs=2,...
    MiniBatchSize=1,...
    GradientDecayFactor=0.9,...
```

```
        SquaredGradientDecayFactor=0.999,...
        InitialLearnRate=0.0002,...
        LearnRateDropPeriod=15,...
        LearnRateDropFactor=0.8,...
        ExecutionEnvironment="cpu",...
        DispatchInBackground=false,...
        BatchNormalizationStatistics="moving",...
        ResetInputNormalization=false);
```

Train the PointPillars object detector to detect classes specified in the input training data. You can use the trained detector to detect objects in a test point cloud by using the `detect` function.

```
[detector,info] = trainPointPillarsObjectDetector(trainingData,detector,options);

*******************************************************************************
Processing data in minibatchqueue....

*******************************************************************************
Data processing complete.

*******************************************************************************
Training a PointPillars Object Detector for the following object classes:

* Car


    Epoch      Iteration      TimeElapsed      LearnRate      TrainingLoss
    _____      _____      _____      _____      _____



*******************************************************************************
Detector training complete.
*******************************************************************************
```

Input Arguments

gTruth — Lidar ground truth label data
groundTruthLidar object

Lidar ground truth label data, specified as a `groundTruthLidar` object or an array of `groundTruthLidar` objects. To create ground truth objects from existing ground truth data, use the `groundTruthLidar` object. You can also use the **Lidar Labeler** app to label a point cloud and generate the ground truth data.

Note The `lidarObjectDetectorTrainingData` function imports only the ground truth data with cuboid ROI labels. Ground truth data with other label types are ignored.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `trainingData = lidarObjectDetectorTrainingData(gTruth, PointCloudFormat='ply')` writes the extracted point clouds to `ply` format.

SamplingFactor — Factor for subsampling point clouds
auto (default) | positive integer | vector of positive integers

Factor for subsampling point clouds in the ground truth data source, specified as one of these values:

- `"auto"` — If the input is a `groundTruthLidar` object or an array of `groundTruthLidar` objects. The function samples data sources with timestamps, such as a point cloud sequence, with a factor of 5, and 1 for a collection of point clouds. This is the default value.
- positive integer — If the input is a `groundTruthLidar` object. Uniform sampling factor is applied to all the point cloud samples in the data source.
- vector of positive integers — If the input is an array of `groundTruthLidar` objects. The k th element in the vector is applied as the sampling factor for data sources in the k th ground truth object in the array.

For a sampling factor of N, the returned training data includes every Nth point cloud sample in the ground truth data source. The function ignores ground truth samples with empty label data.

Use sampled data to reduce repeated data, such as a sequence of point clouds with the same scene and labels. It can also help in reducing training time.

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

WriteLocation — Name of folder
pwd (current working folder) (default) | string scalar | character vector

Folder name to write extracted point cloud samples to, specified as a string scalar or character vector. The specified folder must exist and have write permissions.

Use this name-value argument only if the data source in the `groundTruthLidar` object is a `VelodyneLidarSource`, `LasFileSequenceSource`, `CustomPointCloudSource`, or `RosbagSource` object. You can know this from the `DataSource` property of the `groundTruthLidar` object. For other data sources, the `lidarObjectDetectorTrainingData` function ignores this value, if specified.

Data Types: `char` | `string`

PointCloudFormat — Point cloud file format
pcd (default) | character vector

Point cloud file format, specified as a character vector. File formats must be supported by `pcwrite`. By default, the function writes the point cloud to `pcd` format.

Use this name-value argument only if the data source in the `groundTruthLidar` object is a `VelodyneLidarSource`, `LasFileSequenceSource`, `CustomPointCloudSource`, or `RosbagSource` object. You can know this from the `DataSource` property of the `groundTruthLidar` object. For other data sources, the `lidarObjectDetectorTrainingData` function ignores this value, if specified.

Data Types: `char`

NamePrefix — Prefix for output point cloud file names
string scalar | character vector

Prefix for output point cloud file names, specified as a string scalar or character vector. The point cloud files are named as:

`<source_name><source_number>_<pointcloud_number>.<pointcloud_format>`

The `NamePrefix` parameter sets the value for `<source_name>`. By default, the `<source_name>` is the name of the data source from which the point clouds are extracted. `<source_name>`

Use this name-value argument only if the data source in the `groundTruthLidar` object is a `VelodyneLidarSource`, `LasFileSequenceSource`, `CustomPointCloudSource`, or `RosbagSource` object. You can know this from the `DataSource` property of the `groundTruthLidar` object. For other data sources, the `lidarObjectDetectorTrainingData` function ignores this value, if specified.

Data Types: `char` | `string`

Verbose — Flag to display writing progress
`true` or 1 (default) | `false` or 0

Flag to display writing progress in the MATLAB command window, specified as one of these values:

- `true` or 1 — Displays information about the write progress.
- `false` or 0 — Does not display information about the write progress.

Use this name-value argument only if the data source in the `groundTruthLidar` object is a `VelodyneLidarSource`, `LasFileSequenceSource`, `CustomPointCloudSource`, or `RosbagSource` object. You can know this from the `DataSource` property of the `groundTruthLidar` object. For other data sources, the `lidarObjectDetectorTrainingData` function ignores this value, if specified.

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

Output Arguments

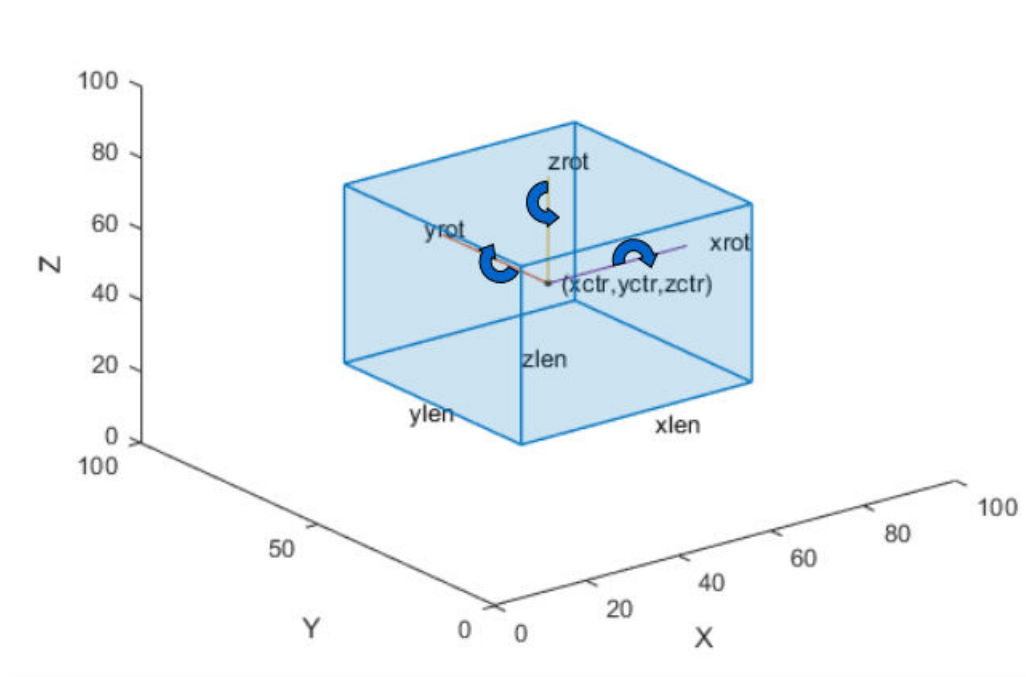**`trainingData` — Labeled data for training the network**
table

Labeled data for training the network, returned as a table with two or more columns. The first column of the table contains point cloud file names with paths. Each of the remaining columns correspond to a cuboid ROI label and contains the locations of bounding boxes in the point cloud sample (specified in the first column), for that label. The bounding boxes are specified as a

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

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

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

Data Types: table

ptds — Extracted point cloud data
fileDatastore object

Extracted point cloud data, returned as a `fileDatastore` object. The point cloud data must contain at least one class label. The function ignores unlabelled point cloud data.

blds — Extracted ROI labels
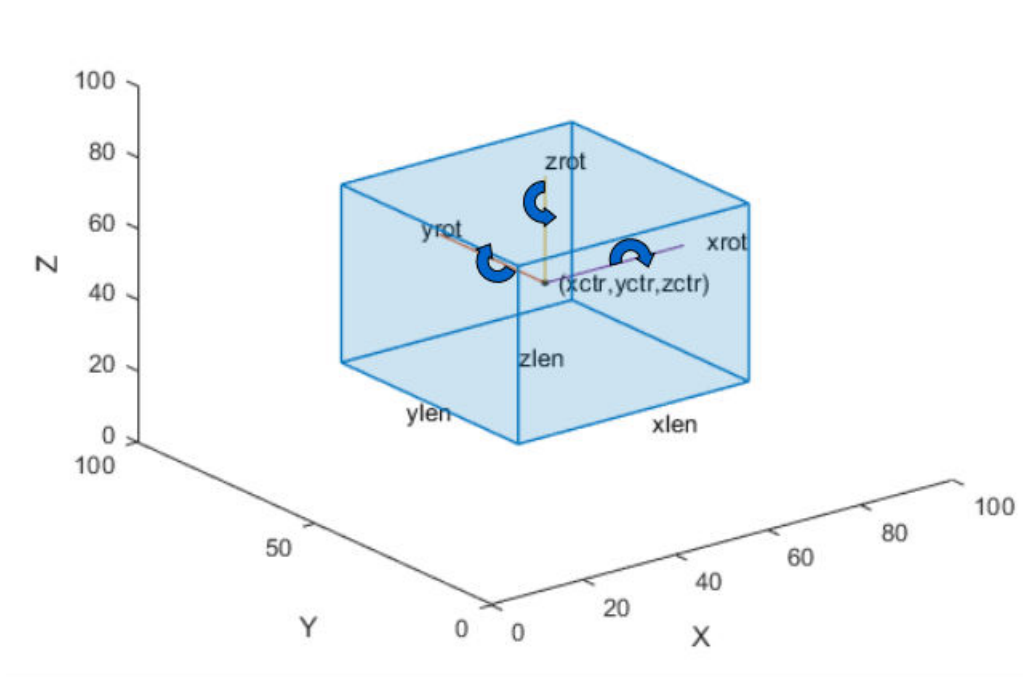boxlabelDatastore object

Extracted ROI labels, returned as a `boxLabelDatastore` object. The datastore contains M-by-9 matrices of M bounding boxes and categorical vectors of cuboid ROI label names.

The bounding boxes are specified as a

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

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

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

Version History
Introduced in R2022a

See Also
groundTruthLidar | trainPointPillarsObjectDetector | boxLabelDatastore | fileDatastore