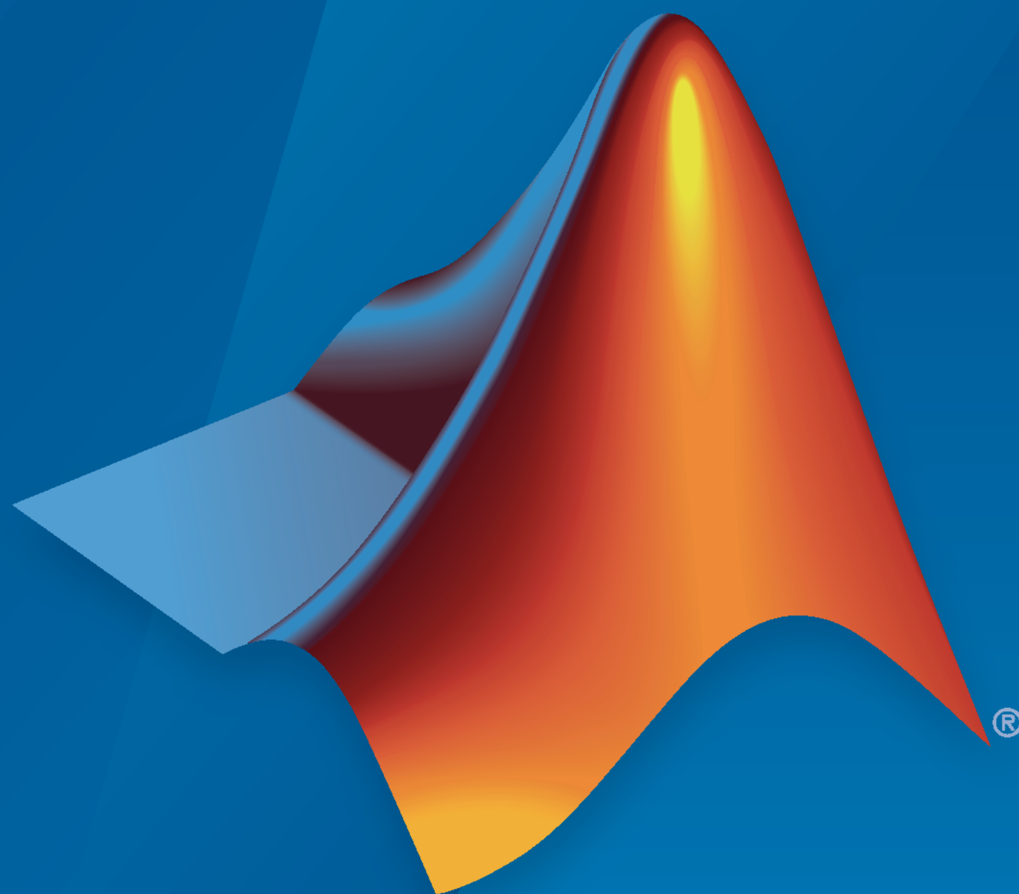
Lidar Toolbox™

User's Guide



MATLAB®

R2020b

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

Lidar Toolbox™ User's Guide

© COPYRIGHT 2020 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

September 2020 Online only New for Version 1.0 (R2020b)

Contents

1 Lidar Toolbox Featured Examples

Lidar and Camera Calibration **1-2**

Lidar Point Cloud Semantic Segmentation Using PointSeg Deep Learning Network **1-12**

Detect, Classify, and Track Vehicles Using Lidar **1-23**

Feature-Based Map Building from Lidar Data **1-38**

Detect Vehicles in Lidar Using Image Labels **1-47**

Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network **1-57**

Code Generation for Lidar Point Cloud Segmentation Network **1-67**

Lidar 3-D Object Detection Using PointPillars Deep Learning **1-74**

Aerial Lidar SLAM Using FPFH Descriptors **1-88**

Collision Warning Using 2-D Lidar **1-101**

Track Vehicles Using Lidar: From Point Cloud to Track List **1-110**

2 Lidar Labeling

Get Started with the Lidar Labeler **2-2**
Load Lidar Data to Label **2-2**
Create Labels and Attributes **2-4**
Label Point Cloud Using Automation **2-7**
View and Adjust the Labels **2-8**
Export the Labels **2-9**

Keyboard Shortcuts and Mouse Actions for Lidar Labeler **2-10**
Label Definitions **2-10**
Frame Navigation and Time Interval Settings **2-10**
Labeling Window **2-10**
Cuboid Resizing and Moving **2-11**
Zooming, Panning, and Rotating **2-12**

App Sessions 2-12

3 Concept Pages

Lidar Processing Overview **3-2**
Introduction 3-2
Point Cloud 3-2

Coordinate Systems in Lidar Toolbox **3-4**
World Coordinate System 3-4
Sensor Coordinate System 3-4
Spatial Coordinate System 3-5
Pattern Coordinate System 3-5

What Is Lidar Camera Calibration? **3-7**
Extrinsic Calibration of Lidar and Camera 3-7

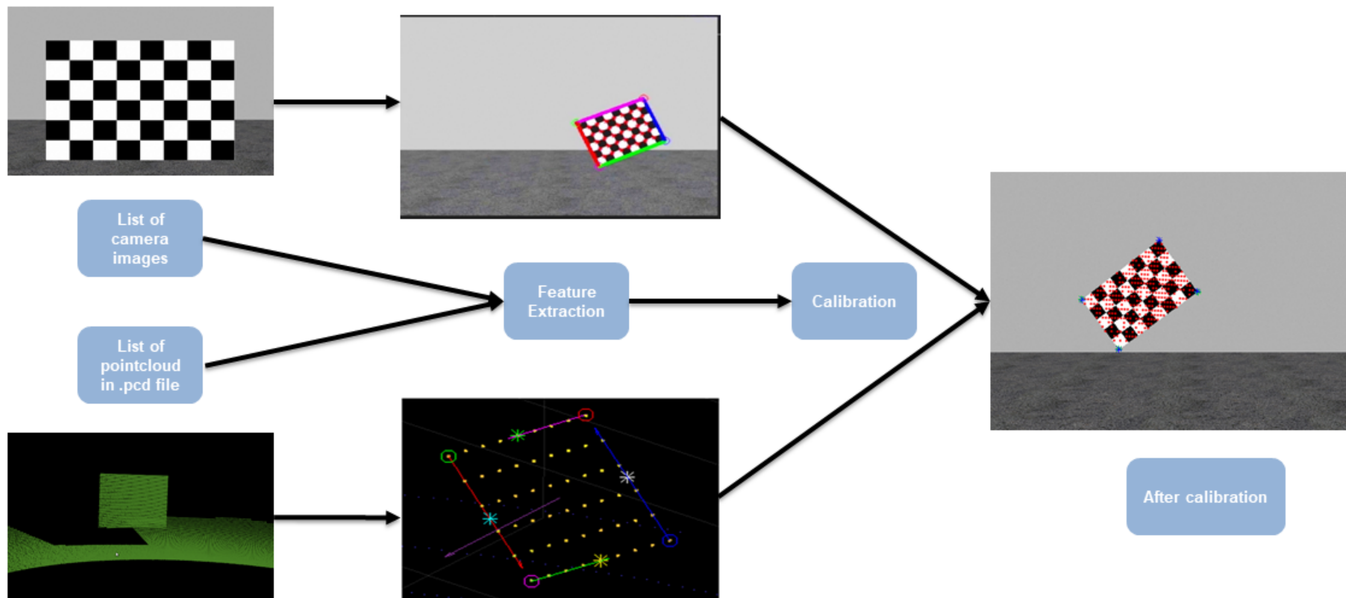
Lidar Toolbox Featured Examples

- “Lidar and Camera Calibration” on page 1-2
- “Lidar Point Cloud Semantic Segmentation Using PointSeg Deep Learning Network” on page 1-12
- “Detect, Classify, and Track Vehicles Using Lidar” on page 1-23
- “Feature-Based Map Building from Lidar Data” on page 1-38
- “Detect Vehicles in Lidar Using Image Labels” on page 1-47
- “Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network” on page 1-57
- “Code Generation for Lidar Point Cloud Segmentation Network” on page 1-67
- “Lidar 3-D Object Detection Using PointPillars Deep Learning” on page 1-74
- “Aerial Lidar SLAM Using FPFH Descriptors” on page 1-88
- “Collision Warning Using 2-D Lidar” on page 1-101
- “Track Vehicles Using Lidar: From Point Cloud to Track List” on page 1-110

Lidar and Camera Calibration

This example shows you how to estimate the rigid transformation between a 3-D lidar and a camera. At the end of this example, you will be able to use the rigid transformation matrix to fuse lidar and camera data.

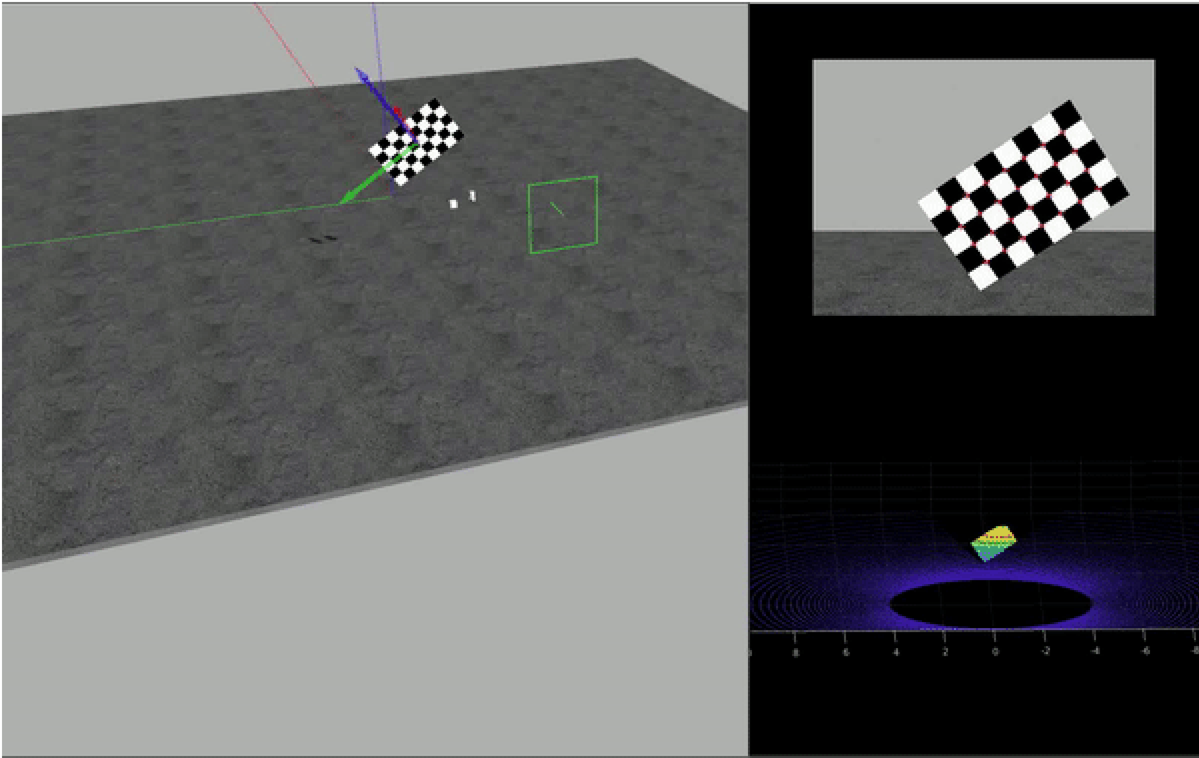
This diagram explains the workflow for the lidar and camera calibration (LCC) process.



Overview

Lidar and camera sensors are the most common vision sensors in autonomous driving applications. Cameras provide rich color information and other features that can be used to extract different characteristics of the detected objects. Lidar sensors, on the other hand, provide an accurate 3-D location and structure of the objects. To enhance the object detection and classification pipeline, data from these two sensors can be fused together to get more detailed and accurate information on the objects.

The transformation matrix in the form of orientation and relative positions between the two sensors is the precursor to fusing data from these two sensors. Lidar camera calibration helps in estimating the transformation matrix between 3-D lidar and a camera mounted on the autonomous vehicle. In this example, you will use data from two different lidar sensors, `HDL64` and `VLP16`. `HDL64` data is collected from a Gazebo environment as shown in this figure.



Data is captured in the form of set of PNG images and corresponding PCD point clouds. This example assumes that the camera's intrinsic parameters are known. For more information on extracting a camera's intrinsic parameters, see Single Camera Calibration.

Load Data

Load Velodyne HDL-64 sensor data from Gazebo.

```
imagePath = fullfile(toolboxdir('lidar'), 'lidardata', 'lcc', 'HDL64', 'images');
ptCloudPath = fullfile(toolboxdir('lidar'), 'lidardata', 'lcc', 'HDL64', 'pointCloud');
cameraParamsPath = fullfile(imagePath, 'calibration.mat');

intrinsic = load(cameraParamsPath); % Load camera intrinsics
imds = imageDatastore(imagePath); % Load images using imageDatastore
pcds = fileDatastore(ptCloudPath, 'ReadFcn', @pcread); % Load point cloud files

imageFileNames = imds.Files;
ptCloudFileNames = pcds.Files;

squareSize = 200; % Square size of the checkerboard

% Set random seed to generate reproducible results.
rng('default');
```

Checkerboard Corner Detection

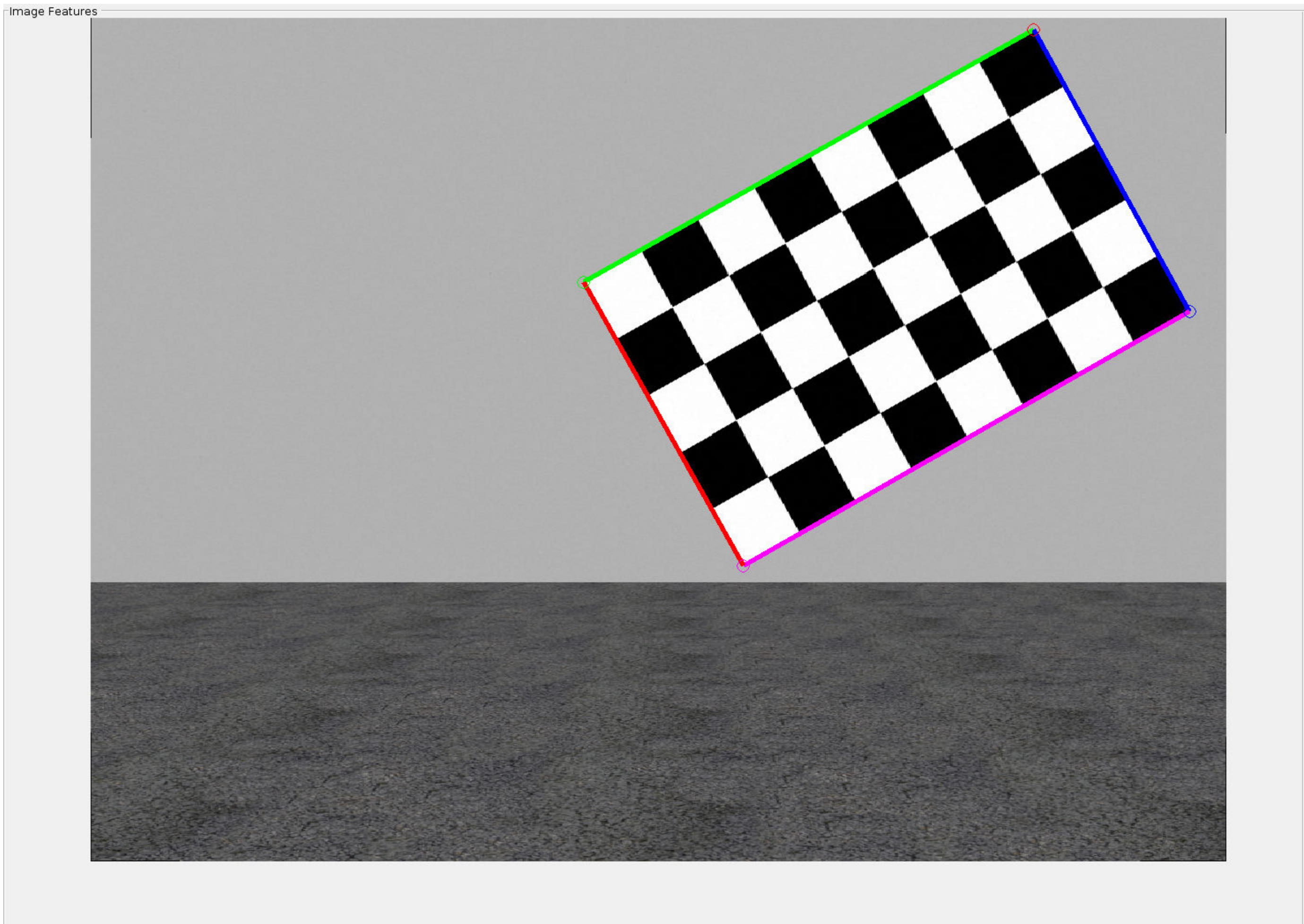
This example uses a checkerboard pattern as the feature of comparison. Checkerboard edges are estimated using lidar and camera sensors. Use `estimateCheckerboardCorners3d` to calculate coordinates of the checkerboard corners and size of actual checkerboard in *mm*. The corners are

estimated in 3-D with respect to the camera's coordinate system. For more details on camera coordinate systems, see“Coordinate Systems in Lidar Toolbox” on page 3-4

```
[imageCorners3d, checkerboardDimension, dataUsed] = ...
    estimateCheckerboardCorners3d(imageFileNames, intrinsic.cameraParams, squareSize);
imageFileNames = imageFileNames(dataUsed); % Remove image files that are not used
```

The results can be visualized using the helper function `helperShowImageCorners`.

```
% Display Checkerboard corners
helperShowImageCorners(imageCorners3d, imageFileNames, intrinsic.cameraParams);
```



Checkerboard Detection in Lidar

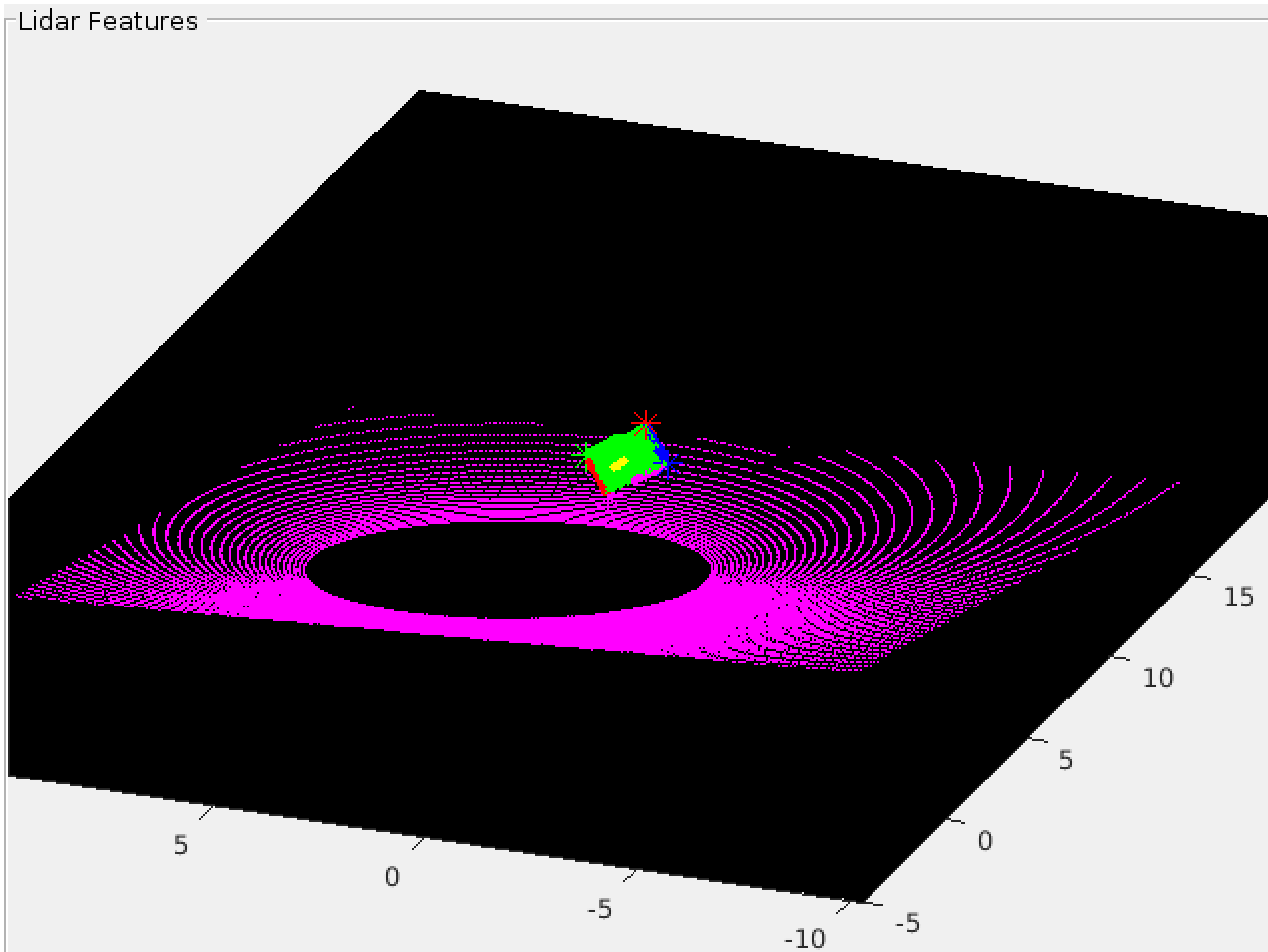
Similarly, use `detectRectangularPlanePoints` function to detect a checkerboard in the lidar data. The function detects rectangular objects in the point cloud based on the input dimensions. In this case, it detects the checkerboard using the board dimensions calculated in the previous section.

```
% Extract ROI from the detected image corners
roi = helperComputeROI(imageCorners3d, 5);

% Filter point cloud files corresponding to the detected images
ptCloudFileNames = ptCloudFileNames(dataUsed);
```

```
[lidarCheckerboardPlanes, framesUsed, indices] = ...
    detectRectangularPlanePoints(ptCloudFileNames, checkerboardDimension, 'ROI', roi);

% Remove ptCloud files that are not used
ptCloudFileNames = ptCloudFileNames(framesUsed);
% Remove image files
imageFileNames = imageFileNames(framesUsed);
% Remove 3D corners from images
imageCorners3d = imageCorners3d(:, :, framesUsed);
```

To visualize the detected checkerboard, use `helperShowLidarCorners` function.

```
helperShowLidarCorners(ptCloudFileNames, indices);
```



Calibrating Lidar and Camera

Use `estimateLidarCameraTransform to` estimate the rigid transformation matrix between lidar and camera.

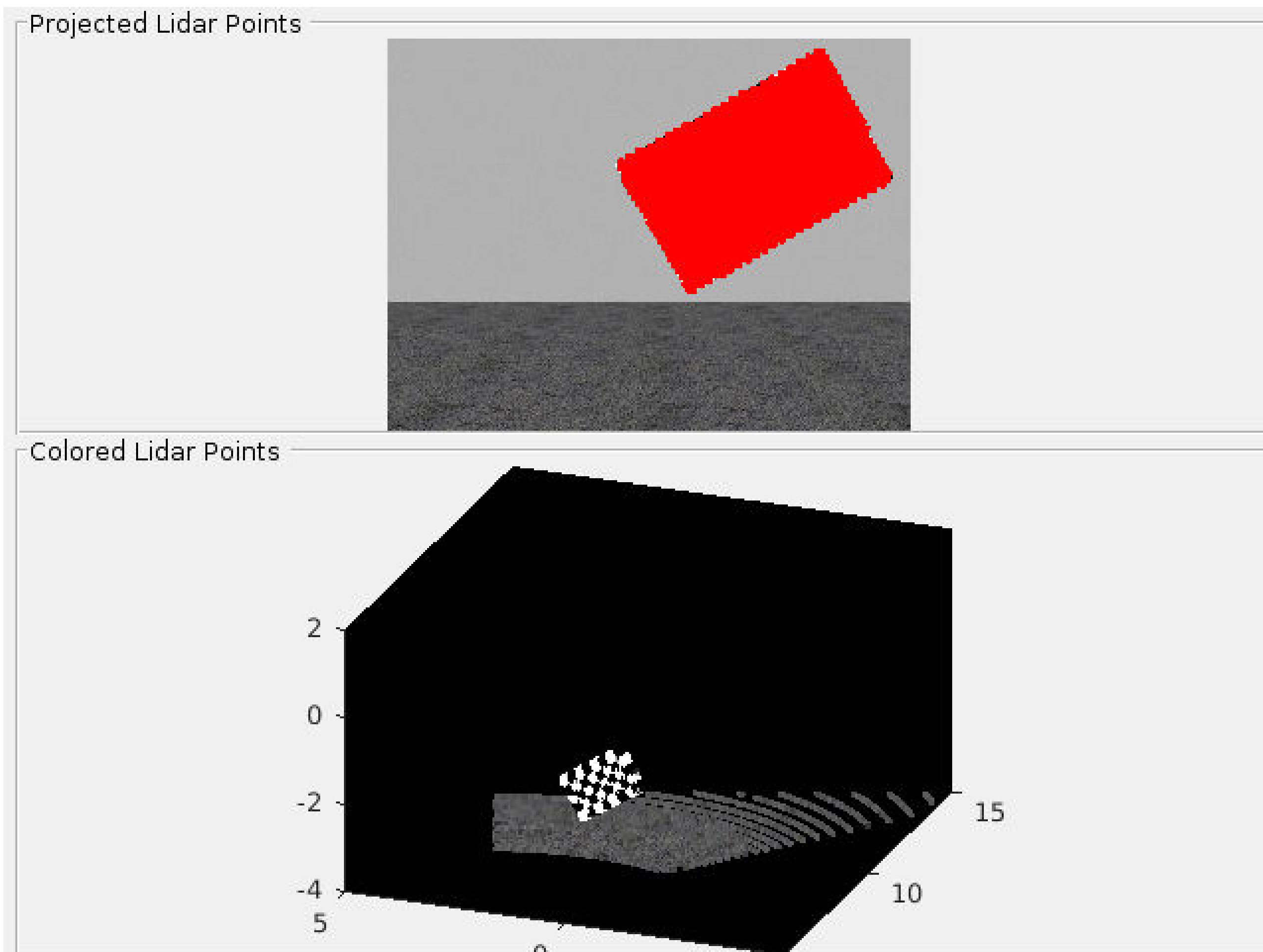
```
[tform, errors] = estimateLidarCameraTransform(lidarCheckerboardPlanes, ...
    imageCorners3d, 'CameraIntrinsic', intrinsic.cameraParams);
```

After calibration is completed, you can use calibration matrix in two ways:

- Project Lidar point cloud on image.
- Enhance Lidar point cloud using color infromation from image.

Use `helperFuseLidarCamera function` to visualize the lidar and the image data fused together.

```
helperFuseLidarCamera(imageFileNames, ptCloudFileNames, indices, ...
    intrinsic.cameraParams, tform);
```



Error Visualization

Three types of errors are defined to estimate the accuracy of the calibration:

- Translation Error: Mean of difference between centroid of checkerboard corners in the lidar and the projected corners in 3-D from an image.
- Rotation Error: Mean of difference between the normals of checkerboard in the point cloud and the projected corners in 3-D from an image.
- Reprojection Error: Mean of difference between the centroid of image corners and projected lidar corners on the image.

Plot the estimated error values by using helperShowError.

```
helperShowError(errors)
```

Results

After calibration check for data with high calibration errors and re-run the calibration.

```
outlierIdx = errors.RotationError < mean(errors.RotationError);
[newTform, newErrors] = estimateLidarCameraTransform(lidarCheckerboardPlanes(outlierIdx), ...
        imageCorners3d(:, :, outlierIdx), 'CameraIntrinsic', intrinsic.cameraParams);
    helperShowError(newErrors);
```

Testing on Real Data

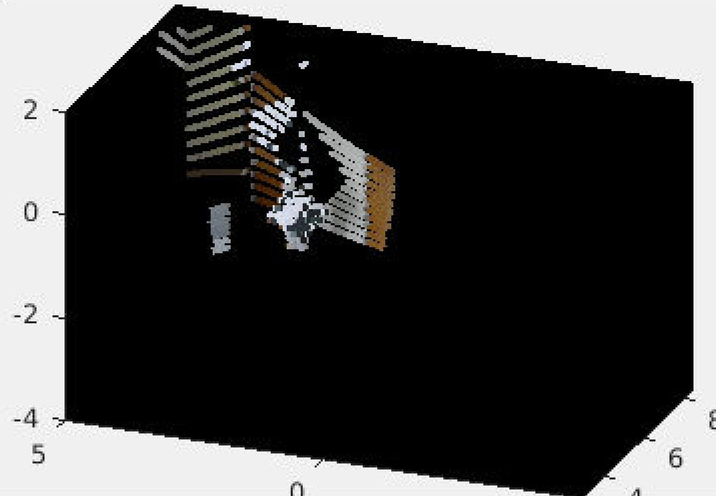
Test the LCC workflow on actual VLP-16 Lidar data to evaluate its performance.

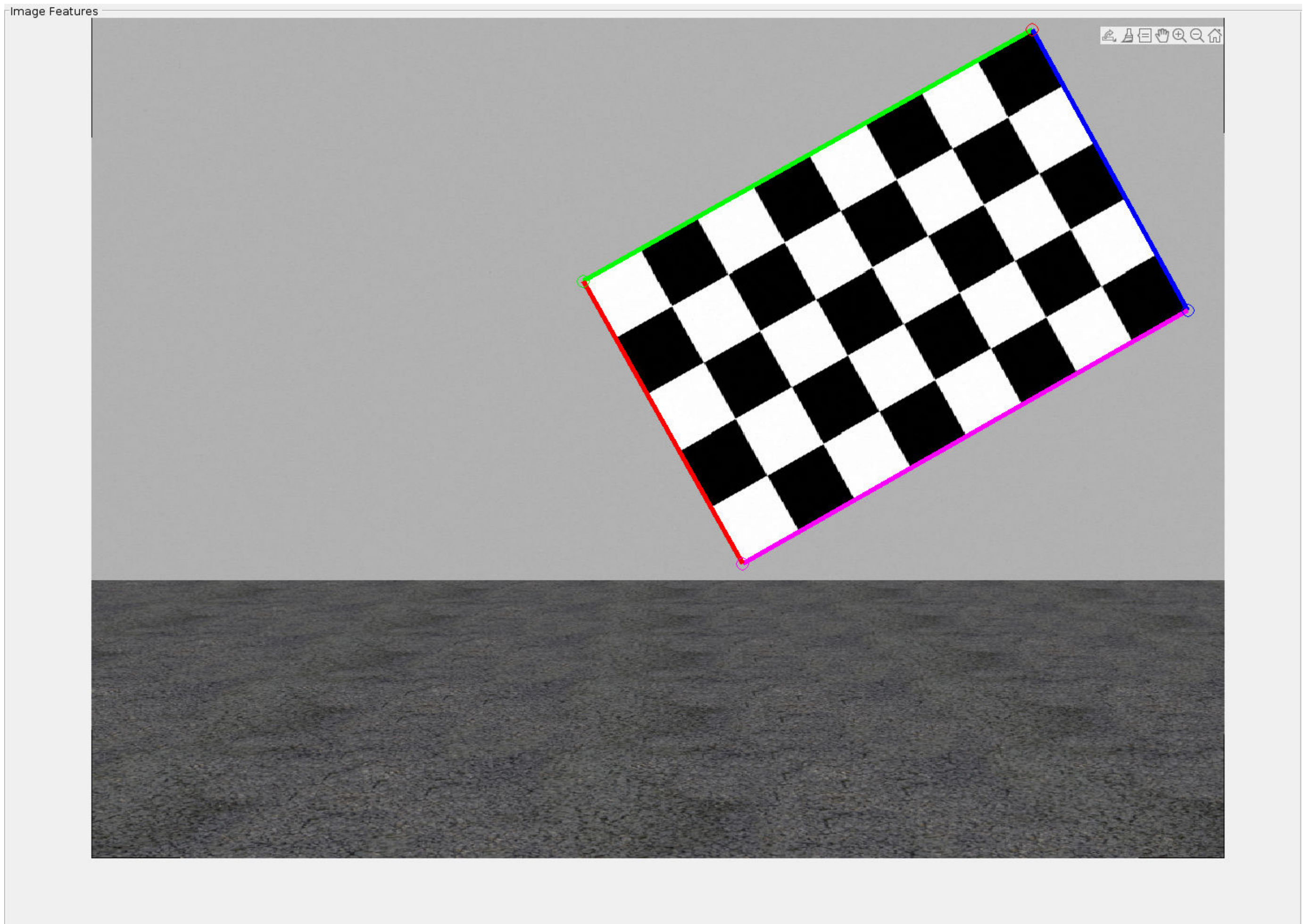
```
clear;
imagePath = fullfile(toolboxdir('lidar'), 'lidardata', 'lcc', 'vlp16', 'images');
ptCloudPath = fullfile(toolboxdir('lidar'), 'lidardata', 'lcc', 'vlp16', 'pointCloud');
cameraParamsPath = fullfile(imagePath, 'calibration.mat');

intrinsic = load(cameraParamsPath); % Load camera intrinscs
imds = imageDatastore(imagePath); % Load images using imageDatastore
pcds = fileDatastore(ptCloudPath, 'ReadFcn', @pcread); % Loadr point cloud files

imageFileNames = imds.Files;
ptCloudFileNames = pcds.Files;

squareSize = 81; % Square size of the checkerboard

% Set random seed to generate reproducible results.
rng('default');

% Extract Checkerboard corners from the images
[imageCorners3d, checkerboardDimension, dataUsed] = ...
    estimateCheckerboardCorners3d(imageFileNames, intrinsic.cameraParams, squareSize);

imageFileNames = imageFileNames(dataUsed); % Remove image files that are not used

% Filter point cloud files corresponding to the detected images
ptCloudFileNames = ptCloudFileNames(dataUsed);
```

```
% Extract ROI from the detected image corners
roi = helperComputeROI(imageCorners3d, 5);

%Extract Checkerboard in lidar data
[lidarCheckerboardPlanes, framesUsed, indices] = detectRectangularPlanePoints(...
    ptCloudFileNames, checkerboardDimension, 'RemoveGround', true, 'ROI', roi);
imageCorners3d = imageCorners3d(:, :, framesUsed);
% Remove ptCloud files that are not used
ptCloudFileNames = ptCloudFileNames(framesUsed);
% Remove image files
imageFileNames = imageFileNames(framesUsed);
[tform, errors] = estimateLidarCameraTransform(lidarCheckerboardPlanes, ...
    imageCorners3d, 'CameraIntrinsic', intrinsic.cameraParams);
helperFuseLidarCamera(imageFileNames, ptCloudFileNames, indices,...
    intrinsic.cameraParams, tform);
```

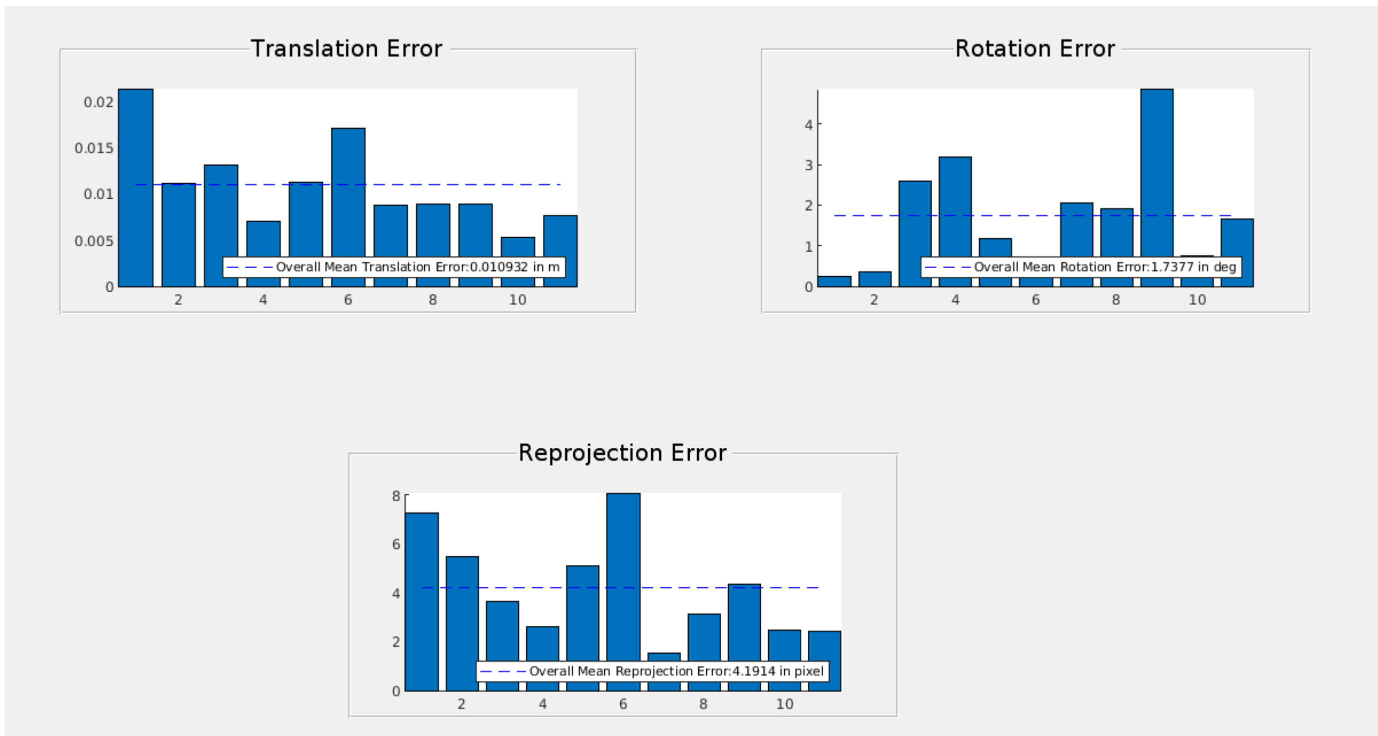
Projected Lidar Points

Colored Lidar Points






```
% Plot the estimated error values
helperShowError(errors);
```



Summary

This example gives an overview on how to get started with the lidar camera calibration workflow, to extract a rigid transformation between the two sensors. This example also shows you how to use the rigid transformation matrix to fuse lidar and camera data.

References

[1] Lipu Zhou and Zimo Li and Michael Kaess, "Automatic Extrinsic Calibration of a Camera and a 3D LiDAR using Line and Plane Correspondences", "IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems, IROS", Oct, 2018.

[2] K. S. Arun, T. S. Huang, and S. D. Blostein, “Least-squares fitting of two 3-D point sets,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-9, no. 5, pp. 698–700, Sept 1987.

Lidar Point Cloud Semantic Segmentation Using PointSeg Deep Learning Network

This example shows how to train a PointSeg semantic segmentation network on 3-D organized lidar point cloud data.

PointSeg [1 on page 1-0] is a convolutional neural network (CNN) for performing end-to-end semantic segmentation of road objects based on an organized lidar point cloud. By using methods such as atrous spatial pyramid pooling (ASPP) and squeeze-and-excitation blocks, the network provides improved segmentation results. The training procedure shown in this example requires 2-D spherical projected images as inputs to the deep learning network.

This example uses a highway scene data set collected using an Ouster OS1 sensor. It contains organized lidar point cloud scans of highway scenes and corresponding ground truth labels for car and truck objects. The size of the data file is approximately 760 MB.

Download Lidar Data Set

Execute this code to download the highway scene data set. The data set contains 1617 point clouds stored as `pointCloud` objects in a cell array. Corresponding ground truth data, which is attached to the example, contains bounding box information of cars and trucks in each point cloud.

```
url = 'https://www.mathworks.com/supportfiles/lidar/data/WPI_LidarData.tar.gz';

outputFolder = fullfile(tempdir,'WPI');
lidarDataTarFile = fullfile(outputFolder,'WPI_LidarData.tar.gz');

if ~exist(lidarDataTarFile, 'file')
    mkdir(outputFolder);

    disp('Downloading WPI Lidar driving data (760 MB)...');
    websave(lidarDataTarFile, url);
    untar(lidarDataTarFile,outputFolder);
end

% Check if tar.gz file is downloaded, but not uncompressed.
if ~exist(fullfile(outputFolder, 'WPI_LidarData.mat'), 'file')
    untar(lidarDataTarFile,outputFolder);
end
lidarData = load(fullfile(outputFolder, 'WPI_LidarData.mat'));

groundTruthData = load('WPI_LidarGroundTruth.mat');
```

Note: Depending on your Internet connection, the download process can take some time. The code suspends MATLAB® execution until the download process is complete. Alternatively, you can download the data set to your local disk using your web browser, and then extract `WPI_LidarData`. To use the file you downloaded from the web, change the `outputFolder` variable in the code to the location of the downloaded file.

Download Pretrained Network

Download the pretrained network to avoid having to wait for training to complete. If you want to train the network, set the `doTraining` variable to true.

```
doTraining = false;
if ~doTraining && ~exist('trainedPointSegNet.mat','file')
```

```
    disp('Downloading pretrained network (14 MB)...');
    pretrainedURL = 'https://www.mathworks.com/supportfiles/lidar/data/trainedPointSegNet.mat';
    websave('trainedPointSegNet.mat', pretrainedURL);
end
```

```
Downloading pretrained network (14 MB)...
```

Prepare Data for Training

Load Lidar Point Clouds and Class Labels

Use the `helperGenerateTrainingData` supporting function, attached to this example, to generate training data from the lidar point clouds. The function uses point cloud and bounding box data to create five-channel input images and pixel label images. To create the pixel label images, the function selects points inside the bounding box and labels them with the bounding box class ID. Each training image is specified as a 64-by-1024-by-5 array:

- The height of each image is 64 pixels.
- The width of each image is 1024 pixels.
- Each image has 5 channels. The five channels specify the 3-D coordinates of the point cloud, intensity, and range: $r = \sqrt{x^2 + y^2 + z^2}$.

A visual representation of the training data follows.

Generate the five-channel training images and pixel label images.

```
imagesFolder = fullfile(outputFolder, 'images');
labelsFolder = fullfile(outputFolder, 'labels');

helperGenerateTrainingData(lidarData, groundTruthData, imagesFolder, labelsFolder);
```

```
Preprocessing data 100.00% complete
```

The five-channel images are saved as MAT files. Pixel labels are saved as PNG files.

Note: Processing can take some time. The code suspends MATLAB® execution until processing is complete.

Create ImageDatastore and PixelLabelDatastore

Use the imageDatastore object to extract and store the five channels of the 2-D spherical images using the helperImageMatReader supporting function, which is a custom MAT file reader. This function is attached to this example as a supporting file.

```
imds = imageDatastore(imagesFolder, ...
         'FileExtensions', '.mat', ...
         'ReadFcn', @helperImageMatReader);
```

Use the pixelLabelDatastore object to store pixel-wise labels from the label images. The object maps each pixel label to a class name. In this example, cars and trucks are the only objects of interest; all other pixels are the background. Specify these classes (car, truck, and background) and assign a unique label ID to each class.

```
classNames = [
    "background"
    "car"
    "truck"
];

numClasses = numel(classNames);

% Specify label IDs from 1 to the number of classes.
labelIDs = 1 : numClasses;

pxds = pixelLabelDatastore(labelsFolder, classNames, labelIDs);
```

Load and display one of the labeled images by overlaying it on the corresponding intensity image using the helperDisplayLidarOverlayImage function, defined in the Supporting Functions on page 1-0 section of this example.

```
imageNumber = 225;

% Point cloud (channels 1, 2, and 3 are for location, channel 4 is for intensity).
I = readimage(imds, imageNumber);

labelMap = readimage(pxds, imageNumber);
figure;
helperDisplayLidarOverlayImage(I, labelMap, classNames);
title('Ground Truth');
```

Prepare Training, Validation, and Test Sets

Use the helperPartitionLidarData supporting function, attached to this example, to split the data into training, validation, and test sets that contain 970, 216, and 431 images, respectively.

```
[imdsTrain, imdsVal, imdsTest, pxdsTrain, pxdsVal, pxdsTest] = ...
    helperPartitionLidarData(imds, pxds);
```

Use the `combine` function to combine the pixel and image datastores for the training and validation data sets.

```
trainingData = combine(imdsTrain, pxdsTrain);
validationData = combine(imdsVal, pxdsVal);
```

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

Augment the training data using the `transform` function with custom preprocessing operations specified by the `augmentData` function, defined in the Supporting Functions on page 1-0 section of this example. This function randomly flips the spherical 2-D image and associated labels in the horizontal direction. Apply data augmentation to only the training data set.

```
augmentedTrainingData = transform(trainingData, @(x) augmentData(x));
```

Balance Classes Using Class Weighting

To see the distribution of class labels in the data set, use the `countEachLabel` function.

```
tbl = countEachLabel(pxds);
tbl(:,{'Name','PixelCount','ImagePixelCount'})
```

```
ans=3×3 table
         Name          PixelCount    ImagePixelCount
    ______________    __________    _______________

    {'background'}    1.0473e+08      1.0597e+08
    {'car'       }    9.7839e+05      8.4738e+07
    {'truck'     }    2.6017e+05      1.9726e+07
```

The classes in this data set are imbalanced, which is a common issue in automotive data sets containing street scenes. The background class covers more area than the car and truck classes. If not handled correctly, this imbalance can be detrimental to the learning process because the learning is biased in favor of the dominant classes.

Use these weights to correct the class imbalance. Use the pixel label counts from the `tbl.PixelCount` property and calculate the median frequency class weights.

```
imageFreq = tbl.PixelCount ./ tbl.ImagePixelCount;
classWeights = median(imageFreq) ./ imageFreq
```

```
classWeights = 3×1

    0.0133
    1.1423
    1.0000
```

Define Network Architecture

Create a PointSeg network using the `createPointSeg` supporting function, which is attached to the example. The code returns the layer graph that you use to train the network.

```
inputSize = [64 1024 5];

lgraph = createPointSeg(inputSize, classNames, classWeights);
```

Use the `analyzeNetwork` (Deep Learning Toolbox) function to display an interactive visualization of the network architecture.

```
analyzeNetwork(lgraph)
```

Specify Training Options

Use the `rmsprop` optimization algorithm to train the network. Specify the hyperparameters for the algorithm by using the `trainingOptions` function.

```
maxEpochs = 30;
initialLearningRate= 5e-4;
miniBatchSize = 8;
l2reg = 2e-4;

options = trainingOptions('rmsprop', ...
    'InitialLearnRate', initialLearningRate, ...
    'L2Regularization', l2reg, ...
    'MaxEpochs', maxEpochs, ...
    'MiniBatchSize', miniBatchSize, ...
    'LearnRateSchedule', 'piecewise', ...
    'LearnRateDropFactor', 0.1, ...
    'LearnRateDropPeriod', 10, ...
    'ValidationData', validationData, ...
    'Plots', 'training-progress', ...
    'VerboseFrequency', 20);
```

Note: Reduce `miniBatchSize` to control memory usage when training.

Train Network

Use the `trainNetwork` (Deep Learning Toolbox) function to train a PointSeg network if `doTraining` is `true`. Otherwise, load the pretrained network.

If you train the network, you can use a CPU or a GPU. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
if doTraining
    [net, info] = trainNetwork(trainingData, lgraph, options);
else
    pretrainedNetwork = load('trainedPointSegNet.mat');
    net = pretrainedNetwork.net;
end
```

Predict Results on Test Point Cloud

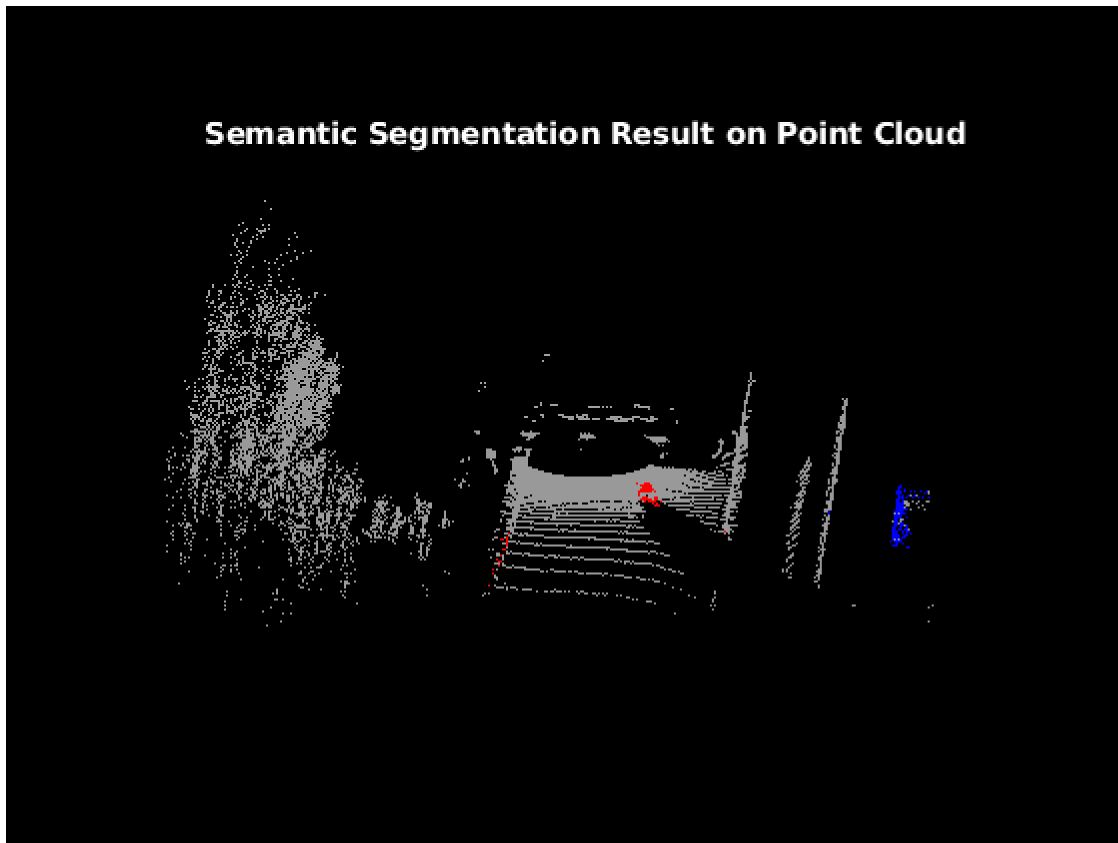
Use the trained network to predict results on a test point cloud and display the segmentation result.

First, read a PCD file and convert the point cloud to a five-channel input image. Predict the labels using the trained network. Display the figure with the segmentation as an overlay.

```
ptCloud = pcread('ousterLidarDrivingData.pcd');
I = helperPointCloudToImage(ptCloud);
predictedResult = semanticseg(I, net);

figure;
helperDisplayLidarOverlayImage(I, predictedResult, classNames);
title('Semantic Segmentation Result');
```

Use the `helperDisplayLidarOverlayPointCloud` helper function, defined in the Supporting Functions on page 1-0 section of this example, to display the segmentation result over the 3-D point cloud object `ptCloud` .

```
figure;
helperDisplayLidarOverlayPointCloud(ptCloud, predictedResult, numClasses);
view([95.71 24.14])
title('Semantic Segmentation Result on Point Cloud');
```



Evaluate Network

Run the `semanticseg` function on the entire test set to measure the accuracy of the network. Set `MiniBatchSize` to a value of 8 to reduce memory usage when segmenting images. You can increase or decrease this value depending on the amount of GPU memory you have on your system.

```
outputLocation = fullfile(tempdir, 'output');
if ~exist(outputLocation,'dir')
    mkdir(outputLocation);
end
pxdsResults = semanticseg(imdsTest, net, ...
                'MiniBatchSize', 8, ...
                'WriteLocation', outputLocation, ...
                'Verbose', false);
```

The `semanticseg` function returns the segmentation results on the test data set as a `PixelLabelDatastore` object. The function writes the actual pixel label data for each test image in the `imdsTest` object to the disk in the location specified by the `'WriteLocation'` argument.

Use the `evaluateSemanticSegmentation` function to compute the semantic segmentation metrics from the test set results.

```
metrics = evaluateSemanticSegmentation(pxdsResults, pxdsTest, 'Verbose', false);
```

You can measure the amount of overlap per class using the intersection-over-union (IoU) metric.

The `evaluateSemanticSegmentation` function returns metrics for the entire data set, for individual classes, and for each test image. To see the metrics at the data set level, use the `metrics.DataSetMetrics` property.

```
metrics.DataSetMetrics
```

```
ans=1×5 table
    GlobalAccuracy    MeanAccuracy    MeanIoU    WeightedIoU    MeanBFScore
    ______________    ____________    _______    ___________    ___________

       0.99209          0.83752       0.67895      0.98685        0.91654
```

The data set metrics provide a high-level overview of network performance. To see the impact each class has on the overall performance, inspect the metrics for each class using the `metrics.ClassMetrics` property.

```
metrics.ClassMetrics
```

```
ans=3×3 table
                  Accuracy      IoU      MeanBFScore
                  ________    _______    ___________

    background    0.99466     0.99212      0.98529
    car           0.75977     0.50096      0.82682
    truck         0.75814     0.54378      0.77119
```

Although the network overall performance is good, the class metrics show that biased classes (car and truck) are not segmented as well as the classes with abundant data (background). You can improve the network performance by training the network on more labeled data containing the car and truck classes.

Supporting Functions

Function to Augment Data

The `augmentData` function randomly flips the 2-D spherical image and associated labels in the horizontal direction.

```
function out = augmentData(inp)
%augmentData Apply random horizontal flipping.

out = cell(size(inp));

% Randomly flip the five-channel image and pixel labels horizontally.
I = inp{1};
sz = size(I);
tform = randomAffine2d('XReflection',true);
rout = affineOutputView(sz,tform,'BoundsStyle','centerOutput');

out{1} = imwarp(I,tform,'OutputView',rout);
out{2} = imwarp(inp{2},tform,'OutputView',rout);
end
```

Function to Display Lidar Segmentation Map Overlaid on 2-D Spherical Image

The `helperDisplayLidarOverlayImage` function overlays the semantic segmentation map over the intensity channel of the 2-D spherical image. The function also resizes the overlaid image for better visualization.

```
function helperDisplayLidarOverlayImage(lidarImage, labelMap, classNames)
%helperDisplayLidarOverlayImage Overlay labels over the intensity image.
%
%  helperDisplayLidarOverlayImage(lidarImage, labelMap, classNames)
%  displays the overlaid image. lidarImage is a five-channel lidar input.
%  labelMap contains pixel labels and classNames is an array of label
%  names.

% Read the intensity channel from the lidar image.
intensityChannel = uint8(lidarImage(:,:,4));

% Load the lidar color map.
cmap = helperLidarColorMap();

% Overlay the labels over the intensity image.
B = labeloverlay(intensityChannel,labelMap,'Colormap',cmap,'Transparency',0.4);

% Resize for better visualization.
B = imresize(B, 'Scale', [3 1], 'method', 'nearest');
imshow(B);

% Display the color bar.
helperPixelLabelColorbar(cmap, classNames);
end
```

Function To Display Lidar Segmentation Map Overlaid on 3-D Point Cloud

The `helperDisplayLidarOverPointCloud` function overlays the segmentation result over a 3-D organized point cloud.

```
function helperDisplayLidarOverlayPointCloud(ptCloud, labelMap, numClasses)
%helperDisplayLidarOverlayPointCloud Overlay labels over a point cloud object.
%
%  helperDisplayLidarOverlayPointCloud(ptCloud, labelMap, numClasses)
%  displays the overlaid pointCloud object. ptCloud is the organized
%  3-D point cloud input. labelMap contains pixel labels and numClasses
%  is the number of predicted classes.

sz = size(labelMap);

% Apply the color red to cars.
carClassCar = zeros(sz(1), sz(2), numClasses, 'uint8');
carClassCar(:,:,1) = 255*ones(sz(1), sz(2), 'uint8');

% Apply the color blue to trucks.
truckClassColor = zeros(sz(1), sz(2), numClasses, 'uint8');
truckClassColor(:,:,3) = 255*ones(sz(1), sz(2), 'uint8');

% Apply the color gray to the background.
backgroundClassColor = 153*ones(sz(1), sz(2), numClasses, 'uint8');

% Extract indices from the labels.
```

```
carIndices = labelMap == 'car';
truckIndices = labelMap == 'truck';
backgroundIndices = labelMap == 'background';

% Extract a point cloud for each class.
carPointCloud = select(ptCloud, carIndices, 'OutputSize','full');
truckPointCloud = select(ptCloud, truckIndices, 'OutputSize','full');
backgroundPointCloud = select(ptCloud, backgroundIndices, 'OutputSize','full');

% Apply colors to different classes.
carPointCloud.Color = carClassCar;
truckPointCloud.Color = truckClassColor;
backgroundPointCloud.Color = backgroundClassColor;

% Merge and add all the processed point clouds with class information.
coloredCloud = pcmerge(carPointCloud, truckPointCloud, 0.01);
coloredCloud = pcmerge(coloredCloud, backgroundPointCloud, 0.01);

% Plot the colored point cloud. Set an ROI for better visualization.
ax = pcshow(coloredCloud);
set(ax,'XLim',[-35.0 35.0],'YLim',[-32.0 32.0],'ZLim',[-3 8], ...
    'XColor','none','YColor','none','ZColor','none');
set(get(ax,'parent'), 'units','normalized');
end
```

Function to Define Lidar Colormap

The `helperLidarColorMap` function defines the colormap used by the lidar data set.

```
function cmap = helperLidarColorMap()

cmap = [
   0.00  0.00   0.00  % background
   0.98  0.00   0.00  % car
   0.00  0.00   0.98  % truck
   ];
end
```

Function to Display Pixel Label Colorbar

The `helperPixelLabelColorbar` function adds a colorbar to the current axis. The colorbar is formatted to display the class names with the color.

```
function helperPixelLabelColorbar(cmap, classNames)

colormap(gca, cmap);

% Add a colorbar to the current figure.
c = colorbar('peer', gca);

% Use class names for tick marks.
c.TickLabels = classNames;
numClasses = size(classNames, 1);

% Center tick labels.
c.Ticks = 1/(numClasses * 2):1/numClasses:1;

% Remove tick marks.
```

```
c.TickLength = 0;
end
```

References

[1] Wang, Yuan, Tianyue Shi, Peng Yun, Lei Tai, and Ming Liu. “PointSeg: Real-Time Semantic Segmentation Based on 3D LiDAR Point Cloud.” *ArXiv:1807.06288 [Cs]*, September 25, 2018. http://arxiv.org/abs/1807.06288.

Detect, Classify, and Track Vehicles Using Lidar

This example shows how to detect, classify, and track vehicles by using lidar point cloud data captured by a lidar sensor mounted on an ego vehicle. The lidar data used in this example is recorded from a highway-driving scenario. In this example, the point cloud data is segmented to determine the class of objects using the `PointSeg` network. A joint probabilistic data association (JPDA) tracker with an interactive multiple model filter is used to track the detected vehicles.

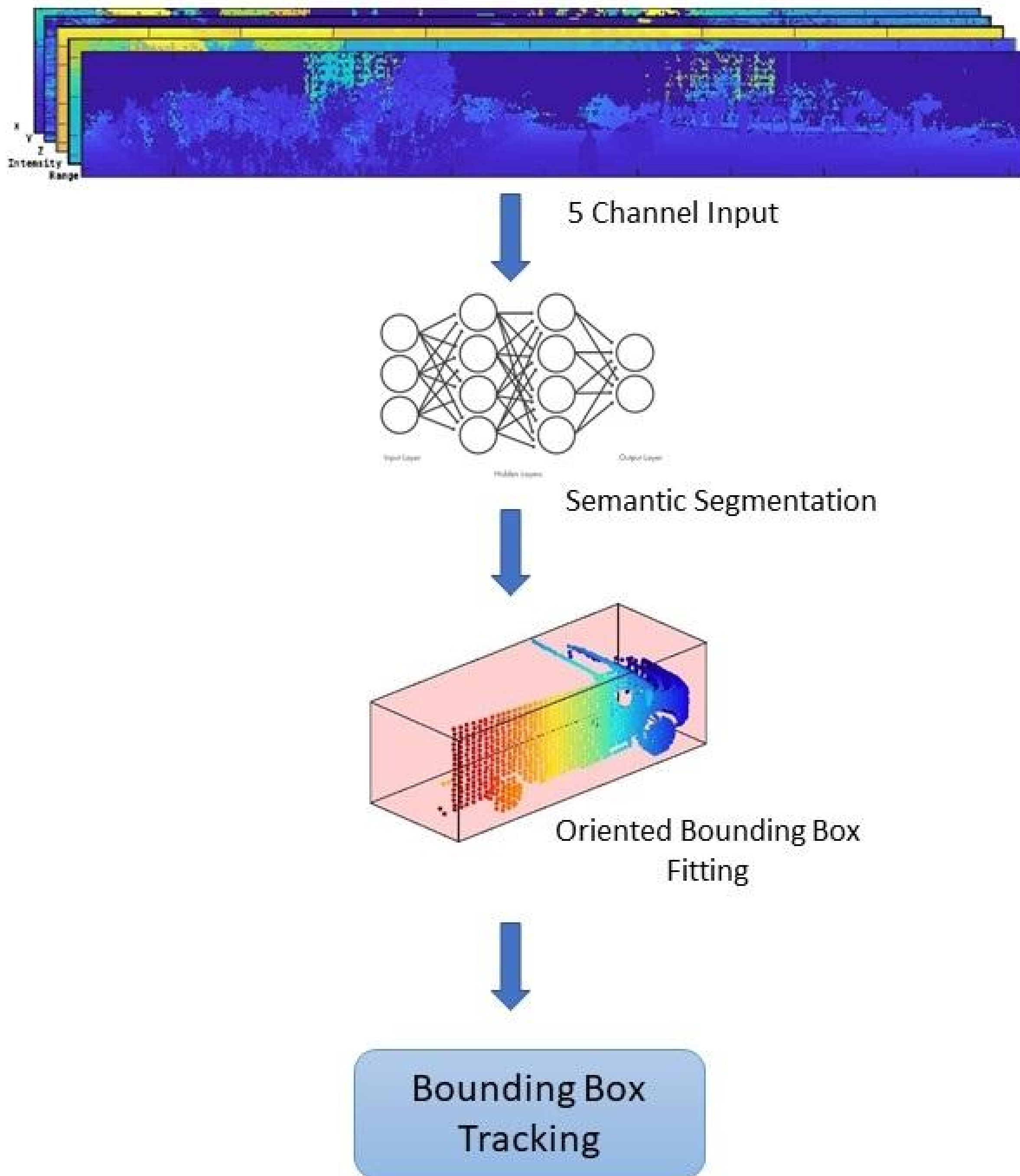
Overview

The perception module plays an important role in achieving full autonomy for vehicles with an ADAS system. Lidar and camera are essential sensors in the perception workflow. Lidar is good at extracting accurate depth information of objects, while camera produces rich and detailed information of the environment which is useful for object classification.

This example mainly includes these parts:

- Ground plane segmentation
- Semantic segmentation
- Oriented bounding box fitting
- Tracking oriented bounding boxes

The flowchart gives an overview of the whole system.



Load Data

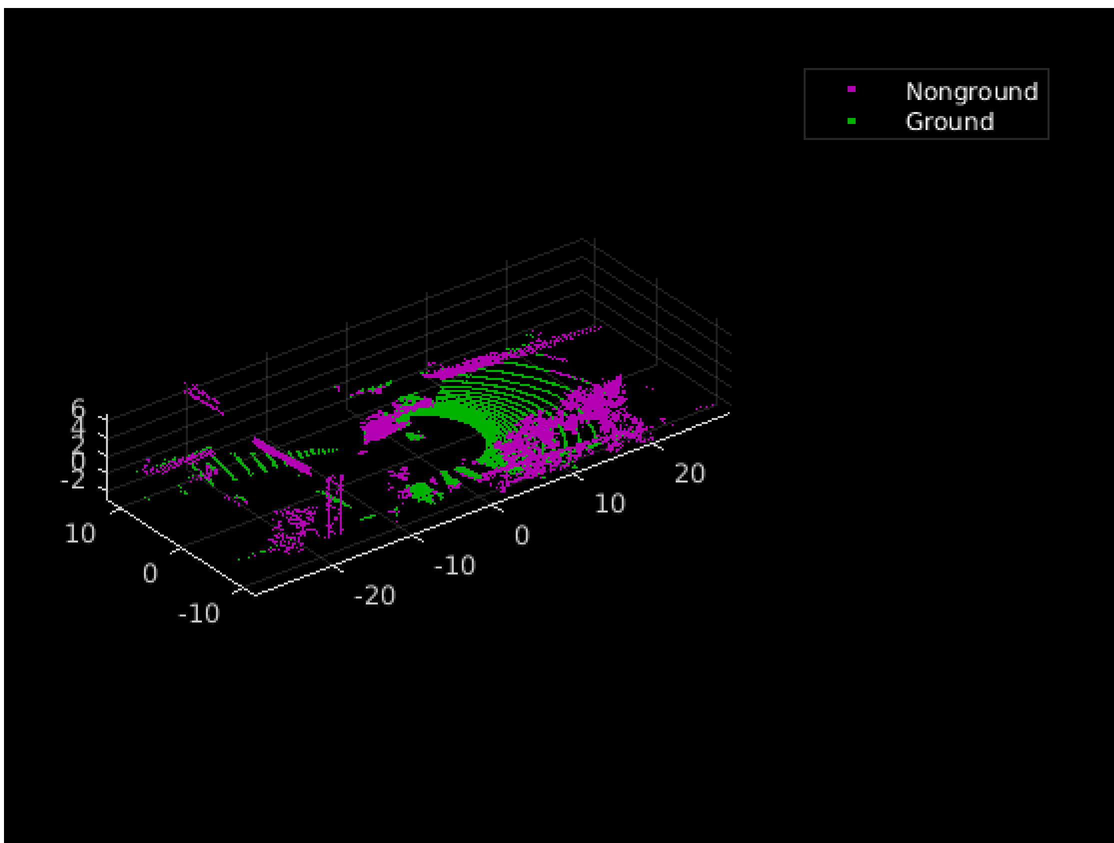
The lidar sensor generates point cloud data either in an organized format or an unorganized format. The data used in this example is collected using an Ouster OS1 lidar sensor. This lidar produces an organized point cloud with 64 horizontal scan lines. The point cloud data is comprised of three channels, representing the *x*-, *y*-, and *z*-coordinates of the points. Each channel is of the size 64-by-1024.

```
dataFile = 'highwayData.mat';
data = load(dataFile);
```

Ground Plane Segmentation

This example employs a hybrid approach that uses the `segmentGroundFromLidarData` and `pcfitplane` functions. First, estimate the ground plane parameters using the `segmentGroundFromLidarData` function. The estimated ground plane is divided into strips along the direction of the vehicle in order to fit the plane, using the `pcfitplane` function on each strip. This hybrid approach robustly fits the ground plane in a piecewise manner and handles variations in the point cloud.

```
% Load point cloud
ptClouds = data.ptCloudData;
ptCloud = ptClouds{1};
% Define ROI for cropping point cloud
xLimit = [-30, 30];
yLimit = [-12, 12];
zLimit = [-3, 15];

roi = [xLimit, yLimit, zLimit];
% Extract ground plane
[nonGround, ground] = helperExtractGround(ptCloud, roi);
figure;
pcshowpair(nonGround, ground);
legend({'\color{white} Nonground','\color{white} Ground'},'Location','northeastoutside');
```



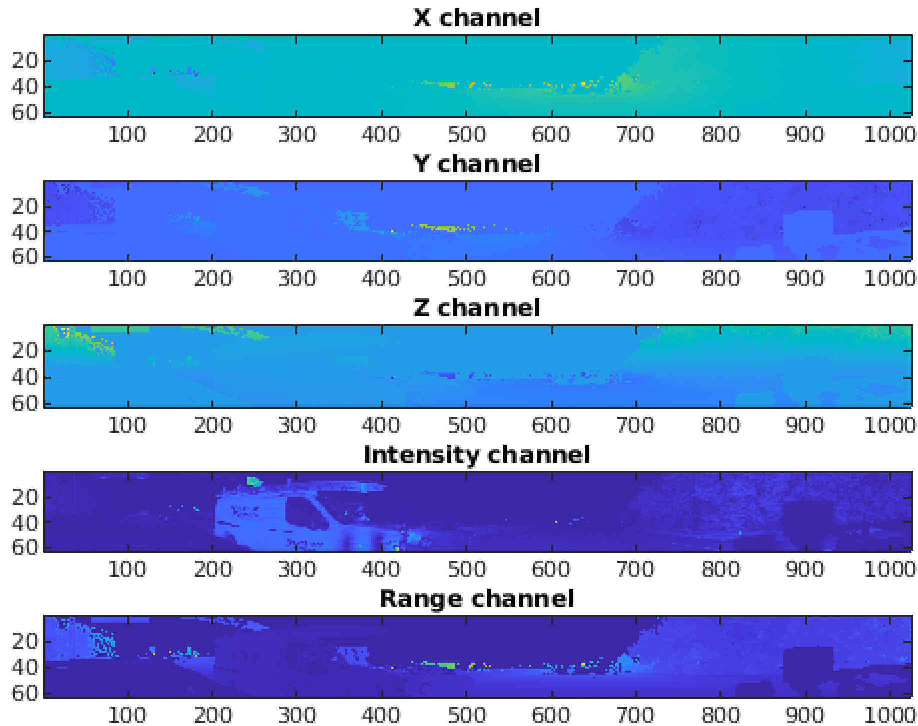
Semantic Segmentation

This example uses a pretrained `PointSeg` network model. `PointSeg` is an end-to-end real-time semantic segmentation network trained for object classes like cars, trucks, and background. The output from the network is a masked image with each pixel labeled per its class. This mask is used to filter different types of objects in the point cloud. The input to the network is five-channel image, that is *x, y, z, intensity, and range*. For more information on the network or how to train the network, refer to the Lidar Point Cloud Semantic Segmentation Using PointSeg Deep Learning Network example

Prepare Input Data

The `helperPrepareData` function generates five-channel data from the loaded point cloud data.

```
% Load and visualize a sample frame
frame = helperPrepareData(ptCloud);
figure;
subplot(5, 1, 1);
imagesc(frame(:, :, 1));
title('X channel');

subplot(5, 1, 2);
imagesc(frame(:, :, 2));
title('Y channel');

subplot(5, 1, 3);
imagesc(frame(:, :, 3));
title('Z channel');

subplot(5, 1, 4);
imagesc(frame(:, :, 4));
title('Intensity channel');

subplot(5, 1, 5);
imagesc(frame(:, :, 5));
title('Range channel');
```



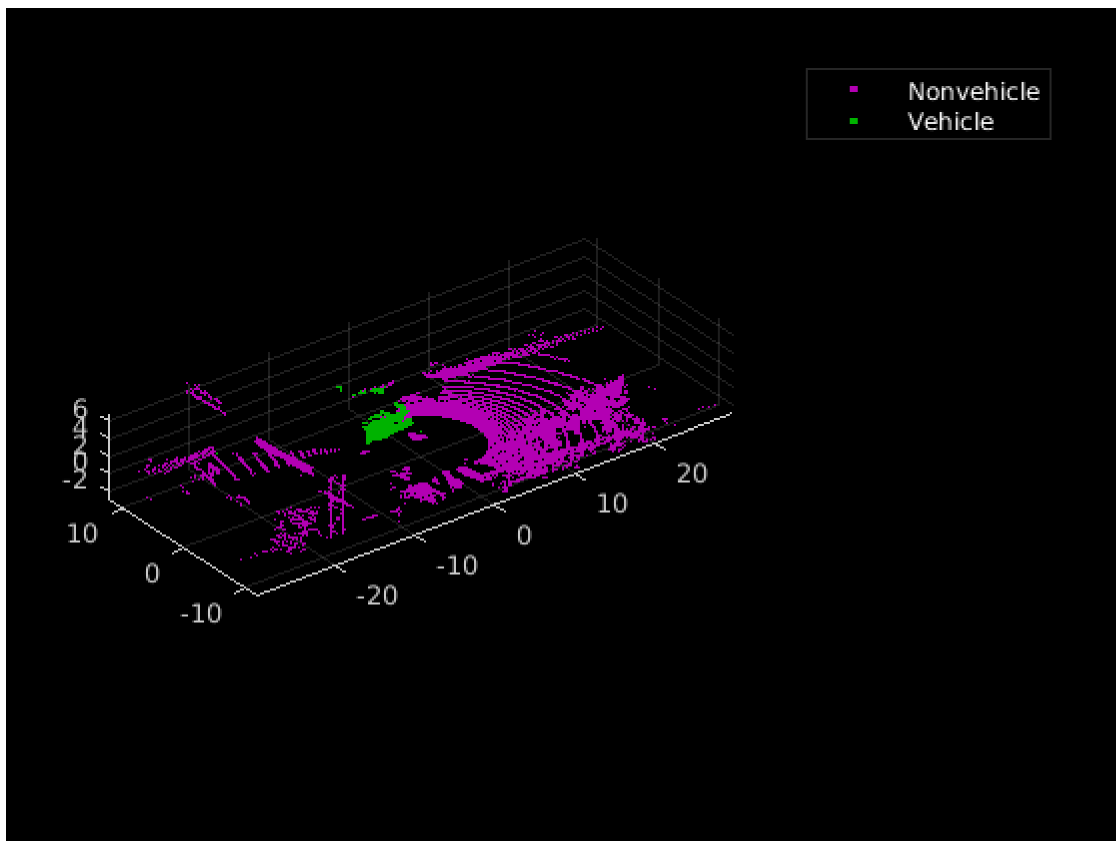
Load the pre-trained network and run forward inference on one frame.

```
% Pretrained PointSeg model file
modelfile = 'pretrainedPointSegModel.mat';

if ~exist('net', 'var')
     load(modelfile);
end

% Define classes
classes = ["background", "car", "truck"];

% Define color map
lidarColorMap = [
            0.98  0.98   0.00  % unknown
            0.01  0.98   0.01  % green color for car
            0.01  0.01   0.98  % blue color for motorcycle
            ];

% Run forward pass
pxdsResults = semanticseg(frame, net);

% Overlay intensity image with segmented output
segmentedImage = labeloverlay(uint8(frame(:, :, 4)), pxdsResults, 'Colormap', lidarColorMap, 'Tra

% Display results
figure;
```

```
imshow(segmentedImage);
helperPixelLabelColorbar(lidarColorMap,classes);
```

Use the generated semantic mask to filter point clouds containing trucks. Similarly, filter point clouds for other classes.

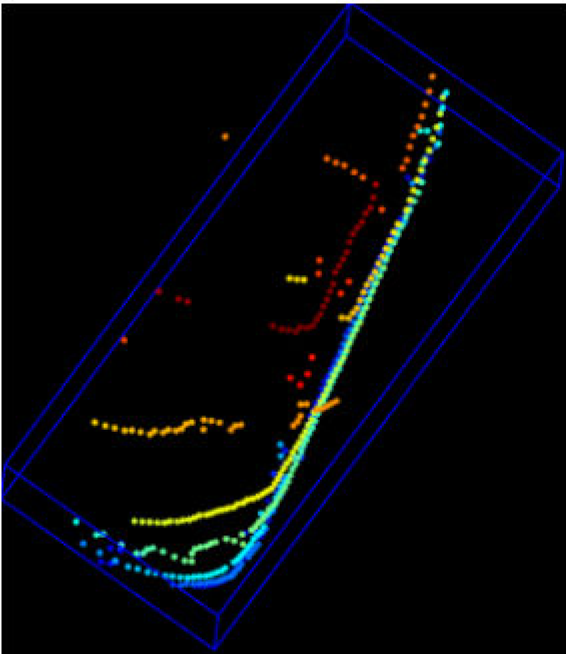
```
truckIndices = pxdsResults == 'truck';
truckPointCloud = select(nonGround, truckIndices, 'OutputSize', 'full');

% Crop point cloud for better display
croppedPtCloud = select(ptCloud, findPointsInROI(ptCloud, roi));
croppedTruckPtCloud = select(truckPointCloud, findPointsInROI(truckPointCloud, roi));

% Display ground and nonground points
figure;
pcshowpair(croppedPtCloud,  croppedTruckPtCloud);
legend({'\color{white} Nonvehicle','\color{white} Vehicle'},'Location','northeastoutside');
```

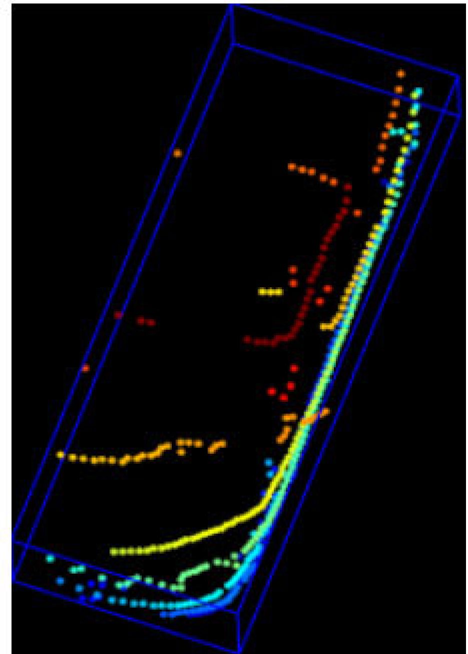


Clustering and Bounding Box Fitting

After extracting point clouds of different object classes, the objects are clustered by applying Euclidean clustering using the `pcsegdist` function. To group all the points belonging to one single cluster, the point cloud obtained as a cluster is used as seed points for growing region in nonground points. Use the `findNearestNeighbors` function to loop over all the points to grow the region. The extracted cluster is fitted in an L-shape bounding box using the `pcfitcuboid` function. These clusters of vehicles resemble the shape of the letter L when seen from a top-down view. This feature helps in estimating the orientation of the vehicle. The oriented bounding box fitting helps in estimating the heading angle of the objects, which is useful in applications such as path planning, and traffic maneuvering traffic.

The cuboid boundaries of the clusters can also be calculated by finding the minimum and maximum spatial extents in each direction. However, this method fails in estimating the orientation of the detected vehicles. The difference between the two methods is shown in the figure.

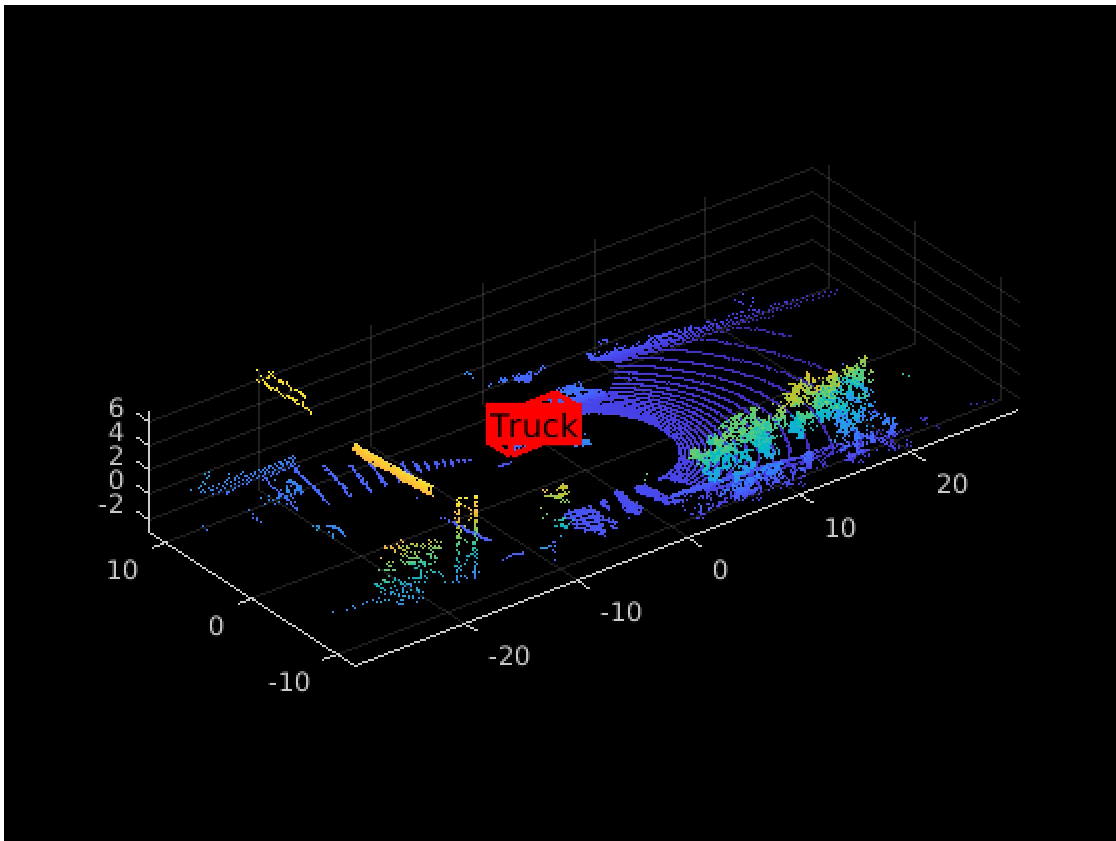




Min. Area Rectangle   L-Shape Fitting

```
[labels, numClusters] = pcsegdist(croppedTruckPtCloud, 1);

% Define cuboid parameters
params = zeros(0, 9);

for clusterIndex = 1:numClusters
    ptsInCluster = labels == clusterIndex;

    pc = select(croppedTruckPtCloud, ptsInCluster);
    location = pc.Location;

    xl = (max(location(:, 1)) - min(location(:, 1)));
    yl = (max(location(:, 2)) - min(location(:, 2)));
```

```
    zl = (max(location(:, 3)) - min(location(:, 3)));

    % Filter small bounding boxes
    if size(location, 1)*size(location, 2) > 20 && any(any(pc.Location)) && xl > 1 && yl > 1
        indices = zeros(0, 1);
        objectPtCloud = pointCloud(location);
        for i = 1:size(location, 1)
            seedPoint = location(i, :);
            indices(end+1) = findNearestNeighbors(nonGround, seedPoint, 1);
        end

        % Remove overlapping indices
        indices = unique(indices);

        % Fit oriented bounding box
        model = pcfitcuboid(select(nonGround, indices));
        params(end+1, :) = model.Parameters;
    end
end

% Display point cloud and detected bounding box
figure;
pcshow(croppedPtCloud.Location, croppedPtCloud.Location(:, 3));
showShape('cuboid', params, "Color", "red", "Label", "Truck");
```



Visualization Setup

Use the `helperLidarObjectDetectionDisplay` class to visualize the complete workflow in one window. The layout of the visualization window is divided into the following sections:

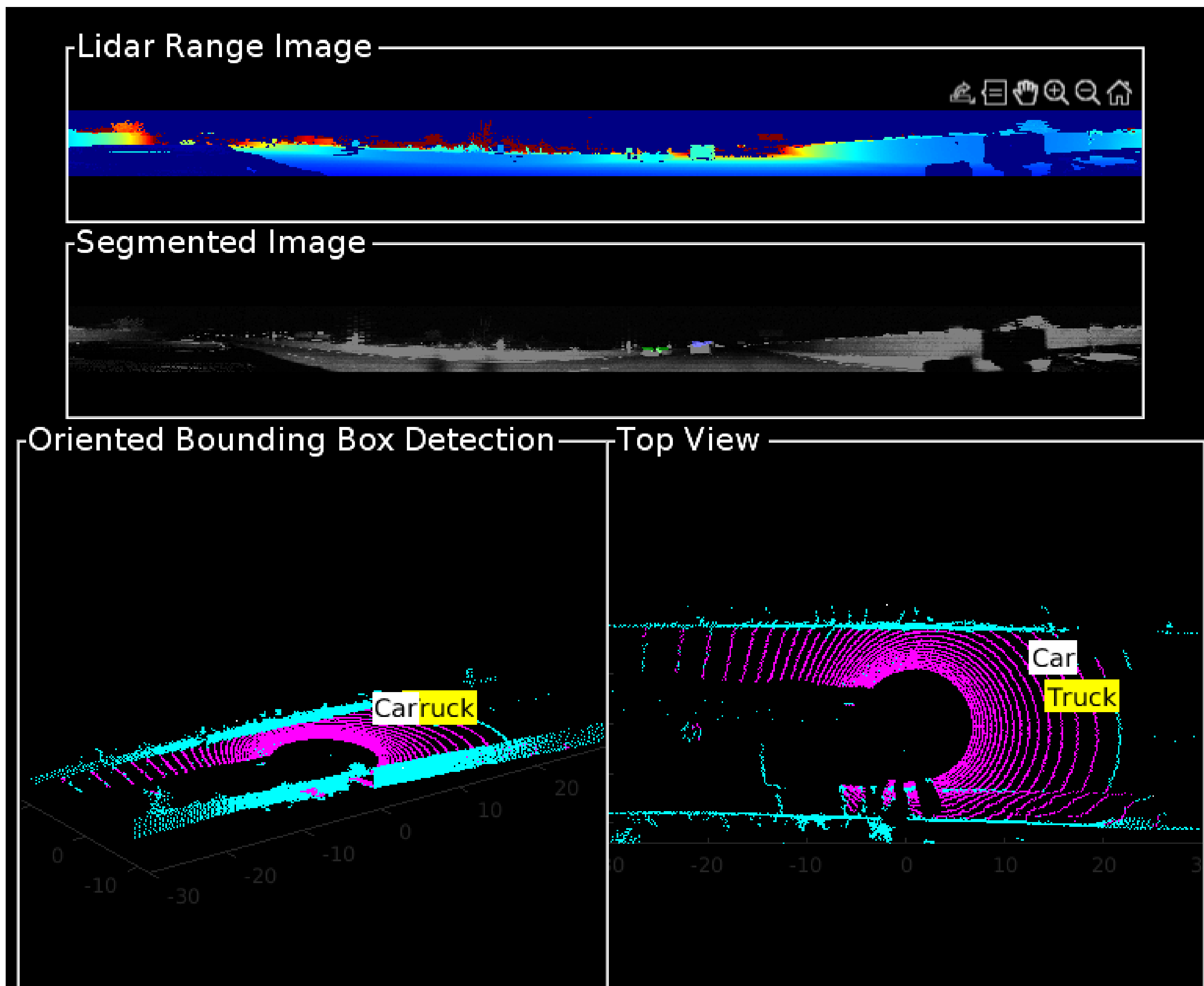
1. Lidar Range Image: point cloud image in 2-D as a range image
2. Segmented Image: Detected labels generated from the semantic segmentation network overlaid with the intensity image or the fourth channel of the data
3. Oriented Bounding Box Detection: 3-D point cloud with oriented bounding boxes
4. Top View: Top view of the point cloud with oriented bounding boxes

```
display = helperLidarObjectDetectionDisplay;
```

Loop Through Data

The `helperLidarObjectDetection` class is a wrapper encapsulating all the segmentation, clustering, and bounding box fitting steps mentioned in the above sections. Use the `findDetections` function to extract the detected objects.

```
% Initialize lidar object detector
lidarDetector = helperLidarObjecDetector('ModelFile',modelfile, 'XLimits', xLimit,...
    'YLimit', yLimit, 'ZLimit', zLimit);

% Prepare 5-D lidar data
inputData = helperPrepareData(data.ptCloudData);

% Set random number generator for reproducible results.
S = rng(2018);

% Initialize the display
initializeDisplay(display);

numFrames = numel(inputData);
for count = 1:numFrames

    % Get current data
    input = inputData{count};

    rangeImage = input(:, :, 5);

    % Extact bounding boxes from lidar data
    [boundingBox, coloredPtCloud, pointLabels] = detectBbox(lidarDetector, input);

    % Update display with colored point cloud
    updatePointCloud(display, coloredPtCloud);

    % Update bounding boxes
    updateBoundingBox(display, boundingBox);

    % Update segmented image
    updateSegmentedImage(display, pointLabels, rangeImage);

    drawnow('limitrate');
end
```



Tracking Oriented Bounding Boxes

In this example, you use a joint probabilistic data association (JPDA) tracker. The time step `dt` is set to 0.1 seconds since the dataset is captured at 10 Hz. The state-space model used in the tracker is based on a cuboid model with parameters, $[x, y, z, \phi, l, w, h]$. For more details on how to track bounding boxes in lidar data, see Track Vehicles Using Lidar: From Point Cloud to Track List. In this example, the class information is provided using the `ObjectAttributes` property of the `objectDetection object`. When creating new tracks, the filter initialization function, defined using the helper function `helperMultiClassInitIMMFilter` uses the class of the detection to set up initial dimensions of the object. This helps the tracker to adjust bounding box measurement model with the appropriate dimensions of the track.

Set up a JPDA tracker object with these parameters.

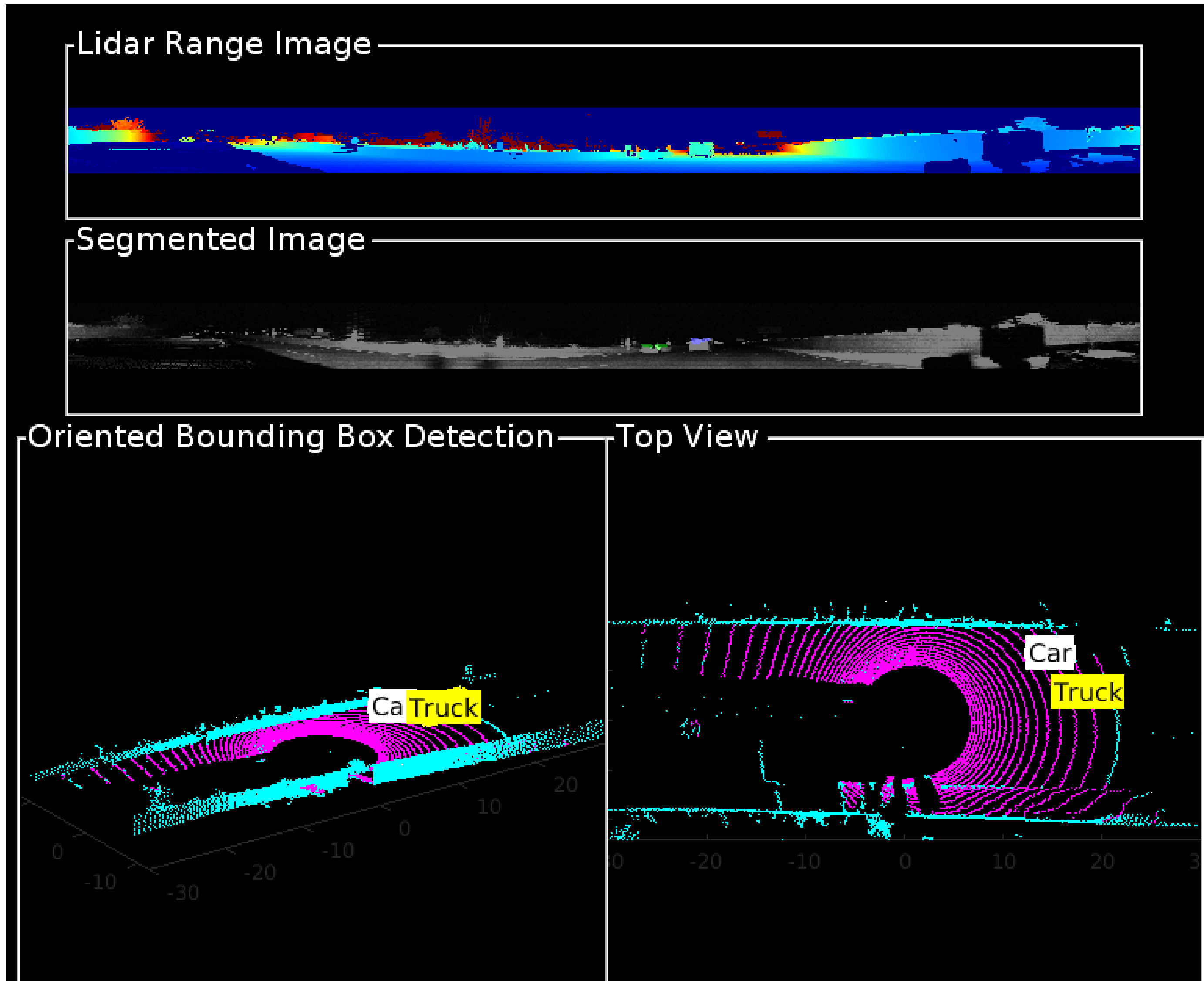
```
assignmentGate = [10 100]; % Assignment threshold;
confThreshold = [7 10];    % Confirmation threshold for history logi
delThreshold = [2 3];     % Deletion threshold for history logic
```

```
Kc = 1e-5;                 % False-alarm rate per unit volume

% IMM filter initialization function
filterInitFcn = @helperMultiClassInitIMMFilter;

% A joint probabilistic data association tracker with IMM filter
tracker = trackerJPDA('FilterInitializationFcn',filterInitFcn,...
    'TrackLogic','History',...
    'AssignmentThreshold',assignmentGate,...
    'ClutterDensity',Kc,...
    'ConfirmationThreshold',confThreshold,...
    'DeletionThreshold',delThreshold,'InitializationThreshold',0);

allTracks = struct([]);
time = 0;
dt = 0.1;

% Define Measurement Noise
measNoise = blkdiag(0.25*eye(3),25,eye(3));

numTracks = zeros(numFrames, 2);
```

The detected objects are assembled as a cell array of `objectDetection` (Automated Driving Toolbox) objects using the `helperAssembleDetections` function.

```
display = helperLidarObjectDetectionDisplay;
initializeDisplay(display);

for count = 1:numFrames
    time = time + dt;
    % Get current data
    input = inputData{count};

    rangeImage = input(:, :, 5);

    % Extact bounding boxes from lidar data
    [boundingBox, coloredPtCloud, pointLabels] = detectBbox(lidarDetector, input);

    % Assemble bounding boxes into objectDetections
    detections = helperAssembleDetections(boundingBox, measNoise, time);

    % Pass detections to tracker
    if ~isempty(detections)
        % Update the tracker
         [confirmedTracks, tentativeTracks, allTracks, info] = tracker(detections, time);
         numTracks(count, 1) = numel(confirmedTracks);
    end

    % Update display with colored point cloud
    updatePointCloud(display, coloredPtCloud);

    % Update segmented image
    updateSegmentedImage(display, pointLabels, rangeImage);

    % Update the display if the tracks are not empty
     if ~isempty(confirmedTracks)
        updateTracks(display, confirmedTracks);
     end
```

```
    drawnow('limitrate');
end
```



Summary

This example showed how to detect and classify vehicles fitted with oriented bounding box on lidar data. You also learned how to use IMM filter to track objects with multiple class information. The semantic segmentation results can be improved further by adding more training data.

Supporting Functions

helperPrepareData

```
function multiChannelData = helperPrepareData(input)
% Create 5-channel data as x, y, z, intensity and range
% of size 64-by-1024-by-5 from pointCloud.
```

```
if isa(input, 'cell')
    numFrames = numel(input);
    multiChannelData = cell(1, numFrames);
    for i = 1:numFrames
        inputData = input{i};

        x = inputData.Location(:,:,1);
        y = inputData.Location(:,:,2);
        z = inputData.Location(:,:,3);

        intensity = inputData.Intensity;
        range = sqrt(x.^2 + y.^2 + z.^2);

        multiChannelData{i} = cat(3, x, y, z, intensity, range);
    end
else
    x = input.Location(:,:,1);
    y = input.Location(:,:,2);
    z = input.Location(:,:,3);

    intensity = input.Intensity;
    range = sqrt(x.^2 + y.^2 + z.^2);

    multiChannelData = cat(3, x, y, z, intensity, range);
end
end
```

pixelLabelColorbar

```
function helperPixelLabelColorbar(cmap, classNames)
% Add a colorbar to the current axis. The colorbar is formatted
% to display the class names with the color.

colormap(gca,cmap)

% Add colorbar to current figure.
c = colorbar('peer', gca);

% Use class names for tick marks.
c.TickLabels = classNames;
numClasses = size(cmap,1);

% Center tick labels.
c.Ticks = 1/(numClasses*2):1/numClasses:1;

% Remove tick mark.
c.TickLength = 0;
end
```

helperExtractGround

```
function [ptCloudNonGround, ptCloudGround] = helperExtractGround(ptCloudIn, roi)
% Crop the point cloud

idx = findPointsInROI(ptCloudIn, roi);
pc = select(ptCloudIn, idx, 'OutputSize', 'full');

% Get the ground plane the indices using piecewise plane fitting
```

```
[ptCloudGround, idx] = piecewisePlaneFitting(pc, roi);

nonGroundIdx = true(size(pc.Location, [1,2]));
nonGroundIdx(idx) = false;
ptCloudNonGround = select(pc, nonGroundIdx, 'OutputSize', 'full');
end


function [groundPlane, idx] = piecewisePlaneFitting(ptCloudIn, roi)
groundPtsIdx = ...
    segmentGroundFromLidarData(ptCloudIn, ...
    'ElevationAngleDelta', 5, 'InitialElevationAngle', 15);
groundPC = select(ptCloudIn,groundPtsIdx, 'OutputSize', 'full');

% Divide x-axis in 3 regions
segmentLength = (roi(2) - roi(1))/3;

x1 = [roi(1), roi(1) + segmentLength];
x2 = [x1(2), x1(2) + segmentLength];
x3 = [x2(2), x2(2) + segmentLength];

roi1 = [x1, roi(3: end)];
roi2 = [x2, roi(3: end)];
roi3 = [x3, roi(3: end)];

idxBack = findPointsInROI(groundPC, roi1);
idxCenter = findPointsInROI(groundPC, roi2);
idxForward = findPointsInROI(groundPC, roi3);

% Break the point clouds in front and back
ptBack = select(groundPC, idxBack, 'OutputSize', 'full');

ptForward = select(groundPC, idxForward, 'OutputSize', 'full');

[~, inliersForward] = planeFit(ptForward);
[~, inliersBack] = planeFit(ptBack);
idx = [inliersForward; idxCenter; inliersBack];
groundPlane = select(ptCloudIn, idx, 'OutputSize', 'full');
end

function [plane, inlinersIdx] = planeFit(ptCloudIn)
[~, inlinersIdx, ~] = pcfitplane(ptCloudIn, 1, [0, 0, 1]);
plane = select(ptCloudIn, inlinersIdx, 'OutputSize', 'full');
end
```

helperAssembleDetections

```
function mydetections = helperAssembleDetections(bboxes, measNoise, timestamp)
% Assemble bounding boxes as cell array of objectDetection

mydetections = cell(size(bboxes, 1), 1);
for i = 1:size(bboxes, 1)
    classid = bboxes(i, end);
    lidarModel = [bboxes(i, 1:3), bboxes(i, end-1), bboxes(i, 4:6)];
    % To avoid direct confirmation by the tracker, the ClassID is passed as
    % ObjectAttributes.
    mydetections{i} = objectDetection(timestamp, ...
        lidarModel', 'MeasurementNoise',...
```

```
        measNoise, 'ObjectAttributes', struct('ClassID',classid));
end
end
```

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 2017

[2] Y. Wang, T. Shi, P. Yun, L. Tai, and M. Liu, “Pointseg: Real-time semantic segmentation based on 3d lidar point cloud,” arXiv preprint arXiv:1807.06288, 2018.

Feature-Based Map Building from Lidar Data

This example demonstrates how to process 3-D lidar data from a sensor mounted on a vehicle to progressively build a map. Such a map is suitable for automated driving workflows such as localization and navigation. These maps can be used to localize a vehicle within a few centimeters.

Overview

There are different ways to register point clouds. The general way is to use the complete point cloud for registration. “Build a Map from Lidar Data” (Automated Driving Toolbox) example uses this approach for map building. This example uses distinctive features extracted from the point cloud for map building.

In this example, you will learn how to:

- Load and visualize recorded driving data.
- Build a map using lidar scans.

Load Recorded Driving Data

The data used in this example represents approximately 100 seconds of lidar, GPS, and IMU data. The data is saved in separate MAT-files as `timetable` objects. Download the lidar data MAT file from the repository and load it into the MATLAB® workspace.

Note: This download can take a few minutes.

```
baseDownloadURL = ['https://github.com/mathworks/udacity-self-driving-data' ...
                   '-subset/raw/master/drive_segment_11_18_16/'];
dataFolder      = fullfile(tempdir, 'drive_segment_11_18_16', filesep);
options         = weboptions('Timeout', Inf);

lidarFileName = dataFolder + "lidarPointClouds.mat";

% Check whether the folder and data file already exist or not
folderExists  = exist(dataFolder, 'dir');
matfilesExist = exist(lidarFileName, 'file');

% Create a new folder if it does not exist
if ~folderExists
    mkdir(dataFolder);
end

% Download the lidar data if it does not exist
if ~matfilesExist
    disp('Downloading lidarPointClouds.mat (613 MB)...');
    websave(lidarFileName, baseDownloadURL + "lidarPointClouds.mat", options);

end
```

```
Downloading lidarPointClouds.mat (613 MB)...
```

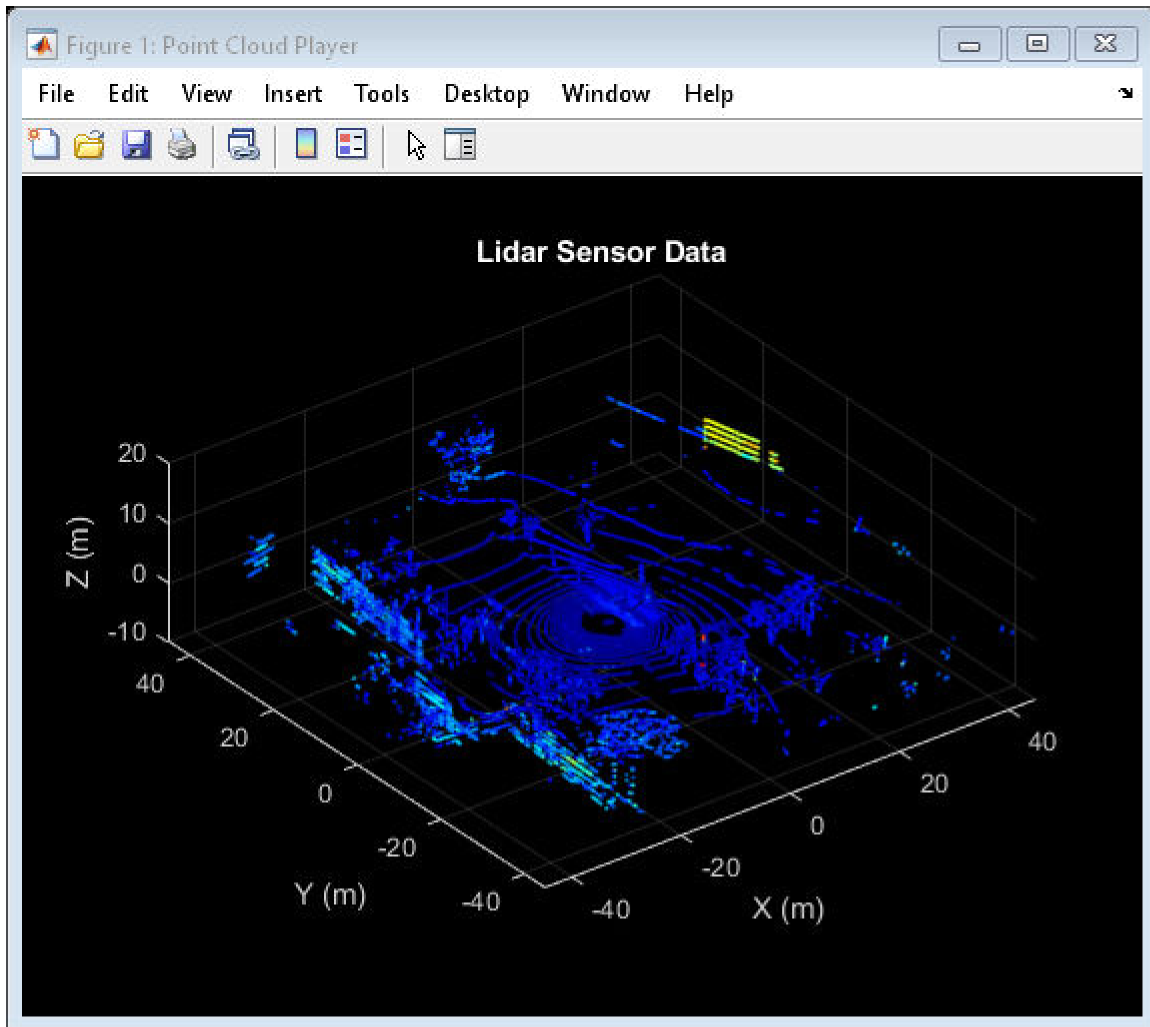
Load the point cloud data saved from a Velodyne® HDL32E lidar sensor. Each lidar scan is stored as a 3-D point cloud using the `pointCloud` object. This object internally organizes the data using a K-d tree data structure for faster search. The timestamp associated with each lidar scan is recorded in the `Time` variable of the `timetable` object.

```
% Load lidar data from MAT-file
data = load(lidarFileName);
lidarPointClouds = data.lidarPointClouds;

% Display first few rows of lidar data
head(lidarPointClouds)

ans=8×1 timetable
        Time            PointCloud
    _____________    ________________

    23:46:10.5115    [1x1 pointCloud]
    23:46:10.6115    [1x1 pointCloud]
    23:46:10.7116    [1x1 pointCloud]
    23:46:10.8117    [1x1 pointCloud]
    23:46:10.9118    [1x1 pointCloud]
    23:46:11.0119    [1x1 pointCloud]
    23:46:11.1120    [1x1 pointCloud]
    23:46:11.2120    [1x1 pointCloud]
```

Visualize Driving Data

To understand what the scene contains, visualize the recorded lidar data using the `pcplayer` object.

```
% Specify limits for the player
xlimits = [-45 45]; % meters
ylimits = [-45 45];
zlimits = [-10 20];

% Create a pcplayer to visualize streaming point clouds from lidar sensor
lidarPlayer = pcplayer(xlimits, ylimits, zlimits);

% Customize player axes labels
xlabel(lidarPlayer.Axes, 'X (m)');
ylabel(lidarPlayer.Axes, 'Y (m)');
zlabel(lidarPlayer.Axes, 'Z (m)');
title(lidarPlayer.Axes, 'Lidar Sensor Data');

% Loop over and visualize the data
for l = 1 : height(lidarPointClouds)

    % Extract point cloud
    ptCloud = lidarPointClouds.PointCloud(l);

    % Update lidar display
    view(lidarPlayer, ptCloud);
end
```



Use Recorded Lidar Data to Build a Map

Lidars can be used to build centimeter-accurate maps which can later be used for in-vehicle localization. A typical approach to build such a map is to align successive lidar scans obtained from a moving vehicle and combine them into a single, large point cloud. The rest of this example explores this approach to building a map.

Pre-processing

Take two point clouds corresponding to nearby lidar scans. Every tenth scan is used to speed up processing and accumulate enough motion between scans.

```
rng(0);
skipFrames = 10;
frameNum   = 100;

fixed  = lidarPointClouds.PointCloud(frameNum);
moving = lidarPointClouds.PointCloud(frameNum + skipFrames);
```

Process the point cloud to retain structures in the point cloud that are distinctive. These steps are executed using the `helperProcessPointCloud` function:

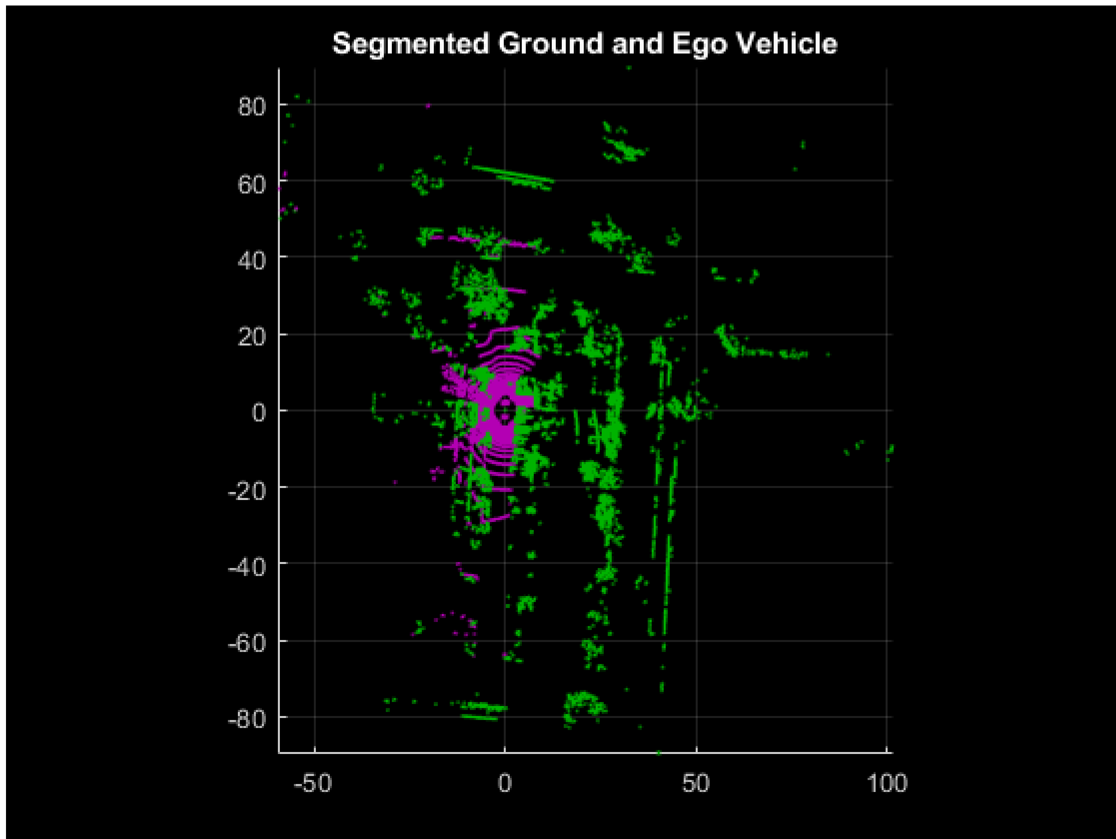
- Detect and remove the ground plane
- Detect and remove ego-vehicle

These steps are described in more detail in the “Ground Plane and Obstacle Detection Using Lidar” (Automated Driving Toolbox) example.

```
fixedProcessed  = helperProcessPointCloud(fixed);
movingProcessed = helperProcessPointCloud(moving);
```

Display the initial and processed point clouds in top-view. Magenta points correspond to the ground plane and ego vehicle.

```
hFigFixed = figure;
axFixed   = axes('Parent', hFigFixed, 'Color', [0,0,0]);

pcshowpair(fixed, fixedProcessed, 'Parent', axFixed);
title(axFixed, 'Segmented Ground and Ego Vehicle');
view(axFixed, 2);
```



Downsample the point clouds to improves registration accuracy and algorithm speed.

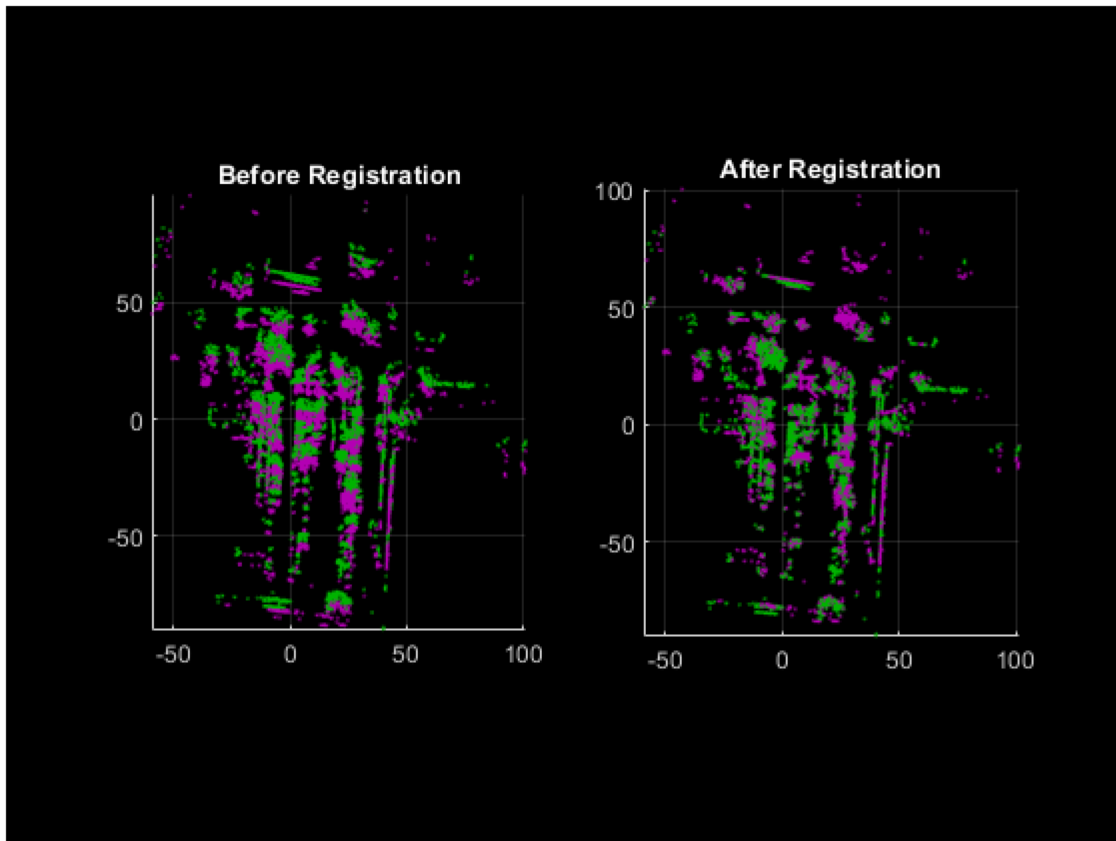
```
gridStep = 0.1;
fixedDownsampled  = pcdownsample(fixedProcessed, "gridAverage", gridStep);
movingDownsampled = pcdownsample(movingProcessed, "gridAverage", gridStep);
```

Feature-Based Registration

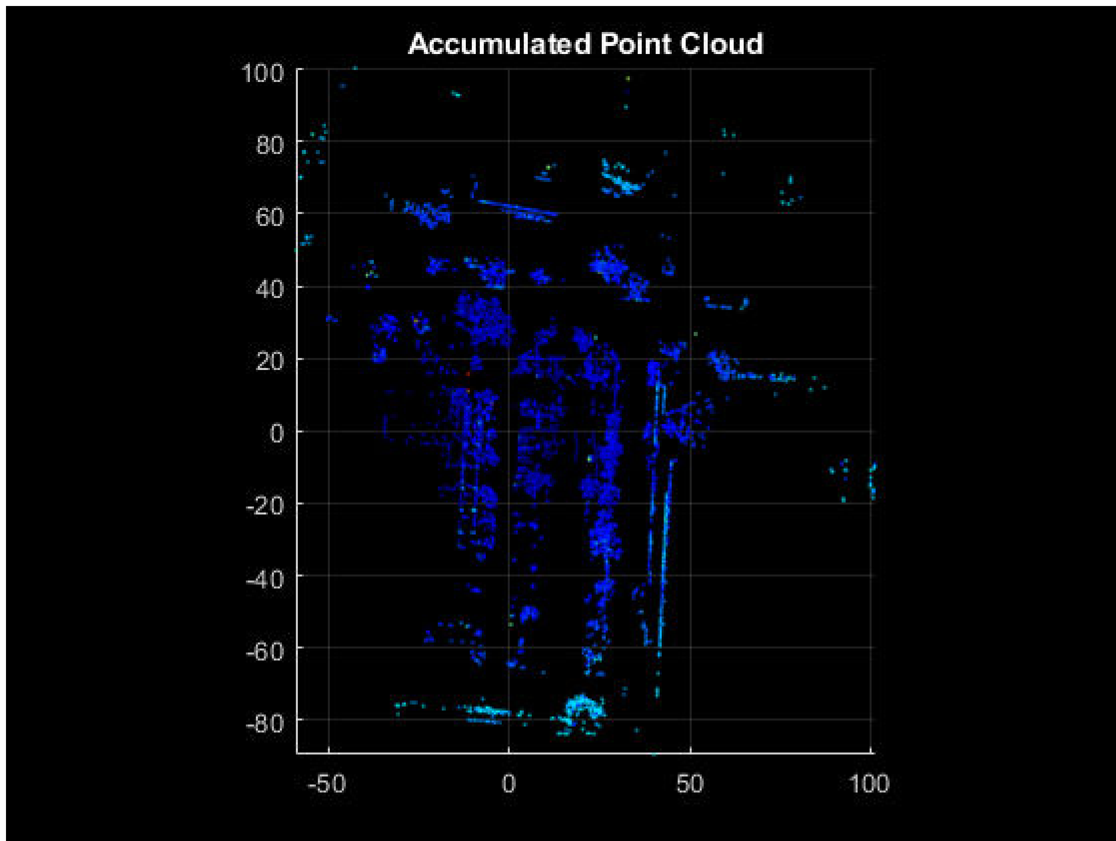
Align and combine successive lidar scans using feature-based registration as follows:

- Extract Fast Point Feature Histogram (FPFH) descriptors from each scan using the `extractFPFHFeatures` function.
- Identify point correspondences by comparing the descriptors using the `pcmatchfeatures` function.
- Estimate the rigid transformation from point correspondences using the `estimateGeometricTransform3D` function.
- Align and merge the point cloud with respect to reference point cloud using the estimated transformation. This is performed using the `pcalign` function.

```
% Extract FPFH Features for each point cloud
neighbors = 60;
[fixedFeature, fixedValidInds]   =  extractFPFHFeatures(fixedDownsampled, ...
    'NumNeighbors', neighbors);
[movingFeature, movingValidInds] = extractFPFHFeatures(movingDownsampled, ...
    'NumNeighbors', neighbors);

fixedValidPts  = select(fixedDownsampled, fixedValidInds);
movingValidPts = select(movingDownsampled, movingValidInds);

% Identify the point correspondences
method    = 'Approximate';
threshold = 1;
ratio     = 0.96;
indexPairs = pcmatchfeatures(movingFeature, fixedFeature, movingValidPts, ...
    fixedValidPts,"Method",method, "MatchThreshold",threshold, ...
    "RejectRatio", ratio);

matchedFixedPts  = select(fixedValidPts, indexPairs(:,2));
matchedMovingPts = select(movingValidPts, indexPairs(:,1));

% Estimate rigid transform of moving point cloud with respect to reference
% point cloud
maxDist = 1.5;
maxNumTrails = 2000;
tform = estimateGeometricTransform3D(matchedMovingPts.Location, ...
    matchedFixedPts.Location, "rigid", "MaxDistance", maxDist, ...
    "MaxNumTrials", maxNumTrails);

% Transform the moving point cloud to the reference point cloud, to
% visualize the alignment before and after registration
movingReg = pctransform(movingProcessed, tform);

% Moving and fixed point clouds are represented by magenta and green colors
hFigAlign = figure;
axAlign1 = subplot(1, 2, 1,'Color', [0, 0, 0], 'Parent', hFigAlign);
pcshowpair(movingProcessed, fixedProcessed, 'Parent', axAlign1);
title(axAlign1, 'Before Registration');
view(axAlign1, 2);

axAlign2 = subplot(1, 2, 2,'Color', [0, 0, 0], 'Parent', hFigAlign);
pcshowpair(movingReg, fixedProcessed, 'Parent', axAlign2);
title(axAlign2, 'After Registration');
view(axAlign2, 2);
```




```
% Align and merge the point clouds
alignGridStep = 1;
ptCloudAccum  = pcalign([fixedProcessed, movingProcessed], ...
    [rigid3d, tform], alignGridStep);

% Visualize the accumulated point cloud
hFigAccum = figure;
axAccum   = axes('Parent', hFigAccum, 'Color', [0,0,0]);

pcshow(ptCloudAccum, 'Parent', axAccum);
title(axAccum, 'Accumulated Point Cloud');
view(axAccum, 2);
```

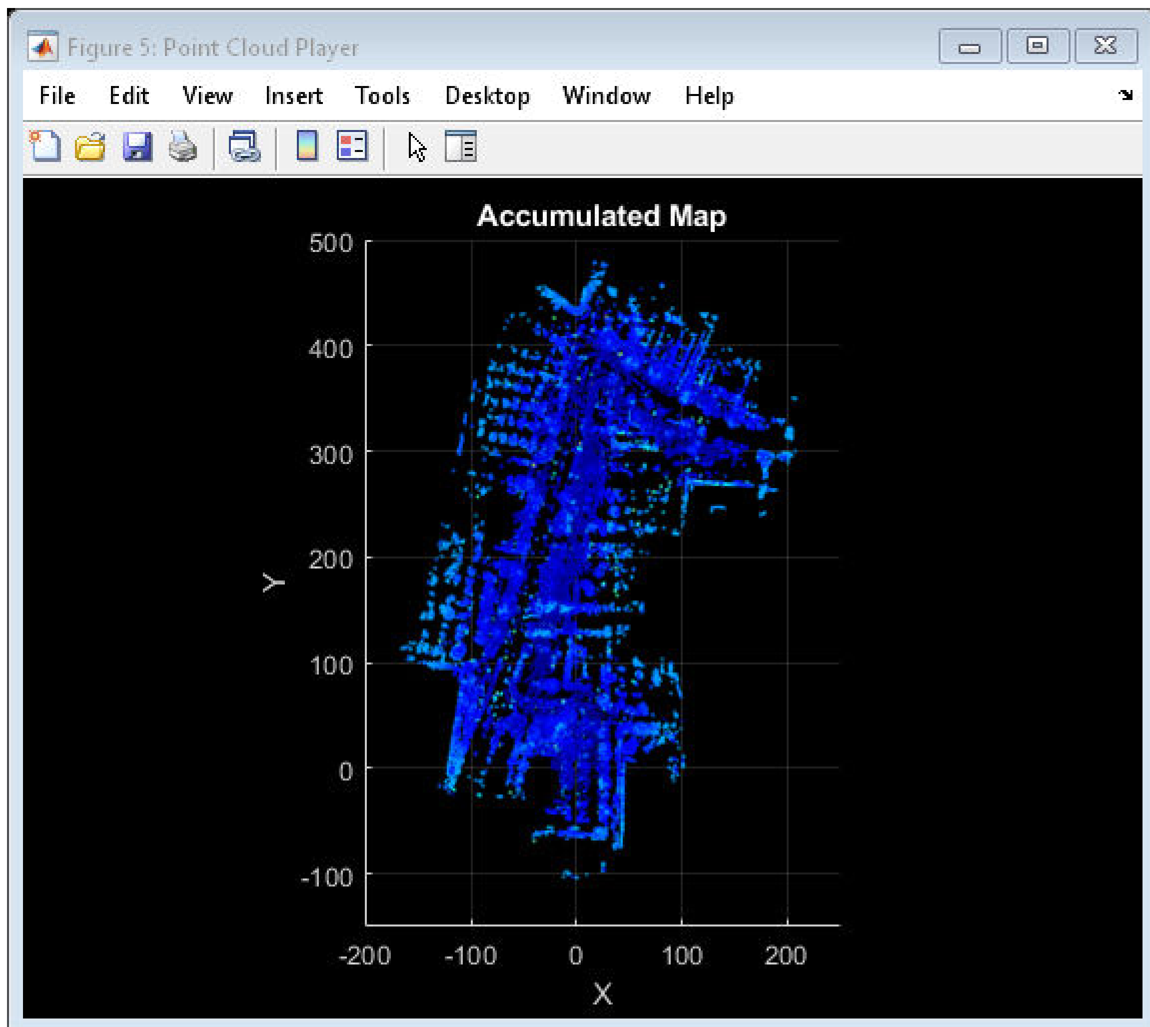


Map Generation

Apply pre-processing and feature-based registration steps in a loop over the entire sequence of recorded data. The result is a map of the environment traversed by the vehicle.

```
rng(0);
numFrames     = height(lidarPointClouds);
accumTform    = rigid3d;
pointCloudMap = pointCloud(zeros(0, 0, 3));

% Specify limits for the player
xlimits = [-200 250]; % meters
ylimits = [-150 500];
zlimits = [-100 100];

% Create a pcplayer to visualize map
mapPlayer = pcplayer(xlimits, ylimits, zlimits);
title(mapPlayer.Axes, 'Accumulated Map');
mapPlayer.Axes.View = [0, 90];

% Loop over the entire data to generate map
for n = 1 : skipFrames : numFrames - skipFrames

    % Get the nth point cloud
    ptCloud = lidarPointClouds.PointCloud(n);

    % Segment ground and remove ego vehicle
```

```
    ptProcessed = helperProcessPointCloud(ptCloud);

    % Downsample the point cloud for speed of operation
    ptDownsampled = pcdownsample(ptProcessed, "gridAverage", gridStep);

    % Extract the features from point cloud
    [ptFeature, ptValidInds]  = extractFPFHFeatures(ptDownsampled, ...
        'NumNeighbors', neighbors);
    ptValidPts  = select(ptDownsampled, ptValidInds);

    if n==1
        moving        = ptValidPts;
        movingFeature = ptFeature;
        pointCloudMap = ptValidPts;
    else
        fixed         = moving;
        fixedFeature  = movingFeature;
        moving        = ptValidPts;
        movingFeature = ptFeature;

        % Match the features to find correspondences
        indexPairs = pcmatchfeatures(movingFeature, fixedFeature, moving, ...
            fixed, "Method",method,"MatchThreshold",threshold, ...
            "RejectRatio", ratio);
        matchedFixedPts  = select(fixed, indexPairs(:,2));
        matchedMovingPts = select(moving, indexPairs(:,1));

        % Register moving point cloud w.r.t reference point cloud
        tform = estimateGeometricTransform3D(matchedMovingPts.Location, ...
            matchedFixedPts.Location, "rigid", "MaxDistance", maxDist, ...
            "MaxNumTrials", maxNumTrails);

        % Compute accumulated transformation to original reference frame
        accumTform = rigid3d(tform.T * accumTform.T);

        % Align and merge moving point cloud to accumulated map
        pointCloudMap = pcalign([pointCloudMap, moving], ...
            [rigid3d, accumTform], alignGridStep);
    end

    % Update map display
    view(mapPlayer, pointCloudMap);
end
```



Functions

`pcdownsample` | `extractFPFHFeatures` | `pcmatchfeatures` | `estimateGeometricTransform3D` | `pctransform` | `pcalign`

Objects

`pcplayer` | `pointCloud`

Related Topics

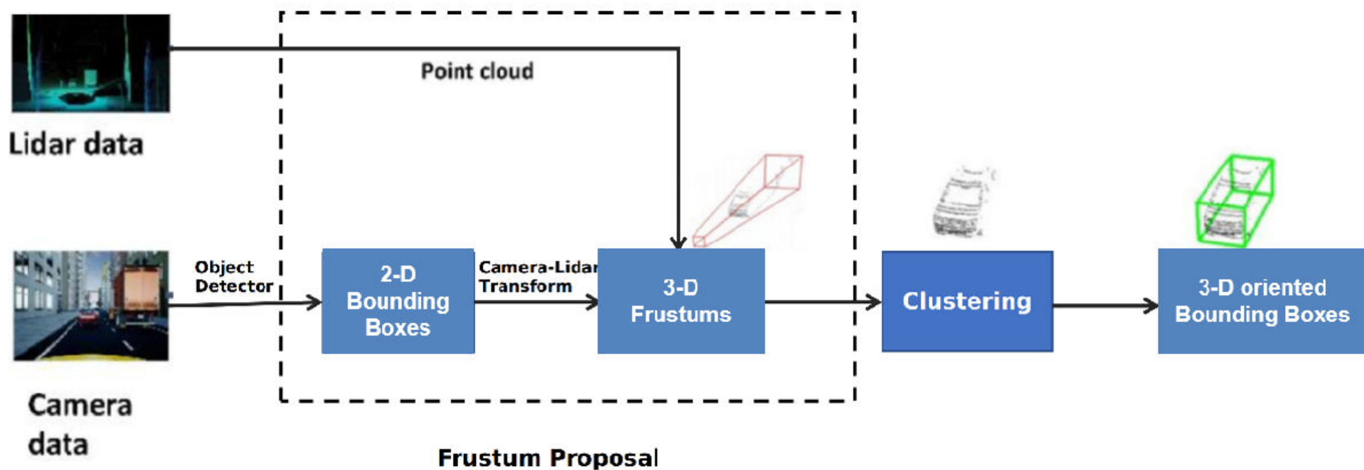
- "Build a Map from Lidar Data" (Automated Driving Toolbox)
- "Ground Plane and Obstacle Detection Using Lidar" (Automated Driving Toolbox)

External Websites

- Udacity Self-Driving Car Data Subset (MathWorks GitHub repository)

Detect Vehicles in Lidar Using Image Labels

This example shows you how to detect vehicles in lidar using label data from a co-located camera with known lidar-to-camera calibration parameters. Use this workflow in MATLAB® to estimate 3-D oriented bounding boxes in lidar based on 2-D bounding boxes in the corresponding image. You will also see how to automatically generate ground truth as a distance for 2-D bounding boxes in a camera image using lidar data. This figure provides an overview of the process.



Load Data

This example uses lidar data collected on a highway from an Ouster OS1 lidar sensor and image data from a front-facing camera mounted on the ego vehicle. The lidar and camera data are approximately time-synced and calibrated to estimate their intrinsic and extrinsic parameters. For more information on lidar camera calibration, see "Lidar and Camera Calibration" on page 1-2.

Note: The download time for the data depends on the speed of your internet connection. During the execution of this code block, MATLAB is temporarily unresponsive.

```
lidarTarFileUrl = 'https://www.mathworks.com/supportfiles/lidar/data/WPI_LidarData.tar.gz';
imageTarFileUrl = 'https://www.mathworks.com/supportfiles/lidar/data/WPI_ImageData.tar.gz';

outputFolder = fullfile(tempdir,'WPI');
lidarDataTarFile = fullfile(outputFolder,'WPI_LidarData.tar.gz');
imageDataTarFile = fullfile(outputFolder,'WPI_ImageData.tar.gz');

if ~exist(outputFolder,'dir')
    mkdir(outputFolder)
end

if ~exist(lidarDataTarFile,'file')
    disp('Downloading WPI Lidar driving data (760 MB)...')
    websave(lidarDataTarFile,lidarTarFileUrl)
    untar(lidarDataTarFile,outputFolder)
end

% Check if lidar tar.gz file is downloaded, but not uncompressed.
if ~exist(fullfile(outputFolder,'WPI_LidarData.mat'),'file')
```

```
    untar(lidarDataTarFile,outputFolder)
end

if ~exist(imageDataTarFile,'file')
    disp('Downloading WPI Image driving data (225 MB)...')
    websave(imageDataTarFile,imageTarFileUrl)
    untar(imageDataTarFile,outputFolder)
end

% Check if image tar.gz file is downloaded, but not uncompressed.
if ~exist(fullfile(outputFolder,'imageData'),'dir')
    untar(imageDataTarFile,outputFolder)
end

imageDataLocation = fullfile(outputFolder,'imageData');
images = imageSet(imageDataLocation);
imageFileNames = images.ImageLocation;

% Load downloaded lidar data into the workspace
lidarData = fullfile(outputFolder,'WPI_LidarData.mat');
load(lidarData);

% Load calibration data
if ~exist('calib','var')
    load('calib.mat')
end

% Define camera to lidar transformation matrix
camToLidar = calib.extrinsics;
intrinsics = calib.intrinsics;
```

Alternatively, you can use your web browser to first download the datasets to your local disk, and then uncompress the files.

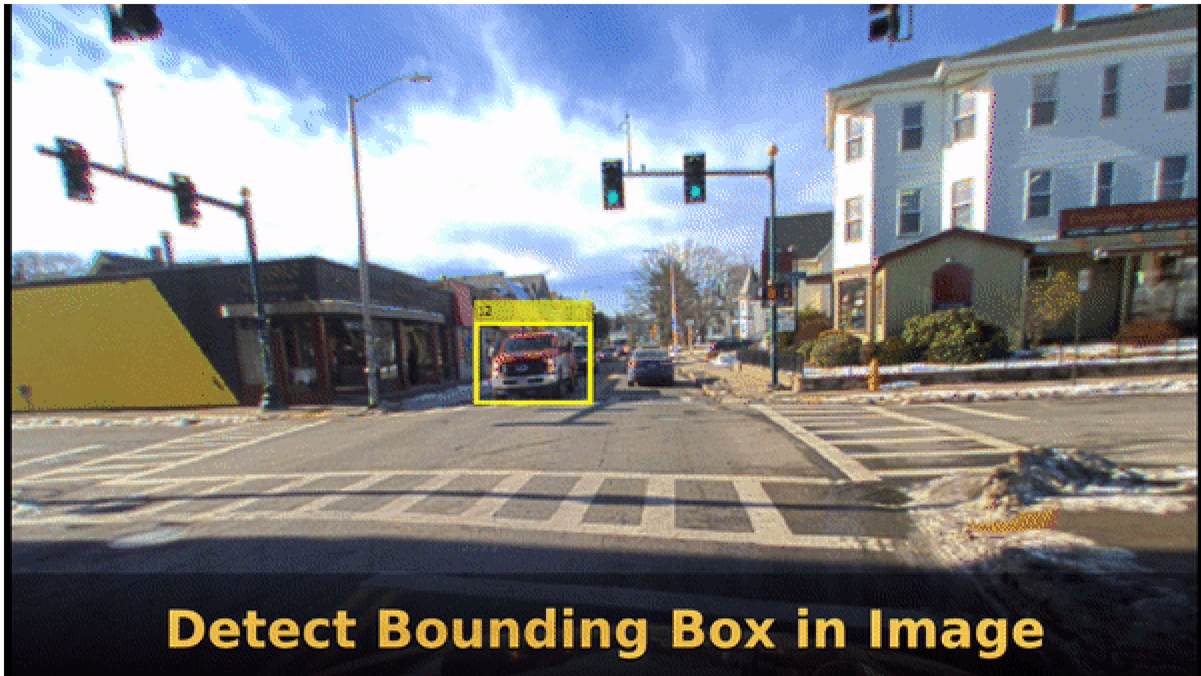
This example uses prelabeled data to serve as ground truth for the 2-D detections from the camera images. These 2-D detections can be generated using deep learning-based object detectors like `vehicleDetectorYOLOv2`, `vehicleDetectorFasterRCNN`, and `vehicleDetectorACF`. For this example, the 2-D detections have been generated using the **Image Labeler** app. These 2-D bounding boxes are vectors of the form: [x y w h], where x and y represent the xy-coordinates of the top-left corner, and w and h represent the width and height of the bounding box respectively.

Read a image frame into the workspace, and display it with the bounding boxes overlaid.

```
load imageGTruth.mat
im = imread(imageFileNames{50});
imBbox = imageGTruth{50};

figure
imshow(im)
showShape('rectangle',imBbox)
```

3-D Region Proposal

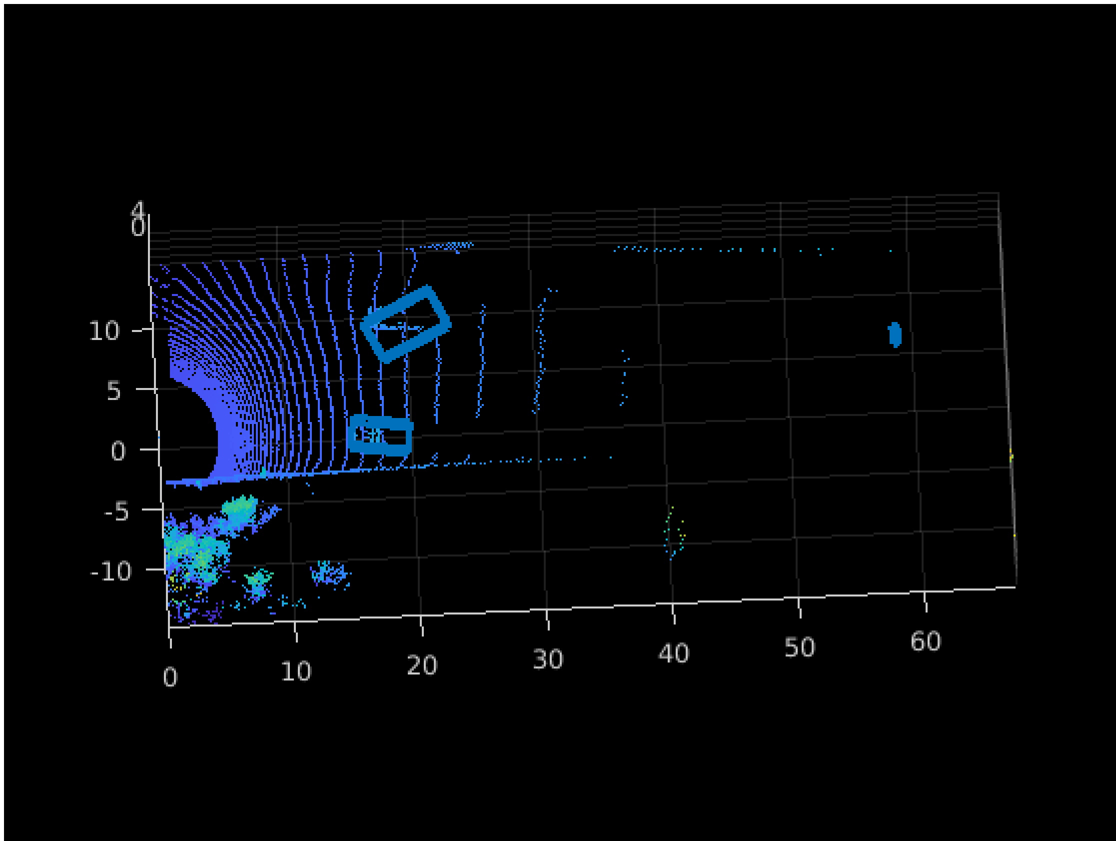
To generate cuboid bounding boxes in lidar from the 2-D rectangular bounding boxes in the image data, a 3-D region is proposed to reduce the search space for bounding box estimation. The corners of each 2-D rectangular bounding box in the image are transformed into 3-D lines using camera intrinsic parameters and camera-to-lidar extrinsic parameters. These 3-D lines form frustum flaring out from the associated 2-D bounding box in the opposite direction of the ego vehicle. The lidar points that fall inside this region are segmented into various clusters based on Euclidean distance. The clusters are fitted with 3-D oriented bounding boxes, and the best cluster is estimated based on the size of these clusters. Estimate the 3-D oriented bounding boxes in a lidar point cloud, based on the 2-D bounding boxes in a camera image, by using the `bboxCameraToLidar` function. This figure shows how 2-D and 3-D bounding boxes relate to each other.



The 3-D cuboids are represented as vectors of the form:[*xcen ycen zcen dimx dimy dimz rotx roty rotz*], where *xcen*, *ycen*, and *zcen* represent the centroid coordinates of the cuboid. *dimx*, *dimy*, and *dimz* represent the length of the cuboid along the *x*-, *y*-, and *z*-axes, and *rotx*, *roty*, and *rotz* represent the rotation ,in degrees, of the cuboid along the *x*-, *y*-, and *z*-axes.

Use ground truth of the image to estimate a 3-D bounding box in the lidar point cloud.

```
pc = lidarData{50};

% Crop point cloud to process only front region
roi = [0 70 -15 15 -3 8];
ind = findPointsInROI(pc,roi);
pc = select(pc,ind);

lidarBbox = bboxCameraToLidar(imBbox,pc,intrinsics, ...
    camToLidar,'ClusterThreshold',2,'MaxDetectionRange',[1,70]);
figure
pcshow(pc.Location,pc.Location(:,3))
showShape('Cuboid',lidarBbox)
view([-2.90 71.59])
```



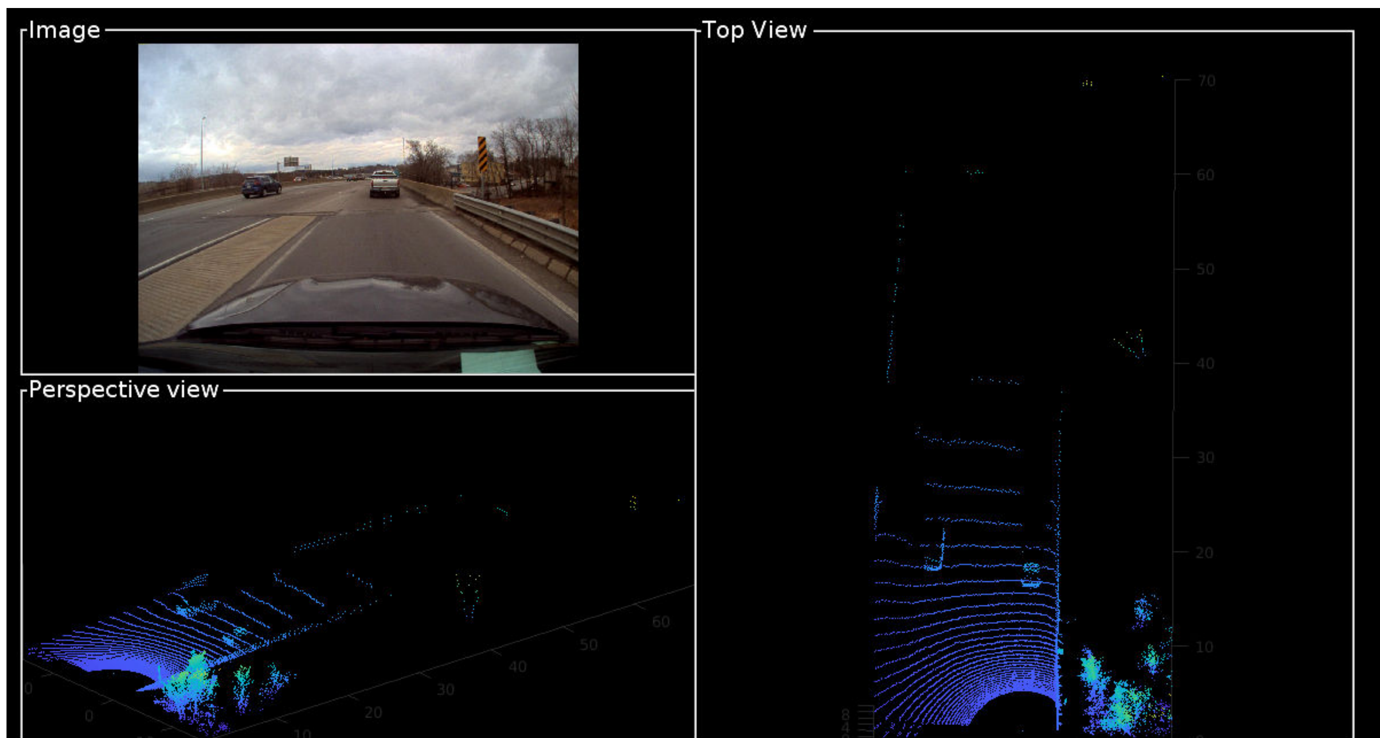
To improve the detected bounding boxes, preprocess the point cloud by removing the ground plane.

Set Up Display

Use the `helperLidarCameraObjectsDisplay` class to visualize the lidar and image data. This visualization provides the capability to view the point cloud, image, 3-D bounding boxes on the point cloud, and 2-D bounding boxes on the image simultaneously. The visualization layout is consists of these windows:

- Image — Visualize an image and associated 2-D bounding boxes
- Perspective View — Visualize the point cloud and associated 3-D bounding boxes in a perspective view
- Top View — Visualize the point cloud and associated 3-D bounding boxes from the top view

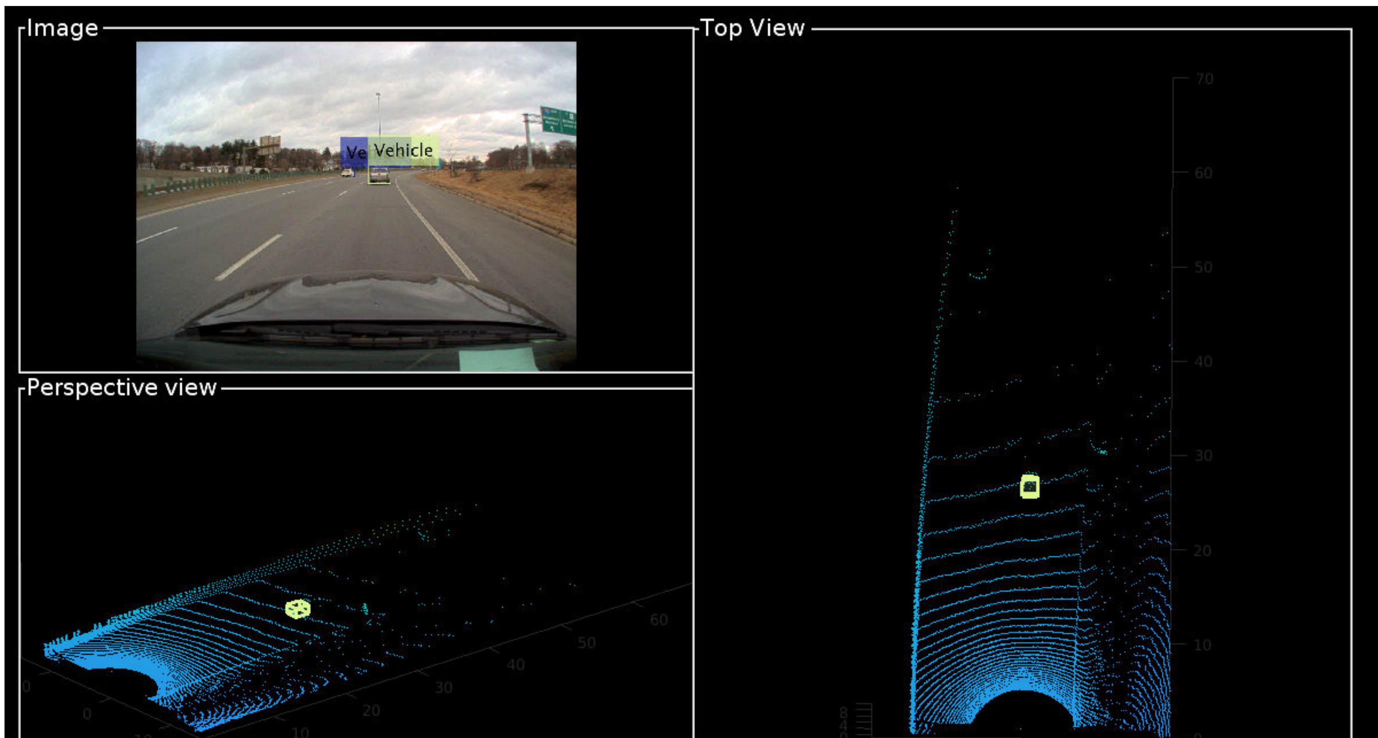
```
% Initialize display
display = helperLidarCameraObjectsDisplay;
initializeDisplay(display)

% Update display with point cloud and image
updateDisplay(display, im, pc)
```



Loop Through Data

Run bboxCameraToLidar on 2-D labels over first 200 frames to generate 3-D cuboids

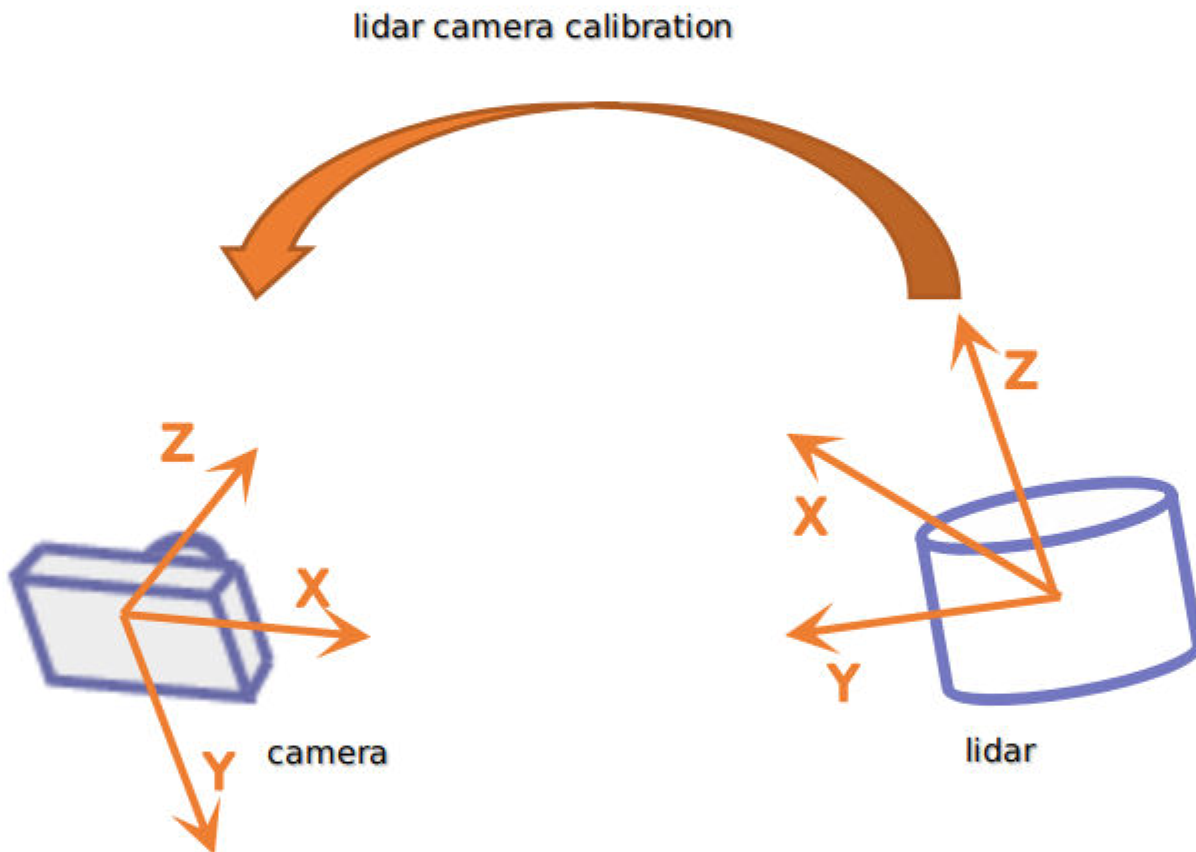
```
for i = 1:200
    % Load point cloud and image
    im = imread(imageFileNames{i});
    pc = lidarData{i};

    % Load image ground truth
    imBbox = imageGTruth{i};

    % Remove ground plane
    groundPtsIndex = segmentGroundFromLidarData(pc,'ElevationAngleDelta',15, ...
        'InitialElevationAngle',10);
    nonGroundPts = select(pc,~groundPtsIndex);

    if imBbox
        [lidarBbox,~,boxUsed] = bboxCameraToLidar(imBbox,nonGroundPts,intrinsics, ...
            camToLidar,'ClusterThreshold',2,'MaxDetectionRange',[1, 70]);
        % Display image with bounding boxes
        im = updateImage(display,im,imBbox);
    end
    % Display point cloud with bounding box
    updateDisplay(display,im,pc);
    updateLidarBbox(display,lidarBbox,boxUsed)
    drawnow
end
```



Detected bounding boxes by using bounding box tracking, such as joint probabilistic data association (JPDA). For more information, see "Track Vehicles Using Lidar: From Point Cloud to Track List" on page 1-110.

Estimate the Distance of Vehicles from the Ego Vehicle

For vehicle safety features such as forward collision warning, accurate measurement of the distance between the ego vehicle and other objects is crucial. A lidar sensor provides the accurate distance of objects from the ego vehicle in 3-D, and it can also be used to create ground truth automatically from 2-D image bounding boxes. To generate ground truth for 2-D bounding boxes, use the `projectLidarPointsOnImage` function to project the points inside the 3-D bounding boxes onto the image. The projected points are associated with 2-D bounding boxes by finding the bounding box with the minimum Euclidean distance from the projected 3-D points. Since the projected points are from lidar to camera, use the inverse of camera-to-lidar extrinsic parameters. This figure illustrates the transformation from lidar to camera.




```
% Initialize display
display = helperLidarCameraObjectsDisplay;
initializeDisplay(display)

% Get lidar to camera matrix
lidarToCam = invert(camToLidar);

% Loop first 200 frames. To loop all frames, replace 200 with numel(imageGTruth)
for i = 1:200
    im = imread(imageFileNames{i});
    pc = lidarData{i};
    imBbox = imageGTruth{i};

    % Remove ground plane
    groundPtsIndex = segmentGroundFromLidarData(pc,'ElevationAngleDelta',15, ...
        'InitialElevationAngle',10);
    nonGroundPts = select(pc,~groundPtsIndex);

    if imBbox
        [lidarBbox,~,boxUsed] = bboxCameraToLidar(imBbox,nonGroundPts,intrinsics, ...
            camToLidar,'ClusterThreshold',2,'MaxDetectionRange',[1, 70]);
        [distance,nearestRect,idx] = helperComputeDistance(imBbox,nonGroundPts,lidarBbox, ...
            intrinsics,lidarToCam);

        % Update image with bounding boxes
        im = updateImage(display,im,nearestRect,distance);
```

```
        updateLidarBbox(display,lidarBbox)
    end

    % Update display
    updateDisplay(display,im,pc)
    drawnow
end
```

Supporting Files

helperComputeDistance

```
function [distance, nearestRect, index] = helperComputeDistance(imBbox, pc, lidarBbox, intrinsic
% helperComputeDistance estimates the distance of 2-D bounding box in a given
% image using 3-D bounding boxes from lidar. It also calculates
% association between 2-D and 3-D bounding boxes

%   Copyright 2020 MathWorks, Inc.

numLidarDetections = size(lidarBbox,1);

nearestRect = zeros(0,4);
distance = zeros(1,numLidarDetections);
index = zeros(0,1);

for i = 1:numLidarDetections
    bboxCuboid = lidarBbox(i,:);

    % Create cuboidModel
    model = cuboidModel(bboxCuboid);
```

```
        % Find points inside cuboid
        ind = findPointsInsideCuboid(model,pc);
        pts = select(pc,ind);

        % Project cuboid points to image
        imPts = projectLidarPointsOnImage(pts,intrinsic,lidarToCam);

        % Find 2-D rectangle corresponding to 3-D bounding box
        [nearestRect(i,:),idx] = findNearestRectangle(imPts,imBbox);
        index(end+1) = idx;
        % Find the distance of the 2-D rectangle
        distance(i) = min(pts.Location(:,1));
    end
    end

    function [nearestRect,idx] = findNearestRectangle(imPts,imBbox)
    numBbox = size(imBbox,1);
    ratio = zeros(numBbox,1);

    % Iterate over all the rectangles
    for i = 1:numBbox
        bbox = imBbox(i,:);
        corners = getCornersFromBbox(bbox);

        % Find overlapping ratio of the projected points and the rectangle
        idx = (imPts(:,1) > corners(1,1)) & (imPts(:,1) < corners(2,1)) & ...
            (imPts(:,2) > corners(1,2)) & (imPts(:,2) < corners(3,1));
        ratio(i) = sum(idx);
    end

    % Get nearest rectangle
    [~,idx] = max(ratio);
    nearestRect = imBbox(idx,:);
    end

    function cornersCamera = getCornersFromBbox(bbox)
    cornersCamera = zeros(4,2);
    cornersCamera(1,1:2) = bbox(1:2);
    cornersCamera(2,1:2) = bbox(1:2) + [bbox(3),0];
    cornersCamera(3,1:2) = bbox(1:2) + bbox(3:4);
    cornersCamera(4,1:2) = bbox(1:2) + [0,bbox(4)];
    end
```

Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network

This example shows how to train a SqueezeSegV2 semantic segmentation network on 3-D organized lidar point cloud data.

SqueezeSegV2 [1 on page 1-0] is a convolutional neural network (CNN) for performing end-to-end semantic segmentation of road objects based on an organized lidar point cloud. The training procedure shown in this example requires 2-D spherical projected images as inputs to the deep learning network.

This example uses a highway scene data set collected using an Ouster OS1 sensor. It contains organized lidar point cloud scans of highway scenes and corresponding ground truth labels for car and truck objects. The size of the data file is approximately 760 MB.

Download Lidar Data Set

Execute this code to download the highway scene data set. This example uses a data set that contains 1617 point clouds stored as `pointCloud` objects in a cell array. Corresponding ground truth data, which is attached to the example, contains bounding box information of cars and trucks in each point cloud.

```
url = 'https://www.mathworks.com/supportfiles/lidar/data/WPI_LidarData.tar.gz';

outputFolder = fullfile(tempdir,'WPI');
lidarDataTarFile = fullfile(outputFolder,'WPI_LidarData.tar.gz');

if ~exist(lidarDataTarFile, 'file')
    mkdir(outputFolder);

    disp('Downloading WPI Lidar driving data (760 MB)...');
    websave(lidarDataTarFile, url);
    untar(lidarDataTarFile,outputFolder);
end

% Check if tar.gz file is downloaded, but not uncompressed.
if ~exist(fullfile(outputFolder, 'WPI_LidarData.mat'), 'file')
    untar(lidarDataTarFile,outputFolder);
end
lidarData = load(fullfile(outputFolder, 'WPI_LidarData.mat'));

groundTruthData = load('WPI_LidarGroundTruth.mat');
```

Note: Depending on your Internet connection, the download process can take some time. The code suspends MATLAB® execution until the download process is complete. Alternatively, you can download the data set to your local disk using your web browser, and then extract `WPI_LidarData`. To use the file you downloaded from the web, change the `outputFolder` variable in the code to the location of the downloaded file.

Download Pretrained Network

Download the pretrained network to avoid having to wait for training to complete. If you want to train the network, set the `doTraining` variable to `true`.

```
doTraining = false;
if ~doTraining && ~exist('trainedSqueezeSegV2Net.mat','file')
```

```
    disp('Downloading pretrained network (2 MB)...');
    pretrainedURL = 'https://www.mathworks.com/supportfiles/lidar/data/trainedSqueezeSegV2Net.ma
    websave('trainedSqueezeSegV2Net.mat', pretrainedURL);
end
```

Prepare Data for Training

Load Lidar Point Clouds and Class Labels

Use the `helperGenerateLidarData` supporting function, attached to this example, to generate training data from the lidar point clouds. The function uses point cloud and bounding box data to create five-channel input images and pixel label images. To create the pixel label images, the function selects points inside the bounding box and labels them with the bounding box class ID. Each training image is specified as a 64-by-1024-by-5 array:

- The height of each image is 64 pixels,
- The width of each image is 1024 pixels.
- Each image has five channels. The five channels specify the 3-D coordinates of the point cloud, intensity, and range: $r = \sqrt{x^2 + y^2 + z^2}$.

A visual representation of the training data follows.

Generate the five-channel training images and pixel label images.

```
imagesFolder = fullfile(outputFolder, 'images');
labelsFolder = fullfile(outputFolder, 'labels');

helperGenerateLidarData(lidarData, groundTruthData, imagesFolder, labelsFolder);
```

```
Preprocessing data 100% complete
```

The five-channel images are saved as MAT files. Labels are saved as PNG files.

Note: Processing can take some time. The code suspends MATLAB® execution until processing is complete.

Create ImageDatastore and PixelLabelDatastore

Use the `imageDatastore` object to extract and store the five channels of the 2-D spherical images using the `helperImageMatReader` supporting function, which is a custom MAT file reader. This function is attached to this example as a supporting file.

```
imds = imageDatastore(imagesFolder, ...
    'FileExtensions', '.mat', ...
    'ReadFcn', @helperImageMatReader);
```

Use the `pixelLabelDatastore` object to store pixel-wise labels from pixel label images. The object maps each pixel label to a class name. In this example, cars and trucks are the only objects of interest; all other pixels are the background. Specify these classes and assign a unique label ID to each class.

```
classNames = [
    "background"
    "car"
    "truck"
    ];

numClasses = numel(classNames);

% Specify label IDs from 1 to the number of classes.
labelIDs = 1 : numClasses;

pxds = pixelLabelDatastore(labelsFolder, classNames, labelIDs);
```

Load and display one of the labeled images by overlaying it on the corresponding intensity image using the `helperDisplayLidarOverlayImage` function, defined in the Supporting Functions on page 1-0 section of this example.

```
imageNumber = 228;

% Point cloud (channels 1, 2, and 3 are for location, and channel 4 is for intensity).
I = readimage(imds, imageNumber);

labelMap = readimage(pxds, imageNumber);
figure;
helperDisplayLidarOverlayImage(I, labelMap, classNames);
title('Ground Truth');
```

Prepare Training, Validation, and Test Sets

Use the `helperPartitionLidarData` supporting function, attached to this example, to split the data into training, validation, and test sets that contain 970, 216, and 431 images, respectively.

```
[imdsTrain, imdsVal, imdsTest, pxdsTrain, pxdsVal, pxdsTest] = ...
    helperPartitionLidarData(imds, pxds);
```

Use the `combine` function to combine the pixel and image datastores for the training and validation data sets.

```
trainingData = combine(imdsTrain, pxdsTrain);
validationData = combine(imdsVal, pxdsVal);
```

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

Augment the training data by using the `transform` function with custom preprocessing operations specified by the `augmentData` function, defined in the Supporting Functions on page 1-0 section of this example. This function randomly flips the multichannel 2-D image and associated labels in the horizontal direction. Apply data augmentation to only the training data set.

```
augmentedTrainingData = transform(trainingData, @(x) augmentData(x));
```

Define Network Architecture

Create a standard SqueezeSegV2 [1 on page 1-0] network by using the `squeezesegv2Layers` function. In the SqueezeSegV2 network, the encoder subnetwork consists of FireModules interspersed with max-pooling layers. This arrangement successively decreases the resolution of the input image. In addition, the SqueezeSegV2 network uses the *focal loss* function to mitigate the effect of the imbalanced class distribution on network accuracy. For more details on how to use the focal loss function in semantic segmentation, see `focalLossLayer`.

Execute this code to create a layer graph that can be used to train the network.

```
inputSize = [64 1024 5];

lgraph = squeezesegv2Layers(inputSize, numClasses);
```

Use the `analyzeNetwork` (Deep Learning Toolbox) function to display an interactive visualization of the network architecture.

```
analyzeNetwork(lgraph)
```

Specify Training Options

Use the `rmsprop` optimization algorithm to train the network. Use the `trainingOptions` function to specify the hyperparameters for `rmsprop`.

```
maxEpochs = 30;
initialLearningRate= 5e-4;
miniBatchSize = 8;
l2reg = 2e-4;
```

```
options = trainingOptions('rmsprop', ...
    'InitialLearnRate', initialLearningRate, ...
    'L2Regularization', l2reg, ...
    'MaxEpochs', maxEpochs, ...
    'MiniBatchSize', miniBatchSize, ...
    'LearnRateSchedule', 'piecewise', ...
    'LearnRateDropFactor', 0.1, ...
    'LearnRateDropPeriod', 10, ...
    'ValidationData', validationData, ...
    'Plots', 'training-progress', ...
    'VerboseFrequency', 20);
```

Note: Reduce `miniBatchSize` to control memory usage when training.

Train Network

The `doTraining` argument is set to `false` (default) to load the pretrained network. You can train the network yourself by setting the `doTraining` argument to `true`. If you train the network, you can use a CPU or a GPU. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher. .

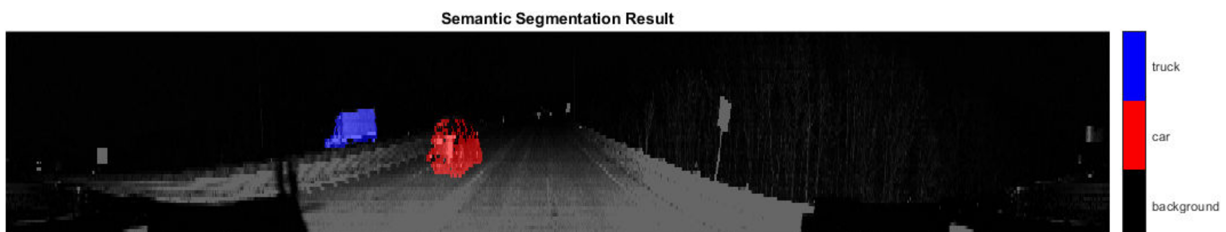
```
if doTraining
    [net, info] = trainNetwork(trainingData, lgraph, options);
else
    pretrainedNetwork = 'trainedSqueezeSegV2Net.mat';
    load(pretrainedNetwork,'net');
end
```

Predict Results on Test Point Cloud

Use the trained network to predict results on a test point cloud and display the segmentation result. First, read a PCD file and convert the point cloud to a five-channel input image. Predict the labels using the trained network.

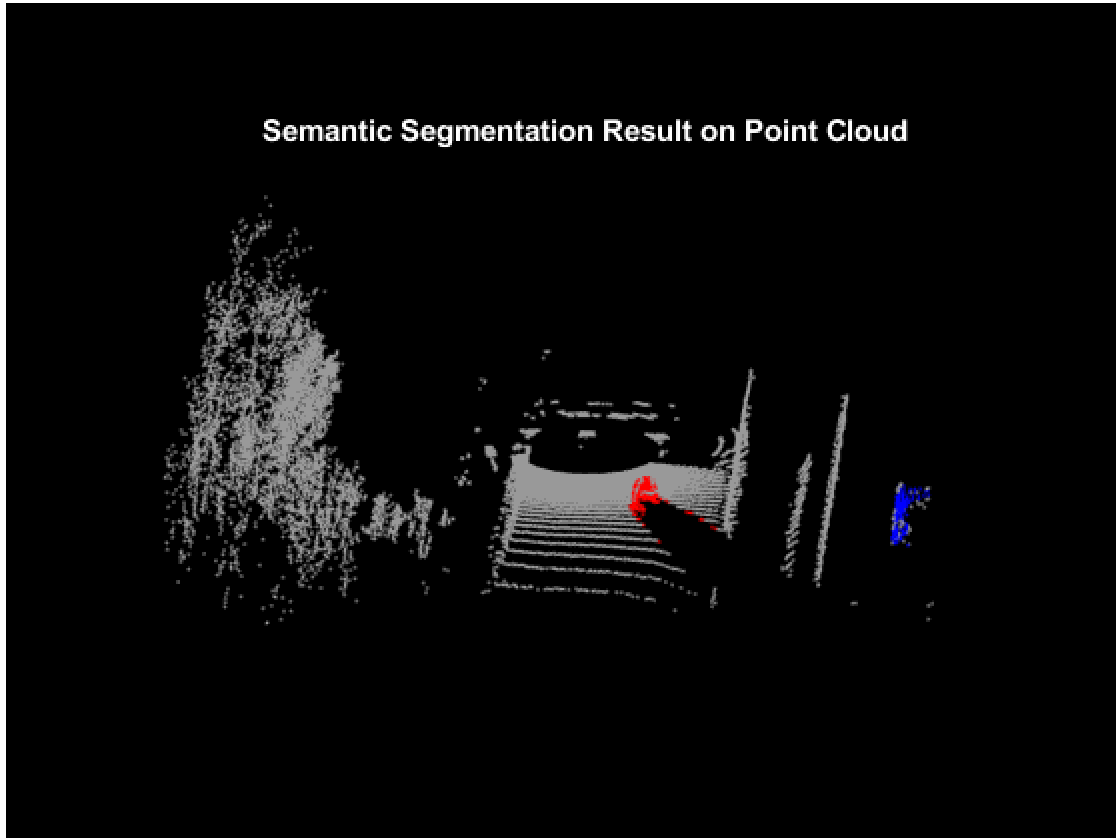
Display the figure with the segmentation as an overlay.

```
ptCloud = pcread('ousterLidarDrivingData.pcd');
I = helperPointCloudToImage(ptCloud);
predictedResult = semanticseg(I, net);

figure;
helperDisplayLidarOverlayImage(I, predictedResult, classNames);
title('Semantic Segmentation Result');
```



Use the `helperDisplayLidarOverlayPointCloud` function, defined in the Supporting Functions on page 1-0 section of this example, to display the segmentation result on the `ptCloud` object.

```
figure;
helperDisplayLidarOverlayPointCloud(ptCloud, predictedResult, numClasses);
view([95.71 24.14])
title('Semantic Segmentation Result on Point Cloud');
```



Evaluate Network

Run the `semanticseg` function on the entire test set to measure the accuracy of the network. Set `MiniBatchSize` to 8 in order to reduce memory usage when segmenting images. You can increase or decrease this value depending on the amount of GPU memory you have on your system.

```
outputLocation = fullfile(tempdir, 'output');
if ~exist(outputLocation,'dir')
    mkdir(outputLocation);
end
pxdsResults = semanticseg(imdsTest, net, ...
    'MiniBatchSize', 8, ...
    'WriteLocation', outputLocation, ...
    'Verbose', false);
```

The `semanticseg` function returns the segmentation results on the test data set as a `PixelLabelDatastore` object. The function writes the actual pixel label data for each test image in the `imdsTest` object to the disk in the location specified by the `'WriteLocation'` argument.

Use the `evaluateSemanticSegmentation` function to compute the semantic segmentation metrics from the test set results.

```
metrics = evaluateSemanticSegmentation(pxdsResults, pxdsTest, 'Verbose', false);
```

You can measure the amount of overlap per class using the intersection-over-union (IoU) metric.

The `evaluateSemanticSegmentation` function returns metrics for the entire data set, for individual classes, and for each test image. To see the metrics at the data set level, use the `metrics.DataSetMetrics` property.

```
metrics.DataSetMetrics
```

```
ans=1×5 table
    GlobalAccuracy    MeanAccuracy    MeanIoU    WeightedIoU    MeanBFScore
    ______________    ____________    _______    ___________    ___________

       0.99376          0.71176       0.63813      0.98912        0.90017
```

The data set metrics provide a high-level overview of network performance. To see the impact each class has on the overall performance, inspect the metrics for each class using the `metrics.ClassMetrics` property.

```
metrics.ClassMetrics
```

```
ans=3×3 table
                  Accuracy      IoU      MeanBFScore
                  ________    _______    ___________

    background    0.99756     0.99463      0.98424
    car           0.71507     0.52999      0.77356
    truck         0.42264     0.38977      0.75395
```

Although the overall network performance is good, the class metrics indicate that you can improve the network performance by training the network on more labeled data containing the car and truck classes.

Supporting Functions

Function to Augment Data

The `augmentData` function randomly flips the spherical image and associated labels in the horizontal direction.

```
function out = augmentData(inp)
% Apply random horizontal flipping.

out = cell(size(inp));

% Randomly flip the five-channel image and pixel labels horizontally.
I = inp{1};
sz = size(I);
tform = randomAffine2d('XReflection',true);
rout = affineOutputView(sz,tform,'BoundsStyle','centerOutput');

out{1} = imwarp(I,tform,'OutputView',rout);
out{2} = imwarp(inp{2},tform,'OutputView',rout);
end
```

Function to Display Lidar Segmentation Map Overlaid on 2-D Spherical Image

The `helperDisplayLidarOverlayImage` function overlays the semantic segmentation map over the intensity channel of the 2-D spherical image. The function also resizes the overlaid image for better visualization.

```
function helperDisplayLidarOverlayImage(lidarImage, labelMap, classNames)
%helperDisplayLidarOverlayImage Overlay labels over the intensity image.
%
%  helperDisplayLidarOverlayImage(lidarImage, labelMap, classNames)
%  displays the overlaid image. lidarImage is a five-channel lidar input.
%  labelMap contains pixel labels and classNames is an array of label
%  names.

% Read the intensity channel from the lidar image.
intensityChannel = uint8(lidarImage(:,:,4));

% Load the lidar color map.
cmap = helperLidarColorMap();

% Overlay the labels over the intensity image.
B = labeloverlay(intensityChannel,labelMap,'Colormap',cmap,'Transparency',0.4);

% Resize for better visualization.
B = imresize(B, 'Scale', [3 1], 'method', 'nearest');
imshow(B);
helperPixelLabelColorbar(cmap, classNames);
end
```

Function to Display Lidar Segmentation Map Overlaid on 3-D Point Cloud

The `helperDisplayLidarOverPointCloud` function overlays the segmentation result over a 3-D organized point cloud.

```
function helperDisplayLidarOverlayPointCloud(ptCloud, labelMap, numClasses)
%helperDisplayLidarOverlayPointCloud Overlay labels over point cloud object.
%
%  helperDisplayLidarOverlayPointCloud(ptCloud, labelMap, numClasses)
%  displays the overlaid pointCloud object. ptCloud is the organized
%  3-D point cloud input. labelMap contains pixel labels and numClasses
%  is the number of predicted classes.

sz = size(labelMap);

% Apply the color red to cars.
carClassCar = zeros(sz(1), sz(2), numClasses, 'uint8');
carClassCar(:,:,1) = 255*ones(sz(1), sz(2), 'uint8');

% Apply the color blue to trucks.
truckClassColor = zeros(sz(1), sz(2), numClasses, 'uint8');
truckClassColor(:,:,3) = 255*ones(sz(1), sz(2), 'uint8');

% Apply the color gray to the background.
backgroundClassColor = 153*ones(sz(1), sz(2), numClasses, 'uint8');

% Extract indices from the labels.
carIndices = labelMap == 'car';
truckIndices = labelMap == 'truck';
```

```
backgroundIndices = labelMap == 'background';

% Extract a point cloud of different classes.
carPointCloud = select(ptCloud, carIndices, 'OutputSize','full');
truckPointCloud = select(ptCloud, truckIndices, 'OutputSize','full');
backgroundPointCloud = select(ptCloud, backgroundIndices, 'OutputSize','full');

% Apply colors to different classes.
carPointCloud.Color = carClassCar;
truckPointCloud.Color = truckClassColor;
backgroundPointCloud.Color = backgroundClassColor;

% Merge and add all the processed point clouds with class information.
coloredCloud = pcmerge(carPointCloud, truckPointCloud, 0.01);
coloredCloud = pcmerge(coloredCloud, backgroundPointCloud, 0.01);

% Plot the colored point cloud. Set an ROI for better visualization.
ax = pcshow(coloredCloud);
set(ax,'XLim',[-35.0 35.0],'YLim',[-32.0 32.0],'ZLim',[-3.0 8.0], ...
    'XColor','none','YColor','none','ZColor','none');
set(get(ax,'parent'), 'units','normalized');
end
```

Function to Define Lidar Colormap

The `helperLidarColorMap` function defines the colormap used by the lidar data set.

```
function cmap = helperLidarColorMap()

cmap = [
    0.00  0.00   0.00  % background
    0.98  0.00   0.00  % car
    0.00  0.00   0.98  % truck
    ];

end
```

Function To Display Pixel Label Colorbar

The `helperPixelLabelColorbar` function adds a colorbar to the current axis. The colorbar is formatted to display the class names with the color.

```
function helperPixelLabelColorbar(cmap, classNames)

colormap(gca, cmap);

% Add a colorbar to the current figure.
c = colorbar('peer', gca);

% Use class names for tick marks.
c.TickLabels = classNames;
numClasses = size(classNames, 1);

% Center tick labels.
c.Ticks = 1/(numClasses * 2):1/numClasses:1;

% Remove tick marks.
c.TickLength = 0;
end
```

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.

Code Generation for Lidar Point Cloud Segmentation Network

This example shows how to generate CUDA® MEX code for a deep learning network for lidar semantic segmentation. This example uses a pretrained SqueezeSegV2 [1] network that can segment organized lidar point clouds belonging to three classes (*background*, *car*, and *truck*). For information on the training procedure for the network, see "Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network" on page 1-57. The generated MEX code takes a point cloud as input and performs prediction on the point cloud by using the `DAGNetwork` object for the SqueezeSegV2 network.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- NVIDIA TensorRT library.
- Environment variables for the compilers and libraries. For details, see "Third-Party Hardware" (GPU Coder) and "Setting Up the Prerequisite Products" (GPU Coder).

Verify GPU Environment

To verify that the compilers and libraries for running this example are set up correctly, use the `coder.checkGpuInstall` (GPU Coder) function.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Segmentation Network

SqueezeSegV2 is a convolutional neural network (CNN) designed for the semantic segmentation of organized lidar point clouds. It is a deep encoder-decoder segmentation network trained on a lidar data set and imported into MATLAB® for inference. In SqueezeSegV2, the encoder subnetwork consists of convolution layers that are interspersed with max-pooling layers. This arrangement successively decreases the resolution of the input image. The decoder subnetwork consists of a series of transposed convolution layers, which successively increase the resolution of the input image. In addition, the SqueezeSegV2 network mitigates the impact of missing data by including context aggregation modules (CAMs). A CAM is a convolutional subnetwork with filterSize of value [7, 7] that aggregates contextual information from a larger receptive field, which improves the robustness of the network to missing data. The SqueezeSegV2 network in this example is trained to segment points belonging to three classes (background, car, and truck).

For more information on training a semantic segmentation network in MATLAB® by using the Mathworks lidar dataset, see "Lidar Point Cloud Semantic Segmentation Using PointSeg Deep Learning Network" on page 1-12.

Download the pretrained SqueezeSegV2 Network.

```
net = getSqueezeSegV2Net();
```

```
Downloading pretrained SqueezeSegV2 (2 MB)...
```

The DAG network contains 238 layers, including convolution, ReLU, and batch normalization layers, and a focal loss output layer. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` (Deep Learning Toolbox) function.

```
analyzeNetwork(net);
```

`squeezesegv2_predict` Entry-Point Function

The `squeezesegv2_predict.m` entry-point function, which is attached to this example, takes a point cloud as input and performs prediction on it by using the deep learning network saved in the `SqueezeSegV2Net.mat` file. The function loads the network object from the `SqueezeSegV2Net.mat` file into a persistent variable `mynet` and reuses the persistent variable in subsequent prediction calls.

```
type('squeezesegv2_predict.m');
```

```
function out = squeezesegv2_predict(in)
%#codegen

% A persistent object mynet is used to load the DAG network object. At
% the first call to this function, the persistent object is constructed and
% setup. When the function is called subsequent times, the same object is
% reused to call predict on inputs, thus avoiding reconstructing and
% reloading the network object.

% Copyright 2020 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('SqueezeSegV2Net.mat');
end

% pass in input
out = predict(mynet,in);
```

Generate CUDA MEX Code

To generate CUDA MEX code for the `squeezesegv2_predict.m` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a `CuDNN` deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command, specifying an input size of [64, 1024, 5]. This value corresponds to the size of the input layer of the SqueezeSegV2 network.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
```

```
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg squeezesegv2_predict -args {ones(64,1024,5,'uint8')} -report
```

```
Code generation successful: View report
```

To generate CUDA C++ code that takes advantage of NVIDIA TensorRT libraries, in the code, specify `coder.DeepLearningConfig('tensorrt')` instead of `coder.DeepLearningConfig('cudnn')`.

For information on how to generate MEX code for deep learning networks on Intel® processors, see “Code Generation for Deep Learning Networks with MKL-DNN” (MATLAB Coder).

Prepare Data

Load an organized test point cloud in MATLAB®. Convert the point cloud to a five-channel image for prediction.

```
ptCloud = pcread('ousterLidarDrivingData.pcd');
I = pointCloudToImage(ptCloud);

% Examine converted data
whos I
```

```
  Name         Size                Bytes  Class     Attributes

  I          64x1024x5            327680  uint8
```

The image has five channels. The *(x,y,z)* point coordinates comprise the first three channels. The fourth channel contains the lidar intensity measurement. The fifth channel contains the range information, which is computed as $r = \sqrt{x^2 + y^2 + z^2}$.

Visualize intensity channel of the image.

```
intensityChannel = I(:,:,4);

figure;
imshow(intensityChannel);
title('Intensity Image');
```

Run Generated MEX on Data

Call `squeezesegv2_predict_mex` on the five-channel image.

```
predict_scores = squeezesegv2_predict_mex(I);
```

The `predict_scores` variable is a three-dimensional matrix that has three channels corresponding to the pixel-wise prediction scores for every class. Compute the channel by using the maximum prediction score to get the pixel-wise labels

```
[~,argmax] = max(predict_scores,[],3);
```

Overlay the segmented labels on the intensity channel image and display the segmented region. Resize the segmented output and add a colorbar for better visualization.

```
classes = [
    "background"
    "car"
    "truck"
    ];

cmap = lidarColorMap();
SegmentedImage = labeloverlay(intensityChannel,argmax,'ColorMap',cmap);
SegmentedImage = imresize(SegmentedImage, 'Scale', [2 1], 'method', 'nearest');
figure;
imshow(SegmentedImage);

N = numel(classes);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(classes),'Ticks',ticks,'TickLength',0,'TickLabelInterpreter','none
colormap(cmap)
title('Semantic Segmentation Result');
```

Run Generated MEX Code on Point Cloud Sequence

Read an input point cloud sequence. The sequence contains 10 organized `pointCloud` frames collected using an Ouster OS1 lidar sensor. The input data has a height of 64 and a width of 1024, so each pointCloud object is of size 64-by-1024.

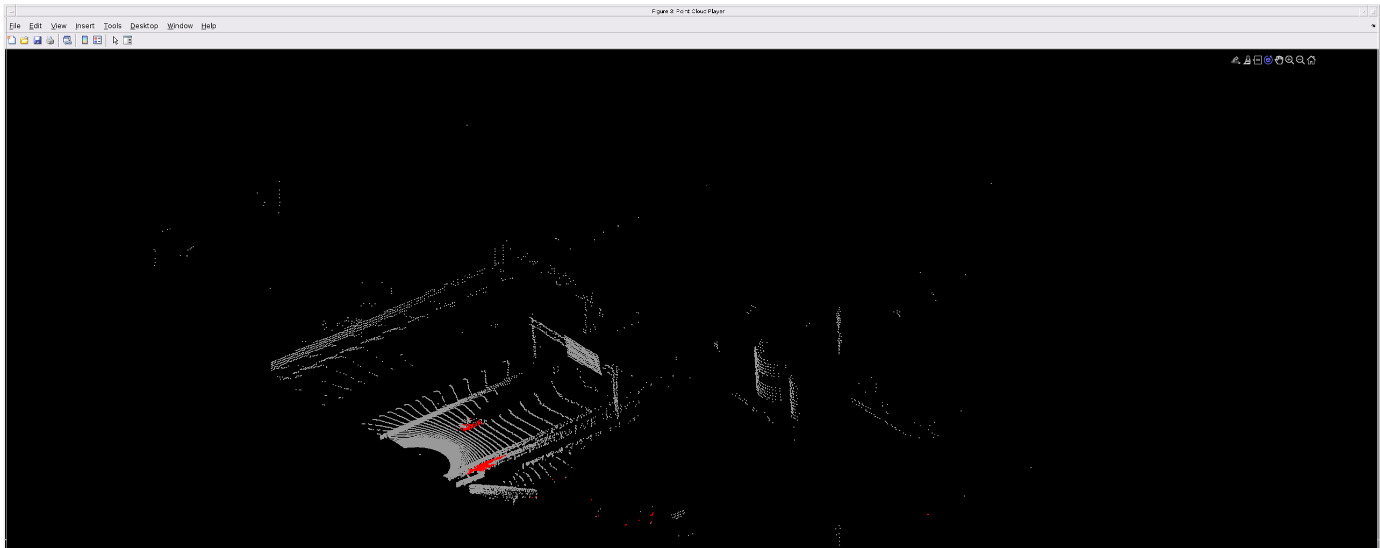
```
dataFile = 'highwaySceneData.mat';

% Load data in workspace.
load(dataFile);
```

Setup different colors to visualize point-wise labels for different classes of interest.

```
% Apply the color red to cars.
carClassCar = zeros(64, 1024, 3, 'uint8');
carClassCar(:,:,1) = 255*ones(64, 1024, 'uint8');

% Apply the color blue to trucks.
truckClassColor = zeros(64, 1024, 3, 'uint8');
truckClassColor(:,:,3) = 255*ones(64, 1024, 'uint8');

% Apply the color gray to background.
backgroundClassColor = 153*ones(64, 1024, 3, 'uint8');
```

Set the `pcplayer` function properties to display the sequence and the output predictions. Read the input sequence frame by frame and detect classes of interest using the model.

```
xlimits = [0 120.0];
ylimits = [-80.7 80.7];
zlimits = [-8.4 27];

player = pcplayer(xlimits, ylimits, zlimits);
set(get(player.Axes,'parent'), 'units','normalized','outerposition',[0 0 1 1]);
zoom(get(player.Axes,'parent'),2);
set(player.Axes,'XColor','none','YColor','none','ZColor','none');

for i = 1 : numel(inputData)
    ptCloud = inputData{i};

    % Convert point cloud to five-channel image for prediction.
    I = pointCloudToImage(ptCloud);

    % Call squeezesegv2_predict_mex on the 5-channel image.
    predict_scores = squeezesegv2_predict_mex(I);

    % Convert the numeric output values to categorical labels.
    [~,predictedOutput] = max(predict_scores,[],3);
    predictedOutput = categorical(predictedOutput, 1:3, classes);

    % Extract the indices from labels.
    carIndices = predictedOutput == 'car';
    truckIndices = predictedOutput == 'truck';
    backgroundIndices = predictedOutput == 'background';

    % Extract a point cloud for each class.
    carPointCloud = select(ptCloud, carIndices, 'OutputSize','full');
    truckPointCloud = select(ptCloud, truckIndices, 'OutputSize','full');
    backgroundPointCloud = select(ptCloud, backgroundIndices, 'OutputSize','full');

    % Fill the colors to different classes.
    carPointCloud.Color = carClassCar;
    truckPointCloud.Color = truckClassColor;
    backgroundPointCloud.Color = backgroundClassColor;

    % Merge and add all the processed point clouds with class information.
    coloredCloud = pcmerge(carPointCloud, truckPointCloud, 0.01);
    coloredCloud = pcmerge(coloredCloud, backgroundPointCloud, 0.01);

    % View the output.
    view(player, coloredCloud);
    drawnow;
end
```



Helper Functions

The helper functions used in this example follow.

```
type pointCloudToImage.m
```

```
function image = pointCloudToImage(ptcloud)
%pointCloudToImage Converts organized 3-D point cloud to 5-channel
%   2-D image.

image = ptcloud.Location;
image(:,:,4) = ptcloud.Intensity;
rangeData = iComputeRangeData(image(:,:,1),image(:,:,2),image(:,:,3));
image(:,:,5) = rangeData;

% Cast to uint8.
image = uint8(image);
end

%--------------------------------------------------------------------------
function rangeData = iComputeRangeData(xChannel,yChannel,zChannel)
rangeData = sqrt(xChannel.*xChannel+yChannel.*yChannel+zChannel.*zChannel);
end
```

```
type lidarColorMap.m
```

```
function cmap = lidarColorMap()

cmap = [
   0.00  0.00   0.00  % background
   0.98  0.00   0.00  % car
   0.00  0.00   0.98  % truck
   ];
end
```

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.” Preprint, submitted September 22, 2018. http://arxiv.org/abs/1809.08495.

Lidar 3-D Object Detection Using PointPillars Deep Learning

This example shows how to train a PointPillars network for object detection in point clouds.

Point cloud data is acquired by a variety of sensors, such as lidar sensors, radar sensors, and depth cameras. These sensors capture 3-D position information about objects in a scene, which is useful for many applications in autonomous driving and augmented reality. However, training robust detectors with point cloud data is challenging because of the sparsity of data per object, object occlusions, and sensor noise. Deep learning techniques have been shown to address many of these challenges by learning robust feature representations directly from point cloud data. One deep learning technique for 3-D object detection is PointPillars [1 on page 1-0]. Using a similar architecture to PointNet, the PointPillars network extracts dense, robust features from sparse point clouds called pillars, then uses a 2-D deep learning network with a modified SSD object detection network to get joint bounding box and class predictions.

This example uses a data set of highway scenes collected using the Ouster OS1 lidar sensor.

Download Pretrained Network

Download the pretrained network from the given URL using the helper function `downloadPretrainedPointPillarsNet`, defined at the end of this example. The pretrained model allows you to run the entire example without having to wait for training to complete. If you want to train the network, set the `doTraining` variable to `true`.

```
outputFolder = fullfile(tempdir,'WPI');
pretrainedNetURL = 'https://ssd.mathworks.com/supportfiles/lidar/data/trainedPointPillars.zip';

doTraining = false;
if ~doTraining
    net = downloadPretrainedPointPillarsNet(outputFolder, pretrainedNetURL);
end
```

Load Data

Download the highway scenes dataset from the given URL using the helper function `downloadWPIData`, defined at the end of this example. The data set contains 1617 full 360-degree view organized lidar point cloud scans of highway scenes and corresponding labels for car and truck objects.

```
lidarURL = 'https://www.mathworks.com/supportfiles/lidar/data/WPI_LidarData.tar.gz';
lidarData = downloadWPIData(outputFolder, lidarURL);
```

Note: Depending on your Internet connection, the download process can take some time. The code suspends MATLAB® execution until the download process is complete. Alternatively, you can download the data set to your local disk using your web browser and extract the file. If you do so, change the `outputFolder` variable in the code to the location of the downloaded file.

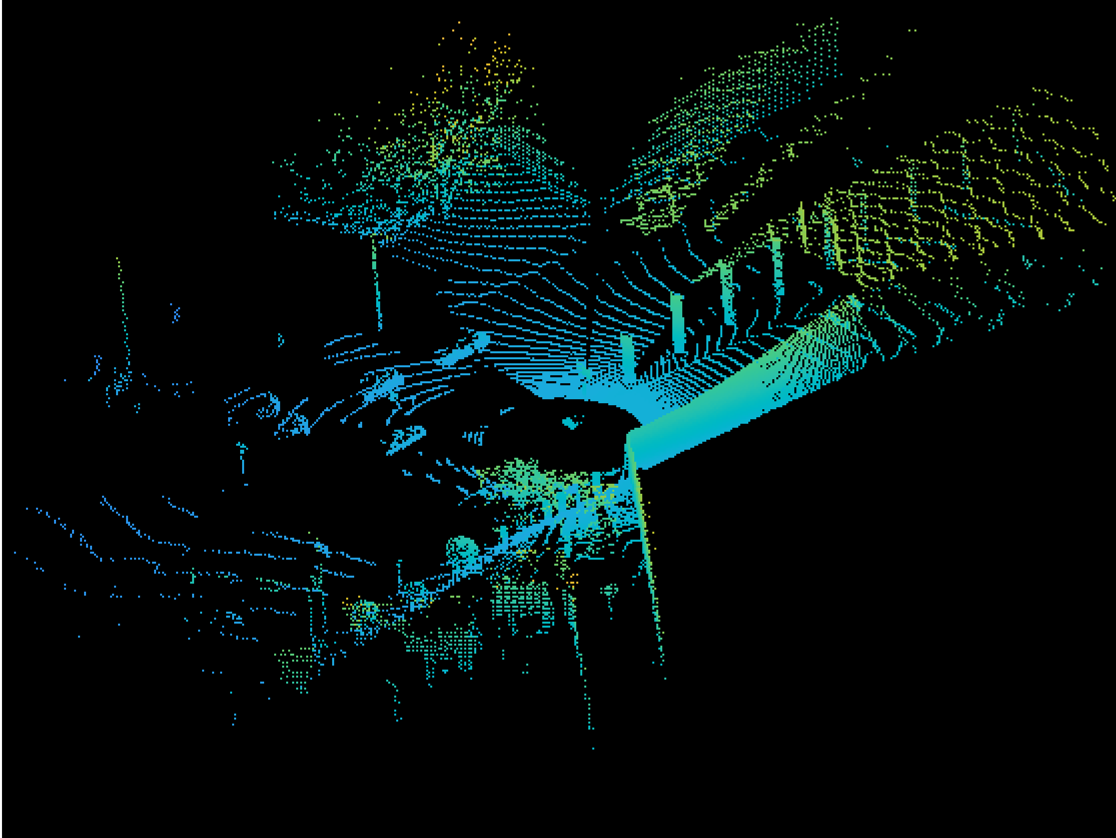
Load the 3-D bounding box labels.

```
load('WPI_LidarGroundTruth.mat','bboxGroundTruth');
Labels = timetable2table(bboxGroundTruth);
Labels = Labels(:,2:end);
```

Display the full-view point cloud.

```
figure
ax = pcshow(lidarData{1,1}.Location);
```

```
set(ax,'XLim',[-50 50],'YLim',[-40 40]);
zoom(ax,2.5);
axis off;
```



Preprocess Data

This example crops full-view point clouds to front-view point clouds using the standard parameters [1 on page 1-0]. These parameters help choose the size of the input passed to the network. Selecting a smaller range of point clouds along x, y, and z-axis helps detect objects that are closer to the origin and also decreases the overall training time of the network.

```
xMin = 0.0;     % Minimum value along X-axis.
yMin = -39.68;  % Minimum value along Y-axis.
zMin = -5.0;    % Minimum value along Z-axis.
xMax = 69.12;   % Maximum value along X-axis.
yMax = 39.68;   % Maximum value along Y-axis.
zMax = 5.0;     % Maximum value along Z-axis.
xStep = 0.16;   % Resolution along X-axis.
yStep = 0.16;   % Resolution along Y-axis.
dsFactor = 2.0; % Downsampling factor.

% Calculate the dimensions for pseudo-image.
Xn = round(((xMax - xMin) / xStep));
Yn = round(((yMax - yMin) / yStep));

% Define pillar extraction parameters.
gridParams = {{xMin,yMin,zMin},{xMax,yMax,zMax},{xStep,yStep,dsFactor},{Xn,Yn}};
```

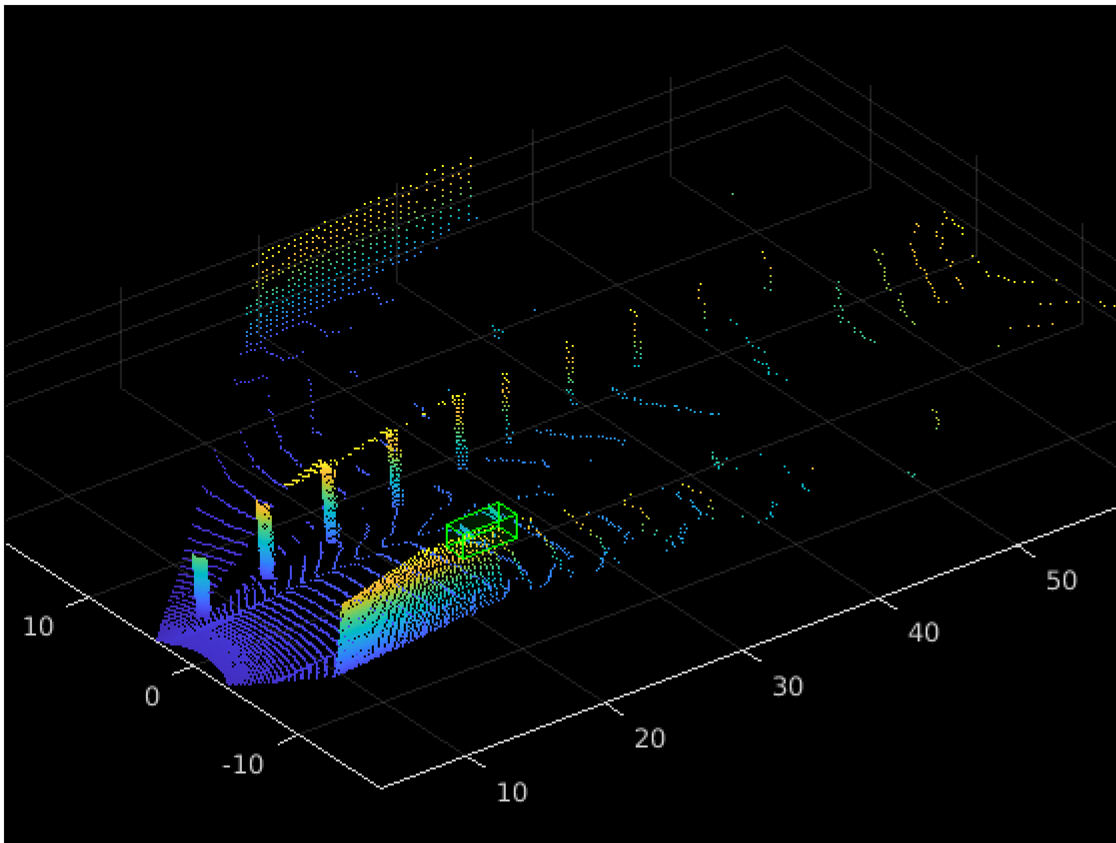
Crop the front view of the point clouds using the `createFrontViewFromLidarData` helper function, attached to this example as a supporting file, and lidar camera calibration values. Skip this step if your data set contains front view point clouds.

```
% Load the calibration parameters.
fview = load('calibrationValues.mat');
[inputPointCloud, boxLabels] = createFrontViewFromLidarData(lidarData, Labels, gridParams, fview
```

```
Processing data 100% complete
```

Display the cropped point cloud.

```
figure
ax1 = pcshow(inputPointCloud{1,1}.Location);
gtLabels = boxLabels.car(1,:);
showShape('cuboid', gtLabels{1,1}, 'Parent', ax1, 'Opacity', 0.1, 'Color', 'green','LineWidth',0
zoom(ax1,2);
```



Create FileDatastore and BoxLabelDatastore Objects for Training

Split the data set into a training set for training the network, and a test set for evaluating the network. Select 70% of the data for training. Use the rest for evaluation.

```
rng(1);
shuffledIndices = randperm(size(inputPointCloud,1));
idx = floor(0.7 * length(shuffledIndices));

trainData = inputPointCloud(shuffledIndices(1:idx),:);
```

```
testData = inputPointCloud(shuffledIndices(idx+1:end),:);

trainLabels = boxLabels(shuffledIndices(1:idx),:);
testLabels = boxLabels(shuffledIndices(idx+1:end),:);
```

For easy access with datastores, save the training data as PCD files. Skip this step if your point cloud data format is supported by `pcread` function.

```
dataLocation = fullfile(outputFolder,'InputData');
saveptCldToPCD(trainData,dataLocation);
```

```
Processing data 100% complete
```

Create `fileDatastore` to load PCD files using the `pcread` function.

```
lds = fileDatastore(dataLocation,'ReadFcn',@(x) pcread(x));
```

Create `boxLabelDatastore` for loading the 3-D bounding box labels.

```
bds = boxLabelDatastore(trainLabels);
```

Use the `combine` function to combine the point clouds and 3-D bounding box labels into a single datastore for training.

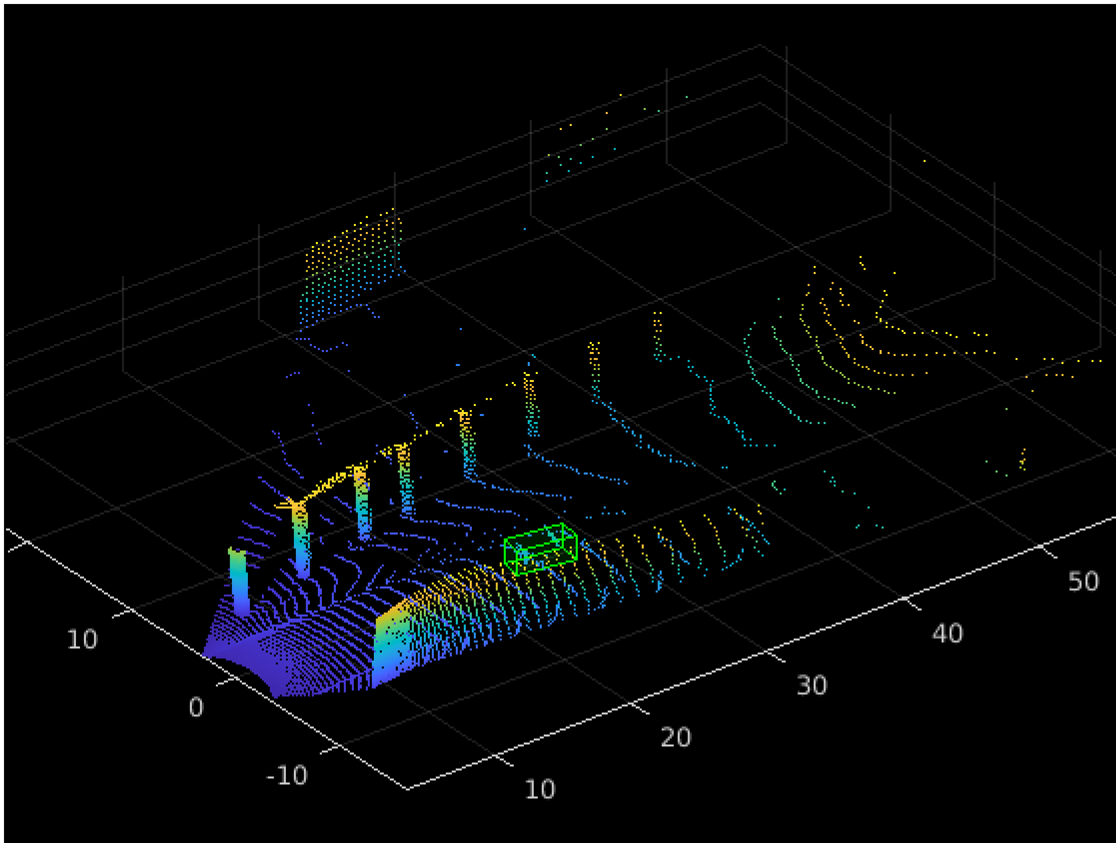
```
cds = combine(lds,bds);
```

Data Augmentation

This example uses ground truth data augmentation and several other global data augmentation techniques to add more variety to the training data and corresponding boxes.

Read and display a point cloud before augmentation.

```
augData = read(cds);
augptCld = augData{1,1};
augLabels = augData{1,2};
figure;
ax2 = pcshow(augptCld.Location);
showShape('cuboid', augLabels, 'Parent', ax2, 'Opacity', 0.1, 'Color', 'green','LineWidth',0.5);
zoom(ax2,2);
```




```
reset(cds);
```

Use `generateGTDataForAugmentation` helper function, attached to this example as a supporting file, to extract all ground truth bounding boxes from training data.

```
gtData = generateGTDataForAugmentation(trainData,trainLabels);
```

Use the `groundTruthDataAugmentation` helper function, attached to this example as a supporting file, to randomly add a fixed number of car class objects to every point cloud.

Use the `transform` function to apply ground truth and custom data augmentations to the training data.

```
cdsAugmented = transform(cds,@(x) groundTruthDataAugmenation(x,gtData));
```

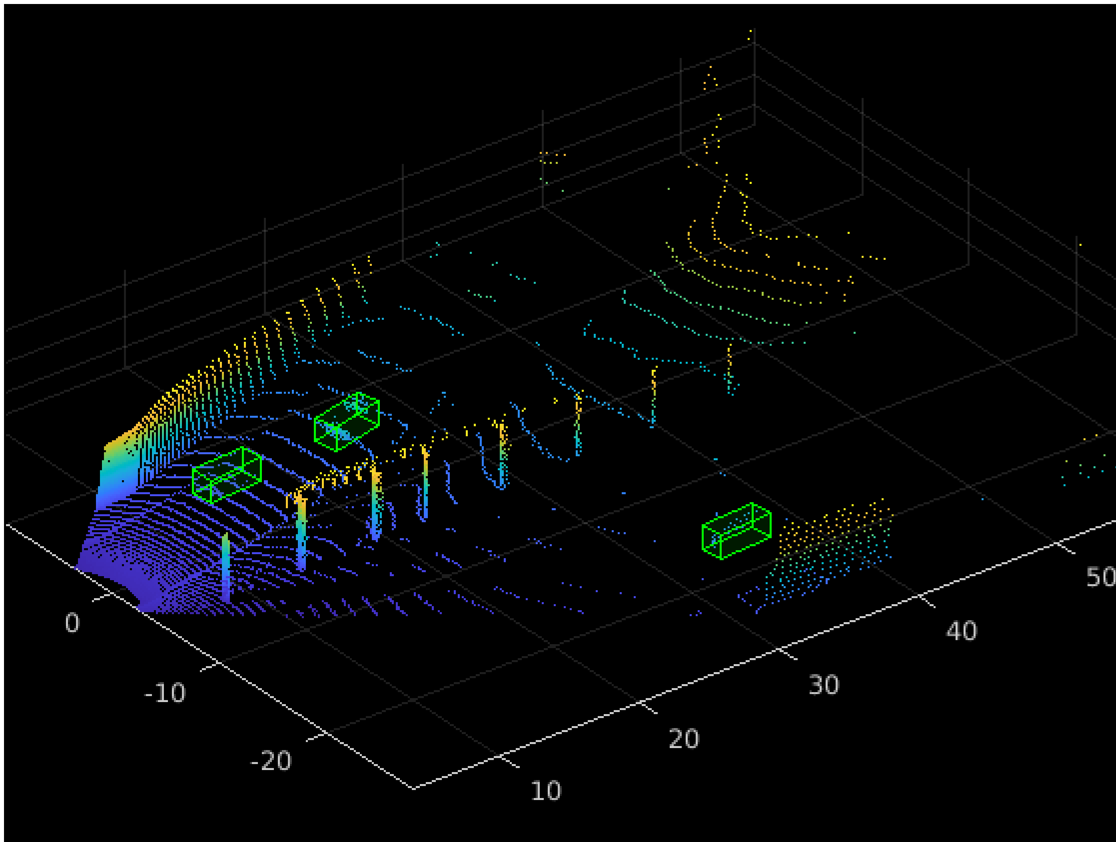
In addition, apply the following data augmentations to every point cloud.

- Random flipping along the x-axis
- Random scaling by 5 percent
- Random rotation along z-axis from [-pi/4, pi/4]
- Random translation by [0.2, 0.2, 0.1] meters along XYZ axis respectively

```
cdsAugmented = transform(cdsAugmented,@(x) augmentData(x));
```

Display an augmented point cloud along with ground truth augmented boxes.

```
augData = read(cdsAugmented);
augptCld = augData{1,1};
augLabels = augData{1,2};
figure;
ax3 = pcshow(augptCld(:,1:3));
showShape('cuboid', augLabels, 'Parent', ax3, 'Opacity', 0.1, 'Color', 'green','LineWidth',0.5);
zoom(ax3,2);
```




```
reset(cdsAugmented);
```

Extract Pillar Information from Point Clouds

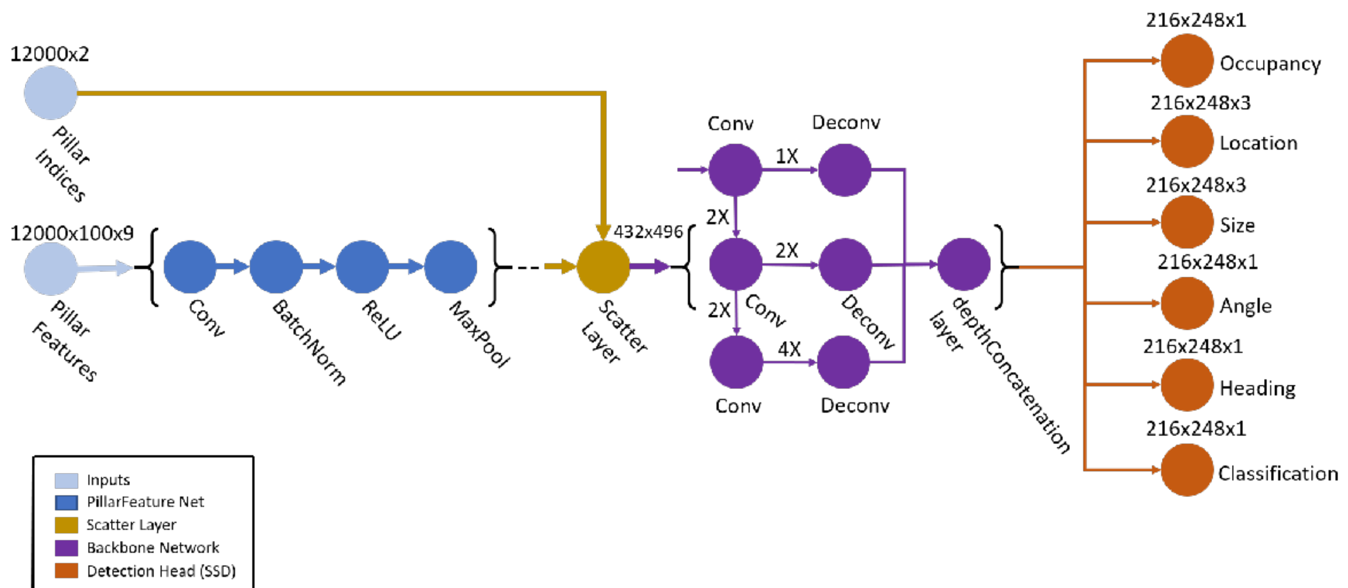
Convert 3-D point cloud to 2-D representation, to apply 2-D convolution architecture to point clouds for faster processing. Use the `transform` function with `createPillars` helper function, attached to this example as a supporting file, to create pillar features and pillar indices from the point clouds. The helper function performs the following operations:

- Discretize 3-D point clouds into evenly spaced grids in the x-y plane to create a set vertical columns called pillars.
- Select prominent pillars (P) based on number of points per pillar (N).
- Compute distance to the arithmetic mean of all points in the pillar.
- Compute offset from the pillar center.
- Use x, y, z location, intensity, distance and offset values to create nine dimensional (9-D) vector for each point in the pillar.

```
% Define number of prominent pillars.
P = 12000;

% Define number of points per pillar.
N = 100;
cdsTransformed = transform(cdsAugmented,@(x) createPillars(x,gridParams,P,N));
```

Define PointPillars Network

The PointPillars network uses a simplified version of PointNet network, that takes pillar features as input. For each pillar feature, a linear layer is applied followed by batch normalization and ReLU layers. Finally, max-pooling operation over the channels is applied to get high level encoded features. These encoded features are scattered back to the original pillar locations to create a pseudo-image using custom layer `helperscatterLayer`, attached to this example as a supporting file. The pseudo-image is then processed with a 2-D convolutional backbone followed by various SSD detection heads to predict the 3-D bounding boxes along with its classes.



Define the anchor box dimensions based on the classes to detect. Typically, these dimensions are the means of all the bounding box values in the training set [1 on page 1-0]. The anchor boxes are defined in the format *{length, width, height, z-centre, yaw angle}*.

```
anchorBoxes = {{3.9, 1.6, 1.56, -1.78, 0}, {3.9, 1.6, 1.56, -1.78, pi/2}};
numAnchors = size(anchorBoxes,2);
classNames = trainLabels.Properties.VariableNames;
```

Next create the PointPillars network using the helper function `pointpillarNetwork`, attached to this example as a supporting file.

```
lgraph = pointpillarNetwork(numAnchors,gridParams,P,N);
```

Specify Training Options

Specify the following training options.

- Set the number of epochs to 160.
- Set the mini batch size as 2. This should be set depending on the available memory.
- Set the learning rate to 0.0002.
- Set `learnRateDropPeriod` to 15. This parameter denotes the number of epochs to drop the learning rate after based on the formula $learningRate \times (iteration \,\%\, learnRateDropPeriod) \times learnRateDropFactor$.
- Set `learnRateDropFactor` to 0.8. This parameter denotes the rate by which to drop the learning rate after each `learnRateDropPeriod`.
- Set the gradient decay factor to 0.9.
- Set the squared gradient decay factor to 0.999.
- Initialize the average of gradients to [ ]. This is used by the Adam optimizer.
- Initialize the average of squared gradients to [ ]. This is used by the Adam optimizer.

```
numEpochs = 160;
miniBatchSize = 2;
learningRate = 0.0002;
learnRateDropPeriod = 15;
learnRateDropFactor = 0.8;
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.999;
trailingAvg = [];
trailingAvgSq = [];
```

Train Model

Train on a GPU, if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher. To automatically detect if you have a GPU available, set `executionEnvironment` to `"auto"`. If you do not have a GPU, or do not want to use one for training, set `executionEnvironment` to `"cpu"`. To ensure the use of a GPU for training, set `executionEnvironment` to `"gpu"`.

Next, create a `minibatchqueue` (Deep Learning Toolbox) to load the data in batches of `miniBatchSize` during training.

```
executionEnvironment = "auto";
if canUseParallelPool
    dispatchInBackground = true;
else
    dispatchInBackground = false;
end

mbq = minibatchqueue(cdsTransformed,3,...
                     "MiniBatchSize",miniBatchSize,...
                     "OutputEnvironment",executionEnvironment,...
                     "MiniBatchFcn",@(features,indices,boxes,labels) createBatchData(features,inc
                     "MiniBatchFormat",["SSCB","SSCB",""],...
                     "DispatchInBackground",dispatchInBackground);
```

To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` (Deep Learning Toolbox) object. Then create the training progress plotter using the helper function `configureTrainingProgressPlotter`, defined at the end of this example.

Finally, specify the custom training loop. For each iteration:

- Read the point clouds and ground truth boxes from the `minibatchqueue` (Deep Learning Toolbox) using the `next` (Deep Learning Toolbox) function.
- Evaluate the model gradients using `dlfeval` (Deep Learning Toolbox) and the `modelGradients` function. The `modelGradients` helper function, defined at the end of example, returns the gradients of the loss with respect to the learnable parameters in `net`, the corresponding mini-batch loss, and the state of the current batch.
- Update the network parameters using the `adamupdate` (Deep Learning Toolbox) function.
- Update the state parameters of `net`.
- Update the training progress plot.

```
if doTraining
    % Convert layer graph to dlnetwork.
    net = dlnetwork(lgraph);

    % Initialize plot.
    fig = figure;
    lossPlotter = configureTrainingProgressPlotter(fig);
    iteration = 0;

    % Custom training loop.
    for epoch = 1:numEpochs

        % Reset datastore.
        reset(mbq);

        while(hasdata(mbq))
            iteration = iteration + 1;

            % Read batch of data.
            [pillarFeatures, pillarIndices, boxLabels] = next(mbq);

            % Evaluate the model gradients and loss using dlfeval and the modelGradients functio
            [gradients, loss, state] = dlfeval(@modelGradients, net, pillarFeatures, pillarIndice
                                                gridParams, anchorBoxes, executionEnvironment);

            % Do not update the network learnable parameters if NaN values
            % are present in gradients or loss values.
            if checkForNaN(gradients,loss)
                continue;
            end

            % Update the state parameters of dlnetwork.
            net.State = state;

            % Update the network learnable parameters using the Adam
            % optimizer.
            [net.Learnables, trailingAvg, trailingAvgSq] = adamupdate(net.Learnables, gradients,
                                                    trailingAvg, trailingAvgSq, itera
                                                    learningRate,gradientDecayFactor,

            % Update training plot with new points.
            addpoints(lossPlotter, iteration,double(gather(extractdata(loss))));
            title("Training Epoch " + epoch +" of " + numEpochs);
            drawnow;
        end
```

```
        % Update the learning rate after every learnRateDropPeriod.
        if mod(epoch,learnRateDropPeriod) == 0
            learningRate = learningRate * learnRateDropFactor;
        end
    end
end
```

Evaluate Model

Computer Vision Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionAOS`). For this example, use the average precision metric. The average precision provides a single number that incorporates the ability of the detector to make correct classifications (precision) and the ability of the detector to find all relevant objects (recall).

Evaluate the trained `dlnetwork` (Deep Learning Toolbox) object `net` on test data by following these steps.

- Specify the confidence threshold to use only detections with confidence scores above this value.
- Specify the overlap threshold to remove overlapping detections.
- Use the `generatePointPillarDetections` helper function, attached to this example as a supporting file, to get the bounding boxes, object confidence scores, and class labels.
- Call `evaluateDetectionAOS` with `detectionResults` and `groundTruthData` as arguments.

```
numInputs = numel(testData);

% Generate rotated rectangles from the cuboid labels.
bds = boxLabelDatastore(testLabels);
groundTruthData = transform(bds,@(x) createRotRect(x));

% Set the threshold values.
nmsPositiveIoUThreshold = 0.5;
confidenceThreshold = 0.25;
overlapThreshold = 0.1;

% Set numSamplesToTest to numInputs to evaluate the model on the entire
% test data set.
numSamplesToTest = 50;
detectionResults = table('Size',[numSamplesToTest 3],...
                        'VariableTypes',{'cell','cell','cell'},...
                        'VariableNames',{'Boxes','Scores','Labels'});

for num = 1:numSamplesToTest
    ptCloud = testData{num,1};

    [box,score,labels] = generatePointPillarDetections(net,ptCloud,anchorBoxes,gridParams,classNa
                                    overlapThreshold,P,N,executionEnvironment);

    % Convert the detected boxes to rotated rectangles format.
    if ~isempty(box)
        detectionResults.Boxes{num} = box(:,[1,2,4,5,7]);
    else
        detectionResults.Boxes{num} = box;
    end
    detectionResults.Scores{num} = score;
    detectionResults.Labels{num} = labels;
```

```
end

metrics = evaluateDetectionAOS(detectionResults,groundTruthData,nmsPositiveIoUThreshold)
```

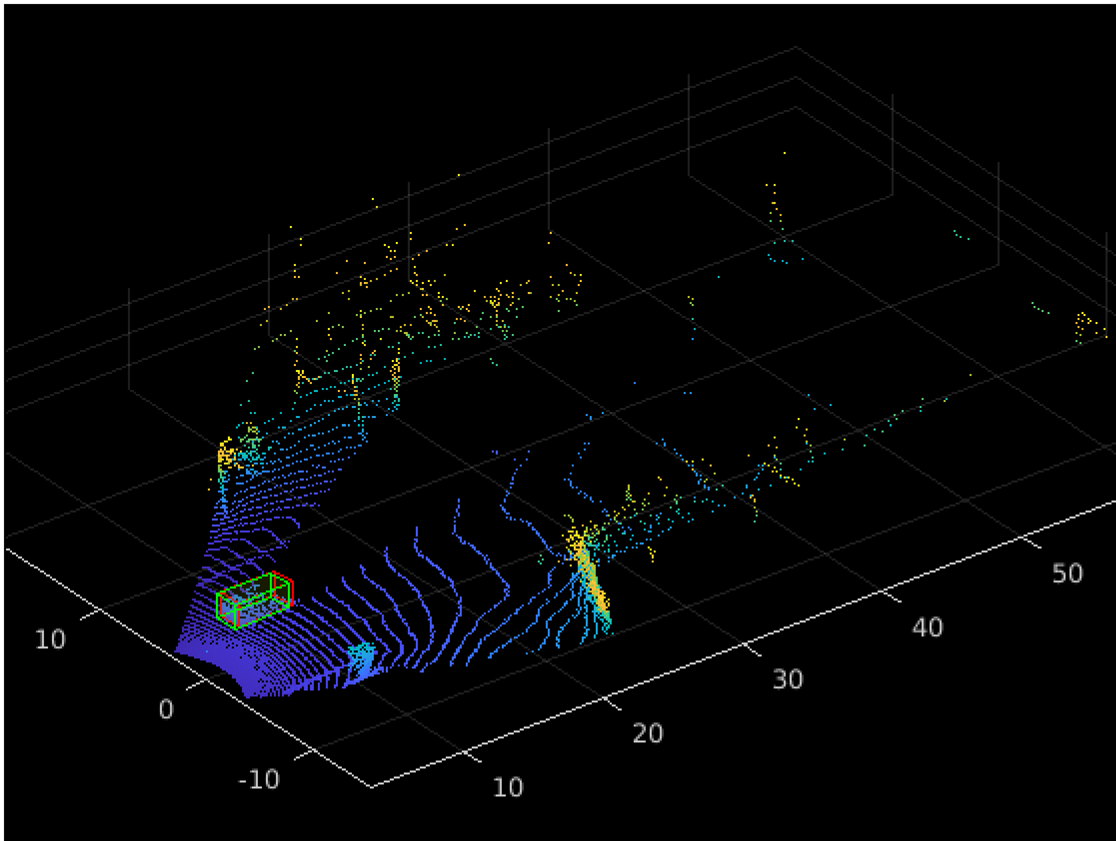
```
metrics=1×5 table
               AOS         AP       OrientationSimilarity       Precision            Recall
             _______    _______    _____________________    ______________    ______________

    car      0.69396    0.69396       {125×1 double}        {125×1 double}    {125×1 double}
```

Detect Objects Using Point Pillars

Use the network for object detection:

- Read the point cloud from the test data.
- Use the `generatePointPillarDetections` helper function, attached to this example as a supporting file, to get the predicted bounding boxes and confidence scores.
- Display the point cloud with bounding boxes.

```
ptCloud = testData{3,1};
gtLabels = testLabels{3,1}{1};

% Display the point cloud.
figure;
ax4 = pcshow(ptCloud.Location);

% The generatePointPillarDetections function detects the bounding boxes, scores for a
% given point cloud.
confidenceThreshold = 0.5;
overlapThreshold = 0.1;
[box,score,labels] = generatePointPillarDetections(net,ptCloud,anchorBoxes,gridParams,classNames
                      overlapThreshold,P,N,executionEnvironment);

% Display the detections on the point cloud.
showShape('cuboid', box, 'Parent', ax4, 'Opacity', 0.1, 'Color', 'red','LineWidth',0.5);hold on;
showShape('cuboid', gtLabels, 'Parent', ax4, 'Opacity', 0.1, 'Color', 'green','LineWidth',0.5);
zoom(ax4,2);
```



Helper Functions

Model Gradients

The function `modelGradients` takes as input the `dlnetwork` (Deep Learning Toolbox) object `net` and a mini-batch of input data `pillarFeautures` and `pillarIndices` with corresponding ground truth boxes, anchor boxes and grid parameters. The function returns the gradients of the loss with respect to the learnable parameters in `net`, the corresponding mini-batch loss, and the state of the current batch.

The model gradients function computes the total loss and gradients by performing these operations.

- Extract the predictions from the network using the `forward` (Deep Learning Toolbox) function.
- Generate the targets for loss computation by using the ground truth data, grid parameters, and anchor boxes.
- Calculate the loss function for all six predictions from the network.
- Compute the total loss as the sum of all losses.
- Compute the gradients of learnables with respect to the total loss.

```
function [gradients, loss, state] = modelGradients(net, pillarFeatures, pillarIndices, boxLabels
                                                   executionEnvironment)

    % Extract the predictions from the network.
    YPredictions = cell(size(net.OutputNames));
    [YPredictions{:}, state] = forward(net,pillarIndices,pillarFeatures);
```

```
    % Generate target for predictions from the ground truth data.
    YTargets = generatePointPillarTargets(YPredictions, boxLabels, pillarIndices, gridParams, anc
    YTargets = cellfun(@ dlarray,YTargets,'UniformOutput', false);
    if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
        YTargets = cellfun(@ gpuArray,YTargets,'UniformOutput', false);
    end

    [angLoss, occLoss, locLoss, szLoss, hdLoss, clfLoss] = computePointPillarLoss(YPredictions, Y

    % Compute the total loss.
    loss = angLoss + occLoss + locLoss + szLoss + hdLoss + clfLoss;

    % Compute the gradients of the learnables with regard to the loss.
    gradients = dlgradient(loss,net.Learnables);

end

function [pillarFeatures, pillarIndices, labels] = createBatchData(features, indices, groundTruth
% Returns pillar features and indices combined along the batch dimension
% and bounding boxes concatenated along batch dimension in labels.

    % Concatenate features and indices along batch dimension.
    pillarFeatures = cat(4, features{:,1});
    pillarIndices = cat(4, indices{:,1});

    % Get class IDs from the class names.
    classNames = repmat({categorical(classNames')}, size(groundTruthClasses));
    [~, classIndices] = cellfun(@(a,b)ismember(a,b), groundTruthClasses, classNames, 'UniformOutp

    % Append the class indices and create a single array of responses.
    combinedResponses = cellfun(@(bbox, classid)[bbox, classid], groundTruthBoxes, classIndices, 
    len = max(cellfun(@(x)size(x,1), combinedResponses));
    paddedBBoxes = cellfun( @(v) padarray(v,[len-size(v,1),0],0,'post'), combinedResponses, 'Uni
    labels = cat(4, paddedBBoxes{:,1});
end

function lidarData = downloadWPIData(outputFolder, lidarURL)
% Download the data set from the given URL into the output folder.

    lidarDataTarFile = fullfile(outputFolder,'WPI_LidarData.tar.gz');
    if ~exist(lidarDataTarFile, 'file')
        mkdir(outputFolder);

        disp('Downloading WPI Lidar driving data (760 MB)...');
        websave(lidarDataTarFile, lidarURL);
        untar(lidarDataTarFile,outputFolder);
    end

    % Extract the file.
    if ~exist(fullfile(outputFolder, 'WPI_LidarData.mat'), 'file')
        untar(lidarDataTarFile,outputFolder);
    end
    load(fullfile(outputFolder, 'WPI_LidarData.mat'),'lidarData');
    lidarData = reshape(lidarData,size(lidarData,2),1);
end

function net = downloadPretrainedPointPillarsNet(outputFolder, pretrainedNetURL)
```

```
% Download the pretrained PointPillars detector.

    preTrainedMATFile = fullfile(outputFolder,'trainedPointPillarsNet.mat');
    preTrainedZipFile = fullfile(outputFolder,'trainedPointPillars.zip');

    if ~exist(preTrainedMATFile,'file')
        if ~exist(preTrainedZipFile,'file')
            disp('Downloading pretrained detector (8.3 MB)...');
            websave(preTrainedZipFile, pretrainedNetURL);
        end
        unzip(preTrainedZipFile, outputFolder);
    end
    pretrainedNet = load(preTrainedMATFile);
    net = pretrainedNet.net;
end

function lossPlotter = configureTrainingProgressPlotter(f)
% The configureTrainingProgressPlotter function configures training
% progress plots for various losses.
    figure(f);
    clf
    ylabel('Total Loss');
    xlabel('Iteration');
    lossPlotter = animatedline;
end

function retValue = checkForNaN(gradients,loss)
% Based on experiments it is found that the last convolution head
% 'occupancy|conv2d' contains NaNs as the gradients. This function checks
% whether gradient values contain NaNs. Add other convolution
% head values to the condition if NaNs are present in the gradients.
    gradValue = gradients.Value((gradients.Layer == 'occupancy|conv2d') & (gradients.Parameter ==
    if (sum(isnan(extractdata(loss)),'all') > 0) || (sum(isnan(extractdata(gradValue{1,1})),'all
        retValue = true;
    else
        retValue = false;
    end
end
```

References

[1] Lang, Alex H., Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. "PointPillars: Fast Encoders for Object Detection From Point Clouds." In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (CVPR), 12689-12697. Long Beach, CA, USA: IEEE, 2019. https://doi.org/10.1109/CVPR.2019.01298.

Aerial Lidar SLAM Using FPFH Descriptors

This example demonstrates how to implement the simultaneous localization and mapping (SLAM) algorithm for aerial mapping using 3-D features. The goal of this example is to estimate the trajectory of a robot and create a point cloud map of its environment.

The SLAM algorithm in this example estimates a trajectory by finding a coarse alignment between downsampled accepted scans, using fast point feature histogram (FPFH) descriptors extracted at each point, then applies the iterative closest point (ICP) algorithm to fine-tune the alignment. 3-D pose graph optimization, from Navigation Toolbox™, reduces the drift in the estimated trajectory.

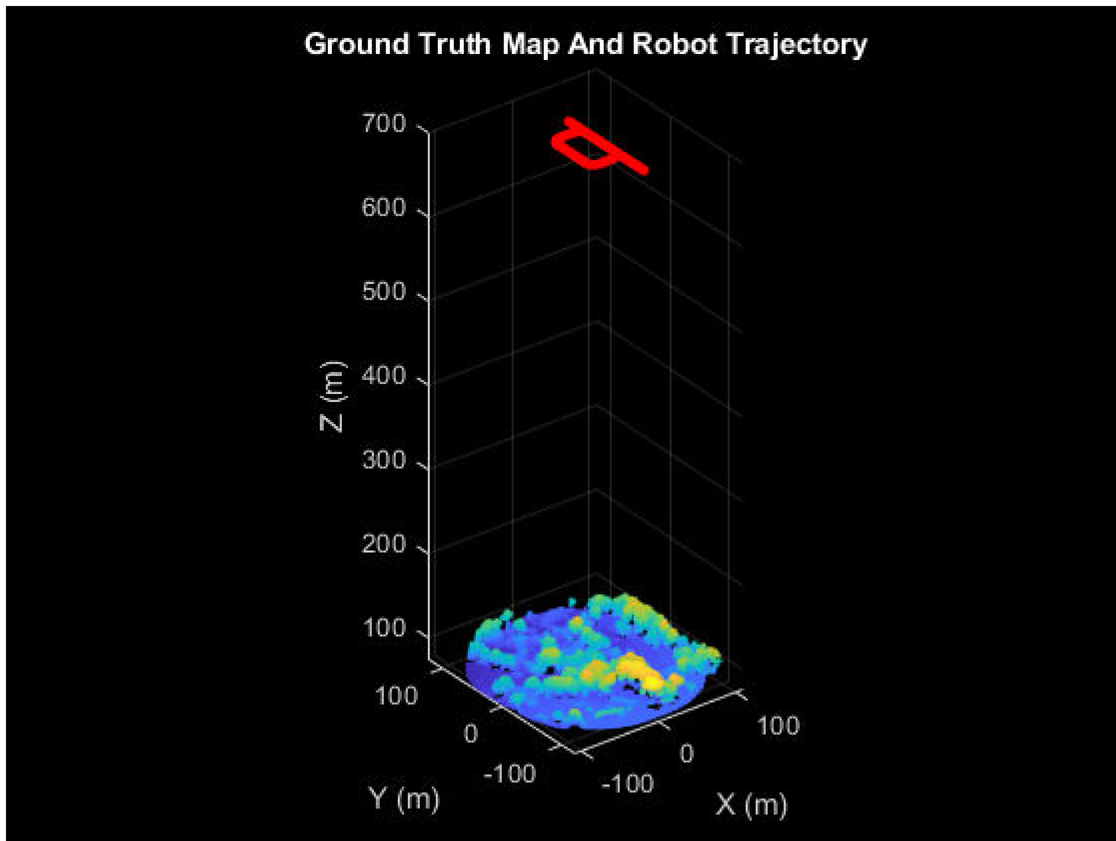
Create and Visualize Data

Create synthetic lidar scans from a patch of aerial data, downloaded from the OpenTopography website [1] on page 1-0. Load a MAT-file containing a sequence of waypoints over aerial data that defines the trajectory of a robot. Compute lidar scans from the data at each waypoint using the `helperCreateDataset` function. The function outputs the lidar scans computed at each waypoint as an array of `pointCloud` objects, original map covered by robot and ground truth waypoints.

```
datafile = 'aerialMap.tar.gz';
wayPointsfile = 'gTruthWayPoints.mat';

% Generate a lidar scan at each waypoint using the helper function
[pClouds,orgMap,gTruthWayPts] = helperCreateDataset(datafile,wayPointsfile);
```

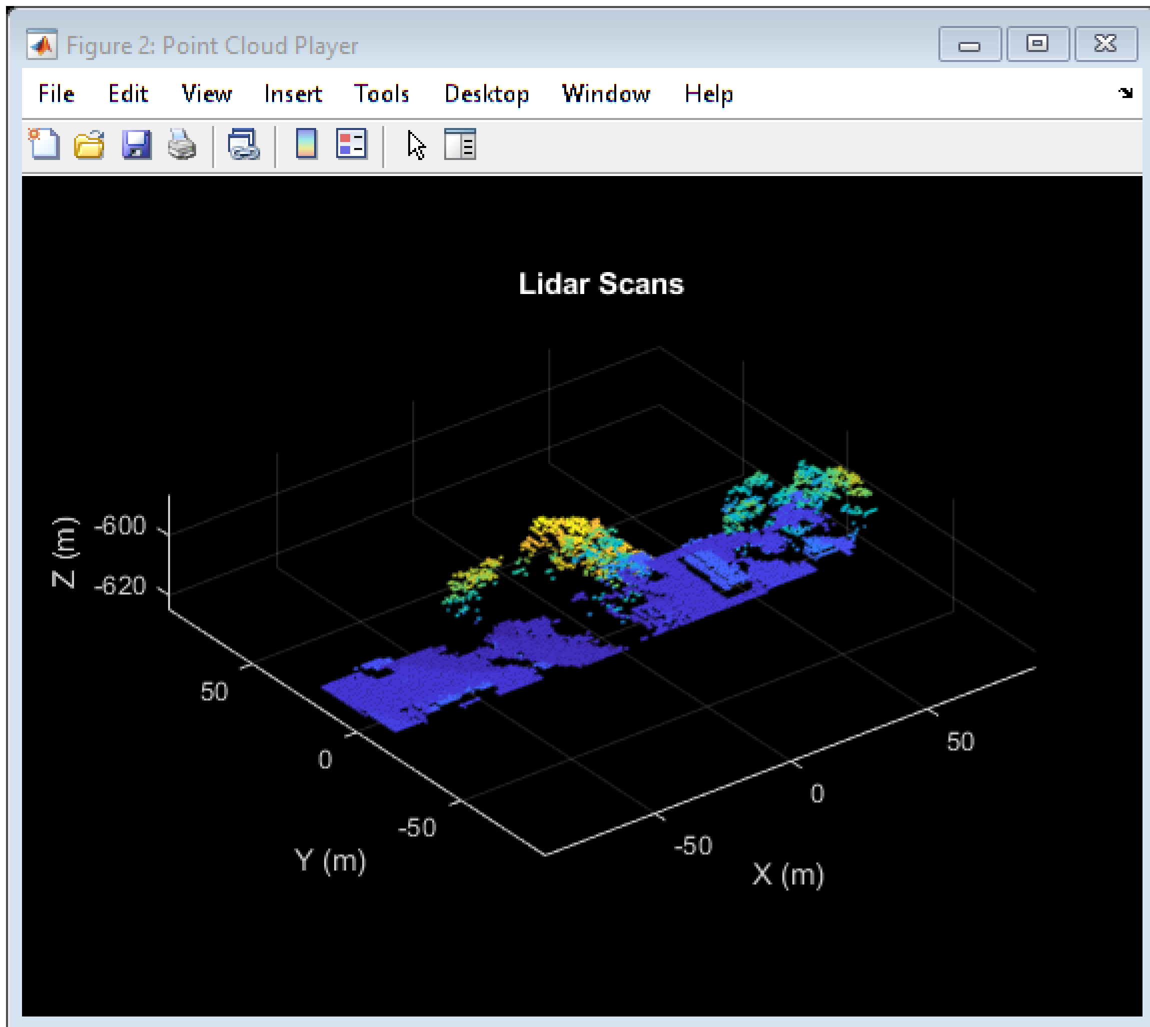
Visualize the ground truth waypoints on the original map covered by the robot.

```
% Create a figure window to visualize the ground truth map and waypoints
hFigGT = figure;
axGT = axes('Parent',hFigGT,'Color','black');
% Visualize the ground truth waypoints
pcshow(gTruthWayPts,'red','MarkerSize',150,'Parent',axGT)
hold on
% Visualize the original map covered by the robot
pcshow(orgMap,'MarkerSize',10,'Parent',axGT)
hold off
% Customize the axis labels
xlabel(axGT,'X (m)')
ylabel(axGT,'Y (m)')
zlabel(axGT,'Z (m)')
title(axGT,'Ground Truth Map And Robot Trajectory')
```



Visualize the extracted lidar scans using the `pcplayer` object.

```
% Specify limits for the player
xlimits = [-90 90];
ylimits = [-90 90];
zlimits = [-625 -587];

% Create a pcplayer object to visualize the lidar scans
lidarPlayer = pcplayer(xlimits,ylimits,zlimits);
% Customize the pcplayer axis labels
xlabel(lidarPlayer.Axes,'X (m)')
ylabel(lidarPlayer.Axes,'Y (m)')
zlabel(lidarPlayer.Axes,'Z (m)')
title(lidarPlayer.Axes,'Lidar Scans')
% Loop over and visualize the data
for l = 1:length(pClouds)
    % Extract the lidar scan
    ptCloud = pClouds(l);
    % Update the lidar display
    view(lidarPlayer,ptCloud)
    pause(0.05)
end
```



Set Up Tunable Parameters

Specify the parameters for trajectory and loop closure estimation. Tune these parameters for your specific robot and environment.

Parameters for Point Cloud Registration

Specify the number of lidar scans to skip between each scan accepted for registration. Since successive scans have high overlap, skipping a few frames can improve algorithm speed without compromising accuracy.

```
skipFrames = 3;
```

Downsampling lidar scans can improve algorithm speed. The box grid filter downsamples the point cloud by averaging all points within each cell to a single point. Specify the size for individual cells of a box grid filter, in meters.

```
gridStep = 1.5; % in meters
```

FPFH descriptors are extracted for each valid point in the downsampled point cloud. Specify the neighborhood size for the k-nearest neighbor (KNN) search method used to compute the descriptors.

```
neighbors = 60;
```

Set the threshold and ratio for matching the FPFH descriptors, used to identify point correspondences.

```
matchThreshold = 0.1;
matchRatio = 0.97;
```

Set the maximum distance and number of trails for relative pose estimation between successive scans.

```
maxDistance = 1;
maxNumTrails = 3000;
```

Specify the percentage of inliers to consider for fine-tuning relative poses.

```
inlierRatio = 0.1;
```

Specify the size of each cell of a box grid filter, used to create maps by aligning lidar scans.

```
alignGridStep = 1.2;
```

Parameters for Loop Closure Estimation

Specify the radius around the current robot location to search for loop closure candidates.

```
loopClosureSearchRadius = 7.9;
```

The loop closure algorithm is based on 3-D submap creation and matching. A submap consists of a specified number of accepted scans `nScansPerSubmap`. The loop closure algorithm also disregards a specified number of the most recent scans `subMapThresh`, while searching for loop candidates. Specify the root mean squared error (RMSE) threshold `loopClosureThreshold,` for accepting a submap as a match. A lower score can indicate a better match, but scores vary based on sensor data and preprocessing.

```
nScansPerSubmap = 3;
subMapThresh = 15;
loopClosureThreshold = 0.6;
```

Specify the maximum acceptable root mean squared error (RMSE) for the estimation of relative pose between loop candidates `rmseThreshold`. Choosing a lower value can result in more accurate loop closure edges, which has a high impact on pose graph optimization, but scores vary based on sensor data and preprocessing.

```
rmseThreshold = 0.6;
```

Initialize Variables

Create a pose graph, using a `poseGraph3D` (Navigation Toolbox) object, to store estimated relative poses between accepted lidar scans.

```
pGraph = poseGraph3D;
% Default serialized upper-right triangle of a 6-by-6 Information Matrix
infoMat = [1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 1];
```

Preallocate and initialize the variables required for computation.

```
% Allocate memory to store submaps
subMaps = cell(floor(length(pClouds)/(skipFrames*nScansPerSubmap)),1);
```

```
% Allocate memory to store each submap pose
subMapPoses = zeros(round(length(pClouds)/(skipFrames*nScansPerSubmap)),3);
% Initialize variables to store accepted scans and their transforms for
% submap creation
pcAccepted = pointCloud.empty(0);
tformAccepted = rigid3d.empty(0);
% Initialize variable to store relative poses from the feature-based approach
% without pose graph optimization
fpfhTform = rigid3d.empty(0);

% Counter to track the number of scans added
count = 1;
```

Create variables to visualize processed lidar scans and estimated trajectory.

```
% Set to 1 to visualize processed lidar scans during build process
viewPC = 0;
% Create a pcplayer object to view the lidar scans while
% processing them sequentially, if viewPC is enabled
if viewPC == 1
    pplayer = pcplayer(xlimits,ylimits,zlimits);
    % Customize player axis labels
    xlabel(pplayer.Axes,'X (m)')
    ylabel(pplayer.Axes,'Y (m)')
    zlabel(pplayer.Axes,'Z (m)')
    title(pplayer.Axes,'Processed Scans')
end
% Create a figure window to visualize the estimated trajectory
hFigTrajUpdate = figure;
axTrajUpdate = axes('Parent', hFigTrajUpdate, 'Color', 'black');
title(axTrajUpdate, 'Sensor Pose Trajectory')
```

Trajectory Estimation and Refinement

The trajectory of the robot is a collection of its poses which consists of its location and orientation in 3-D space. Estimate a robot pose from a 3-D lidar scan instance using the 3-D lidar SLAM algorithm. Use these processes to perform 3-D SLAM estimation:

- Point cloud downsampling
- Point cloud registration
- Submap creation
- Loop closure querying
- Pose graph optimization

Iteratively process the lidar scans to estimate the trajectory of the robot.

```
rng(0) % Set random seed to guarantee consistent results in pcmatchfeatures
for FrameIdx = 1:skipFrames:length(pClouds)
```

Point Cloud Downsampling

Point cloud downsampling can improve the speed of the point cloud registration algorithm. Downsampling should be tuned for your specific needs.

```
    % Downsample the current scan
    curScan = pcdownsample(pClouds(FrameIdx),'gridAverage',gridStep);
```

```
    if viewPC == 1
        % Visualize down sampled point cloud
        view(pplayer,curScan)
    end
```

Point Cloud Registration

Point cloud registration estimates the relative pose between the current scan and a previous scan. The example uses these steps for registration:

- Extracts FPFH descriptors from each scan using the `extractFPFHFeatures` function
- Identifies point correspondences by comparing descriptors using the `pcmatchfeatures` function
- Estimates the relative pose from point correspondences using the `estimateGeometricTransform3D` function
- Fine-tunes the relative pose using the `pcregistericp` function

```
    % Extract FPFH features
    curFeature = extractFPFHFeatures(curScan,'NumNeighbors',neighbors);

    if FrameIdx == 1
        % Update the acceptance scan and its tform
        pcAccepted(count,1) = curScan;
        tformAccepted(count,1) = rigid3d;
        % Update the initial pose to the first waypoint of ground truth for
        % comparison
        fpfhTform(count,1) = rigid3d(eye(3),gTruthWayPts(1,:));
    else
        % Identify correspondences by matching current scan to previous scan
        indexPairs = pcmatchfeatures(curFeature,prevFeature,curScan,prevScan, ...
            'MatchThreshold',matchThreshold,'RejectRatio',matchRatio);
        matchedPrevPts = select(prevScan,indexPairs(:,2));
        matchedCurPts = select(curScan,indexPairs(:,1));

        % Estimate relative pose between current scan and previous scan
        % using correspondences
        tform1 = estimateGeometricTransform3D(matchedCurPts.Location, ...
            matchedPrevPts.Location,'rigid','MaxDistance',maxDistance, ...
            'MaxNumTrials',maxNumTrails);

        % Perform ICP registration to fine-tune relative pose
        tform = pcregistericp(curScan,prevScan,'InitialTransform',tform1, ...
            'InlierRatio',inlierRatio);
```

Convert the rigid transformation to an *xyz*-position and a quaternion orientation to add it to the pose graph.

```
        relPose = [tform2trvec(tform.T') tform2quat(tform.T')];
        % Add relative pose to pose graph
        addRelativePose(pGraph,relPose);
```

Store the accepted scans and their poses for submap creation.

```
        % Update counter and store accepted scans and their poses
        count = count + 1;
        pcAccepted(count,1) = curScan;
        accumPose = pGraph.nodes(height(pGraph.nodes));
        tformAccepted(count,1) = rigid3d((trvec2tform(accumPose(1:3)) * ...
```

```
            quat2tform(accumPose(4:7)))');
        % Update estimated poses
        fpfhTform(count) = rigid3d(tform.T * fpfhTform(count-1).T);
    end
```

Submap Creation

Create submaps by aligning sequential, accepted scans with each other in groups of the specified size `nScansPerSubmap`, using the `pcalign` function. Using submaps can result in faster loop closure queries.

```
    currSubMapId = floor(count/nScansPerSubmap);
    if rem(count,nScansPerSubmap) == 0
        % Align an array of lidar scans to create a submap
        subMaps{currSubMapId} = pcalign(...
            pcAccepted((count - nScansPerSubmap + 1):count,1), ...
            tformAccepted((count - nScansPerSubmap + 1):count,1), ...
            alignGridStep);
        % Assign center scan pose as pose of submap
        subMapPoses(currSubMapId,:) = tformAccepted(count - ...
            floor(nScansPerSubmap/2),1).Translation;
    end
```

Loop Closure Query

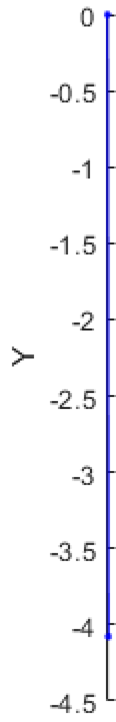
Use a *loop closure query* to identify previously visited places. A loop closure query consists of finding a similarity between the current scan and previous submaps. If you find a similar scan, then the current scan is a loop closure candidate. Loop closure candidate estimation consists of these steps:

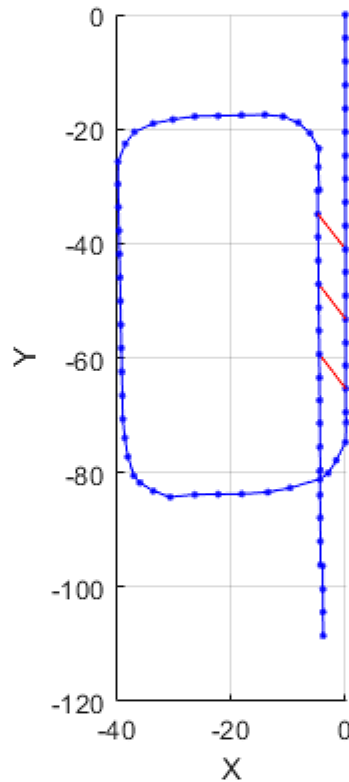
- Estimate matches between previous submaps and the current scan using the `helperEstimateLoopCandidates` function. A submap is a match, if the RMSE between the current scan and submap is less than the specified value of `loopClosureThreshold.` The previous scans represented by all matching submaps are loop closure candidates.
- Compute the relative pose between the current scan and the loop closure candidate using the ICP registration algorithm. The loop closure candidate with the lowest RMSE is the best probable match for the current scan.

A loop closure candidate is accepted as a valid loop closure only when the RMSE is lower than the specified threshold.

```
    if currSubMapId > subMapThresh
        mostRecentScanCenter = pGraph.nodes(height(pGraph.nodes));
        % Estimate possible loop closure candidates by matching current
        % scan with submaps
        [loopSubmapIds,~] = helperEstimateLoopCandidates(subMaps,curScan, ...
            subMapPoses,mostRecentScanCenter,currSubMapId,subMapThresh, ...
            loopClosureSearchRadius,loopClosureThreshold);

        if ~isempty(loopSubmapIds)
            rmseMin = inf;
            % Estimate the best match for the current scan from the matching submap ids
            for k = 1:length(loopSubmapIds)
                % Check every scan within the submap
                for kf = 1:nScansPerSubmap
                    probableLoopCandidate = ...
                        loopSubmapIds(k)*nScansPerSubmap - kf + 1;
                    [pose_Tform,~,rmse] = pcregistericp(curScan, ...
```

```
                        pcAccepted(probableLoopCandidate));

                    % Update the best loop closure candidate
                    if rmse < rmseMin
                        rmseMin = rmse;
                        matchingNode = probableLoopCandidate;
                        Pose = [tform2trvec(pose_Tform.T') ...
                            tform2quat(pose_Tform.T')];
                    end
                end
            end
            % Check if loop closure candidate is valid
            if rmseMin < rmseThreshold

                % Add relative pose of loop closure candidate to pose graph
                addRelativePose(pGraph,Pose,infoMat,matchingNode, ...
                    pGraph.NumNodes);
            end
        end
    end
    % Update previous point cloud and feature
    prevScan = curScan;
    prevFeature = curFeature;

    % Visualize the estimated trajectory from the accepted scan.
    show(pGraph,'IDs','off','Parent',axTrajUpdate);
    drawnow
end
```


0
-0.5
-1
-1.5
-2
Y
-2.5
-3
-3.5
-4
-4.5



Pose Graph Optimization

Reduce the drift in the estimated trajectory by using the `optimizePoseGraph` (Navigation Toolbox) function. In general, the pose of the first node in the pose graph represents the origin. To compare the estimated trajectory with the ground truth waypoints, specify the first ground truth waypoint as the pose of the first node.

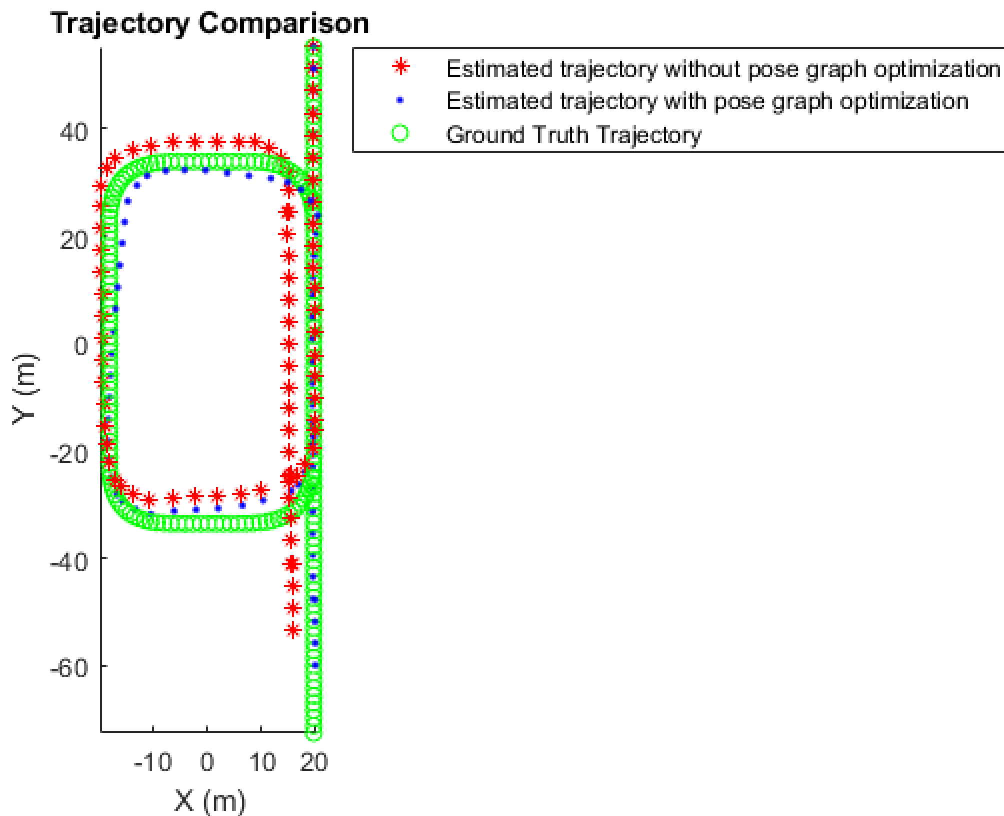
```
pGraph = optimizePoseGraph(pGraph,'FirstNodePose',[gTruthWayPts(1,:),1,0,0,0]);
```

Visualize Trajectory Comparisons

Visualize the estimated trajectory using the features without pose graph optimization, the features with pose graph optimization, and the ground truth data.

```
% Get estimated trajectory from pose graph
pGraphTraj = pGraph.nodes;
% Get estimated trajectory from feature-based registration without pose
% graph optimization
fpfhEstimatedTraj = zeros(count,3);
for i = 1:count
    fpfhEstimatedTraj(i,:) = fpfhTform(i).Translation;
end

% Create a figure window to visualize the ground truth and estimated
% trajectories
hFigTraj = figure;
axTraj = axes('Parent',hFigTraj,'Color','black');
```

```
plot3(fpfhEstimatedTraj(:,1),fpfhEstimatedTraj(:,2),fpfhEstimatedTraj(:,3), ...
    'r*','Parent',axTraj)
hold on
plot3(pGraphTraj(:,1),pGraphTraj(:,2),pGraphTraj(:,3),'b.','Parent',axTraj)
plot3(gTruthWayPts(:,1),gTruthWayPts(:,2),gTruthWayPts(:,3),'go','Parent',axTraj)
hold off
axis equal
view(axTraj,2)
xlabel(axTraj,'X (m)')
ylabel(axTraj,'Y (m)')
zlabel(axTraj,'Z (m)')
title(axTraj,'Trajectory Comparison')
legend(axTraj,'Estimated trajectory without pose graph optimization', ...
    'Estimated trajectory with pose graph optimization', ...
    'Ground Truth Trajectory','Location','bestoutside')
```

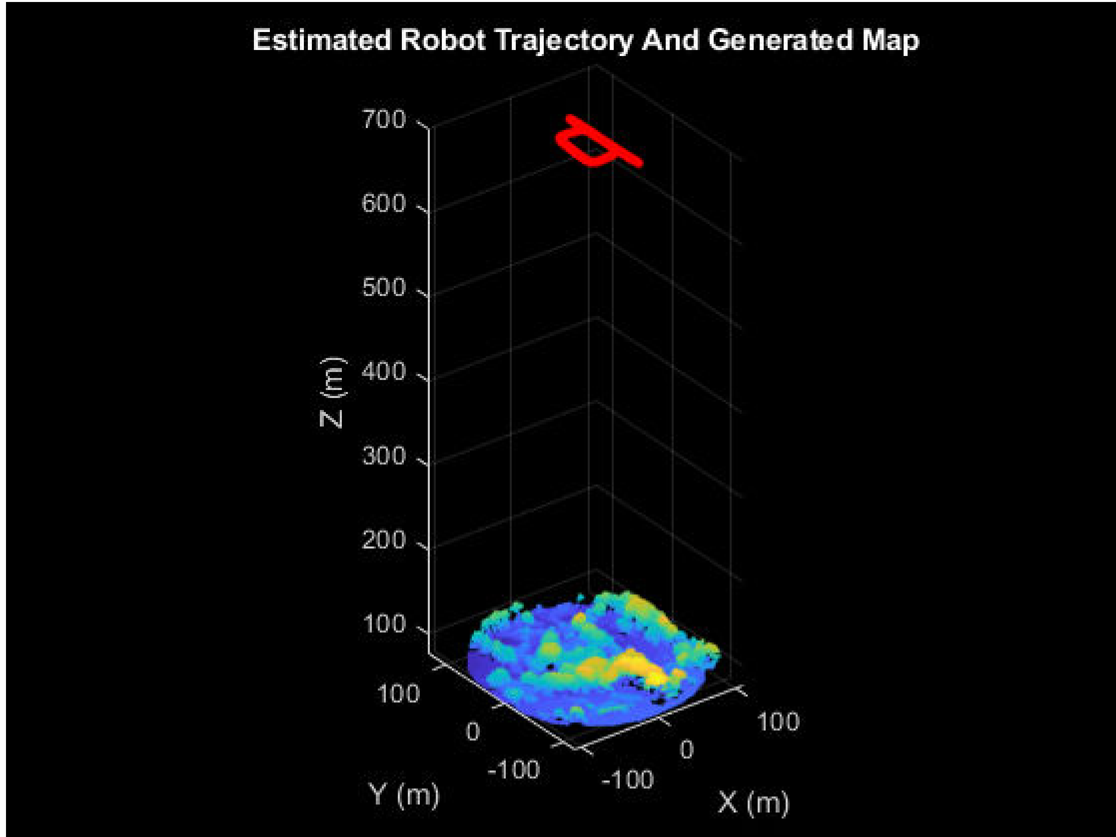


Build and Visualize Generated Map

Transform and merge lidar scans using estimated poses to create an accumulated point cloud map.

```
% Get the estimated trajectory from poses
estimatedTraj = pGraphTraj(:,1:3);

% Convert relative poses to rigid transformations
estimatedTforms = rigid3d.empty(0);
for idx=1:pGraph.NumNodes
    pose = pGraph.nodes(idx);
    rigidPose = rigid3d((trvec2tform(pose(1:3)) * quat2tform(pose(4:7)))');
```

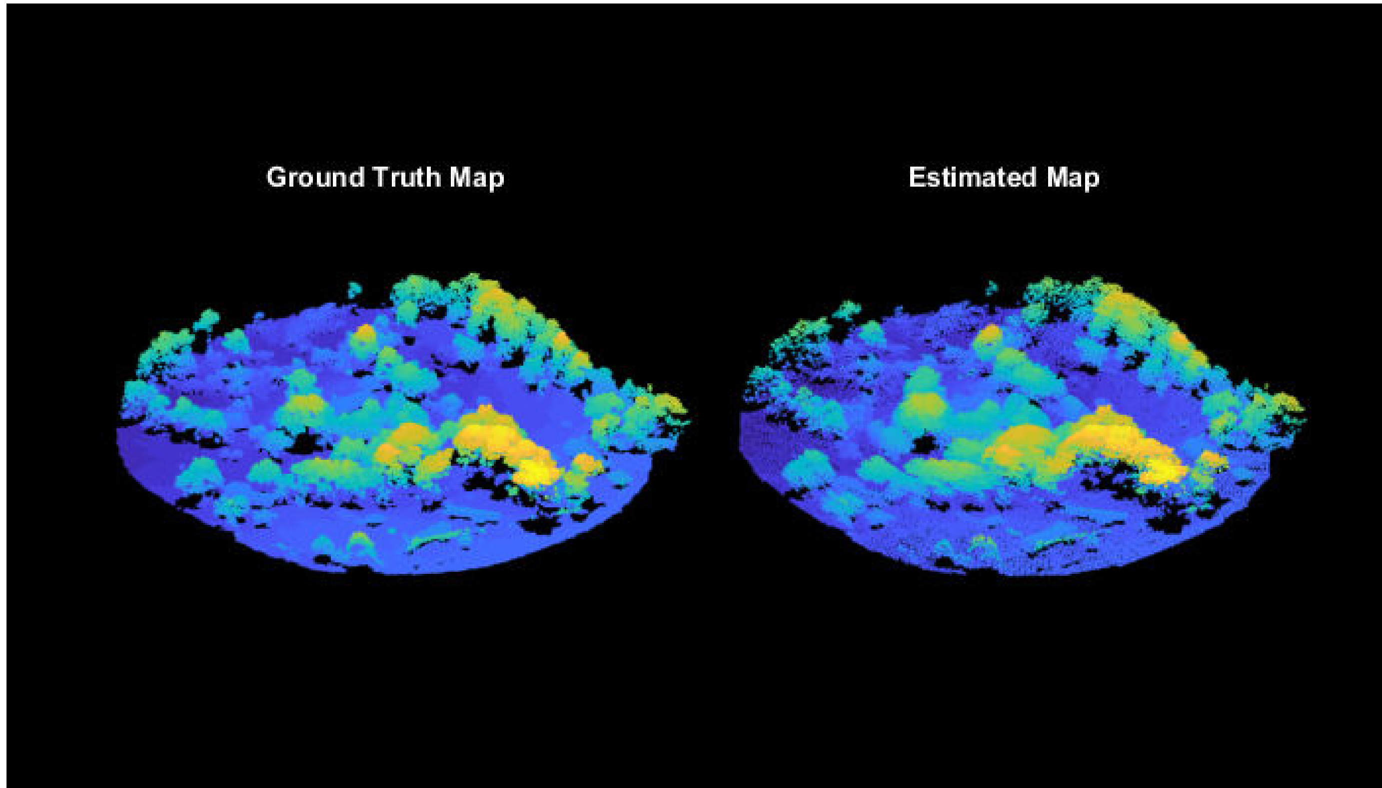
```
    estimatedTforms(idx,1) = rigidPose;
end

% Create global map from processed point clouds and their relative poses
globalMap = pcalign(pcAccepted,estimatedTforms,alignGridStep);

% Create a figure window to visualize the estimated map and trajectory
hFigTrajMap = figure;
axTrajMap = axes('Parent',hFigTrajMap,'Color','black');
pcshow(estimatedTraj,'red','MarkerSize',150,'Parent',axTrajMap)
hold on
pcshow(globalMap,'MarkerSize',10,'Parent',axTrajMap)
hold off
% Customize axis labels
xlabel(axTrajMap,'X (m)')
ylabel(axTrajMap,'Y (m)')
zlabel(axTrajMap,'Z (m)')
title(axTrajMap,'Estimated Robot Trajectory And Generated Map')
```



Visualize the ground truth map and the estimated map.

```
% Create a figure window to display both the ground truth map and estimated map
hFigMap = figure('Position',[0 0 700 400]);
axMap1 = subplot(1,2,1,'Color','black','Parent',hFigMap);
axMap1.Position = [0 0.2 0.55 0.55];
pcshow(orgMap,'Parent',axMap1)
axis off
title(axMap1,'Ground Truth Map')
```

```
axMap2 = subplot(1,2,2,'Color','black','Parent',hFigMap);
axMap2.Position = [0.45,0.2,0.55,0.55];
pcshow(globalMap,'Parent',axMap2)
axis off
title(axMap2,'Estimated Map')
```



References

[1] "Tuscaloosa, AL: Seasonal Inundation Dynamics And Invertebrate Communities." OpenTopography. National Center for Airborne Laser Mapping, January 12, 2011. https://portal.opentopography.org/datasetMetadata?otCollectionID=OT.112011.26916.1.

Objects

`lasFileReader` | `pointCloud` | `pcplayer` | `occupancyMap3D` (Navigation Toolbox) | `poseGraph3D` (Navigation Toolbox) | `rigid3d`

Functions

`readPointCloud` | `insertPointCloud` (Navigation Toolbox) | `rayIntersection` (Navigation Toolbox) | `addRelativePose` (Navigation Toolbox) | `optimizePoseGraph` (Navigation Toolbox) | `show` (Navigation Toolbox) | `extractFPFHFeatures` | `pcmatchfeatures` | `estimateGeometricTransform3D` | `pcdownsample` | `pctransform` | `pcregistericp` | `pcalign` | `tform2trvec` (Navigation Toolbox) | `tform2quat` (Navigation Toolbox)

Collision Warning Using 2-D Lidar

This example shows how to detect obstacles and warn of possible collisions using 2-D lidar data.

Overview

Logistics warehouses are increasingly mounting 2-D lidars on automatic guided vehicles (AGV) for navigation purposes, due to the affordability, long range, and high resolution of the sensor. The sensors assist in collision detection, which is an important task for the safe navigation of AGVs in complex environments. This example shows how to represent a robot workspace populated with obstacles, generate 2-D lidar data, detect obstacles, and provide a warning before an impending collision.

Create a Warehouse Map

To represent the environment of the robot workspace, define a `binaryOccupancyMap` object that contains the floor plan of the warehouse. Each cell in the occupancy grid has a logical value. An occupied location is represented as 1 and a free location is represented as 0. You can use the occupancy information to generate synthetic 2-D lidar data.

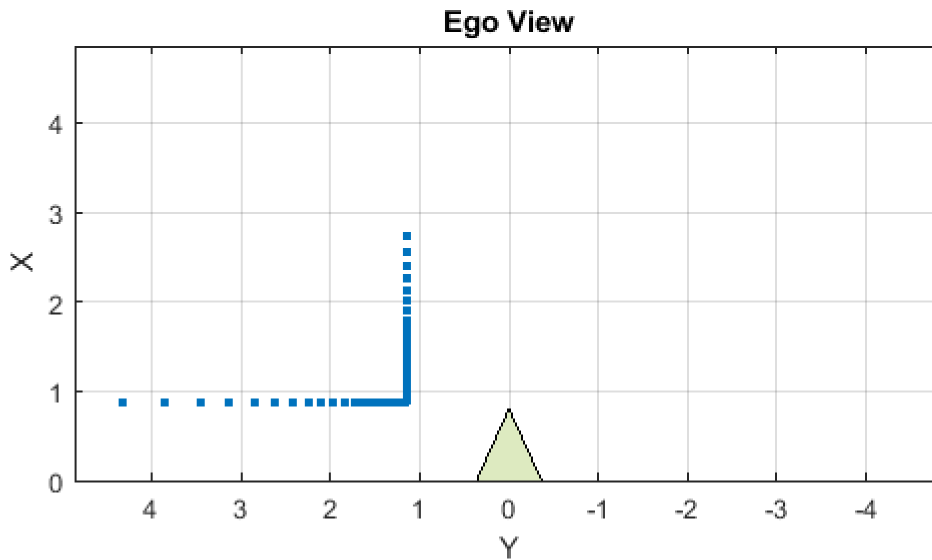
Add obstacles to the map near to the defined route saved in the `waypoints.mat` file that the AGV traverses.

```
% Create a binary warehouse map and place obstacles at defined locations
map = helperCreateBinaryOccupancyMap;

% Visualize map with obstacles
figure
show(map)
title('Warehouse Floor Plan With Obstacles')

% Add AGV to the map
pose = [5 40 0];
helperPlotRobot(gca,pose);
```

Simulate a 2-D Lidar Sensor

Simulate a 2-D lidar sensor by using a `rangeSensor` System object™ to gather lidar readings for the generated map. Load a MAT-file containing the predefined waypoints of the AGV into the workspace. Use the simulated `lidar` sensor to return range and angle readings for a pose of the AGV, and then use the ranges and angles to generate a `lidarScan` object that contains the 2-D lidar scan.

```
% Simulate lidar sensor. Use default detection angle of [-pi pi]
lidar = rangeSensor;
% Set min and max values of the detectable range of the sensor in meters
lidar.Range = [0 5];

% Load waypoints through which AGV moves
load waypoints.mat
traj = waypointsMap;

% Select a waypoint to visualize scan data
Vehiclepose = traj(350,:);
% Generate lidar readings
[ranges,angles] = lidar(Vehiclepose,map);
% Store annd visualize 2-D lidar scan
scan = lidarScan(ranges,angles);
plot(scan)
title('Ego View')
helperPlotRobot(gca,[0 0 Vehiclepose(3)]);
```



Set Up Visualization

Set up a figure window that displays AGV movement in the warehouse, the associated lidar scans of the environment, displays obstacles as filled circles in bird's eye view, and color-coded collision warning messages. The color used for each warning signifies the likelihood of a collision based on the detection area zone that the obstacle occupies at that waypoint.

```
% Set up display
display = helperVisualizer;

% Plot warehouse map in the display window
hRobot =  plotBinaryMap(display,map,pose);
```

Collision Warning Based on Zones

Collision warnings only appear if an obstacle falls within the detection area of the AGV.

Define the Detection Area

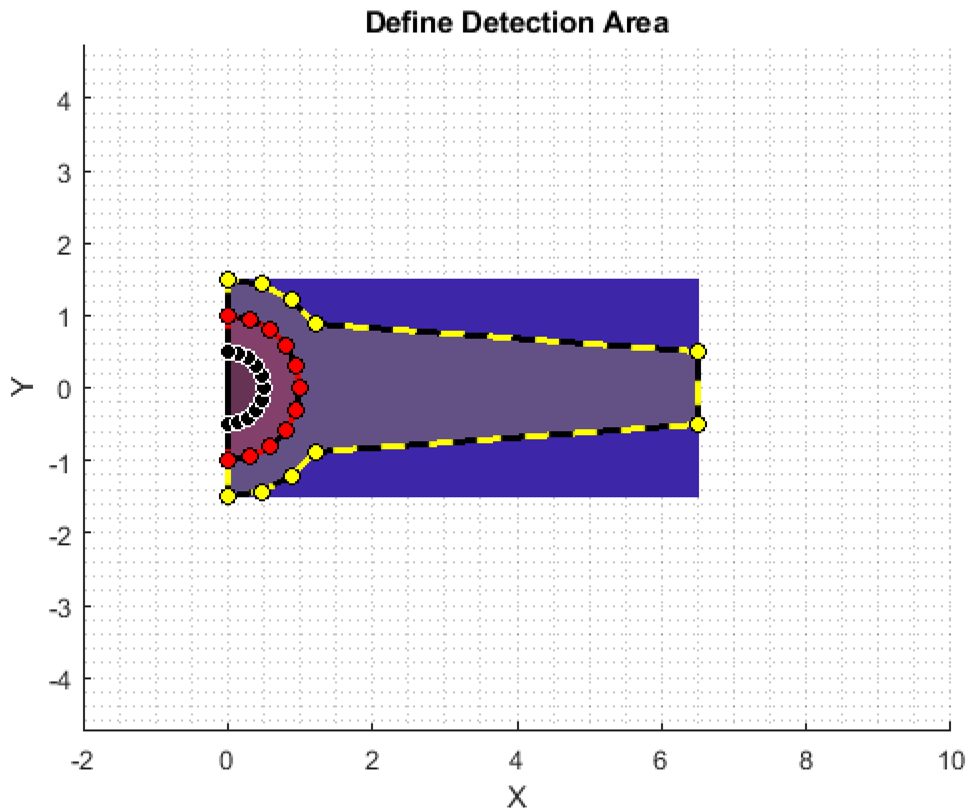
Create a custom detectable area with different colors, and shapes, and modify the area of color regions on figure GUI. Run the below portion of code and modify the polygon handles to accommodate your requirements of the detection area. The code below creates 3 polygon handles of semi-circular regions with a radius of 1.5, 1, 0.5 in meters and AGV is positioned at [0 0]. Modify the radius or change the polygon objects to create a custom detection area.

```
figure
detAxes = gca;
```

```
title(detAxes,'Define Detection Area')
axis(detAxes,[-2 10 -2 4])
xlabel(detAxes,'X')
ylabel(detAxes,'Y')
axis(detAxes,'equal')
grid(detAxes,'minor')
t = linspace(-pi/2,pi/2,11)';
% Specify colors
colors = ["yellow" "red" "black"];
% Specify radius
radius = [1.5 1 0.5];
% Create a 3x1 matrix of type Polygon
detAreaHandles = repmat(images.roi.Polygon,[3 1]);
% Create a polygon with a tunnel shape
pos = [cos(t) sin(t)]*radius(1);
posLine = [6.5 -0.5; 6.5 0.5];
pos = [pos(1:4,:); posLine; pos(end-3:end,:)];
detAreaHandles(1) = drawpolygon( ...
    'Parent',detAxes, ...
    'InteractionsAllowed','reshape', ...
    'Position',pos, ...
    'StripeColor','black', ...
    'Color',colors(1));
% Create semicircle polygons
for k = 2:numel(colors)
    pos = [cos(t) sin(t)]*radius(k);
    detAreaHandles(k) = drawpolygon( ...
    'Parent',detAxes, ...
    'InteractionsAllowed','reshape', ...
    'Position',pos, ...
    'StripeColor','black', ...
    'Color',colors(k));
end
pause(2) % Pausing for the detection area window to load
```

To save the created detection area, run the `helperSaveDetectionArea` function. Use the axes handle of the figure with the detection area polygons and the associated `detAreaHandles` variable as input arguments. The function outputs the detection area, as a matrix of datatype `uint8`, and a bounding box. The violet rectangle around the detection area represents the bounding box.

```
axesDet = gca; % Axes of the figure window containing the polygon handles
[detArea,bbox] = helperSaveDetectionArea(axesDet,detAreaHandles);
```




```
% Make detection area transparent by scaling colors
alphadata = double(detArea ~= 0)*0.5;
ax3 = getDetectionAreaAxes(display);
h = imagesc(ax3,[bbox(1) (bbox(1) + bbox(3))], ...
    -[bbox(2) (bbox(2) + bbox(4))], ...
    detArea,'AlphaData',alphadata);
colormap(ax3,[1 1 1; 1 1 0; 1 0 0; 0 0 0]);
% Plot detection area
plotObstacleDisplay(display)
```

Run Simulation

The detection area is divided into three levels as: black, red, and yellow. Each region is associated with a specific degree of danger:

- Black — Collision is imminent
- Red — High chance of collision
- Yellow — Apply caution measures

All obstacles that do not fall within the detection range as they are far from AGV. These are the primary steps involved in collision warning:

- Simulate 2-D lidar and extract point cloud data
- Segment point cloud data into obstacle clusters
- Loop over each obstacle to check for possible collisions

- Issue a warning based on the danger level of each obstacle
- Display obstacles close to the AGV

```
% Move AGV through waypoints
for ij = 27:size(traj,1)
    currentPose = traj(ij,:);
```

Simulate 2-D Lidar and Extract Point Cloud Data

Gather lidar readings of the map using the simulated sensor. Load the current pose of the AGV from the waypoints file. Use the `rangeSensor` System object you created to get range and angle measurement.

```
    currentPosition = currentPose(1:2);
    currentOrientation = currentPose(3);
    robotCurrentPose = [currentPosition(1:2) currentOrientation];
    % Retrieve lidar scans
    [ranges,angles] = lidar(robotCurrentPose,map);
    % Store 2-D scan as point cloud
    scan = lidarScan(ranges,angles);
    cart = scan.Cartesian;
    cart(:,3) = 0;
    pc = pointCloud(cart);
```

Segment Point Cloud Data into Obstacle Clusters

Use the `pcsegdist` function to segment the scanned point cloud into clusters, using minimum Euclidean distance between the points as the segmentation criterion.

```
    % Segment point cloud into clusters based on Euclidean distance
    minDistance = 0.9;
    [labels,numClusters] = pcsegdist(pc,minDistance);
```

Update Visualization Window with Map and Scan Data

```
    % Update display map
    updateMapDisplay(display,hRobot,currentPose);
    % Plot 2-D lidar scans
    plotLidarScan(display,scan,currentOrientation);
    % Delete obstacles from last scan to plot next scan line
    if exist('sc','var')
        delete(sc)
        clear sc
    end
```

Loop Over Each Obstacle to Find the Likelihood of Collisions

Loop through the clusters based on their labels, to extract the points located inside them.

```
    % Loop through all the clusters in pc
    for i = 1:numClusters
        c = find(labels == i);
        % XY coordinate extraction of obstacle
        xy = pc.Location(c,1:2);
```

Convert the world position of each obstacle into the camera coordinate system.

```
        % Convert to normalized coordinate system (0-> minimum location of detection
        % area, 1->maximum position of detection area)
```

```
a = [xy(:,1) xy(:,2)] - repmat(bbox([1 2]),[size(xy,1) 1]);
b = repmat(bbox([3 4]),[size(xy,1) 1]);
xy_org = a./b;
% Coordinate system (0,0)->(0,0), (1,1)->(width,height) of detArea
idx = floor(xy_org.*repmat([size(detArea,2) size(detArea,1)],[size(xy_org,1),1]));
```

Extract the indices of only the obstacle points that lie in the detection area.

```
% Extracts as an index only the coordinates in detArea
validIdx = 1 <= idx(:,1) & 1 <= idx(:,2) & ...
    idx(:,1) <= size(detArea,2) & idx(:,2) <= size(detArea,1);
```

For each valid obstacle point, find the associated value in the detection matrix. The maximum value of all associated points in the detection matrix determines the level of danger represented by that obstacle. Display a colored circle based on the danger level of the obstacle in the **Warning Color** pane of the visualization window.

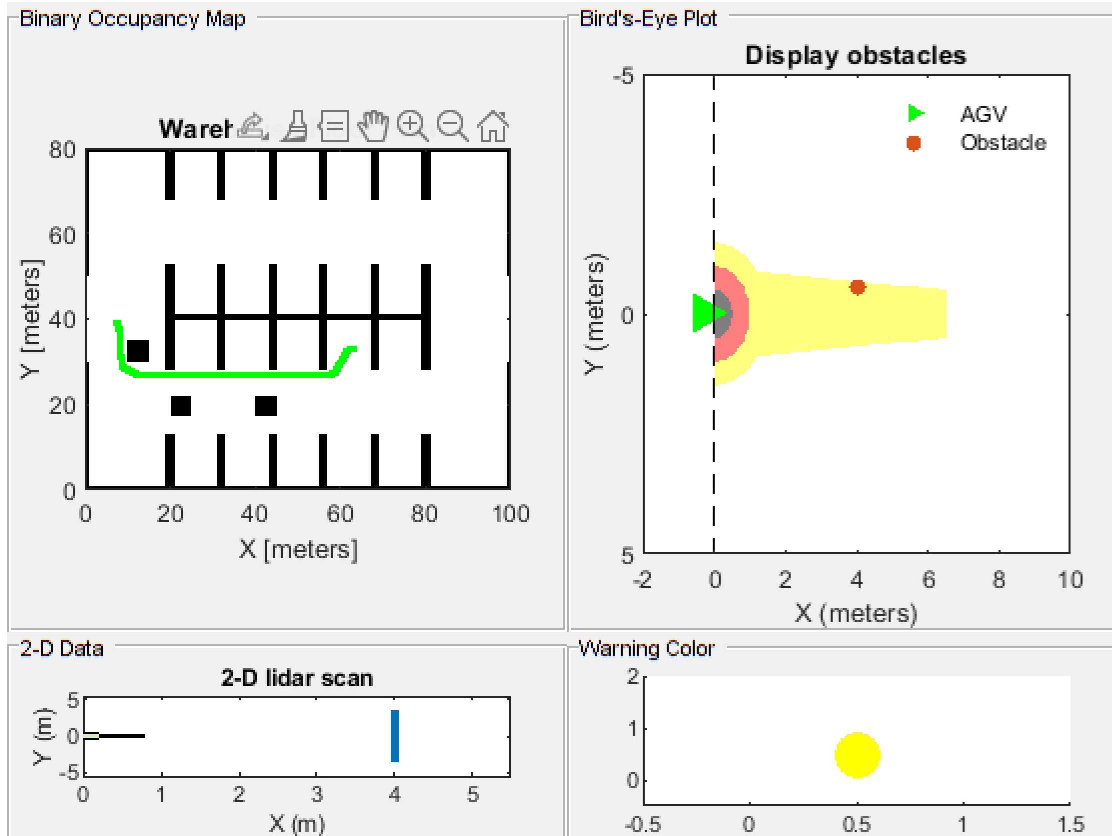
```
% Rounding the index and getting the level of each obstacle from detArea
cols = idx(validIdx,1);
rows = idx(validIdx,2);
levels = double(detArea(sub2ind(size(detArea),rows,cols)));
% Display a warning color representing the danger level. If the
% obstacle does not fall in the detection area, do not display a color.
circleDisplay(display,'white')
if~isempty(levels)
    [level,index] = max(levels);
    % Get the coordinates of obstacle that is in detection area
    rs = rows(index);
    cs = cols(index);
    nearxy = xy(idx(:,2) == rs & idx(:,1) == cs,:);
    switch level
        % Black region
        case 3
            circleDisplay(display,'black')
        % Red region
        case 2
            circleDisplay(display,'red')
        % Yellow region
        case 1
            circleDisplay(display,'yellow')
        % Default case
        otherwise
            circleDisplay(display,'white')
    end
end
```

Display Points of Obstacles Closest to the AGV

As most of the obstacles in the warehouse are linear and long, display only the point of each obstacle cluster closest to the AGV. Obstacles display as filled circles in the **Bird's-Eye Plot** pane of the visualization window.

```
if(isempty(levels))
    % Get nearest data item of each cluster
    nearxy = helperNearObstacles(xy);
end
% Display obstacles if exist in the mentioned range of axes3
sc(i,:) = displayObstacles(display,nearxy);
```

```
    end
    updateDisplay(display)
    pause(0.01)
end
```



Supporting Files

helperCreateBinaryOccupancyMap creates a warehouse map of the robot workspace

```
function map = helperCreateBinaryOccupancyMap()
% helperCreateBinaryOccupancyMap Creates a warehouse map with specific
% resolution passed as arguments to binaryOccupancyMap

map = binaryOccupancyMap(100, 80, 1);
occ = zeros(80, 100);
occ(1,:) = 1;
occ(end,:) = 1;
occ([1:30, 51:80],1) = 1;
occ([1:30, 51:80],end) = 1;
occ(40,20:80) = 1;
occ(28:52, [20:21 32:33 44:45 56:57 68:69 80:81]) = 1;
occ(1:12, [20:21 32:33 44:45 56:57 68:69 80:81]) = 1;
occ(end-12:end, [20:21 32:33 44:45 56:57 68:69 80:81]) = 1;
% Set occupancy value of locations
setOccupancy(map, occ);

% Add obstacles to the map at specific locations
% Inputs to helperAddObstacle are obstacleWidth, obstacleHeight and obstacleLocation.
```

```
helperAddObstacle(map, 5, 5, [10,30]);
helperAddObstacle(map, 5, 5, [20,17]);
helperAddObstacle(map, 5, 5, [40,17]);
end

%helperAddObstacle Adds obstacles to the occupancy map
function helperAddObstacle(map, obstacleWidth, obstacleHeight, obstacleLocation)
values = ones(obstacleHeight, obstacleWidth);
setOccupancy(map, obstacleLocation, values)
end
```

See also

binaryOccupancyMap (Navigation Toolbox) | lidarScan | rangeSensor | pcsegdist

Track Vehicles Using Lidar: From Point Cloud to Track List

This example shows you how to track vehicles using measurements from a lidar sensor mounted on top of an ego vehicle. Lidar sensors report measurements as a point cloud. The example illustrates the workflow in MATLAB® for processing the point cloud and tracking the objects. For a Simulink® version of the example, refer to "Track Vehicles Using Lidar Data in Simulink" (Sensor Fusion and Tracking Toolbox).The lidar data used in this example is recorded from a highway driving scenario. In this example, you use the recorded data to track vehicles with a joint probabilistic data association (JPDA) tracker and an interacting multiple model (IMM) approach.

3-D Bounding Box Detector Model

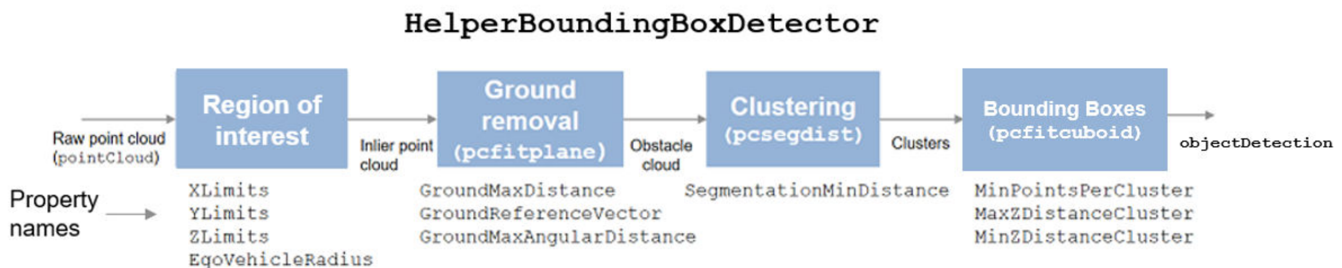
Due to high resolution capabilities of the lidar sensor, each scan from the sensor contains a large number of points, commonly known as a point cloud. This raw data must be preprocessed to extract objects of interest, such as cars, cyclists, and pedestrians. In this example, you use a classical segmentation algorithm using a distance-based clustering algorithm. For more details about segmentation of lidar data into objects such as the ground plane and obstacles, refer to the "Ground Plane and Obstacle Detection Using Lidar" (Automated Driving Toolbox) example. For a deep learning segmentation workflow, refer to the "Detect, Classify, and Track Vehicles Using Lidar" on page 1-23 example. In this example, the point clouds belonging to obstacles are further classified into clusters using the `pcsegdist` function, and each cluster is converted to a bounding box detection with the following format:

$$[x\ y\ z\ \theta\ l\ w\ h]$$

x, y and z refer to the x-, y- and z-positions of the bounding box, θ refers to its yaw angle and l, w and h refer to its length, width, and height, respectively. The `pcfitcuboid` function uses L-shape fitting algorithm to determine the yaw angle of the bounding box.

The detector is implemented by a supporting class `HelperBoundingBoxDetector`, which wraps around point cloud segmentation and clustering functionalities. An object of this class accepts a `pointCloud` input and returns a list of `objectDetection` objects with bounding box measurements.

The diagram shows the processes involved in the bounding box detector model and the Lidar Toolbox™ functions used to implement each process. It also shows the properties of the supporting class that control each process.



The lidar data is available at the following location: https://ssd.mathworks.com/supportfiles/lidar/data/TrackVehiclesUsingLidarExampleData.zip

Download the data files into your temporary directory, whose location is specified by MATLAB's `tempdir` function. If you want to place the files in a different folder, change the directory name in the subsequent instructions.

```
% Load data if unavailable. The lidar data is stored as a cell array of
% pointCloud objects.
if ~exist('lidarData','var')
    dataURL = 'https://ssd.mathworks.com/supportfiles/lidar/data/TrackVehiclesUsingLidarExampleDa
    datasetFolder = fullfile(tempdir,'LidarExampleDataset');
    if ~exist(datasetFolder,'dir')
       unzip(dataURL,datasetFolder);
    end
    % Specify initial and final time for simulation.
    initTime = 0;
    finalTime = 35;
    [lidarData, imageData] = loadLidarAndImageData(datasetFolder,initTime,finalTime);
end

% Set random seed to generate reproducible results.
S = rng(2018);

% A bounding box detector model.
detectorModel = HelperBoundingBoxDetector(...
    'XLimits',[-50 75],...              % min-max
    'YLimits',[-5 5],...                % min-max
    'ZLimits',[-2 5],...                % min-max
    'SegmentationMinDistance',1.8,...   % minimum Euclidian distance
    'MinDetectionsPerCluster',1,...     % minimum points per cluster
    'MeasurementNoise',blkdiag(0.25*eye(3),25,eye(3)),...       % measurement noise in detection
    'GroundMaxDistance',0.3);           % maximum distance of ground points from ground plane
```

Target State and Sensor Measurement Model

The first step in tracking an object is defining its state, and the models that define the transition of state and the corresponding measurement. These two sets of equations are collectively known as the state-space model of the target. To model the state of vehicles for tracking using lidar, this example uses a cuboid model with following convention:

$$x = [x_{kin}\ \theta\ l\ w\ h]$$

x_{kin} refers to the portion of the state that controls the kinematics of the motion center, and θ is the yaw angle. The length, width, height of the cuboid are modeled as a constants, whose estimates evolve in time during correction stages of the filter.

In this example, you use two state-space models: a constant velocity (cv) cuboid model and a constant turn-rate (ct) cuboid model. These models differ in the way they define the kinematic part of the state, as described below:

$$x_{cv} = [x\ \dot{x}\ y\ \dot{y}\ z\ \dot{z}\ \theta\ l\ w\ h]$$

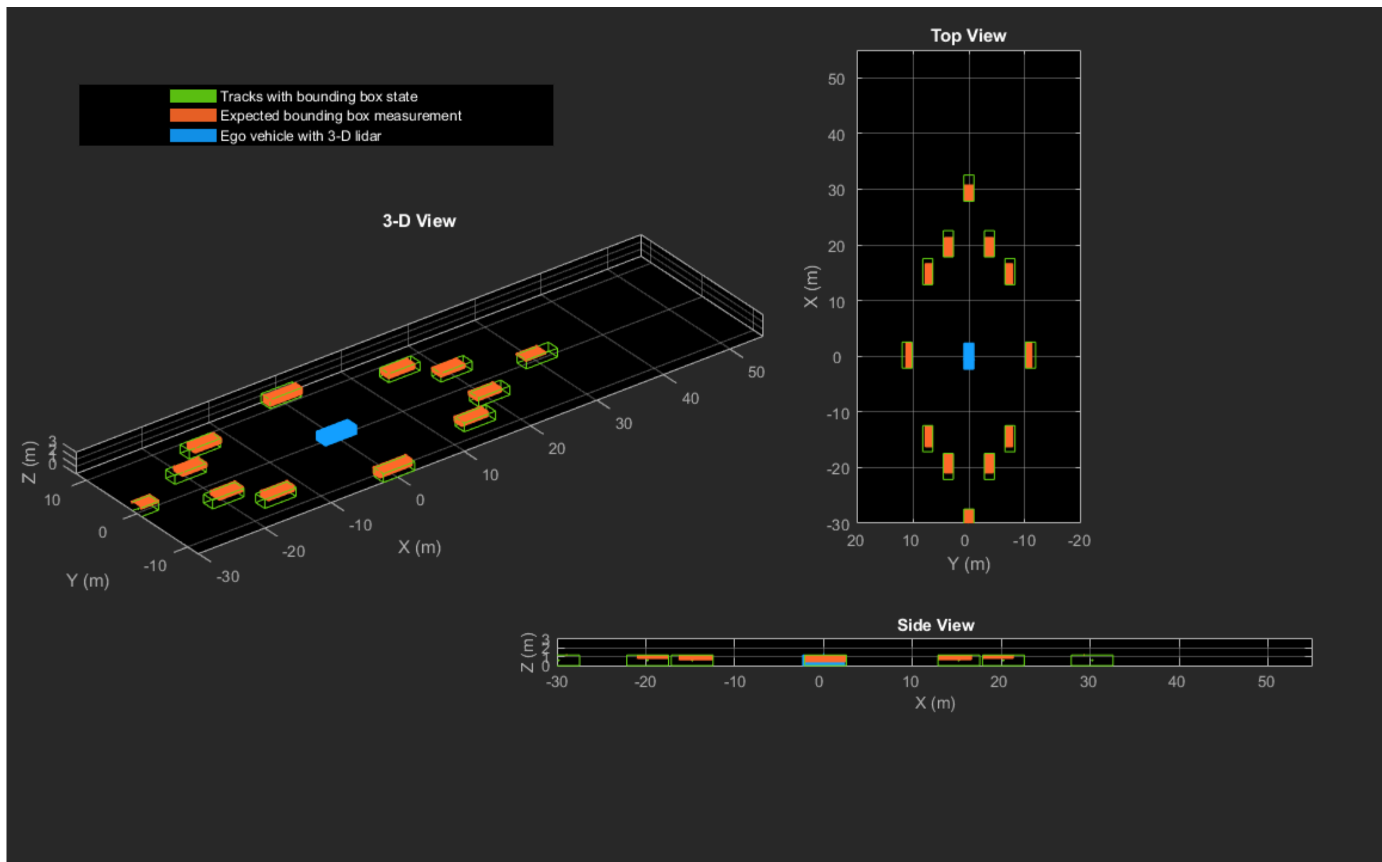
$$x_{ct} = [x\ \dot{x}\ y\ \dot{y}\ \dot{\theta}\ z\ \dot{z}\ \theta\ l\ w\ h]$$

For information about their state transition, refer to the `helperConstvelCuboid` and `helperConstturnCuboid` functions used in this example.

The `helperCvmeasCuboid` and `helperCtmeasCuboid` measurement models describe how the sensor perceives the constant velocity and constant turn-rate states respectively, and they return

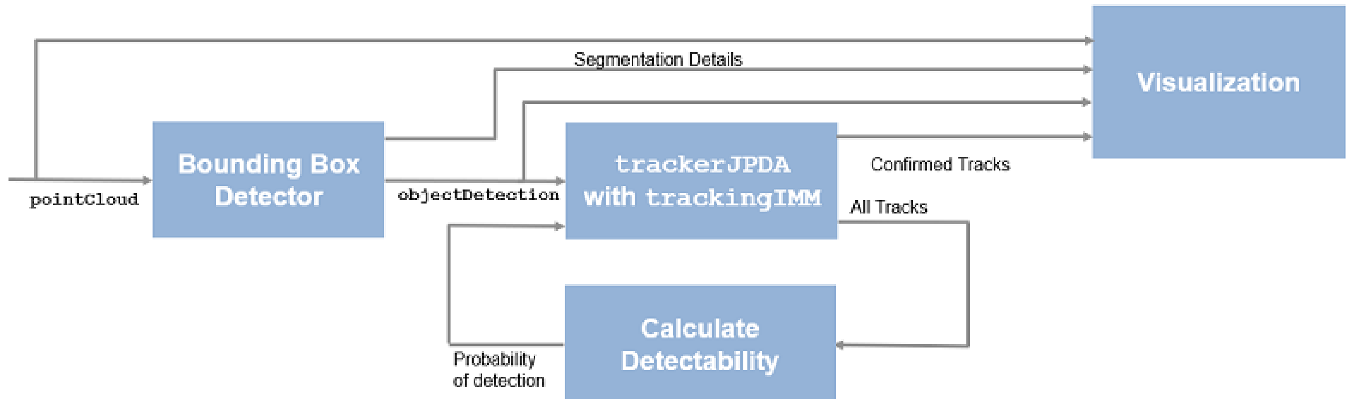
bounding box measurements. Because the state contains information about size of the target, the measurement model includes the effect of center-point offset and bounding box shrinkage, as perceived by the sensor, due to effects like self-occlusion [1]. This effect is modeled by a shrinkage factor that is directly proportional to the distance from the tracked vehicle to the sensor.

The image below demonstrates the measurement model operating at different state-space samples. Notice the modeled effects of bounding box shrinkage and center-point offset as the objects move around the ego vehicle.



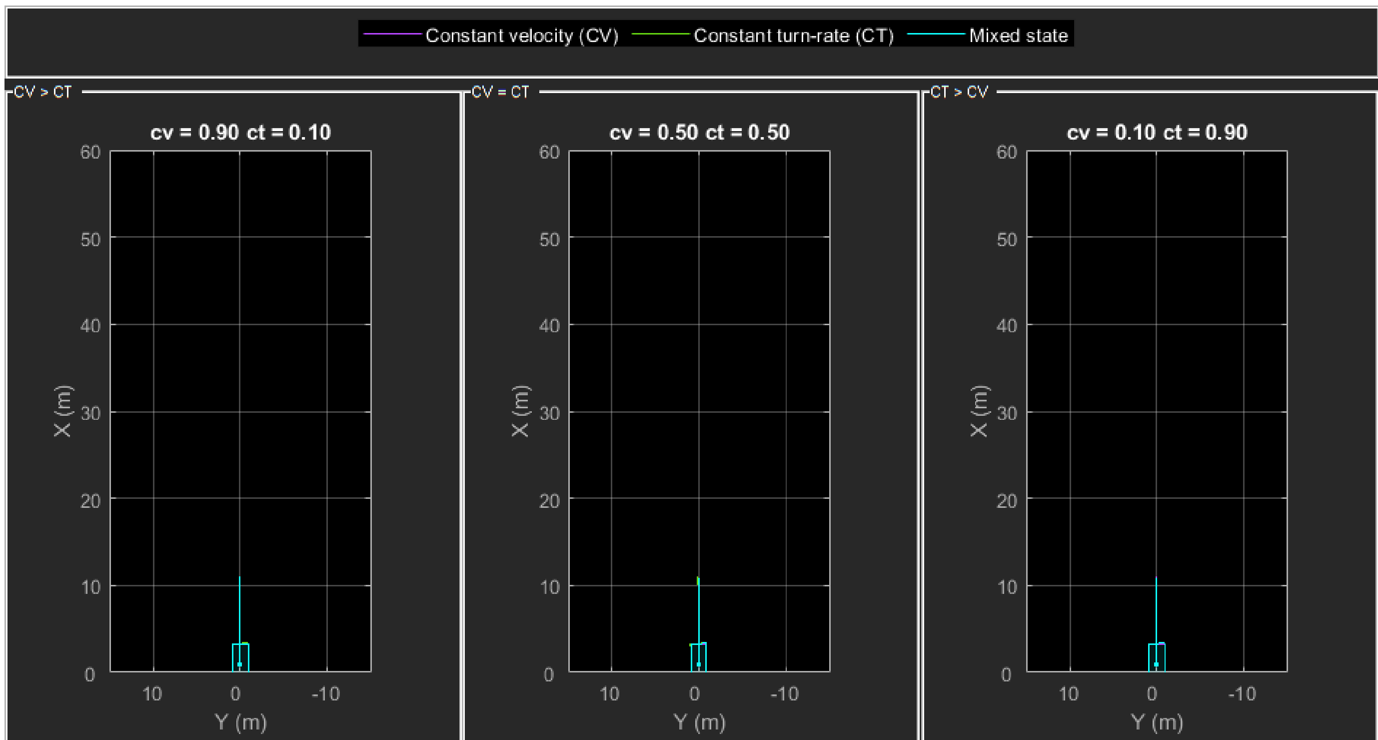
Set Up Tracker and Visualization

The image below shows the complete workflow to obtain a list of tracks from a pointCloud input.

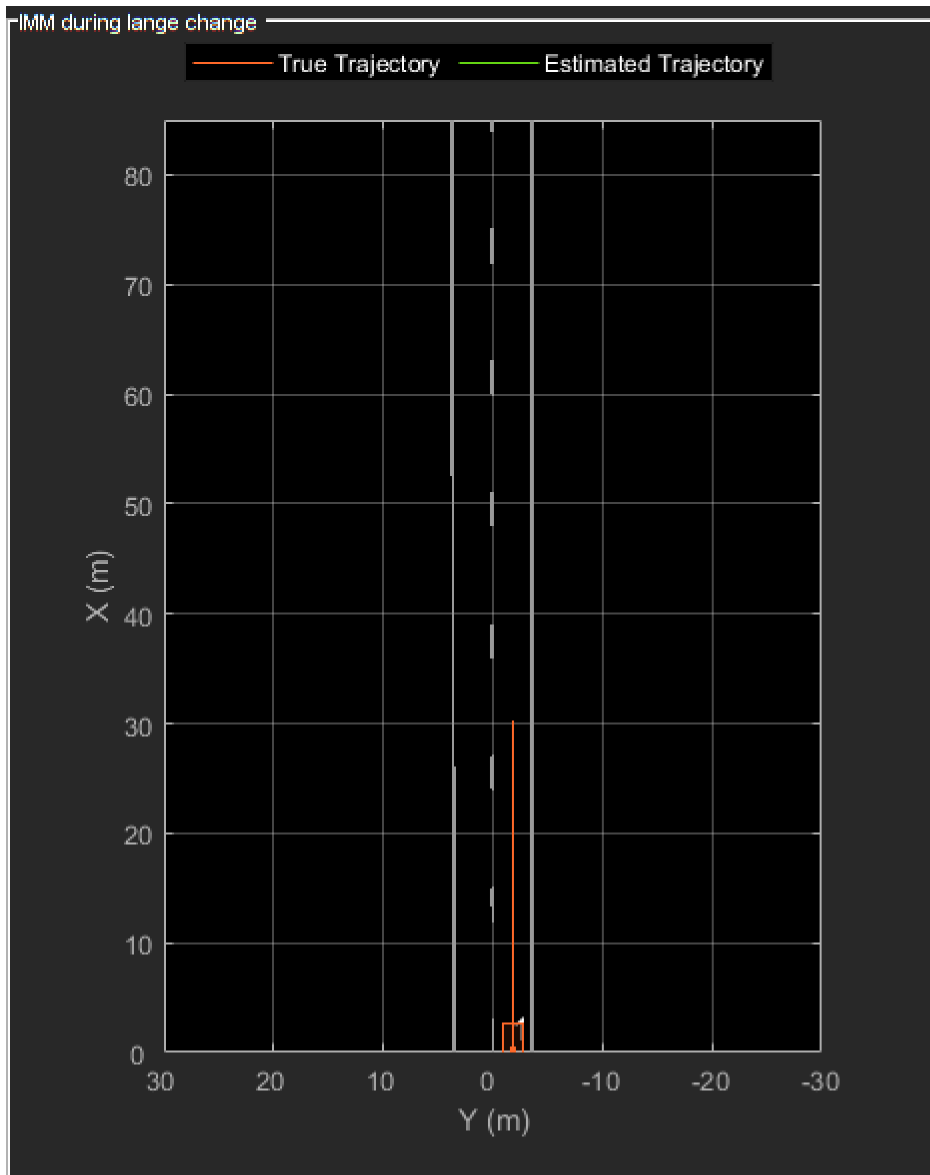


Now, set up the tracker and the visualization used in the example.

A joint probabilistic data association tracker (`trackerJPDA`) coupled with an IMM filter (`trackingIMM`) is used to track objects in this example. The IMM filter uses a constant velocity and constant turn-rate model and is initialized using the supporting function, `helperInitIMMFilter`, included with this example. The IMM approach helps a track to switch between motion models and thus achieve good estimation accuracy during events like maneuvering or lane changing. The animation below shows the effect of mixing the constant velocity and constant turn-rate model during prediction stages of the IMM filter.



The IMM filter updates the probability of each model when it is corrected with detections from the object. The animation below shows the estimated trajectory of a vehicle during a lane change event and the corresponding estimated probabilities of each model.



Set the `HasDetectableTrackIDsInput` property of the tracker as `true`, which enables you to specify a state-dependent probability of detection. The detection probability of a track is calculated by the `helperCalcDetectability` function, listed at the end of this example.

```
assignmentGate = [75 1000]; % Assignment threshold;
confThreshold = [7 10];    % Confirmation threshold for history logic
delThreshold = [8 10];     % Deletion threshold for history logic
Kc = 1e-9;                 % False-alarm rate per unit volume

% IMM filter initialization function
filterInitFcn = @helperInitIMMFilter;

% A joint probabilistic data association tracker with IMM filter
tracker = trackerJPDA('FilterInitializationFcn',filterInitFcn,...
    'TrackLogic','History',...
    'AssignmentThreshold',assignmentGate,...
```

```
    'ClutterDensity',Kc,...
    'ConfirmationThreshold',confThreshold,...
    'DeletionThreshold',delThreshold,...
    'HasDetectableTrackIDsInput',true,...
    'InitializationThreshold',0,...
    'HitMissThreshold',0.1);
```

The visualization is divided into these main categories:

1. Lidar Preprocessing and Tracking - This display shows the raw point cloud, segmented ground, and obstacles. It also shows the resulting detections from the detector model and the tracks of vehicles generated by the tracker.
2. Ego Vehicle Display - This display shows the 2-D bird's-eye view of the scenario. It shows the obstacle point cloud, bounding box detections, and the tracks generated by the tracker. For reference, it also displays the image recorded from a camera mounted on the ego vehicle and its field of view.
3. Tracking Details - This display shows the scenario zoomed around the ego vehicle. It also shows finer tracking details, such as error covariance in estimated position of each track and its motion model probabilities, denoted by cv and ct.

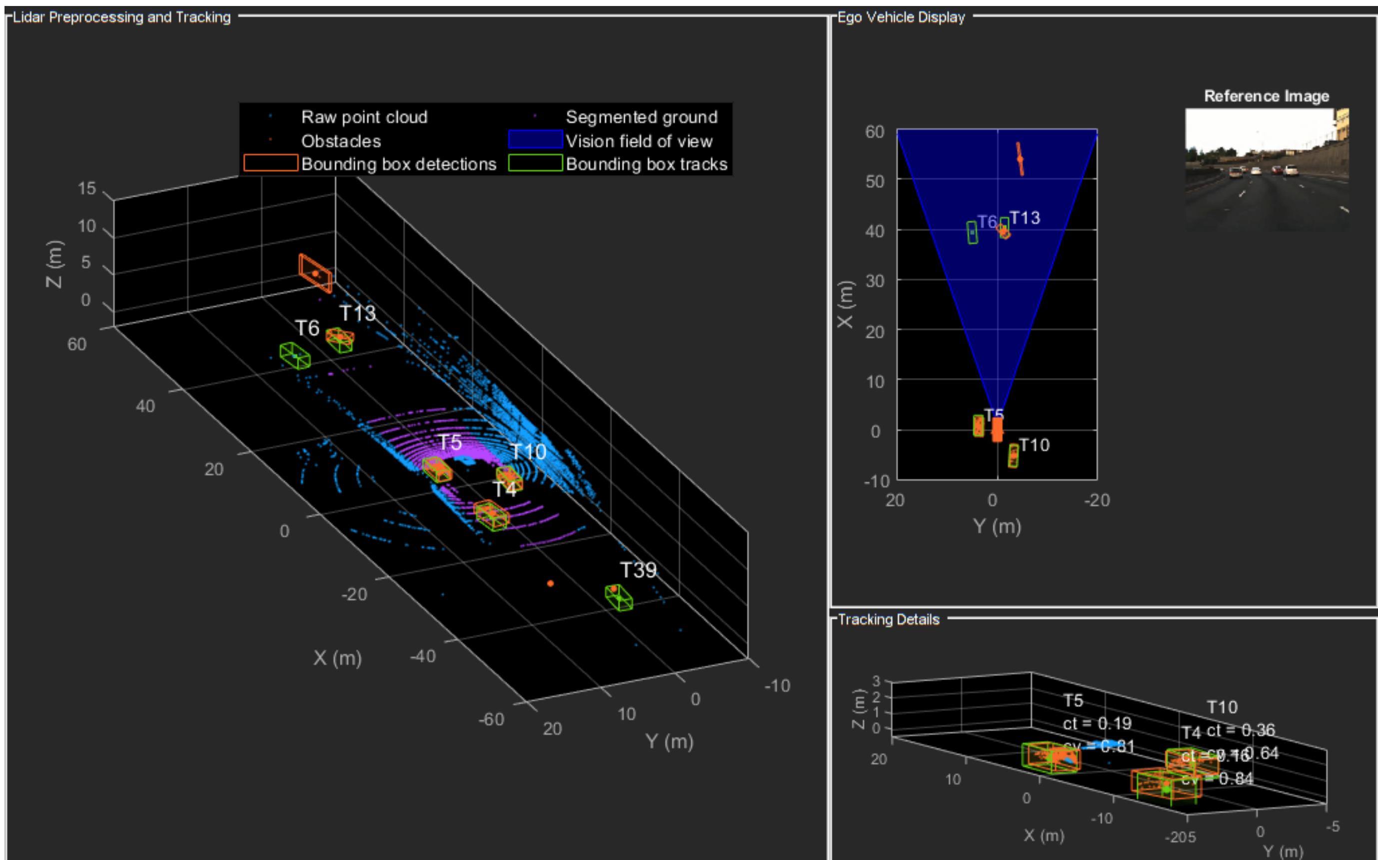
```
% Create display
displayObject = HelperLidarExampleDisplay(imageData{1},...
    'PositionIndex',[1 3 6],...
    'VelocityIndex',[2 4 7],...
    'DimensionIndex',[9 10 11],...
    'YawIndex',8,...
    'MovieName','',...  % Specify a movie name to record a movie.
    'RecordGIF',false); % Specify true to record new GIFs
```

Loop Through Data

Loop through the recorded lidar data, generate detections from the current point cloud using the detector model and then process the detections using the tracker.

```
time = 0;       % Start time
dT = 0.1;       % Time step

% Initiate all tracks.
allTracks = struct([]);

% Initiate variables for comparing MATLAB and MEX simulation.
numTracks = zeros(numel(lidarData),2);

% Loop through the data
for i = 1:numel(lidarData)
    % Update time
    time = time + dT;

    % Get current lidar scan
    currentLidar = lidarData{i};

    % Generator detections from lidar scan.
    [detections,obstacleIndices,groundIndices,croppedIndices] = detectorModel(currentLidar,time)

    % Calculate detectability of each track.
    detectableTracksInput = helperCalcDetectability(allTracks,[1 3 6]);
```

```
    % Pass detections to track.
    [confirmedTracks,tentativeTracks,allTracks,info] = tracker(detections,time,detectableTracksIn
    numTracks(i,1) = numel(confirmedTracks);

    % Get model probabilities from IMM filter of each track using
    % getTrackFilterProperties function of the tracker.
    modelProbs = zeros(2,numel(confirmedTracks));
    for k = 1:numel(confirmedTracks)
        c1 = getTrackFilterProperties(tracker,confirmedTracks(k).TrackID,'ModelProbabilities');
        modelProbs(:,k) = c1{1};
    end

    % Update display
    if isvalid(displayObject.PointCloudProcessingDisplay.ObstaclePlotter)
        % Get current image scan for reference image
        currentImage = imageData{i};

        % Update display object
        displayObject(detections,confirmedTracks,currentLidar,obstacleIndices,...
            groundIndices,croppedIndices,currentImage,modelProbs);
    end

    % Snap a figure at time = 18
    if abs(time - 18) < dT/2
        snapnow(displayObject);
    end
end

% Write movie if requested
if ~isempty(displayObject.MovieName)
    writeMovie(displayObject);
end

% Write new GIFs if requested.
if displayObject.RecordGIF
    % second input is start frame, third input is end frame and last input
    % is a character vector specifying the panel to record.
    writeAnimatedGIF(displayObject,10,170,'trackMaintenance','ego');
    writeAnimatedGIF(displayObject,310,330,'jpda','processing');
    writeAnimatedGIF(displayObject,120,140,'imm','details');
end
```



The figure above shows the three displays at time = 18 seconds. The tracks are represented by green bounding boxes. The bounding box detections are represented by orange bounding boxes. The detections also have orange points inside them, representing the point cloud segmented as obstacles. The segmented ground is shown in purple. The cropped or discarded point cloud is shown in blue.

Generate C Code

You can generate C code from the MATLAB® code for the tracking and the preprocessing algorithm using MATLAB Coder™. C code generation enables you to accelerate MATLAB code for simulation. To generate C code, the algorithm must be restructured as a MATLAB function, which can be compiled into a MEX file or a shared library. For this purpose, the point cloud processing algorithm and the tracking algorithm is restructured into a MATLAB function, `mexLidarTracker`. Some variables are defined as `persistent` to preserve their state between multiple calls to the function (see `persistent`). The inputs and outputs of the function can be observed in the function description provided in the "Supporting Files" section at the end of this example.

MATLAB coder requires specifying the properties of all the input arguments. An easy way to do this is by defining the input properties by example at the command line using the `-args` option. For more information, see “Define Input Properties by Example at the Command Line” (MATLAB Coder). Note that the top-level input arguments cannot be objects of the `handle` class. Therefore, the function accepts the *x*, *y* and *z* locations of the point cloud as an input. From the stored point cloud, this information can be extracted using the `Location` property of the `pointCloud` object. This information is also directly available as the raw data from the lidar sensor.

```
% Input lists
inputExample = {lidarData{1}.Location, 0};

% Create configuration for MEX generation
cfg = coder.config('mex');

% Replace cfg with the following to generate static library and perform
% software-in-the-loop simulation. This requires Embedded Coder license.
%
% cfg = coder.config('lib'); % Static library
% cfg.VerificationMode = 'SIL'; % Software-in-the-loop

% Generate code if file does not exist.
if ~exist('mexLidarTracker_mex','file')
    h = msgbox({'Generating code. This may take a few minutes...';'This message box will close wl
    % -config allows specifying the codegen configuration
    % -o allows specifying the name of the output file
    codegen -config cfg -o mexLidarTracker_mex mexLidarTracker -args inputExample
    close(h);
else
    clear mexLidarTracker_mex;
end
```

Rerun simulation with MEX Code

Rerun the simulation using the generated MEX code, `mexLidarTracker_mex`. Reset time

```
time = 0;

for i = 1:numel(lidarData)
    time = time + dT;

    currentLidar = lidarData{i};

    [detectionsMex,obstacleIndicesMex,groundIndicesMex,croppedIndicesMex,...
        confirmedTracksMex, modelProbsMex] = mexLidarTracker_mex(currentLidar.Location,time);

    % Record data for comparison with MATLAB execution.
    numTracks(i,2) = numel(confirmedTracksMex);
end
```

Compare results between MATLAB and MEX Execution

```
disp(isequal(numTracks(:,1),numTracks(:,2)));
```

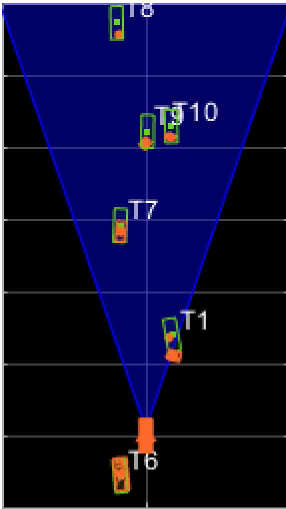
```
   1
```

Notice that the number of confirmed tracks is the same for MATLAB and MEX code execution. This assures that the lidar preprocessing and tracking algorithm returns the same results with generated C code as with the MATLAB code.

Results

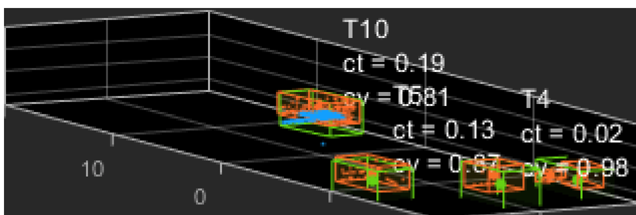
Now, analyze different events in the scenario and understand how the combination of lidar measurement model, joint probabilistic data association, and interacting multiple model filter, helps achieve a good estimation of the vehicle tracks.

Track Maintenance



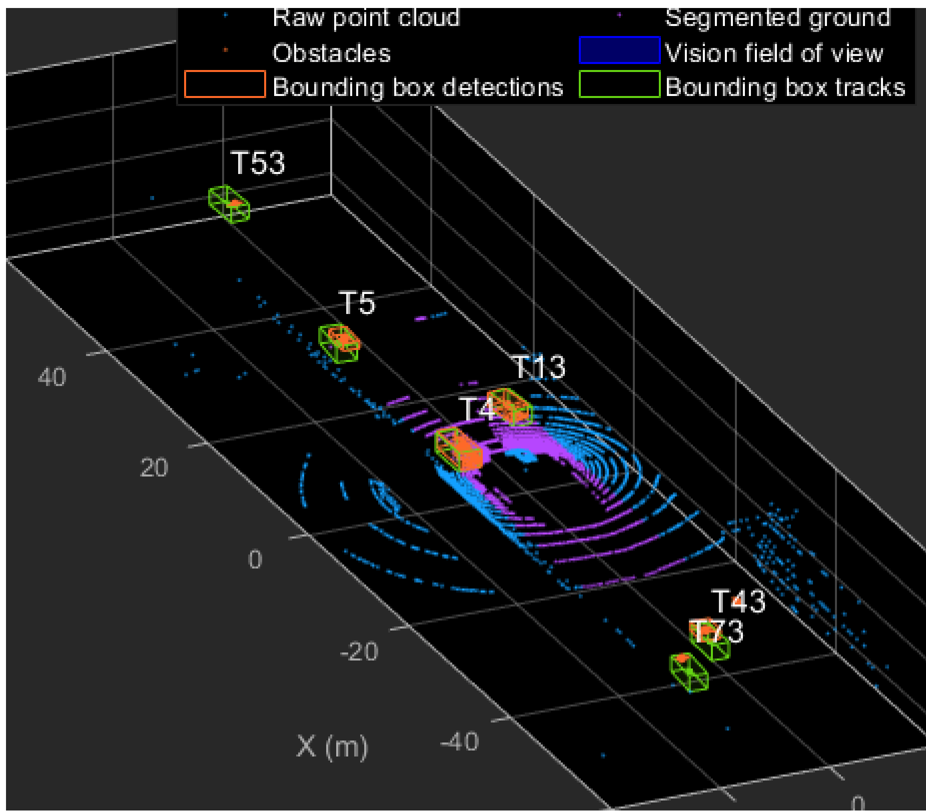
The animation above shows the simulation between time = 3 seconds and time = 16 seconds. Notice that tracks such as T10 and T6 maintain their IDs and trajectory during the time span. However, track T9 is lost because the tracked vehicle was missed (not detected) for a long time by the sensor. Also, notice that the tracked objects are able to maintain their shape and kinematic center by positioning the detections onto the visible portions of the vehicles. For example, as Track T7 moves forward, bounding box detections start to fall on its visible rear portion and the track maintains the actual size of the vehicle. This illustrates the offset and shrinkage effect modeled in the measurement functions.

Capturing Maneuvers



The animation shows that using an IMM filter helps the tracker to maintain tracks on maneuvering vehicles. Notice that the vehicle tracked by T4 changes lanes behind the ego vehicle. The tracker is able maintain a track on the vehicle during this maneuvering event. Also notice in the display that its probability of following the constant turn model, denoted by ct, increases during the lane change maneuver.

Joint Probabilistic Data Association



This animation shows that using a joint probabilistic data association tracker helps in maintaining tracks during ambiguous situations. Here, vehicles tracked by T43 and T73, have a low probability of detection due to their large distance from the sensor. Notice that the tracker is able to maintain tracks during events when one of the vehicles is not detected. During the event, the tracks first coalesce, which is a known phenomenon in JPDA, and then separate as soon as the vehicle was detected again.

Summary

This example showed how to use a JPDA tracker with an IMM filter to track objects using a lidar sensor. You learned how a raw point cloud can be preprocessed to generate detections for conventional trackers, which assume one detection per object per sensor scan. You also learned how to define a cuboid model to describe the kinematics, dimensions, and measurements of extended objects being tracked by the JPDA tracker. In addition, you generated C code from the algorithm and verified its execution results with the MATLAB simulation.

Supporting Files

helperLidarModel

This function defines the lidar model to simulate shrinkage of the bounding box measurement and center-point offset. This function is used in the `helperCvmeasCuboid` and `helperCtmeasCuboid` functions to obtain bounding box measurement from the state.

```
function meas = helperLidarModel(pos,dim,yaw)
% This function returns the expected bounding box measurement given an
% object's position, dimension, and yaw angle.
```

```
% Copyright 2019 The MathWorks, Inc.

% Get x,y and z.
x = pos(1,:);
y = pos(2,:);
z = pos(3,:) - 2; % lidar mounted at height = 2 meters.

% Get spherical measurement.
[az,~,r] = cart2sph(x,y,z);

% Shrink rate
s = 3/50; % 3 meters radial length at 50 meters.
sz = 2/50; % 2 meters height at 50 meters.

% Get length, width and height.
L = dim(1,:);
W = dim(2,:);
H = dim(3,:);

az = az - deg2rad(yaw);

% Shrink length along radial direction.
Lshrink = min(L,abs(s*r.*(cos(az))));
Ls = L - Lshrink;

% Shrink width along radial direction.
Wshrink = min(W,abs(s*r.*(sin(az))));
Ws = W - Wshrink;

% Shrink height.
Hshrink = min(H,sz*r);
Hs = H - Hshrink;

% Similar shift is for x and y directions.
shiftX = Lshrink.*cosd(yaw) + Wshrink.*sind(yaw);
shiftY = Lshrink.*sind(yaw) + Wshrink.*cosd(yaw);
shiftZ = Hshrink;

% Modeling the affect of box origin offset
x = x - sign(x).*shiftX/2;
y = y - sign(y).*shiftY/2;
z = z + shiftZ/2 + 2;

% Measurement format
meas = [x;y;z;yaw;Ls;Ws;Hs];

end
```

helperInverseLidarModel

This function defines the inverse lidar model to initiate a tracking filter using a lidar bounding box measurement. This function is used in the `helperInitIMMFilter` function to obtain state estimates from a bounding box measurement.

```
function [pos,posCov,dim,dimCov,yaw,yawCov] = helperInverseLidarModel(meas,measCov)
```

```
% This function returns the position, dimension, yaw using a bounding
% box measurement.

% Copyright 2019 The MathWorks, Inc.

% Shrink rate.
s = 3/50;
sz = 2/50;

% x,y and z of measurement
x = meas(1,:);
y = meas(2,:);
z = meas(3,:);

[az,~,r] = cart2sph(x,y,z);

% Shift x and y position.
Lshrink = abs(s*r.*(cos(az)));
Wshrink = abs(s*r.*(sin(az)));
Hshrink = sz*r;

shiftX = Lshrink;
shiftY = Wshrink;
shiftZ = Hshrink;

x = x + sign(x).*shiftX/2;
y = y + sign(y).*shiftY/2;
z = z - shiftZ/2;

pos = [x;y;z];
posCov = measCov(1:3,1:3,:);

yaw = meas(4,:);
yawCov = measCov(4,4,:);

% Dimensions are initialized for a standard passenger car with low
% uncertainity.
dim = [4.7;1.8;1.4];
dimCov = 0.01*eye(3);
end
```

HelperBoundingBoxDetector

This is the supporting class `HelperBoundingBoxDetector` to accept a point cloud input and return a list of `objectDetection`

```
classdef HelperBoundingBoxDetector < matlab.System
    % HelperBoundingBoxDetector A helper class to segment the point cloud
    % into bounding box detections.
    % The step call to the object does the following things:
    %
    % 1. Removes point cloud outside the limits.
    % 2. From the survived point cloud, segments out ground
    % 3. From the obstacle point cloud, forms clusters and puts bounding
    %    box on each cluster.
```

```
    % Cropping properties
    properties
        % XLimits XLimits for the scene
        XLimits = [-70 70];
        % YLimits YLimits for the scene
        YLimits = [-6 6];
        % ZLimits ZLimits fot the scene
        ZLimits = [-2 10];
    end

    % Ground Segmentation Properties
    properties
        % GroundMaxDistance Maximum distance of point to the ground plane
        GroundMaxDistance = 0.3;
        % GroundReferenceVector Reference vector of ground plane
        GroundReferenceVector = [0 0 1];
        % GroundMaxAngularDistance Maximum angular distance of point to reference vector
        GroundMaxAngularDistance = 5;
    end

    % Bounding box Segmentation properties
    properties
        % SegmentationMinDistance Distance threshold for segmentation
        SegmentationMinDistance = 1.6;
        % MinDetectionsPerCluster Minimum number of detections per cluster
        MinDetectionsPerCluster = 2;
        % MaxZDistanceCluster Maximum Z-coordinate of cluster
        MaxZDistanceCluster = 3;
        % MinZDistanceCluster Minimum Z-coordinate of cluster
        MinZDistanceCluster = -3;
    end

    % Ego vehicle radius to remove ego vehicle point cloud.
    properties
        % EgoVehicleRadius Radius of ego vehicle
        EgoVehicleRadius = 3;
    end

    properties
        % MeasurementNoise Measurement noise for the bounding box detection
        MeasurementNoise = blkdiag(eye(3),10,eye(3));
    end

    properties (Nontunable)
        MeasurementParameters = struct.empty(0,1);
    end

    methods
        function obj = HelperBoundingBoxDetector(varargin)
            setProperties(obj,nargin,varargin{:})
        end
    end

    methods (Access = protected)
        function [bboxDets,obstacleIndices,groundIndices,croppedIndices] = stepImpl(obj,currentPo
            % Crop point cloud
            [pcSurvived,survivedIndices,croppedIndices] = cropPointCloud(currentPointCloud,obj.XL
            % Remove ground plane
```

```
                [pcObstacles,obstacleIndices,groundIndices] = removeGroundPlane(pcSurvived,obj.Ground
                % Form clusters and get bounding boxes
                detBBoxes = getBoundingBoxes(pcObstacles,obj.SegmentationMinDistance,obj.MinDetection
                % Assemble detections
                if isempty(obj.MeasurementParameters)
                    measParams = {};
                else
                    measParams = obj.MeasurementParameters;
                end
                bboxDets = assembleDetections(detBBoxes,obj.MeasurementNoise,measParams,time);
            end
        end
    end

    function detections = assembleDetections(bboxes,measNoise,measParams,time)
    % This method assembles the detections in objectDetection format.
    numBoxes = size(bboxes,2);
    detections = cell(numBoxes,1);
    for i = 1:numBoxes
        detections{i} = objectDetection(time,cast(bboxes(:,i),'double'),...
            'MeasurementNoise',double(measNoise),'ObjectAttributes',struct,...
            'MeasurementParameters',measParams);
    end
    end

    function bboxes = getBoundingBoxes(ptCloud,minDistance,minDetsPerCluster,maxZDistance,minZDistanc
        % This method fits bounding boxes on each cluster with some basic
        % rules.
        % Cluster must have at least minDetsPerCluster points.
        % Its mean z must be between maxZDistance and minZDistance.
        % length, width and height are calculated using min and max from each
        % dimension.
        [labels,numClusters] = pcsegdist(ptCloud,minDistance);
        pointData = ptCloud.Location;
        bboxes = nan(7,numClusters,'like',pointData);
        isValidCluster = false(1,numClusters);
        for i = 1:numClusters
            thisPointData = pointData(labels == i,:);
            meanPoint = mean(thisPointData,1);
            if size(thisPointData,1) > minDetsPerCluster && ...
                    meanPoint(3) < maxZDistance && meanPoint(3) > minZDistance
                cuboid = pcfitcuboid(pointCloud(thisPointData));
                yaw = cuboid.Orientation(3);
                L = cuboid.Dimensions(1);
                W = cuboid.Dimensions(2);
                H = cuboid.Dimensions(3);
                if abs(yaw) > 45
                    possibles = yaw + [-90;90];
                    [~,toChoose] = min(abs(possibles));
                    yaw = possibles(toChoose);
                    temp = L;
                    L = W;
                    W = temp;
                end
                bboxes(:,i) = [cuboid.Center yaw L W H]';
                isValidCluster(i) = L < 20 & W < 20;
            end
        end
```

```
    bboxes = bboxes(:,isValidCluster);
end

function [ptCloudOut,obstacleIndices,groundIndices] = removeGroundPlane(ptCloudIn,maxGroundDist,r
    % This method removes the ground plane from point cloud using
    % pcfitplane.
    [~,groundIndices,outliers] = pcfitplane(ptCloudIn,maxGroundDist,referenceVector,maxAngularDis
    ptCloudOut = select(ptCloudIn,outliers);
    obstacleIndices = currentIndices(outliers);
    groundIndices = currentIndices(groundIndices);
end

function [ptCloudOut,indices,croppedIndices] = cropPointCloud(ptCloudIn,xLim,yLim,zLim,egoVehicle
    % This method selects the point cloud within limits and removes the
    % ego vehicle point cloud using findNeighborsInRadius
    locations = ptCloudIn.Location;
    locations = reshape(locations,[],3);
    insideX = locations(:,1) < xLim(2) & locations(:,1) > xLim(1);
    insideY = locations(:,2) < yLim(2) & locations(:,2) > yLim(1);
    insideZ = locations(:,3) < zLim(2) & locations(:,3) > zLim(1);
    inside = insideX & insideY & insideZ;

    % Remove ego vehicle
    nearIndices = findNeighborsInRadius(ptCloudIn,[0 0 0],egoVehicleRadius);
    nonEgoIndices = true(ptCloudIn.Count,1);
    nonEgoIndices(nearIndices) = false;
    validIndices = inside & nonEgoIndices;
    indices = find(validIndices);
    croppedIndices = find(~validIndices);
    ptCloudOut = select(ptCloudIn,indices);
end
```

mexLidarTracker

This function implements the point cloud preprocessing display and the tracking algorithm using a functional interface for code generation.

```
function [detections,obstacleIndices,groundIndices,croppedIndices,...
    confirmedTracks, modelProbs] = mexLidarTracker(ptCloudLocations,time)

persistent detectorModel tracker detectableTracksInput currentNumTracks


if isempty(detectorModel) || isempty(tracker) || isempty(detectableTracksInput) || isempty(curren

    % Use the same starting seed as MATLAB to reproduce results in SIL
    % simulation.
    rng(2018);

    % A bounding box detector model.
    detectorModel = HelperBoundingBoxDetector(...
                    'XLimits',[-50 75],...              % min-max
                    'YLimits',[-5 5],...                % min-max
                    'ZLimits',[-2 5],...                % min-max
```

```
                    'SegmentationMinDistance',1.8,...   % minimum Euclidian distance
                    'MinDetectionsPerCluster',1,...     % minimum points per cluster
                    'MeasurementNoise',blkdiag(0.25*eye(3),25,eye(3)),...       % measurement noi
                    'GroundMaxDistance',0.3);           % maximum distance of ground points from

    assignmentGate = [75 1000]; % Assignment threshold;
    confThreshold = [7 10];    % Confirmation threshold for history logic
    delThreshold = [8 10];     % Deletion threshold for history logic
    Kc = 1e-9;                 % False-alarm rate per unit volume

    filterInitFcn = @helperInitIMMFilter;

    tracker = trackerJPDA('FilterInitializationFcn',filterInitFcn,...
                      'TrackLogic','History',...
                      'AssignmentThreshold',assignmentGate,...
                      'ClutterDensity',Kc,...
                      'ConfirmationThreshold',confThreshold,...
                      'DeletionThreshold',delThreshold,...
                      'HasDetectableTrackIDsInput',true,...
                      'InitializationThreshold',0,...
                      'MaxNumTracks',30,...
                      'HitMissThreshold',0.1);

    detectableTracksInput = zeros(tracker.MaxNumTracks,2);

    currentNumTracks = 0;
end

ptCloud = pointCloud(ptCloudLocations);

% Detector model
[detections,obstacleIndices,groundIndices,croppedIndices] = detectorModel(ptCloud,time);

% Call tracker
[confirmedTracks,~,allTracks] = tracker(detections,time,detectableTracksInput(1:currentNumTracks
% Update the detectability input
currentNumTracks = numel(allTracks);
detectableTracksInput(1:currentNumTracks,:) = helperCalcDetectability(allTracks,[1 3 6]);

% Get model probabilities
modelProbs = zeros(2,numel(confirmedTracks));
if isLocked(tracker)
    for k = 1:numel(confirmedTracks)
        c1 = getTrackFilterProperties(tracker,confirmedTracks(k).TrackID,'ModelProbabilities');
        probs = c1{1};
        modelProbs(1,k) = probs(1);
        modelProbs(2,k) = probs(2);
    end
end

end
```

helperCalcDetectability

The function calculate the probability of detection for each track. This function is used to generate the "DetectableTracksIDs" input for the `trackerJPDA`.

```
function detectableTracksInput = helperCalcDetectability(tracks,posIndices)
% This is a helper function to calculate the detection probability of
% tracks for the lidar tracking example. It may be removed in a future
% release.

% Copyright 2019 The MathWorks, Inc.

% The bounding box detector has low probability of segmenting point clouds
% into bounding boxes are distances greater than 40 meters. This function
% models this effect using a state-dependent probability of detection for
% each tracker. After a maximum range, the Pd is set to a high value to
% enable deletion of track at a faster rate.
if isempty(tracks)
    detectableTracksInput = zeros(0,2);
    return;
end
rMax = 75;
rAmbig = 40;
stateSize = numel(tracks(1).State);
posSelector = zeros(3,stateSize);
posSelector(1,posIndices(1)) = 1;
posSelector(2,posIndices(2)) = 1;
posSelector(3,posIndices(3)) = 1;
pos = getTrackPositions(tracks,posSelector);
if coder.target('MATLAB')
    trackIDs = [tracks.TrackID];
else
    trackIDs = zeros(1,numel(tracks),'uint32');
    for i = 1:numel(tracks)
        trackIDs(i) = tracks(i).TrackID;
    end
end
[~,~,r] = cart2sph(pos(:,1),pos(:,2),pos(:,3));
probDetection = 0.9*ones(numel(tracks),1);
probDetection(r > rAmbig) = 0.4;
probDetection(r > rMax) = 0.99;
detectableTracksInput = [double(trackIDs(:)) probDetection(:)];
end
```

loadLidarAndImageData

Stitches Lidar and Camera data for processing using initial and final time specified.

```
function [lidarData,imageData] = loadLidarAndImageData(datasetFolder,initTime,finalTime)
initFrame = max(1,floor(initTime*10));
lastFrame = min(350,ceil(finalTime*10));
load (fullfile(datasetFolder,'imageData_35seconds.mat'),'allImageData');
imageData = allImageData(initFrame:lastFrame);

numFrames = lastFrame - initFrame + 1;
lidarData = cell(numFrames,1);

% Each file contains 70 frames.
initFileIndex = floor(initFrame/70) + 1;
lastFileIndex = ceil(lastFrame/70);
```

```
frameIndices = [1:70:numFrames numFrames + 1];

counter = 1;
for i = initFileIndex:lastFileIndex
    startFrame = frameIndices(counter);
    endFrame = frameIndices(counter + 1) - 1;
    load(fullfile(datasetFolder,['lidarData_',num2str(i)]),'currentLidarData');
    lidarData(startFrame:endFrame) = currentLidarData(1:(endFrame + 1 - startFrame));
    counter = counter + 1;
end
end
```

References

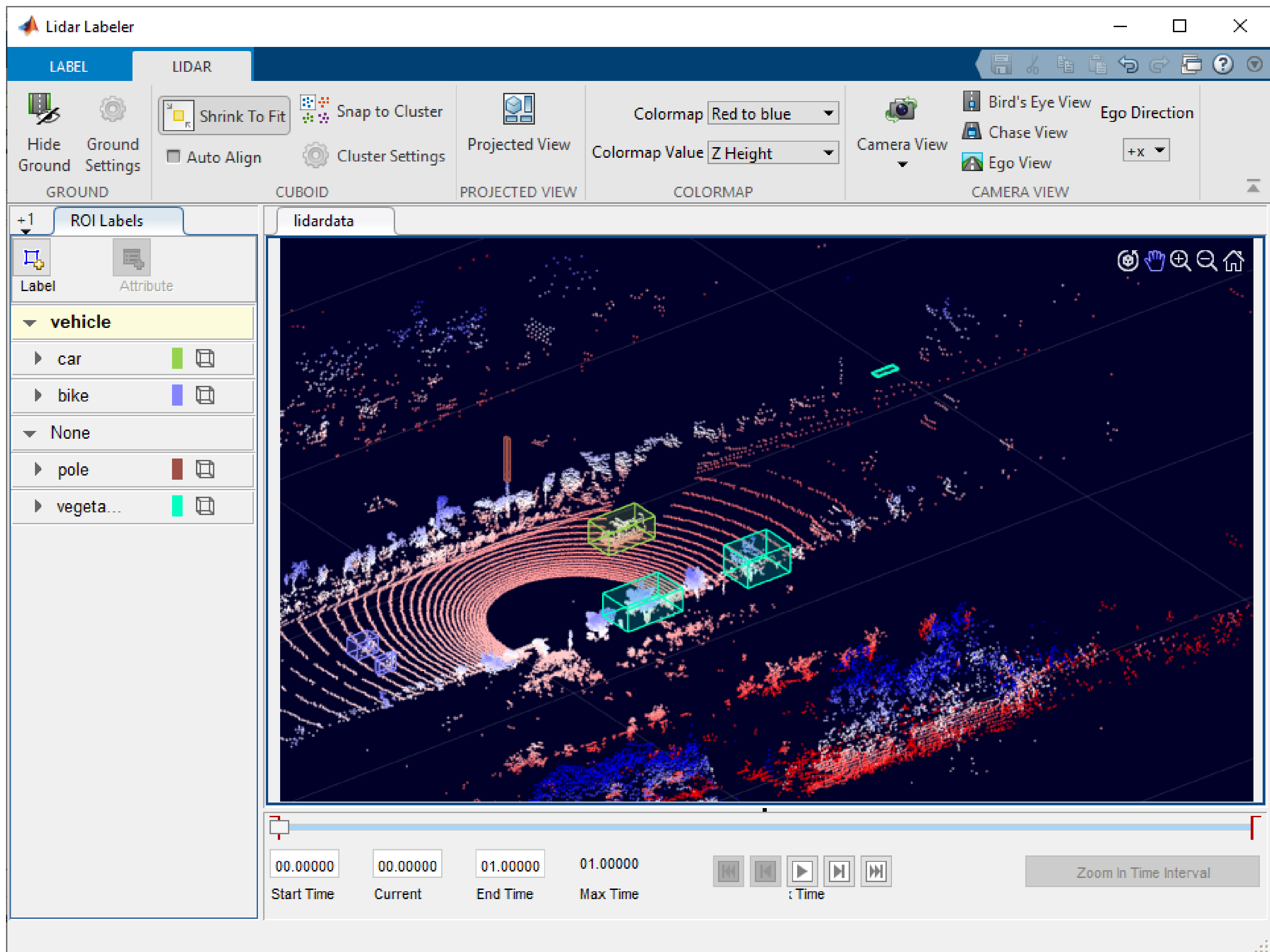
[1] Arya Senna Abdul Rachman, Arya. "3D-LIDAR Multi Object Tracking for Autonomous Driving: Multi-target Detection and Tracking under Urban Road Uncertainties." (2017).

Lidar Labeling

- “Get Started with the Lidar Labeler” on page 2-2
- “Keyboard Shortcuts and Mouse Actions for Lidar Labeler” on page 2-10

Get Started with the Lidar Labeler

The **Lidar Labeler** app enables you to interactively label ground truth data in a point cloud or a point cloud sequence and generate corresponding ground truth data.



This example demonstrates the capabilities of the **Lidar Labeler** app in a lidar ground truth data labeling workflow.

Load Lidar Data to Label

Use the **Lidar Labeler** app to interactively label point cloud files and sequences of point cloud files.

Open Lidar Labeler App

To open the **Lidar Labeler** app, at the MATLAB® command prompt, enter this command.

```
lidarLabeler
```

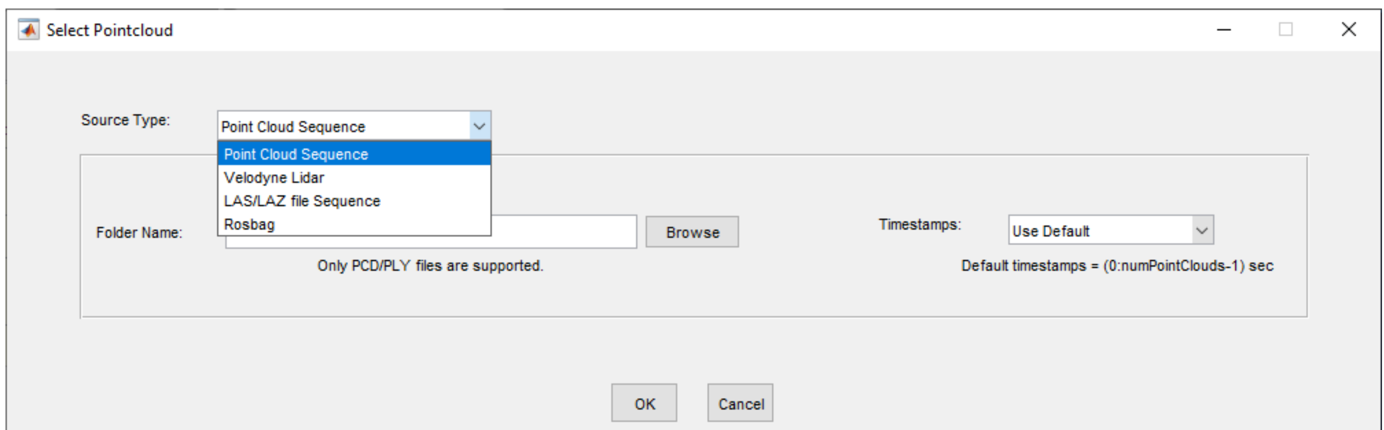
The app opens to an empty session.

Alternatively, you can open the app from the **Apps** tab, under **Image Processing and Computer Vision**.

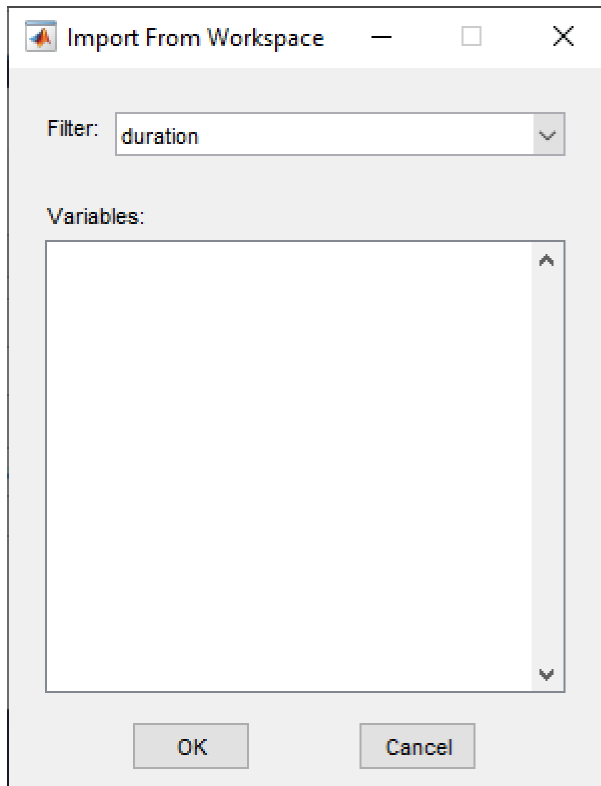
Load Signals from Data Sources

The **Lidar Labeler** app enables you to load signals from multiple types of data sources. In the app, a data source is a file or folder containing one or more signals to label. Use this process to load the data for a point cloud sequence.

1. On the app toolstrip, click **Open** > **Add point cloud**. The **Select Pointcloud** window opens with the **Source Type** parameter already set to `Point Cloud Sequence`.



2. In the **Folder Name** parameter, browse to the folder that contains the sequence of point cloud data(PCD) files that you want to load and click **Select Folder**.
3. If you have a variable of timestamps in the MATLAB workspace, set the **Timestamps** parameter to `From Workspace` and, in the `Import From Workspace` window, select the variable and click **OK**. Otherwise, set it to `Use Default`.



4. In the **Select Pointcloud** window, click **OK**. The point cloud sequence loads into the app.

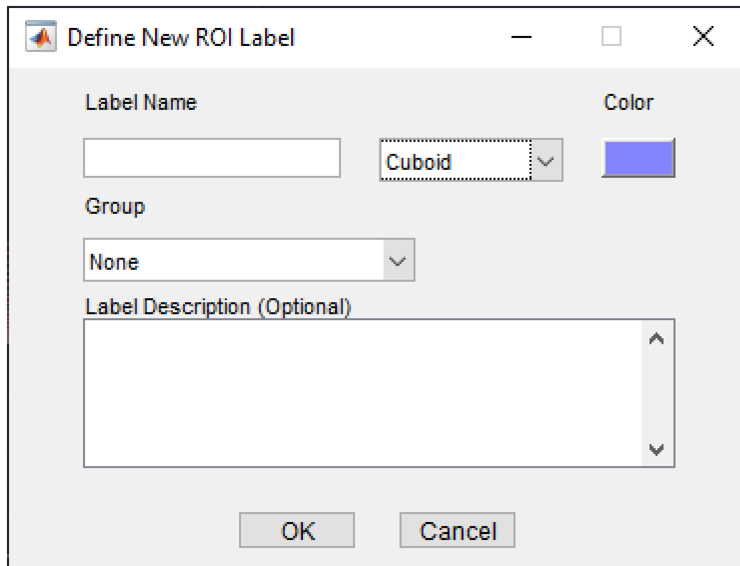
Create Labels and Attributes

After loading the point cloud data into the **Lidar Labeler** app, create label definitions and attributes. Label definitions contain the information about the labels that you wish to annotate the points with. You can create label definitions interactively within the app or programmatically by using a `labelDefinitionCreatorLidar` object.

Create an ROI Label Definition

An *ROI label* is a label that corresponds to a region of interest (ROI).

1. On the **ROI Labels** tab in the left pane, click **Label**.
2. Create a `Cuboid` label and provide a name for the label.



3 From the `Group` list, select `New Group` and provide a name for the group. Adding labels to groups is optional.
4 The specified group name appears on the **ROI Labels** tab with the specified label name under it.

For more details about these labels, see "ROI Labels and Attributes".

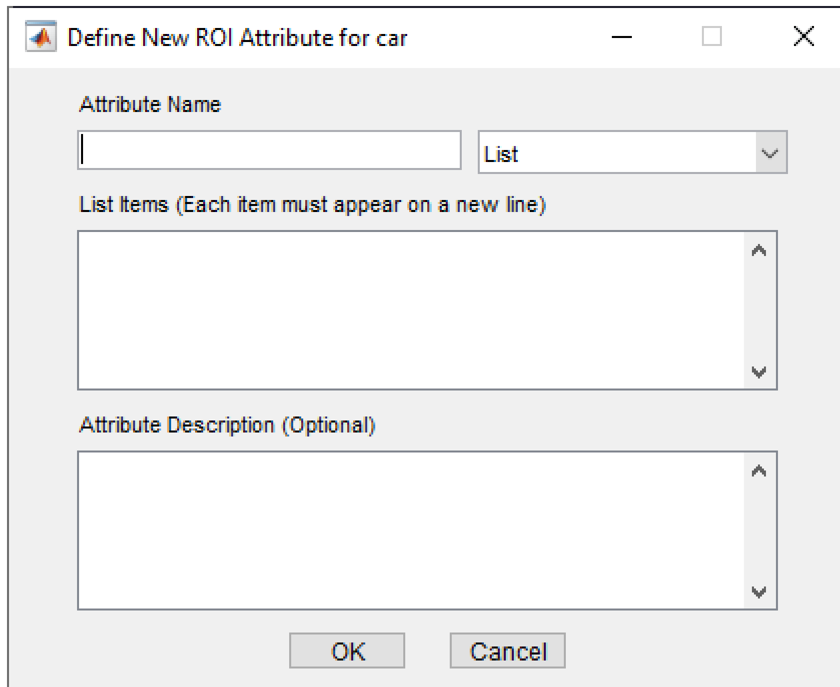
Create an ROI Attribute

An ROI attribute specifies additional information about an ROI label. You can define ROI attributes of these types.

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

Use this process to create an attribute.

1 On the **ROI Labels** tab in the left pane, select a label and click **Attribute**.
2 Provide a name in the **Attribute Name** box. Select the attribute type and click **OK**.



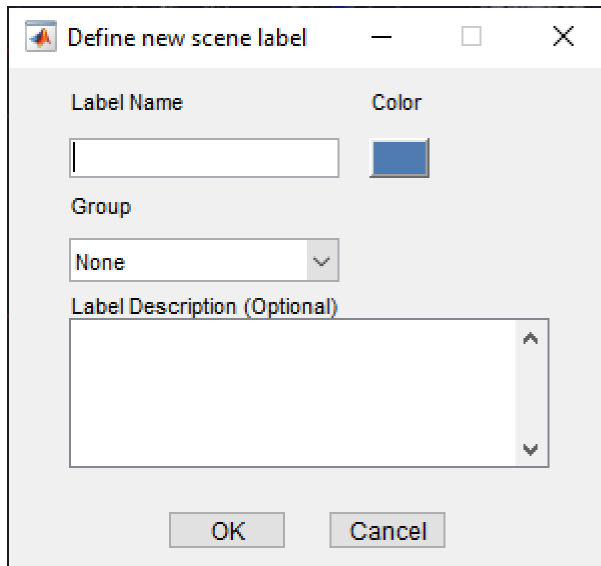
For more details about these attributes, see "ROI Labels and Attributes".

Create a Scene Label Definition

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

Use this process to create a scene label definition.

1. Select the **Scene Labels** tab in the left pane of the app and click **Define new scene label**.
2. In the Define new scene label window provide a name for the label.
3. Choose a **Color** for the label.



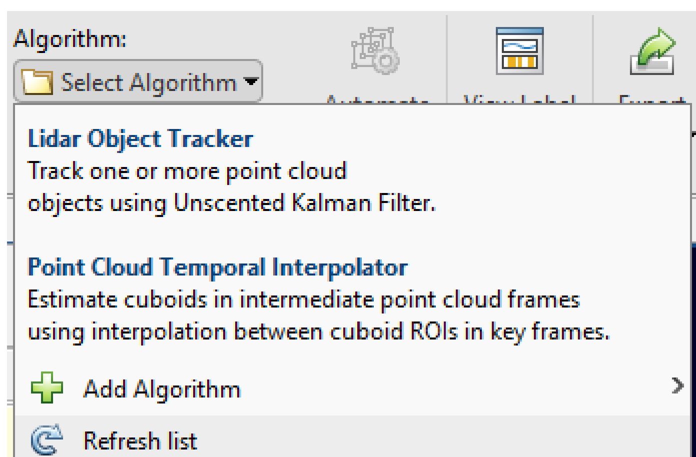
4 From the `Group` list, select `New Group` and provide a name for the group. Adding labels to groups is optional.
5 The **Scene Labels** pane shows the scene label definition.

Label Point Cloud Using Automation

You can use an automation algorithm to automatically label your data by using one of the included algorithms or by creating and importing a custom automation algorithm. For more details on creating a custom automation algorithm, see “Create Automation Algorithm for Labeling”. The app includes the **Lidar Object Tracker** and **Point Cloud Temporal Interpolator** labeling automation algorithms.

Use this process to label point cloud data using an automation algorithm.

1 Load the data into the app and create a ROI label definition.
2 On the **LABEL** tab of the app toolstrip, in the **Automate Labeling** section, click **Select Algorithm**.



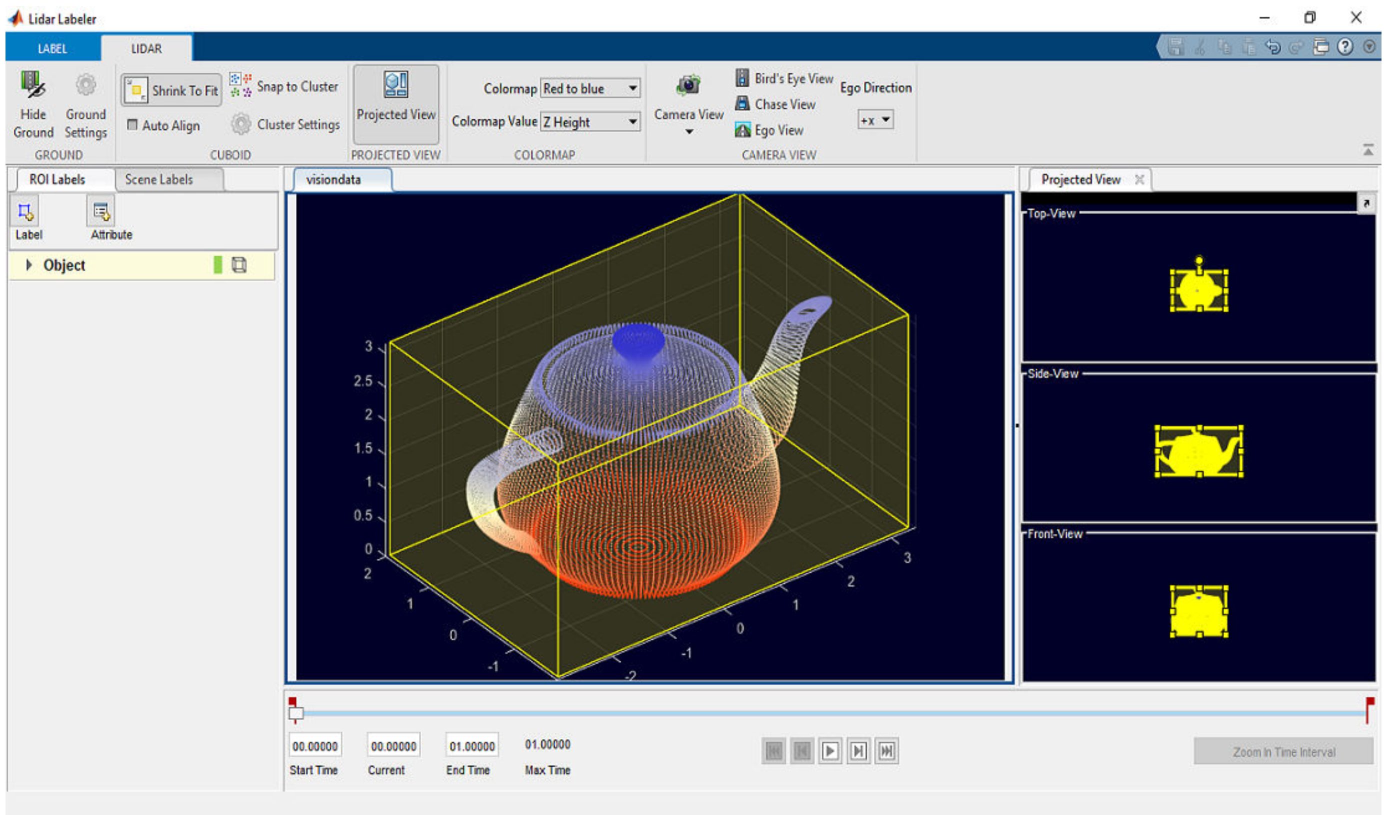
3 Select an algorithm for automation.

4 Click **Automate** and then follow the automation instructions in the right pane of the app.

View and Adjust the Labels

Once you have created labels for your point cloud data, the app provides options for viewing, adjusting, and comparing your point cloud and label data.

- On the **LIDAR** tab of the app toolstrip, click **Projected View** to view the selected label in front-view, top-view, and side-view simultaneously. Use these views to manually adjust the accuracy of your labels.



- Enable the **Auto Align** option to fit the cuboid to the label data and align the label in the direction of the object. This image shows the difference in a label with and without the **Auto Align** option enabled.

Label without **Auto Align** option	Label with **Auto Align** option

- Use the **Camera View** option to save and reuse custom views of the point cloud data. You can rotate, pan, and zoom the view, then save the view by clicking **Camera View** and selecting **Save Camera View**. Specify a name for the view and click **OK**. You can return to the saved view at any time by clicking **Camera View** and selecting the saved view from the drop-down list.

- Connect an external tool to the application to display time-synchronized images for use as reference while labeling. See the `lidar.syncImageViewer.SyncImageViewer` class.

Export the Labels

On the **LABEL** tab of the app toolstrip, select **Export Labels > To Workspace**. In the Export to workspace window, leave the default export variable name, `gTruth`, and click **OK**. The app exports a `groundTruthLidar` object, `gTruth`, to the MATLAB workspace. This object contains the ground truth lidar label data captured from the app session.

The properties of the `groundTruthLidar` object, `gTruth`, contain information about the signal data source, label definitions, and labels from the associated app session. Display information about the object and each of its properties using these commands.

- `gTruth` — Display the properties of the `groundTruthLidar` object.
- `gTruth.DataSource` — Display the information about source of the point cloud data.
- `gTruth.LabelDefinitions` — Display the table of information about label definitions.
- `gTruth.LabelData` — Display the ROI and scene label data.

See Also

Apps
Lidar Labeler

Objects
`groundTruthLidar` | `labelDefinitionCreatorLidar`

More About

- “Choose an App to Label Ground Truth Data”
- “Keyboard Shortcuts and Mouse Actions for Lidar Labeler” on page 2-10

Keyboard Shortcuts and Mouse Actions for Lidar Labeler

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

Label Definitions

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

Frame Navigation and Time Interval Settings

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

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

Labeling Window

Perform labeling actions, such as adding, moving, and deleting regions of interest (ROIs).

Task	Action
Undo labeling action	**Ctrl+Z**
Redo labeling action	**Ctrl+Y**
Select all cuboid ROIs	**Ctrl+A**
Select specific cuboid ROIs	Hold **Ctrl** and click the ROIs you want to select
Cut selected cuboid ROIs	**Ctrl+X**

Task	Action
Copy selected cuboid ROIs to clipboard	**Ctrl+C**
Paste copied cuboid ROIs	**Ctrl+V**
Switch between selected cuboid ROI labels.	**Tab** or **Shift+Tab**
Delete selected Cuboid ROIs	**Delete**

Cuboid Resizing and Moving

Draw cuboids to label lidar point clouds. For examples on how to use these shortcuts to label lidar point clouds efficiently, see “Label Lidar Point Clouds for Object Detection” (Automated Driving Toolbox).

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

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

Zooming, Panning, and Rotating

Task	Action
Zoom in on or out of a point cloud frame	In the top-left corner of the display, click the Zoom In or Zoom Out button. Then, move the scroll wheel up (zoom in) or down (zoom out). Alternatively, move the cursor up or right (zoom in) or down or left (zoom out). Zooming in and out is supported in all modes (pan, zoom, and rotate).
Pan across a point cloud frame	Hold **Shift** and press the up, down, left, or right arrows
Rotate a point cloud frame	Hold **R** and click and drag the point cloud frame **Note** Only yaw rotation is allowed.

App Sessions

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

See Also

Lidar Labeler

More About

- “Get Started with the Lidar Labeler” on page 2-2

Concept Pages

- “Lidar Processing Overview” on page 3-2
- “Coordinate Systems in Lidar Toolbox” on page 3-4
- “What Is Lidar Camera Calibration?” on page 3-7

Lidar Processing Overview

Introduction

Lidar is an acronym for light detection and ranging. It is an active sensing system that can be used for perception, navigation, and mapping of advanced driving assistance systems (ADAS), robots, and unmanned aerial vehicles (UAVs).

Lidar is an active remote sensing system. In an active system, the sensor generates energy by itself. Lidar sensors emit laser pulses that reflect off of objects, allowing them to perceive the structure of their surroundings. The sensors record the reflected light energy, to determine the distances to objects. The distance computation is based on the time of flight (TOF) principle. Lidar sensors are comparable to radar sensors, which emit radio waves.

Most modern autonomous or semi-autonomous vehicles are equipped with sensor suites that contain multiple sensors like a camera, IMU, and radar. Lidar sensors can resolve the drawbacks of some of these other sensors. Radar sensors can provide constant distance and velocity measurements, but the results lack resolution, and they have trouble with reflected energy and precision at longer ranges. Camera sensors can be significantly affected by environmental and lighting conditions. Lidar sensors address these issues by providing depth perception capabilities over long ranges, even in challenging weather and lighting conditions.

There are a wide variety of lidar sensors available in the industry, from companies such as Velodyne, Ouster, Quanergy, and Ibeo. These sensors generate lidar data in various formats. Lidar Toolbox™ currently supports reading data in the PLY, PCAP, PCD, LAS, LAZ, and Ibeo sensor formats. For more information, see “Lidar and Point Cloud I/O”. For more information about streaming data from Velodyne LiDAR® sensors, see “Lidar Toolbox Supported Hardware”.

Point Cloud

A point cloud is the representation of output data from a lidar sensor, similar to how an image is the representation of output data from a camera. It is a large collection of points that describe a 3-D map of the environment around the sensor. You can use a `pointCloud` object to store point cloud data. Lidar Toolbox provides basic processing for point clouds such as downsampling, median filtering, aligning, transforming, and extracting features from point clouds. For more information, see “Lidar Point Cloud Processing”.

There are two types of point clouds: organized and unorganized. These describe point cloud data stored in an arbitrary fashion or in a structured manner. An organized point cloud resembles a 2-D matrix, where the data is divided into rows and columns. The data is divided according to the spatial relationship between the points. As a result, the memory layout of an organized point cloud relates to the spatial layout represented by the *xyz*-coordinates of its points. In contrast, unorganized point clouds are stored as a single stream of 3-D coordinates, each representing a single point.

You can also differentiate these point clouds based on the shape of their data. Organized point clouds are specified as *M*-by-*N*-by-3 arrays. The three channels represent the *x*, *y*, and *z* coordinates of the points. Unorganized point clouds are specified as *M*-by-3 matrices, where *M* is the total number of points in the point cloud.

These are some of the major lidar processing applications:

- **Labeling point cloud data** — Labeling objects in point clouds helps with organizing and analyzing the data. Labeled point clouds can be used to train object segmentation and detection models. To learn more about labeling, see “Get Started with the Lidar Labeler” on page 2-2.
- **Semantic segmentation** — Semantic segmentation is the process of labeling specific regions of a point cloud as belonging to an object. The goal of the process is to associate each point in a point cloud with its corresponding class or label, such as car, truck, or vegetation in a driving scenario. It does not differentiate between multiple instances of objects from the same class. Semantic segmentation models can be used in autonomous driving applications to parse the environment of the vehicle. To learn more about the semantic segmentation workflow, see “Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network” on page 1-57.
- **Object detection and tracking** — Object detection and tracking usually follows point cloud segmentation. Objects in a point cloud can be detected and represented using cuboid bounding boxes. Tracking is the process of identifying the detected objects in one frame of a point cloud sequence throughout the sequence of point clouds. For detailed information on the complete workflow of segmentation, detection, and tracking, see “Detect, Classify, and Track Vehicles Using Lidar” on page 1-23.
- **Lidar camera calibration** — Due to the positional differences of the sensors in a sensor suite, the recorded data from each sensor is in a different coordinate system. Rotational and translational transformations are required to calibrate and fuse data from these sensors to each other. For more information, see “What Is Lidar Camera Calibration?” on page 3-7.

See Also

More About

- “The PLY Format”
- “Point Cloud Registration and Mapping Overview”
- “Getting Started with Point Clouds Using Deep Learning”

Coordinate Systems in Lidar Toolbox

Lidar Toolbox uses these coordinate systems:

- Word — A fixed, universal coordinate system in which the physical sensors exist.
- Sensor — Specific to each particular sensor, such as a lidar sensor or a camera.
- Spatial — Specific to an image captured by a camera. Locations in spatial coordinates are expressed in pixels.
- Pattern — A checkerboard pattern coordinate system, typically used to calibrate camera sensors.

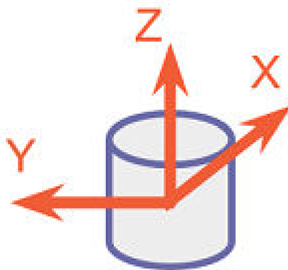
World Coordinate System

The world coordinate system is a fixed universal system that works as an absolute reference for all sensors. Lidar Toolbox uses the right-handed Cartesian world coordinate system defined in ISO 8855, where the positive *z*-axis points up from the ground. Units are in meters.

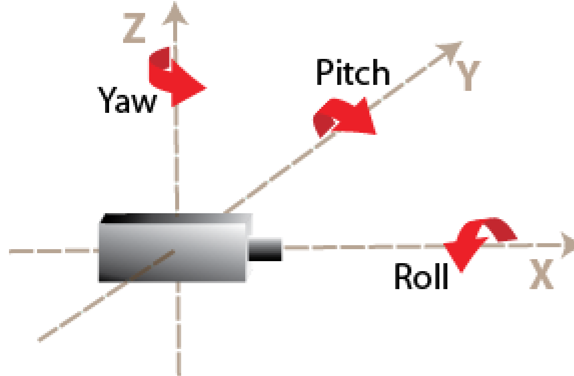
Sensor Coordinate System

A sensor coordinate system in Lidar Toolbox is anchored to a specific sensor, such as a lidar sensor or a camera. The location of each sensor contains the origin of its coordinate system. For example, the optical center of a camera typically acts as the origin of the camera coordinate system. Points in the sensor coordinate system follow these axes conventions:

- The *x*-axis points forward from the sensor.
- The *y*-axis points to the left, as viewed when facing forward.
- The *z*-axis points up from the ground to maintain the right-handed coordinate system.

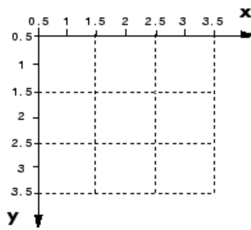


The yaw, pitch, and roll angles of sensors follow an ISO convention. These angles are clockwise-positive when looking in the positive direction of the *z*-, *y*-, and *x*-axes, respectively.



Spatial Coordinate System

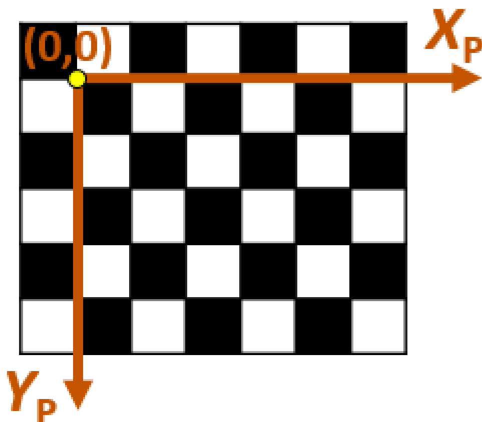
Spatial coordinates enable you to specify a location in an image with greater granularity than pixel coordinates. In the pixel coordinate system, each pixel is treated as a discrete unit, uniquely identified by an integer row and column pair, such as (3,4). In the spatial coordinate system, locations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



For more information on the spatial coordinate system, see “Spatial Coordinates”.

Pattern Coordinate System

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



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

See Also

More About

- “Coordinate Systems”
- “Image Coordinate Systems”

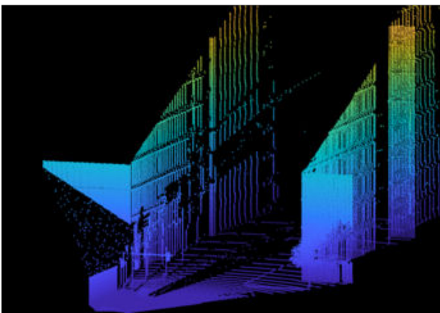
What Is Lidar Camera Calibration?

For applications such as automated driving, robotics, navigation systems, and 3-D scene reconstruction, data of the same scene is often captured using both lidar and camera sensors. To accurately interpret the objects in a scene, it is necessary to fuse the lidar and the camera outputs together. Lidar camera calibration estimates a rigid transformation matrix that establishes the correspondences between the points in the 3-D lidar plane and the pixels in the image plane. There are two parts to lidar camera calibration:

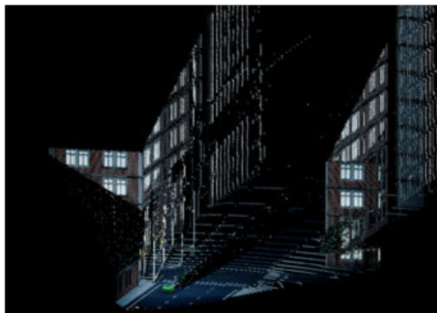
- Calibration for intrinsic parameters
- Calibration for extrinsic parameters between the lidar and camera

The intrinsic parameters of the lidar sensors are calibrated in advance by the manufacturers.

Lidar point cloud data

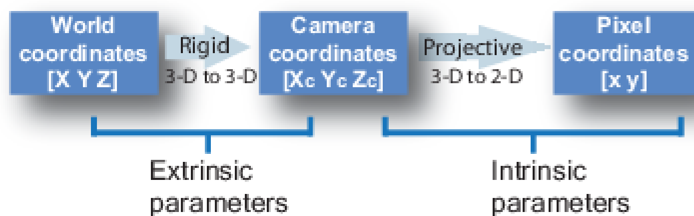


Fused lidar and camera data

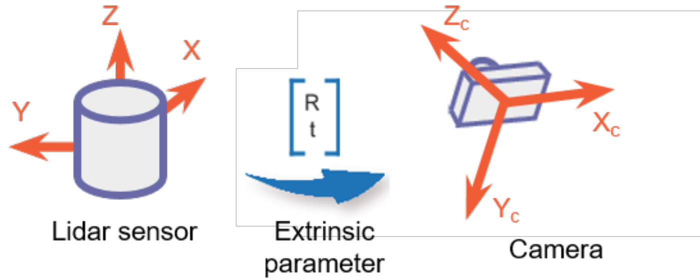


Extrinsic Calibration of Lidar and Camera

Extrinsic calibration of lidar and camera sensors generally uses calibration objects, such as planar boards with chessboard patterns, in the captured scene. The corner points of the calibration object are detected in the data captured by each sensor and used to establish the point correspondences between them. You can compute the image plane coordinates corresponding to the 3-D lidar points by using the extrinsic calibration and the intrinsic camera parameters.



The extrinsic calibration is a rigid transformation that maps points from the 3-D lidar coordinate system to the 3-D camera coordinate system. The extrinsic parameters consist of a rotation, R, and a translation, t.



You can estimate the rigid transformation matrix by using the `estimateLidarCameraTransform` function.

Then, compute the 2-D image plane coordinates from the 3-D lidar points and the extrinsic parameter.

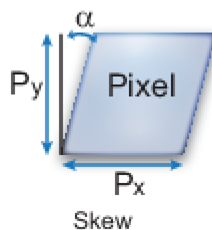
$$[x\ y\ 1] = [X\ Y\ Z\ 1]\begin{bmatrix}R \\ t\end{bmatrix}K$$

Image points; World points; Extrinsics Rotation and translation; Intrinsic matrix

K is the camera intrinsic matrix defined by the intrinsic parameters: focal length, optical center (also known as the *principal point*), and skew coefficient.

$$K = \begin{bmatrix} f_x & 0 & 0 \\ s & f_y & 0 \\ c_x & c_y & 1 \end{bmatrix}$$

$[c_x\ c_y]$ — Optical center (the principal point), in pixels.
(f_x, f_y) — Focal length in pixels.
$f_x = F/p_x$
$f_y = F/p_y$
F — Focal length in world units, typically expressed in millimeters.
(p_x, p_y) — Size of the pixel in world units.
s — Skew coefficient, which is non-zero if the image axes are not perpendicular.
$s = f_x \tan\alpha$
The pixel skew is defined as:



You can estimate the camera intrinsic parameters by using the `cameraIntrinsics` function. Using the estimated extrinsic calibration and camera intrinsic parameters, you can project lidar points onto the image or fuse the camera and the lidar sensor outputs. For more details, see the `projectLidarPointsOnImage` and `fuseCameraToLidar` functions.

References

[1] Zhou, Lipu, Zimo Li, and Michael Kaess. “Automatic Extrinsic Calibration of a Camera and a 3D LiDAR Using Line and Plane Correspondences.” In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 5562–69. Madrid: IEEE, 2018. https://doi.org/10.1109/IROS.2018.8593660.

See Also

`bboxCameraToLidar` | `estimateLidarCameraTransform` | `fuseCameraToLidar` | `projectLidarPointsOnImage`

Related Examples

- “Lidar and Camera Calibration” on page 1-2
- “Detect Vehicles in Lidar Using Image Labels” on page 1-47