

Lidar Toolbox™

User's Guide



MATLAB®

R2021a



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– 2021 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)
March 2021		Revised for Version 1.1 (R2021a)

1

Lidar Toolbox Featured Examples

Build Map and Localize Using Segment Matching	1-2
Lidar and Camera Calibration	1-21
Lidar Point Cloud Semantic Segmentation Using PointSeg Deep Learning Network	1-30
Detect, Classify, and Track Vehicles Using Lidar	1-41
Feature-Based Map Building from Lidar Data	1-56
Detect Vehicles in Lidar Using Image Labels	1-65
Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network	1-75
Code Generation for Lidar Point Cloud Segmentation Network	1-84
Lidar 3-D Object Detection Using PointPillars Deep Learning	1-91
Aerial Lidar SLAM Using FPFH Descriptors	1-107
Collision Warning Using 2-D Lidar	1-121
Track Vehicles Using Lidar: From Point Cloud to Track List	1-131
Build Map from 2-D Lidar Scans Using SLAM	1-150
Terrain Classification for Aerial Lidar Data	1-157
Data Augmentations for Lidar Object Detection Using Deep Learning	1-163
Unorganized to Organized Conversion of Point Clouds Using Spherical Projection	1-173
Lane Detection in 3-D Lidar Point Cloud	1-180
Automate Ground Truth Labeling For Vehicle Detection Using PointPillars	1-197
Track-Level Fusion of Radar and Lidar Data	1-209

Code Generation For Lidar Object Detection Using PointPillars Deep Learning	1-229
Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning	1-235

Lidar Labeling

2

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	2-11
Keyboard Shortcuts and Mouse Actions for Lidar Labeler	2-13
Label Definitions	2-13
Frame Navigation and Time Interval Settings	2-13
Labeling Window	2-13
Cuboid Resizing and Moving	2-14
Zooming, Panning, and Rotating	2-15
App Sessions	2-15
Use Custom Point Cloud Source Reader for Labeling	2-16
Create Custom Reader Function	2-16
Import Data Source into Lidar Labeler App	2-16

Concept Pages

3

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
Calibration Guidelines and Procedure	3-10
Checkerboard Guidelines	3-10
Guidelines for Capturing Data	3-11

4

Estimate Transformation Between Two Point Clouds Using Features	4-2
Match and Visualize Corresponding Features in Point Clouds	4-5
Read Lidar and Camera Data from Rosbag File	4-8

Lidar Toolbox Featured Examples

- “Build Map and Localize Using Segment Matching” on page 1-2
- “Lidar and Camera Calibration” on page 1-21
- “Lidar Point Cloud Semantic Segmentation Using PointSeg Deep Learning Network” on page 1-30
- “Detect, Classify, and Track Vehicles Using Lidar” on page 1-41
- “Feature-Based Map Building from Lidar Data” on page 1-56
- “Detect Vehicles in Lidar Using Image Labels” on page 1-65
- “Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network” on page 1-75
- “Code Generation for Lidar Point Cloud Segmentation Network” on page 1-84
- “Lidar 3-D Object Detection Using PointPillars Deep Learning” on page 1-91
- “Aerial Lidar SLAM Using FPFH Descriptors” on page 1-107
- “Collision Warning Using 2-D Lidar” on page 1-121
- “Track Vehicles Using Lidar: From Point Cloud to Track List” on page 1-131
- “Build Map from 2-D Lidar Scans Using SLAM” on page 1-150
- “Terrain Classification for Aerial Lidar Data” on page 1-157
- “Data Augmentations for Lidar Object Detection Using Deep Learning” on page 1-163
- “Unorganized to Organized Conversion of Point Clouds Using Spherical Projection” on page 1-173
- “Lane Detection in 3-D Lidar Point Cloud” on page 1-180
- “Automate Ground Truth Labeling For Vehicle Detection Using PointPillars” on page 1-197
- “Track-Level Fusion of Radar and Lidar Data” on page 1-209
- “Code Generation For Lidar Object Detection Using PointPillars Deep Learning” on page 1-229
- “Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning” on page 1-235

Build Map and Localize Using Segment Matching

This example shows how to build a map with lidar data and localize the position of a vehicle on the map using SegMatch [1] on page 1-0 , a place recognition algorithm based on segment matching.

Autonomous driving systems use localization to determine the position of the vehicle within a mapped environment. Autonomous navigation requires accurate localization, which relies on an accurate map. Building an accurate map of large scale environments is difficult because the map accumulates drift over time, and detecting loop closures to correct for accumulated drift is challenging due to dynamic obstacles. The SegMatch algorithm is robust to dynamic obstacles and reliable in large scale environments. The algorithm is a segment-based approach that takes advantage of descriptive shapes and recognizes places by matching segments.

Overview

Like the “Build a Map from Lidar Data Using SLAM” example, this example uses 3-D lidar data to build a map and corrects for the accumulated drift using graph SLAM. However, this example does not require global pose estimates from other sensors, such as an inertial measurement unit (IMU). After building the map, this example uses it to localize the vehicle in a known environment.

In this example, you learn how to:

- Use SegMatch to find the relative transformation between two point clouds that correspond to the same place
- Build a map using SegMatch for loop closure detection
- Localize on a prebuilt map using SegMatch

Download Data

The data used in this example is part of the Velodyne SLAM Dataset. It includes approximately 6 minutes of data recorded with a Velodyne HDL64E-S2 scanner. Download the data to a temporary directory. This can take a few minutes.

```
baseDownloadURL = 'https://www.mrt.kit.edu/z/publ/download/velodyneslam/data/scenario1.zip';  
dataFolder = fullfile(tempdir, 'kit_velodyneslam_data_scenario1', filesep);  
options = weboptions('Timeout', Inf);
```

```
zipFileName = dataFolder + "scenario1.zip";
```

```
folderExists = exist(dataFolder, 'dir');  
if ~folderExists  
    % Create a folder in a temporary directory to save the downloaded zip file.  
    mkdir(dataFolder)
```

```
    disp('Downloading scenario1.zip (153 MB) ...')  
    websave(zipFileName, baseDownloadURL, options);
```

```
    % Unzip downloaded file.  
    unzip(zipFileName, dataFolder)
```

```
end
```

```
Downloading scenario1.zip (153 MB) ...
```

Load and Select Data

The downloaded dataset stores point cloud data in PNG files. Create a file datastore using the helperReadVelodyneSLAMData on page 1-0 function to load point cloud data from the PNG files

and convert distance values to 3-D coordinates. The helper function is a custom read function, which is designed for the Velodyne SLAM Dataset. Select a subset of the data and split the data to use for map building and for localization.

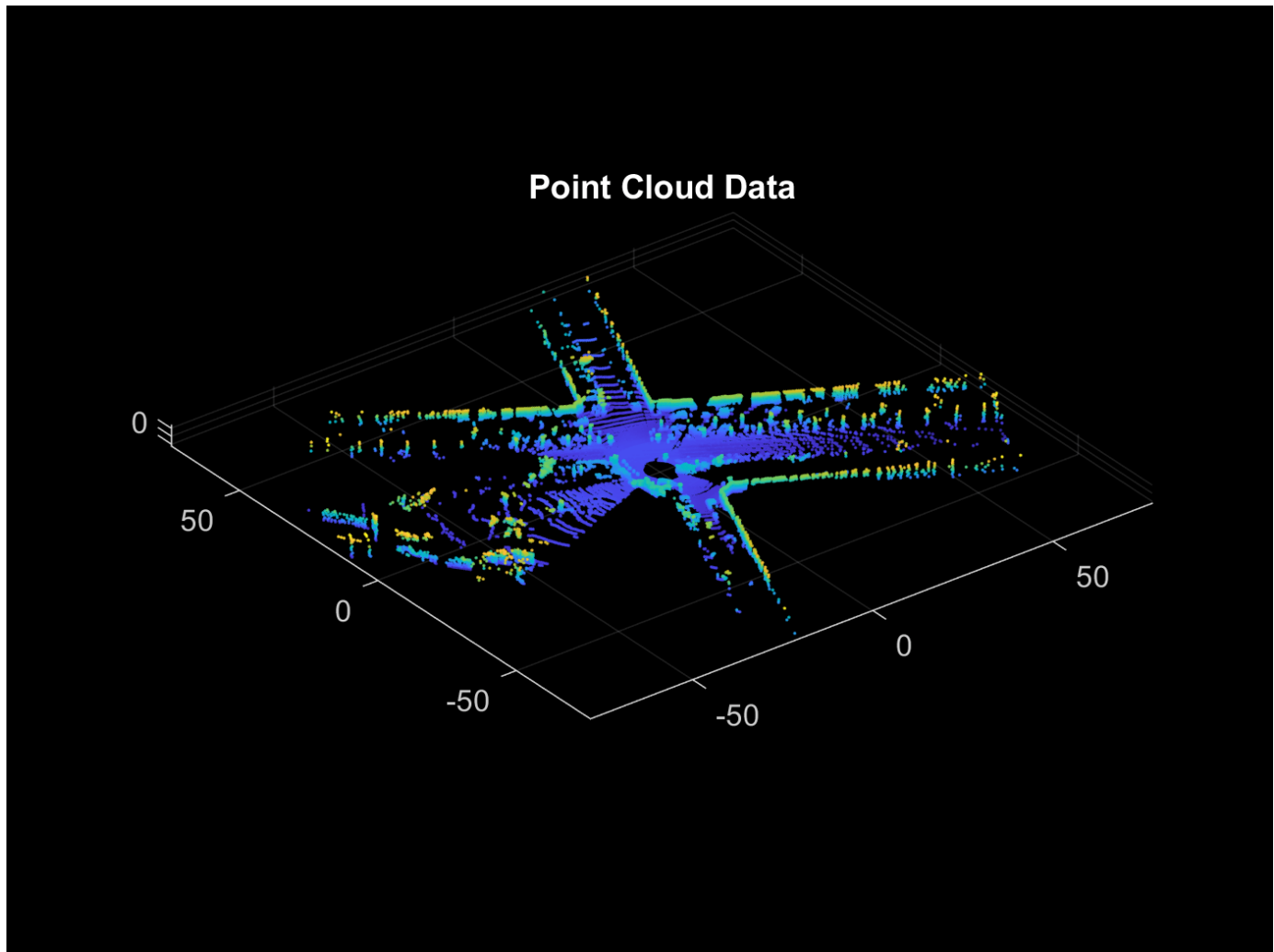
```
% Get the full file path to the PNG files in the scenario1 folder.
pointCloudFilePattern = fullfile(dataFolder, 'scenario1', 'scan*.png');

% Create a file datastore to read files in the right order.
fileDS = fileDatastore(pointCloudFilePattern, 'ReadFcn', ...
    @helperReadVelodyneSLAMData);

% Read the point clouds.
ptCloudArr = readall(fileDS);

% Select a subset of point cloud scans, and split the data to use for
% map building and for localization.
ptCloudMap = vertcat(ptCloudArr{1:5:1550});
ptCloudLoc = vertcat(ptCloudArr{2:5:1550});

% Visualize the first point cloud.
figure
pcshow(ptCloudMap(1))
title('Point Cloud Data')
```



SegMatch Overview

The SegMatch algorithm consists of four different components: point cloud segmentation, feature extraction, segment matching, and geometric verification. For best results, preprocess the point cloud before performing these four steps.

Preprocess Point Cloud

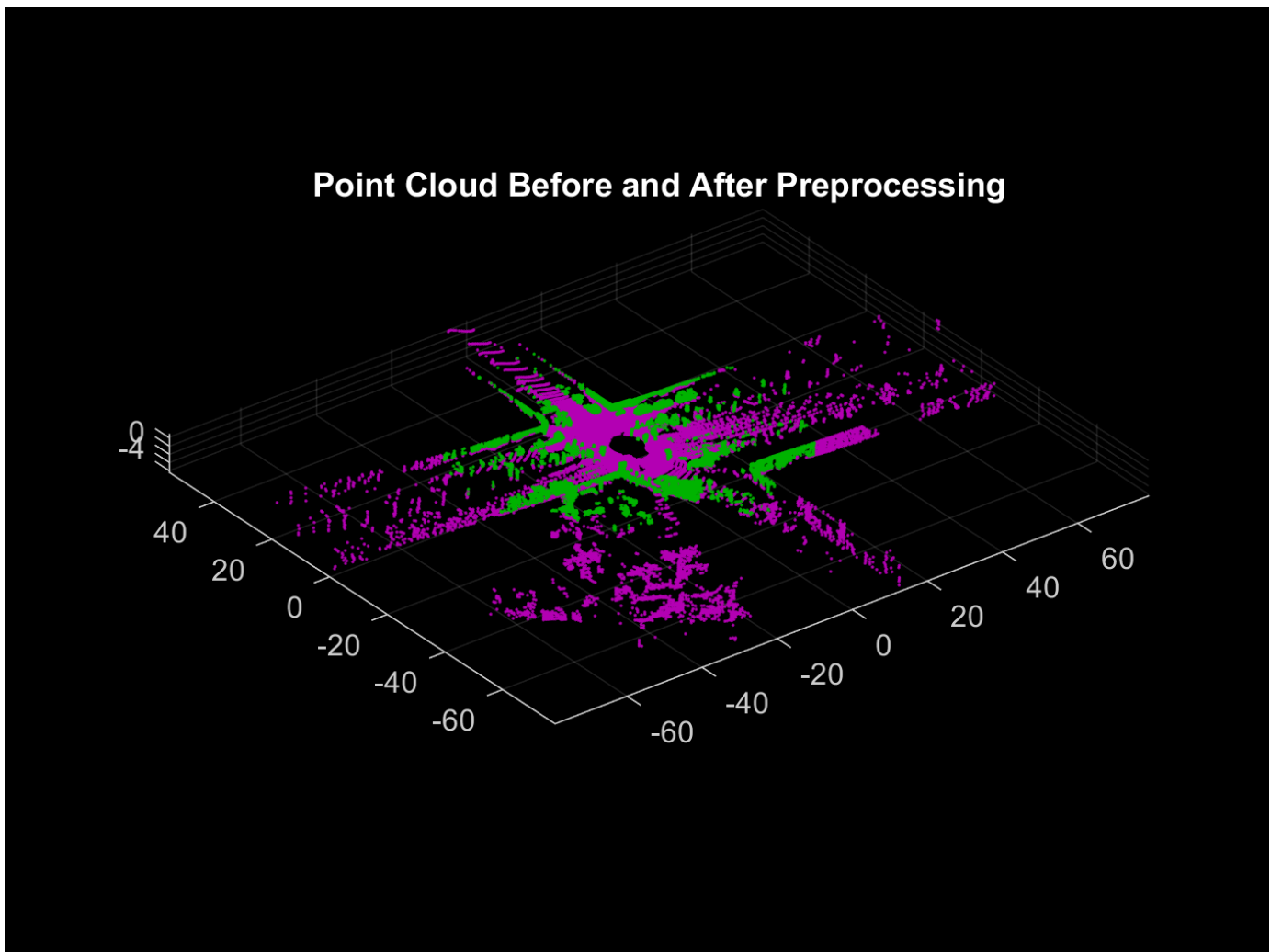
To select the most relevant point cloud data, perform the following preprocessing steps:

- 1 Select a cylindrical neighborhood centered around the vehicle to extract a local point cloud of interest. First, specify a cylindrical neighborhood based on the distance of the points from the origin in the x and y directions. Then, select the area of interest using `select`.
- 2 Remove the ground in preparation to segment the point cloud into distinct objects. Use `segmentGroundSMRF` to segment the ground.

```
% Select a point cloud from the map for preprocessing.  
ptCloud = ptCloudMap(25);
```

```
% Set the cylinder radius and ego radius.
```

```
cylinderRadius = 40;  
egoRadius = 1;  
  
% Compute the distance between each point and the origin.  
dists = hypot(ptCloud.Location(:,:,1), ptCloud.Location(:,:,2));  
  
% Select the points inside the cylinder radius and outside the ego radius.  
cylinderIdx = dists <= cylinderRadius & dists >= egoRadius;  
cylinderPtCloud = select(ptCloud, cylinderIdx, 'OutputSize', 'full');  
  
% Remove the ground.  
[~, ptCloudNoGround] = segmentGroundSMRF(cylinderPtCloud, 'ElevationThreshold', 0.05);  
  
% Visualize the point cloud before and after preprocessing.  
figure  
pcshowpair(ptCloud, ptCloudNoGround)  
title('Point Cloud Before and After Preprocessing')
```



Segmentation and Feature Extraction

Next, segment the point cloud and extract features from each segment.

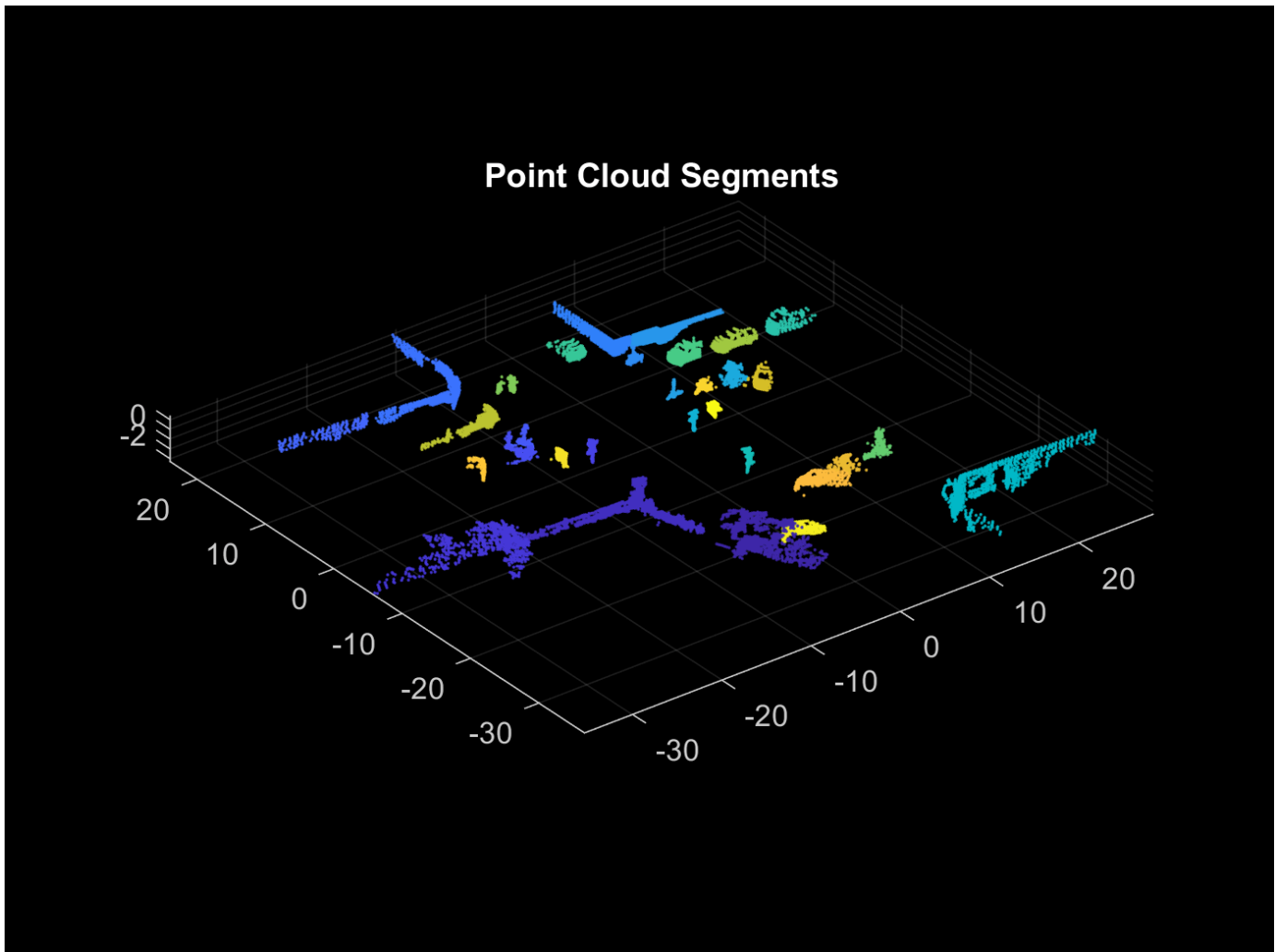
Segment the point cloud by using the `segmentLidarData` function and visualize the segments. For this example, each segment must have a minimum of 150 points. This produces segment clusters that represent distinct objects and have enough points to characterize the area in the map.

Different datasets require different parameters for segmentation. Requiring fewer points for segments can lead to false positive loop closures, and limiting the segments to larger clusters can eliminate segments that are important for place recognition. You must also tune the distance and angle thresholds to ensure that each segment corresponds to one object. A small distance threshold can result in many segments that correspond to the same object, and a large distance threshold and small angle threshold can result in segments that combine many objects.

```
minNumPoints = 150;
distThreshold = 1;
angleThreshold = 180;
[labels, numClusters] = segmentLidarData(ptCloudNoGround, distThreshold, ...
    angleThreshold, 'NumClusterPoints', minNumPoints);

% Remove points that contain a label value of 0 for visualization.
idxValidPoints = find(labels);
labelColorIndex = labels(idxValidPoints);
segmentedPtCloud = select(ptCloudNoGround, idxValidPoints);

figure
pcshow(segmentedPtCloud.Location, labelColorIndex)
title('Point Cloud Segments')
```



Extract features from each segment by using the `extractEigenFeatures` function. Eigenvalue-based features are geometric features. Each feature vector includes linearity, planarity, scattering, omnivariance, anisotropy, eigenentropy, and change in curvature.

```
[features, segments] = extractEigenFeatures(ptCloud, labels);
disp(features)
```

```
31x1 eigenFeature array with properties:
```

```
Feature
Centroid
```

```
disp(segments)
```

```
31x1 pointCloud array with properties:
```

```
Location
Count
XLimits
YLimits
ZLimits
```

```
Color
Normal
Intensity
```

Segment Matching and Geometric Verification

Find the matching segments and the transformation between the segments for two point cloud scans that correspond to a loop closure.

Preprocess and extract segment features from the two point clouds. The `helperPreProcessPointCloud` on page 1-0 function includes the preprocessing steps in the Preprocess Point Cloud on page 1-0 section, to simplify preprocessing the point cloud throughout the workflow.

```
ptCloud1 = ptCloudMap(27);
ptCloud2 = ptCloudMap(309);

ptCloud1 = helperPreProcessPointCloud(ptCloud1, egoRadius, cylinderRadius);
ptCloud2 = helperPreProcessPointCloud(ptCloud2, egoRadius, cylinderRadius);

labels1 = segmentLidarData(ptCloud1, distThreshold, ...
    angleThreshold, 'NumClusterPoints', minNumPoints);
labels2 = segmentLidarData(ptCloud2, distThreshold, ...
    angleThreshold, 'NumClusterPoints', minNumPoints);

[features1, segments1] = extractEigenFeatures(ptCloud1, labels1);
[features2, segments2] = extractEigenFeatures(ptCloud2, labels2);
```

Find the possible segment matches based on the normalized euclidean distance between the feature vectors by using the `pcmatchfeatures` function.

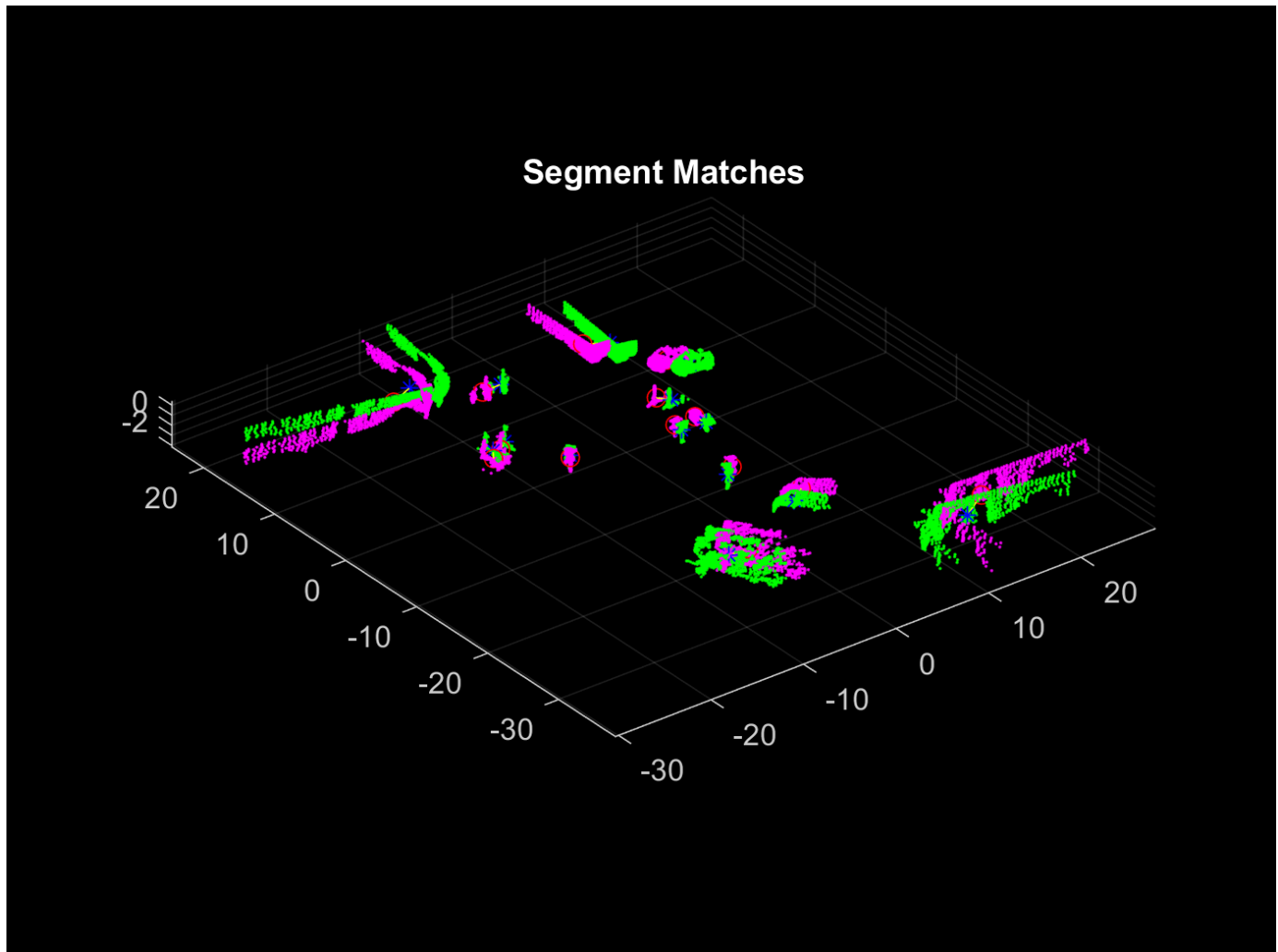
```
featureMatrix1 = vertcat(features1.Feature);
featureMatrix2 = vertcat(features2.Feature);
indexPairs = pcmatchfeatures(featureMatrix1, featureMatrix2);
```

Perform geometric verification by identifying inliers and finding the 3-D rigid transformation between segment matches using the `estimateGeometricTransform3D` function. Based on the number of inliers, the point clouds can be classified as a loop closure.

```
centroids1 = vertcat(features1(indexPairs(:,1)).Centroid);
centroids2 = vertcat(features2(indexPairs(:,2)).Centroid);
[tform, inlierPairs] = estimateGeometricTransform3D(centroids1, centroids2, 'rigid');
```

Visualize the segment matches by using the `pcshowMatchedFeatures` function.

```
inlierIdx1 = indexPairs(inlierPairs,1);
inlierIdx2 = indexPairs(inlierPairs,2);
figure
pcshowMatchedFeatures(segments1(inlierIdx1), segments2(inlierIdx2), ...
    features1(inlierIdx1), features2(inlierIdx2))
title('Segment Matches')
```

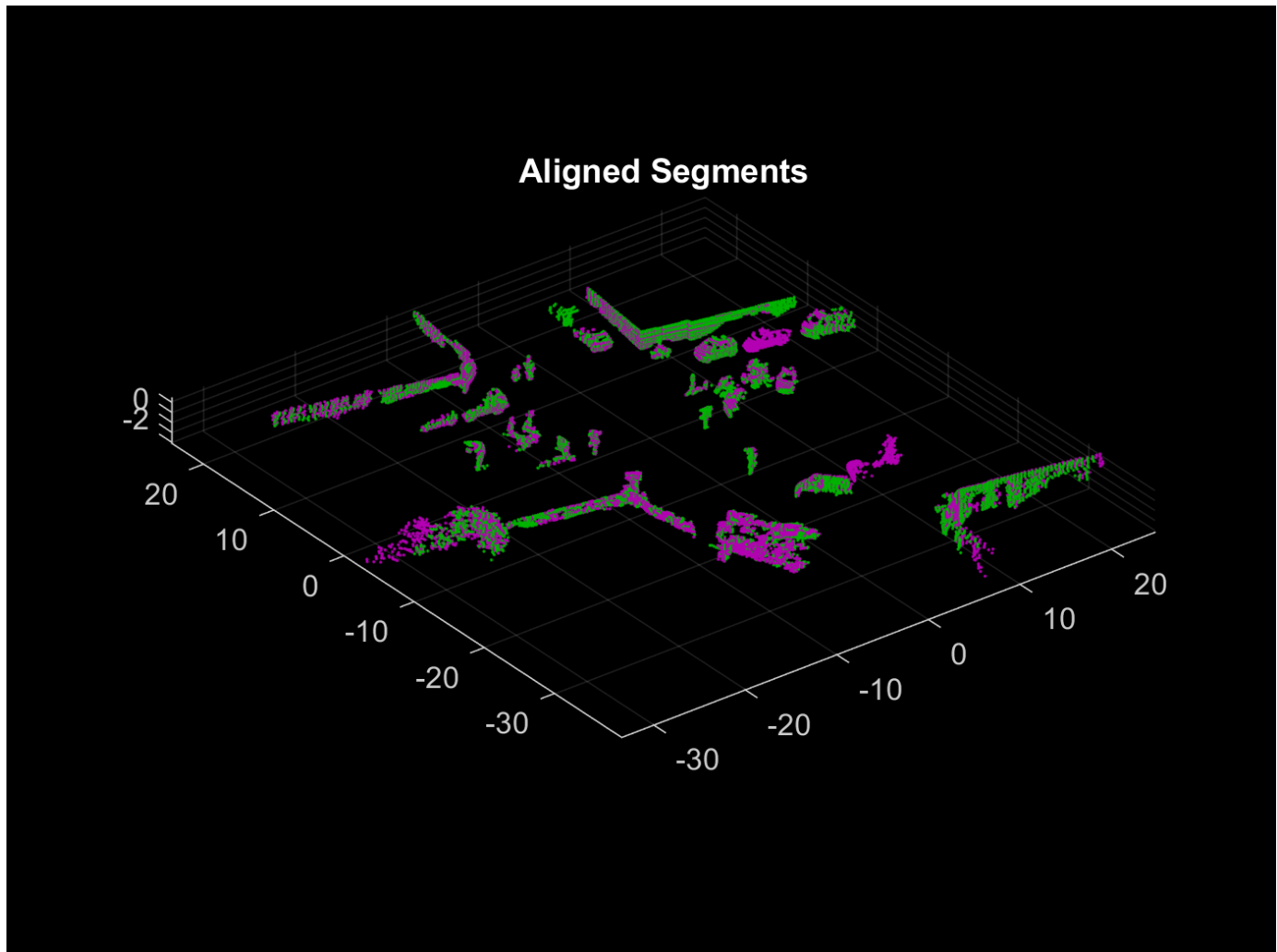


Align the segments with the transformation from the geometric verification step using `pccat` and `pctransform`.

```
ptCloudSegments1 = pccat(segments1);  
ptCloudSegments2 = pccat(segments2);  
tformedPtCloudSegments1 = pctransform(ptCloudSegments1, tform);
```

Visualize the aligned segments using `pcshowpair`.

```
figure  
pcshowpair(tformedPtCloudSegments1, ptCloudSegments2)  
title('Aligned Segments')
```



Build Map

The map building procedure consists of the following steps:

- 1 Preprocess the point cloud
- 2 Register the point cloud
- 3 Segment the point cloud and extract features
- 4 Detect loop closures

Preprocess Point Cloud

Preprocess the previous and current point cloud using `helperPreProcessPointCloud` on page 1-0 . Downsample the point clouds using `pcdownsample` to improve registration speed and accuracy. To tune the downsample percentage input, find the lowest value that maintains the desired registration accuracy when the vehicle turns.

```
currentViewId = 2;
```

```
prevPtCloud = helperPreProcessPointCloud(ptCloudMap(currentViewId - 1), ...
```



```

    egoRadius, cylinderRadius);
ptCloud = helperPreProcessPointCloud(ptCloudMap(currentViewId), ...
    egoRadius, cylinderRadius);

downsamplePercent = 0.5;

prevPtCloudFiltered = pcdsample(prevPtCloud, 'random', downsamplePercent);
ptCloudFiltered = pcdsample(ptCloud, 'random', downsamplePercent);

```

Register Point Cloud

Register the current point cloud with the previous point cloud to find the relative transformation.

```

gridStep = 3;
relPose = pcregisterndt(ptCloudFiltered, prevPtCloudFiltered, gridStep);

```

Use a `pcviewset` object to track absolute poses and connections between registered point clouds. Create an empty `pcviewset`.

```
vSet = pcviewset;
```

Initialize the pose of the first point cloud to an identity rigid transformation, and add it to the view set using `addView`.

```

initAbsPose = rigid3d;
vSet = addView(vSet, currentViewId - 1, initAbsPose);

```

Compute the absolute pose of the second point cloud using the relative pose estimated during registration, and add it to the view set.

```

absPose = rigid3d(relPose.T * initAbsPose.T);
vSet = addView(vSet, currentViewId, absPose);

```

Connect the two views using `addConnection`.

```
vSet = addConnection(vSet, currentViewId - 1, currentViewId, relPose);
```

Transform the current point cloud to align it to the global map.

```
ptCloud = pctransform(ptCloud, absPose);
```

Segment Point Cloud and Extract Features

Segment the previous and current point clouds using `segmentLidarData`.

```

labels1 = segmentLidarData(prevPtCloud, distThreshold, angleThreshold, ...
    'NumClusterPoints', minNumPoints);
labels2 = segmentLidarData(ptCloud, distThreshold, angleThreshold, ...
    'NumClusterPoints', minNumPoints);

```

Extract features from the previous and current point cloud segments using `extractEigenFeatures`.

```

[prevFeatures, prevSegments] = extractEigenFeatures(prevPtCloud, labels1);
[features, segments] = extractEigenFeatures(ptCloud, labels2);

```

Track the segments and features using a `pcmapsegmatch` object. Create an empty `pcmapsegmatch`.

```
sMap = pcmapsegmatch;
```

Add the views, features, and segments for the previous and current point clouds to the `pcmapsegmatch` using `addView`.

```
sMap = addView(sMap, currentViewId - 1, prevFeatures, prevSegments);  
sMap = addView(sMap, currentViewId, features, segments);
```

Detect Loop Closures

The estimated poses accumulate drift as more point clouds are added to the map. Detecting loop closures helps correct for the accumulated drift and produce a more accurate map.

Detect loop closures using `findPose`.

```
[absPoseMap, loopClosureViewId] = findPose(sMap, absPose);  
isLoopClosure = ~isempty(absPoseMap);
```

If `findPose` detects a loop closure, find the transformation between the current view and the loop closure view and add it to the `pcviewset` object.

Use the absolute pose of the current view without the accumulated drift, `absPoseMap`, and the absolute pose of the loop closure view, `absPoseLoop`, to compute the relative pose between the loop closure poses without the drift.

```
if isLoopClosure  
    absPoseLoop = poses(vSet, loopClosureViewId).AbsolutePose;  
    relPoseLoopToCurrent = rigid3d(absPoseMap.T * invert(absPoseLoop).T);
```

Add the loop closure relative pose as a connection using `addConnection`.

```
vSet = addConnection(vSet, loopClosureViewId, currentViewId, ...  
    relPoseLoopToCurrent);
```

Correct for the accumulated drift using pose graph optimization. Consider finding more than one loop closure connection before optimizing the poses, since optimizing the pose graph and updating the `pcmapsegmatch` object are both computationally intensive.

Save the poses before optimization. The poses are needed to update the segments and centroid locations in the `pcmapsegmatch` object.

```
prevPoses = vSet.Views.AbsolutePose;
```

Create a pose graph from the view set using `createPoseGraph`, and optimize the pose graph using `optimizePoseGraph` (Navigation Toolbox).

```
G = createPoseGraph(vSet);  
optimG = optimizePoseGraph(G, 'g2o-levenberg-marquardt');  
vSet = updateView(vSet, optimG.Nodes);
```

Find the transformations from the poses before and after correcting for drift and use them to update the map segments and centroid locations using `updateMap`.

```
optimizedPoses = vSet.Views.AbsolutePose;  
  
relPoseOpt = rigid3d.empty;  
for k = 1:numel(prevPoses)  
    relPoseOpt(k) = rigid3d(invert(prevPoses(k)).T * ...  
        optimizedPoses(k).T);  
end
```

```

    sMap = updateMap(sMap, relPose0pt);
end

```

To build the map and correct for accumulated drift, apply these steps to the rest of the point cloud scans.

```

% Set the random seed for example reproducibility.
rng(0)

% Update display every 5 scans.
figure
updateRate = 5;

% Initialize variables for registration.
prevPtCloud = ptCloudFiltered;
prevPose = rigid3d;

% Keep track of the loop closures to optimize the poses once enough loop
% closures are detected.
totalLoopClosures = 0;

for i = 3:numel(ptCloudMap)
    ptCloud = ptCloudMap(i);

    % Preprocess and register the point cloud.
    ptCloud = helperPreProcessPointCloud(ptCloud, egoRadius, cylinderRadius);
    ptCloudFiltered = pcdsample(ptCloud, 'random', downsamplePercent);
    relPose = pcregisterndt(ptCloudFiltered, prevPtCloud, gridStep, ...
        'InitialTransform', relPose);
    ptCloud = pctransform(ptCloud, absPose);

    % Store the current point cloud to register the next point cloud.
    prevPtCloud = ptCloudFiltered;

    % Compute the absolute pose of the current point cloud.
    absPose = rigid3d(relPose.T * absPose.T);

    % If the vehicle has moved at least 2 meters since last time, continue
    % with segmentation, feature extraction, and loop closure detection.
    if norm(absPose.Translation - prevPose.Translation) >= 2

        % Segment the point cloud and extract features.
        labels = segmentLidarData(ptCloud, distThreshold, angleThreshold, ...
            'NumClusterPoints', minNumPoints);
        [features, segments] = extractEigenFeatures(ptCloud, labels);

        % Keep track of the current view id.
        currentViewId = currentViewId + 1;

        % Add the view to the point cloud view set and map representation.
        vSet = addView(vSet, currentViewId, absPose);
        vSet = addConnection(vSet, currentViewId-1, currentViewId, ...
            rigid3d(absPose.T * invert(prevPose).T));
        sMap = addView(sMap, currentViewId, features, segments);

        % Update the view set display.
        if mod(currentViewId, updateRate) == 0

```

```
        plot(vSet)
        drawnow
    end

    % Check if there is a loop closure.
    [absPoseMap, loopClosureViewId] = findPose(sMap, absPose, 'MatchThreshold', 1, ...
        'MinNumInliers', 5, 'NumSelectedClusters', 4, 'NumExcludedViews', 150);
    isLoopClosure = ~isempty(absPoseMap);

    if isLoopClosure
        totalLoopClosures = totalLoopClosures + 1;

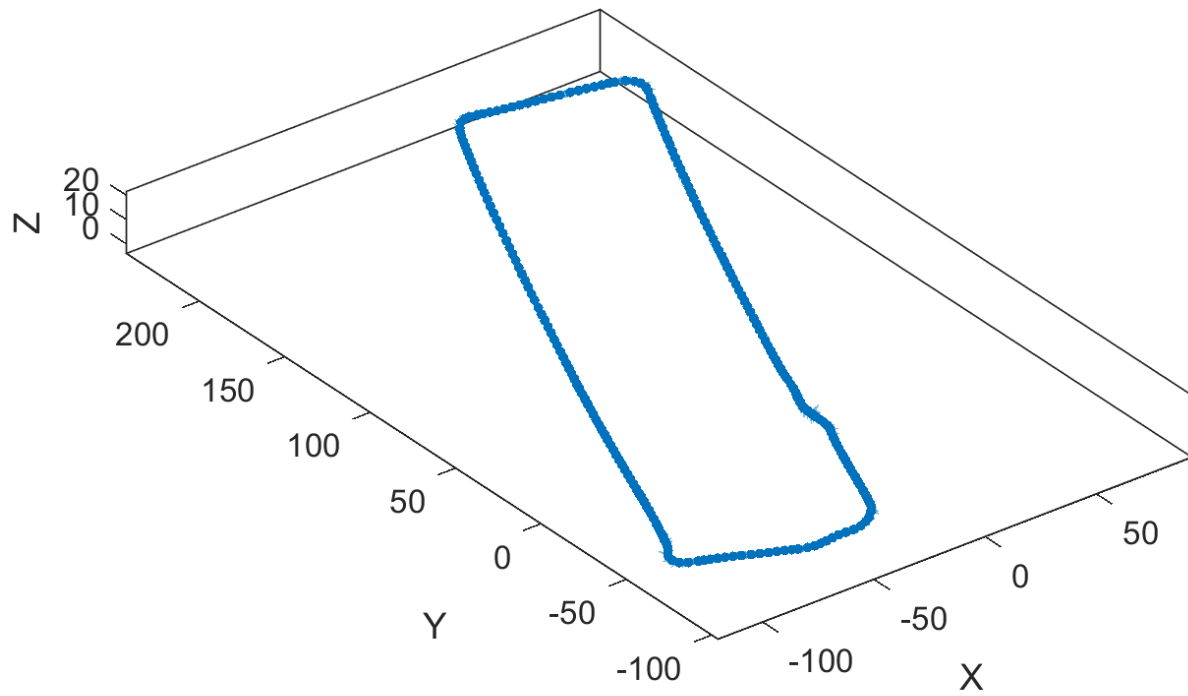
        % Find the relative pose between the loop closure poses.
        absPoseLoop = poses(vSet, loopClosureViewId).AbsolutePose;
        relPoseLoopToCurrent = rigid3d(absPoseMap.T * invert(absPoseLoop).T);
        vSet = addConnection(vSet, loopClosureViewId, currentViewId, ...
            relPoseLoopToCurrent);

        % Optimize the graph of poses and update the map every time 3
        % loop closures are detected.
        if mod(totalLoopClosures, 3) == 0
            prevPoses = vSet.Views.AbsolutePose;

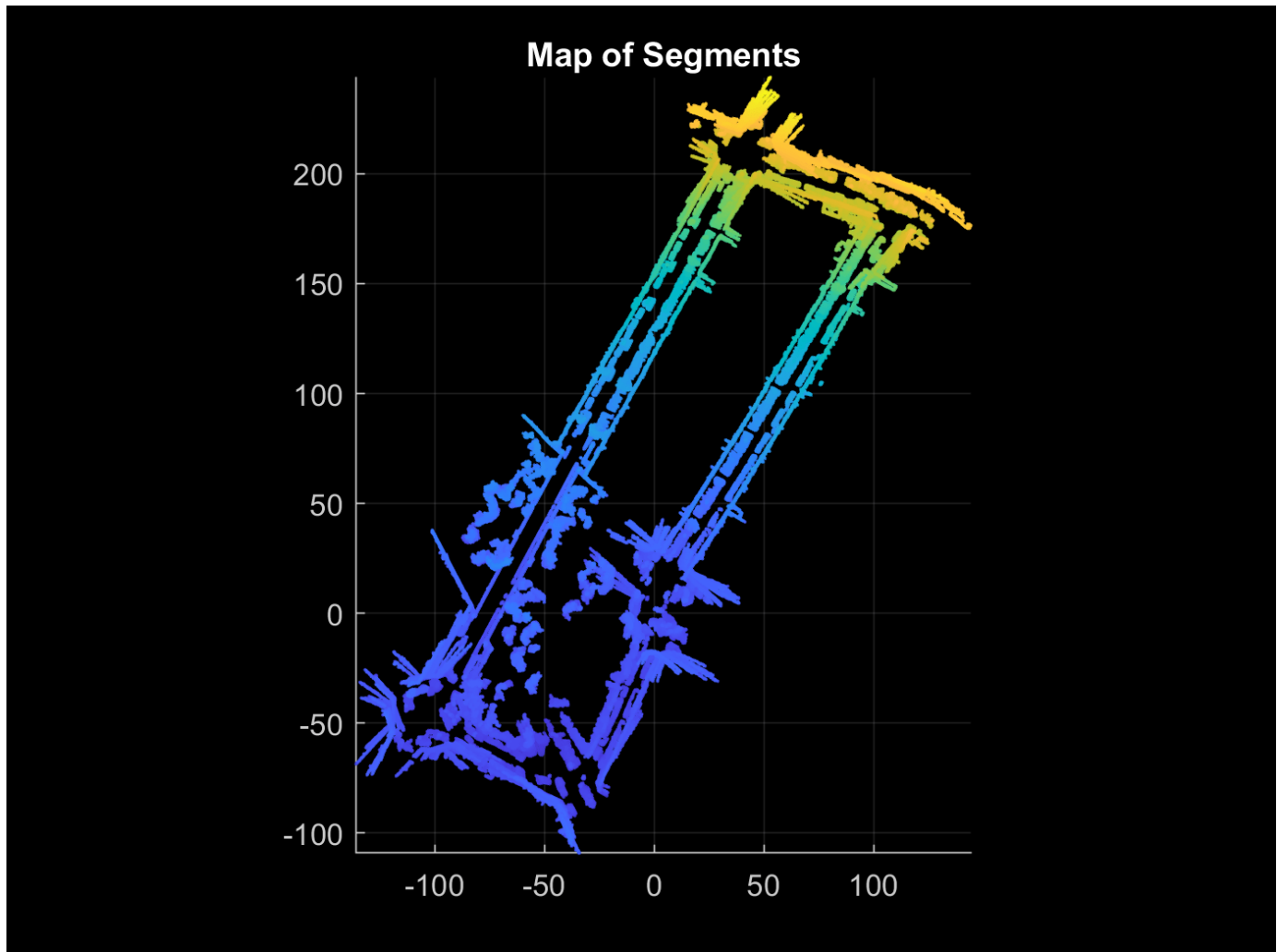
            % Correct for accumulated drift.
            G = createPoseGraph(vSet);
            optimG = optimizePoseGraph(G, 'g2o-levenberg-marquardt');
            vSet = updateView(vSet, optimG.Nodes);

            % Update the map.
            optimizedPoses = vSet.Views.AbsolutePose;
            relPoseOpt = rigid3d.empty;
            for k = 1:numel(prevPoses)
                relPoseOpt(k) = rigid3d(invert(prevPoses(k)).T * ...
                    optimizedPoses(k).T);
            end
            sMap = updateMap(sMap, relPoseOpt);

            % Update the absolute pose after pose graph optimization.
            absPose = optimizedPoses(end);
        end
    end
    prevPose = absPose;
end
end
```



```
% Visualize the map of segments from the top view.  
figure  
show(sMap)  
view(2)  
title('Map of Segments')
```



Localize Vehicle in Known Map

The preprocessing steps for localization using SegMatch are the same preprocessing steps used for map building. Since the algorithm relies on consistent segmentation, use the same segmentation parameters for best results.

```
ptCloud = ptCloudLoc(1);

% Preprocess the point cloud.
ptCloud = helperPreProcessPointCloud(ptCloud, egoRadius, cylinderRadius);

% Segment the point cloud and extract features.
labels = segmentLidarData(ptCloud, distThreshold, angleThreshold, ...
    'NumClusterPoints', minNumPoints);
features = extractEigenFeatures(ptCloud, labels);
```

Because there is no position estimate for the vehicle, you must use the extent of the map for initial vehicle localization. Select the extent of the map to localize for the first time using `selectSubmap`.

```
sMap = selectSubmap(sMap, [sMap.XLimits, sMap.YLimits, sMap.ZLimits]);
```

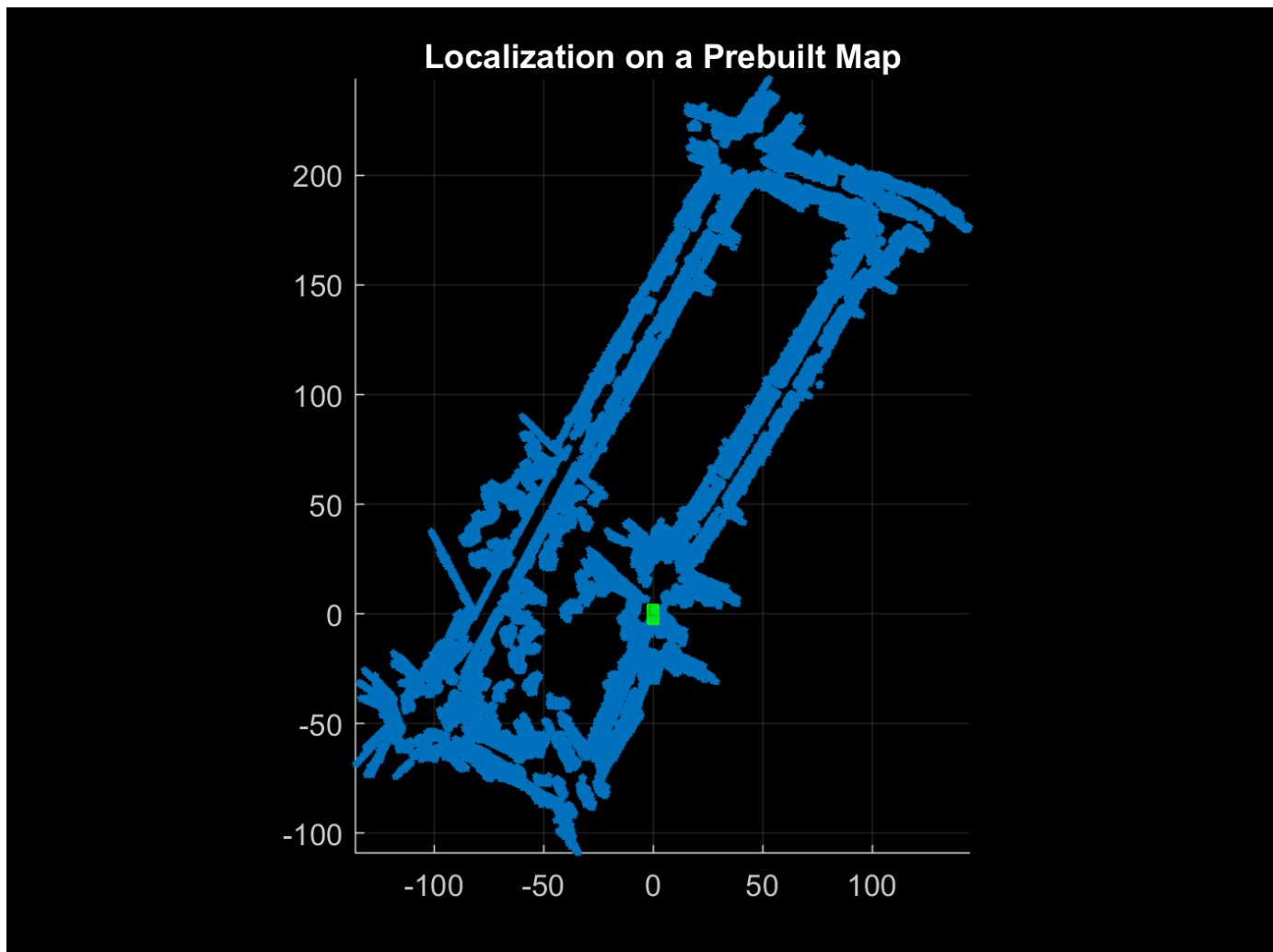
Use the `findPose` object function of `pcmapsegment` to localize the vehicle on the prebuilt map.

```
absPoseMap = findPose(sMap, features, 'MatchThreshold', 1, 'MinNumInliers', 5);
```

Visualize the map, and use `showShape` to visualize the vehicle on the map as a cuboid.

```
mapSegments = pccat(sMap.Segments);
hAxLoc = pcshow(mapSegments.Location, 'p');
title('Localization on a Prebuilt Map')
view(2)

poseTranslation = absPoseMap.Translation;
eul = rotm2eul(absPoseMap.Rotation);
theta = rad2deg(eul);
pos = [poseTranslation, 5, 9, 3.5, theta(2), theta(3), theta(1)];
showShape('cuboid', pos, 'Color', 'green', 'Parent', hAxLoc, 'Opacity', 0.8, 'LineWidth', 0.5)
```



To improve localization speed for the rest of the scans, select a submap using `selectSubmap`.

```
submapSize = [65, 65, 200];
sMap = selectSubmap(sMap, poseTranslation, submapSize);
```

Continue localizing the vehicle using the rest of the point cloud scans. Use `isInsideSubmap` and `selectSubmap` to keep the submap updated. If there are not enough segments to localize the vehicle using segment matching, use registration to estimate the pose.

```
% Visualize the map.
figure('Visible', 'on')
hAx = pcshow(mapSegments.Location, 'p');
title('Localization on a Prebuilt Map')

% Set parameter to update submap.
submapThreshold = 30;

% Initialize the poses and previous point cloud for registration.
prevPtCloud = ptCloud;
relPose = rigid3d;
prevAbsPose = rigid3d;

% Segment each point cloud and localize by finding segment matches.
for n = 2:numel(ptCloudLoc)
    ptCloud = ptCloudLoc(n);

    % Preprocess the point cloud.
    ptCloud = helperPreProcessPointCloud(ptCloud, egoRadius, cylinderRadius);

    % Segment the point cloud and extract features.
    labels = segmentLidarData(ptCloud, distThreshold, angleThreshold, ...
        'NumClusterPoints', minNumPoints);
    features = extractEigenFeatures(ptCloud, labels);

    % Localize the point cloud.
    absPoseMap = findPose(sMap, features, 'MatchThreshold', 1, 'MinNumInliers', 5);

    % Do registration when the position cannot be estimated with segment
    % matching.
    if isempty(absPoseMap)
        relPose = pcregisterndt(ptCloud, prevPtCloud, gridStep, ...
            'InitialTransform', relPose);
        absPoseMap = rigid3d(relPose.T * prevAbsPose.T);
    end

    % Display position estimate in the map.
    poseTranslation = absPoseMap.Translation;
    eul = rotm2eul(absPoseMap.Rotation');
    theta = rad2deg(eul);
    pos = [poseTranslation, 5, 9, 3.5, theta(2), theta(3), theta(1)];
    showShape('cuboid', pos, 'Color', 'green', 'Parent', hAx, 'Opacity', 0.8, 'LineWidth', 0.5)

    % Determine if selected submap needs to be updated.
    [isInside, distToEdge] = isInsideSubmap(sMap, poseTranslation);
    needSelectSubmap = ~isInside ... % Current pose is outside submap.
        || any(distToEdge(1:2) < submapThreshold); % Current pose is close to submap edge

    % Select a new submap.
    if needSelectSubmap
        sMap = selectSubmap(sMap, poseTranslation, submapSize);
    end

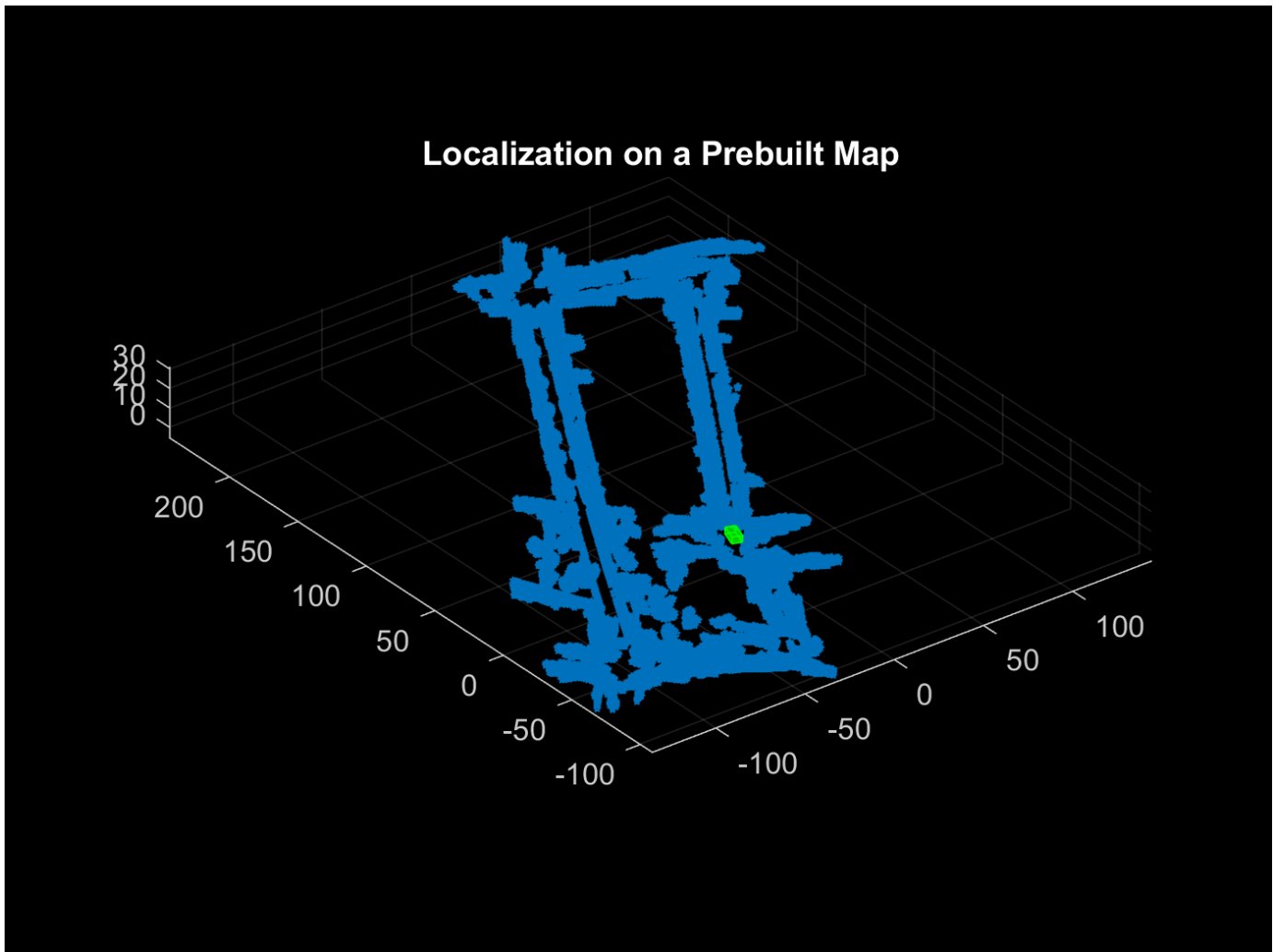
    prevAbsPose = absPoseMap;
end
```



```

    prevPtCloud = ptCloud;
end

```



References

[1] Dube, Renaud, Daniel Dugas, Elena Stumm, Juan Nieto, Roland Siegwart, and Cesar Cadena. "SegMatch: Segment Based Place Recognition in 3D Point Clouds." In 2017 IEEE International Conference on Robotics and Automation (ICRA), 5266-72. Singapore: IEEE, 2017. <https://doi.org/10.1109/ICRA.2017.7989618>.

Supporting Functions

helperReadVelodyneSLAMData reads point clouds from PNG image files from the Velodyne SLAM Dataset.

helperPreProcessPointCloud selects a cylindrical neighborhood and removes the ground from a point cloud.

```
function ptCloud = helperPreProcessPointCloud(ptCloud, egoRadius, cylinderRadius)
```

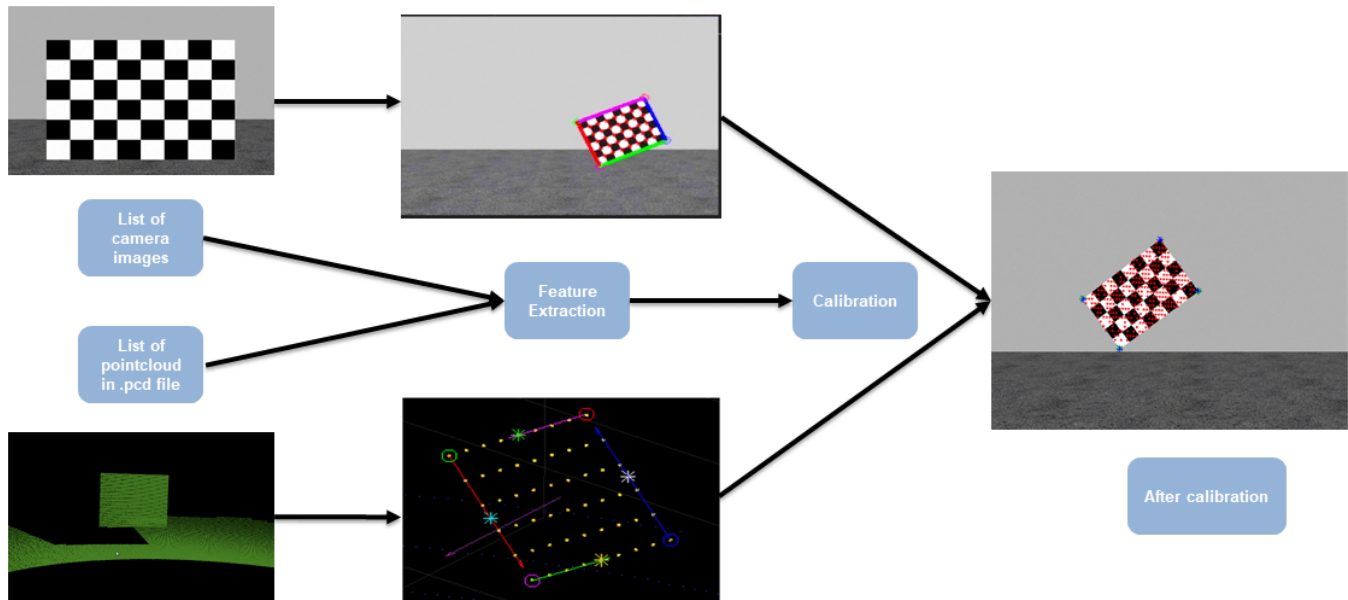
```
% Compute the distance between each point and the origin.
```

```
dists = hypot(ptCloud.Location(:,:,1), ptCloud.Location(:,:,2));  
  
% Select the points inside the cylinder radius and outside the ego radius.  
cylinderIdx = dists <= cylinderRadius & dists >= egoRadius;  
ptCloud = select(ptCloud, cylinderIdx, 'OutputSize', 'full');  
  
% Remove ground.  
[~, ptCloud] = segmentGroundSMRF(ptCloud, 'ElevationThreshold', 0.05);  
  
end
```

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.

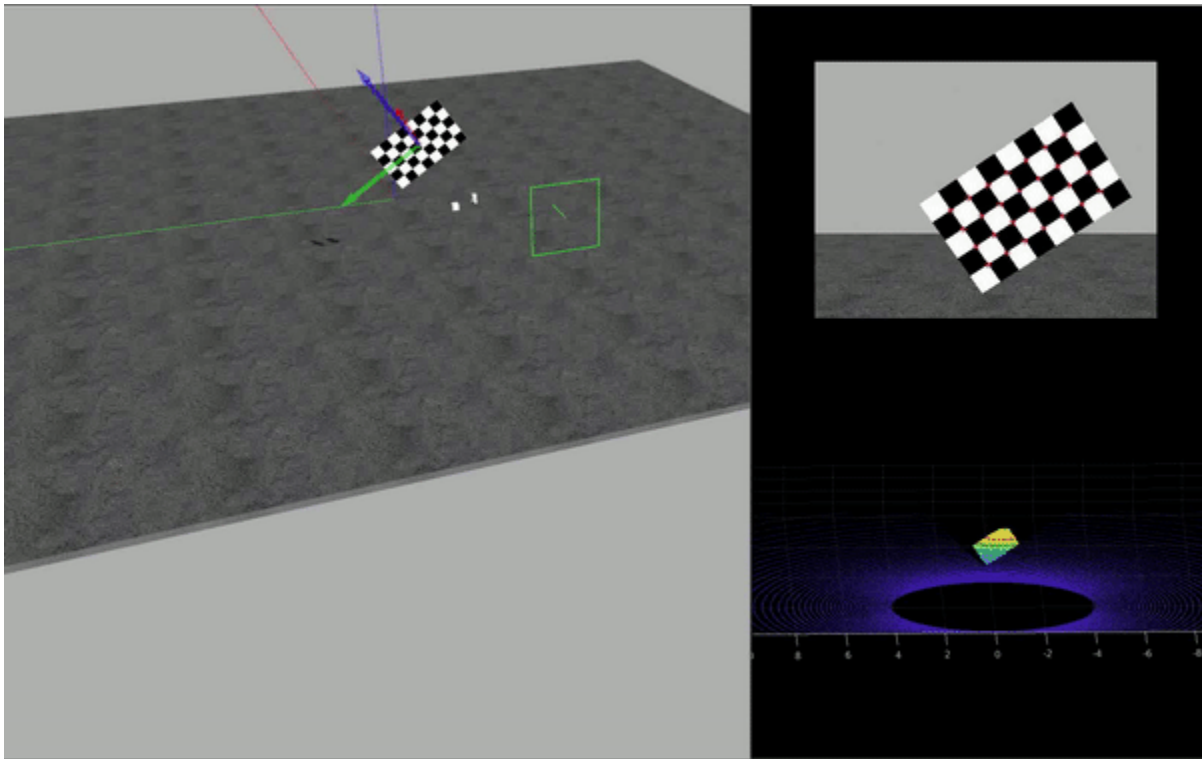
This diagram explains the workflow for the lidar and camera calibration (LCC) process.



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

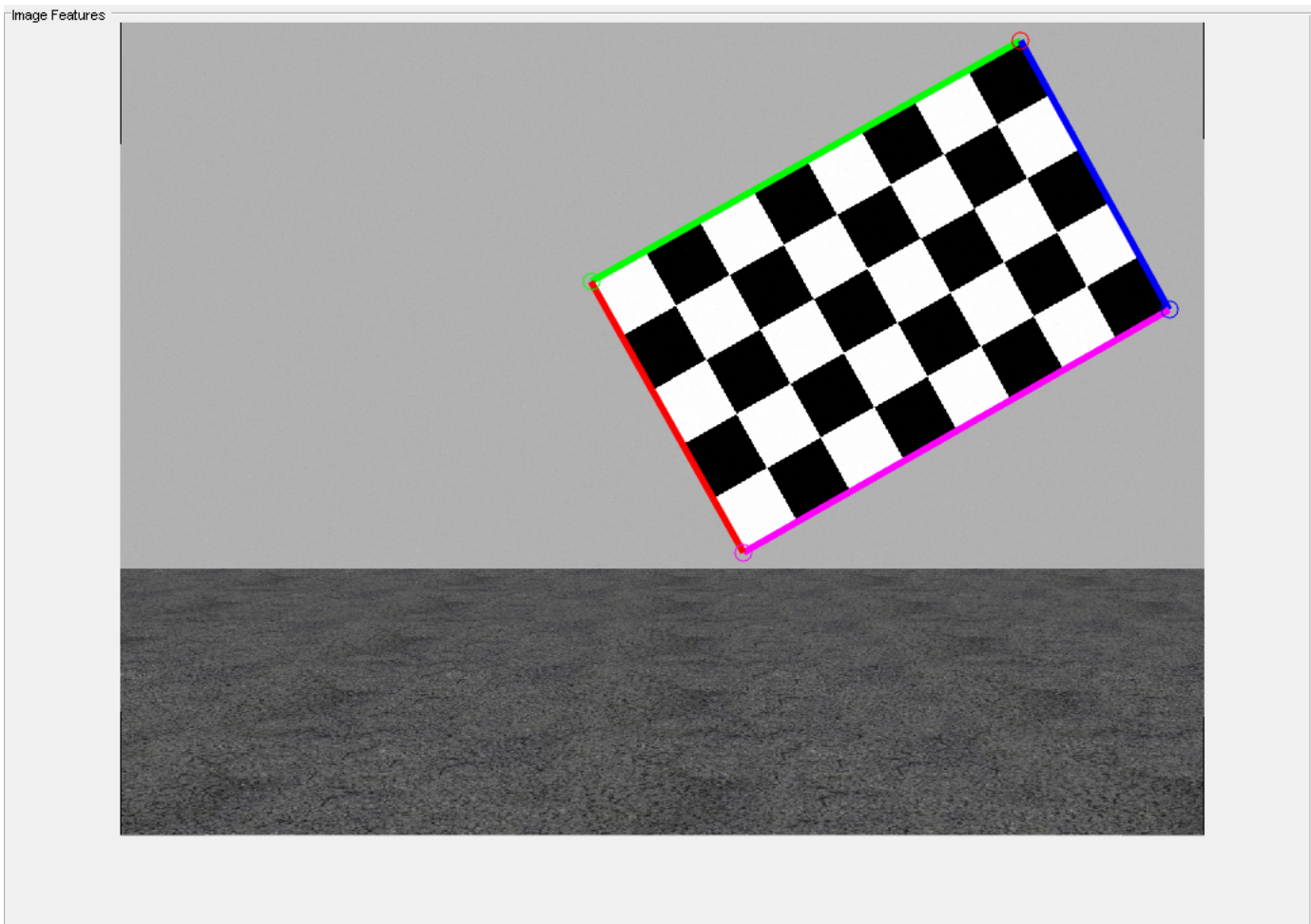
This example uses a checkerboard pattern as the feature of comparison. Checkerboard edges are estimated using lidar and camera sensors. Use `estimateCheckerboardCorners3d` to calculate coordinates of the checkerboard corners and size of actual checkerboard in *mm*. The corners are

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);

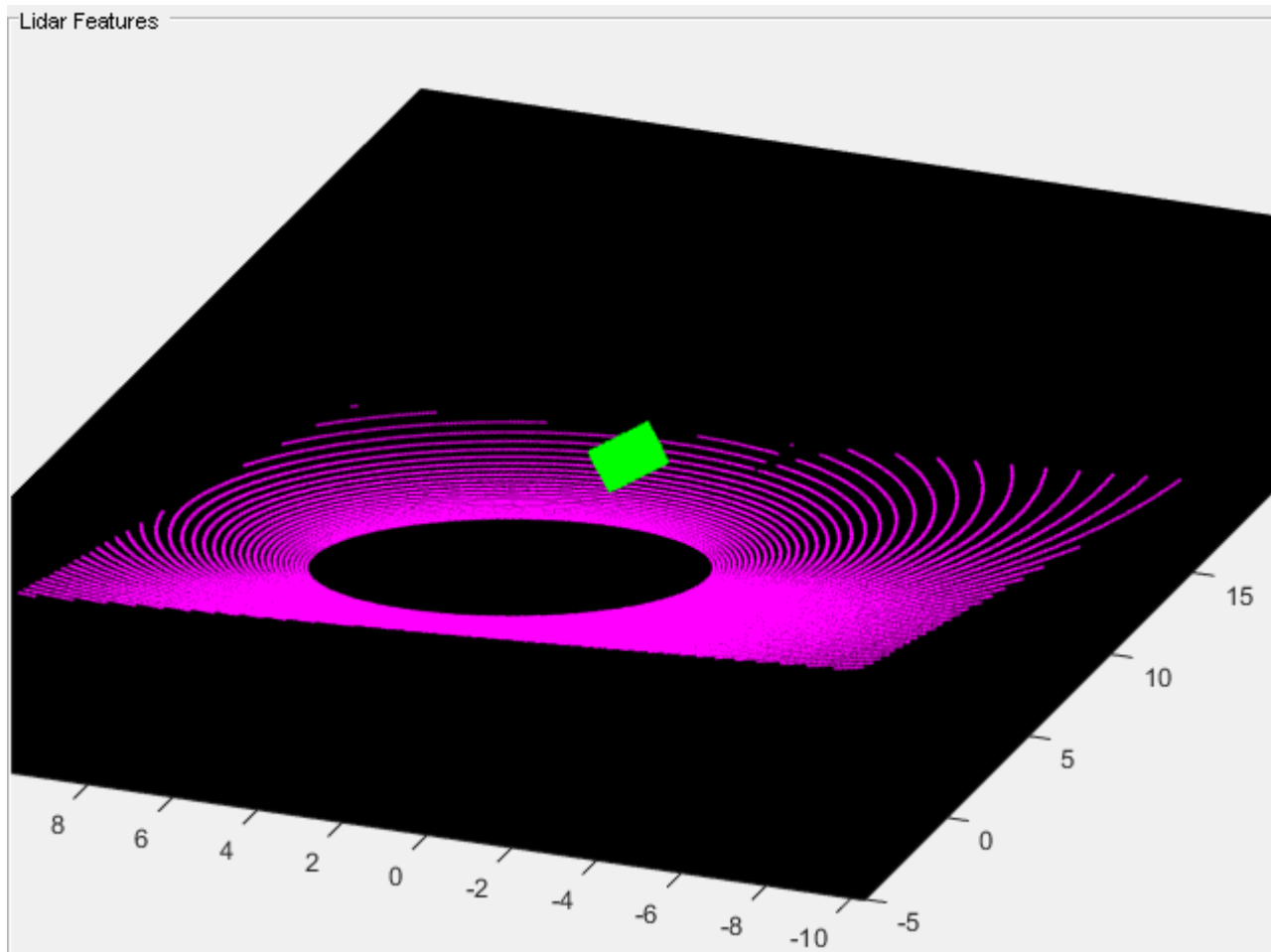
% Filter point cloud files corresponding to the detected images
ptCloudFileNames = ptCloudFileNames(dataUsed);
```

```
[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 the helperShowCheckerboardPlanes function.

```
helperShowCheckerboardPlanes(ptCloudFileNames, indices);
```



Calibrating Lidar and Camera

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

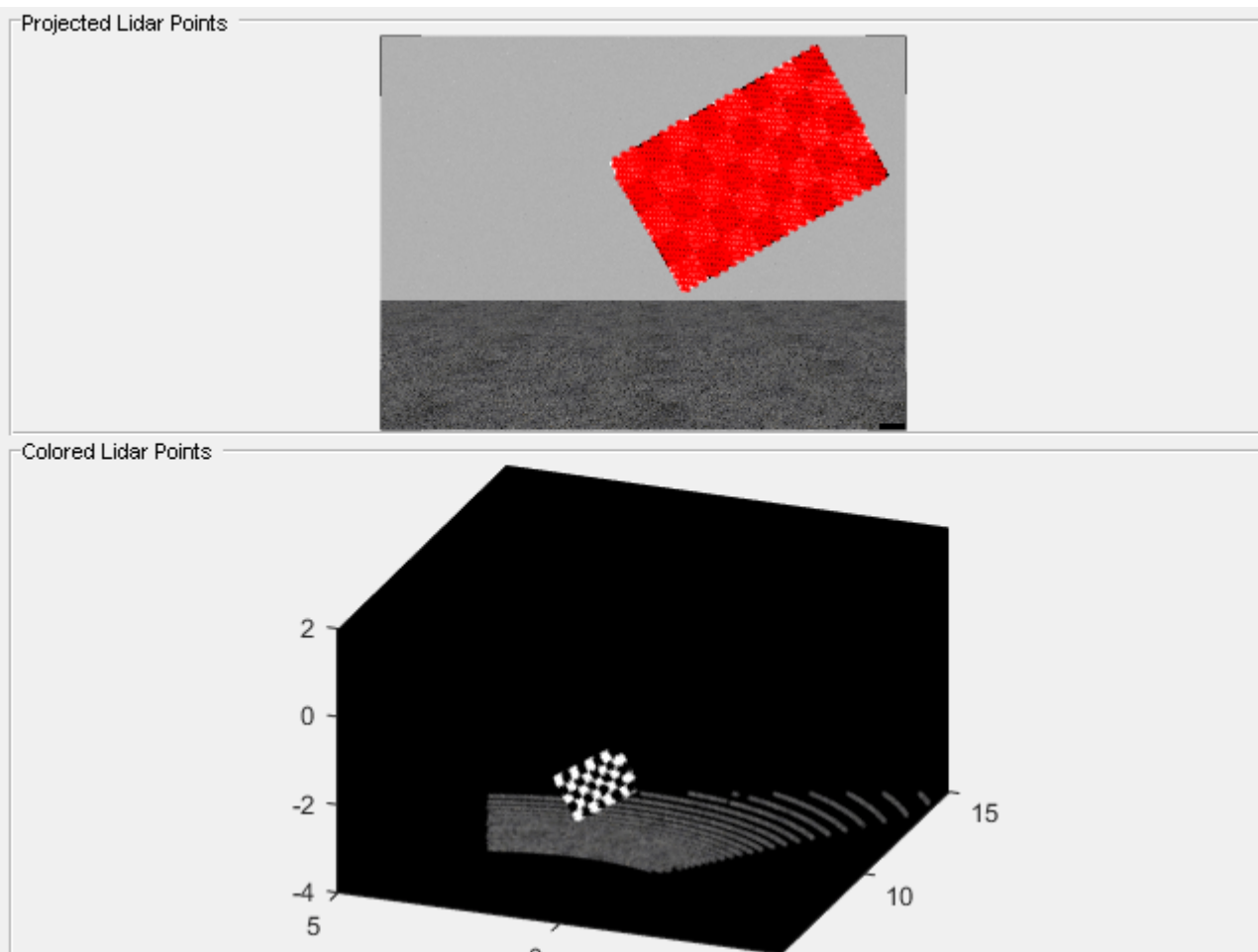
```
[tform, errors] = estimateLidarCameraTransform(lidarCheckerboardPlanes, ...
    imageCorners3d, 'CameraIntrinsic', intrinsic.cameraParams);
```

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

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

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

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



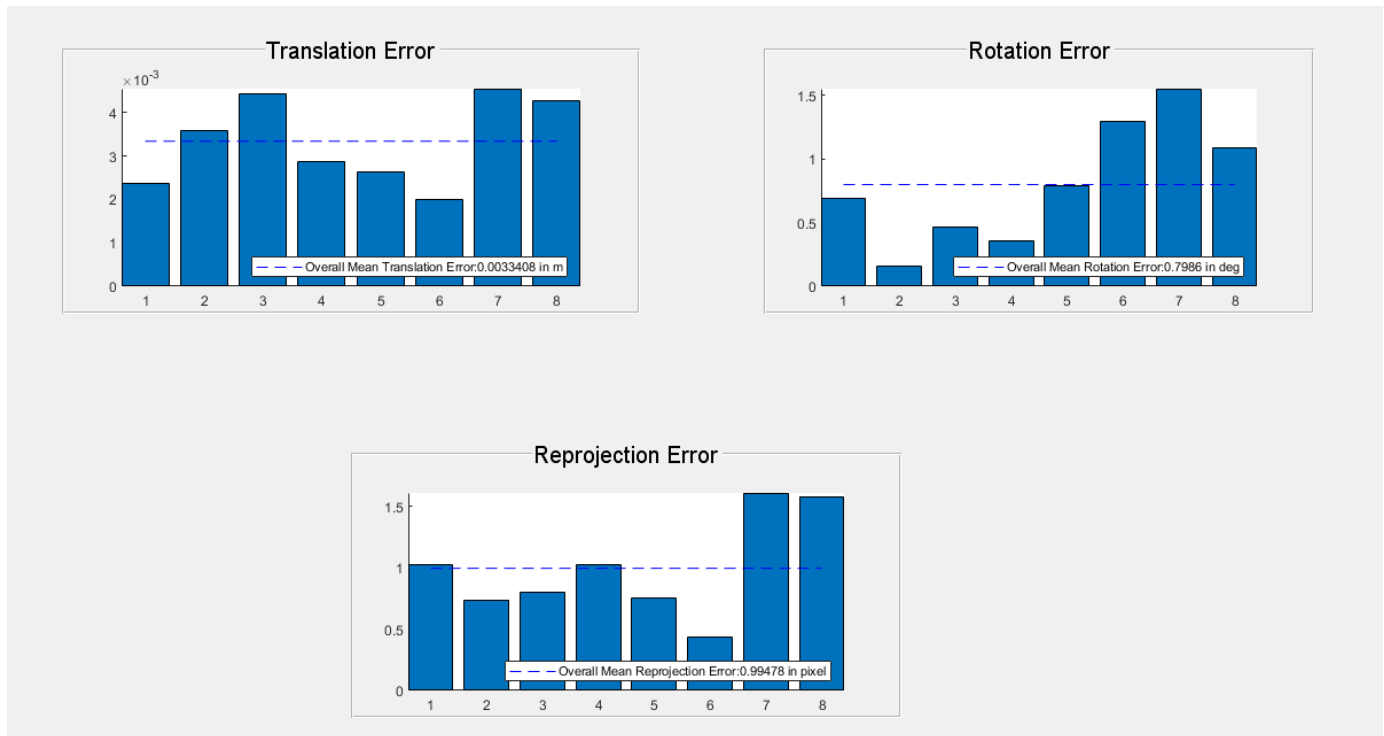
Error Visualization

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 intrinsics
imds = imageDatastore(imagePath); % Load images using imageDatastore
pcds = fileDatastore(ptCloudPath, 'ReadFcn', @pcread); % Load 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);

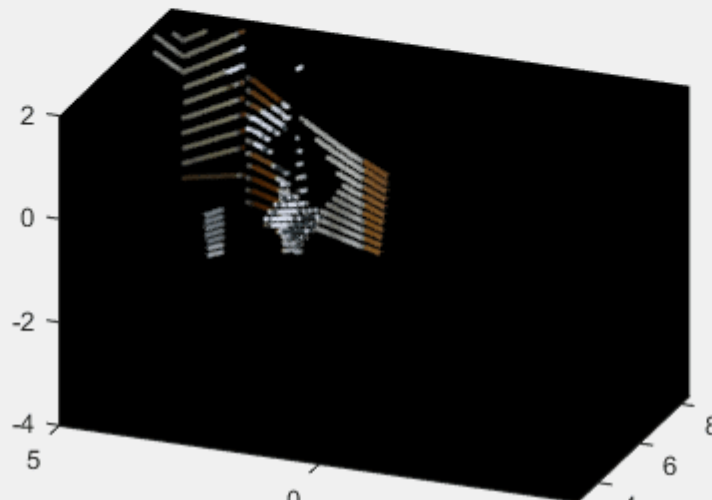
% Extract ROI from the detected image corners
```

```
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=3x3 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 = 3x1

    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. For more information, see “GPU Support by Release” (Parallel Computing Toolbox).

```
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.

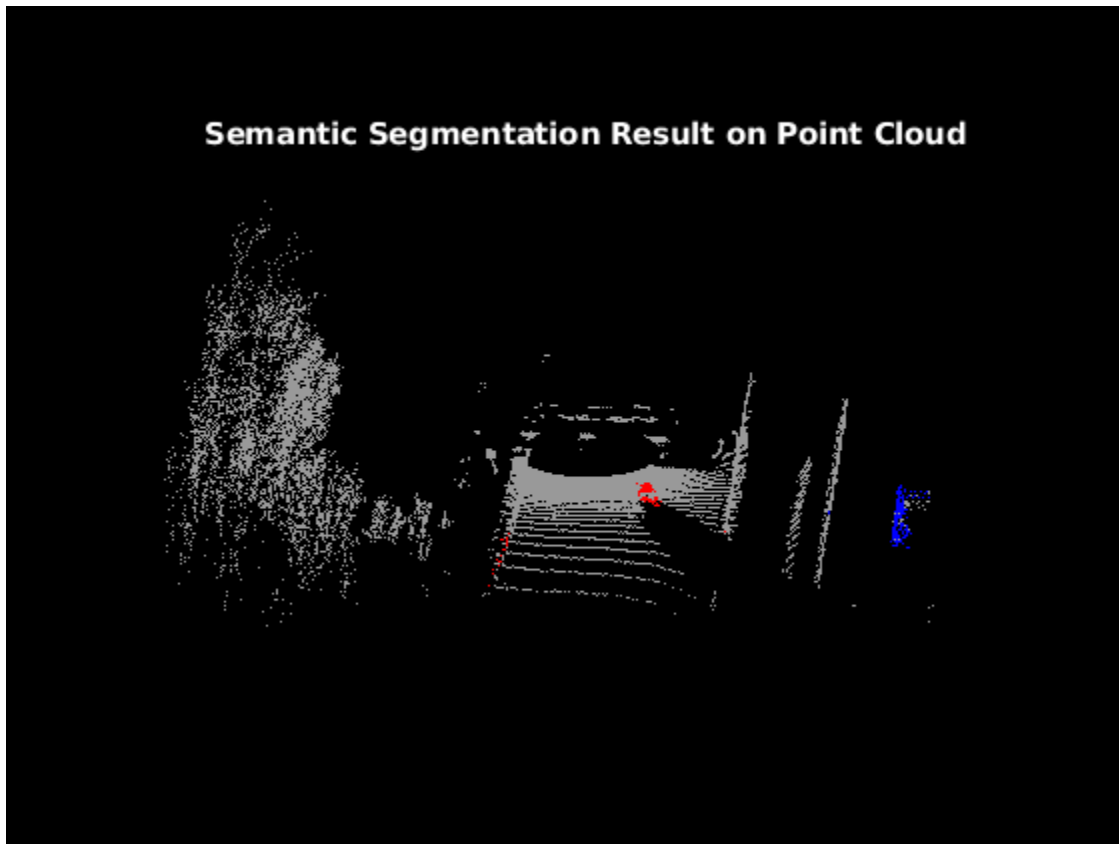
First, read a PCD file and convert the point cloud to a five-channel input image. Predict the labels using the trained network. Display the figure with the segmentation as an overlay.

```
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=1x5 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=3x3 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 = carClassColor;
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.

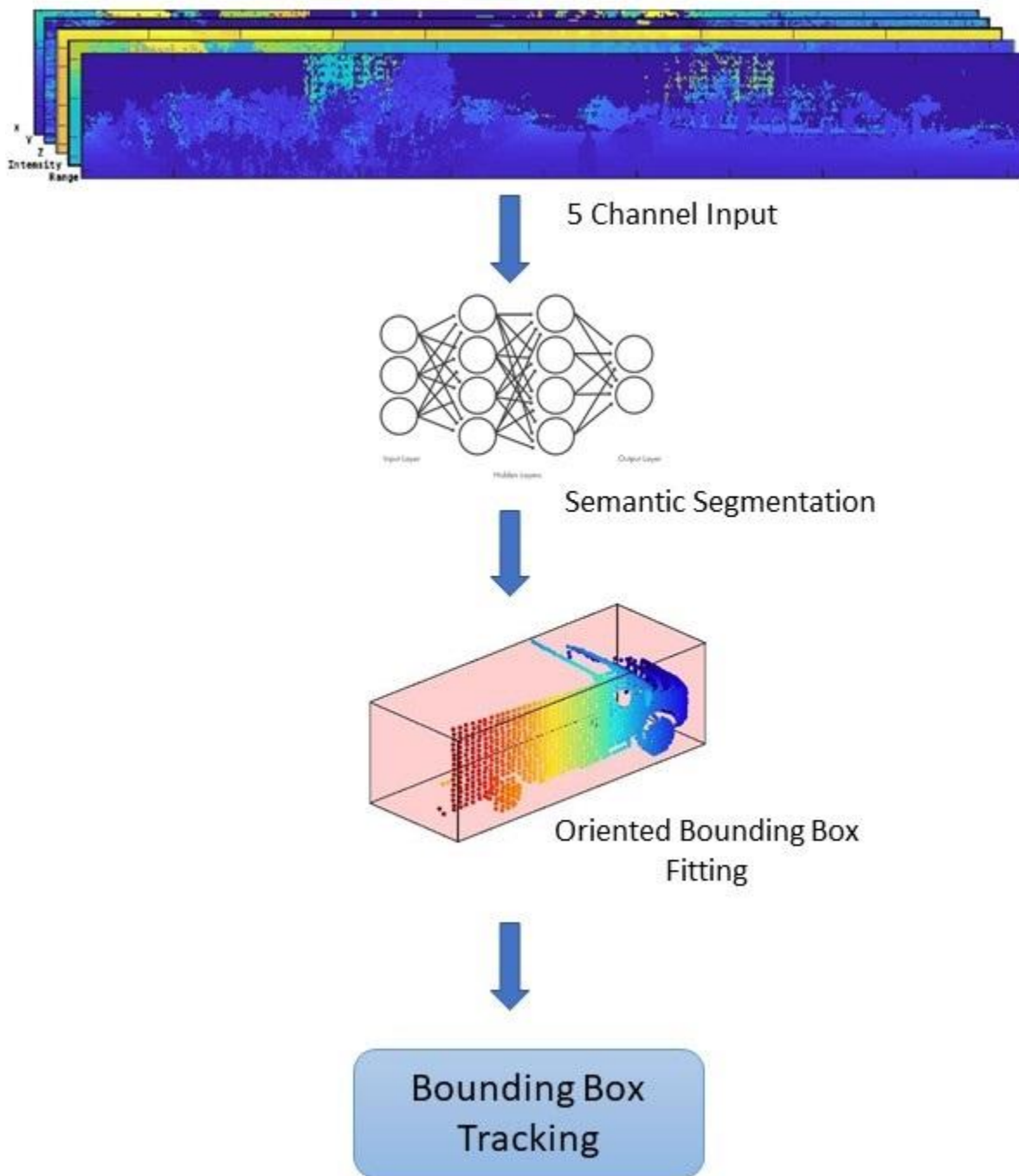
Overview

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. Use the helper function `helperDownloadData` to download the data and load them into the MATLAB® workspace.

Note: This download can take a few minutes.

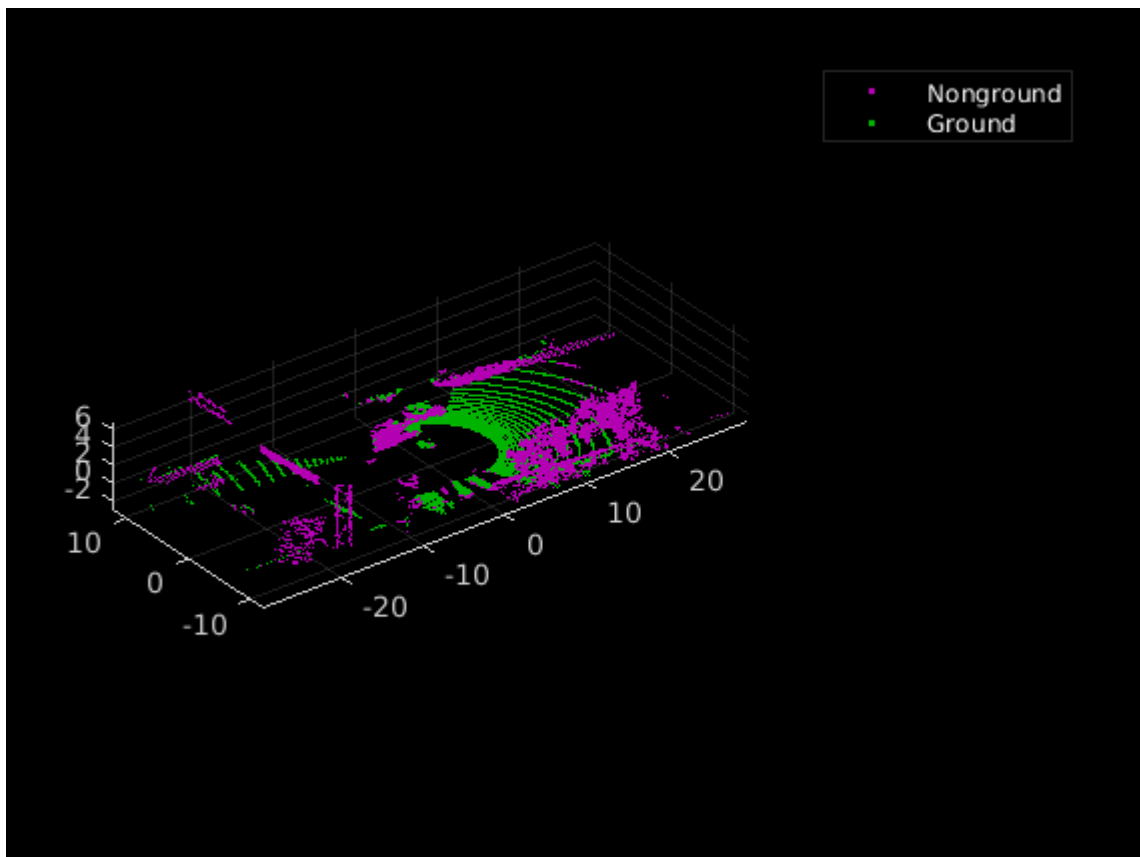
```
[ptClouds, pretrainedModel] = helperDownloadData;
```

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
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” on page 1-30 example.

Prepare Input Data

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

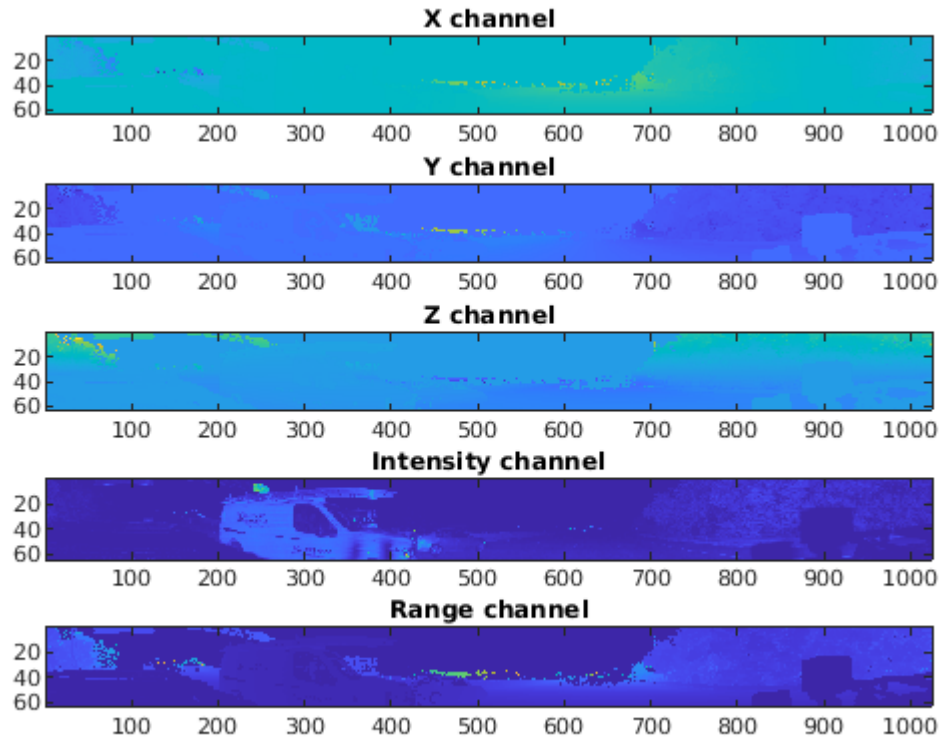
```
% Load and visualize a sample frame
frame = helperPrepareData(ptCloud);
figure;
subplot(5,1,1);
imagesc(frame(:,:,1));
title('X channel');

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');
```



Run forward inference on one frame from the loaded pre-trained network.

```

if ~exist('net','var')
    net = pretrainedModel.net;
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,'Transparent');

% Display results
figure;
imshow(segmentedImage);
helperPixelLabelColorbar(lidarColorMap,classes);

```

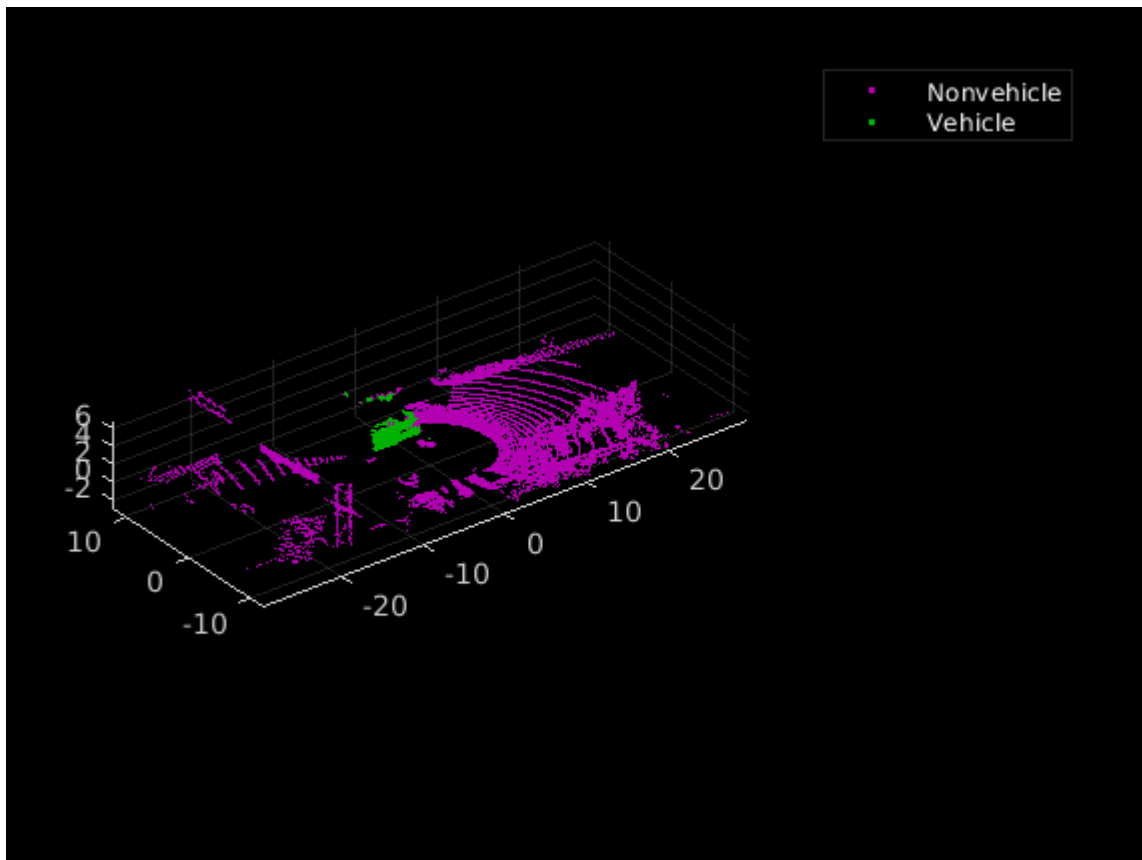


Use the generated semantic mask to filter point clouds containing trucks. Similarly, filter point clouds for other classes.

```
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');
```

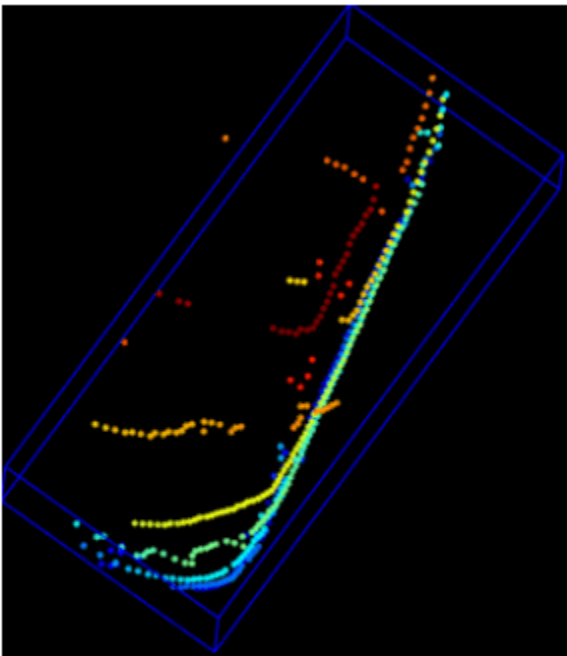


Clustering and Bounding Box Fitting

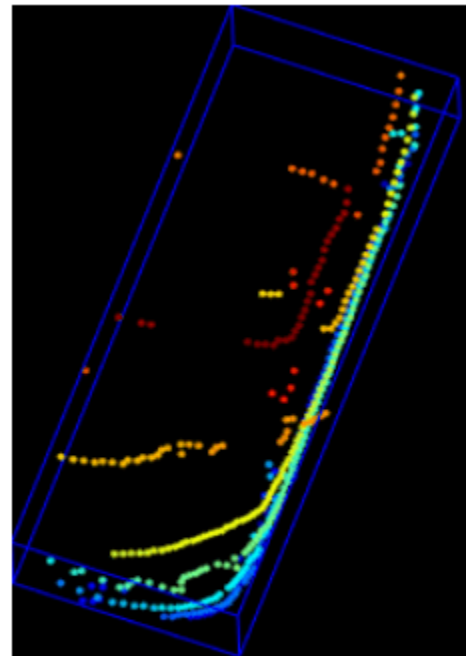
After extracting point clouds of different object classes, the objects are clustered by applying Euclidean clustering using the `pcsegdist` function. To group all the points belonging to one single

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)));

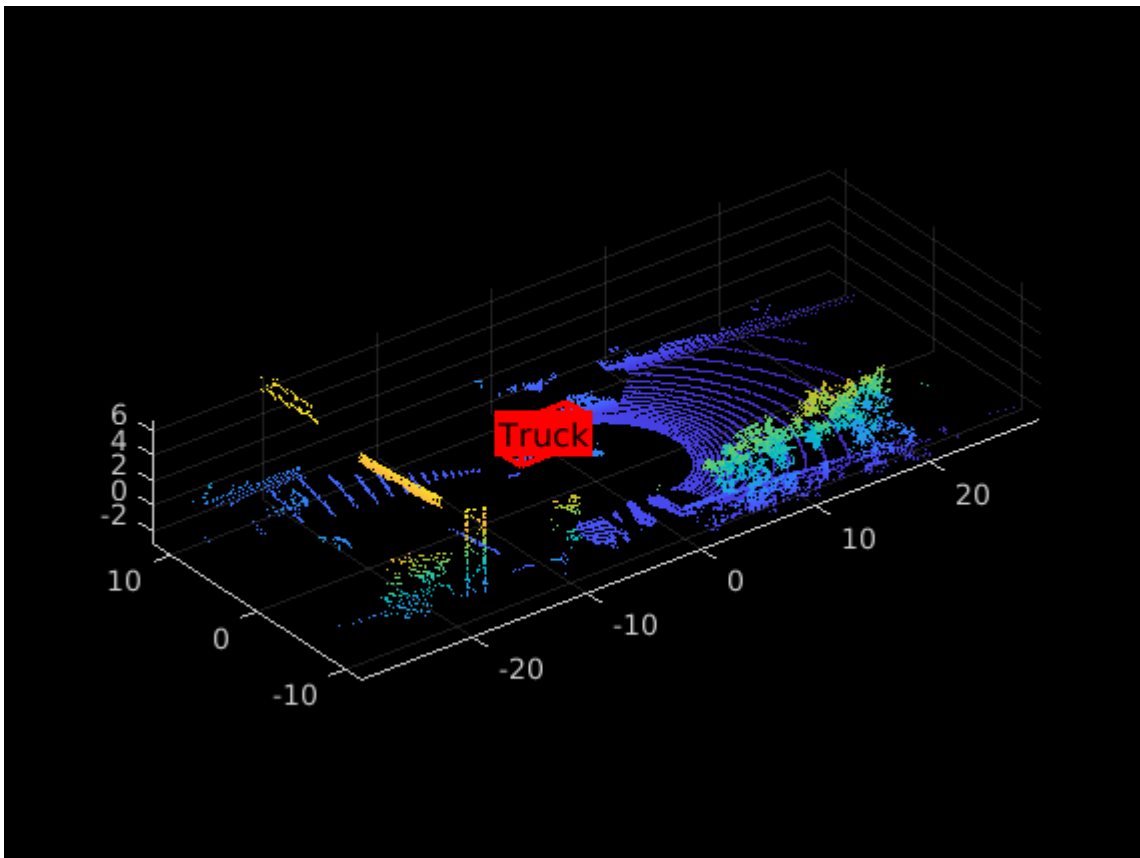
    % Filter small bounding boxes
    if size(location,1)*size(location,2) > 20 && any(any(pc.Location)) && xl > 1 && yl > 1
```

```
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('Model',net,'XLimits',xLimit,...
    'YLimit',yLimit,'ZLimit',zLimit);

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

% 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);

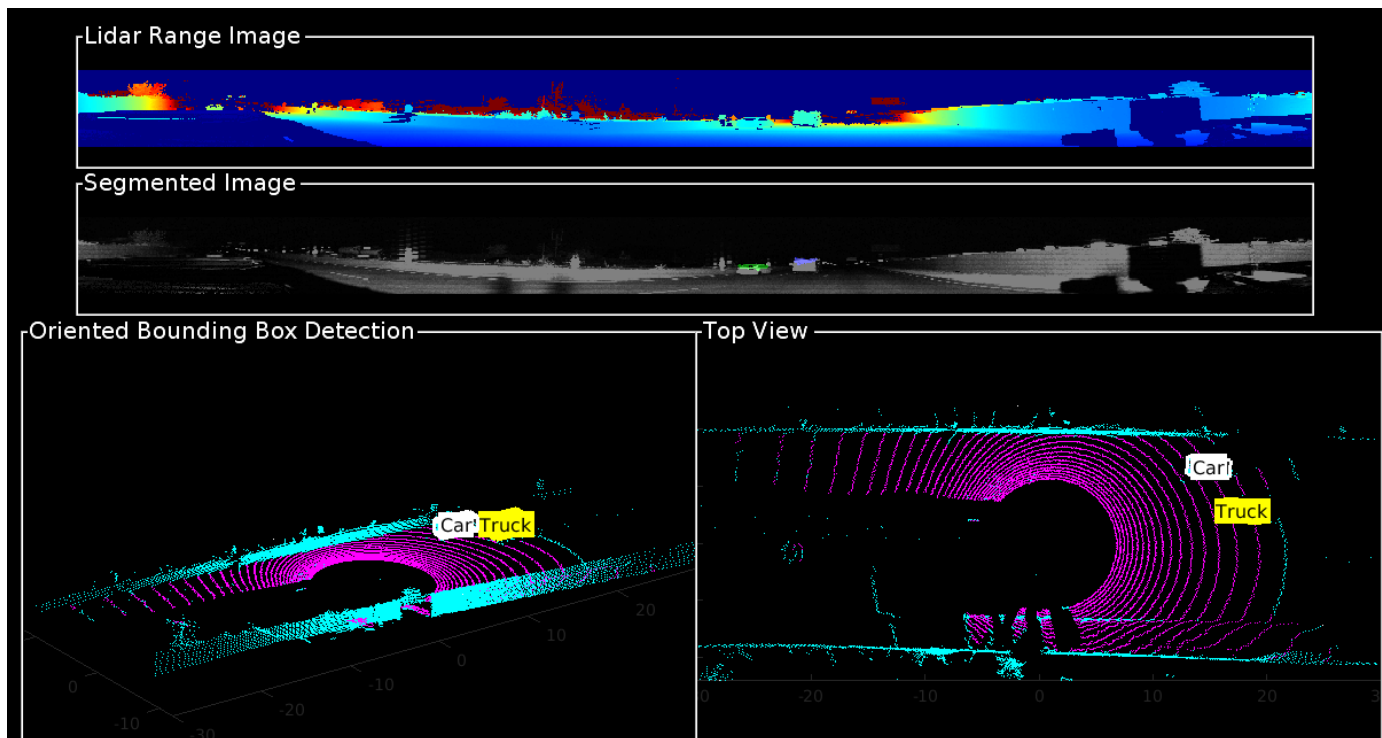
    % Extact bounding boxes from lidar data
    [boundingBox,coloredPtCloud,pointLabels] = detectBbox(lidarDetector,input);

    % 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 the “Track Vehicles Using Lidar: From Point Cloud to Track List” (Sensor Fusion and Tracking Toolbox) example. 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 logic
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);

    % Extract 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

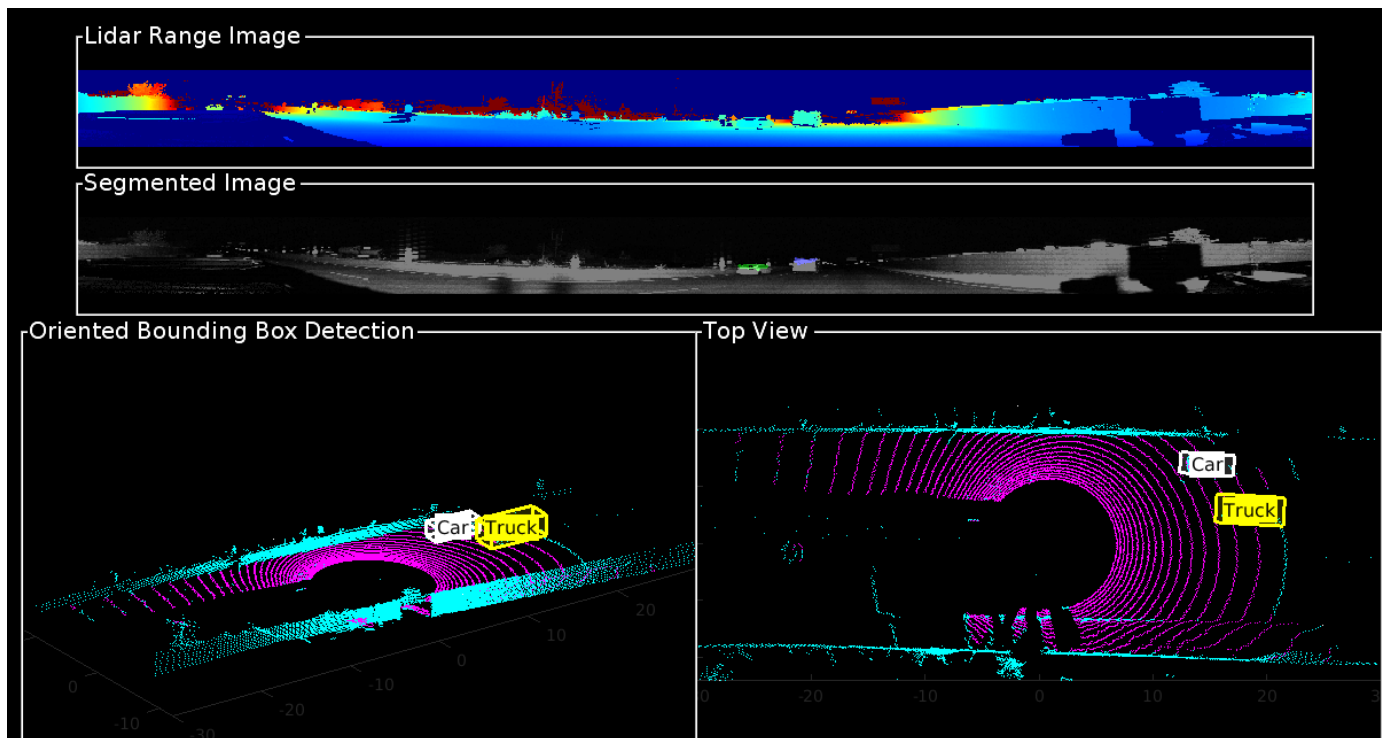
    % Update display with colored point cloud
    updatePointCloud(display,coloredPtCloud);

    % 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
```

helperDownloadData

```
function [lidarData, pretrainedModel] = helperDownloadData
outputFolder = fullfile(tempdir,'WPI');
url = 'https://ssd.mathworks.com/supportfiles/lidar/data/lidarSegmentationAndTrackingData.tar.gz';
lidarDataTarFile = fullfile(outputFolder,'lidarSegmentationAndTrackingData.tar.gz');
if ~exist(lidarDataTarFile,'file')
    mkdir(outputFolder);
    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
% Load lidar data
data = load(fullfile(outputFolder,'highwayData.mat'));
lidarData = data.ptCloudData;

% Download pretrained model
url = 'https://ssd.mathworks.com/supportfiles/lidar/data/pretrainedPointSegModel.mat';
modelFile = fullfile(outputFolder,'pretrainedPointSegModel.mat');
if ~exist(modelFile,'file')
    websave(modelFile,url);
end
pretrainedModel = load(fullfile(outputFolder,'pretrainedPointSegModel.mat'));
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 typical approach 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
```

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 Kd-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=8x1 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];

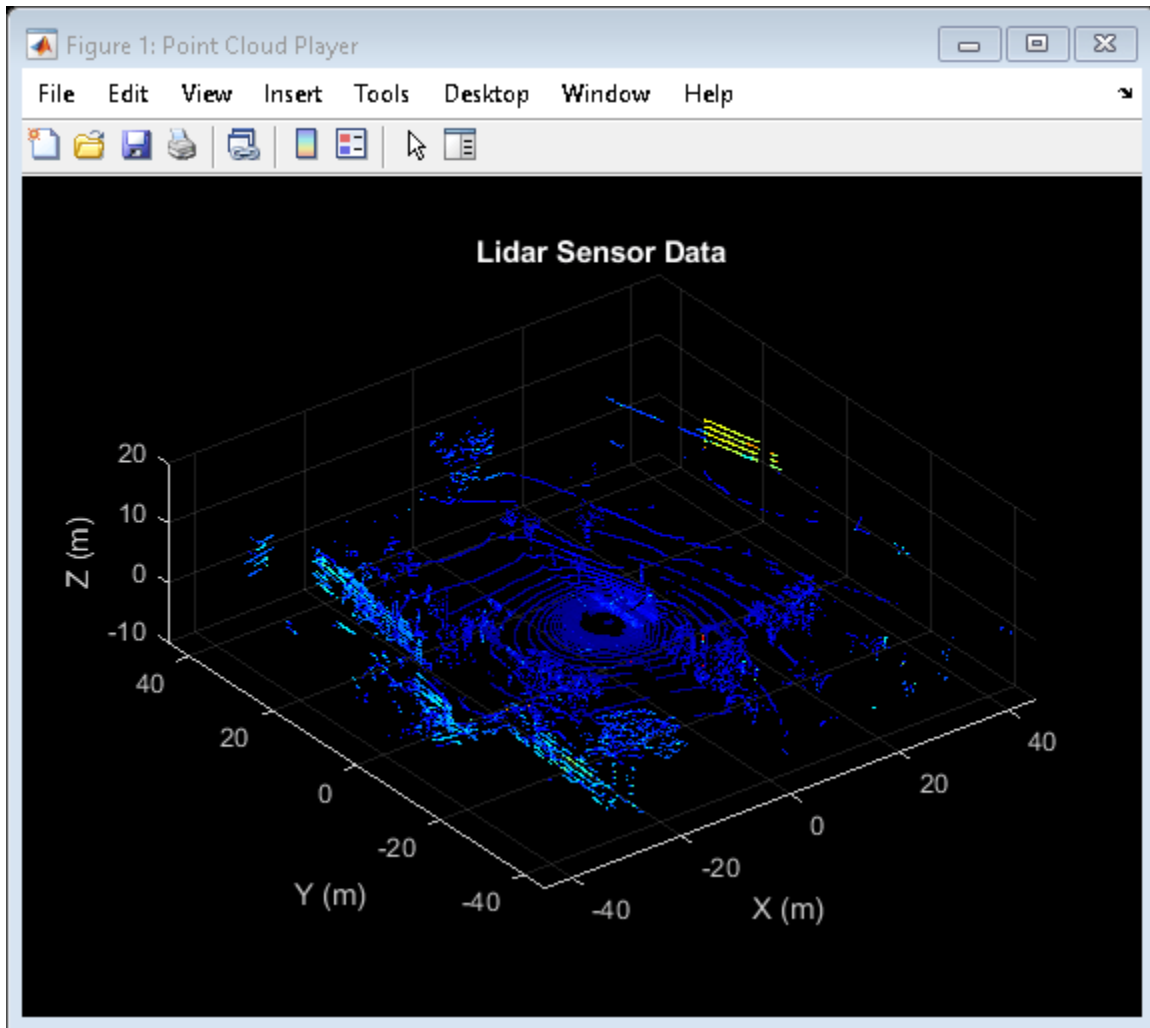
% Create a pcplayer to visualize streaming point clouds from lidar sensor
lidarPlayer = pcplayer(xlimits, ylimits, zlimits);

% 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 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.

Preprocessing

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('default');  
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.

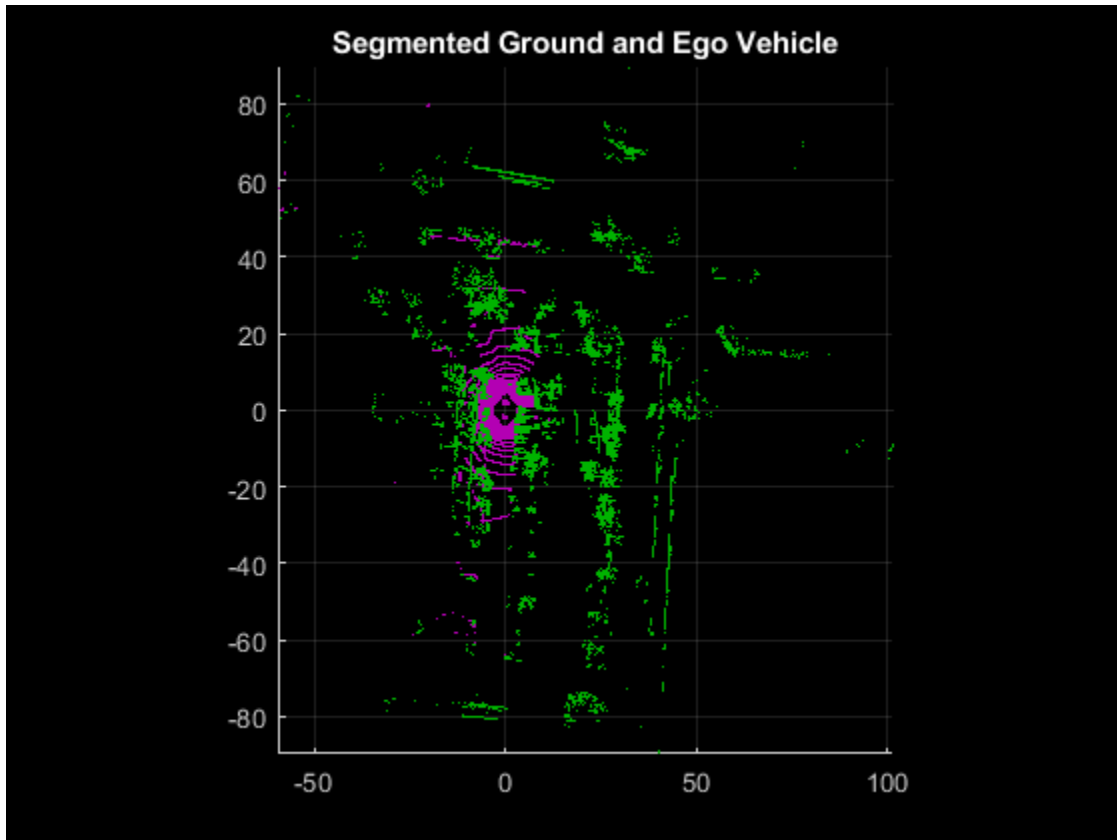
These steps are described in more detail in the “Ground Plane and Obstacle Detection Using Lidar” (Automated Driving Toolbox) example.

```
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 improve registration accuracy and algorithm speed.

```
gridStep = 0.5;
fixedDownsampled = pcdsample(fixedProcessed, "gridAverage", gridStep);
movingDownsampled = pcdsample(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 = 40;
[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 = 'Exhaustive';
threshold = 1;
ratio = 0.96;
indexPairs = pcmatchfeatures(movingFeature, fixedFeature, movingValidPts, ...
    fixedValidPts, "Method", method, "MatchThreshold", threshold, ...
    "RejectRatio", ratio);

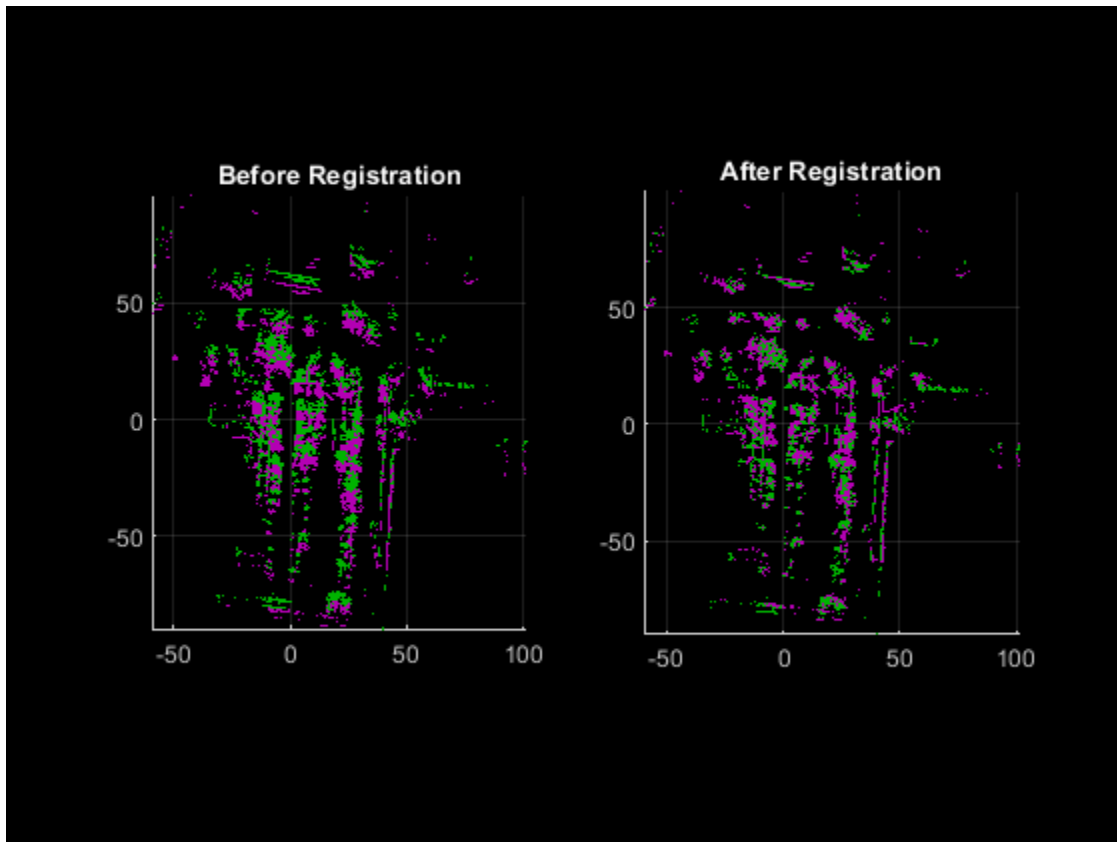
matchedFixedPts = select(fixedValidPts, indexPairs(:,2));
matchedMovingPts = select(movingValidPts, indexPairs(:,1));

% Estimate rigid transform of moving point cloud with respect to reference
% point cloud
maxDist = 2;
maxNumTrials = 3000;
tform = estimateGeometricTransform3D(matchedMovingPts.Location, ...
    matchedFixedPts.Location, "rigid", "MaxDistance", maxDist, ...
    "MaxNumTrials", maxNumTrials);

% 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);

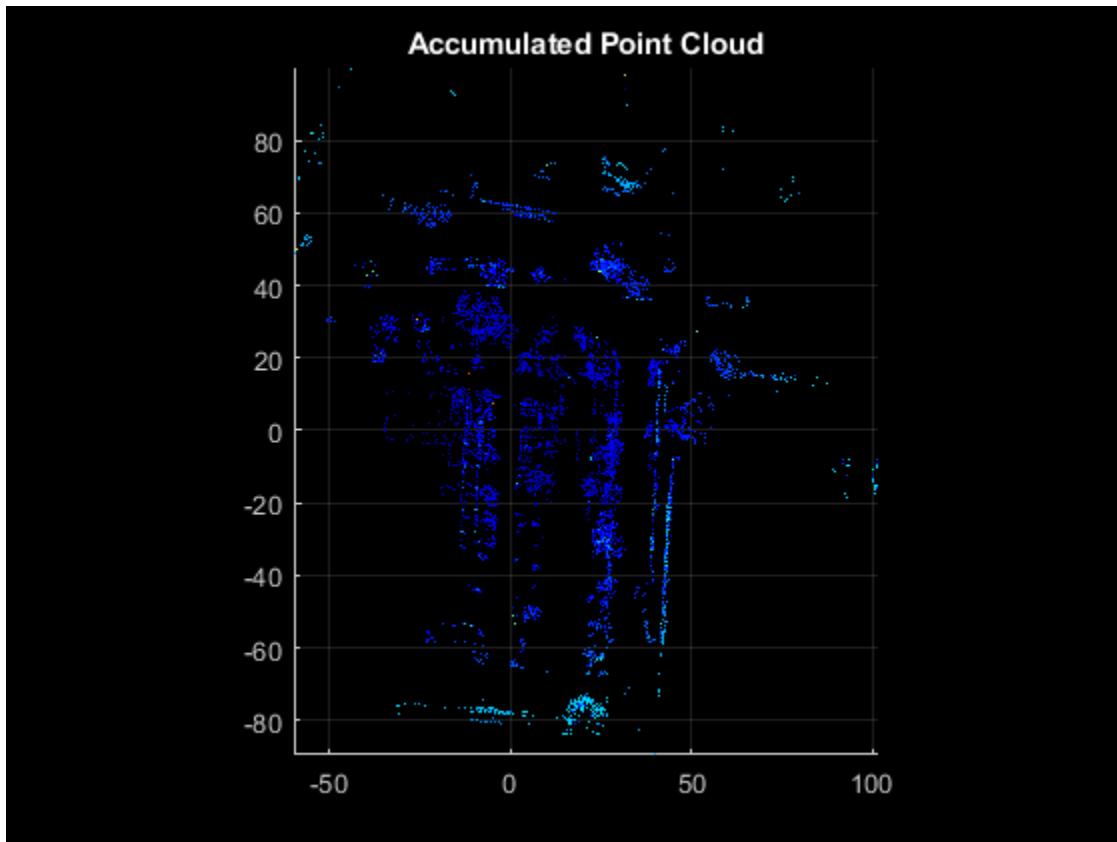
axAlign2 = subplot(1, 2, 2, 'Color', [0, 0, 0], 'Parent', hFigAlign);
pcshowpair(movingReg, fixedProcessed, 'Parent', axAlign2);
title(axAlign2, 'After Registration');
view(axAlign2, 2);
```



```
% 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 preprocessing 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('default');
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 = pcdownsampling(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));

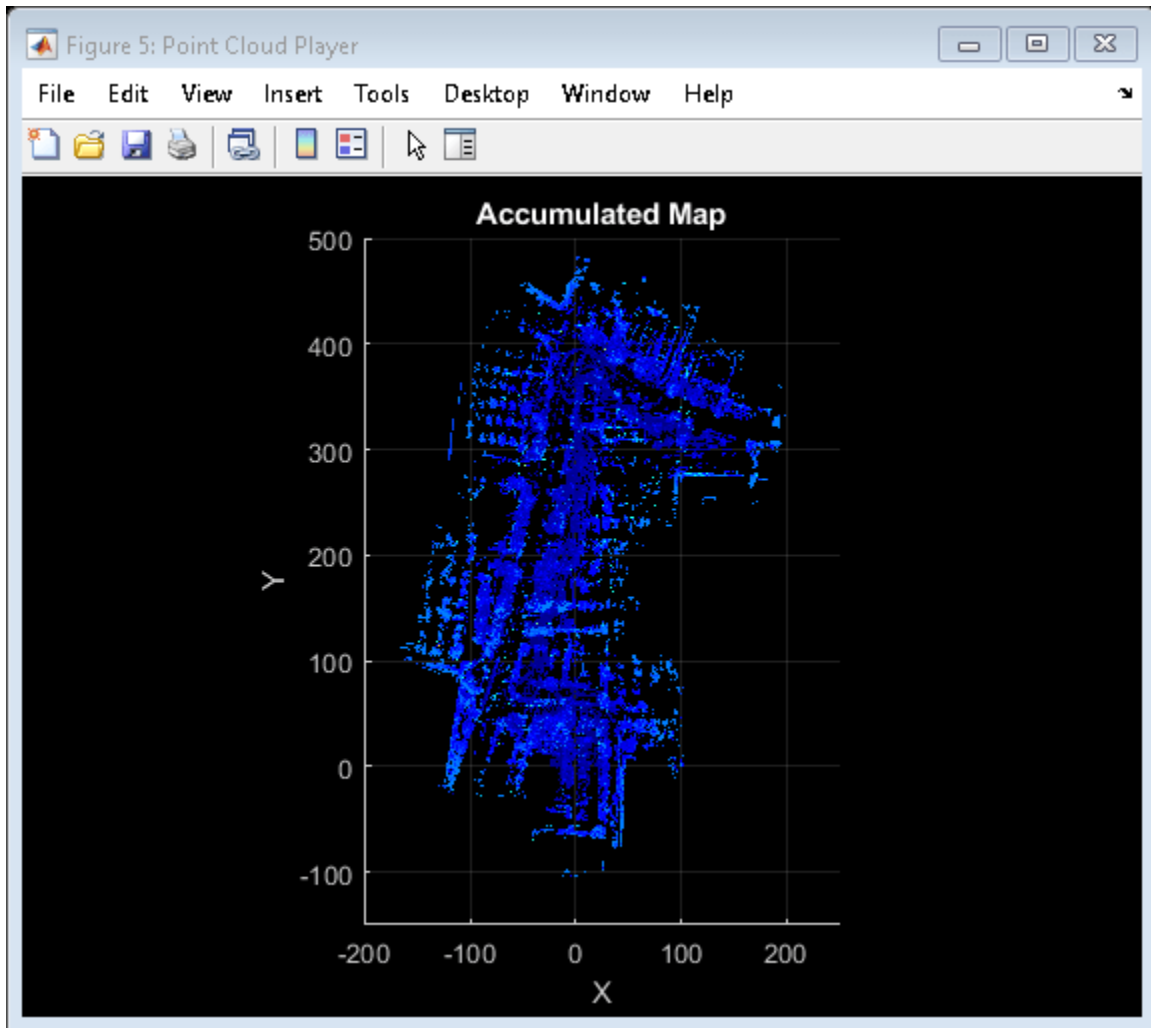
    % Register moving point cloud w.r.t reference point cloud
    tform = estimateGeometricTransform3D(matchedMovingPts.Location, ...
        matchedFixedPts.Location, "rigid", "MaxDistance", maxDist, ...
        "MaxNumTrials", maxNumTrials);

    % 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

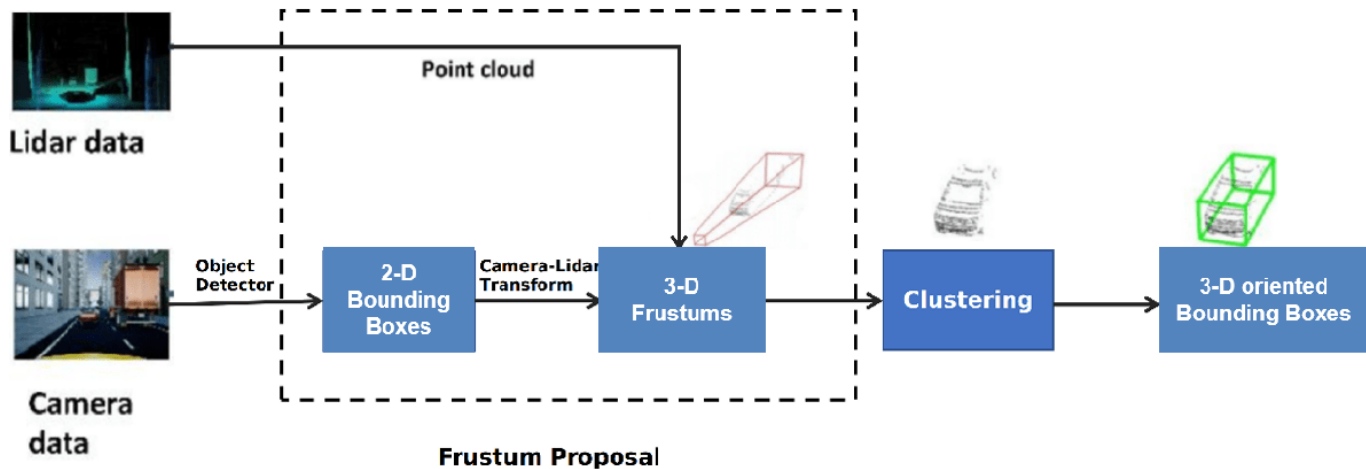
- “Build a Map from Lidar Data” (Automated Driving Toolbox)
- “Ground Plane and Obstacle Detection Using Lidar” (Automated Driving Toolbox)

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 1-21.

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.

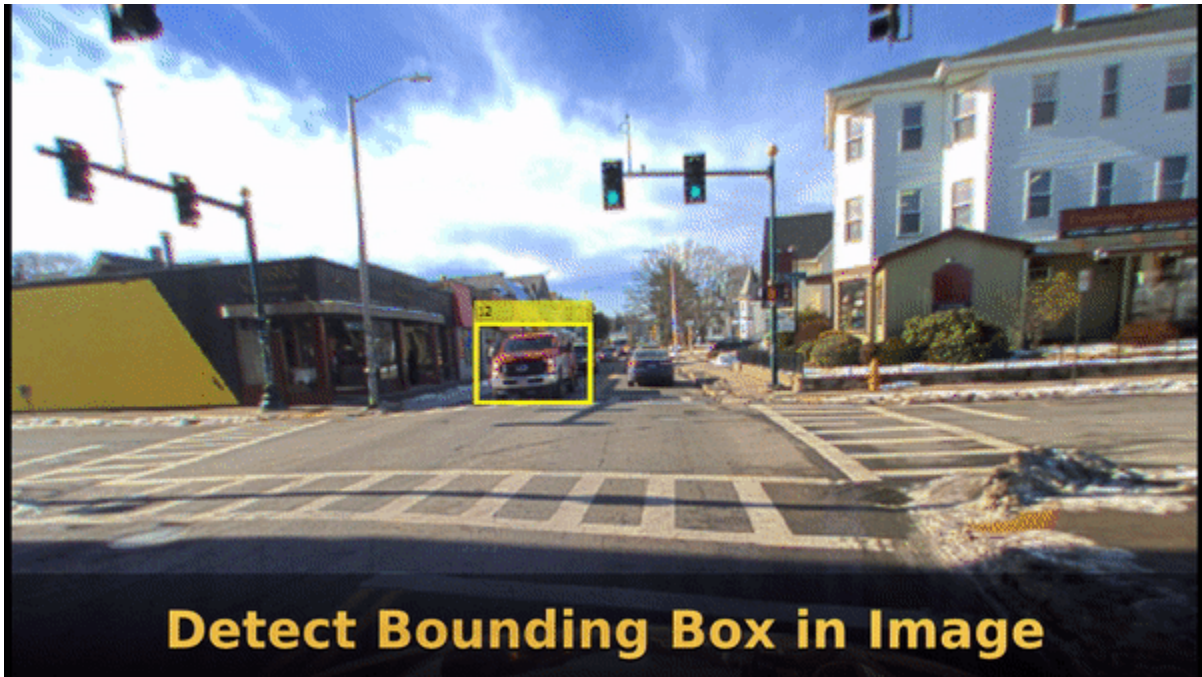
```
load imageGTruth.mat
im = imread(imageFileNames{50});
imBbox = imageGTruth{50};

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.



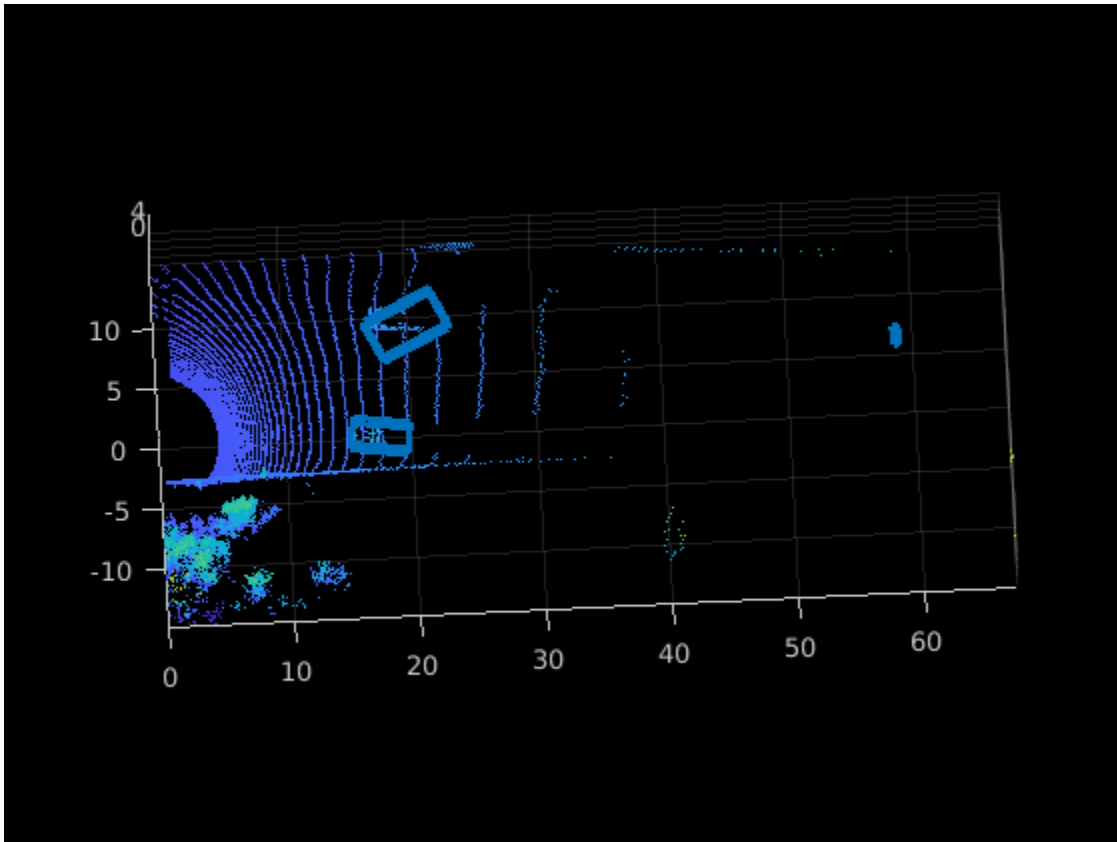
The 3-D cuboids are represented as vectors of the form: $[xcen\ ycen\ zcen\ dimx\ dimy\ dimz\ rotx\ roty\ rotz]$, where $xcen$, $ycen$, and $zcen$ represent the centroid coordinates of the cuboid. $dimx$, $dimy$, and $dimz$ represent the length of the cuboid along the x -, y -, and z -axes, and $rotx$,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

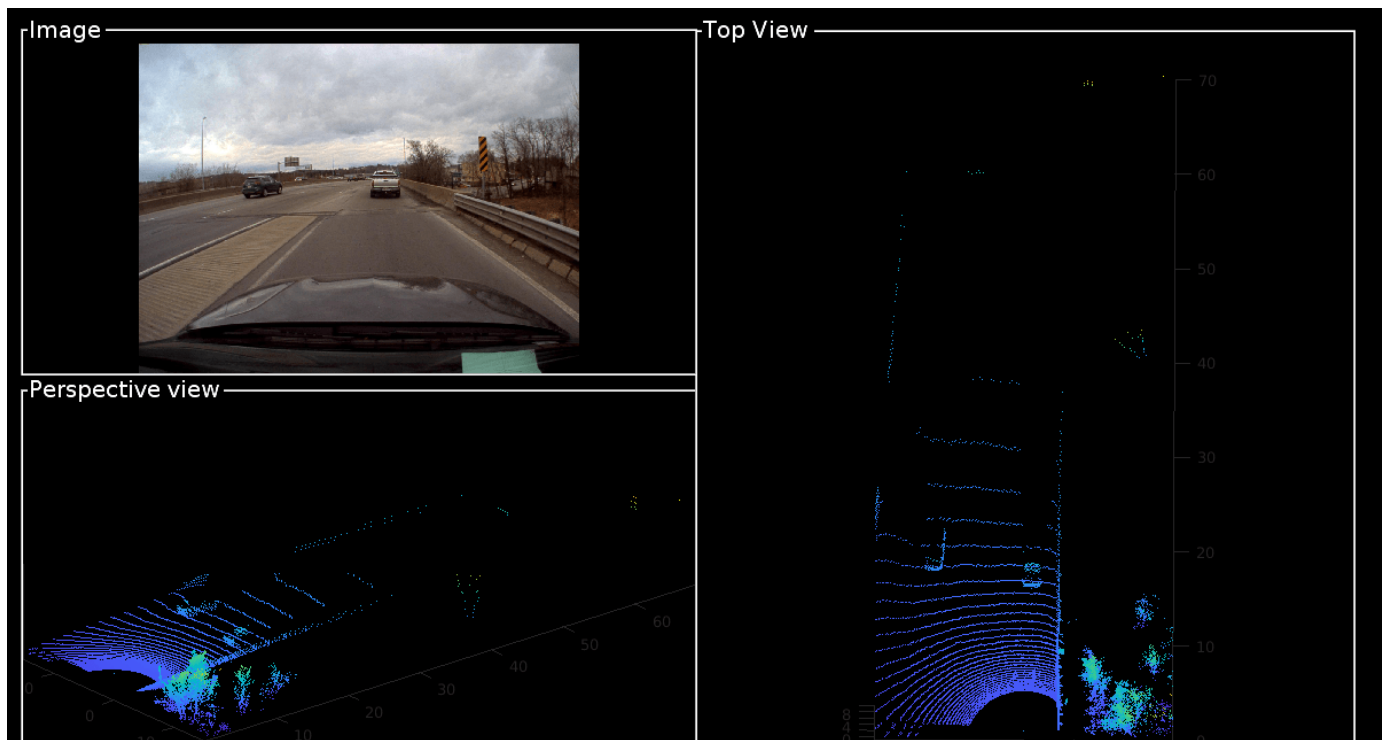
Set Up Display

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 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

```

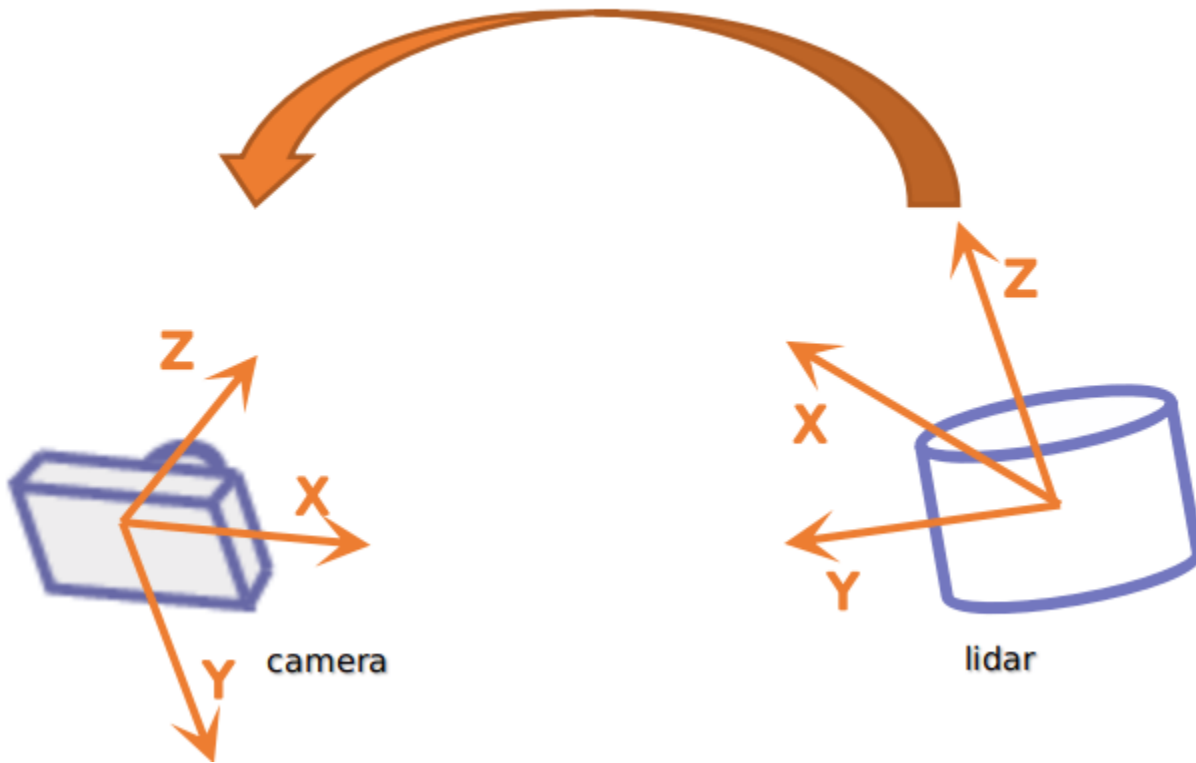


Detected bounding boxes by using bounding box tracking, such as joint probabilistic data association (JPDA). For more information, see “Track Vehicles Using Lidar: From Point Cloud to Track List” on page 1-131.

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.

lidar camera calibration



```

% 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);
    end
end

```

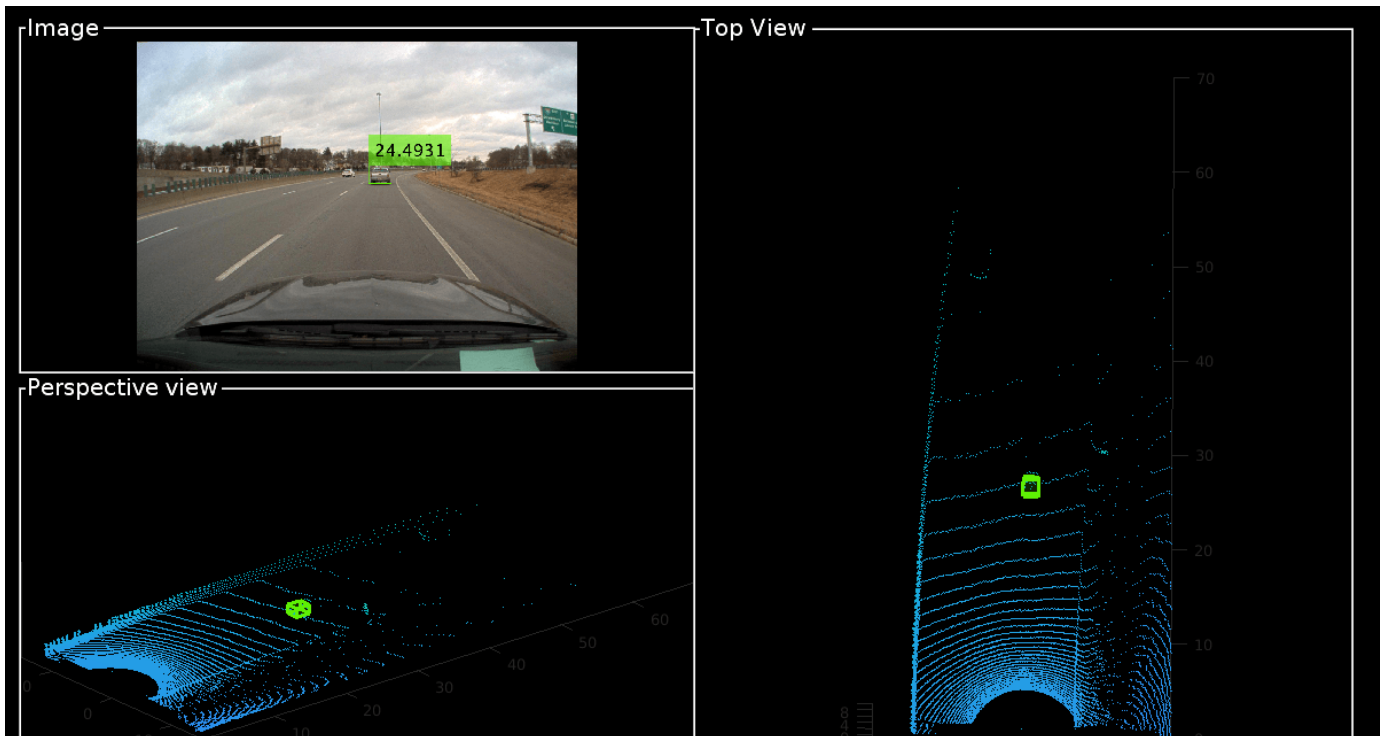


```

        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 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 PandaSet data set from Hesai and Scale [2] on page 1-0 . The PandaSet contains 4800 unorganized lidar point cloud scans of the various city scenes captured using the Pandar 64 sensor. The data set provides semantic segmentation labels for 42 different classes including car, road, and pedestrian.

Download Lidar Data Set

This example uses a subset of PandaSet, that contains 2560 preprocessed organized point clouds. Each point cloud is specified as a 64-by-1856 matrix. The corresponding ground truth contains the semantic segmentation labels for 12 classes. The point clouds are stored in PCD format, and the ground truth data is stored in PNG format. The size of the data set is 5.2 GB. Execute this code to download the data set.

```
url = 'https://ssd.mathworks.com/supportfiles/lidar/data/Pandaset_LidarData.tar.gz';
outputFolder = fullfile(tempdir, 'Pandaset');
lidarDataTarFile = fullfile(outputFolder, 'Pandaset_LidarData.tar.gz');
if ~exist(lidarDataTarFile, 'file')
    mkdir(outputFolder);
    disp('Downloading Pandaset Lidar driving data (5.2 GB)...');
    websave(lidarDataTarFile, url);
    untar(lidarDataTarFile, outputFolder);
end
% Check if tar.gz file is downloaded, but not uncompressed.
if (~exist(fullfile(outputFolder, 'Lidar'), 'file'))...
    && (~exist(fullfile(outputFolder, 'semanticLabels'), 'file'))
    untar(lidarDataTarFile, outputFolder);
end
lidarData = fullfile(outputFolder, 'Lidar');
labelsFolder = fullfile(outputFolder, 'semanticLabels');
```

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 Pandaset_LidarData folder. To use the file you downloaded from the web, change the outputFolder variable in the code to the location of the downloaded file.

The training procedure for this example is for organized point clouds. For an example showing how to convert unorganized to organized point clouds, see “Unorganized to Organized Conversion of Point Clouds Using Spherical Projection” on page 1-173.

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;
pretrainedNetURL = ...
'https://ssd.mathworks.com/supportfiles/lidar/data/trainedSqueezeSegV2PandasetNet.zip';
if ~doTraining
    downloadPretrainedSqueezeSegV2Net(outputFolder, pretrainedNetURL);
end
```

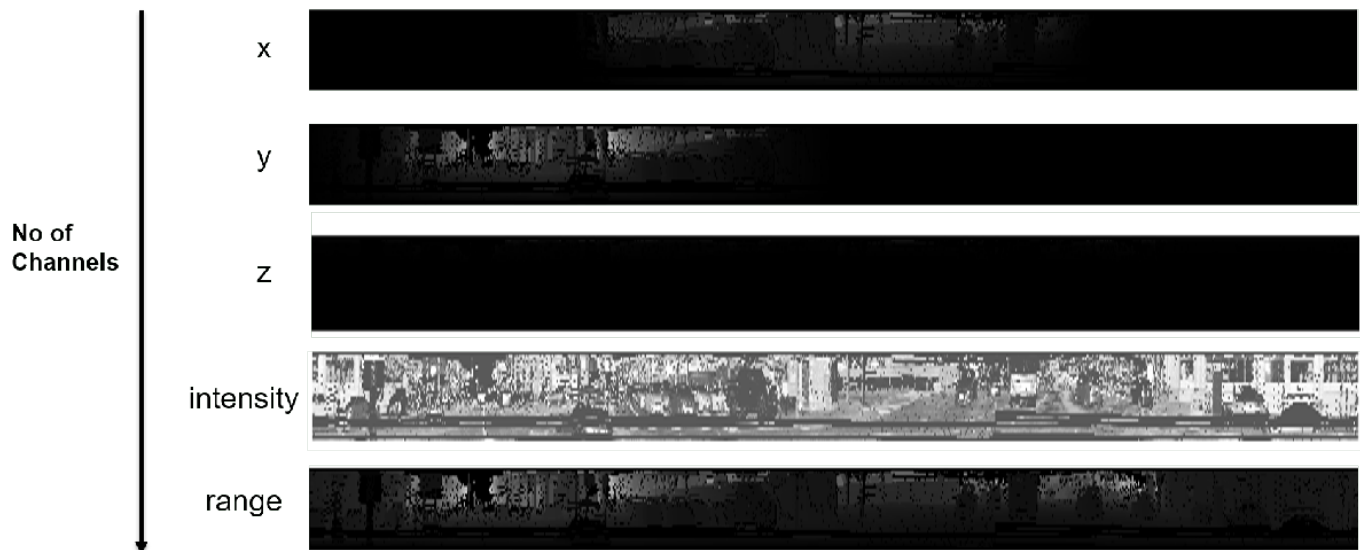
Prepare Data for Training

Load Lidar Point Clouds and Class Labels

Use the `helperTransformOrganizedPointCloudToTrainingData` supporting function, attached to this example, to generate training data from the lidar point clouds. The function uses point cloud data to create five-channel input images. Each training image is specified as a 64-by-1856-by-5 array:

- The height of each image is 64 pixels.
- The width of each image is 1856 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.

```
imagesFolder = fullfile(outputFolder, 'images');
helperTransformOrganizedPointCloudToTrainingData(lidarData, imagesFolder);
```

Preprocessing data 100% complete

The five-channel images are saved as MAT files.

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

Create imageDatastore and pixelLabelDatastore

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

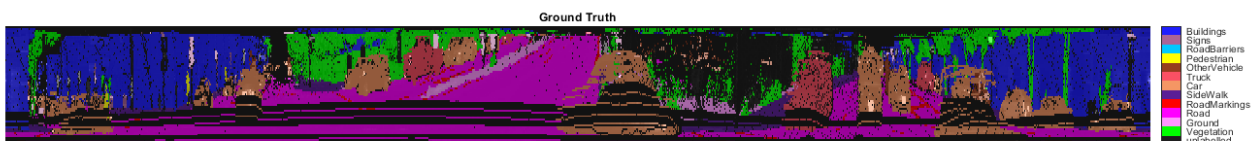
```
imds = imageDatastore(imagesFolder, ...
    'FileExtensions', '.mat', ...
    'ReadFcn', @helperImageMatReader);
```

Create a pixel label datastore using `pixelLabelDatastore` to store pixel-wise labels from the pixel label images. The object maps each pixel label to a class name. In this example, the vegetation, ground, road, road markings, sidewalk, cars, trucks, other vehicles, pedestrian, road barrier, signs, and buildings are the objects of interest; all other pixels are the background. Specify these classes and assign a unique label ID to each class.

```
classNames = ["unlabelled"
    "Vegetation"
    "Ground"
    "Road"
    "RoadMarkings"
    "SideWalk"
    "Car"
    "Truck"
    "OtherVehicle"
    "Pedestrian"
    "RoadBarriers"
    "Signs"
    "Buildings"];
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 `helperDisplayLidarOverlaidImage` function, defined in the Supporting Functions on page 1-0 section of this example.

```
% Point cloud (channels 1, 2, and 3 are for location, channel 4 is for intensity, and channel 5 :
I = read(imds);
labelMap = read(pxds);
figure;
helperDisplayLidarOverlaidImage(I, labelMap{1,1}, classNames);
title('Ground Truth');
```



Prepare Training, Validation, and Test Sets

Use the `helperPartitionLidarSegmentationDataset` supporting function, attached to this example, to split the data into training, validation, and test sets. You can split the training data

according to the percentage specified by the `trainingDataPercentage`. Divide the rest of the data in a 2:1 ratio into validation and testing data. Default value of `trainingDataPercentage` is 0.7.

```
[imdsTrain, imdsVal, imdsTest, pxdsTrain, pxdsVal, pxdsTest] = ...  
helperPartitionLidarSegmentationDataset(imds, pxds, 'trainingDataPercentage', 0.75);
```

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

```
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 `helperAugmentData` 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) helperAugmentData(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 1856 5];  
lgraph = squeezesegv2Layers(inputSize, ...  
numClasses, 'NumEncoderModules', 4, 'NumContextAggregationModules', 2);
```

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

```
analyzeNetwork(lgraph);
```

Specify Training Options

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

```
maxEpochs = 30;  
initialLearningRate = 1e-3;  
miniBatchSize = 8;  
l2reg = 2e-4;  
options = trainingOptions('adam', ...  
    'InitialLearnRate', initialLearningRate, ...  
    'L2Regularization', l2reg, ...
```

```

'MaxEpochs', maxEpochs, ...
'MiniBatchSize', miniBatchSize, ...
'LearnRateSchedule', 'piecewise', ...
'LearnRateDropFactor', 0.1, ...
'LearnRateDropPeriod', 10, ...
'ValidationData', validationData, ...
'Plots', 'training-progress', ...
'VerboseFrequency', 20);

```

Note: Reduce the miniBatchSize value to control memory usage when training.

Train 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 supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Otherwise, load a pretrained network.

```

if doTraining
    [net, info] = trainNetwork(trainingData, lgraph, options);
else
    load(fullfile(outputFolder, 'trainedSqueezeSegV2PandasetNet.mat'), 'net');
end

```

Predict Results on Test Point Cloud

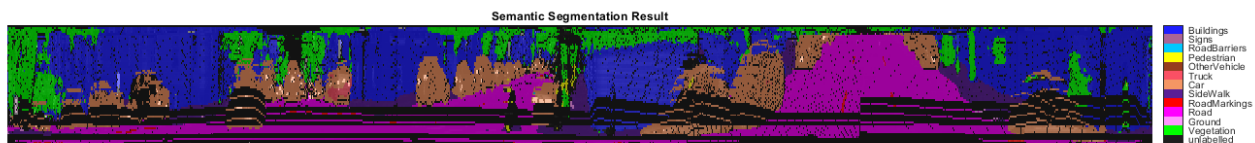
Use the trained network to predict results on a test point cloud and display the segmentation result. First, read a five-channel input image and predict the labels using the trained network.

Display the figure with the segmentation as an overlay.

```

I = read(imdsTest);
predictedResult = semanticseg(I, net);
figure;
helperDisplayLidarOverlaidImage(I, predictedResult, classNames);
title('Semantic Segmentation Result');

```

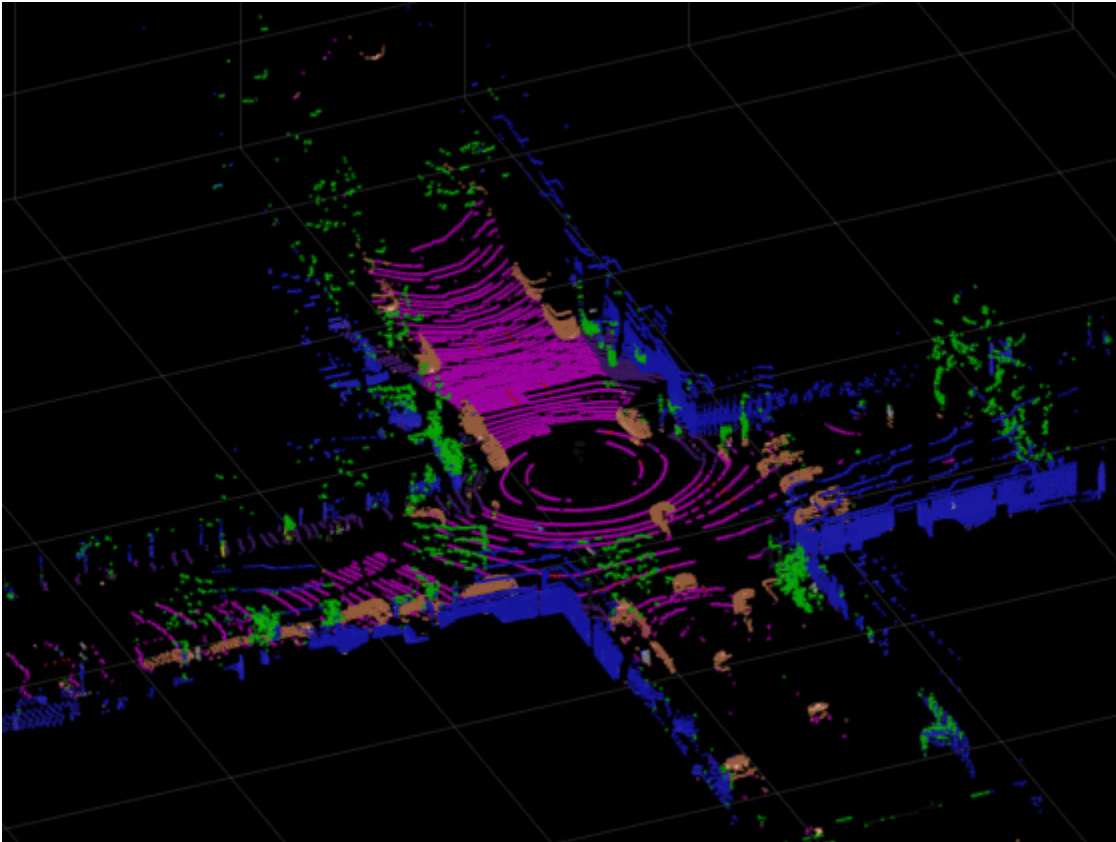


Use the helperDisplayLabelOverlaidPointCloud function, defined in the Supporting Functions on page 1-0 section of this example, to display the segmentation result on the point cloud.

```

figure;
helperDisplayLabelOverlaidPointCloud(I, predictedResult);
view([39.2 90.0 60]);
title('Semantic Segmentation Result on Point Cloud');

```



Evaluate Network

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

```
outputLocation = fullfile(tempdir, 'output');
if ~exist(outputLocation, 'dir')
    mkdir(outputLocation);
end
pxdsResults = semanticseg(imdsTest, net, ...
    'MiniBatchSize', 4, ...
    'WriteLocation', outputLocation, ...
    'Verbose', false);
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=1x5 table
   GlobalAccuracy   MeanAccuracy   MeanIoU   WeightedIoU   MeanBFScore
```

0.89724 0.61685 0.54431 0.81806 0.74537

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=13x3 table`

	Accuracy	IoU	MeanBFScore
unlabelled	0.94	0.9005	0.99911
Vegetation	0.77873	0.64819	0.95466
Ground	0.69019	0.59089	0.60657
Road	0.94045	0.83663	0.99084
RoadMarkings	0.37802	0.34149	0.77073
Sidewalk	0.7874	0.65668	0.93687
Car	0.9334	0.81065	0.95448
Truck	0.30352	0.27401	0.37273
OtherVehicle	0.64397	0.58108	0.47253
Pedestrian	0.26214	0.20896	0.45918
RoadBarriers	0.23955	0.21971	0.19433
Signs	0.17276	0.15613	0.44275
Buildings	0.94891	0.85117	0.96929

Although the overall network performance is good, the class metrics for some classes like RoadMarkings and Truck indicate that more training data is required for better performance.

Supporting Functions

Function to Augment Data

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

```
function out = helperAugmentData(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 `helperDisplayLidarOverlaidImage` 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 helperDisplayLidarOverlaidImage(lidarImage, labelMap, classNames)
% helperDisplayLidarOverlaidImage Overlay labels over the intensity image.
```



```
%
% helperDisplayLidarOverlaidImage(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 = helperPandasetColorMap;
% 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 `helperDisplayLabelOverlaidPointCloud` function overlays the segmentation result over a 3-D organized point cloud.

```
function helperDisplayLabelOverlaidPointCloud(I,predictedResult)
% helperDisplayLabelOverlaidPointCloud Overlay labels over point cloud object.
% helperDisplayLabelOverlaidPointCloud(I, predictedResult)
% displays the overlaid pointCloud object. I is the 5 channels organized
% input image. predictedResult contains pixel labels.
ptCloud = pointCloud(I(:,:,1:3),'Intensity',I(:,:,4));
cmap = helperPandasetColorMap;
B = ...
labeloverlay(uint8(ptCloud.Intensity),predictedResult,'Colormap',cmap,'Transparency',0.4);
pc = pointCloud(ptCloud.Location,'Color',B);
figure;
ax = pcshow(pc);
set(ax,'XLim',[-70 70],'YLim',[-70 70]);
zoom(ax,3.5);
end
```

Function to Define Lidar Colormap

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

```
function cmap = helperPandasetColorMap
cmap = [[30,30,30]; % Unlabeled
[0,255,0]; % Vegetation
[255, 150, 255]; % Ground
[255,0,255]; % Road
[255,0,0]; % Road Markings
[90, 30, 150]; % Sidewalk
[245,150,100]; % Car
[250, 80, 100]; % Truck
[150, 60, 30]; % Other Vehicle
[255, 255, 0]; % Pedestrian
[0, 200, 255]; % Road Barriers
[170,100,150]; % Signs
[30, 30, 255]]; % Building
cmap = cmap./255;
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
```

Function to Download Pretrained Model

The `downloadPretrainedSqueezeSegV2Net` function downloads the pretrained model.

```
function downloadPretrainedSqueezeSegV2Net(outputFolder, pretrainedNetURL)
    preTrainedMATFile = fullfile(outputFolder, 'trainedSqueezeSegV2PandasetNet.mat');
    preTrainedZipFile = fullfile(outputFolder, 'trainedSqueezeSegV2PandasetNet.zip');

    if ~exist(preTrainedMATFile, 'file')
        if ~exist(preTrainedZipFile, 'file')
            disp('Downloading pretrained model (5 MB)...');
            websave(preTrainedZipFile, pretrainedNetURL);
        end
        unzip(preTrainedZipFile, outputFolder);
    end
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>.

[2] Hesai and Scale. PandaSet. <https://scale.com/open-datasets/pandaset>

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 1-75. 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 1-30.

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
```

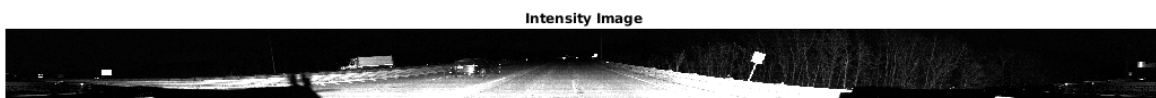
Name	Size	Bytes	Class	Attributes
I	64x1024x5	327680	uint8	

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.
carClassColor = zeros(64, 1024, 3, 'uint8');
carClassColor(:,:,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);

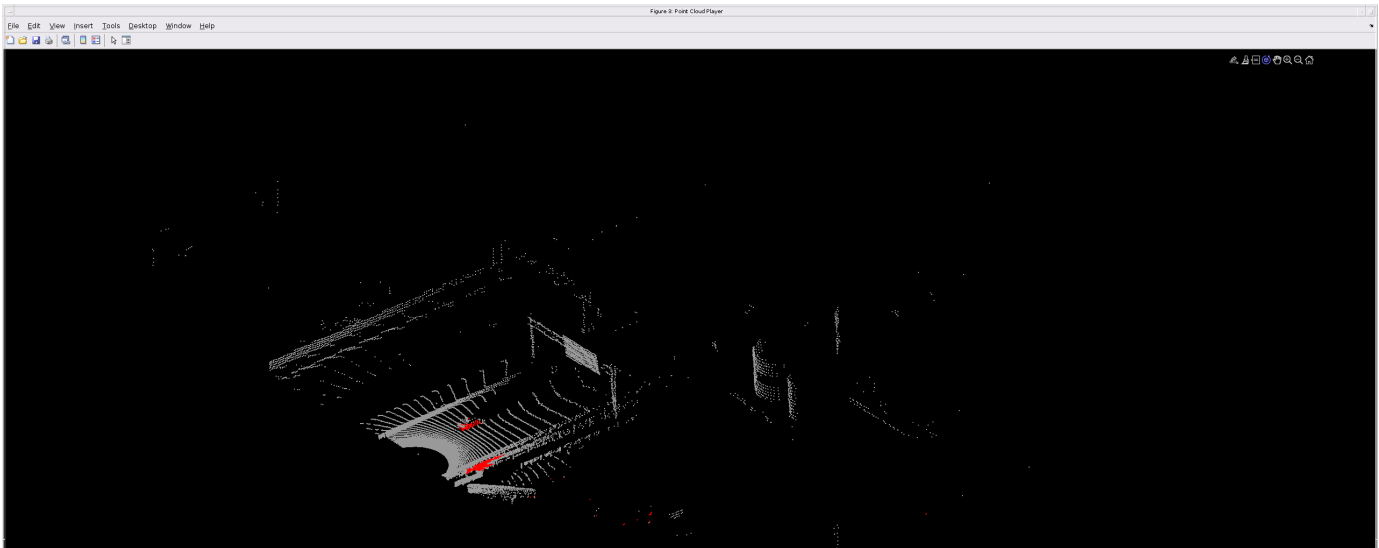
    % Extract the indices from labels.
    carIndices = predictedOutput == 'car';
    truckIndices = predictedOutput == 'truck';
    backgroundIndices = predictedOutput == 'background';

    % 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 = carClassColor;
    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.

Lidar point cloud data can be acquired by a variety of lidar sensors, including Velodyne®, Pandar, and Ouster sensors. 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 estimate joint 3-D bounding boxes, orientations, and class predictions.

This example uses the PandaSet [2 on page 1-0] data set from Hesai and Scale. PandaSet contains 8240 unorganized lidar point cloud scans of various city scenes captured using a Pandar64 sensor. The data set provides 3-D bounding box labels for 18 different object classes, including car, truck, and pedestrian.

Download Lidar Data Set

This example uses a subset of PandaSet that contains 2560 preprocessed organized point clouds. Each point cloud covers 360° of view, and is specified as a 64-by-1856 matrix. The point clouds are stored in PCD format and their corresponding ground truth data is stored in the PandaSetLidarGroundTruth.mat file. The file contains 3-D bounding box information for three classes, which are car, truck, and pedestrian. The size of the data set is 5.2 GB.

Download the Pandaset dataset from the given URL using the helperDownloadPandasetData helper function, defined at the end of this example.

```
outputFolder = fullfile(tempdir, 'Pandaset');
lidarURL = ['https://ssd.mathworks.com/supportfiles/lidar/data/' ...
    'Pandaset_LidarData.tar.gz'];
helperDownloadPandasetData(outputFolder, lidarURL);
```

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.

Download Pretrained Network

Download the pretrained network from the given URL using the helperDownloadPretrainedPointPillarsNet helper function, 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.

```
pretrainedNetURL = ['https://ssd.mathworks.com/supportfiles/lidar/data/' ...
    'trainedPointPillarsPandasetNet.zip'];

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

Load Data

Create a file datastore to load the PCD files from the specified path using the `pcread` function.

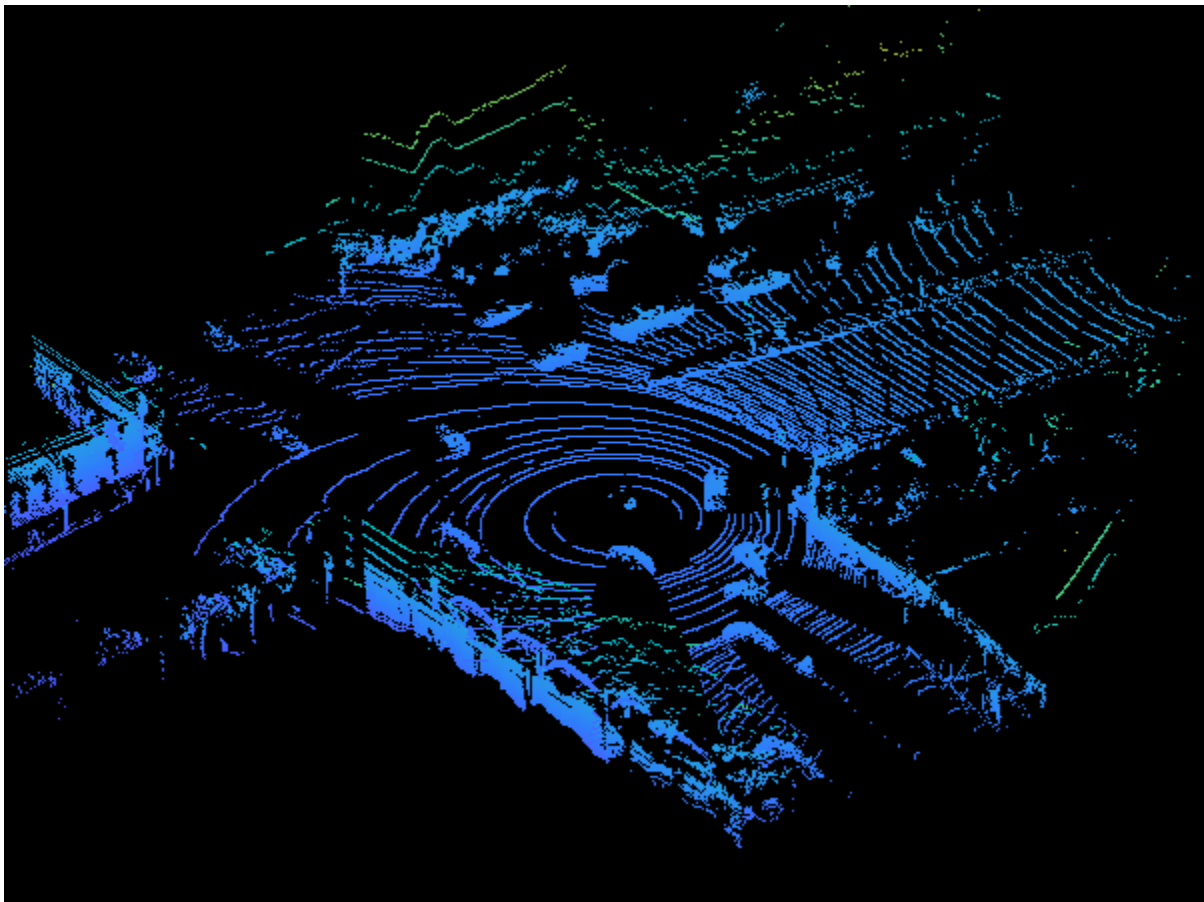
```
path = fullfile(outputFolder, 'Lidar');  
lidarData = fileDatastore(path, 'ReadFcn', @(x) pcread(x));
```

Load the 3-D bounding box labels of the car and truck objects.

```
gtPath = fullfile(outputFolder, 'Cuboids', 'PandaSetLidarGroundTruth.mat');  
data = load(gtPath, 'lidarGtLabels');  
Labels = timetable2table(data.lidarGtLabels);  
boxLabels = Labels(:, 2:3);
```

Display the full-view point cloud.

```
figure  
ptCld = read(lidarData);  
ax = pcshow(ptCld.Location);  
set(ax, 'XLim', [-50 50], 'YLim', [-40 40]);  
zoom(ax, 2.5);  
axis off;
```



```
reset(lidarData);
```

Preprocess Data

The PandaSet data consists of full-view point clouds. For this example, crop the full-view point clouds to front-view point clouds using the standard parameters [1 on page 1-0]. These parameters determine the size of the input passed to the network. Selecting a smaller range of point clouds along the 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 the pseudo-image.
Xn = round((xMax - xMin) / xStep);
Yn = round((yMax - yMin) / yStep);

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

Use the `cropFrontViewFromLidarData` helper function, attached to this example as a supporting file, to:

- Crop the front view from the input full-view point cloud.
- Select the box labels that are inside the ROI specified by `gridParams`.

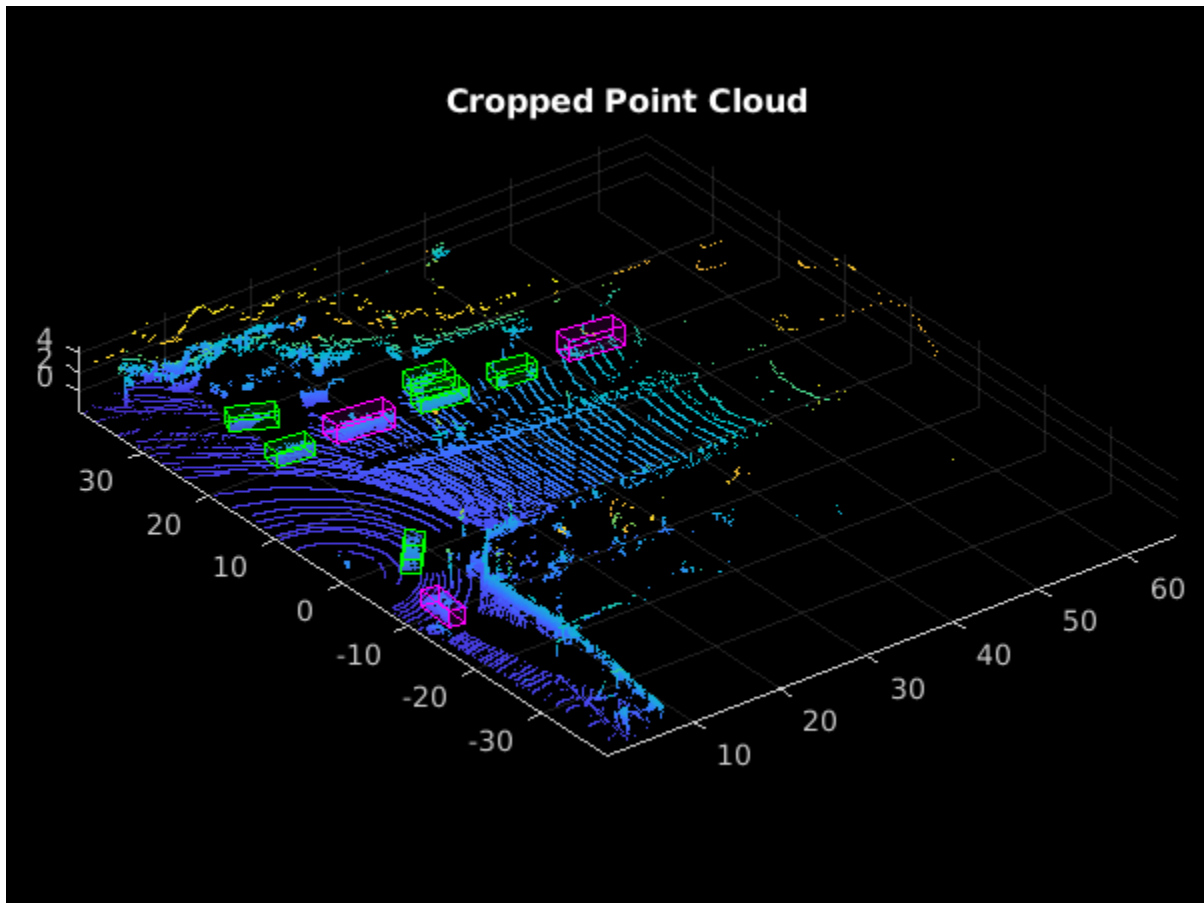
```
[croppedPointCloudObj,processedLabels] = cropFrontViewFromLidarData(...
    lidarData,boxLabels,gridParams);
```

Processing data 100% complete

Display the cropped point cloud and the ground truth box labels using the `helperDisplay3DBoxesOverlaidPointCloud` helper function defined at the end of the example.

```
pc = croppedPointCloudObj{1,1};
gtLabelsCar = processedLabels.Car{1};
gtLabelsTruck = processedLabels.Truck{1};

helperDisplay3DBoxesOverlaidPointCloud(pc.Location,gtLabelsCar,...
    'green',gtLabelsTruck,'magenta','Cropped Point Cloud');
```



```
reset(lidarData);
```

Create Datastore Objects for Training

Split the data set into training and test sets. Select 70% of the data for training the network and the rest for evaluation.

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

trainData = croppedPointCloudObj(shuffledIndices(1:idx),:);
testData = croppedPointCloudObj(shuffledIndices(idx+1:end),:);

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

So that you can easily access the datastores, save the training data as PCD files by using the `saveptCldToPCD` helper function, attached to this example as a supporting file. You can set `writeFiles` to "false" if your training data is saved in a folder and is supported by the `pcread` function.

```
writeFiles = true;
dataLocation = fullfile(outputFolder,'InputData');
[trainData,trainLabels] = saveptCldToPCD(trainData,trainLabels,...
    dataLocation,writeFiles);
```

Processing data 100% complete

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

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

Create a box label datastore using `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

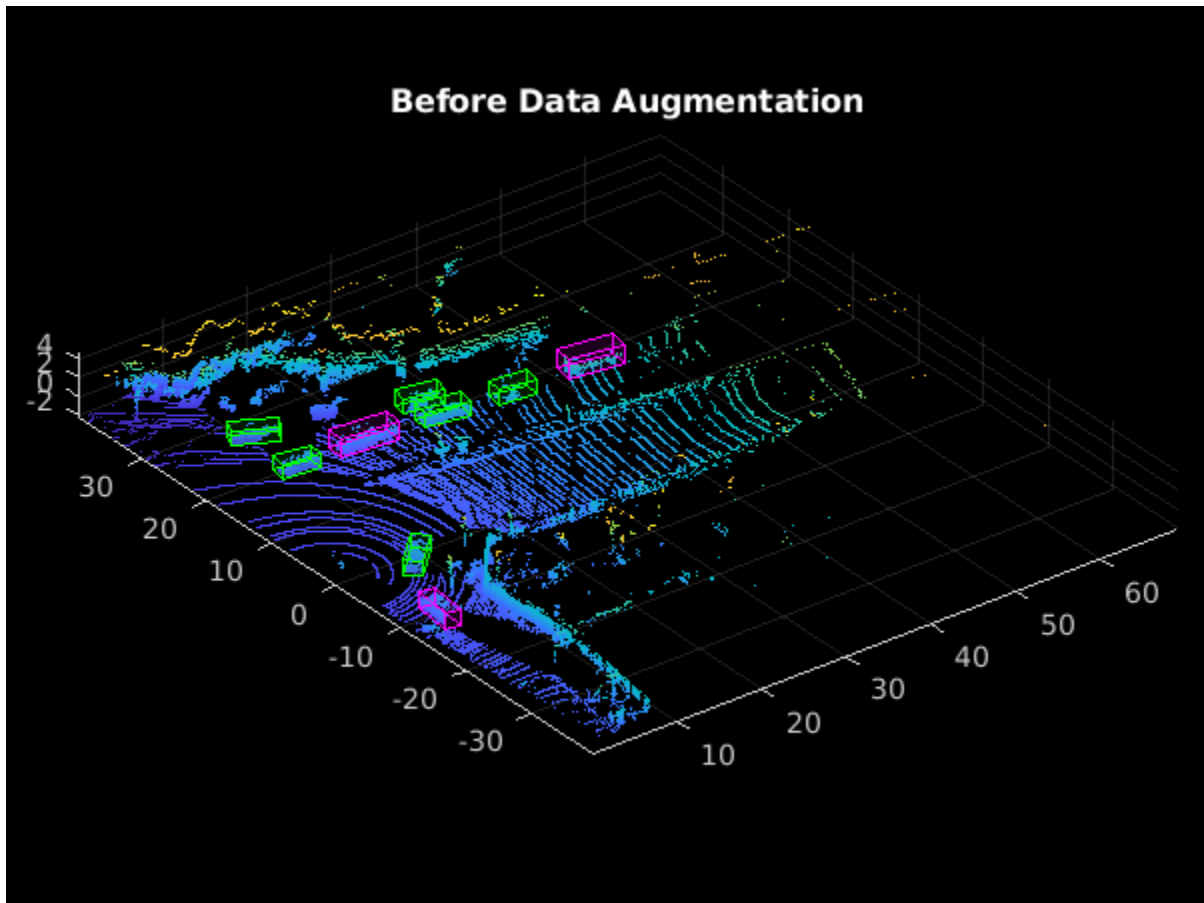
This example uses ground truth data augmentation and several other global data augmentation techniques to add more variety to the training data and corresponding boxes. For more information on typical data augmentation techniques used in 3-D object detection workflows with lidar data, see “Data Augmentations for Lidar Object Detection Using Deep Learning” on page 1-163.

Read and display a point cloud before augmentation using the `helperDisplay3DBoxesOverlaidPointCloud` helper function, defined at the end of the example..

```
augData = read(cds);
augptCld = augData{1,1};
augLabels = augData{1,2};
augClass = augData{1,3};
```

```
labelsCar = augLabels(augClass=='Car',:);
labelsTruck = augLabels(augClass=='Truck',:);
```

```
helperDisplay3DBoxesOverlaidPointCloud(augptCld.Location, labelsCar, 'green', ...
    labelsTruck, 'magenta', 'Before Data Augmentation');
```



```
reset(cds);
```

Use the `generateGTDataForAugmentation` helper function, attached to this example as a supporting file, to extract all the ground truth bounding boxes from the 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 and truck class objects to every point cloud. Use the `transform` function to apply the ground truth and custom data augmentations to the training data.

```
samplesToAdd = struct('Car',10,'Truck',10);
cdsAugmented = transform(cds,@(x) groundTruthDataAugmentation(x,gtData,samplesToAdd));
```

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 the z-axis from $[-\pi/4, \pi/4]$
- Random translation by $[0.2, 0.2, 0.1]$ meters along the x-, y-, and z-axis respectively

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

Display an augmented point cloud along with ground truth augmented boxes using the `helperDisplay3DBoxesOverlaidPointCloud` helper function, defined at the end of the example.

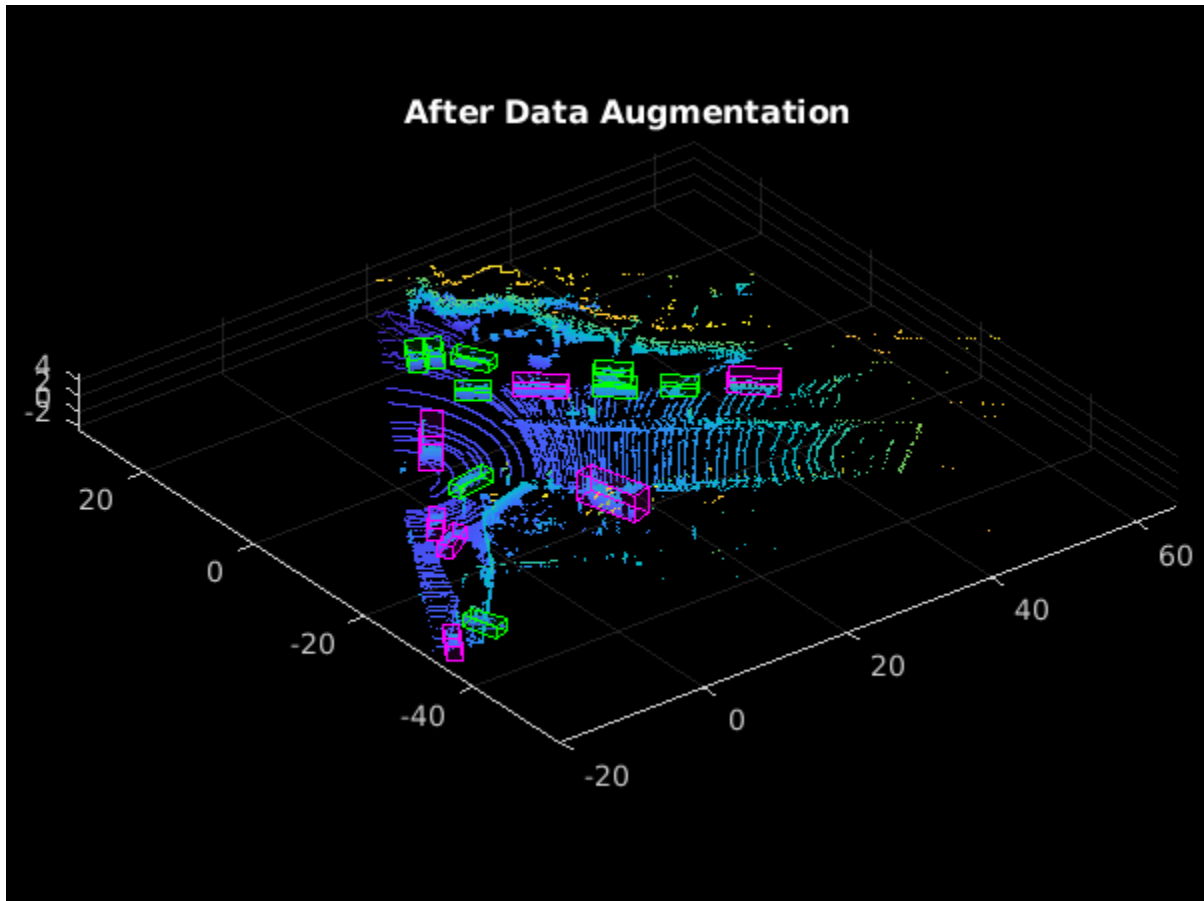
```

augData = read(cdsAugmented);
augptCld = augData{1,1};
augLabels = augData{1,2};
augClass = augData{1,3};

labelsCar = augLabels(augClass=='Car',:);
labelsTruck = augLabels(augClass=='Truck',:);

helperDisplay3DBoxesOverlaidPointCloud(augptCld(:,1:3),labelsCar,'green',...
    labelsTruck,'magenta','After Data Augmentation');

```



```
reset(cdsAugmented);
```

Extract Pillar Information

You can apply a 2-D convolution architecture to the point clouds for faster processing. To do so, first convert the 3-D point clouds to 2-D representation. Use the `transform` function with the `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 of vertical columns called pillars.
- Select prominent pillars (P) based on the number of points per pillar (N).

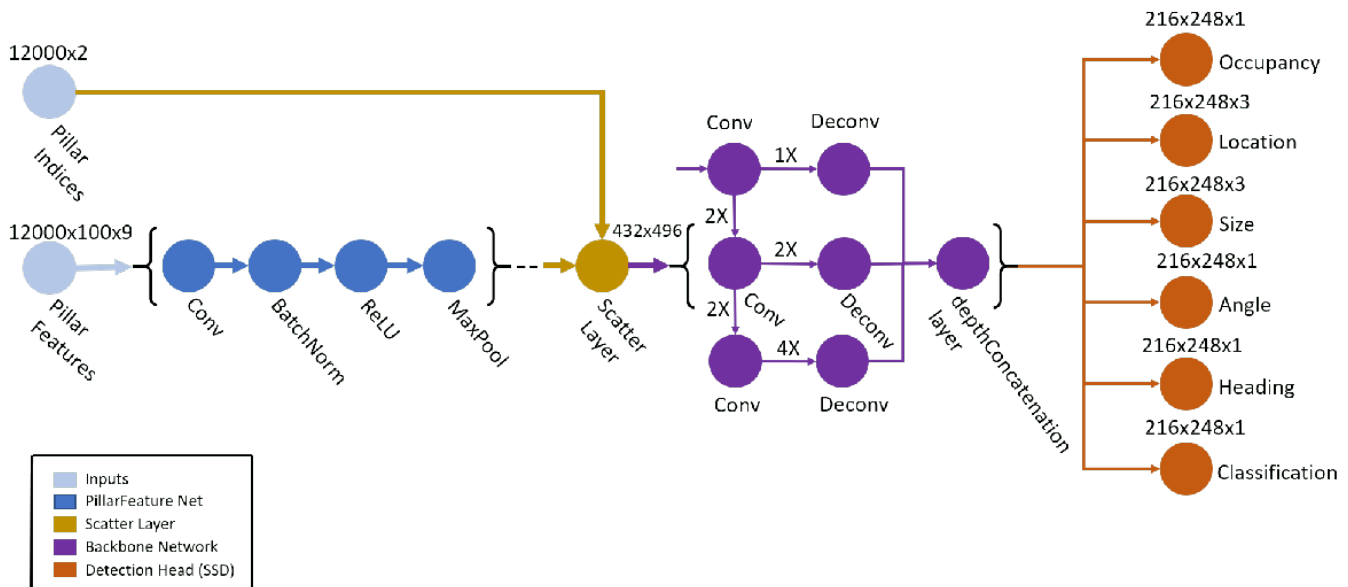
- Compute the distance to the arithmetic mean of all points in the pillar.
- Compute the offset from the pillar center.
- Use the x, y, z location, intensity, distance, and offset values to create a nine dimensional (9-D) vector for each point in the pillar.

```
% 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 Network

The PointPillars network uses a simplified version of the PointNet network that takes pillar features as input. For each pillar feature, the network applies a linear layer, followed by batch normalization and ReLU layers. Finally, the network applies a max-pooling operation over the channels to get high-level encoded features. These encoded features are scattered back to the original pillar locations to create a pseudo-image using the custom layer `helperscatterLayer`, attached to this example as a supporting file. The network then processes the pseudo-image 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]. Alternatively, you can also use the `calculateAnchorBoxes` helper function, attached to the example, for obtaining the appropriate anchor boxes from any training set. The anchor boxes are defined in the format $\{length, width, height, z-center, yaw\ angle\}$.

```
anchorBoxes = calculateAnchorsPointPillars(trainLabels);
numAnchors = size(anchorBoxes,2);
classNames = trainLabels.Properties.VariableNames;
numClasses = numel(classNames);
```


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

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

Specify Training Options

Specify the following training options.

- Set the number of epochs to 60.
- Set the mini-batch size to 2. You can set the mini-batch size to a higher value if you have more available memory.
- Set the learning rate to 0.0002.
- Set `learnRateDropPeriod` to 15. This parameter denotes the number of epochs after which to drop the learning rate based on the formula $\text{learningRate} \times (\text{iteration} \% \text{learnRateDropPeriod}) \times \text{learnRateDropFactor}$.
- 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 = 60;
miniBatchSize = 2;
learningRate = 0.0002;
learnRateDropPeriod = 15;
learnRateDropFactor = 0.8;
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.999;
trailingAvg = [];
trailingAvgSq = [];
```

Train Model

Train the network using a CPU or GPU. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). 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) ...
helperCreateBatchData(features,indices,boxes,labels,classNames),...
"MiniBatchFormat",["SSCB","SSCB",""],...
"DispatchInBackground",true);
```

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 `helperConfigureTrainingProgressPlotter` helper function, 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) object 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 = helperConfigureTrainingProgressPlotter(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 function.
            [gradients,loss,state] = dlfeval(@modelGradients,net,...
                pillarFeatures,pillarIndices,boxLabels,gridParams,...
                anchorBoxes,numClasses,executionEnvironment);

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

```

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, iteration, learningRate, ...
        gradientDecayFactor, squaredGradientDecayFactor);

% 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
else
    preTrainedMATFile = fullfile(outputFolder, 'trainedPointPillarsPandasetNet.mat');
    pretrainedNetwork = load(preTrainedMATFile, 'net');
    net = pretrainedNetwork.net;
end
end

```

Generate Detections

Use the trained network to detect objects in the test data:

- 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 using the `helperDisplay3DBoxesOverlaidPointCloud` helper function, defined at the end of the example.

```

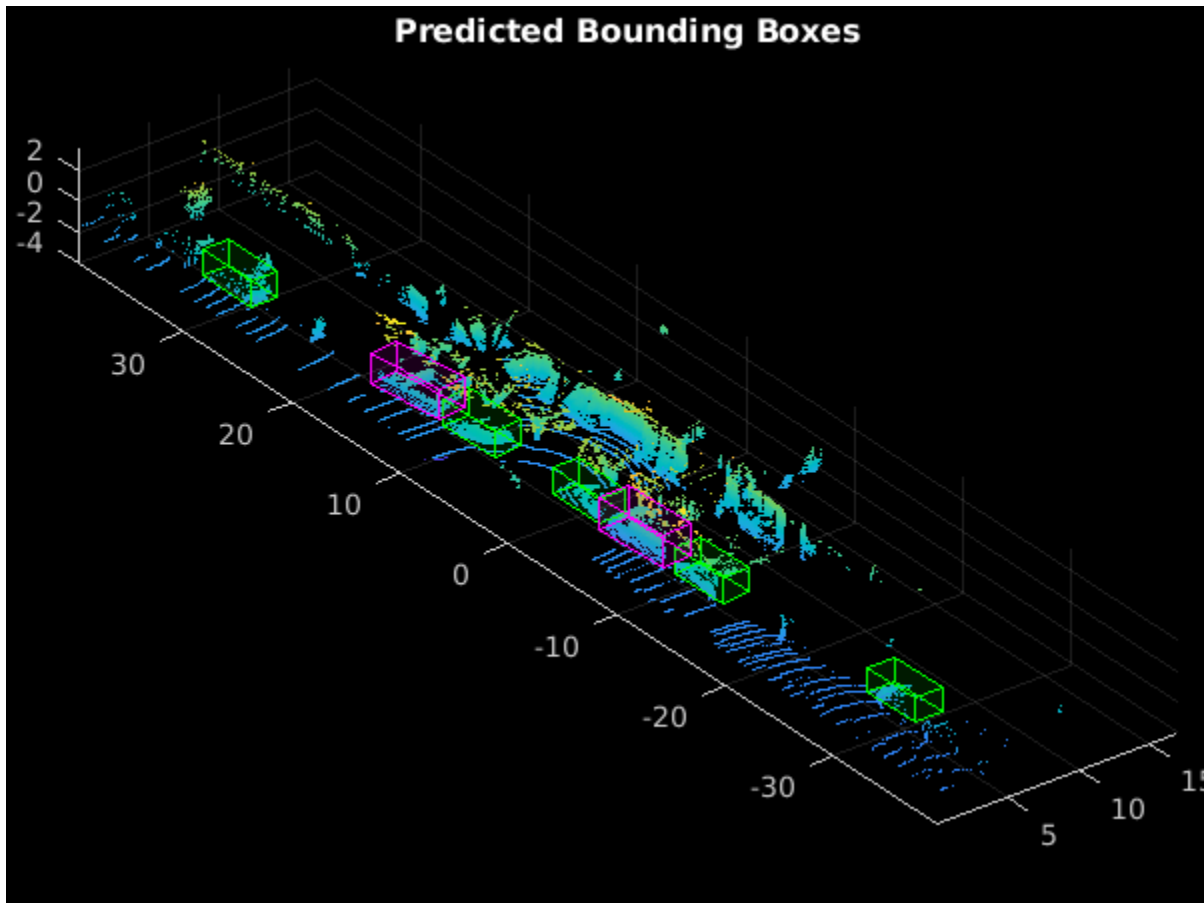
ptCloud = testData{45,1};
gtLabels = testLabels(45,:);

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

boxlabelsCar = box(labels=='Car',:);
boxlabelsTruck = box(labels=='Truck',:);

% Display the predictions on the point cloud.
helperDisplay3DBoxesOverlaidPointCloud(ptCloud.Location, boxlabelsCar, 'green', ...
    boxlabelsTruck, 'magenta', 'Predicted Bounding Boxes');

```



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,classNames,confidenceThreshold,overlapThreshold,...
        P,N,executionEnvironment);

    % Convert the detected boxes to rotated rectangle 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);
disp(metrics(:,1:2))

```

	AOS	AP
Car	0.86746	0.86746
Truck	0.61463	0.61463

Helper Functions

Model Gradients

The function `modelGradients` takes as input the `dlnetwork` (Deep Learning Toolbox) object `net` and a mini-batch of input data `pillarFeatures` 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,gridParams,anchorBoxes,...
    numClasses,executionEnvironment)

    numAnchors = size(anchorBoxes,2);

    % 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,anchorBoxes,numClasses);
    YTargets = cellfun(@ dlarray,YTargets,'UniformOutput',false);
    if (executionEnvironment=="auto" && canUseGPU) || executionEnvironment=="gpu"
        YTargets = cellfun(@ gpuArray,YTargets,'UniformOutput',false);
    end

    [angLoss,occlLoss,locLoss,szLoss,hdLoss,clfLoss] = ...
        computePointPillarLoss(YPredictions,YTargets,gridParams,...
            numClasses,numAnchors);

    % Compute the total loss.
    loss = angLoss + occlLoss + 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] = helperCreateBatchData(...
    features,indices,groundTruthBoxes,groundTruthClasses,classNames)
% Return 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,'UniformOutput',false);

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

end

```
function helperDownloadPandasetData(outputFolder,lidarURL)
% Download the data set from the given URL to the output folder.
```

```

lidarDataTarFile = fullfile(outputFolder, 'Pandaset_LidarData.tar.gz');

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

    disp('Downloading PandaSet Lidar driving data (5.2 GB)...');
    websave(lidarDataTarFile, lidarURL);
    untar(lidarDataTarFile, outputFolder);
end

% Extract the file.
if (~exist(fullfile(outputFolder, 'Lidar'), 'dir'))...
    && (~exist(fullfile(outputFolder, 'Cuboids'), 'dir'))
    untar(lidarDataTarFile, outputFolder);
end

end

function helperDownloadPretrainedPointPillarsNet(outputFolder, pretrainedNetURL)
% Download the pretrained PointPillars network.

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

if ~exist(preTrainedMATFile, 'file')
    if ~exist(preTrainedZipFile, 'file')
        disp('Downloading pretrained detector (8.4 MB)...');
        websave(preTrainedZipFile, pretrainedNetURL);
    end
    unzip(preTrainedZipFile, outputFolder);
end

end

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

function retVal = helperCheckForNaN(gradients, loss)
% The last convolution head 'occupancy|conv2d' is known to contain NaNs
% 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 == 'Bias'));
if (sum(isnan(extractdata(loss)), 'all') > 0) || ...
    (sum(isnan(extractdata(gradValue{1,1})), 'all') > 0))
    retVal = true;
else
    retVal = false;
end

end

function helperDisplay3DBoxesOverlaidPointCloud(ptCld, labelsCar, carColor, ...

```

```
    labelsTruck, truckColor, titleForFigure)
% Display the point cloud with different colored bounding boxes for different
% classes.
    figure;
    ax = pcshow(ptCld);
    showShape('cuboid', labelsCar, 'Parent', ax, 'Opacity', 0.1, ...
        'Color', carColor, 'LineWidth', 0.5);
    hold on;
    showShape('cuboid', labelsTruck, 'Parent', ax, 'Opacity', 0.1, ...
        'Color', truckColor, 'LineWidth', 0.5);
    title(titleForFigure);
    zoom(ax, 1.5);
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>.
- [2] Hesai and Scale. PandaSet. <https://scale.com/open-datasets/pandaset>.

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);
```

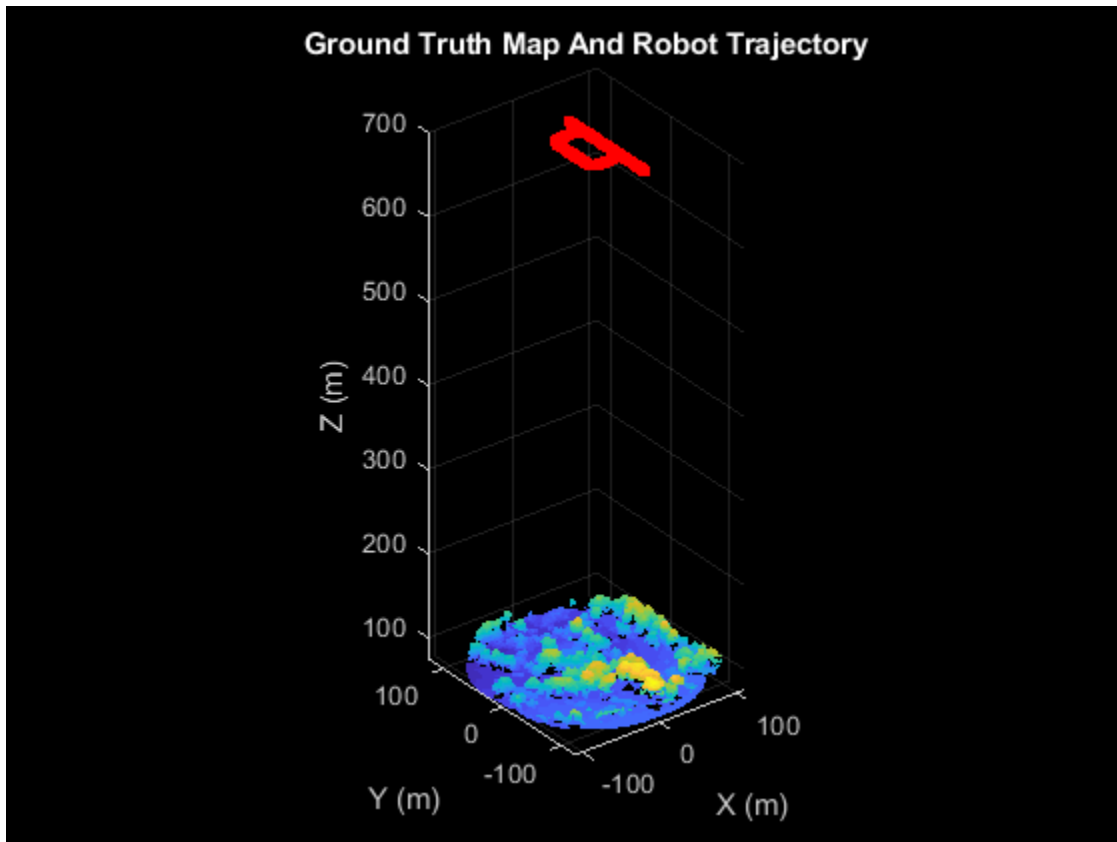
Visualize the ground truth waypoints on the original map covered by the robot.

```
% 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
xlims = [-90 90];
ylims = [-90 90];
zlims = [-625 -587];

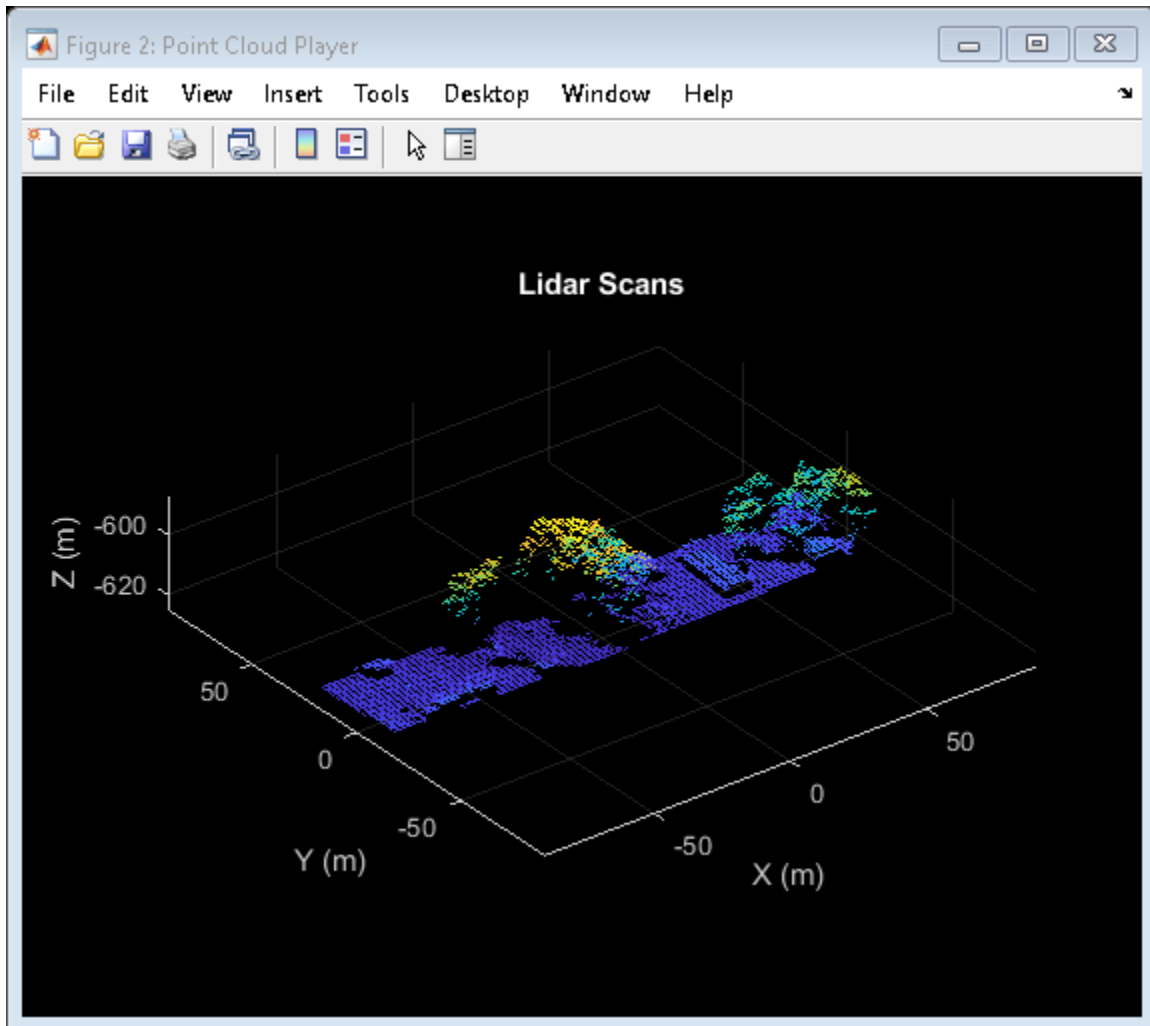
% Create a pcplayer object to visualize the lidar scans
lidarPlayer = pcplayer(xlims,ylims,zlims);

% 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:

- 1 Point cloud downsampling
- 2 Point cloud registration
- 3 Submap creation
- 4 Loop closure querying
- 5 Pose graph optimization

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

```

rng('default') % Set random seed to guarantee consistent results in pmatchfeatures
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 = pcdsample(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:

- 1 Extracts FPFH descriptors from each scan using the `extractFPFHFeatures` function
- 2 Identifies point correspondences by comparing descriptors using the `pcmatchfeatures` function
- 3 Estimates the relative pose from point correspondences using the `estimateGeometricTransform3D` function
- 4 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', maxNumTrials);

    % 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 `pcaalign` 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} = pcaalign(...
        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:

- 1 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.
- 2 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] = pregistericp(curScan, ...
                pAccepted(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

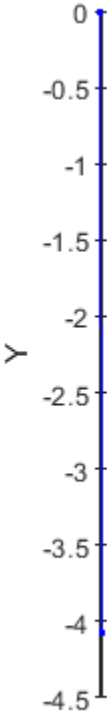
    % Check if loop closure candidate is valid
    if rmseMin < rmseThreshold

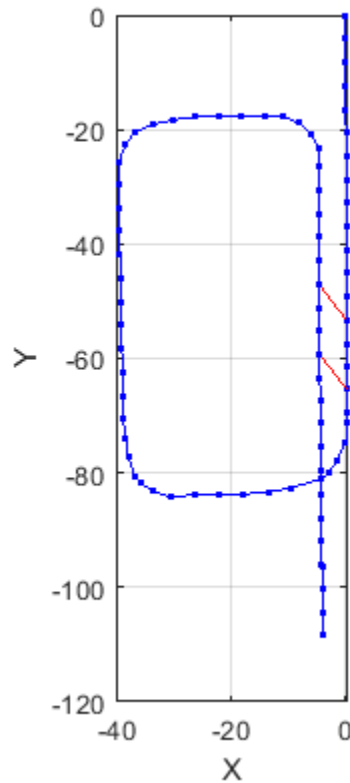
        % Add relative pose of loop closure candidate to pose graph
        addRelativePose(pGraph,Pose,infoMat,matchingNode, ...
            pGraph.NumNodes);
    end
end

% 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

```



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;

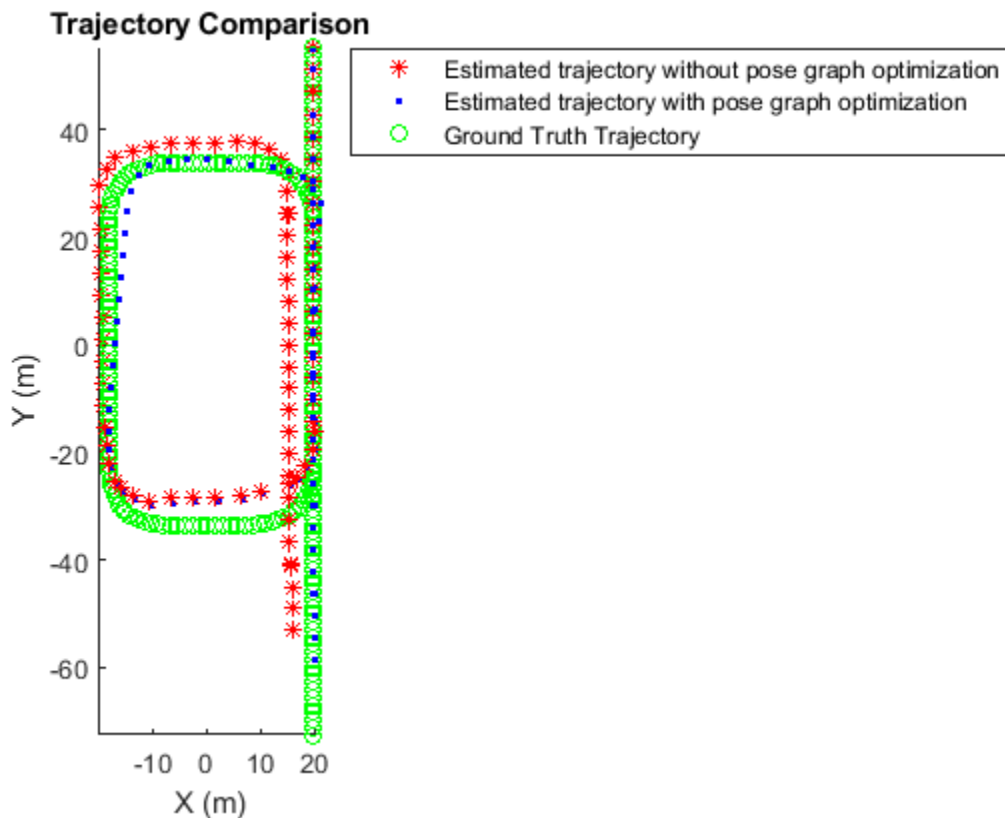
% Get estimated trajectory from feature-based registration without pose
% graph optimization
fpfhEstimatedTraj = zeros(count,3);
for i = 1:count
    fpfhEstimatedTraj(i,:) = fpfhTform(i).Translation;
end

% 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);

```

```

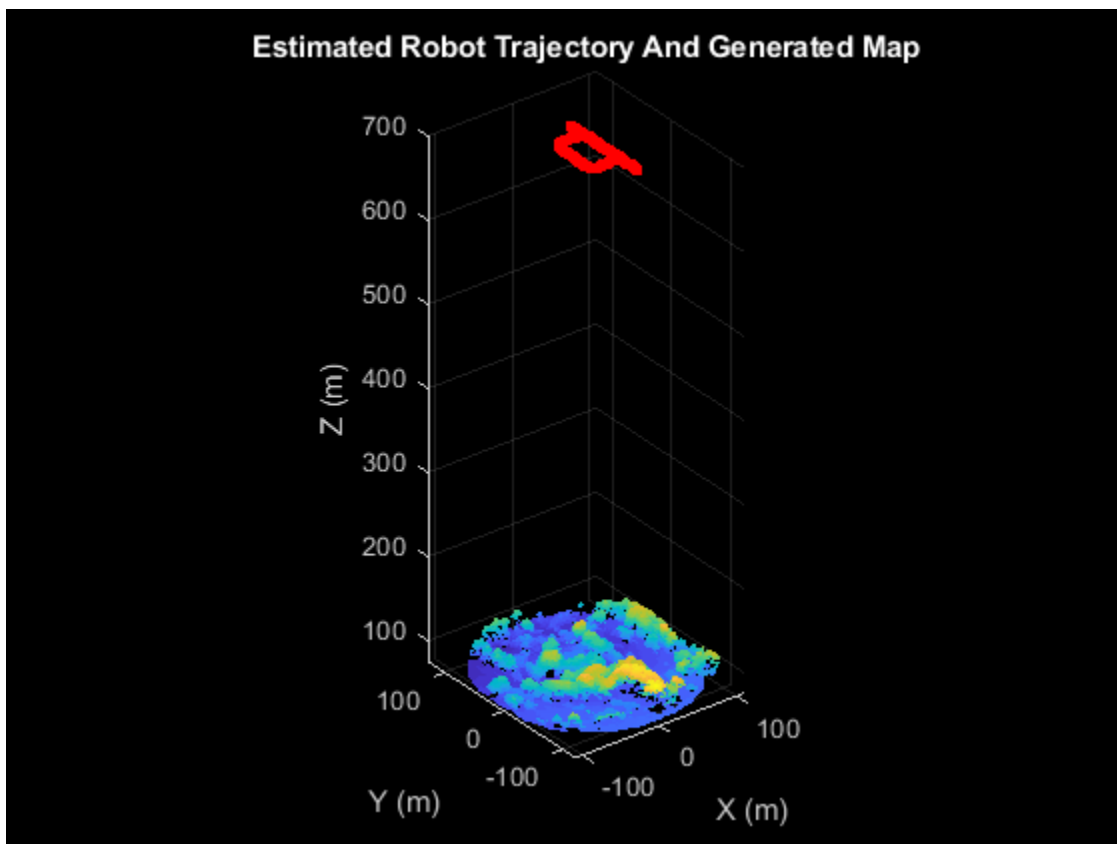
    rigidPose = rigid3d((trvec2tform(pose(1:3)) * quat2tform(pose(4:7))));
    estimatedTforms(idx,1) = rigidPose;
end

% 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 = [0.08 0.2 0.4 0.55];
pcshow(orgMap,'Parent',axMap1)

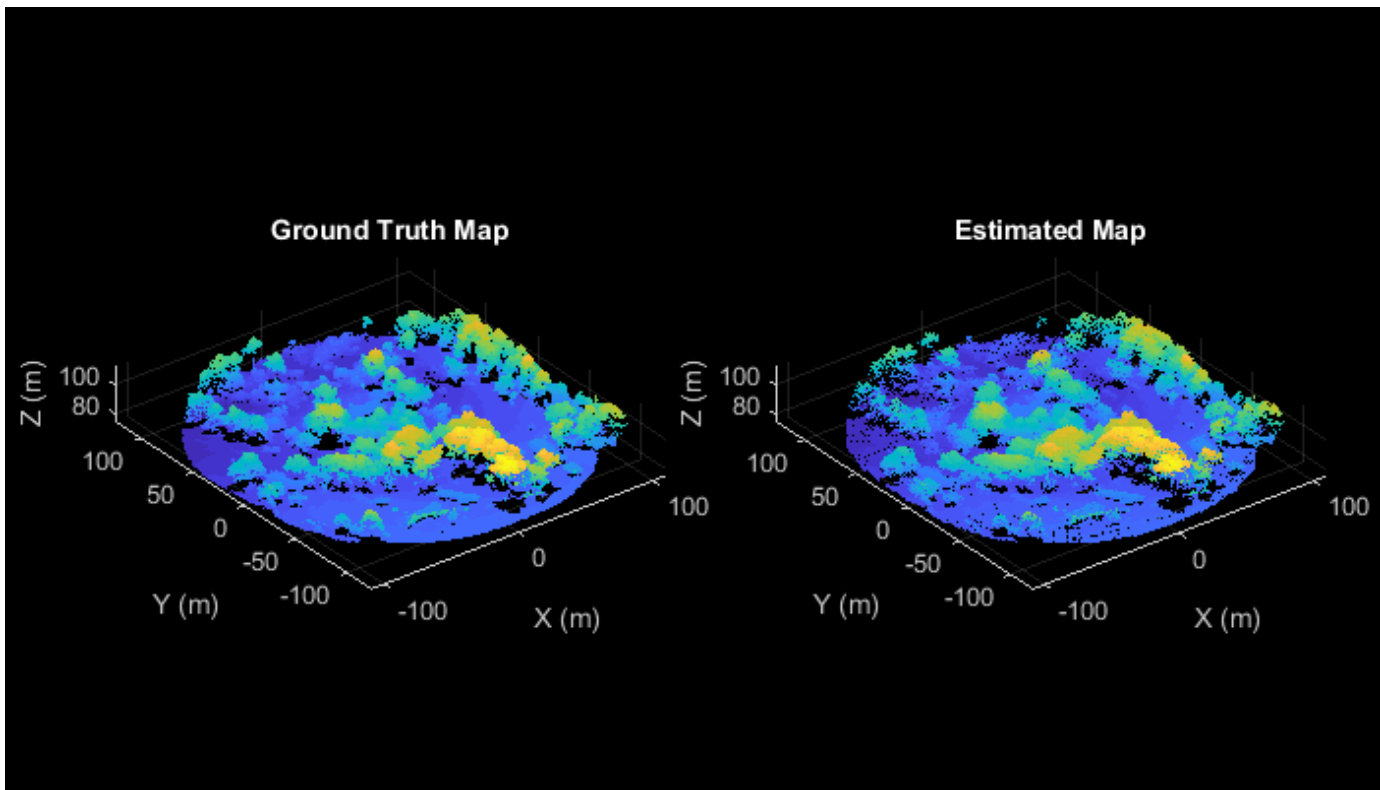
```

```

xlabel(axMap1,'X (m)')
ylabel(axMap1,'Y (m)')
zlabel(axMap1,'Z (m)')
title(axMap1,'Ground Truth Map')

axMap2 = subplot(1,2,2,'Color','black','Parent',hFigMap);
axMap2.Position = [0.56 0.2 0.4 0.55];
pcshow(globalMap,'Parent',axMap2)
xlabel(axMap2,'X (m)')
ylabel(axMap2,'Y (m)')
zlabel(axMap2,'Z (m)')
title(axMap2,'Estimated Map')

```



See Also

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)

Objects

[lasFileReader](#) | [pointCloud](#) | [pcplayer](#) | [occupancyMap3D](#) (Navigation Toolbox) | [poseGraph3D](#) (Navigation Toolbox) | [rigid3d](#)

References

[1] Starr, Scott. "Tuscaloosa, AL: Seasonal Inundation Dynamics and Invertebrate Communities." National Center for Airborne Laser Mapping, December 1, 2011. OpenTopography(<https://doi.org/10.5069/G9SF2T3K>).

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

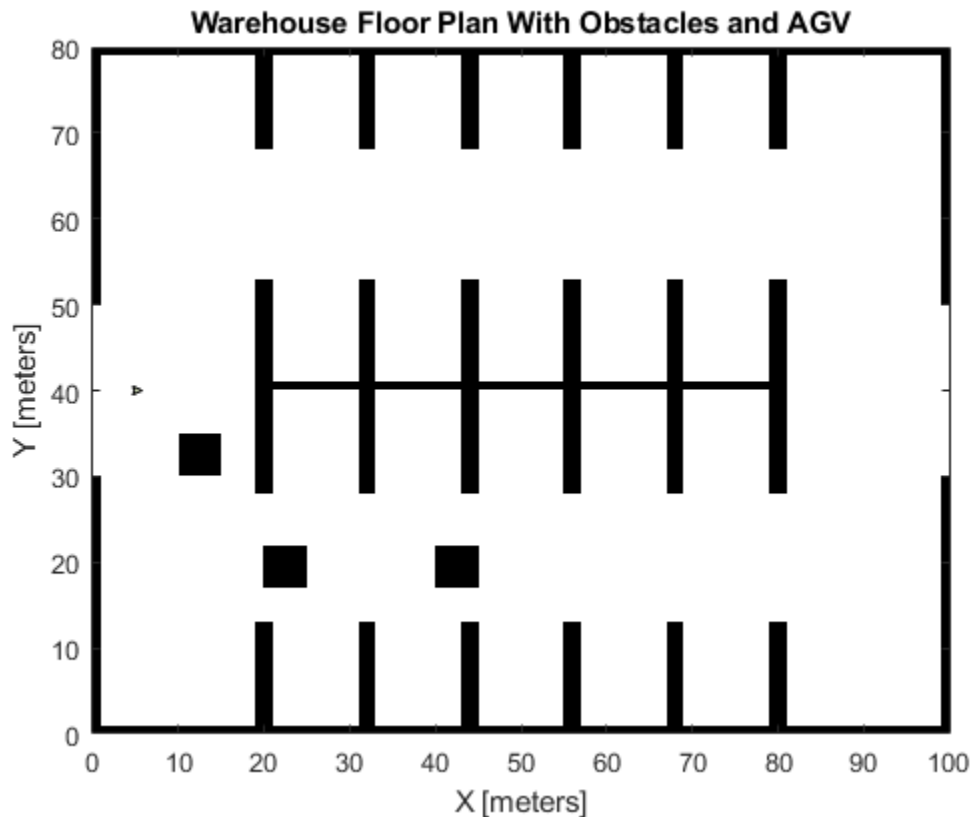
To represent the environment of the robot workspace, define a `binaryOccupancyMap` (Navigation Toolbox) object that contains the floor plan of the warehouse. Each cell in the occupancy grid has a logical value. An occupied location is represented as 1 and a free location is represented as 0. You can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

Add obstacles to the map near to the defined route of AGV.

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

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

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



Simulate 2-D Lidar Sensor

Simulate 2-D lidar sensor using a `rangeSensor` 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 and set the detection angles to [-pi/2 pi/2]
lidar = rangeSensor;
lidar.HorizontalAngle = [-pi/2 pi/2];
% 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;

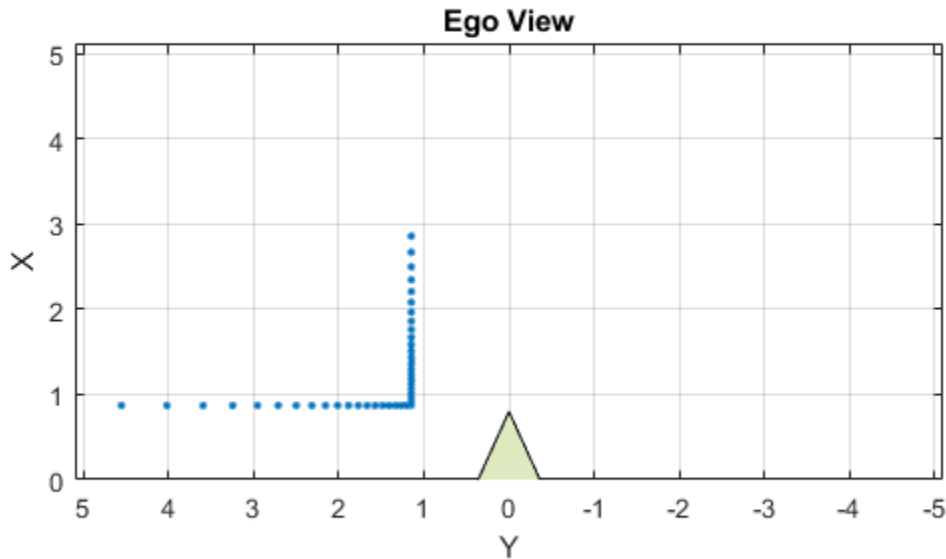
% Select a waypoint to visualize scan data
Vehiclepose = traj(350, :);

% Generate lidar readings
[ranges, angles] = lidar(Vehiclepose, map);

% Store and 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, 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 5, 2, 1 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, 30)';
% Specify color values - white, yellow, orange, red
colors = [1 1 1; 1 1 0; 1 0.5 0; 1 0 0];

% Specify radius in meters
radius = [5 2 1];

% Create a 3x1 matrix of type Polygon
detAreaHandles = repmat(images.roi.Polygon, [3 1]);

pos = [cos(t) sin(t)] * radius(1);
pos = [0 -2; pos(14:17, :); 0 2];
detAreaHandles(1) = drawpolygon(...
    'Parent', detAxes, ...
    'InteractionsAllowed', 'reshape', ...
    'Position', pos, ...
    'StripeColor', 'black', ...
    'Color', colors(2, :));

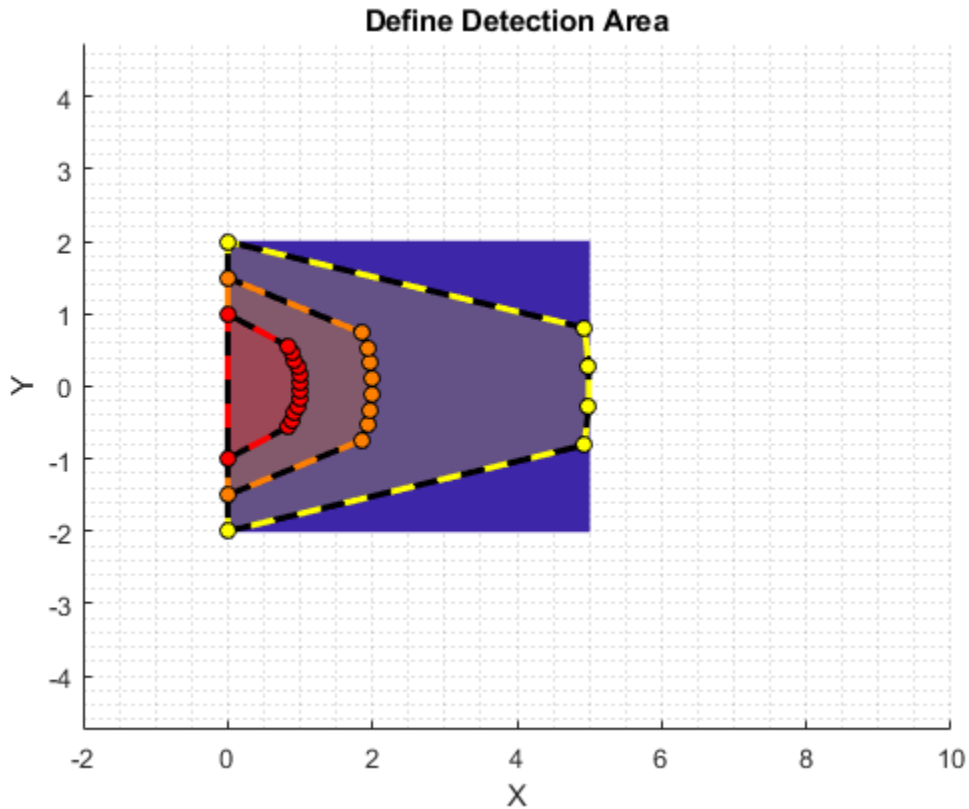
pos = [cos(t) sin(t)] * radius(2);
pos = [0 -1.5; pos(12:19, :); 0 1.5];
detAreaHandles(2) = drawpolygon(...
    'Parent', detAxes, ...
    'InteractionsAllowed', 'reshape', ...
    'Position', pos, ...
    'StripeColor', 'black', ...
    'Color', colors(3, :));

pos = [cos(t) sin(t)] * radius(3);
pos = [0 -1; pos(10:21, :); 0 1];
detAreaHandles(3) = drawpolygon(...
    'Parent', detAxes, ...
    'InteractionsAllowed', 'reshape', ...
    'Position', pos, ...
    'StripeColor', 'black', ...
    'Color', colors(4, :));

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 blue 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);

colormap(ax3, colors);

% Plot detection area
plotObstacleDisplay(display)

```

Run Simulation

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

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

Obstacles that do not fall within the detection range are at safe distance 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 obstacles.
- 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 object you created to get range and angle measurement.

```
% Retrieve lidar scans
[ranges, angles] = lidar(currentPose, map);
scan = lidarScan(ranges, angles);

% Store 2-D scan as point cloud
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, currentPose(3));

% 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.

```
nearxy = zeros(numClusters, 2);
maxlevel = -inf;
% 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 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.

```
% Round the index and get the level of each obstacle from detArea
cols = idx(validIdx, 1);
rows = idx(validIdx, 2);
levels = double(detArea(sub2ind(size(detArea), rows, cols)));

if ~isempty(levels)
    level = max(levels);
    maxlevel = max(maxlevel, level);
    xyInds = find(validIdx);
    xyInds = xyInds(levels == level);
    % Get the nearest coordinates of obstacle in detection area
    nearxy(i, :) = helperNearObstacles(xy(xyInds, :));
else
    % Get the nearest coordinates of obstacle in the cluster
    nearxy(i, :) = helperNearObstacles(xy);
end
end
% Display a warning color representing the danger level. If the
% obstacle does not fall in the detection area, do not display a color.
switch maxlevel
    % Red region
    case 3
        circleDisplay(display, colors(4, :))
    % Orange region
    case 2
        circleDisplay(display, colors(3, :))
    % Yellow region
    case 1
        circleDisplay(display, colors(2, :))
    % Default case
    otherwise
        circleDisplay(display, [])
end
```

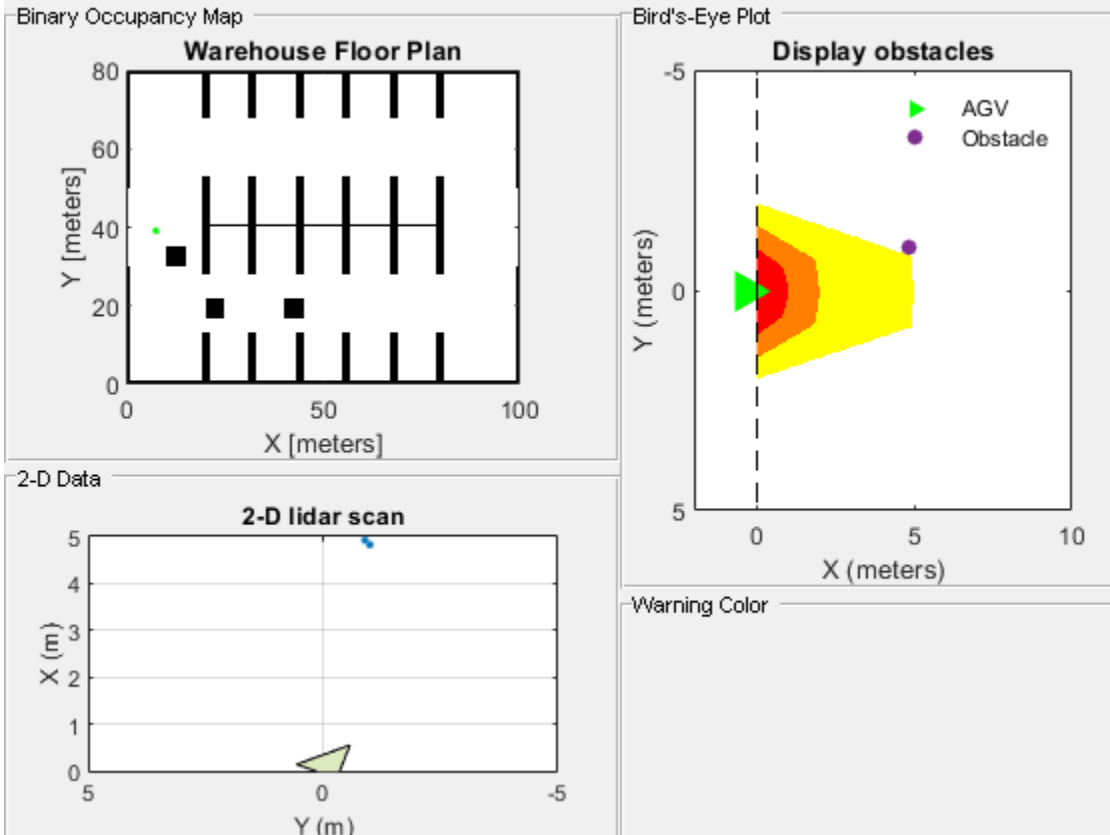
Display Points of Obstacles Closest to the AGV

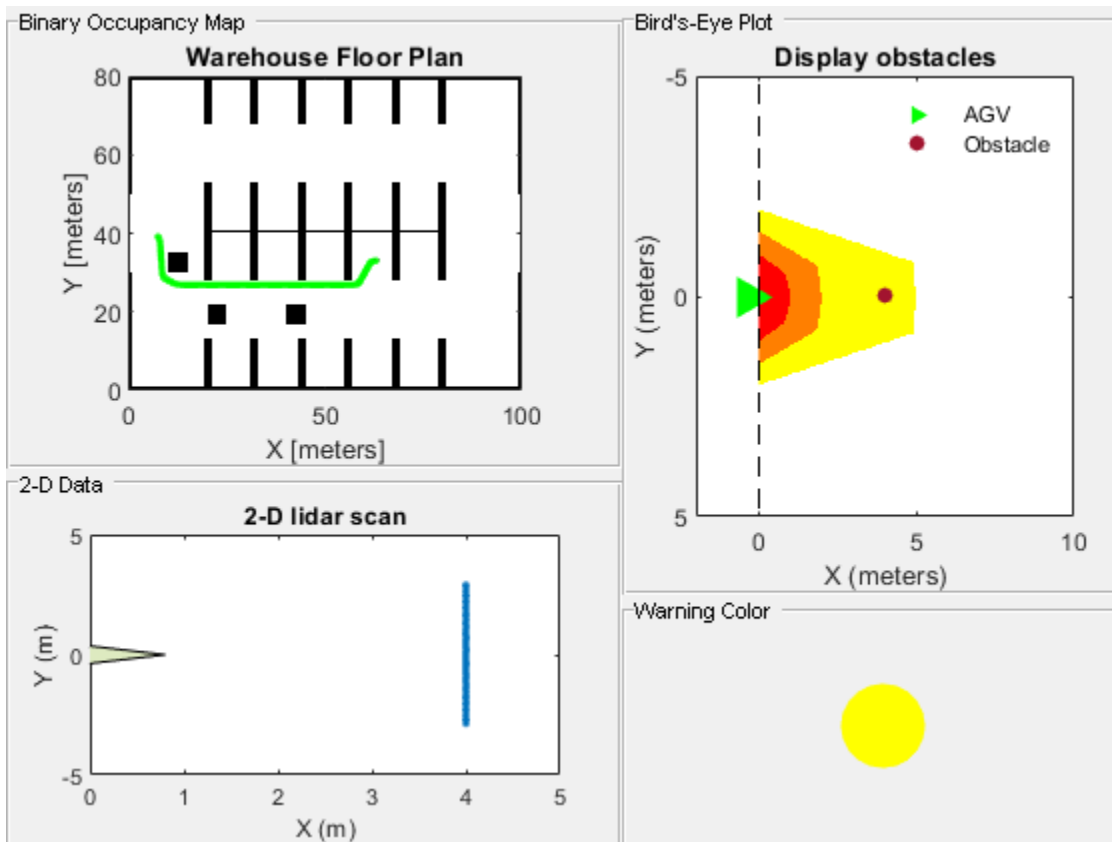
As most of the obstacles in the warehouse are linear and long, display only the point of each obstacle cluster closest to the AGV. Obstacles display as filled circles in the **Bird's-Eye Plot** pane of the visualization window.

```

for i = 1:numClusters
    % Display obstacles if exist in the mentioned range of axes3
    sc(i, :) = displayObstacles(display, nearxy(i, :));
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

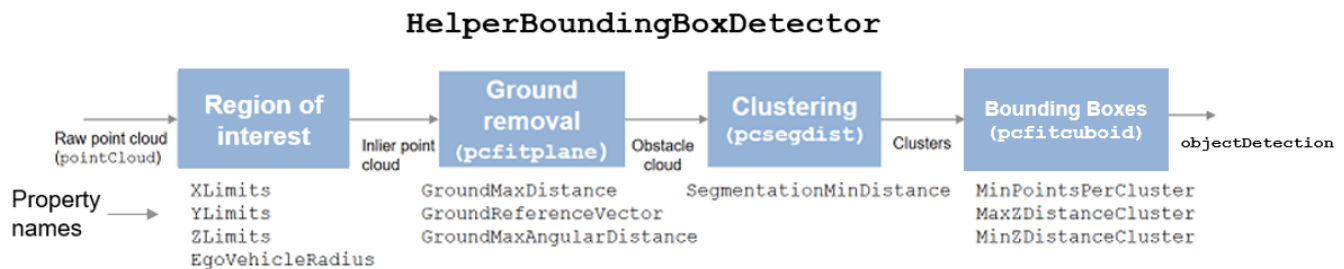
Due to high resolution capabilities of the lidar sensor, each scan from the sensor contains a large number of points, commonly known as a point cloud. This raw data must be preprocessed to extract objects of interest, such as cars, cyclists, and pedestrians. In this example, you use a classical segmentation 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 1-41 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/TrackVehiclesUsingLidarExampleD';
    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:

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

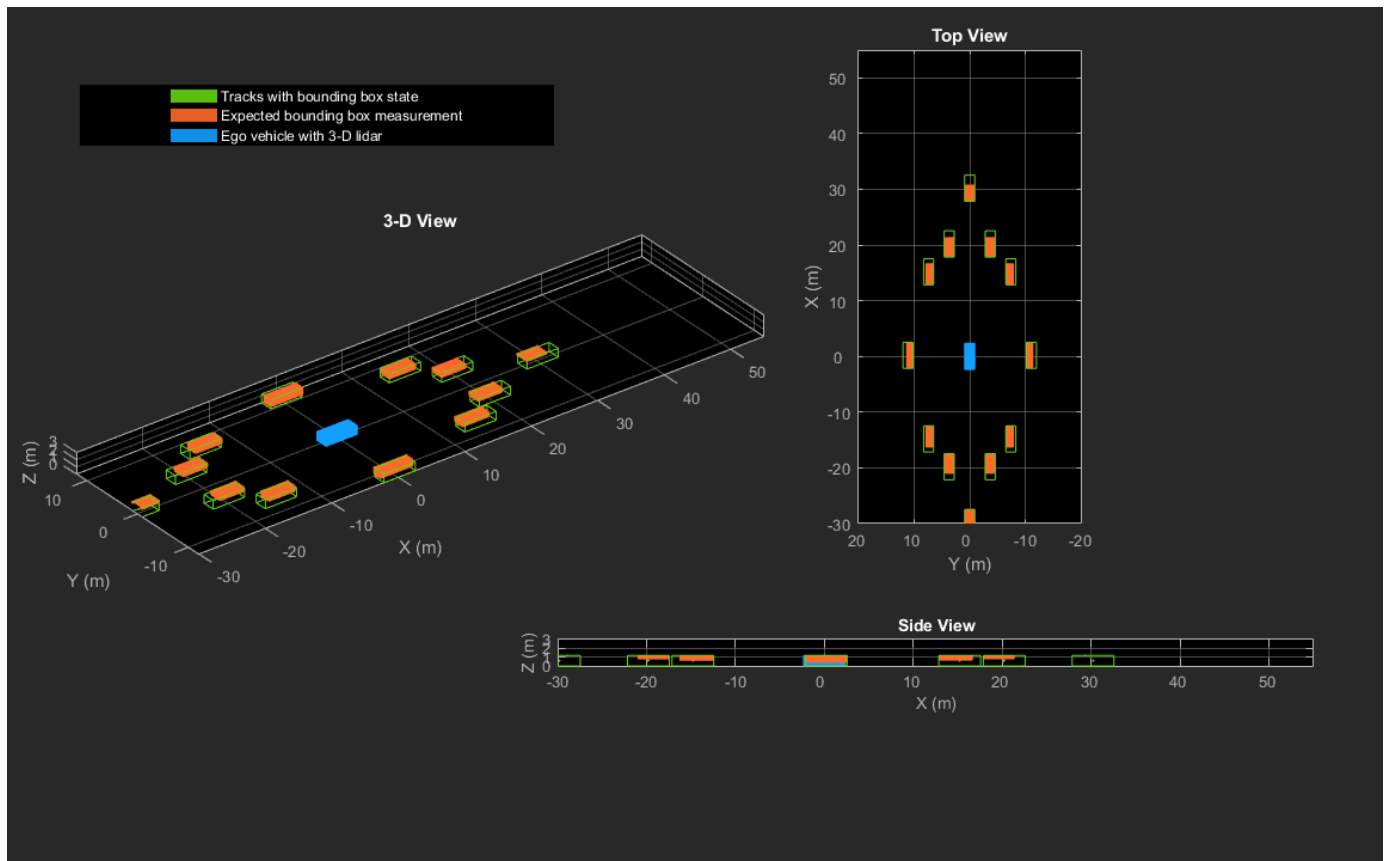
$$x_{ct} = [x \ \dot{x} \ y \ \dot{y} \ \dot{\theta} \ z \ \dot{z} \ \theta \ l \ w \ h]$$

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

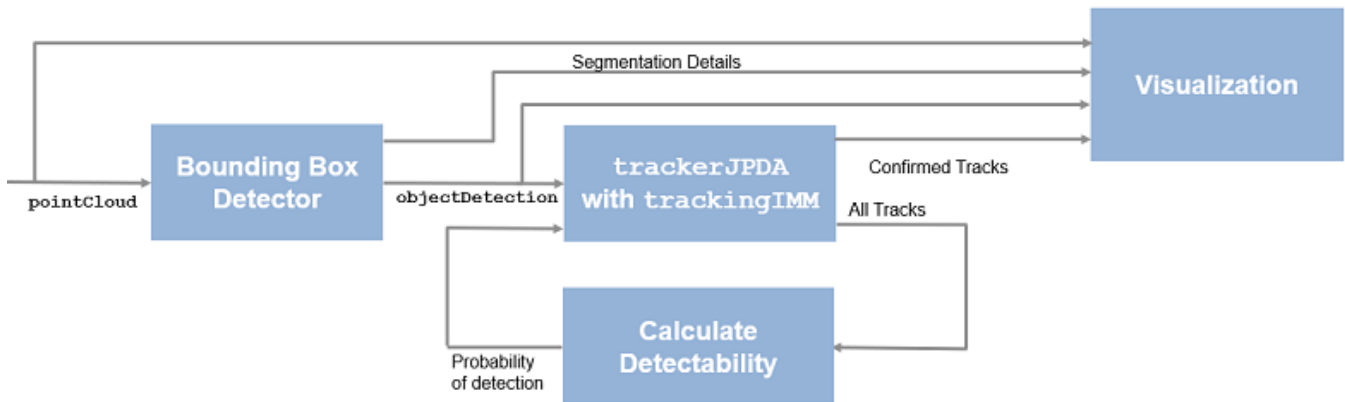
bounding box measurements. Because the state contains information about size of the target, the measurement model includes the effect of center-point offset and bounding box shrinkage, as perceived by the sensor, due to effects like self-occlusion [1]. This effect is modeled by a shrinkage factor that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



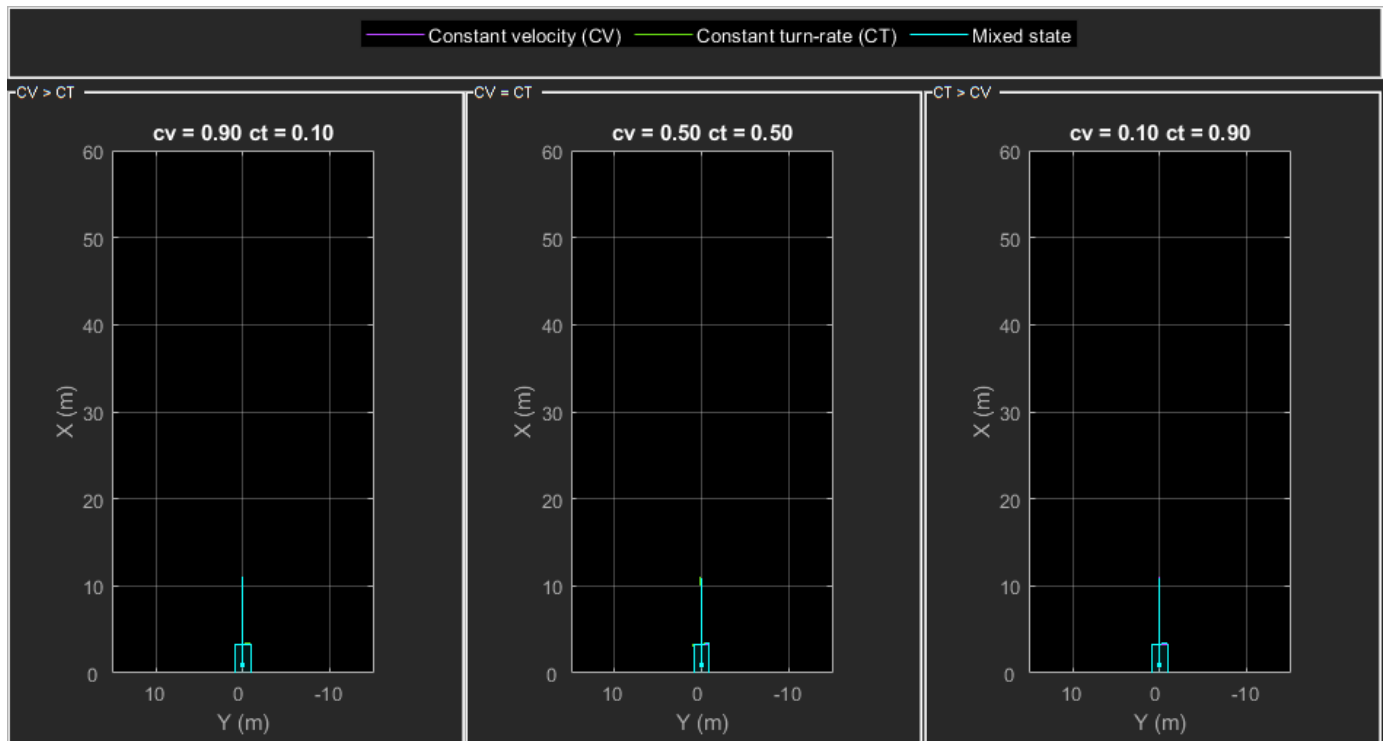
Set Up Tracker and Visualization

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

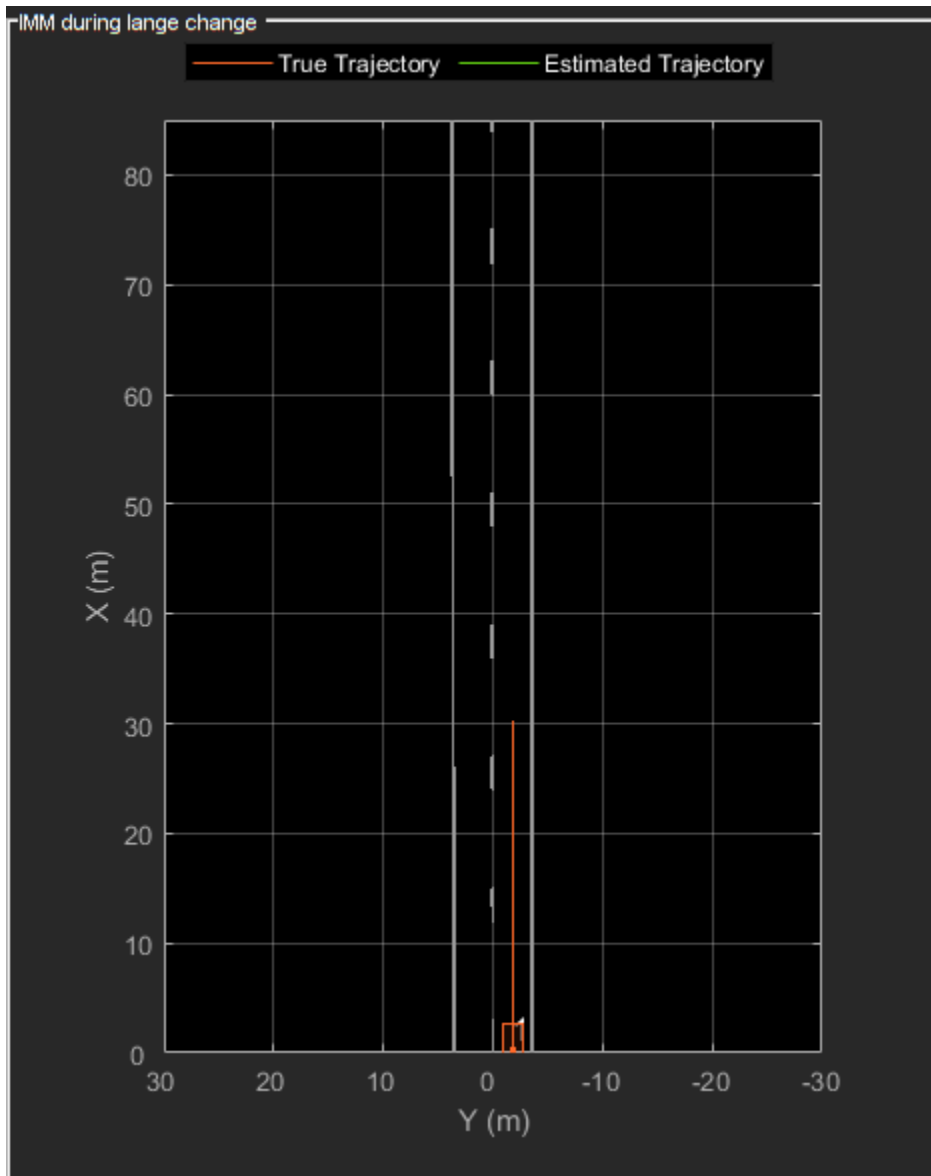


Now, set up the tracker and the visualization used in the example.

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,detectableTracksI
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

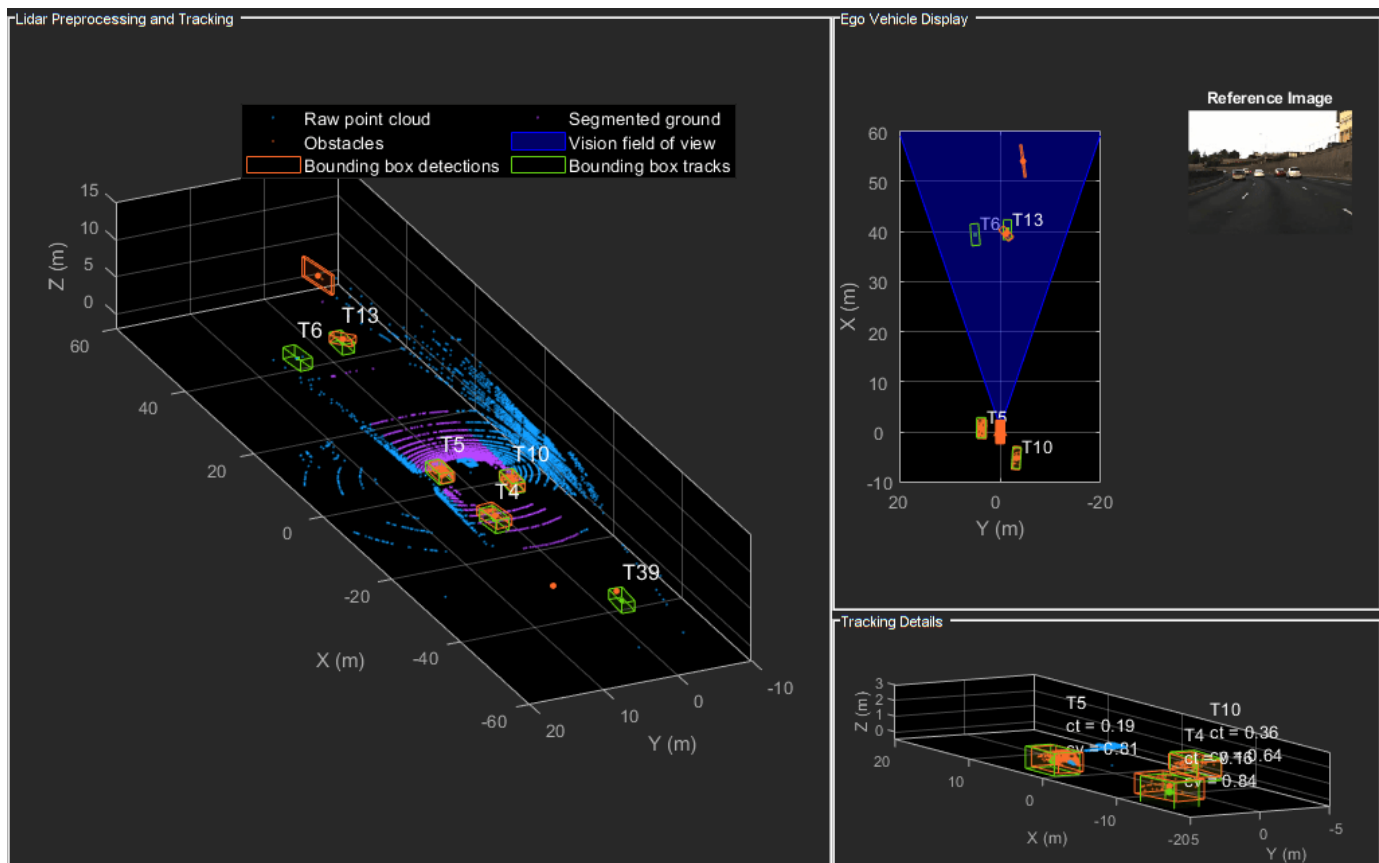
% Snap a figure at time = 18
if abs(time - 18) < dT/2
    snapnow(displayObject);
end

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 w
    % -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)));

```

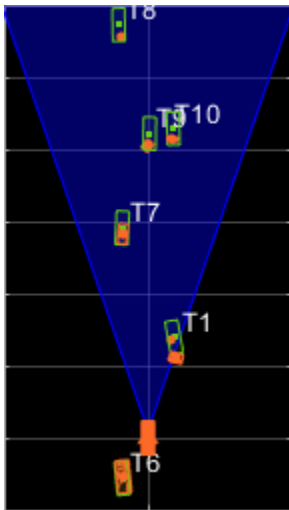
1

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

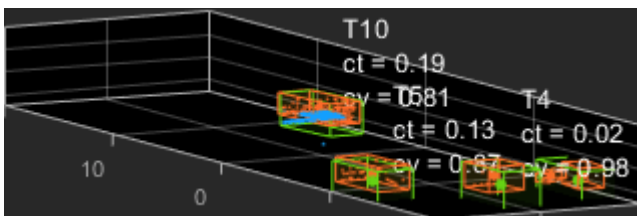
Now, analyze different events in the scenario and understand how the combination of lidar measurement model, joint probabilistic data association, and interacting multiple model filter, helps achieve a good estimation of the vehicle tracks.

Track Maintenance



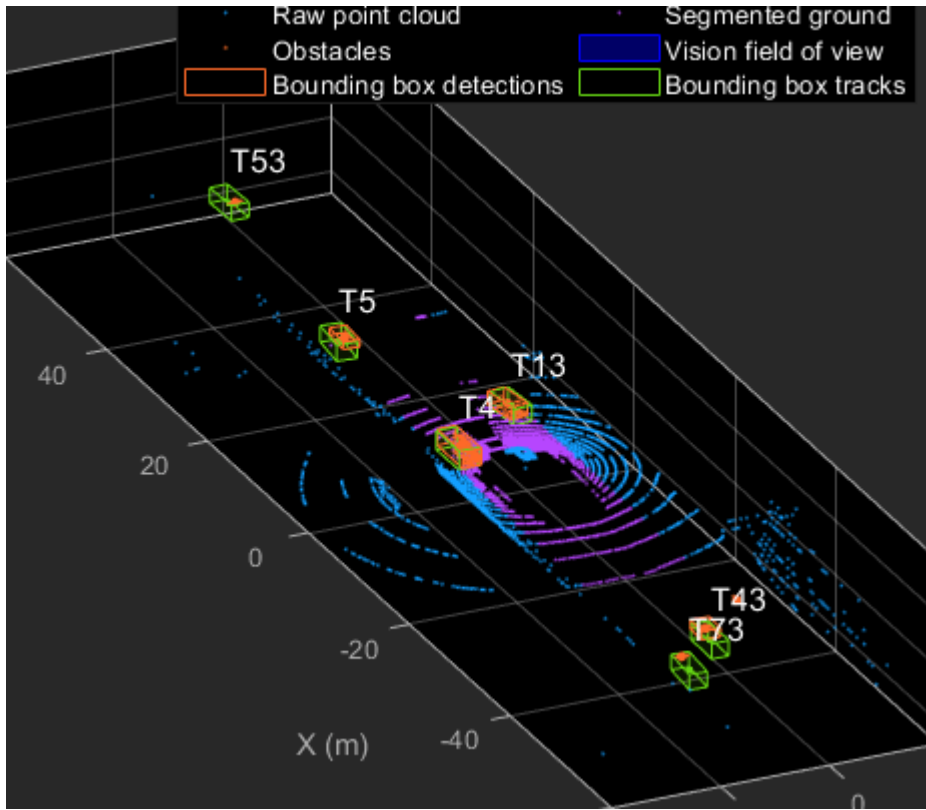
The animation above shows the simulation between time = 3 seconds and time = 16 seconds. Notice that tracks such as T10 and T6 maintain their IDs and trajectory during the time span. However, track T9 is lost because the tracked vehicle was missed (not detected) for a long time by the sensor. Also, notice 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
% uncertainty.
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,currentPC)
        % Crop point cloud
        [pcSurvived,survivedIndices,croppedIndices] = cropPointCloud(currentPointCloud,obj.XLimits)
        % Remove ground plane
```

```

        [pcObstacles,obstacleIndices,groundIndices] = removeGroundPlane(pcSurvived,obj.GroundPlane);
        % Form clusters and get bounding boxes
        detBBoxes = getBoundingBoxes(pcObstacles,obj.SegmentationMinDistance,obj.MinDetectionDistance);
        % Assemble detections
        if isempty(obj.MeasurementParameters)
            measParams = {};
        else
            measParams = obj.MeasurementParameters;
        end
        bboxDets = assembleDetections(detBBoxes,obj.MeasurementNoise,measParams,time);
    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,minZDistance)
% 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
end

```

```

        bboxes = bboxes(:,isValidCluster);
    end

    function [ptCloudOut,obstacleIndices,groundIndices] = removeGroundPlane(ptCloudIn,maxGroundDist,
        % This method removes the ground plane from point cloud using
        % pcfplane.
        [~,groundIndices,outliers] = pcfplane(ptCloudIn,maxGroundDist,referenceVector,maxAngularDi
        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 noise
        '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
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).

Build Map from 2-D Lidar Scans Using SLAM

This example shows you how to implement the simultaneous localization and mapping (SLAM) algorithm on a series of 2-D lidar scans using scan processing algorithms and pose graph optimization (PGO). The goal of this example is to estimate the trajectory of the robot and build a map of the environment.

The SLAM algorithm in this example incrementally processes the lidar scans and builds a pose graph to create the map of the environment. To overcome the drift accumulated in estimated robot trajectory, the example recognizes previously visited places through scan matching and utilizes the loop closure information to optimize poses and update the map of the environment. To optimize the pose graph, this example uses 2-D pose graph optimization from Navigation Toolbox™.

In this example, you learn how to:

- Estimate robot trajectory from a series of scans using scan registration algorithms.
- Optimize the drift in estimated robot trajectory by identifying previously visited places (loop closures).
- Visualize the map of the environment using scans and their absolute poses.

Load Laser Scans

This example uses data collected in an indoor environment using a Jackal™ robot from Clearpath Robotics™. The robot is equipped with a SICK™ TiM-511 laser scanner with a maximum range of 10 meters. Load the `offlineSlamData.mat` file containing the laser scans into the workspace.

```
data = load('offlineSlamData.mat');
scans = data.scans;
```

Robot Trajectory Estimation

The example uses the `matchScansGrid` and `matchScans` functions to estimate the relative pose between successive scans. The `matchScansGrid` function provides the initial estimate for the relative pose, which is accurate up to the specified resolution. The `matchScans` function uses the estimate as an initial guess, and refines the relative pose for better estimation.

```
% Set maximum lidar range to be slightly smaller than maximum range of the
% scanner, as the laser readings are less accurate near maximum range
maxLidarRange = 8;
% Set the map resolution to 10 cells per meter, which gives a precision of
% 10cm
mapResolution = 10;
% Create a pose graph object and define information matrix
pGraph = poseGraph;
infoMat = [1 0 0 1 0 1];
% Loop over each scan and estimate relative pose
prevScan = scans{1};
for i = 2:numel(scans)
    currScan = scans{i};
    % Estimate relative pose between current scan and previous scan
    [relPose,stats] = matchScansGrid(currScan,prevScan, ...
        'MaxRange',maxLidarRange,'Resolution',mapResolution);
    % Refine the relative pose
    relPoseRefined = matchScans(currScan,prevScan,'initialPose',relPose);
    % Add relative pose to the pose graph object
```




Drift Correction

Correct the drift in trajectory by accurately detecting the *loops*, which are places the robot has previously visited. Add the loop closure edges to the pose graph, which helps to correct the drift in trajectory during pose graph optimization.

Loop Closure Detection

Loop closure detection determines whether the robot has previously visited the current location. The search is performed by matching the current scan against the previous scans around the current robot location, within the radius specified by `loopClosureSearchRadius`. A scan is accepted as a match if the match score is greater than the specified `loopClosureThreshold`. Loop closures are detected using the `helperDetectLoop` helper function, which is attached to this example as a supporting file.

Adjust the loop closure parameters based on the quality of your results. You can increase the `loopClosureThreshold` value to reject false positives in loop closure detection, but the function might still return bad matches in environments with similar or repeated features. To address this, increase the `loopClosureSearchRadius` value to search a larger radius around the current pose estimate for loop closures, though this increases computation time.

```
loopClosureThreshold = 110;
loopClosureSearchRadius = 2;
[loopClosureEdgeIds,loopClosurePoses] = helperDetectLoop(scans,pGraph, ...
    loopClosureSearchRadius,loopClosureThreshold);
```

Trajectory Optimization

Add the detected loop closure edges to the pose graph to correct the drift in the estimated trajectory. Use the `optimizePoseGraph` (Navigation Toolbox) function to optimize the pose graph.

```
% Add loop closure edges to pose graph
if ~isempty(loopClosureEdgeIds)
    for k = 1:size(loopClosureEdgeIds,1)
        pGraph.addRelativePose(loopClosurePoses(k,:),infoMat, ...
            loopClosureEdgeIds(k,1),loopClosureEdgeIds(k,2));
    end
end
% Optimize pose graph
updatedPGraph = optimizePoseGraph(pGraph);
```

Visualization

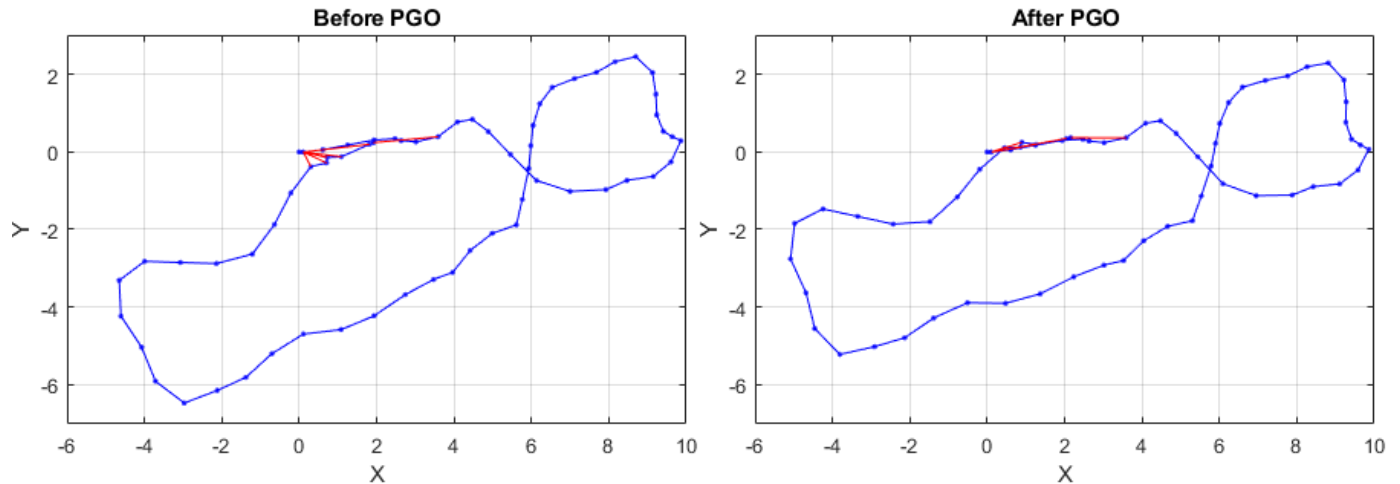
Visualize the change in robot trajectory before and after pose graph optimization. The red lines represent loop closure edges.

```
hFigTraj = figure('Position',[0 0 900 450]);

% Visualize robot trajectory before optimization
axPGraph = subplot(1,2,1,'Parent',hFigTraj);
axPGraph.Position = [0.04 0.1 0.45 0.8];
show(pGraph,'IDs','off','Parent',axPGraph);
title(axPGraph,'Before PGO')

% Visualize robot trajectory after optimization
axUpdatedPGraph = subplot(1,2,2,'Parent',hFigTraj);
axUpdatedPGraph.Position = [0.54 0.1 0.45 0.8];
show(updatedPGraph,'IDs','off','Parent',axUpdatedPGraph);
title(axUpdatedPGraph,'After PGO')
axis([axPGraph axUpdatedPGraph],[-6 10 -7 3])
sgtitle('Robot Trajectory','FontWeight','bold')
```

Robot Trajectory



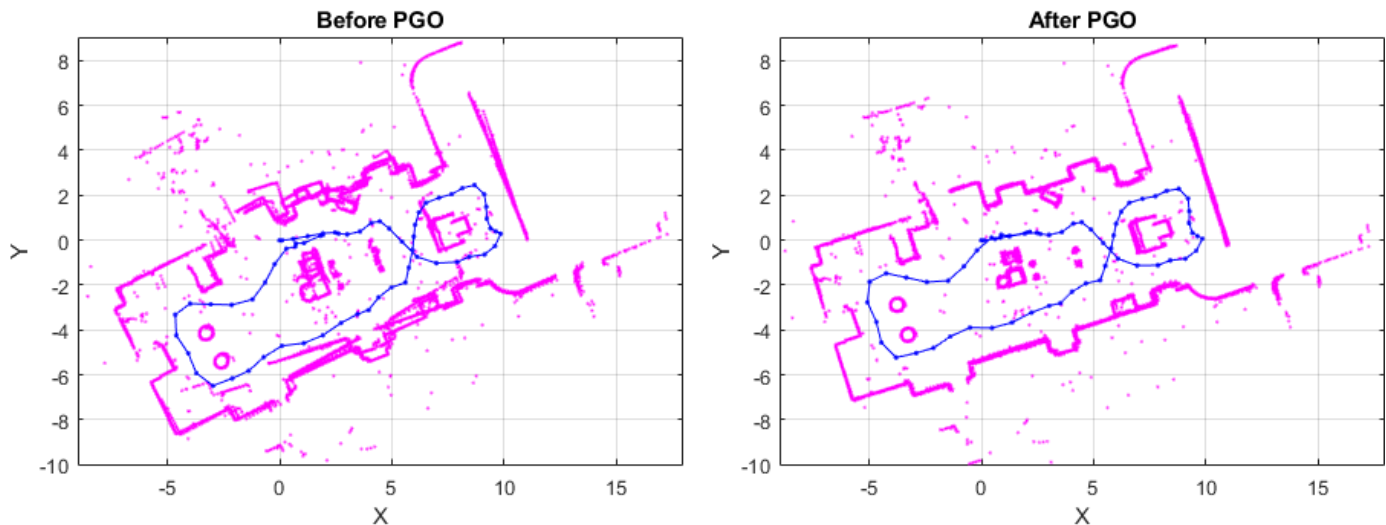
Visualize the map of the environment and robot trajectory before and after pose graph optimization.

```
hFigMapTraj = figure('Position',[0 0 900 450]);

% Visualize map and robot trajectory before optimization
axOldMap = subplot(1,2,1,'Parent',hFigMapTraj);
axOldMap.Position = [0.05 0.1 0.44 0.8];
helperShow(scans,pGraph,maxLidarRange,axOldMap)
title(axOldMap,'Before PGO')

% Visualize map and robot trajectory after optimization
axUpdatedMap = subplot(1,2,2,'Parent',hFigMapTraj);
axUpdatedMap.Position = [0.56 0.1 0.44 0.8];
helperShow(scans,updatedPGraph,maxLidarRange,axUpdatedMap)
title(axUpdatedMap,'After PGO')
axis([axOldMap axUpdatedMap],[-9 18 -10 9])
sgtitle('Map of the Environment and Robot Trajectory','FontWeight','bold')
```


Map of the Environment and Robot Trajectory



Supporting Functions

The `helperShow` helper function visualizes the map of the environment and trajectory of the robot. The function transforms lidar scans using their corresponding poses to create a map of the environment.

```
function helperShow(scans,pGraph,maxRange,ax)
    hold(ax,'on')
    for i = 1:numel(scans)
        sc = transformScan(scans{i}.removeInvalidData('RangeLimits',[0.02 maxRange]), ...
            pGraph.nodes(i));
        scPoints = sc.Cartesian;
        plot(ax,scPoints(:,1),scPoints(:,2),'.','MarkerSize',3,'color','m')
    end
    nds = pGraph.nodes;
    plot(ax,nds(:,1),nds(:,2),'-','MarkerSize',5,'color','b')
    hold(ax,'off')
    axis(ax,'equal')
    box(ax,'on')
    grid(ax,'on')
    xlabel('X')
    ylabel('Y')
end
```

See Also

Functions

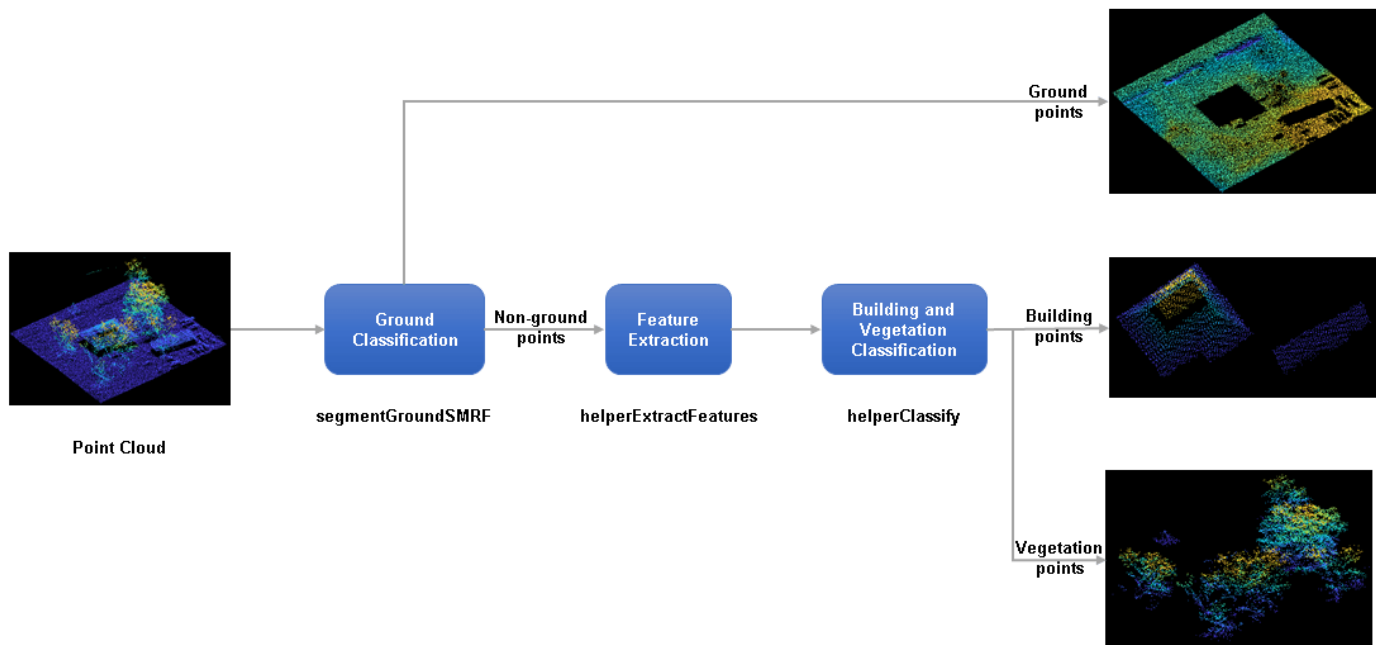
[matchScansGrid](#) | [matchScans](#) | [addRelativePose](#) (Navigation Toolbox) | [show](#) (Navigation Toolbox) | [optimizePoseGraph](#) (Navigation Toolbox)

Objects

lidarScan | poseGraph (Navigation Toolbox)

Terrain Classification for Aerial Lidar Data

This example shows you how to segment and classify terrain in aerial lidar data as ground, building, and vegetation. The example uses a LAZ file captured by an airborne lidar system as input. First, classify the point cloud data in the LAZ file into ground and non-ground points. Then, further classify non-ground points into building and vegetation points based on normals and curvature features. This figure provides an overview of the process.

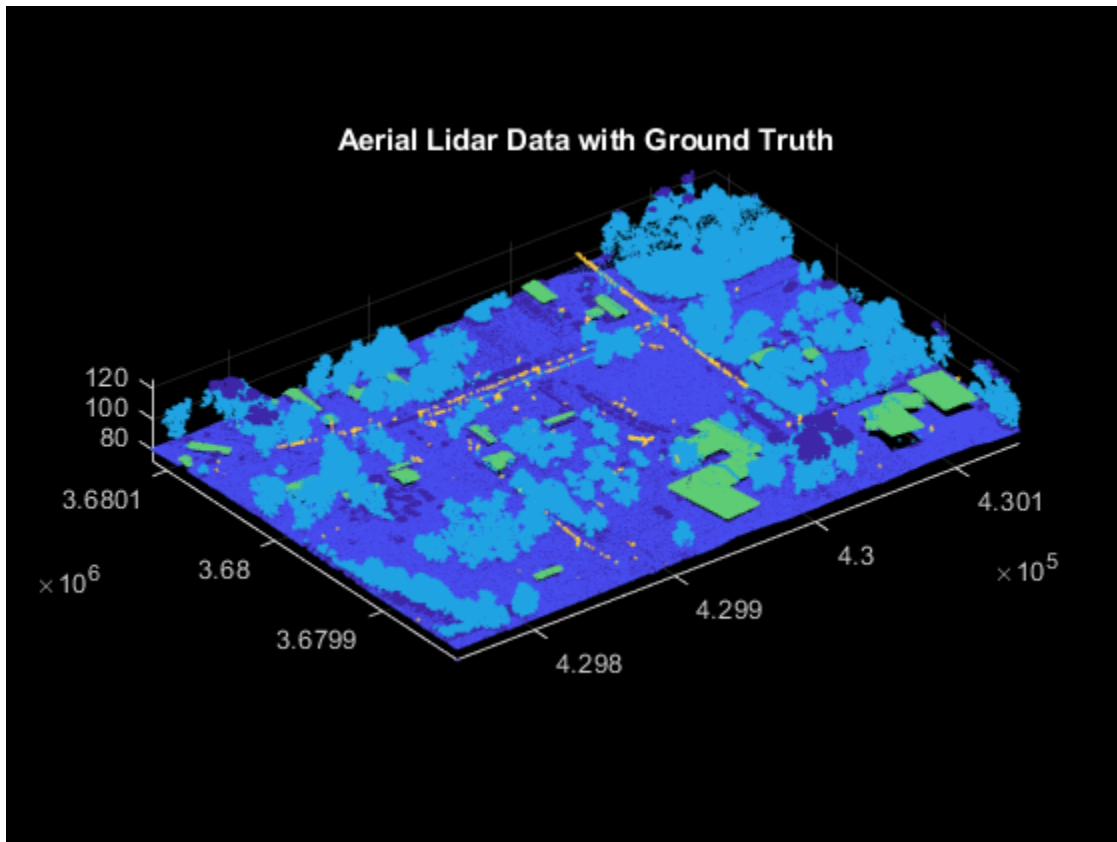


Load and Visualize Data

Load the point cloud data and corresponding ground truth labels from the LAZ file, `aerialLidarData.laz`, obtained from the Open Topography Dataset [1] on page 1-0 . The point cloud consists of various classes, including ground, building, and vegetation. Load the point cloud data and the corresponding ground truth labels into the workspace using the `readPointCloud` object function of the `lasFileReader` object. Visualize the point cloud, color-coded according to the ground truth labels, using the `pcshow` function.

```

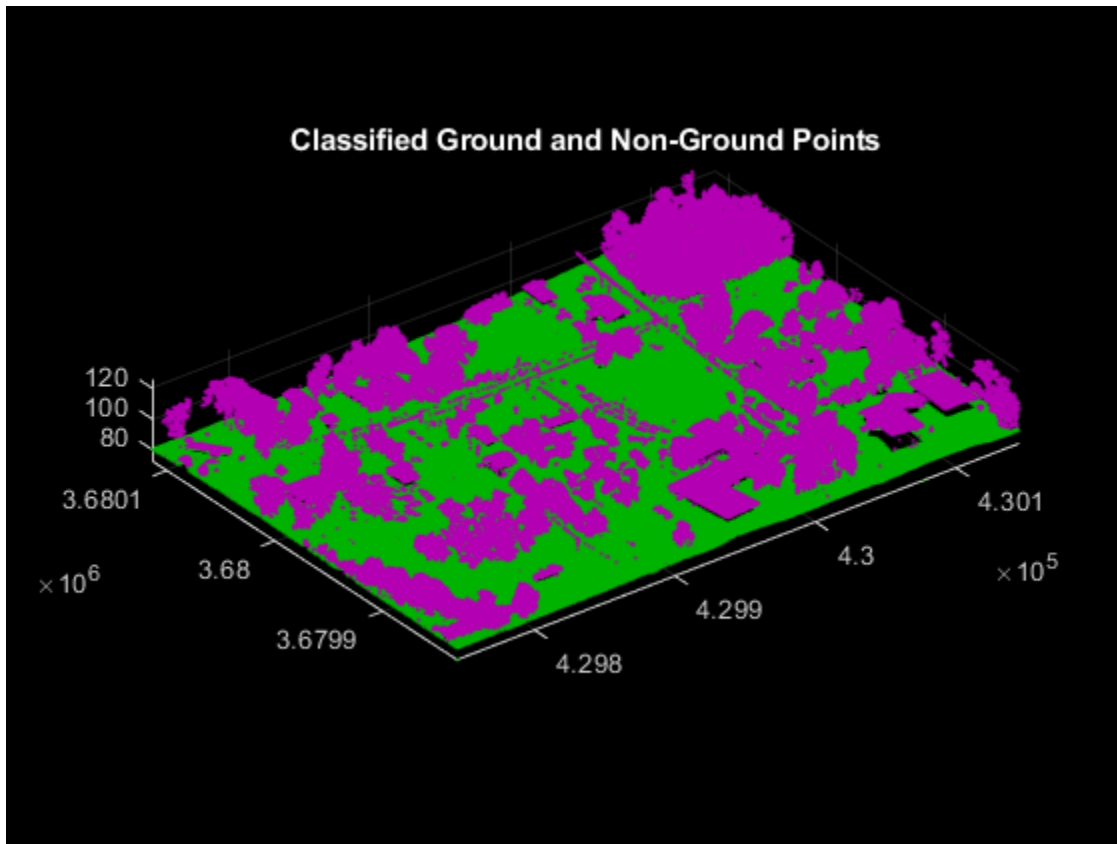
lazfile = fullfile(toolboxdir('lidar'),'lidardata','las','aerialLidarData.laz');
% Read LAZ data from file
lazReader = lasFileReader(lazfile);
% Read point cloud and corresponding ground truth labels
[ptCloud,pointAttributes] = readPointCloud(lazReader, ...
    'Attributes','Classification');
grdTruthLabels = pointAttributes.Classification;
% Visualize the input point cloud with corresponding ground truth labels
figure
pcshow(ptCloud.Location,grdTruthLabels)
title('Aerial Lidar Data with Ground Truth')
  
```



Ground Classification

Ground classification is a preprocessing step to segment the input point cloud as ground and non-ground. Segment the data loaded from the LAZ file into ground and non-ground points using the `segmentGroundSMRF` function.

```
[groundPtsIdx,nonGroundPtCloud,groundPtCloud] = segmentGroundSMRF(ptCloud);  
% Visualize ground and non-ground points in green and magenta, respectively  
figure  
pcshowpair(nonGroundPtCloud,groundPtCloud)  
title('Classified Ground and Non-Ground Points')
```



Feature Extraction

Extract features from the point cloud using the `helperExtractFeatures` function. All the helper functions are attached to this example as supporting files. The helper function estimates the normal and curvature values for each point in the point cloud. These features provide underlying structure information at each point by correlating it with the points in its neighborhood.

You can specify the number of neighbors to consider. If the number of neighbors is too low, the helper function overclusters vegetation points. If the number of neighbors is too high, there is no defining boundary between buildings and vegetation, as vegetation points near the building points are misclassified.

```
neighbors = 10;
[normals,curvatures,neighInds] = helperExtractFeatures(nonGroundPtCloud, ...
    neighbors);
```

Building and Vegetation Classification

The helper function uses the variation in normals and curvature to distinguish between buildings and vegetation. The buildings are more planar compared to vegetation, so the change in curvature and the relative difference of normals between neighbors is less for points belonging to buildings. Vegetation points are more scattered, which results in a higher change in curvatures as compared to buildings. The `helperClassify` function classifies the non-ground points into building and vegetation. The helper function classifies the points as building based on the following criteria:

- The curvature of each point must be small, within the specified curvature threshold, `curveThresh`.

- The neighboring points must have similar normals. The cosine similarity between neighboring normals must be greater than the specified normal threshold, `normalThresh`.

The points that do not satisfy the above criteria are marked as vegetation. The helper function labels the points belonging to vegetation as 1 and building as 2.

```
% Specify the normal threshold and curvature threshold
normalThresh = 0.85;
curveThresh = 0.02;
% Classify the points into building and vegetation
labels = helperClassify(normals,curvatures,neighInds, ...
    normalThresh,curveThresh);
```

Extract the building and vegetation class labels from the ground truth label data. As the LAZ file has many classes, you must first isolate the ground, building and vegetation classes. The classification labels are in compliance with the ASPRS standard for LAZ file formats.

- Classification Value 2 — Represents ground points
- Classification Values 3, 4, and 5 — Represent low, medium, and high vegetation points
- Classification Value 6 — Represents building points

Define `maskData` to extract points belonging to the ground, buildings, and vegetation from the input point cloud.

```
maskData = grdTruthLabels>=2 & grdTruthLabels<=6;
```

Modify the ground truth labels of the input point cloud, specified as `grdTruthLabels`.

```
% Compress low, medium, and high vegetation to a single value
grdTruthLabels(grdTruthLabels>=3 & grdTruthLabels<=5) = 4;
% Update grdTruthLabels for metrics calculation
grdTruthLabels(grdTruthLabels == 2) = 1;
grdTruthLabels(grdTruthLabels == 4) = 2;
grdTruthLabels(grdTruthLabels == 6) = 3;
```

Store the predicted labels acquired from previous classification steps in `estimatedLabels`.

```
estimatedLabels = zeros(ptCloud.Count,1);
estimatedLabels(groundPtsIdx) = 1;
estimatedLabels(labels == 1) = 2;
estimatedLabels(labels == 2) = 3;
```

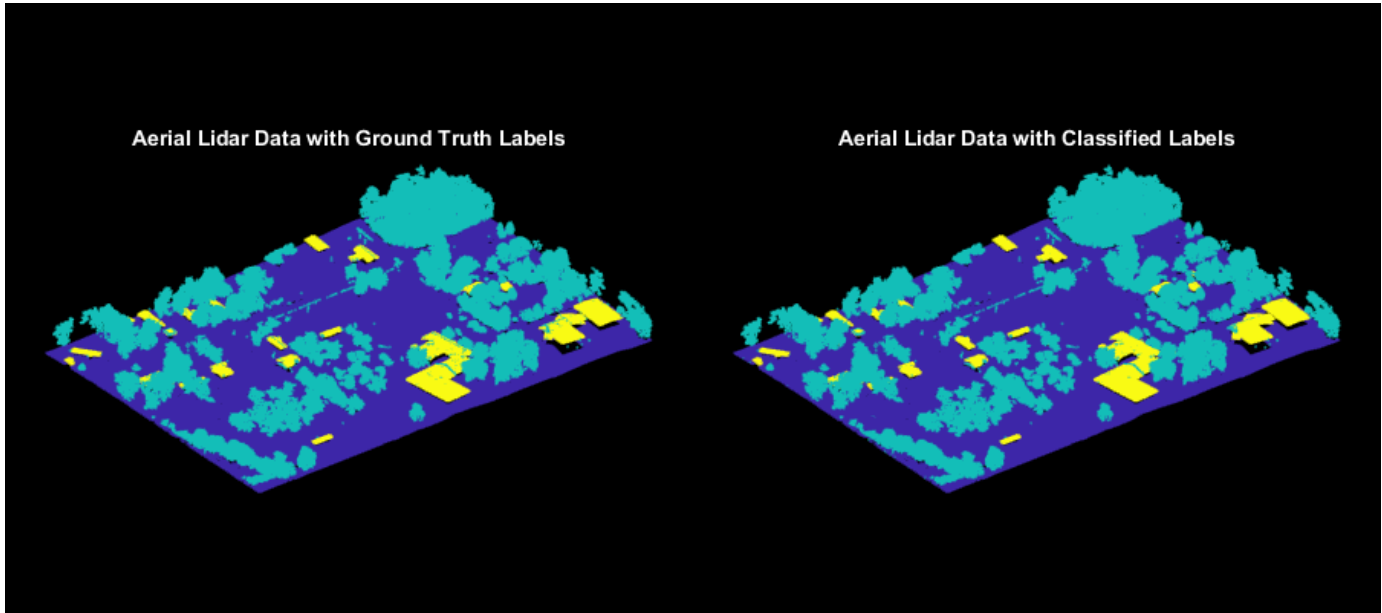
Extract the labels belonging to ground, buildings, and vegetation.

```
grdTruthLabels = grdTruthLabels(maskData);
estimatedLabels = estimatedLabels(maskData);
```

Visualize the terrain with the ground truth and estimated labels.

```
ptCloud = select(ptCloud,maskData);
hFig = figure('Position',[0 0 900 400]);
axMap1 = subplot(1,2,1,'Color','black','Parent',hFig);
axMap1.Position = [0 0.2 0.5 0.55];
pcshow(ptCloud.Location,grdTruthLabels,'Parent',axMap1)
axis off
title(axMap1,'Aerial Lidar Data with Ground Truth Labels')
axMap2 = subplot(1,2,2,'Color','black','Parent',hFig);
```

```
axMap2.Position = [0.5,0.2,0.5,0.55];
pcshow(ptCloud.Location,estimatedLabels,'Parent',axMap2)
axis off
title(axMap2,'Aerial Lidar Data with Classified Labels')
```



Validation

Validate the classification by computing the total accuracy on the given point cloud along with the class accuracy, intersection-over-union (IoU), and weighted IoU.

```
confusionMatrix = segmentationConfusionMatrix(estimatedLabels,double(grdTruthLabels));
ssm = evaluateSemanticSegmentation({confusionMatrix}, ...
    {'Ground' 'Vegetation' 'Building'},'Verbose',0);
disp(ssm.DataSetMetrics)
```

GlobalAccuracy	MeanAccuracy	MeanIoU	WeightedIoU
0.99762	0.99417	0.98168	0.99533

```
disp(ssm.ClassMetrics)
```

	Accuracy	IoU
Ground	0.99996	0.99996
Vegetation	0.99195	0.9898
Building	0.99059	0.95526

See Also

Functions

readPointCloud | segmentGroundSMRF | pcnormals | pcshow | pcshowpair |
segmentationConfusionMatrix | evaluateSemanticSegmentation

Objects

lasFileReader

References

[1] Starr, Scott. "Tuscaloosa, AL: Seasonal Inundation Dynamics and Invertebrate Communities." National Center for Airborne Laser Mapping, December 1, 2011. OpenTopography (<https://doi.org/10.5069/G9SF2T3K>)

Data Augmentations for Lidar Object Detection Using Deep Learning

This example shows how to perform typical data augmentation techniques used in 3-D object detection workflows with lidar data.

Lidar object detection methods directly predict 3-D bounding boxes around the objects of interest. Data augmentation methods can help you improve prediction accuracy and avoid overfitting issues while training. This example covers global and local augmentation techniques: global augmentation techniques are applied to the entire point cloud of a scene and local augmentation techniques are applied only to points belonging to individual objects in the scene.

Load Data

Extract the ZIP file attached to this example to the temp directory.

```
unzip('sampleWPIPointClouds.zip',tempdir);
dataLocation = fullfile(tempdir,'sampleWPIPointClouds');
```

Create a file datastore to load PCD files using the `pcread` function.

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

Create a box label datastore for loading the 3-D ground truth bounding boxes.

```
load('sampleWPILabels.mat','trainLabels')
bds = boxLabelDatastore(trainLabels);
```

Use the `combine` function to combine the point clouds and 3-D bounding boxes into a single datastore.

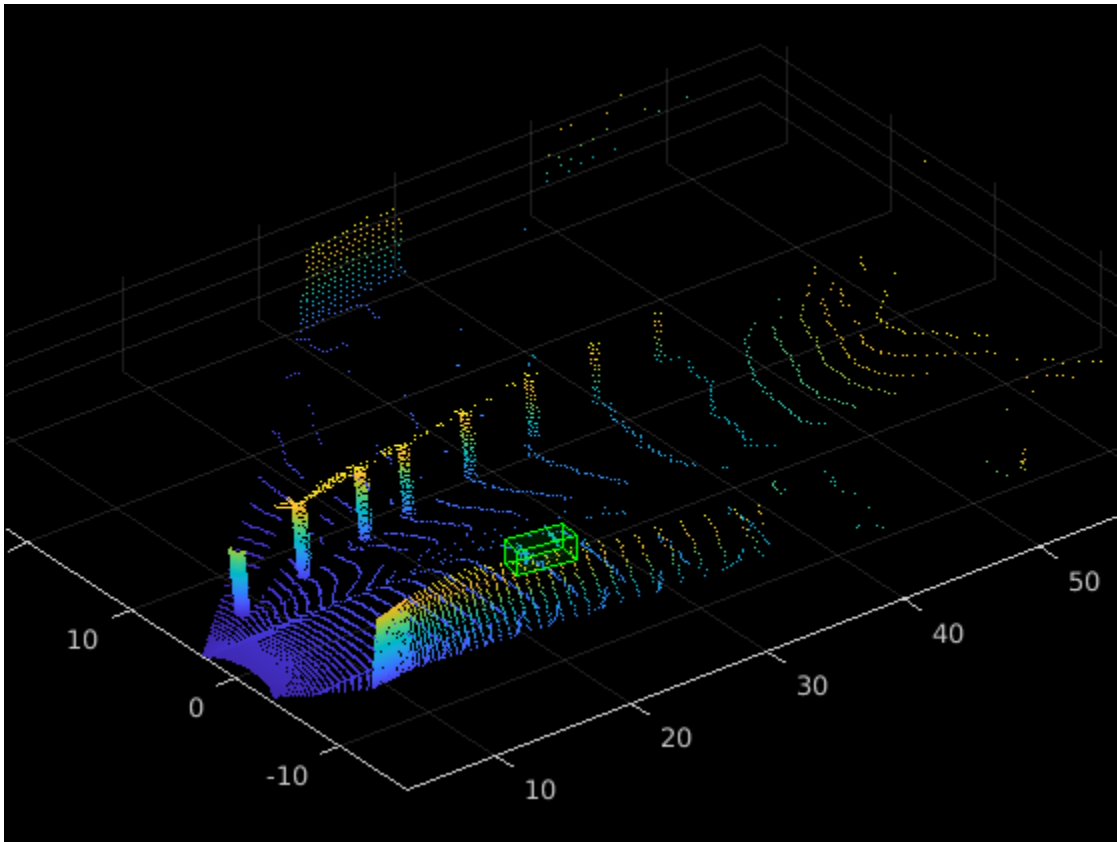
```
cds = combine(lds,bds);
```

Display the point cloud.

```
inputData = read(cds);
ptCloud = inputData{1,1};
gtLabels = inputData{1,2};
figure;
ax = pcshow(ptCloud.Location);
```

Draw the 3-D bounding boxes over the point cloud.

```
showShape('cuboid',gtLabels,'Parent',ax,'Opacity',0.1, ...
'Color','green','LineWidth',0.5);
zoom(ax,2);
```



```
reset(cds);
```

Global Data Augmentation

Global data augmentation techniques are used when the point clouds in a dataset have little variation. A global technique applies a transformation to the entire point cloud to generate new samples of the point cloud that are not present in the original data set. The same transformation is applied to all corresponding ground truth boxes. The following four global data augmentation techniques are commonly used [1 on page 1-0].

- 1 Random rotation
- 2 Random scaling
- 3 Random translation
- 4 Random flipping

Random Rotation of Point Cloud

Randomly rotate the point cloud and the 3-D bounding boxes within the specified range of angles along the z-axis. By randomly rotating the point cloud, you can simulate data points, such as a vehicle taking a turn. The typical range for rotation is [-45 45] degrees.

Set the random seed for reproducibility.

```
rng(1);
```

Define minimum and maximum yaw angles for rotation.

```
minYawAngle = -45;  
maxYawAngle = 45;
```

Define the grid size to bin the point cloud to.

```
gridSize = [32 32 32];
```

Define the limits of the region of interest within the point cloud.

```
axisLimits = [-100 100];
```

Create an output view for the affine transformation.

```
outView = imref3d(gridSize,axisLimits,axisLimits,axisLimits);
```

Calculate a random angle from the specified yaw angle range.

```
theta = minYawAngle + rand*(maxYawAngle - minYawAngle)
```

```
theta = -7.4680
```

Create a transformation that rotates the point clouds and 3-D bounding boxes.

```
tform = randomAffine3d('Rotation',@() deal([0,0,1],theta));
```

Apply the transformation to the point cloud.

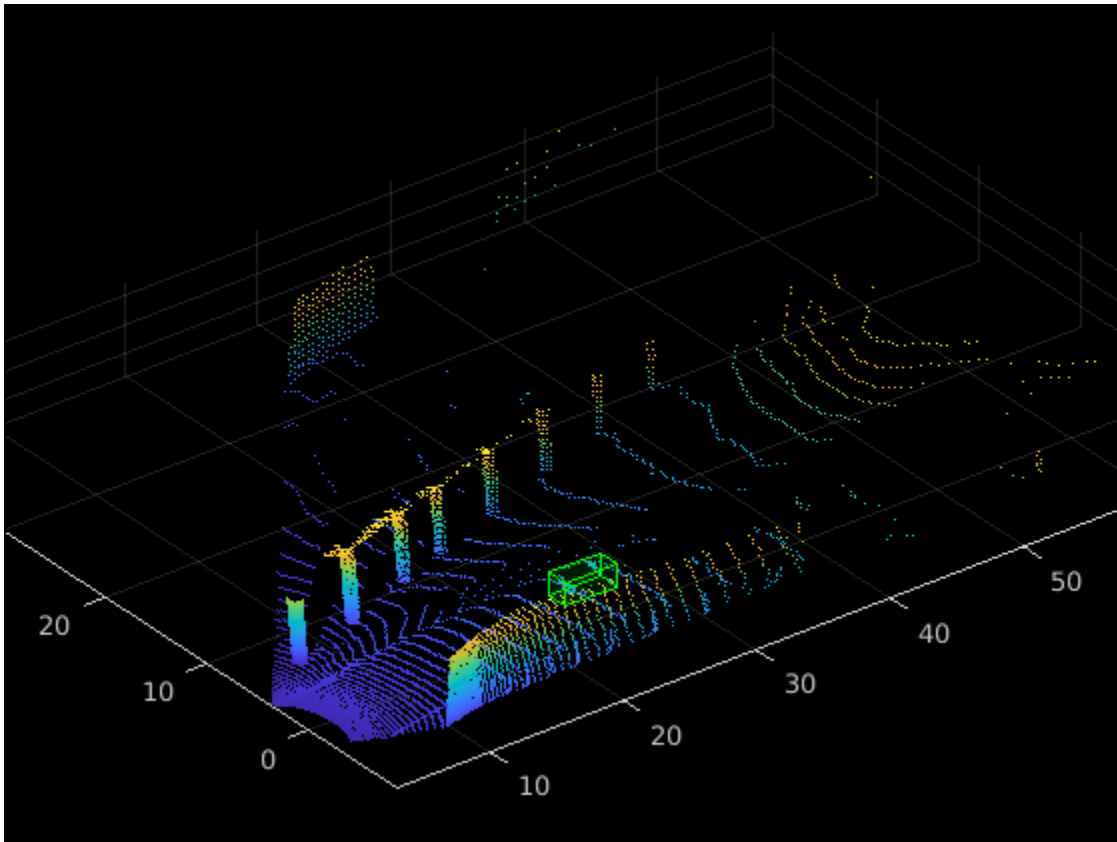
```
ptCloudTransformed = pctransform(ptCloud,tform);
```

Apply the same transformation to the 3-D bounding boxes.

```
gtLabelsTranformed = bboxwarp(gtLabels,tform,outView);
```

Display the rotated point cloud and the ground truth boxes.

```
figure;  
ax1 = pcshow(ptCloudTransformed.Location);  
showShape('cuboid',gtLabelsTranformed,'Parent',ax1,'Opacity',0.1, ...  
          'Color','green','LineWidth',0.5);  
zoom(ax1,2);
```



Random Scaling of Point Cloud

Randomly scale the point cloud and the 3-D bounding boxes from the specified range of scales. The typical scaling range is $[0.95 \ 1.05]$. The example uses the range $[0.5 \ 0.7]$ for better visualization.

Create a transformation to scale the point cloud and 3-D bounding boxes.

```
tform = randomAffine3d('Scale',[0.5 0.7]);
```

Apply the transformation to the point cloud.

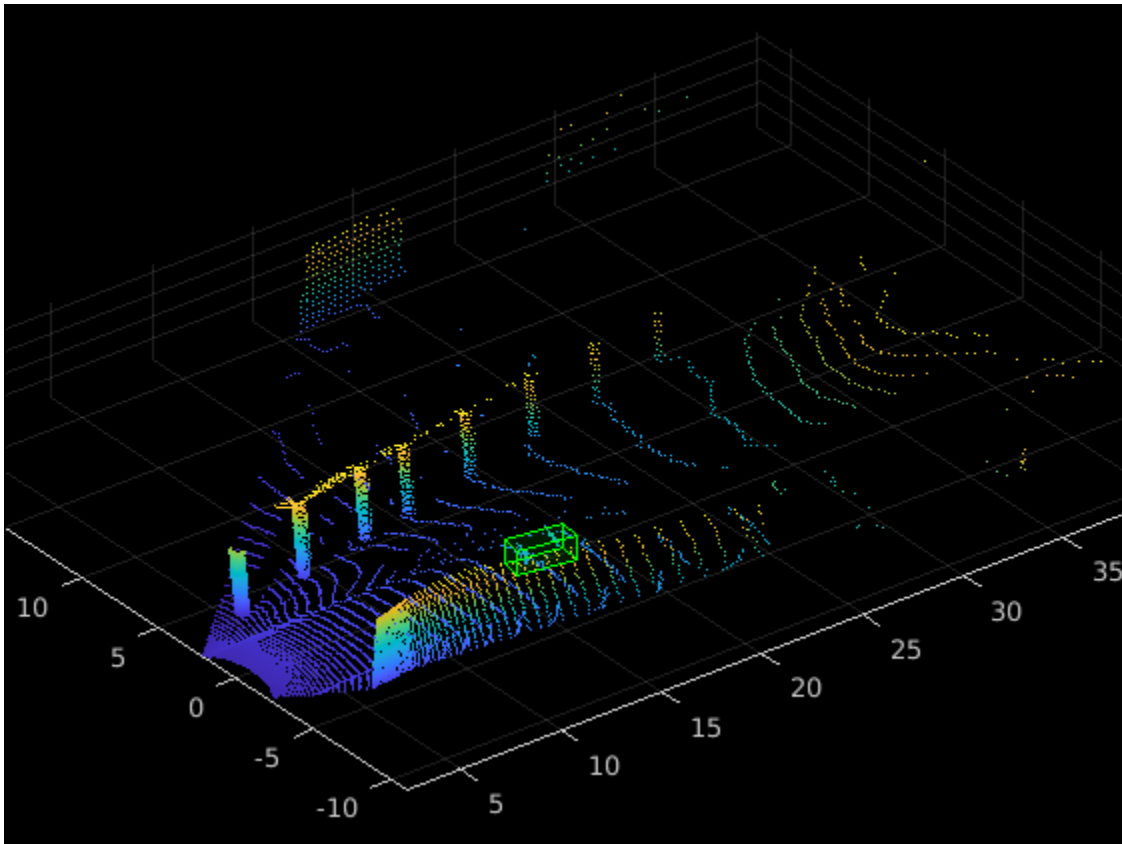
```
ptCloudTransformed = pctransform(ptCloud,tform);
```

Apply the same transformation to the 3-D bounding boxes.

```
gtLabelsTranformed = bboxwarp(gtLabels,tform,outView);
```

Display the scaled point cloud and the ground truth boxes.

```
figure;
ax2 = pcshow(ptCloudTransformed.Location);
showShape('cuboid',gtLabelsTranformed,'Parent',ax2,'Opacity',0.1, ...
          'Color','green','LineWidth',0.5);
zoom(ax2,2);
```



Random Translation of Point Cloud

Randomly translate the point cloud and the 3-D bounding boxes along the x -, y -, and z -axis from the specified range.

Create a transformation to translate the point cloud and 3-D bounding boxes.

```
tform = randomAffine3d('XTranslation',[0 0.2],...
                      'YTranslation',[0 0.2],...
                      'ZTranslation',[0 0.1]);
```

Apply the transformation to the point cloud.

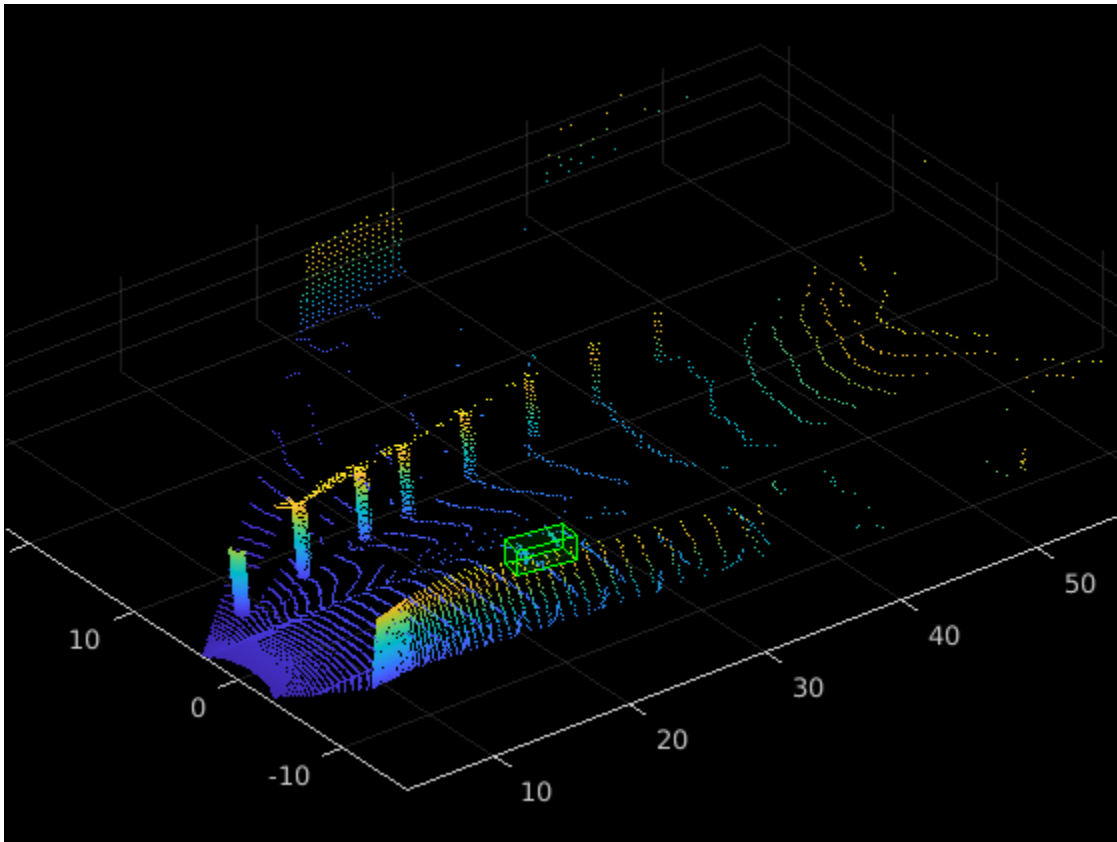
```
ptCloudTransformed = pctransform(ptCloud,tform);
```

Apply the same transformation to the 3-D bounding boxes.

```
gtLabelsTranformed = bboxwarp(gtLabels,tform,outView);
```

Display the translated point cloud and the ground truth boxes.

```
figure;
ax3 = pcshow(ptCloudTransformed.Location);
showShape('cuboid',gtLabelsTranformed,'Parent',ax3,'Opacity',0.1,'Color',...
          'green','LineWidth',0.5);
zoom(ax3,2);
```



Random Flipping Along Axis

Randomly flip the point cloud and the 3-D bounding boxes along the y-axis. Do not flip along the x-axis, as the bounding box annotations are provided in the camera field of view.

Create a transformation to flip the point cloud and 3-D bounding boxes.

```
tform = randomAffine3d('YReflection',true);
```

Apply the transformation to the point cloud.

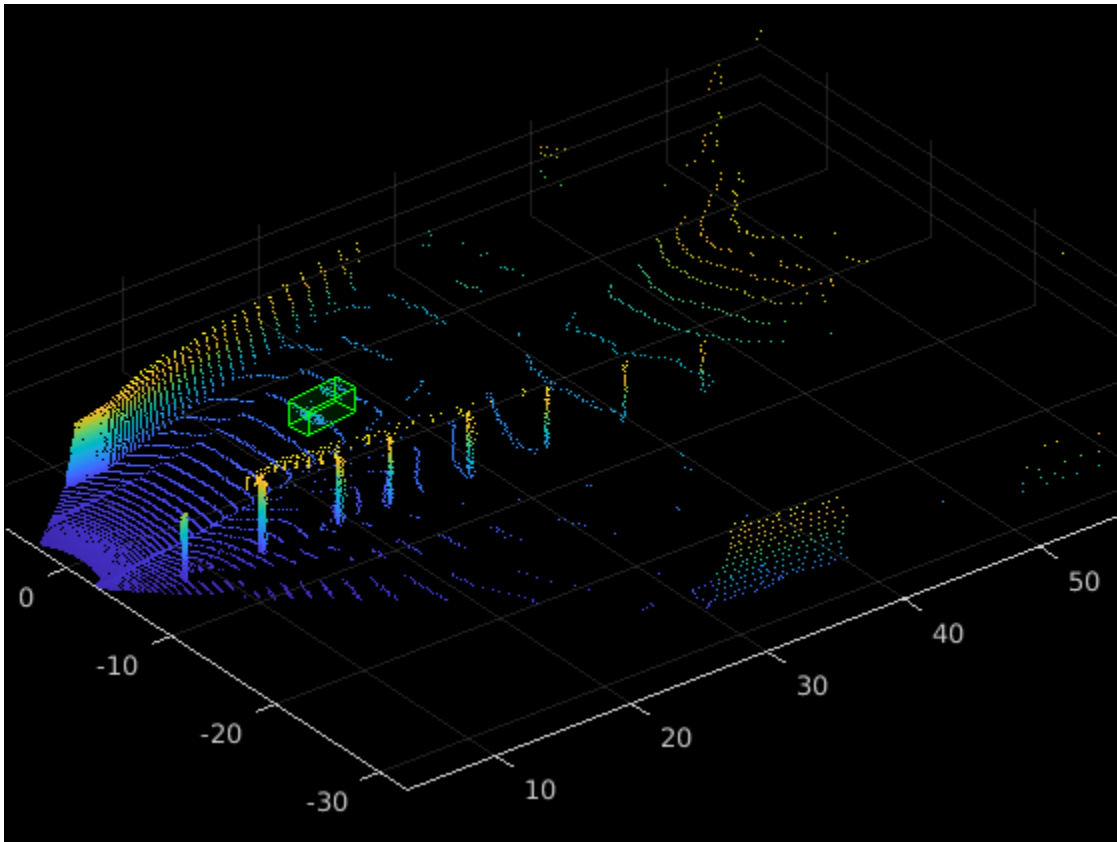
```
ptCloudTransformed = pctransform(ptCloud,tform);
```

Apply the same transformation to the 3-D bounding boxes using the helper function `flipBbox`, attached to this example as a supporting file.

```
gtLabels = flipBbox(gtLabels,tform);
```

Display the flipped point cloud and the ground truth boxes.

```
figure;
ax4 = pcshow(ptCloudTransformed.Location);
showShape('cuboid',gtLabels,'Parent',ax4,'Opacity',0.1, ...
          'Color','green','LineWidth',0.5);
zoom(ax4,2);
```



Ground Truth Data Augmentation

Ground truth data augmentation is a technique where during training, randomly selected ground truth boxes from other point clouds are introduced into the current training point cloud [1 on page 1-00]. Using this approach, you can increase the number of ground truth boxes per point cloud and simulate objects existing in different environments. To avoid physically impossible outcomes, you perform a collision test on the samples to be added and the ground truth boxes of the current point cloud. Use this augmentation technique when there is a class imbalance in the data set.

Use the `generateGTDataForAugmentation` and `groundTruthDataAugmentation` helper functions, attached to this example as supporting files, to randomly add a fixed number of objects to the point cloud from the car class.

```
trainData = lds.readall;
gtData = generateGTDataForAugmentation(trainData,trainLabels);
```

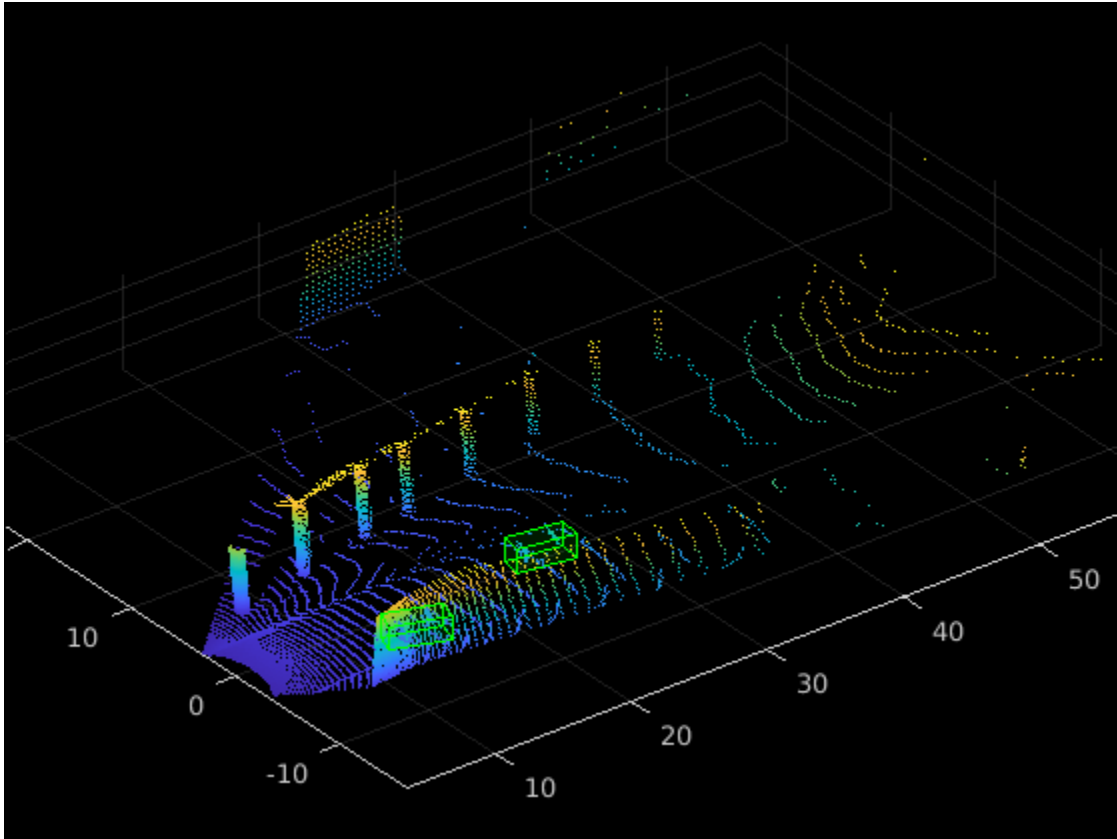
Define the number of ground truth boxes to add.

```
numSamplesToAdd = 3;
cdsAugmented = transform(cds,@(x) groundTruthDataAugmentation(x,gtData,numSamplesToAdd));
```

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

```
augData = read(cdsAugmented);
augptCld = augData{1,1};
augLabels = augData{1,2};
figure;
```

```
ax5 = pcshow(augptCld(:,1:3));
showShape('cuboid',augLabels,'Parent',ax5,'Opacity',0.1, ...
          'Color','green','LineWidth',0.5);
zoom(ax5,2);
```



Local Data Augmentation

Local data augmentation applies augmentation only to the points inside the ground truth boxes, not to the entire point cloud [1 on page 1-0]. Local data augmentation can be used when you want to apply the transformations only to the points inside the ground truth boxes. The rest of the point cloud remains the same.

Read the point cloud and corresponding ground truth label.

```
inputData = read(cds);
ptCloud = inputData{1,1};
gtLabels = inputData{1,2};
gtLabelsTransformed = zeros(size(gtLabels));
for i = 1:size(gtLabels,1)
    labelParams = gtLabels(i,:);
    centroidLoc = labelParams(1,1:3);
    model = cuboidModel(labelParams);
    indices = findPointsInsideCuboid(model, ptCloud);
    numPointsInside = size(indices,1);
```

Segregate the ground truth points from the original point cloud using the helper function `removeIndicesFromPointCloud`, attached to this example as a supporting file.


```

updatedPtCloud = removeIndicesFromPtCloud(ptCloud,indices);
cubPtCloud = select(ptCloud,indices);

% Shift the segregated point cloud to the origin.
numPoints = cubPtCloud.Count;
shiftRange = -1.*repmat(centroidLoc,[numPoints 1]);
cubPtCloud = pctransform(cubPtCloud,shiftRange);

% Define the minimum and maximum yaw angles for rotation.
minYawAngle = -45;
maxYawAngle = 45;

% Calculate a random angle from the specified yaw angle range.
theta = minYawAngle + rand*(maxYawAngle - minYawAngle);

% Create a transformation that rotates, translates, and scales the
% point clouds and 3-D bounding boxes.
tform = randomAffine3d('Rotation',@() deal([0,0,1],theta),...
    'Scale',[0.95,1.05],...
    'XTranslation',[0,0.2],...
    'YTranslation',[0,0.2],...
    'ZTranslation',[0,0.1]);

% Apply the transformation to the 3-D bounding box.
labelParams(1,1:3) = labelParams(1,1:3) - centroidLoc;
labelParamsTransformed = bboxwarp(labelParams,tform,outView);

% Calculate the overlap ratio between the transformed box and the
% original ground truth boxes by converting them to rotated rectangle
% format, defined as [xcenter,ycenter,width,height,yaw].
overlapRatio = bboxOverlapRatio(labelParamsTransformed(:,[1,2,4,5,9]), ...
    gtLabels(:,[1,2,4,5,9]));
[maxOverlapRatio, maxOverlapIdx] = max(overlapRatio);

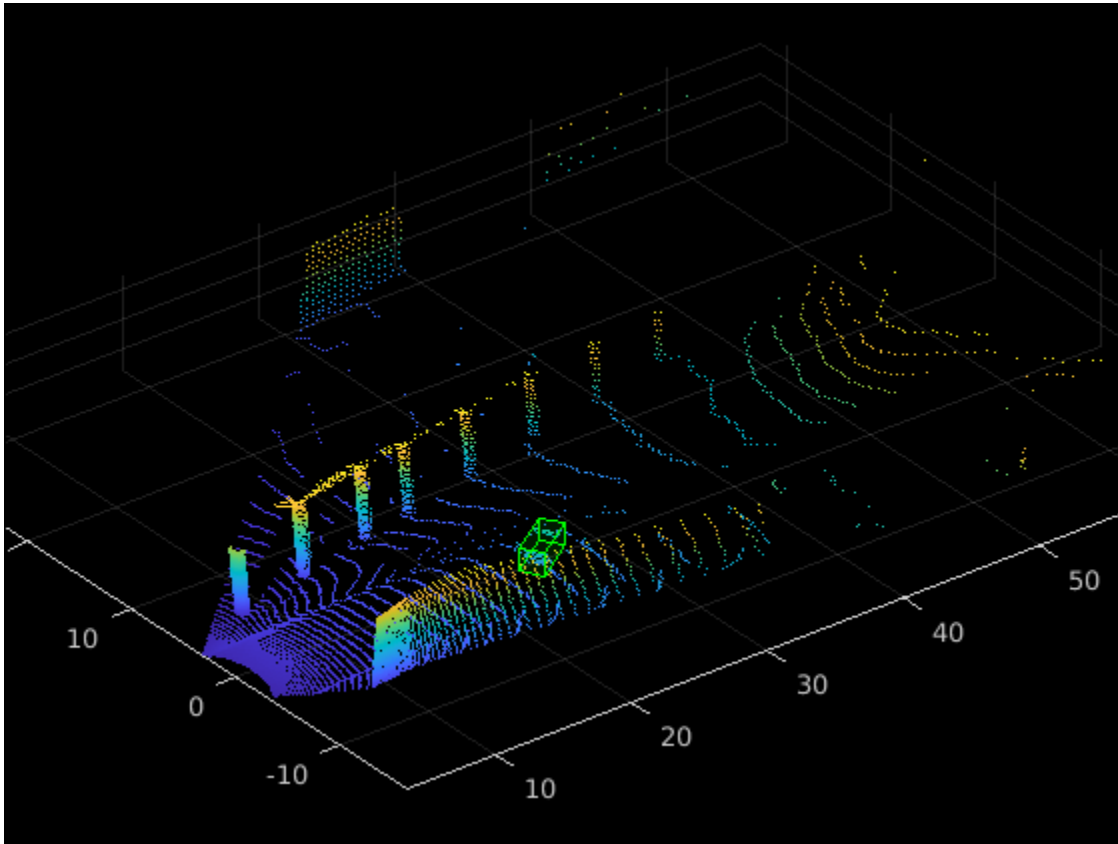
% Check to see if any transformed boxes overlap with the ground truth
% boxes.
if (maxOverlapRatio > 0) && (maxOverlapIdx ~= i)
    shiftRange = -1.*shiftRange;
    cubPtCloud = pctransform(cubPtCloud,shiftRange);
    updatedPtCloud = pccat([updatedPtCloud,cubPtCloud]);
    gtLabelsTransformed(i,1) = labelParams;
else
    cubPtCloudTransformed = pctransform(cubPtCloud,tform);
    shiftRange = -1.*shiftRange;
    cubPtCloudTransformed = pctransform(cubPtCloudTransformed,shiftRange);
    updatedPtCloud = pccat([updatedPtCloud,cubPtCloudTransformed]);
    gtLabelsTransformed(i,:) = labelParamsTransformed;
end
gtLabelsTransformed(i,1:3) = gtLabelsTransformed(i,1:3) + centroidLoc;
ptCloud = updatedPtCloud;
end

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

figure;
ax6 = pcshow(updatedPtCloud.Location);
showShape('cuboid',gtLabelsTransformed,'Parent',ax6,'Opacity',0.1, ...

```

```
    'Color','green','LineWidth',0.5);  
zoom(ax6,2);
```



```
reset(cds);
```

References

[1] Hahner, Martin, Dengxin Dai, Alexander Liniger, and Luc Van Gool. "Quantifying Data Augmentation for LiDAR Based 3D Object Detection." Preprint, submitted April 3, 2020. <https://arxiv.org/abs/2004.01643>.

Unorganized to Organized Conversion of Point Clouds Using Spherical Projection

This example shows how to convert unorganized point clouds to organized format using spherical projection.

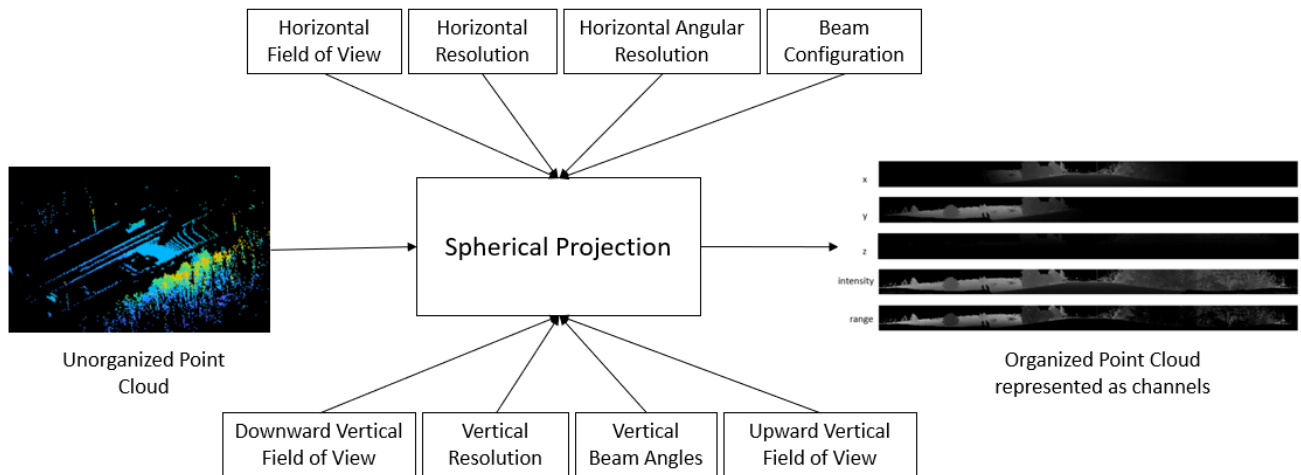
Introduction

A 3-D lidar point cloud is usually represented as a set of Cartesian coordinates (x, y, z) . The point cloud can also contain additional information such as intensity, and RGB values. Unlike the distribution of image pixels, the distribution of a lidar point cloud is usually sparse and irregular. Processing such sparse data is inefficient. To obtain a compact representation, you project lidar point clouds onto a sphere to create a dense, grid-based representation known as organized representation [1 on page 1-0]. To learn more about the differences between organized and unorganized point clouds, see “Lidar Processing Overview” on page 3-2. Ground plane extraction and key point detector methods require organized point clouds. Additionally, you must convert your point cloud to organized format if you want to use most deep learning segmentation networks, including SqueezeSegV1, SqueezeSegV2, RangeNet++ [2 on page 1-0], and SalsaNext [3 on page 1-0]. For an example showing how to use deep learning with an organized point cloud see “Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network” on page 1-75 example.

Lidar Sensor Parameters

To convert an unorganized point cloud to organized format using spherical projection, you must specify the parameters of the lidar sensor used to create the point cloud. Determine which parameters to specify by referring to the datasheet for your sensor. You can specify the following parameters.

- Beam configuration — 'uniform' or 'gradient'. Specify 'uniform' if the beams have equal spacing. Specify 'gradient' if the beams at the horizon are tightly packed, and those toward the top and bottom of the sensor field of view are more spaced out.
- Vertical resolution — Number of channels in the vertical direction, that is, the number of lasers. Typical values are 32, and 64.
- Vertical beam angles — Angular position of each vertical channel. You must specify this parameter when beam configuration is 'gradient'.
- Upward vertical field of view — Field of view in the vertical direction above the horizon (in degrees).
- Downward vertical field of view — Field of view in the vertical direction below the horizon (in degrees).
- Horizontal resolution — Number of channels in horizontal direction. Typical values are 512, and 1024.
- Horizontal angular resolution — The angular resolution between each channel along horizontal direction. You must specify this parameter when horizontal resolution is not mentioned in the datasheet.
- Horizontal field of view — Field of view covered in the horizontal direction (in degrees). In most cases, this value is 360 degrees.



Ouster OS-1 Sensor

Read the point cloud using the `pcread` function.

```
fileName = fullfile(matlabroot, 'examples', 'deeplearning_shared', 'data', 'ousterLidarDrivingData.p
ptCloud = pcread(fileName);
```

Check the size of the sample point cloud. If the point cloud coordinates are in the form, M -by- N -by-3, the point cloud is an organized point cloud.

```
disp(size(ptCloud.Location))
        64         1024          3
```

Convert the point cloud to unorganized format using `removeInvalidPoints` function. The coordinates of an unorganized point cloud are in the form M -by-3.

```
ptCloudUnOrg = removeInvalidPoints(ptCloud);
disp(size(ptCloudUnOrg.Location))
        65536          3
```

The point cloud data was collected from an Ouster OS1 Gen1 sensor. Specify the following parameters, which are given by the device datasheet [4 on page 1-0].

```
vResolution = 64;
hResolution = 1024;
vFOVUp = 16.6;
vFOVDown = -16.6;
hFOV = 360;
beamConfig = 'Uniform';
```

Calculate the beam angles along the horizontal and vertical directions.

```
if strcmp(beamConfig, 'Uniform')
    vbeamAngles = linspace(vFOVUp, vFOVDown, vResolution);
end
hbeamAngles = linspace(0, hFOV, hResolution);
```

Convert the unorganized point cloud to organized format using the `convertUnorgToOrg` on page 1-0 helper function, defined at the end of the example.

```
ptCloudOrg = convertUnorgToOrg(ptCloudUnOrg,vResolution,hResolution,vbeamAngles,hbeamAngles);
```

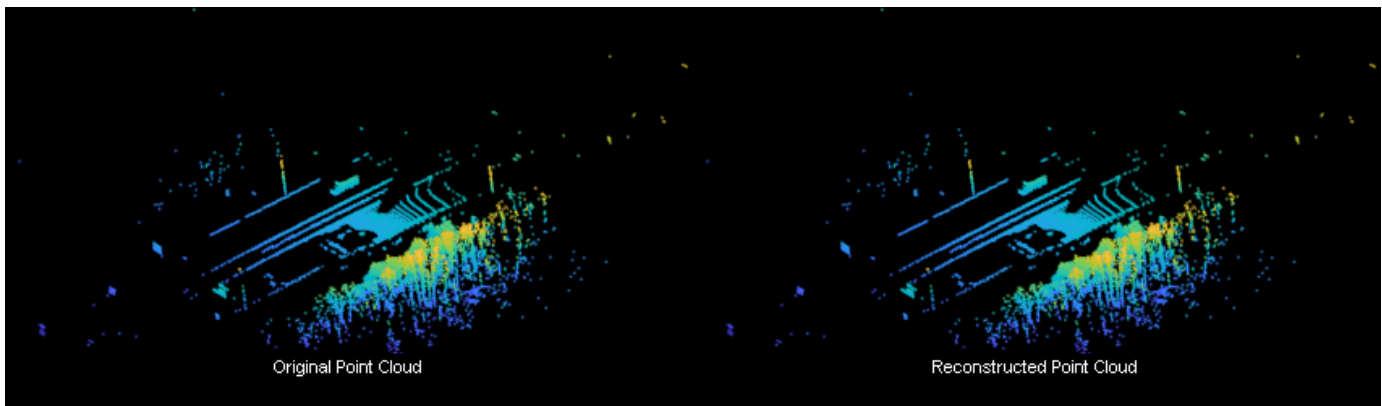
Display the intensity channel of the original and reconstructed organized point clouds.

```
figure
montage({uint8(ptCloud.Intensity),uint8(ptCloudOrg.Intensity)});
title("Intensity Channel of Original Point Cloud(Top) vs. Reconstructed Organized Point Cloud(Bottom)
```

Intensity Channel of Original Point Cloud(Top) vs. Reconstructed Organized Point Cloud(Bottom)

Display both the original organized point cloud and the reconstructed organized point cloud using the `helperShowUnorgAndOrgPair` helper function, attached to this example as a supporting file.

```
display1 = helperShowUnorgAndOrgPair();
display1.plotLidarScan(ptCloudUnOrg,ptCloudOrg,3.5);
```



Velodyne Sensor

Read the point cloud using the `pcread` function.

```
ptCloudUnOrg = pcread('HDL64LidarData.pcd');
```

The point cloud data is collected from the Velodyne HDL-64 sensor. Specify the following parameters, which are given by the device datasheet [5 on page 1-0].

```
vResolution = 64;
hResolution = 1024;
vFOVUp = 2;
vFOVDown = -24.9;
hFOV = 360;
beamConfig = 'Uniform';
```

Calculate the beam angles along the horizontal and vertical directions.

```

if strcmp(beamConfig,'Uniform')
    vbeamAngles = linspace(vFOVUp,vFOVDown,vResolution);
end
hbeamAngles = linspace(0,hFOV,hResolution);

```

Convert the unorganized point cloud to organized format using the `convertUnorgToOrg` on page 1-0 helper function, defined at the end of the example.

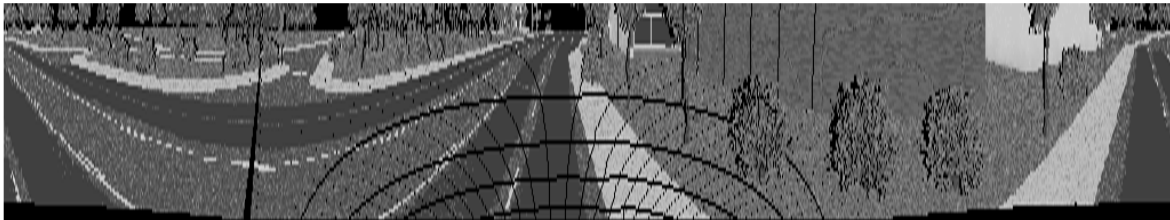
```
ptCloudOrg = convertUnorgToOrg(ptCloudUnOrg,vResolution,hResolution,vbeamAngles,hbeamAngles);
```

Display the intensity channel of the reconstructed organized point cloud. Replace NaNs with zeros and resize the image for better visualization.

```

intensityChannel = ptCloudOrg.Intensity;
intensityChannel(isnan(intensityChannel)) = 0;
figure
intensityChannel = imresize(intensityChannel,'Scale',[3 1]);
imshow(intensityChannel);

```

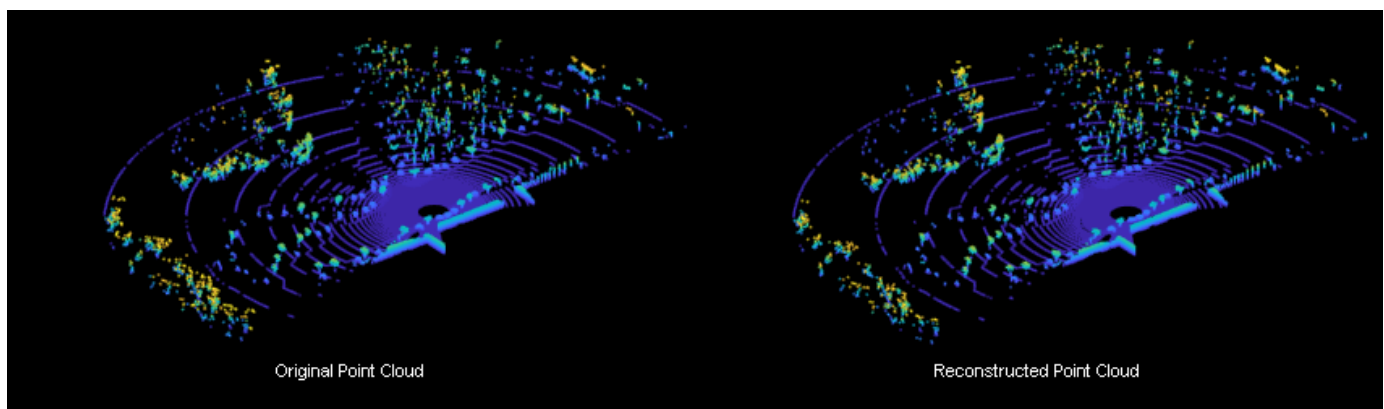


Display both the original organized point cloud and the reconstructed organized point cloud using the `helperShowUnorgAndOrgPair` helper function, attached to this example as a supporting file.

```

display2 = helperShowUnorgAndOrgPair();
display2.plotLidarScan(ptCloudUnOrg,ptCloudOrg,2.5);

```



Pandar Sensor

Read the point cloud using the `pcread` function. The point cloud is obtained from [6 on page 1-0].

```
ptCloudUnOrg = pcread('Pandar64LidarData.pcd');
```

The point cloud data is collected using a Pandar-64 sensor. Specify the following parameters, which are given by the device datasheet [7 on page 1-0].

```
vResolution = 64;
hAngResolution = 0.2;
hFOV = 360;
beamConfig = 'gradient';
```

The beam configuration is 'gradient', meaning that the beam spacing is not uniform. Specify the beam angle values along the vertical direction, which are given by the datasheet.

```
vbeamAngles = [15.0000  11.0000   8.0000   5.0000   3.0000   2.0000   1.8333   1.6667   :
                0.5000   0.3333   0.1667   0          -0.1667  -0.3333  -0.5000  -0.6667  -0.8333
               -1.8333  -2.0000  -2.1667  -2.3333  -2.5000  -2.6667  -2.8333  -3.0000  -3.1667
               -4.1667  -4.3333  -4.5000  -4.6667  -4.8333  -5.0000  -5.1667  -5.3333  -5.5000
               -9.0000 -10.0000 -11.0000 -12.0000 -13.0000 -14.0000 -19.0000 -25.0000];
```

Calculate the horizontal beam angles.

```
hResolution = round(360/hAngResolution);
hbeamAngles = linspace(0,hFOV,hResolution);
```

Convert the unorganized point cloud to organized format using the `convertUnorgToOrg` on page 1-0 helper function, defined at the end of the example.

```
ptCloudOrg = convertUnorgToOrg(ptCloudUnOrg,vResolution,hResolution,vbeamAngles,hbeamAngles);
```

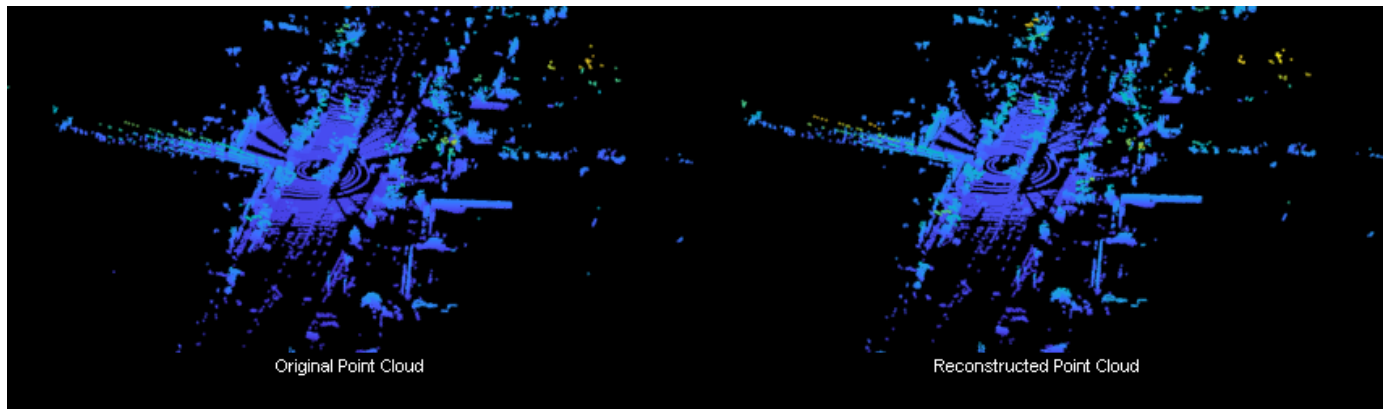
Display the intensity channel of the reconstructed organized point cloud. Replace NaNs with zeros and resize the image. Use `histeq` for better visualization.

```
intensityChannel = ptCloudOrg.Intensity;
intensityChannel(isnan(intensityChannel)) = 0;
figure
intensityChannel = imresize(intensityChannel,'Scale',[3 1]);
histeq(intensityChannel./max(intensityChannel(:)));
```



Display both the original organized point cloud and the reconstructed organized point cloud using the `helperShowUnorgAndOrgPair` helper function, attached to this example as a supporting file.

```
display3 = helperShowUnorgAndOrgPair();
display3.plotLidarScan(ptCloudUnOrg,ptCloudOrg,4);
```



Helper Functions

Use the `convertUnorgToOrg` function to find pixel coordinates of the projection image for all (x, y, z) points. The function follows these steps.

- 1 Calculate the pitch and yaw angles for every point of the point cloud.
- 2 Calculate the row and column indices for each point based on the beam, pitch, and yaw angles.
- 3 Create an organized point cloud based on the row and column indices.

```
function ptCloudOrganized = convertUnorgToOrg(ptCloud,vResolution,hResolution,vbeamAngles,hbeamAngles)

    locations = ptCloud.Location;

    if ~isempty(ptCloud.Intensity)
        intensity = ptCloud.Intensity;
    else
        intensity = zeros(size(ptCloud.Location,1),1);
    end

    % Calculate the radial distance for every point.
    r = sqrt(locations(:,1).^2 + locations(:,2).^2);
    r(r==0) = 1e-6;

    % Calculate the pitch and yaw angles for each point in the point cloud.
    pitch = atan2d(locations(:,3),r);
    yaw = atan2d(locations(:,2),locations(:,1));

    % Shift the range of the yaw angle from [-pi,pi] to [0,2*pi].
    yaw = 180-yaw;

    % Calculate the row indices for all points based on the bin in which the pitch angle for each point.
    [~,~,rowIdx] = histcounts(pitch,flip(vbeamAngles));
    rowIdx(rowIdx == 0) = NaN;
    rowIdx = vResolution - rowIdx;

    % Calculate the column indices for all points based on the bin in which the yaw angle for each point.
    [~,~,colIdx] = histcounts(yaw,hbeamAngles);
    colIdx(colIdx == 0) = NaN;

    % Create a pseudo image and fill in the values with the corresponding location and intensity
    psuedoImage = NaN(vResolution,hResolution,4);
    for i = 1:size(locations,1)
```



```

    if ~isnan(rowIdx(i,1)) && ~isnan(colIdx(i,1))
        pseduoImage(rowIdx(i,1),colIdx(i,1),1) = locations(i,1);
        pseduoImage(rowIdx(i,1),colIdx(i,1),2) = locations(i,2);
        pseduoImage(rowIdx(i,1),colIdx(i,1),3) = locations(i,3);
        pseduoImage(rowIdx(i,1),colIdx(i,1),4) = intensity(i,1);
    end
end

% Create a point cloud object from the locations and intensity.
if ~isempty(ptCloud.Intensity)
    ptCloudOrganized = pointCloud(pseduoImage(:,:,1:3), 'Intensity', pseduoImage(:,:,4));
else
    ptCloudOrganized = pointCloud(pseduoImage(:,:,1:3));
end
end

```

References

- [1] Wu, Bichen, Alvin Wan, Xiangyu Yue, and Kurt Keutzer. "SqueezeSeg: Convolutional Neural Nets with Recurrent CRF for Real-Time Road-Object Segmentation from 3D LiDAR Point Cloud." In 2018 *IEEE International Conference on Robotics and Automation (ICRA)*, 1887-93. Brisbane, QLD: IEEE, 2018. <https://doi.org/10.1109/ICRA.2018.8462926>.
- [2] Milioto, Andres, Ignacio Vizzo, Jens Behley, and Cyrill Stachniss. "RangeNet ++: Fast and Accurate LiDAR Semantic Segmentation." In 2019 *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 4213-20. Macau, China: IEEE, 2019. <https://doi.org/10.1109/IROS40897.2019.8967762>.
- [3] Cortinhal, Tiago, George Tzelepis, and Eren Erdal Aksoy. "SalsaNext: Fast, Uncertainty-Aware Semantic Segmentation of LiDAR Point Clouds for Autonomous Driving." *ArXiv:2003.03653 [Cs]*, July 9, 2020. <http://arxiv.org/abs/2003.03653>.
- [4] Kim, Jaden. "UCS - Ouster LiDAR." Accessed December 22, 2020. <https://ucssolution.com/OS1-Mid-range-lidar-sensor>.
- [5] Velodyne Lidar. "HDL-64E Durable Surround Lidar Sensor." Accessed December 22, 2020. <https://velodynelidar.com/products/hdl-64e/>.
- [6] "PandaSet Open Datasets - Scale." Accessed December 22, 2020. <https://scale.com/open-datasets/pandaset>.
- [7] "Pandar64 User Manual." Accessed December 22, 2020. https://hesaiweb2019.blob.core.chinacloudapi.cn/uploads/Pandar64_User's_Manual.pdf.

Lane Detection in 3-D Lidar Point Cloud

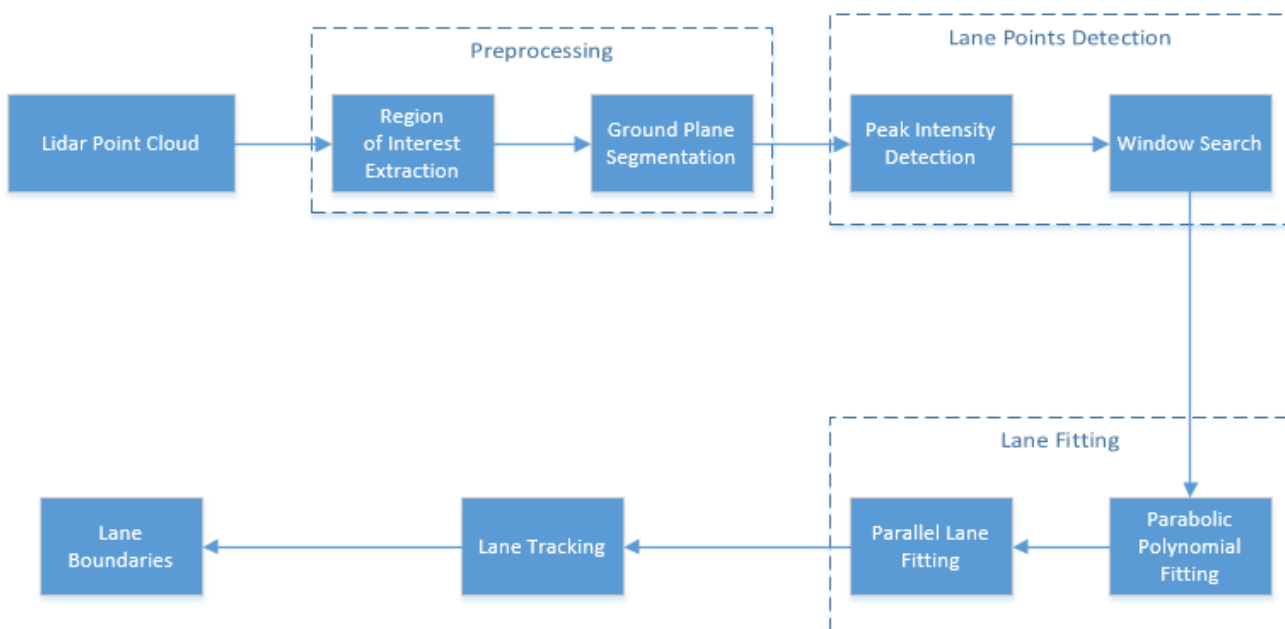
This example shows how to detect lanes in lidar point clouds. You can use the intensity values returned from lidar point clouds to detect ego vehicle lanes. You can further improve the lane detection by using a curve-fitting algorithm and tracking the curve parameters. Lidar lane detection enables you to build complex workflows like lane keep assist, lane departure warning, and adaptive cruise control for autonomous driving. A test vehicle collects the lidar data using a lidar sensor mounted on its rooftop.

Introduction

Lane detection in lidar involves detection of the immediate left and right lanes, also known as ego vehicle lanes, with respect to the lidar sensor. It involves the following steps:

- Region of interest extraction
- Ground plane segmentation
- Peak intensity detection
- Lane detection using window search
- Parabolic polynomial fitting
- Parallel lane fitting
- Lane tracking

This flowchart gives an overview of the workflow presented in this example.



The advantages of using lidar data for lane detection are :

- Lidar point clouds give a better 3-D representation of the road surface than image data, thus reducing the required calibration parameters to find the bird's-eye view.

- Lidar is more robust against adverse climatic conditions than image-based detection.
- Lidar data has a centimeter level of accuracy, leading to accurate lane localization.

Download and Prepare Lidar Data Set

The lidar data used in this example has been collected using the Ouster OS1-64 channel lidar sensor, producing high-resolution point clouds. This data set contains point clouds stored as a cell array of `pointCloud` object. Each 3-D point cloud consists of XYZ locations along with intensity information.

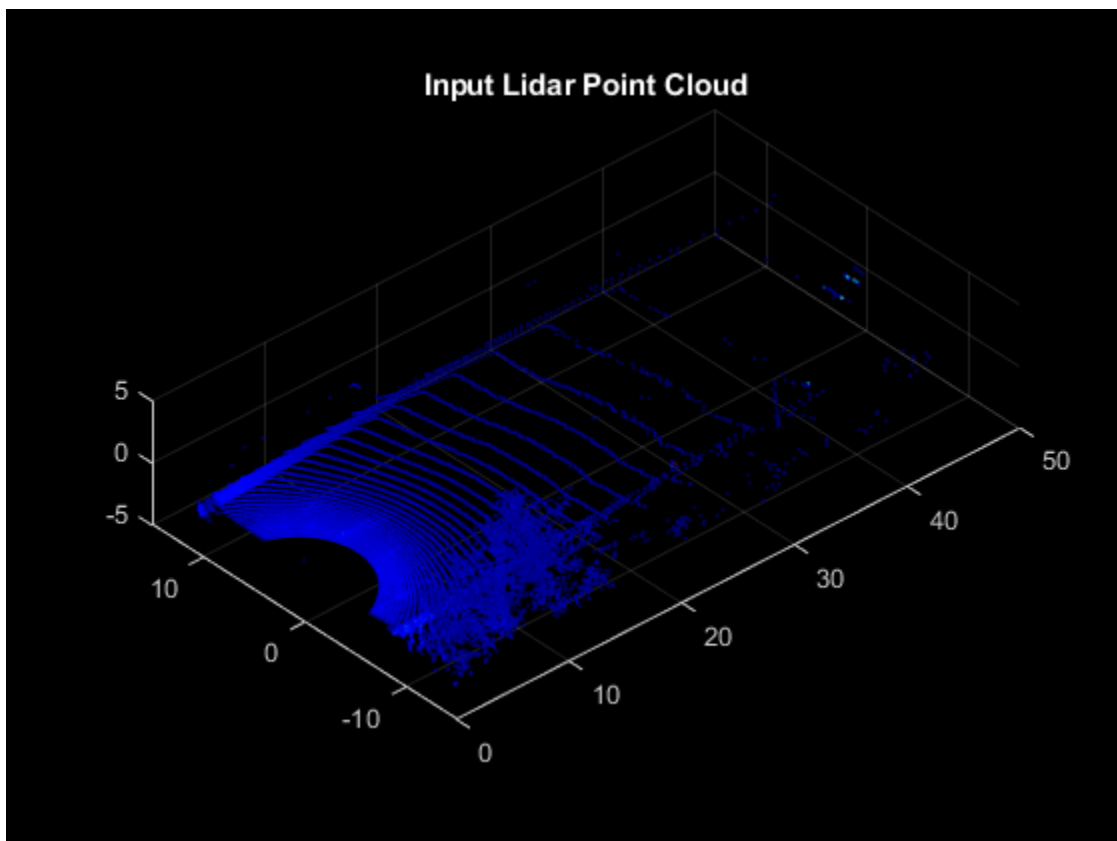
Note: Download time of the data depends on your internet connection. MATLAB will be temporarily unresponsive during the execution of this code block.

```
% Download lidar data
lidarData = helperGetDataset;
```

Selecting the first frame of the dataset for further processing.

```
% Select first frame
ptCloud = lidarData{1};

% Visualize input point cloud
figure
pcshow(ptCloud)
title('Input Lidar Point Cloud')
axis([0 50 -15 15 -5 5])
view([-42 35])
```



Preprocessing

To estimate the lane points, first preprocess the lidar point clouds. Preprocessing involves the following steps:

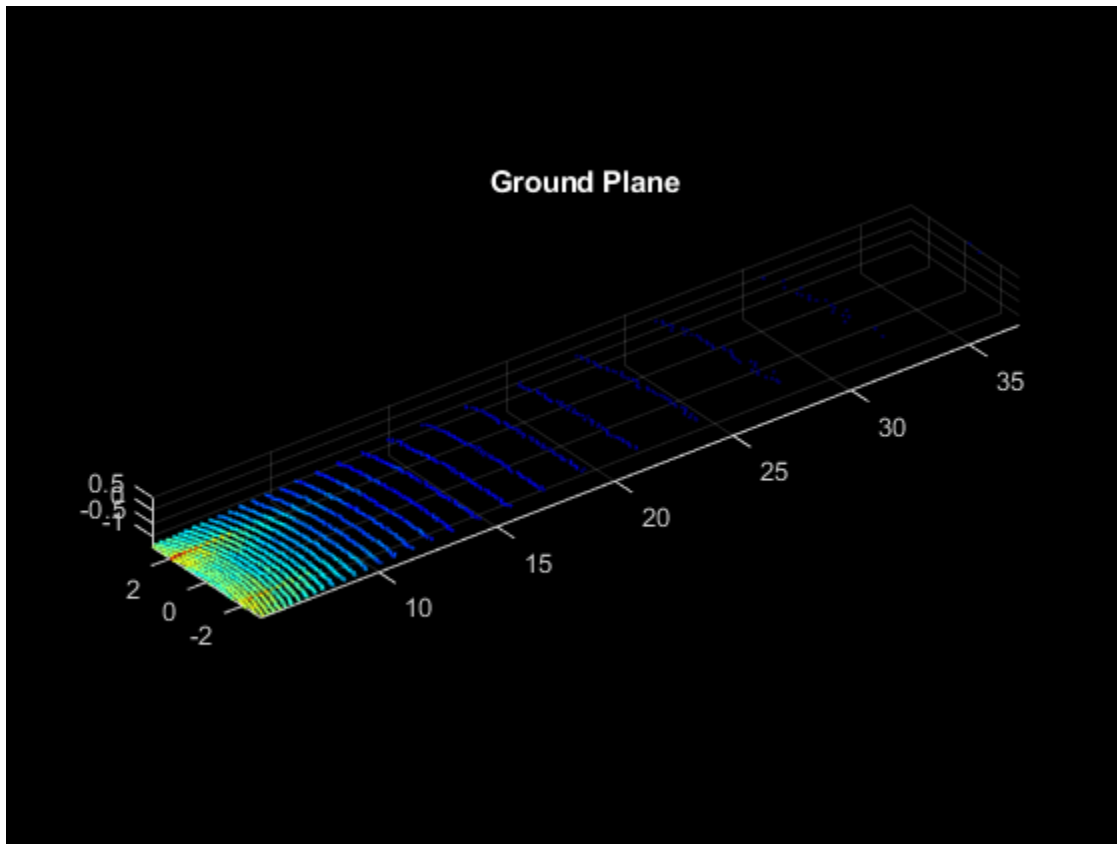
- Region of interest extraction
- Ground plane segmentation

```
% Define ROI in meters
xlim = [5 55];
ylim = [-3 3];
zlim = [-4 1];
roi = [xlim ylim zlim];

% Crop point cloud using ROI
indices = findPointsInROI(ptCloud,roi);
croppedPtCloud = select(ptCloud,indices);

% Remove ground plane
maxDistance = 0.1;
referenceVector = [0 0 1];
[model,inliers,outliers] = pcfitplane(croppedPtCloud,maxDistance,referenceVector);
groundPts = select(croppedPtCloud,inliers);

figure
pcshow(groundPts)
title('Ground Plane')
view(3)
```



Lane Point Detection

Detect lane points by using a sliding window search, where the initial estimates for the sliding windows are made using an intensity-based histogram.

Lane point detection consists primarily of these two steps:

- Peak intensity detection
- Window search

Peak Intensity Detection

Lane points in the lidar point cloud have a distinct distribution of intensities. Usually, these intensities occupy the upper region in the histogram distribution and appear as high peaks. Compute a histogram of intensities from the detected ground plane along the ego vehicle axis (positive X-axis). The `helperComputeHistogram` helper function creates a histogram of intensity points. Control the number of bins for the histogram by specifying the bin resolution.

```
histBinResolution = 0.2;
[histVal,yvals] = helperComputeHistogram(groundPts,histBinResolution);

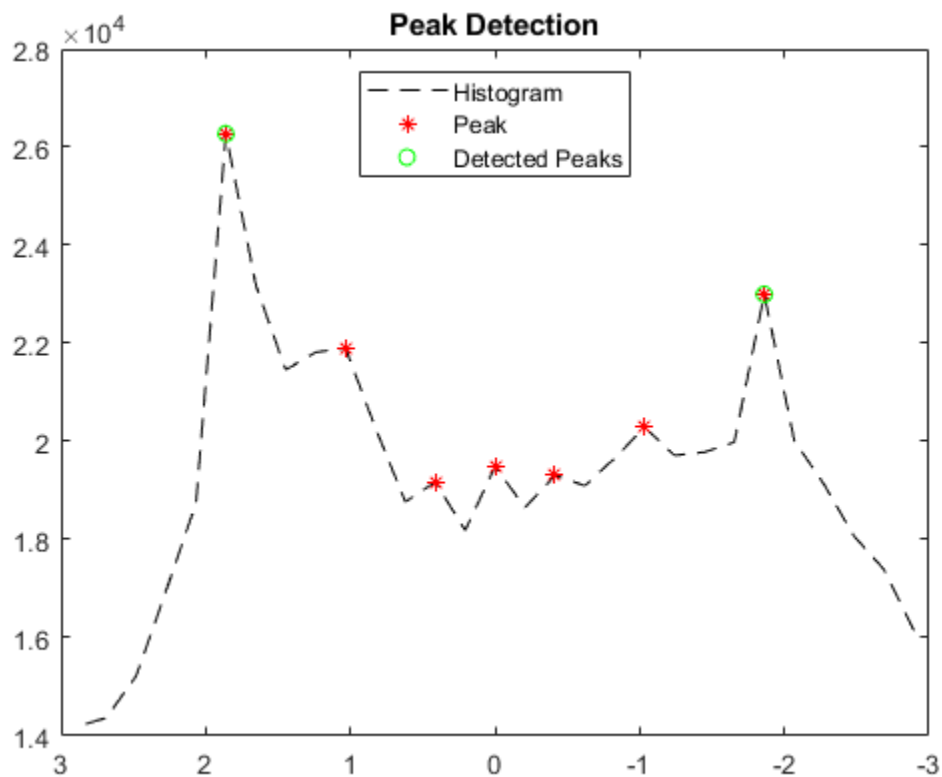
figure
plot(yvals,histVal,'--k')
set(gca,'XDir','reverse')
hold on
```

Obtain peaks in the histogram by using the `helperfindpeaks` helper function. Further filter the possible lane points based on lane width using the `helperInitialWindow` helper function.

```
[peaks,locs] = helperfindpeaks(histVal);
startYs = yvals(locs);

laneWidth = 4;
[startLanePoints,detectedPeaks] = helperInitialWindow(startYs,peaks,laneWidth);

plot(startYs,peaks,'*r')
plot(startLanePoints,detectedPeaks,'og')
legend('Histogram','Peak','Detected Peaks','Location','North')
title('Peak Detection')
hold off
```



Window Search

Window search is used to detect lane points by sliding the windows along the lane curvature direction. Window search consists of two steps:

- Window initialization
- Sliding window

Window Initialization

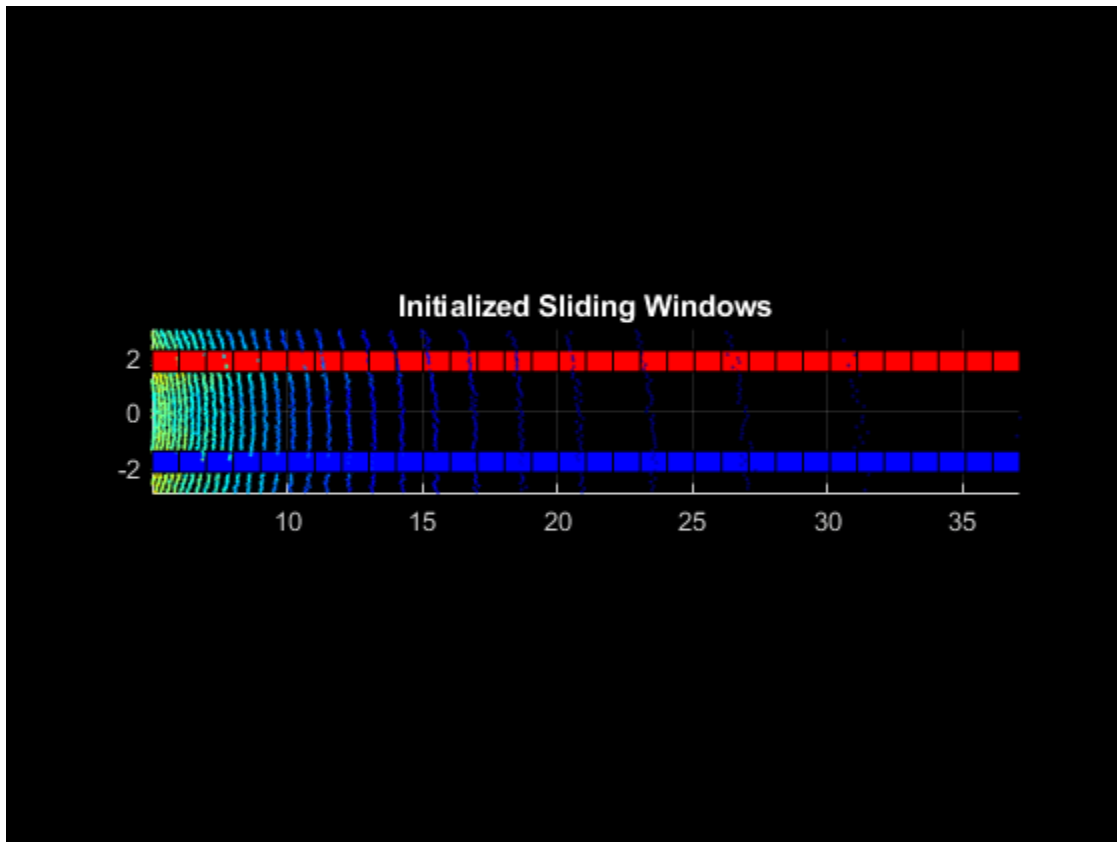
The detected peaks help in the initialization of the search window. Initialize the search windows as multiple bins with red and blue colors for left and right lanes respectively.

```

vBinRes = 1;
hBinRes = 0.8;
numVerticalBins = ceil((groundPts.XLimits(2) - groundPts.XLimits(1))/vBinRes);
laneStartX = linspace(groundPts.XLimits(1),groundPts.XLimits(2),numVerticalBins);

% Display bin windows
figure
pcshow(groundPts)
view(2)
helperDisplayBins(laneStartX,startLanePoints(1),hBinRes/2,groundPts,'red');
helperDisplayBins(laneStartX,startLanePoints(2),hBinRes/2,groundPts,'blue');
title('Initialized Sliding Windows')

```



Sliding Window

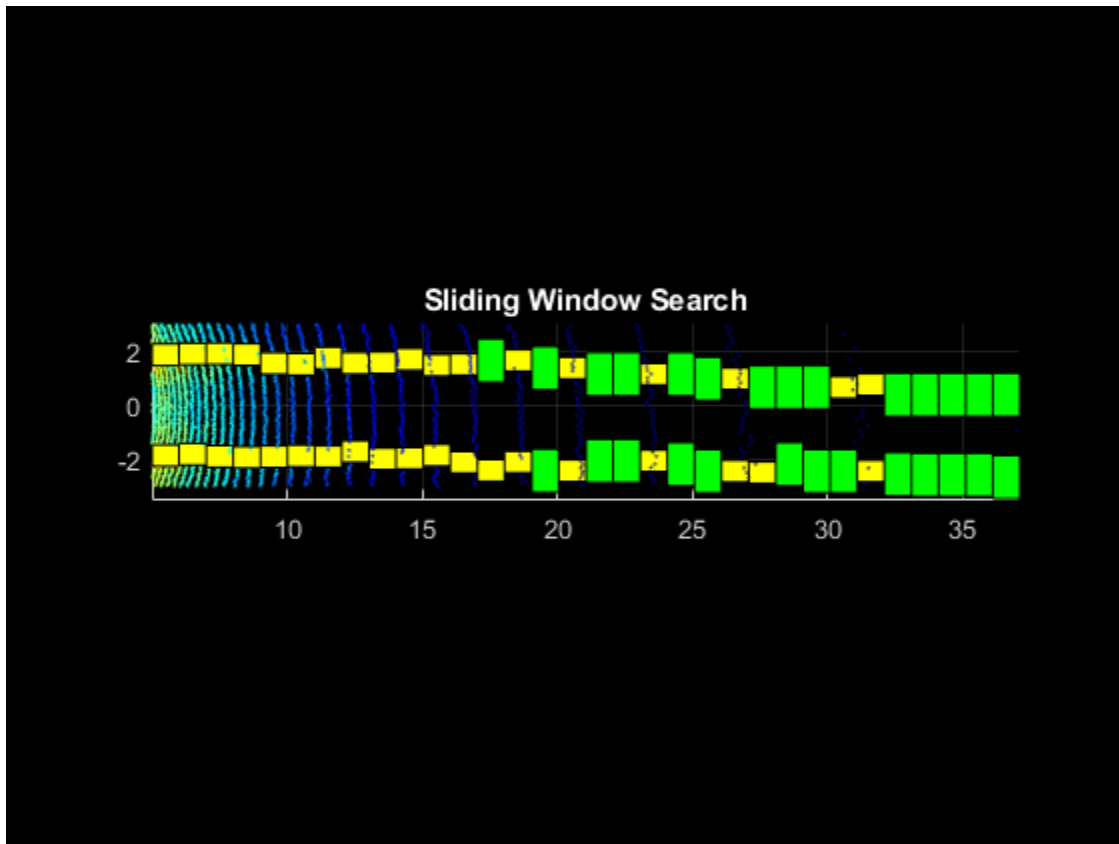
The sliding bins are initialized from the bin locations, and iteratively move along the ego vehicle (positive X-axis) direction. The `helperDetectLanes` helper function detects lane points and visualizes the sliding bins. Consecutive bins slide along the Y-axis based upon the intensity values inside the previous bin.

At regions where the lane points are missing, the function predicts the sliding bins using second-degree polynomial. This condition commonly arises when there are moving object's crossing the lanes. The sliding bins are yellow and the bins that are predicted using the polynomial are green.

```

display = true;
lanes = helperDetectLanes(groundPts,hBinRes, ...
    numVerticalBins,startLanePoints,display);

```

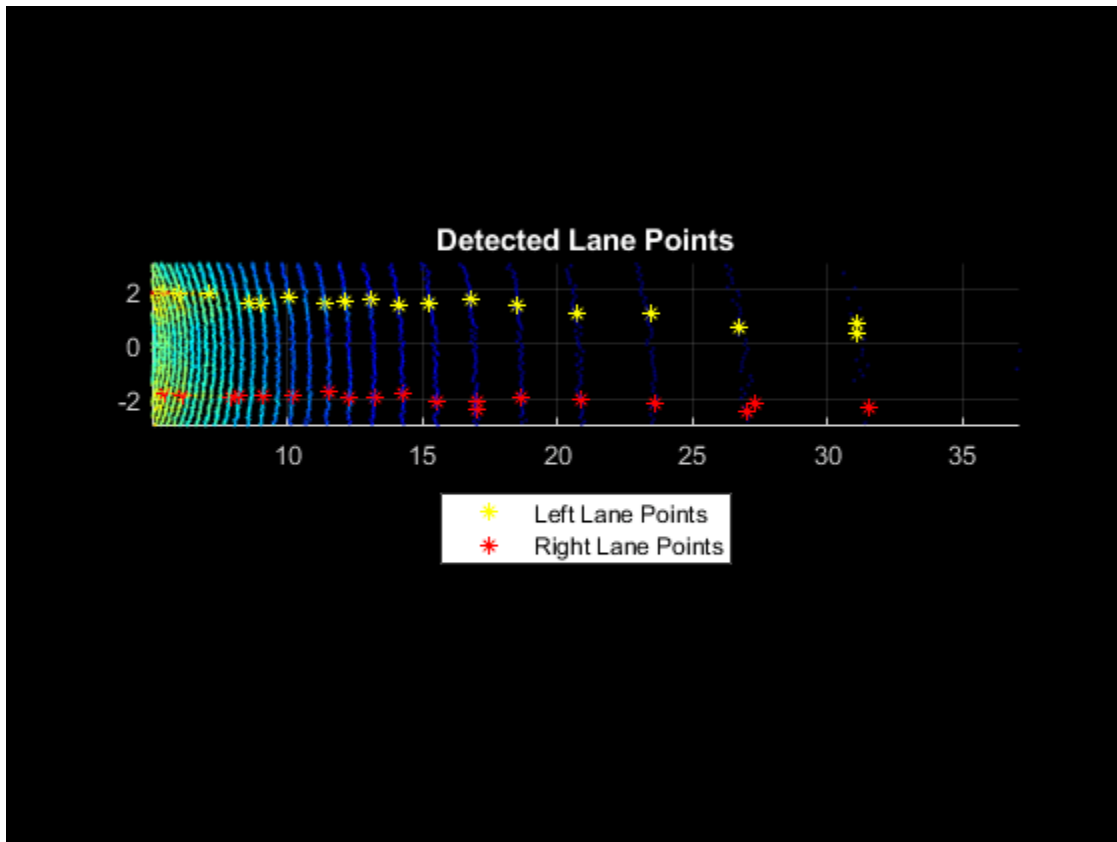


```

% Plot final lane points
lane1 = lanes{1};
lane2 = lanes{2};

figure
pcshow(groundPts)
title('Detected Lane Points')
hold on
p1 = plot3(lane1(:, 1),lane1(:, 2),lane1(:, 3),'*y');
p2 = plot3(lane2(:, 1),lane2(:, 2),lane2(:, 3),'*r');
hold off
view(2)
lgnd = legend([p1 p2],{'Left Lane Points','Right Lane Points'});
set(lgnd,'color','White','Location','southoutside')

```

Lane Fitting

Lane fitting involves estimating a polynomial curve on the detected lane points. These polynomials are used along with parallel lane constraint for lane fitting.

Parabolic Polynomial Fitting

The polynomial is fitted on X-Y points using a 2-degree polynomial represented as $ax^2 + bx + c$, where a , b , and c are polynomial parameters. To perform curve fitting, use the `helperFitPolynomial` helper function, which also handles outliers using the random sample consensus (RANSAC) algorithm. To estimate the 3-D lane points, extract the parameters from the plane model created during the preprocessing step. The plane model is represented as $ax + by + cz + d = 0$, where the Z-coordinate is unknown. Estimate the Z-coordinate by substituting the X- and Y-coordinates in the plane equation.

```
[P1,error1] = helperFitPolynomial(lane1(:,1:2),2,0.1);
[P2,error2] = helperFitPolynomial(lane2(:,1:2),2,0.1);

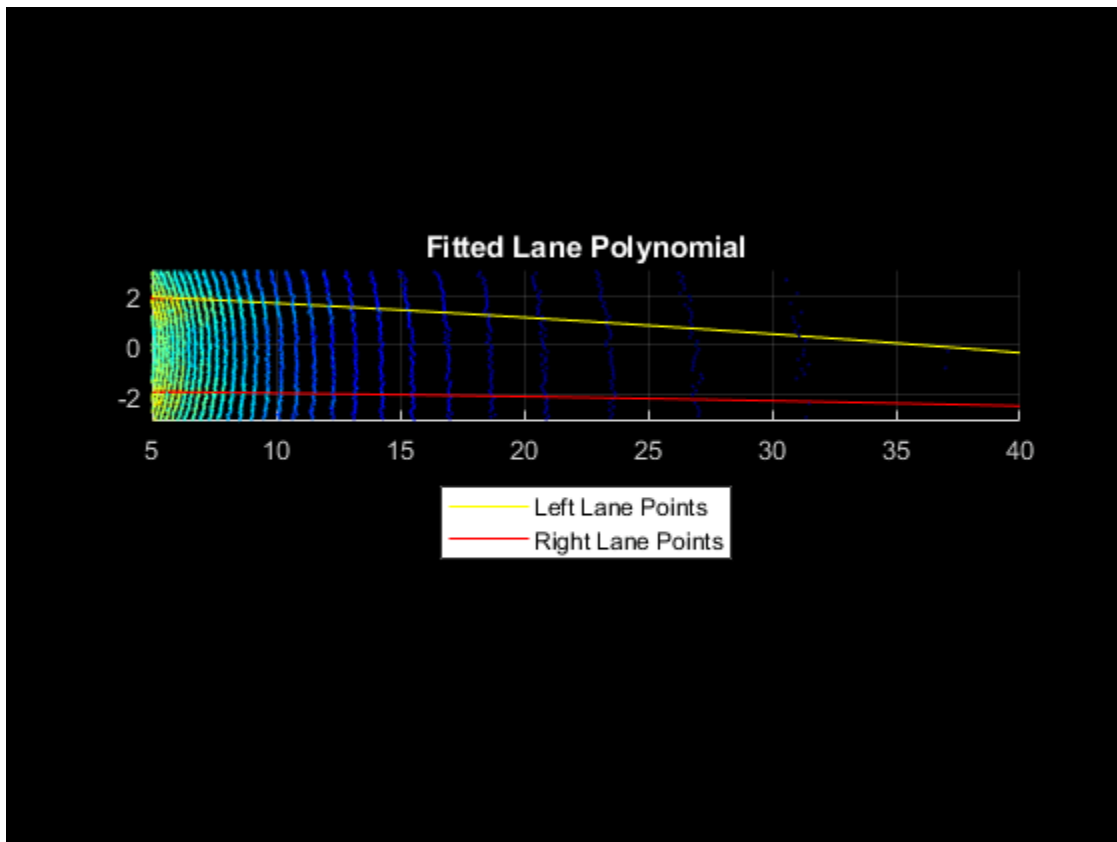
xval = linspace(5,40,80);
yval1 = polyval(P1,xval);
yval2 = polyval(P2,xval);

% Z-coordinate estimation
modelParams = model.Parameters;
zWorld1 = (-modelParams(1)*xval - modelParams(2)*yval1 - modelParams(4))/modelParams(3);
zWorld2 = (-modelParams(1)*xval - modelParams(2)*yval2 - modelParams(4))/modelParams(3);
```

```

% Visualize fitted lane
figure
pcshow(croppedPtCloud)
title('Fitted Lane Polynomial')
hold on
p1 = plot3(xval,yval1,zWorld1,'y','LineWidth',0.2);
p2 = plot3(xval,yval2,zWorld2,'r','LineWidth',0.2);
lgnd = legend([p1 p2],{'Left Lane Points','Right Lane Points'});
set(lgnd,'color','White','Location','southoutside')
view(2)
hold off

```



Parallel Lane Fitting

The lanes are usually parallel to each other along the road. To make the lane fitting robust, use this parallel constraint. When fitting the polynomials, the `helperFitPolynomial` helper function also computes the fitting error. Update the lanes having erroneous points with the new polynomial. Update this polynomial by shifting it along the Y-axis.

```

lane3d1 = [xval' yval1' zWorld1'];
lane3d2 = [xval' yval2' zWorld2'];

% Shift the polynomial with a high score along the Y-axis towards
% the polynomial with a low score
if error1 > error2
    lanePolynomial = P2;
    if lane3d1(1,2) > 0

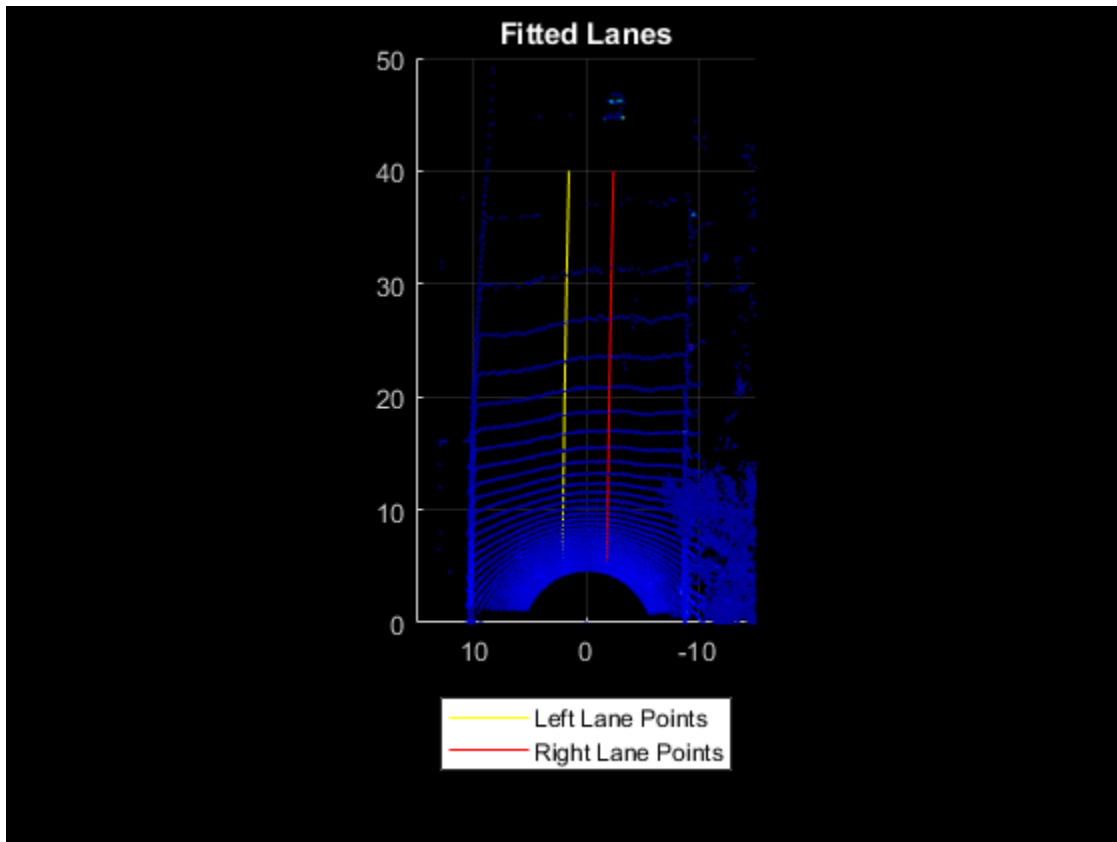
```

```

        lanePolynomial(3) = lane3d2(1,2) + laneWidth;
    else
        lanePolynomial(3) = lane3d2(1,2) - laneWidth;
    end
    lane3d1(:,2) = polyval(lanePolynomial, lane3d1(:,1));
    lanePolynomials = [lanePolynomial; P2];
else
    lanePolynomial = P1;
    if lane3d2(1,2) > 0
        lanePolynomial(3) = lane3d1(1,2) + laneWidth;
    else
        lanePolynomial(3) = lane3d1(1,2) - laneWidth;
    end
    lane3d2(:,2) = polyval(lanePolynomial, lane3d2(:,1));
    lanePolynomials = [P1; lanePolynomial]
end

% Visualize lanes after parallel fitting
figure
pcshow(ptCloud)
axis([0 50 -15 15 -5 5])
hold on
p1 = plot3(lane3d1(:,1), lane3d1(:,2), lane3d1(:,3), 'y', 'LineWidth', 0.2);
p2 = plot3(lane3d2(:,1), lane3d2(:,2), lane3d2(:,3), 'r', 'LineWidth', 0.2);
view([-90 90])
title('Fitted Lanes')
lgnd = legend([p1 p2], {'Left Lane Points', 'Right Lane Points'});
set(lgnd, 'color', 'White', "Location", "southoutside")
hold off

```



Lane Tracking

Lane tracking helps in stabilizing the lane curvature caused by sudden jerks and drifts. These changes can occur because of missing lane points, vehicles moving over the lanes, and erroneous lane detection. Lane tracking is a two-step process:

- Track lane polynomial parameters(a , b) to control the curvature of the polynomial.
- Track the start points coming from peak detection. This parameter is denoted as c in the polynomial.

These parameters are updated using a Kalman filter with a constant acceleration motion model. To initiate a Kalman filter, use the `configureKalmanFilter`.

```
% Initial values
curveInitialParameters = lanePolynomials(1,1:2);
driftInitialParameters = lanePolynomials(:,3)';
initialEstimateError = [1 1 1] * 1e-1;
motionNoise = [1 1 1] * 1e-7;
measurementNoise = 10;

% Configure Kalman filter
curveFilter = configureKalmanFilter('ConstantAcceleration', ...
    curveInitialParameters, initialEstimateError, motionNoise, measurementNoise);
driftFilter = configureKalmanFilter('ConstantAcceleration', ...
    driftInitialParameters, initialEstimateError, motionNoise, measurementNoise);
```

Loop Through Data

Loop through and process the lidar data by using the `helperLaneDetector` helper class. This helper class implements all previous steps, and also performs additional preprocessing to remove the vehicles from the point cloud. This ensures that the detected ground points are flat and the plane model is accurate. The class method `detectLanes` detects and extracts the lane points for the left and right lane as a two-element cell array, where the first element corresponds to the left lane and the second element to the right lane.

```
% Initialize the random number generator
rng(2020)
numFrames = numel(lidarData);
detector = helperLaneDetector('ROI',[5 40 -3 3 -4 1]);

% Turn on display
player = pcplayer([0 50],[-15 15],[-5 5]);

drift = zeros(numFrames,1);
filteredDrift = zeros(numFrames,1);
curveSmoothness = zeros(numFrames,1);
filteredCurveSmoothness = zeros(numFrames,1);
for i = 1:numFrames
    ptCloud = lidarData{i};

    % Detect lanes
    detectLanes(detector,ptCloud);

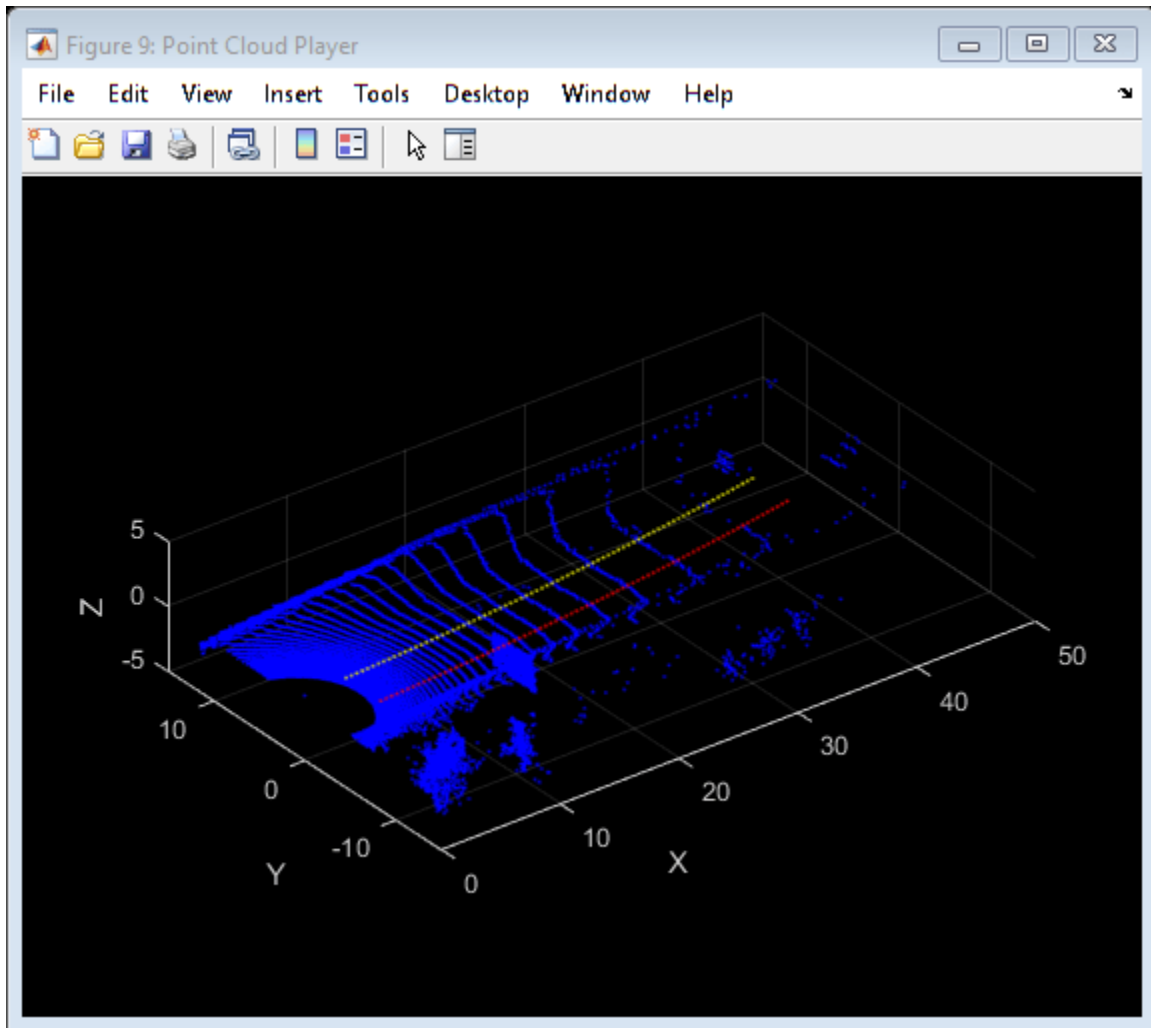
    % Predict polynomial from Kalman filter
    predict(curveFilter);
    predict(driftFilter);

    % Correct polynomial using Kalman filter
    lanePolynomials = detector.LanePolynomial;
    drift(i) = mean(lanePolynomials(:,3));
    curveSmoothness(i) = mean(lanePolynomials(:,1));
    updatedCurveParams = correct(curveFilter, lanePolynomials(1,1:2));
    updatedDriftParams = correct(driftFilter, lanePolynomials(:,3)');

    % Update lane polynomials
    updatedLanePolynomials = [repmat(updatedCurveParams,[2 1]),updatedDriftParams'];

    % Estimate new lane points with updated polynomial
    lanes = updateLanePolynomial(detector,updatedLanePolynomials);
    filteredDrift(i) = mean(updatedLanePolynomials(:,3));
    filteredCurveSmoothness(i) = mean(updatedLanePolynomials(:,1));

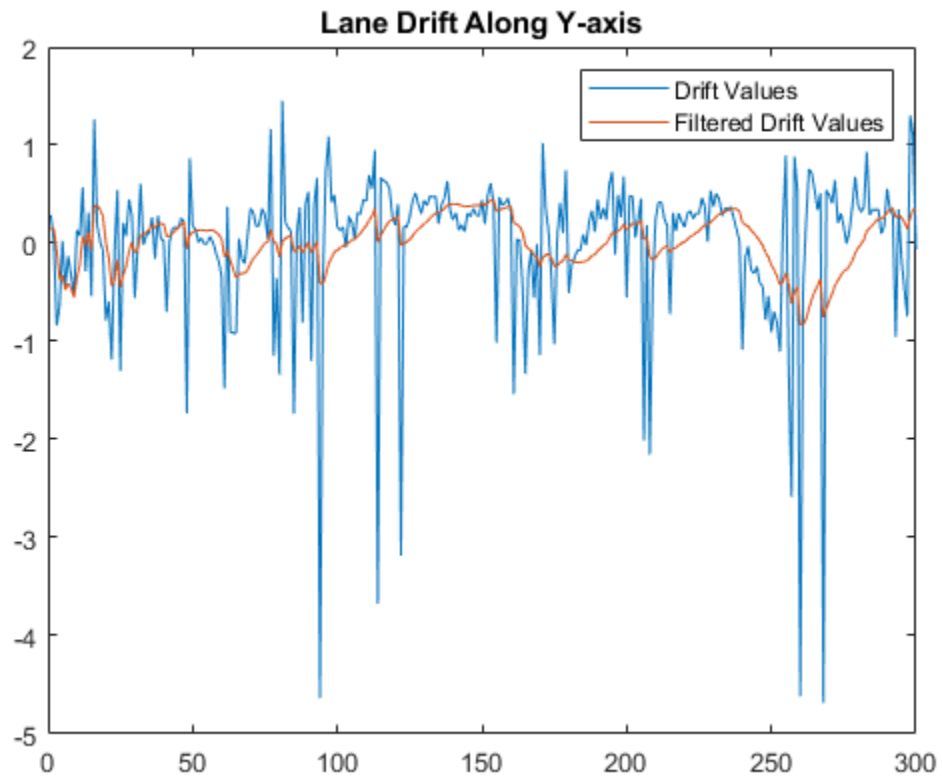
    % Visualize lanes after parallel fitting
    ptCloud.Color = uint8(repmat([0 0 255],ptCloud.Count,1));
    lane3dPc1 = pointCloud(lanes{1});
    lane3dPc1.Color = uint8(repmat([255 0 0],lane3dPc1.Count,1));
    lanePc = pccat([ptCloud lane3dPc1]);
    lane3dPc2 = pointCloud(lanes{2});
    lane3dPc2.Color = uint8(repmat([255 255 0],lane3dPc2.Count,1));
    lanePc = pccat([lanePc lane3dPc2]);
    view(player, lanePc)
end
```



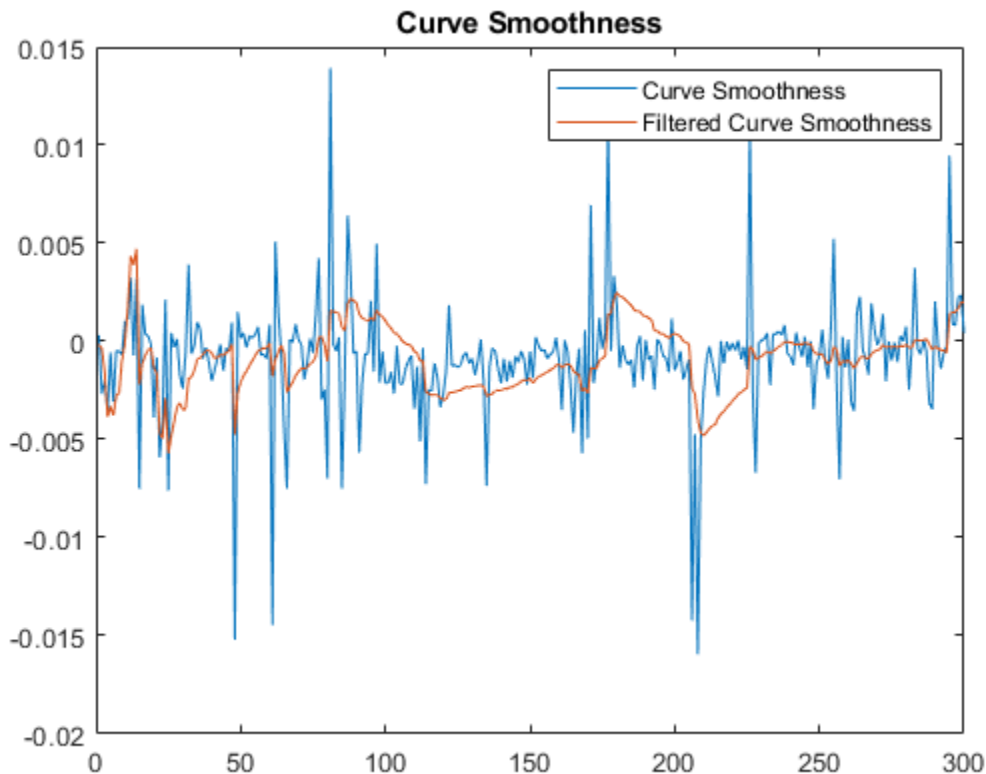
Results

To analyze the lane detection results, compare them against the tracked lane polynomials by plotting them in figures. Each plot compares the parameters with and without the Kalman filter. The first plot compares the drift of lanes along the Y-axis, and the second plot compares the smoothness of the lane polynomials. Smoothness is defined as the rate of change of the slope of the lane curve.

```
figure
plot(drift)
hold on
plot(filteredDrift)
hold off
title('Lane Drift Along Y-axis')
legend('Drift Values','Filtered Drift Values')
```



```
figure
plot(curveSmoothness)
hold on
plot(filteredCurveSmoothness)
hold off
title('Curve Smoothness')
legend('Curve Smoothness','Filtered Curve Smoothness')
```



Summary

This example has shown you how to detect lanes on the intensity channel of point clouds coming from lidar sensor. You have also learned how to fit a 2-D polynomial on detected lane points, and leverage ground plane model to estimate 3-D lane points. You have also used Kalman filter tracking to further improve lane detection.

Supporting Functions

The helperLoadData helper function loads the lidar data set into the MATLAB workspace.

```
function reflidarData = helperGetDataset()
outputFolder = fullfile(tempdir,'WPI');
url = 'https://www.mathworks.com/supportfiles/lidar/data/WPI_LidarData.tar.gz';
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
```



```

% Load lidar data
load(fullfile(outputFolder,'WPI_LidarData.mat'), 'lidarData');

% Select region with a prominent intensity value
reflidarData = cell(300,1);
count = 1;
roi = [-50 50 -30 30 -inf inf];
for i = 81:380
    pc = lidarData{i};
    ind = findPointsInROI(pc,roi);
    reflidarData{count} = select(pc,ind);
    count = count + 1;
end
end

```

This helperInitialWindow helper function computes the starting points of the search window using the detected histogram peaks.

```

function [yval,detectedPeaks] = helperInitialWindow(yvals,peaks,laneWidth)
leftLanesIndices = yvals >= 0;
rightLanesIndices = yvals < 0;
leftLaneYs = yvals(leftLanesIndices);
rightLaneYs = yvals(rightLanesIndices);
peaksLeft = peaks(leftLanesIndices);
peaksRight = peaks(rightLanesIndices);
diff = zeros(sum(leftLanesIndices),sum(rightLanesIndices));
for i = 1:sum(leftLanesIndices)
    for j = 1:sum(rightLanesIndices)
        diff(i,j) = abs(laneWidth - (leftLaneYs(i) - rightLaneYs(j)));
    end
end
[~,minIndex] = min(diff(:));
[row,col] = ind2sub(size(diff),minIndex);
yval = [leftLaneYs(row) rightLaneYs(col)];
detectedPeaks = [peaksLeft(row) peaksRight(col)];
estimatedLaneWidth = leftLaneYs(row) - rightLaneYs(col);

% If the calculated lane width is not within the bounds,
% return the lane with highest peak
if abs(estimatedLaneWidth - laneWidth) > 0.5
    if max(peaksLeft) > max(peaksRight)
        yval = [leftLaneYs(maxLeftInd) NaN];
        detectedPeaks = [peaksLeft(maxLeftInd) NaN];
    else
        yval = [NaN rightLaneYs(maxRightInd)];
        detectedPeaks = [NaN rightLaneYs(maxRightInd)];
    end
end
end
end

```

This helperFitPolynomial helper function fits a RANSAC-based polynomial to the detected lane points and computes the fitting score.

```

function [P,score] = helperFitPolynomial(pts,degree,resolution)
P = fitPolynomialRANSAC(pts,degree,resolution);
ptsSquare = (polyval(P,pts(:,1)) - pts(:,2)).^2;
score = sqrt(mean(ptsSquare));
end

```

This `helperComputeHistogram` helper function computes the histogram of the intensity values of the point clouds.

```
function [histVal,yvals] = helperComputeHistogram(ptCloud,histogramBinResolution)
numBins = ceil((ptCloud.YLimits(2) - ptCloud.YLimits(1))/histogramBinResolution);
histVal = zeros(1,numBins-1);
binStartY = linspace(ptCloud.YLimits(1),ptCloud.YLimits(2),numBins);
yvals = zeros(1,numBins-1);
for i = 1:numBins-1
    roi = [-inf 15 binStartY(i) binStartY(i+1) -inf inf];
    ind = findPointsInROI(ptCloud,roi);
    subPc = select(ptCloud,ind);
    if subPc.Count
        histVal(i) = sum(subPc.Intensity);
        yvals(i) = (binStartY(i) + binStartY(i+1))/2;
    end
end
end
```

This `helperfindpeaks` helper function extracts peaks from the histogram values.

```
function [pkHistVal,pkIdx] = helperfindpeaks(histVal)
pkIdxTemp = (1:size(histVal,2))';
histValTemp = [NaN; histVal'; NaN];
tempIdx = (1:length(histValTemp)).';

% keep only the first of any adjacent pairs of equal values (including NaN)
yFinite = ~isnan(histValTemp);
iNeq = [1; 1 + find((histValTemp(1:end-1) ~= histValTemp(2:end)) & ...
    (yFinite(1:end-1) | yFinite(2:end))))];
tempIdx = tempIdx(iNeq);

% Take the sign of the first sample derivative
s = sign(diff(histValTemp(tempIdx)));

% Find local maxima
maxIdx = 1 + find(diff(s)<0);

% Index into the original index vector without the NaN bookend
pkIdx = tempIdx(maxIdx) - 1;

% Fetch the coordinates of the peak
pkHistVal = histVal(pkIdx);
pkIdx = pkIdxTemp(pkIdx)';
end
```

References

- [1] Ghallabi, Farouk, Fawzi Nashashibi, Ghayath El-Haj-Shhade, and Marie-Anne Mittet. "LIDAR-Based Lane Marking Detection for Vehicle Positioning in an HD Map." In 2018 21st International Conference on Intelligent Transportation Systems (ITSC), 2209-14. Maui: IEEE, 2018. <https://doi.org/10.1109/ITSC.2018.8569951>.
- [2] Thuy, Michael and Fernando León. "Lane Detection and Tracking Based on Lidar Data." *Metrology and Measurement Systems* 17, no. 3 (2010): 311-322. <https://doi.org/10.2478/v10178-010-0027-3>.

Automate Ground Truth Labeling For Vehicle Detection Using PointPillars

This example shows how to automate vehicle detections in a point cloud using a pretrained PointPillars object detection network in the Lidar Labeler. In this example, you use the `AutomationAlgorithm` interface to automate labeling in the Lidar Labeler app.

Lidar Labeler App

Good ground truth data is crucial for developing automated driving algorithms and evaluating performance. However, creating and maintaining a diverse, high-quality, and labeled data set requires significant effort. The Lidar Labeler app provides a framework to automate the labeling process using the `AutomationAlgorithm` interface. You can create a custom algorithm and use it in the app to label your entire data set. You can also edit the results to account for challenging scenarios missed by the algorithm.

In this example, you:

- Use a pretrained PointPillars object detection network to detect objects of class 'vehicle'.
- Create an automation algorithm that you can use in the Lidar Labeler app to automatically label vehicles in the point cloud using the PointPillars network.

Detect Vehicles Using PointPillars Network

Detect vehicles in a point cloud using a pretrained PointPillars object detection network. This network has been trained to detect vehicles in a point cloud. For information on how to train a PointPillars network yourself, see “Lidar 3-D Object Detection Using PointPillars Deep Learning” on page 1-91. Network performance depends on how generalizable the network is. The network might not perform well when it is applied to unseen data. Iteratively introducing custom training data to the learning process can improve the performance on similar data sets.

Download the PointPillars network, which is trained on the WPI data set.

```
pretrainedNetURL = 'https://ssd.mathworks.com/supportfiles/lidar/data/trainedPointPillars.zip';
preTrainedMATFile = fullfile(tempdir, 'trainedPointPillarsNet.mat');
preTrainedZipFile = fullfile(tempdir, 'trainedPointPillars.zip');

if ~exist(preTrainedMATFile, 'file')
    if ~exist(preTrainedZipFile, 'file')
        disp('Downloading pretrained detector (8.3 MB)...');
        websave(preTrainedZipFile, pretrainedNetURL);
    end
    unzip(preTrainedZipFile, tempdir);
end
```

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.

Use the network to detect vehicles in the point cloud by following these steps.

- Add a folder containing the helper functions to the search path.
- Read the point cloud from `lidardata`.

- Specify anchor boxes which are predefined bounding boxes in the format {length, width, height, z-center, yaw angle}.
- Specify grid parameters to crop the full-view point cloud to front-view in the format {{xMin, yMin, zMin}, {xMax, yMax, zMax}, {xStep, yStep, dsFactor}, {Xn, Yn}} where $X_n = \text{round}(((x_{\text{Max}} - x_{\text{Min}}) / x_{\text{Step}}))$ and $Y_n = \text{round}(((y_{\text{Max}} - y_{\text{Min}}) / y_{\text{Step}}))$.
- Specify the label name for the detected object.
- Specify the confidence threshold to use only detections with confidence scores above this value.
- Specify the overlap threshold to remove overlapping detections.
- Select prominent pillars (P) based on the number of points per pillar (N).
- Set executionEnvironment.
- Use the helperCropFrontViewFromLidarData helper function, attached to this example as a supporting file, to crop the point cloud.
- Use the generatePointPillarDetections helper function from the added search path, to get the bounding boxes.
- Display the point cloud with bounding boxes.

```
% Add folder to path.
addpath(fullfile(matlabroot, 'examples', 'deeplearning_shared', 'main'));

% Load the pretrained network.
pretrainedNet = load(preTrainedMATFile);

% Load a point cloud.
ptCloud = pcread(fullfile(toolboxdir('lidar'), 'lidardata', 'highwayScene.pcd'));

% Anchor boxes.
anchorBoxes = {{3.9, 1.6, 1.56, -3.6, 0}, {3.9, 1.6, 1.56, -3.6, pi/2}};

% Cropping parameters.
gridParams = {{0.0, -39.68, -5.0}, {69.12, 39.68, 5.0}, {0.16, 0.16, 2.0}, {432, 496}};

% Label name for detected object.
classNames = {'vehicle'};

% Confidence threshold.
confidenceThreshold = 0.45;

% Overlap threshold.
overlapThreshold = 0.1;

% Number of prominent pillars.
P = 12000;

% Number of points per pillar.
N = 100;

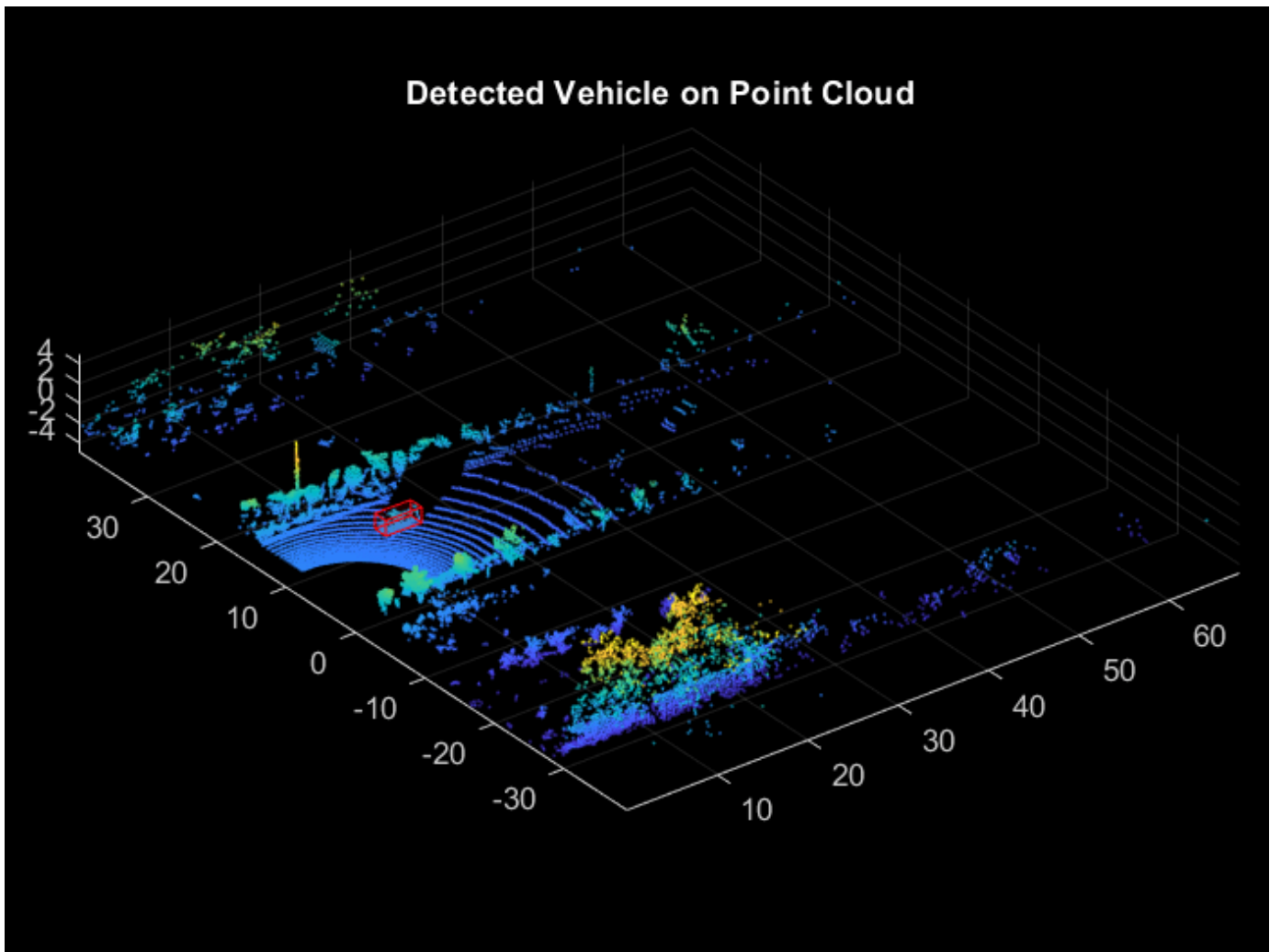
% Set the execution environment.
executionEnvironment = "auto";

% Crop the front view of the point cloud.
processedPointCloud = helperCropFrontViewFromLidarData(ptCloud, gridParams);

% Detect the bounding boxes.
```

```
[box, ~, ~] = generatePointPillarDetections(pretrainedNet.net, processedPointCloud, ...
    anchorBoxes, gridParams, classNames, confidenceThreshold, ...
    overlapThreshold, P, N, executionEnvironment);

% Display the detections on the point cloud.
figure
ax = pcshow(processedPointCloud.Location);
showShape('cuboid',box,'Parent',ax,'Opacity',0.1,'Color','red','LineWidth',0.5)
hold on
zoom(ax, 1.5)
title("Detected Vehicle on Point Cloud")
```



Prepare Lidar Vehicle Detector Automation Class

Construct an automation class for the lidar vehicle detector algorithm. The class inherits from the `vision.labeler.AutomationAlgorithm` (Computer Vision Toolbox) abstract base class. The base class defines properties and signatures for methods that the app uses to configure and run the custom algorithm. The Lidar Labeler app provides an initial automation class template. For more information, see [Create Automation Algorithm for Labeling](#) (Computer Vision Toolbox). The `LidarVehicleDetector` class is based on this template and provides you with a ready-to-use

automation class for vehicle detection in a point cloud. The comments of the class outline the basic steps needed to implement each API call.

Algorithm Properties

Step 1 contains properties that define the name and description of the algorithm and the directions for using the algorithm.

```
% -----  
% Step 1: Define the required properties describing the algorithm. This  
% includes Name, Description, and UserDirections.  
properties(Constant)  
  
    % Name Algorithm Name  
    % Character vector specifying the name of the algorithm.  
    Name = 'Lidar Vehicle Detector';  
  
    % Description Algorithm Description  
    % Character vector specifying the short description of the algorithm.  
    Description = 'Detect vehicles in point cloud using PointPillars network';  
  
    % UserDirections Algorithm Usage Directions  
    % Cell array of character vectors specifying directions for  
    % algorithm users to follow to use the algorithm.  
    UserDirections = {'ROI Label Definition Selection: select one of ' ...  
        'the ROI definitions to be labeled', ...  
        ['Run: Press RUN to run the automation algorithm. '], ...  
        ['Review and Modify: Review automated labels over the interval ', ...  
        'using playback controls. Modify/delete/add ROIs that were not ' ...  
        'satisfactorily automated at this stage. If the results are ' ...  
        'satisfactory, click Accept to accept the automated labels.'], ...  
        ['Change Settings and Rerun: If automated results are not ' ...  
        'satisfactory, you can try to re-run the algorithm with ' ...  
        'different settings. To do so, click Undo Run to undo ' ...  
        'current automation run, click Settings, make changes to Settings,' ...  
        'and press Run again.'], ...  
        ['Accept/Cancel: If the results of automation are satisfactory, ' ...  
        'click Accept to accept all automated labels and return to ' ...  
        'manual labeling. If the results of automation are not ' ...  
        'satisfactory, click Cancel to return to manual labeling ' ...  
        'without saving the automated labels.']};  
end
```

Custom Properties

Step 2 contains the custom properties needed for the core algorithm.

```
% -----  
% Step 2: Define properties you want to use during the algorithm  
% execution.  
properties  
  
    % SelectedLabelName  
    % Name of the selected label. Vehicles detected by the algorithm  
    % are assigned this variable name.  
    SelectedLabelName  
  
    % PretrainedNetwork
```

```
% PretrainedNetwork saves the pretrained PointPillars dlnetwork.
PretrainedNetwork

% Range of point clouds along x,y,and z-axis used to crop
% full-view point clouds to front-view point clouds.
% These parameters guide the calculation of the size of the
% input [xn,yn] passed to 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.

% Dimensions for the pseudo-image calculated as
% round(((xMax - xMin) / xStep));.
% Xn = 432;
% Yn = 496;

% GridParams
% Parameter used to crop full-view point cloud to front-view,
% defined as {{xMin,yMin,zMin}, {xMax,yMax,zMax},
% {xStep,yStep,dsFactor}}, {Xn,Yn}};
GridParams = {{0.0,-39.68,-5.0}, {69.12,39.68,5.0}, {0.16,0.16,2.0}, {432,496}}

% AnchorBoxes
% Predefined bounding box dimensions based on the classes to
% detect, defined in the format {length, width, height,
% z-center, yaw angle}
AnchorBoxes = {{3.9, 1.6, 1.56, -1.78, 0}, {3.9, 1.6, 1.56, -1.78, pi/2}};

% P
% Number of prominent pillars.
P = 12000;

% N
% Number of points per pillar.
N = 100;

% ExecutionEnvironment
% Set the execution environment.
ExecutionEnvironment = "auto";

% ConfidenceThreshold
% Specify the confidence threshold to use only detections with
% confidence scores above this value.
ConfidenceThreshold = 0.45;

% OverlapThreshold
% Specify the overlap threshold to remove overlapping detections.
OverlapThreshold = 0.1;

end
```

Function Definitions

Step 3 deals with function definitions.

The `checkSignalType` function checks if the signal data is supported for automation. The lidar vehicle detector supports signals of type `PointCloud`.

```
function isValid = checkSignalType(signalType)
    % Only point cloud signal data is valid for the Lidar Vehicle
    % detector algorithm.
    isValid = (signalType == vision.labeler.loading.SignalType.PointCloud);
end
```

The `checkLabelDefinition` function checks if the label definition is the appropriate type for automation. The lidar vehicle detector requires the `Cuboid` label type.

```
function isValid = checkLabelDefinition(~, labelDef)
    % Only cuboid ROI label definitions are valid for the Lidar
    % vehicle detector algorithm.
    isValid = labelDef.Type == labelType.Cuboid;
end
```

The `checkSetup` function checks if an ROI label definition is selected for automation.

```
function isReady = checkSetup(algObj)
    % Is there one selected ROI Label definition to automate.
    isReady = ~isempty(algObj.SelectedLabelDefinitions);
end
```

The `settingsDialog` function obtains and modifies the properties defined in Step 2. This API call lets you create a dialog box that opens when you click the **Settings** icon in the **Automate** tab. To create this dialog box, use the `dialog` function to quickly create a simple modal window to optionally modify the confidence and overlap thresholds. The `lidarVehicleDetectorSettings` method contains the code for settings and input validation steps.

```
function settingsDialog(algObj)
    % Invoke dialog with options for modifying the
    % threshold and confidence threshold.
    lidarVehicleDetectorSettings(algObj)
end
```

Execution Functions

Step 4 specifies the execution functions. The `initialize` function populates the initial algorithm state based on the existing labels in the app. In this example, the `initialize` function performs the following steps:

- Store the name of the selected label definition.
- Add the folder containing helper functions to the search path.
- Load the pretrained `PointPillars` object detection network from `tempdir` and save it to the `PretrainedNetwork` property.

```
function initialize(algObj,~)
    % Store the name of the selected label definition. Use this
    % name to label the detected vehicles.
    algObj.SelectedLabelName = algObj.SelectedLabelDefinitions.Name;
```



```

% Add the folder containing helper functions to the search path.
addpath(fullfile(matlabroot,'examples','deeplearning_shared','main'))

% Point to tempdir, where pretrainedNet was downloaded.
preTrainedMATFile = fullfile(tempdir,'trainedPointPillarsNet.mat');
assert(exist(preTrainedMATFile,'file'), ...
    sprintf(['File : %s does not exists \n Download Pretrained PointPillars ' ...
        'network with the link provided in the example'],preTrainedMATFile));
pretrainedNet = load(preTrainedMATFile);
algObj.PretrainedNetwork = pretrainedNet.net;
end

```

The run function defines the core lidar vehicle detector algorithm of this automation class. The run function is called for each frame of the point cloud sequence, and expects the automation class to return a set of labels. You can extend the algorithm to any category the network is trained on. For the purposes of this example, restrict the network to detect objects of class 'Vehicle'.

```

function autoLabels = run(algObj, pointCloud)
    bBoxes = [];
    for i = 1:2
        if i == 2
            % Rotate the point cloud by 180 degrees.
            theta = pi;
            rot = [cos(theta) sin(theta) 0; ...
                -sin(theta) cos(theta) 0; ...
                0 0 1];
            trans = [0, 0, 0];
            tform = rigid3d(rot, trans);
            pointCloud = pctransform(pointCloud, tform);
        end

        % Crop the front view of the point clouds.
        processedPointCloud = helperCropFrontViewFromLidarData(pointCloud, ...
            algObj.GridParams);

        % Detect vehicles using the PointPillars network.
        [box, ~, ~] = generatePointPillarDetections(algObj.PretrainedNetwork, ...
            processedPointCloud, algObj.AnchorBoxes, algObj.GridParams, ...
            {algObj.SelectedLabelName}, algObj.ConfidenceThreshold, ...
            algObj.OverlapThreshold, algObj.P, algObj.N, algObj.ExecutionEnvironment);

        if ~isempty(box)
            if i == 2
                box(:,1) = -box(:,1);
                box(:,2) = -box(:,2);
                box(:,9) = -box(:,9);
            end
            bBoxes = [bBoxes;box];
        end
    end
    if ~isempty(bBoxes)
        % Add automated labels at bounding box locations detected
        % by the vehicle detector, of type Cuboid and with the name
        % of the selected label.
        autoLabels.Name = algObj.SelectedLabelName;
        autoLabels.Type = labelType.Cuboid;
        autoLabels.Position = bBoxes;
    else

```

```
        autoLabels = [];  
    end  
end
```

The `terminate` function handles any cleanup or tear-down required after the automation is done. This algorithm does not require any cleanup, so the function is empty.

Use Lidar Vehicle Detector Automation Class in App

The properties and methods described in the previous section are implemented in the `LidarVehicleDetector` automation algorithm class file. Use the class in the app.

First, create the folder structure `+vision/+labeler` required under the current folder, and copy the automation class into it.

```
mkdir('+vision/+labeler');  
copyfile(fullfile(matlabroot, 'examples', 'lidar', 'main', 'LidarVehicleDetector.m'), ...  
    '+vision/+labeler');
```

Download the point cloud sequence (PCD) and image sequence. For illustration purposes, this example uses WPI lidar data collected on a highway from an Ouster OS1 lidar sensor. Execute the following code block to download and save the lidar data in a temporary folder. 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.

Download the point cloud sequence to a temporary location.

```
lidarURL = 'https://www.mathworks.com/supportfiles/lidar/data/WPI_LidarData.tar.gz';  
lidarDataFolder = fullfile(tempdir, 'WPI_LidarData', filesep);  
lidarDataTarFile = lidarDataFolder + "WPI_LidarData.tar.gz";  
  
if ~exist(lidarDataFolder)  
    mkdir(lidarDataFolder)  
end  
  
if ~exist(lidarDataTarFile, 'file')  
    disp('Downloading WPI Lidar driving data (760 MB)...');  
    websave(lidarDataTarFile, lidarURL);  
    untar(lidarDataTarFile, lidarDataFolder);  
end  
  
% Check if lidar tar.gz file is downloaded, but not uncompressed.  
if ~exist(fullfile(lidarDataFolder, 'WPI_LidarData.mat'), 'file')  
    untar(lidarDataTarFile, lidarDataFolder);  
end
```

The Lidar Labeler app supports the loading of point cloud sequences composed of PCD or PLY files. Save the downloaded point cloud data to PCD files. This example uses only a subset of the WPI point cloud data, from frames 920 to 940.

```
% Load downloaded lidar data into the workspace.  
load(fullfile(lidarDataFolder, 'WPI_LidarData.mat'), 'lidarData');  
lidarData = reshape(lidarData, size(lidarData,2),1);  
  
% Create new folder and write lidar data to PCD files.  
pcdDataFolder = lidarDataFolder + "lidarData";
```

```

if ~exist(pcdDataFolder, 'dir')
    mkdir(fullfile(lidarDataFolder, 'lidarData'));
end

disp('Saving WPI Lidar driving data to PCD files ...');
for i = 920:940
    filename = strcat(fullfile(lidarDataFolder, 'lidarData', filesep), ...
        num2str(i, '%06.0f'), '.pcd');
    pcwrite(lidarData{i}, filename);
end

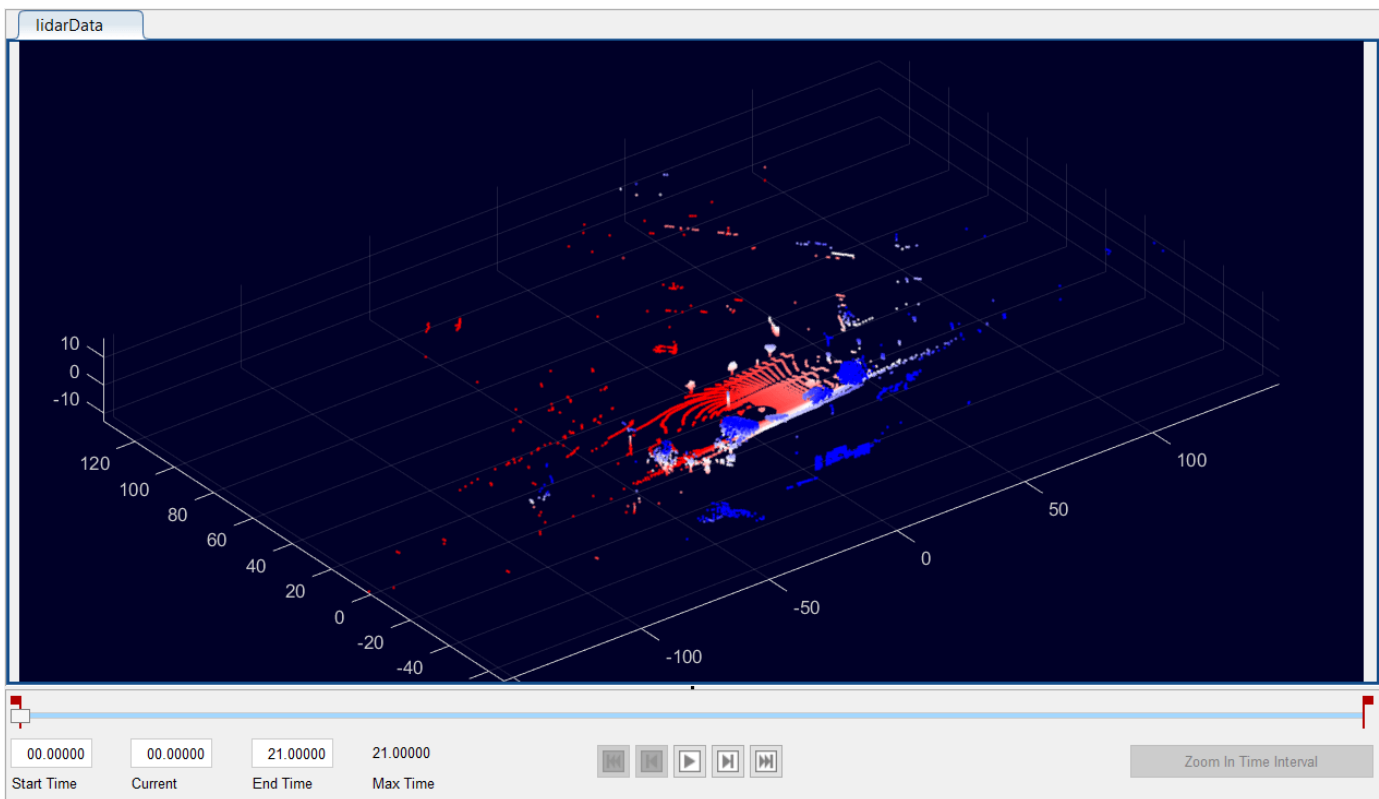
```

Open the Lidar Labeler app and load the point cloud sequence.

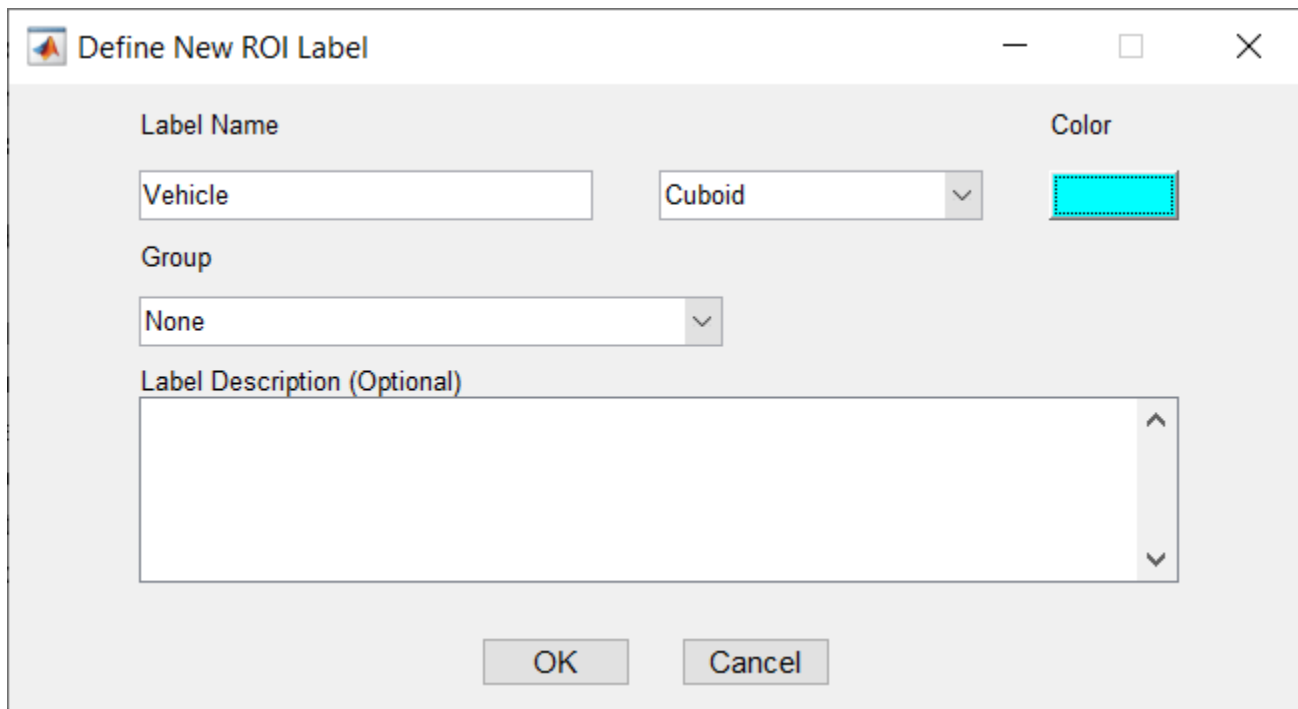
```

pointCloudDir = fullfile(lidarDataFolder, 'lidarData');
lidarLabeler(pointCloudDir);

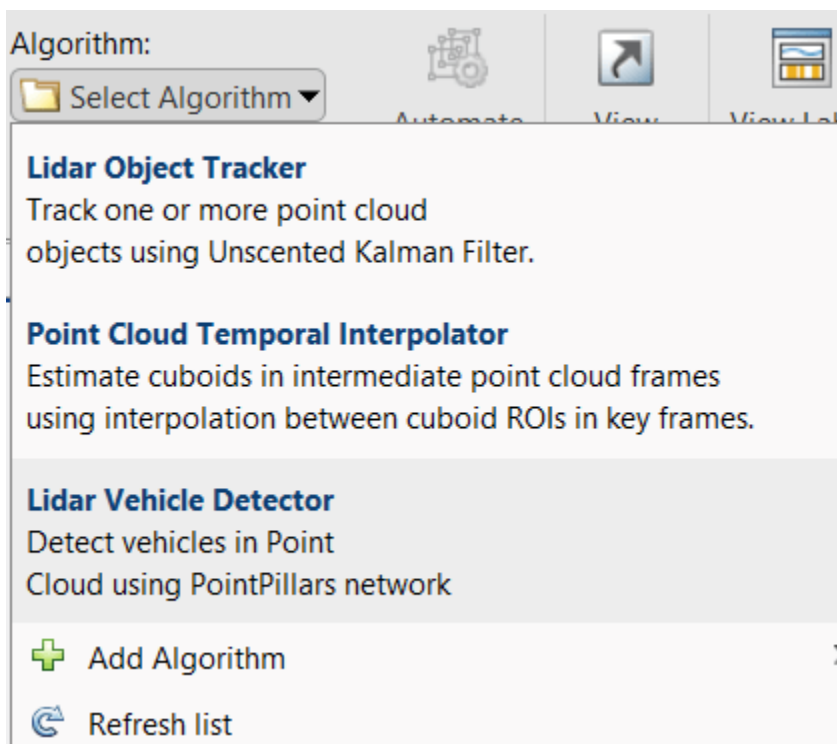
```



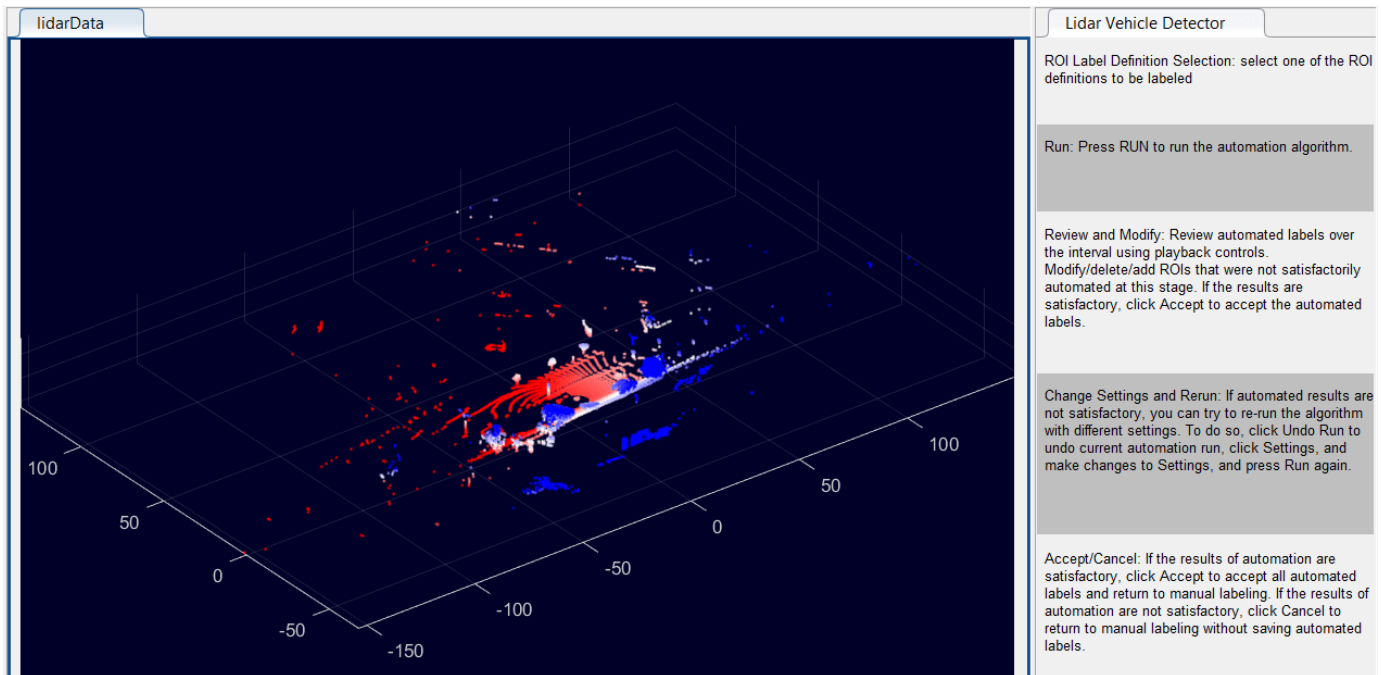
In the **ROI Labels** tab in the left pane, click **Label**. Define an ROI label with the name **Vehicle** and the **Cuboid**. Optionally, you can select a color. Click **OK**.



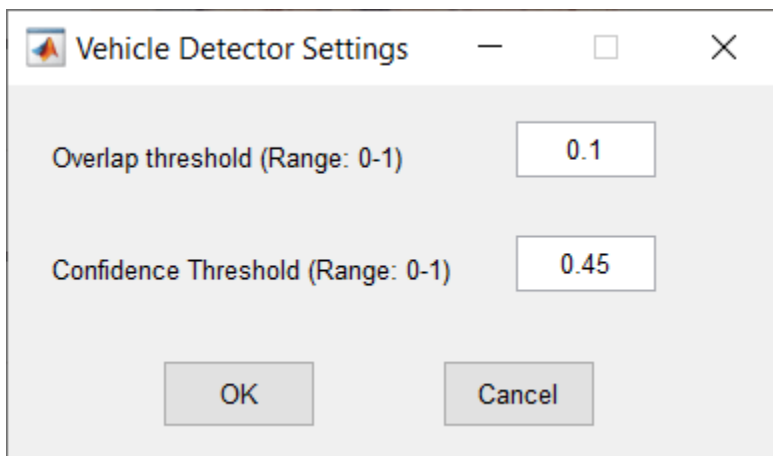
Under **Select Algorithm**, select **Refresh list**. Then, select **Algorithm > Lidar Vehicle Detector**. If you do not see this option, verify that the current working folder has a folder called +vision/+labeler, with a file named LidarVehicleDetector.m in it.



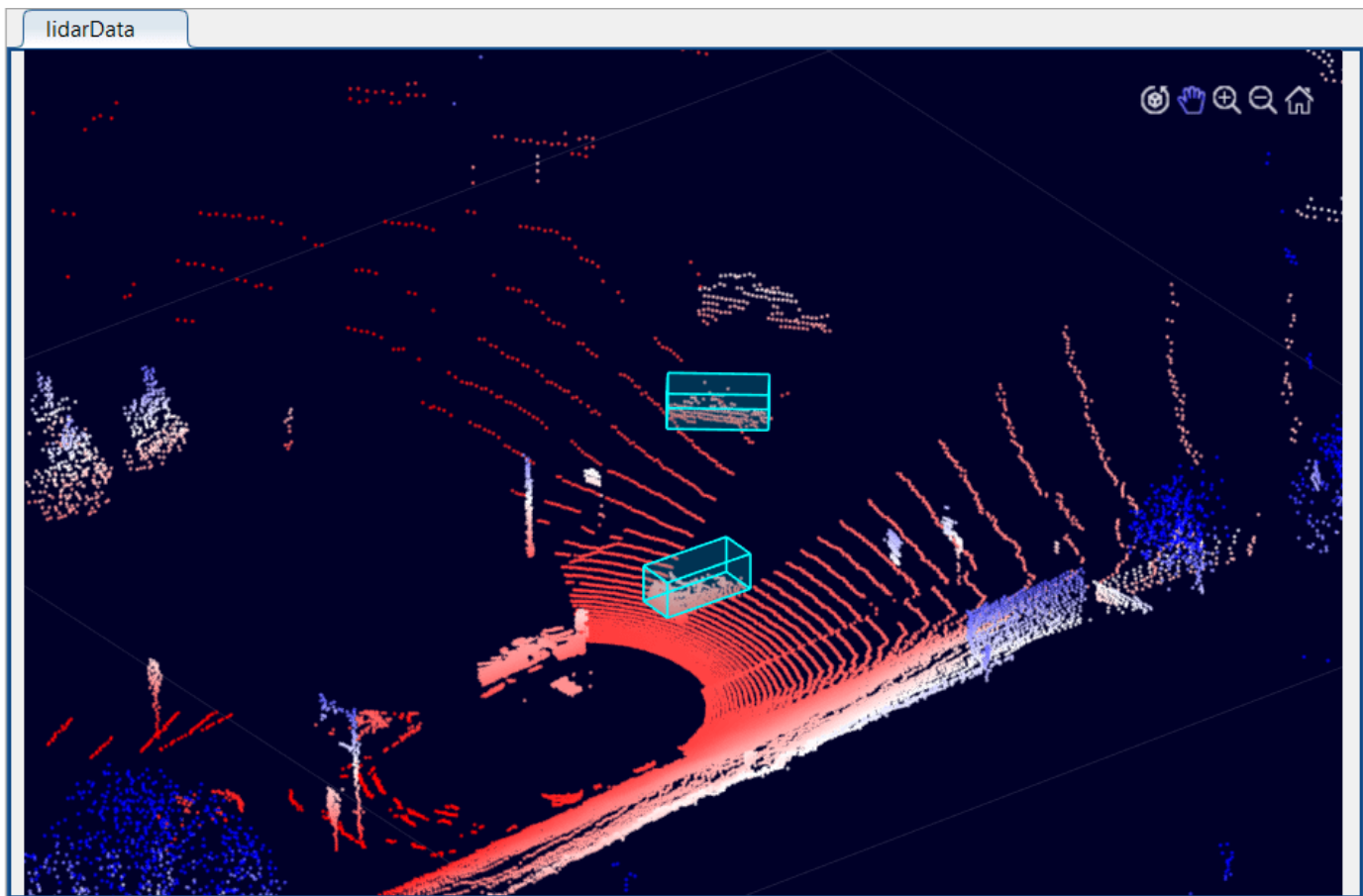
Click **Automate**. The app opens an automation session for the selected signals and displays directions for using the algorithm.



Click **Settings**, and in the dialog box that opens, modify the parameters if needed and click **OK**.



Click **Run**. The created algorithm executes on each frame of the sequence and detects vehicles by using the `Vehicle` label type. After the app completes the automation run, use the slider or arrow keys to scroll through the sequence to locate the frame where the automation algorithm labeled incorrectly. Use the zoom, pan, and 3-D rotation options for viewing and rotating the point cloud. Manually tweak the results by adjusting the detected bounding boxes or adding new bounding boxes.



When you are satisfied with the detected vehicle bounding boxes for the entire sequence, click **Accept**. You can then continue to manually adjust labels or export the labeled ground truth to the MATLAB workspace.

You can use the concepts described in this example to create your own custom automation algorithms and extend the functionality of the app.

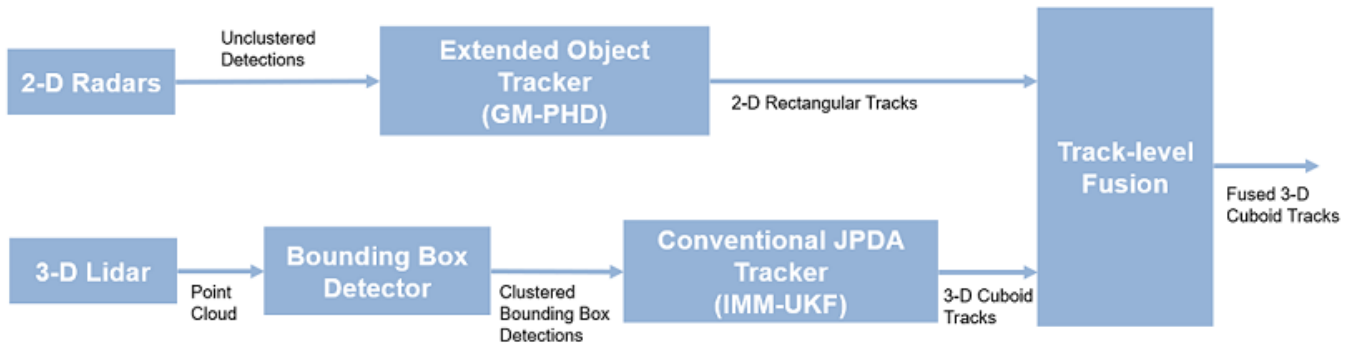
Helper Functions

helperCropFrontViewFromLidarData

```
function processedData = helperCropFrontViewFromLidarData(ptCloud, gridParams)
% Function to crop the front view of the point clouds
% Set the limits for the point cloud.
[row, column] = find(ptCloud.Location(:, :, 1) < gridParams{1,2}{1} ...
    & ptCloud.Location(:, :, 1) > gridParams{1,1}{1} ...
    & ptCloud.Location(:, :, 2) < gridParams{1,2}{2} ...
    & ptCloud.Location(:, :, 2) > gridParams{1,1}{2} ...
    & ptCloud.Location(:, :, 3) < gridParams{1,2}{3} ...
    & ptCloud.Location(:, :, 3) > gridParams{1,1}{3});
ptCloud = select(ptCloud, row, column, 'OutputSize', 'full');
finalPC = removeInvalidPoints(ptCloud);
processedData = finalPC;
end
```

Track-Level Fusion of Radar and Lidar Data

This example shows you how to generate an object-level track list from measurements of a radar and a lidar sensor and further fuse them using a track-level fusion scheme. You process the radar measurements using an extended object tracker and the lidar measurements using a joint probabilistic data association (JPDA) tracker. You further fuse these tracks using a track-level fusion scheme. The schematics of the workflow is shown below.



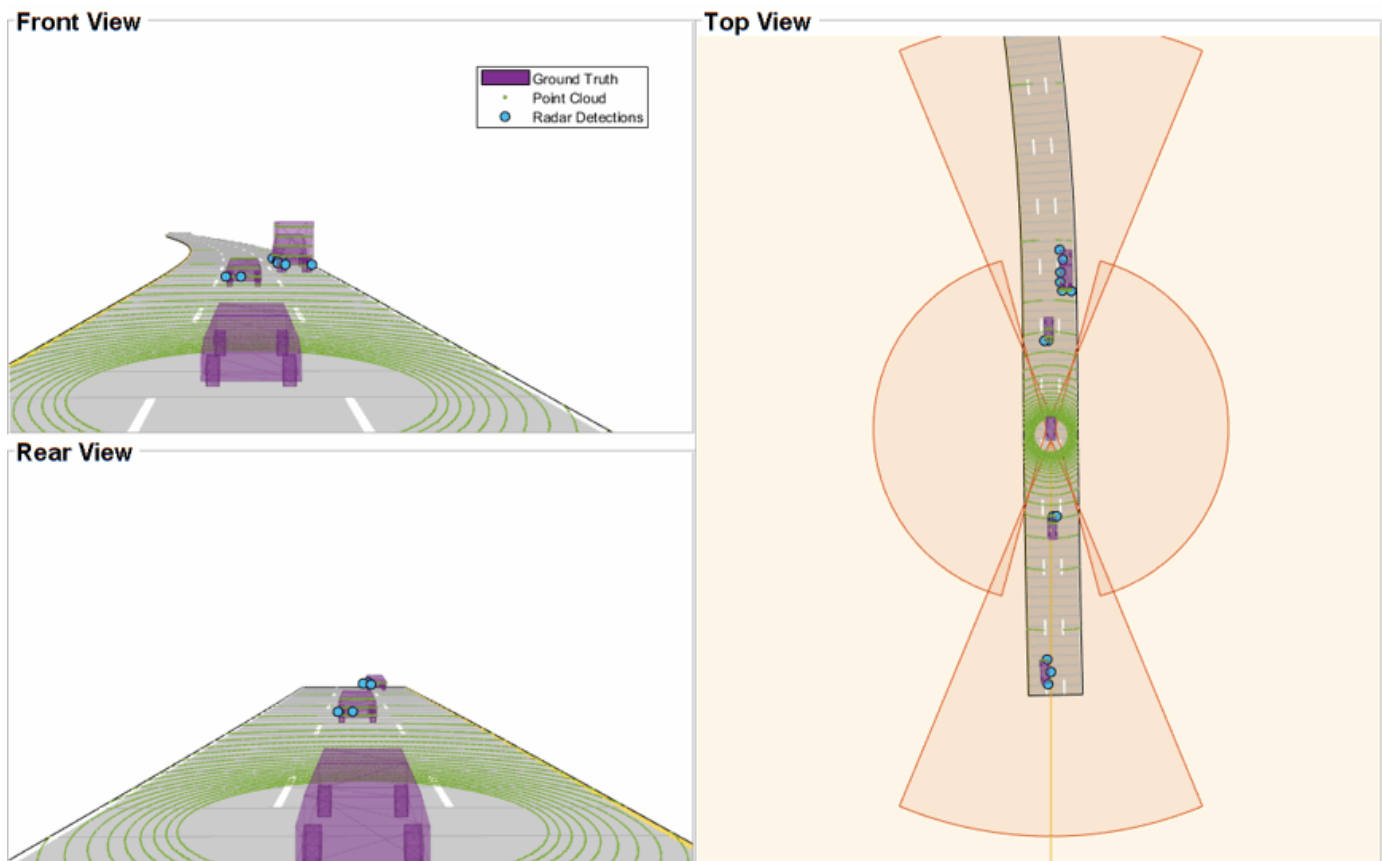
Setup Scenario for Synthetic Data Generation

The scenario used in this example is created using `drivingScenario` (Automated Driving Toolbox). The data from radar and lidar sensors is simulated using `drivingRadarDataGenerator` (Automated Driving Toolbox) and `lidarPointCloudGenerator` (Automated Driving Toolbox), respectively. The creation of the scenario and the sensor models is wrapped in the helper function `helperCreateRadarLidarScenario`. For more information on scenario and synthetic data generation, refer to “Create Driving Scenario Programmatically” (Automated Driving Toolbox).

```
% For reproducible results
rng(2019);
```

```
% Create scenario, ego vehicle and get radars and lidar sensor
[scenario, egoVehicle, radars, lidar] = helperCreateRadarLidarScenario;
```

The Ego vehicle is mounted with four 2-D radar sensors. The front and rear radar sensors have a field of view of 45 degrees. The left and right radar sensors have a field of view of 150 degrees. Each radar has a resolution of 6 degrees in azimuth and 2.5 meters in range. The Ego is also mounted with one 3-D lidar sensor with a field of view of 360 degrees in azimuth and 40 degrees in elevation. The lidar has a resolution of 0.2 degrees in azimuth and 1.25 degrees in elevation (32 elevation channels). Visualize the configuration of the sensors and the simulated sensor data in the animation below. Notice that the radars have higher resolution than objects and therefore return multiple measurements per object. Also notice that the lidar interacts with the low-poly mesh of the actor as well as the road surface to return multiple points from these objects.

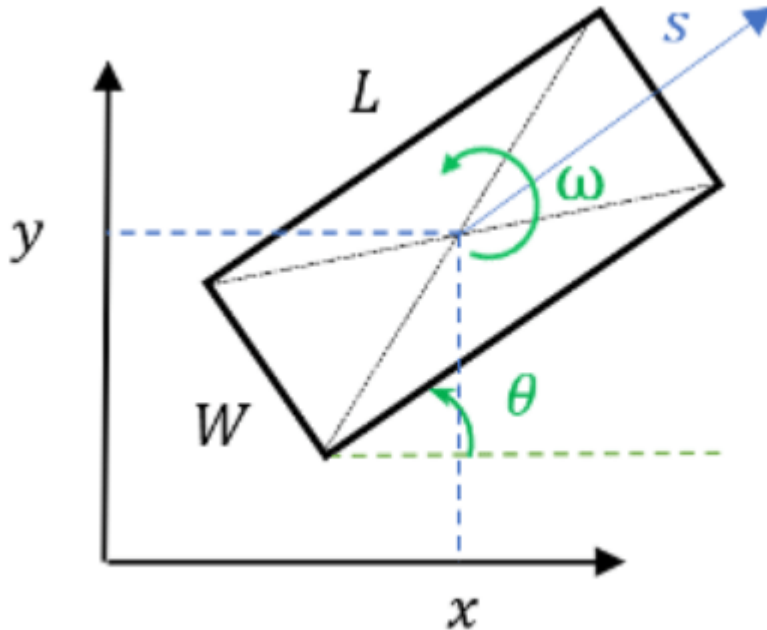


Radar Tracking Algorithm

As mentioned, the radars have higher resolution than the objects and return multiple detections per object. Conventional trackers such as Global Nearest Neighbor (GNN) and Joint Probabilistic Data Association (JPDA) assume that the sensors return at most one detection per object per scan. Therefore, the detections from high-resolution sensors must be either clustered before processing it with conventional trackers or must be processed using extended object trackers. Extended object trackers do not require pre-clustering of detections and usually estimate both kinematic states (for example, position and velocity) and the extent of the objects. For a more detailed comparison between conventional trackers and extended object trackers, refer to the “Extended Object Tracking of Highway Vehicles with Radar and Camera” (Sensor Fusion and Tracking Toolbox) example.

In general, extended object trackers offer better estimation of objects as they handle clustering and data association simultaneously using temporal history of tracks. In this example, the radar detections are processed using a Gaussian mixture probability hypothesis density (GM-PHD) tracker (`trackerPHD` (Sensor Fusion and Tracking Toolbox) and `gmphd` (Sensor Fusion and Tracking Toolbox)) with a rectangular target model. For more details on configuring the tracker, refer to the "GM-PHD Rectangular Object Tracker" section of the “Extended Object Tracking of Highway Vehicles with Radar and Camera” (Sensor Fusion and Tracking Toolbox) example.

The algorithm for tracking objects using radar measurements is wrapped inside the helper class, `helperRadarTrackingAlgorithm`, implemented as a System object™. This class outputs an array of `objectTrack` (Sensor Fusion and Tracking Toolbox) objects and define their state according to the following convention:

$[x \ y \ s \ \theta \ L \ W]$


```
radarTrackingAlgorithm = helperRadarTrackingAlgorithm(radars);
```

Lidar Tracking Algorithm

Similar to radars, the lidar sensor also return multiple measurements per object. Further, the sensor returns a large number of points from the road, which must be removed to before used as inputs for an object-tracking algorithm. While lidar data from obstacles can be directly processed via extended object tracking algorithm, conventional tracking algorithms are still more prevalent for tracking using lidar data. The first reason for this trend is mainly observed due to higher computational complexity of extended object trackers for large data sets. The second reason is the investments into advanced Deep learning-based detectors such as PointPillars [1], VoxelNet [2] and PIXOR [3], which can segment a point cloud and return bounding box detections for the vehicles. These detectors can help in overcoming the performance degradation of conventional trackers due to improper clustering.

In this example, the lidar data is processed using a conventional joint probabilistic data association (JPDA) tracker, configured with an interacting multiple model (IMM) filter. The pre-processing of lidar data to remove point cloud is performed by using a RANSAC-based plane-fitting algorithm and bounding boxes are formed by performing a Euclidian-based distance clustering algorithm. For more information about the algorithm, refer to the “Track Vehicles Using Lidar: From Point Cloud to Track List” (Sensor Fusion and Tracking Toolbox) example. Compared the linked example, the tracking is performed in the scenario frame and the tracker is tuned differently to track objects of different sizes. Further the states of the variables are defined differently to constraint the motion of the tracks in the direction of its estimated heading angle.

The algorithm for tracking objects using lidar data is wrapped inside the helper class, `helperLidarTrackingAlgorithm` implemented as System object. This class outputs an array of

objectTrack (Sensor Fusion and Tracking Toolbox) objects and defines their state according to the following convention:

$$[x \ y \ s \ \theta \ \omega \ z \ \dot{z} \ L \ W \ H]$$

The states common to the radar algorithm are defined similarly. Also, as a 3-D sensor, the lidar tracker outputs three additional states, z , \dot{z} and H , which refer to z-coordinate (m), z-velocity (m/s), and height (m) of the tracked object respectively.

```
lidarTrackingAlgorithm = helperLidarTrackingAlgorithm(lidar);
```

Set Up Fuser, Metrics, and Visualization

Fuser

Next, you will set up a fusion algorithm for fusing the list of tracks from radar and lidar trackers. Similar to other tracking algorithms, the first step towards setting up a track-level fusion algorithm is defining the choice of state vector (or state-space) for the fused or central tracks. In this case, the state-space for fused tracks is chosen to be same as the lidar. After choosing a central track state-space, you define the transformation of the central track state to the local track state. In this case, the local track state-space refers to states of radar and lidar tracks. To do this, you use a `fuserSourceConfiguration` (Sensor Fusion and Tracking Toolbox) object.

Define the configuration of the radar source. The `helperRadarTrackingAlgorithm` outputs tracks with `SourceIndex` set to 1. The `SourceIndex` is provided as a property on each tracker to uniquely identify it and allows a fusion algorithm to distinguish tracks from different sources. Therefore, you set the `SourceIndex` property of the radar configuration as same as those of the radar tracks. You set `IsInitializingCentralTracks` to `true` to let that unassigned radar tracks initiate new central tracks. Next, you define the transformation of a track in central state-space to the radar state-space and vice-versa. The helper functions `central2radar` and `radar2central` perform the two transformations and are included at the end of this example.

```
radarConfig = fuserSourceConfiguration('SourceIndex',1,...
    'IsInitializingCentralTracks',true,...
    'CentralToLocalTransformFcn',@central2radar,...
    'LocalToCentralTransformFcn',@radar2central);
```

Define the configuration of the lidar source. Since the state-space of a lidar track is same as central track, you do not define any transformations.

```
lidarConfig = fuserSourceConfiguration('SourceIndex',2,...
    'IsInitializingCentralTracks',true);
```

The next step is to define the state-fusion algorithm. The state-fusion algorithm takes multiple states and state covariances in the central state-space as input and return a fused estimate of the state and the covariances. In this example, you use a covariance intersection algorithm provided by the helper function, `helperRadarLidarFusionFcn`. A generic covariance intersection algorithm for two Gaussian estimates with mean x_i and covariance P_i can be defined according to the following equations:

$$P_F^{-1} = w_1 P_1^{-1} + w_2 P_2^{-1}$$

$$x_F = P_F (w_1 P_1^{-1} x_1 + w_2 P_2^{-1} x_2)$$

where x_F and P_F are the fused state and covariance and w_1 and w_2 are mixing coefficients from each estimate. Typically, these mixing coefficients are estimated by minimizing the determinant or the trace of the fused covariance. In this example, the mixing weights are estimated by minimizing the determinant of positional covariance of each estimate. Furthermore, as the radar does not estimate 3-D states, 3-D states are only fused with lidars. For more details, refer to the `helperRadarLidarFusionFcn` function shown at the end of this script.

Next, you assemble all the information using a `trackFuser` object.

```
% The state-space of central tracks is same as the tracks from the lidar,
% therefore you use the same state transition function. The function is
% defined inside the helperLidarTrackingAlgorithm class.
f = lidarTrackingAlgorithm.StateTransitionFcn;

% Create a trackFuser object
fuser = trackFuser('SourceConfigurations',{radarConfig;lidarConfig},...
    'StateTransitionFcn',f,...
    'ProcessNoise',diag([1 3 1]),...
    'HasAdditiveProcessNoise',false,...
    'AssignmentThreshold',[250 inf],...
    'ConfirmationThreshold',[3 5],...
    'DeletionThreshold',[5 5],...
    'StateFusion','Custom',...
    'CustomStateFusionFcn',@helperRadarLidarFusionFcn);
```

Metrics

In this example, you assess the performance of each algorithm using the Generalized Optimal SubPattern Assignment Metric (GOSPA) metric. You setup three separate metrics using `trackGOSPAMetric` (Sensor Fusion and Tracking Toolbox) for each of the trackers. GOSPA metric aims to evaluate the performance of a tracking system by providing a scalar cost. A lower value of the metric indicates better performance of the tracking algorithm.

To use the GOSPA metric with custom motion models like the one used in this example, you set the `Distance` property to 'custom' and define a distance function between a track and its associated ground truth. These distance functions, shown at the end of this example are `helperRadarDistance`, and `helperLidarDistance`.

```
% Radar GOSPA
gospaRadar = trackGOSPAMetric('Distance','custom',...
    'DistanceFcn',@helperRadarDistance,...
    'CutoffDistance',25);

% Lidar GOSPA
gospaLidar = trackGOSPAMetric('Distance','custom',...
    'DistanceFcn',@helperLidarDistance,...
    'CutoffDistance',25);

% Central/Fused GOSPA
gospaCentral = trackGOSPAMetric('Distance','custom',...
    'DistanceFcn',@helperLidarDistance,...% State-space is same as lidar
    'CutoffDistance',25);
```

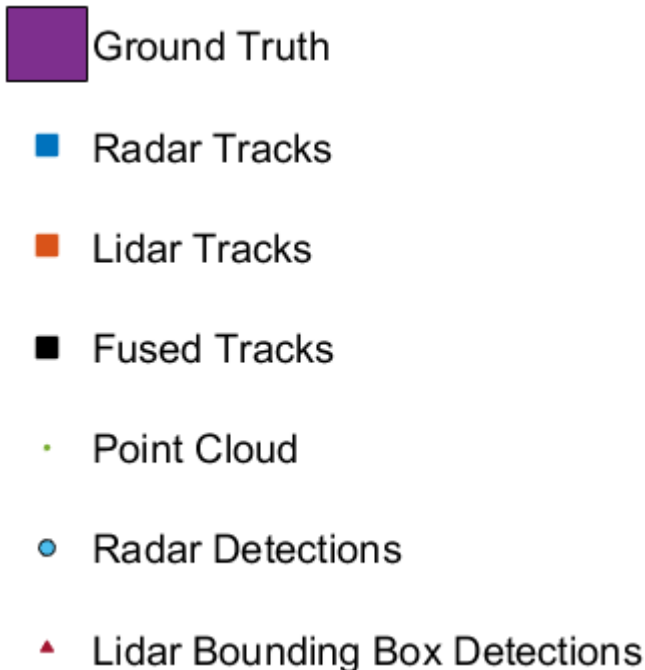
Visualization

The visualization for this example is implemented using a helper class `helperLidarRadarTrackFusionDisplay`. The display is divided into 4 panels. The display plots

the measurements and tracks from each sensor as well as the fused track estimates. The legend for the display is shown below. Furthermore, the tracks are annotated by their unique identity (TrackID) as well as a prefix. The prefixes "R", "L" and "F" stand for radar, lidar, and fused estimate, respectively.

```
% Create a display.
% FollowActorID controls the actor shown in the close-up
% display
display = helperLidarRadarTrackFusionDisplay('FollowActorID',3);

% Show persistent legend
showLegend(display,scenario);
```



Run Scenario and Trackers

Next, you advance the scenario, generate synthetic data from all sensors and process it to generate tracks from each of the system. You also compute the metric for each tracker using the ground truth available from the scenario.

```
% Initialize GOSPA metric and its components for all tracking algorithms.
gospa = zeros(3,0);
missTarget = zeros(3,0);
falseTracks = zeros(3,0);

% Initialize fusedTracks
fusedTracks = objectTrack.empty(0,1);

% A counter for time steps elapsed for storing gospa metrics.
idx = 1;

% Ground truth for metrics. This variable updates every time-step
% automatically being a handle to the actors.
```

```

groundTruth = scenario.actors(2:end);

while advance(scenario)
    % Current time
    time = scenario.SimulationTime;

    % Collect radar and lidar measurements and ego pose to track in
    % scenario frame. See helperCollectSensorData below.
    [radarDetections, ptCloud, egoPose] = helperCollectSensorData(egoVehicle, radars, lidar, time);

    % Generate radar tracks
    radarTracks = radarTrackingAlgorithm(egoPose, radarDetections, time);

    % Generate lidar tracks and analysis information like bounding box
    % detections and point cloud segmentation information
    [lidarTracks, lidarDetections, segmentationInfo] = ...
        lidarTrackingAlgorithm(egoPose, ptCloud, time);

    % Concatenate radar and lidar tracks
    localTracks = [radarTracks;lidarTracks];

    % Update the fuser. First call must contain one local track
    if ~(isempty(localTracks) && ~isLocked(fuser))
        fusedTracks = fuser(localTracks,time);
    end

    % Capture GOSPA and its components for all trackers
    [gospa(1,idx),~,~,~,missTarget(1,idx),falseTracks(1,idx)] = gospaRadar(radarTracks, groundTruth);
    [gospa(2,idx),~,~,~,missTarget(2,idx),falseTracks(2,idx)] = gospaLidar(lidarTracks, groundTruth);
    [gospa(3,idx),~,~,~,missTarget(3,idx),falseTracks(3,idx)] = gospaCentral(fusedTracks, groundTruth);

    % Update the display
    display(scenario, radars, radarDetections, radarTracks, ...
        lidar, ptCloud, lidarDetections, segmentationInfo, lidarTracks,...
        fusedTracks);

    % Update the index for storing GOSPA metrics
    idx = idx + 1;
end

% Update example animations
updateExampleAnimations(display);

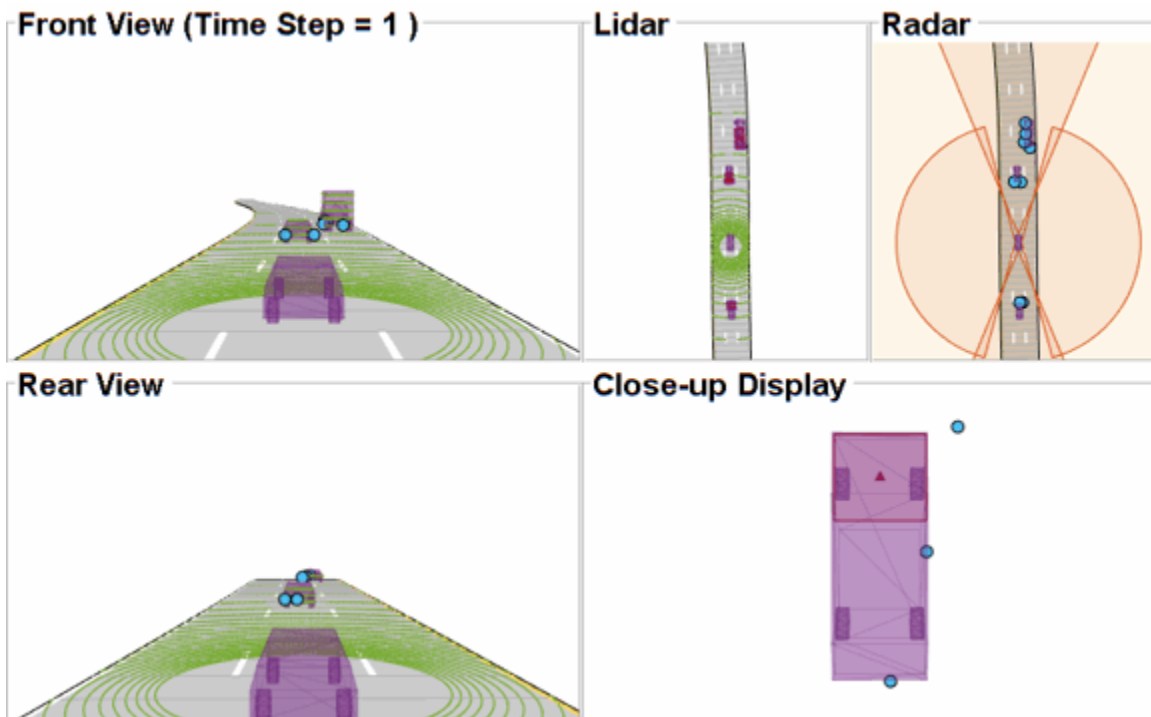
```

Evaluate Performance

Evaluate the performance of each tracker using visualization as well as quantitative metrics. Analyze different events in the scenario and understand how the track-level fusion scheme helps achieve a better estimation of the vehicle state.

Track Maintenance

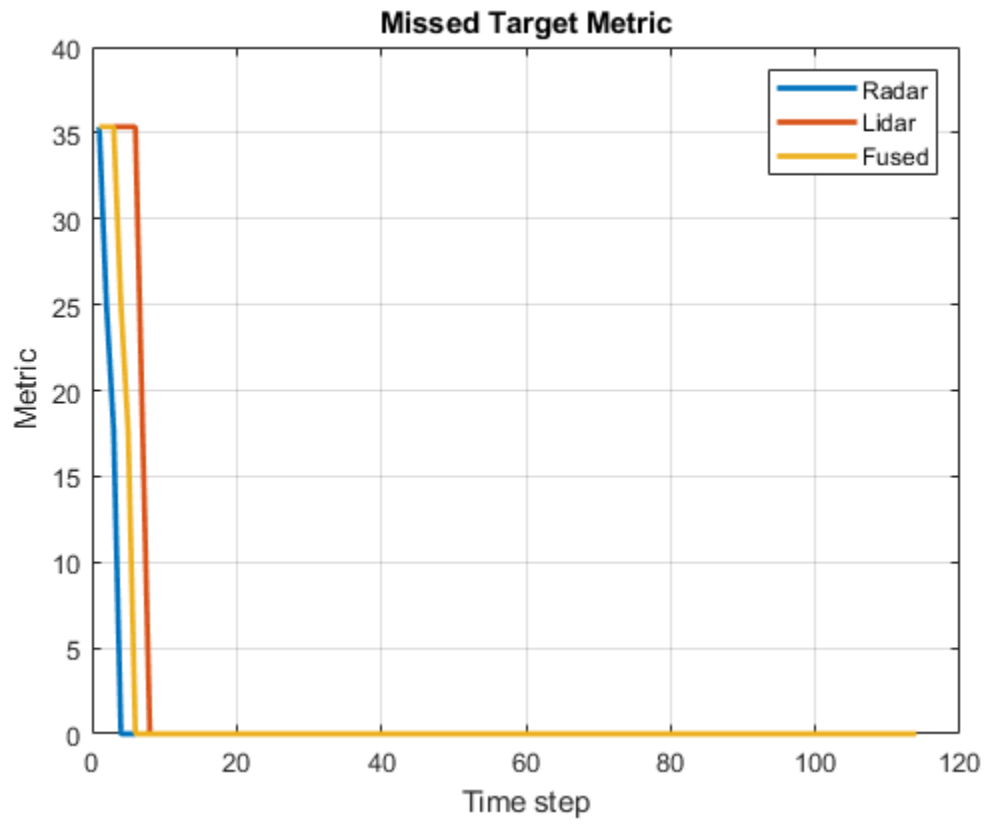
The animation below shows the entire run every three time-steps. Note that each of the three tracking systems - radar, lidar, and the track-level fusion - were able to track all four vehicles in the scenario and no false tracks were confirmed.

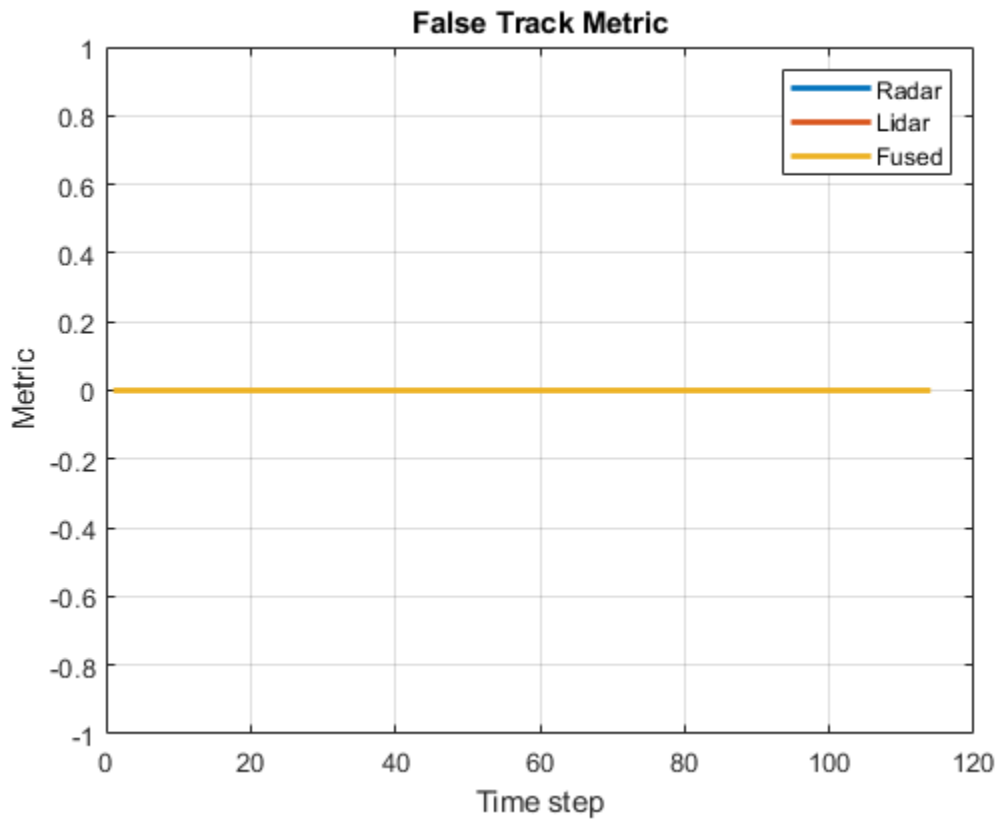


You can also quantitatively measure this aspect of the performance using "missed target" and "false track" components of the GOSPA metric. Notice in the figures below that missed target component starts from a higher value due to establishment delay and goes down to zero in about 5-10 steps for each tracking system. Also, notice that the false track component is zero for all systems, which indicates that no false tracks were confirmed.

```
% Plot missed target component
figure; plot(missTarget','LineWidth',2); legend('Radar','Lidar','Fused');
title("Missed Target Metric"); xlabel('Time step'); ylabel('Metric'); grid on;
```

```
% Plot false track component
figure; plot(falseTracks','LineWidth',2); legend('Radar','Lidar','Fused');
title("False Track Metric"); xlabel('Time step'); ylabel('Metric'); grid on;
```



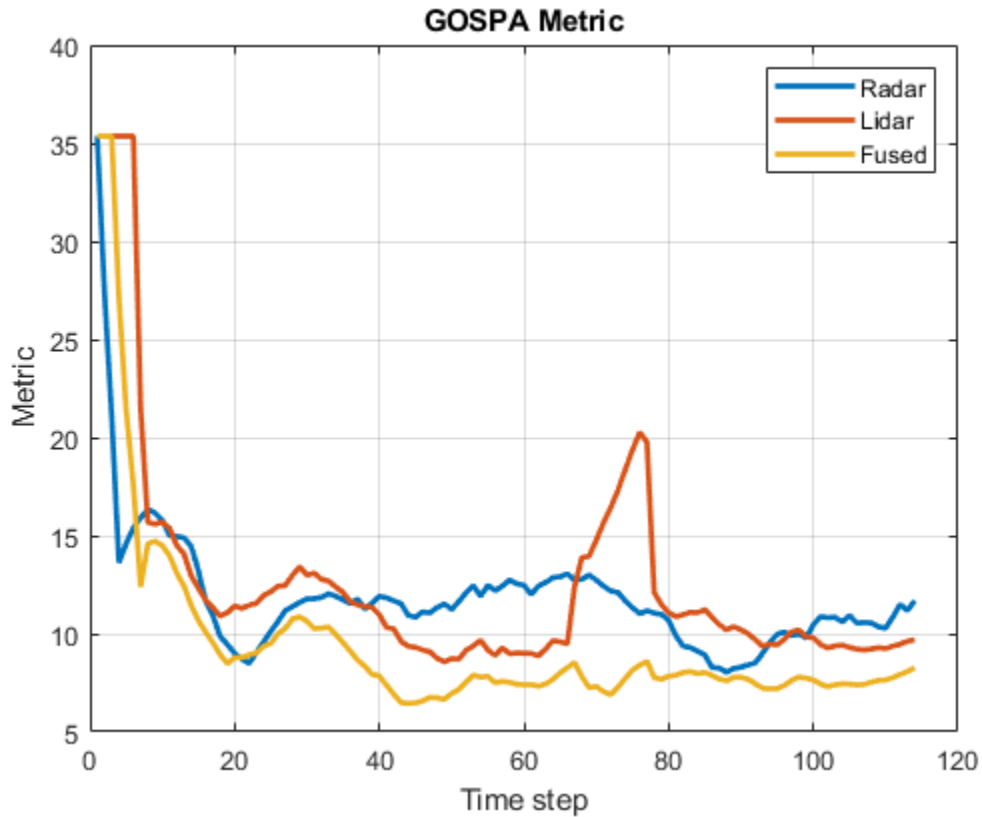


Track-level Accuracy

The track-level or localization accuracy of each tracker can also be quantitatively assessed by the GOSPA metric at each time step. A lower value indicates better tracking accuracy. As there were no missed targets or false tracks, the metric captures the localization errors resulting from state estimation of each vehicle.

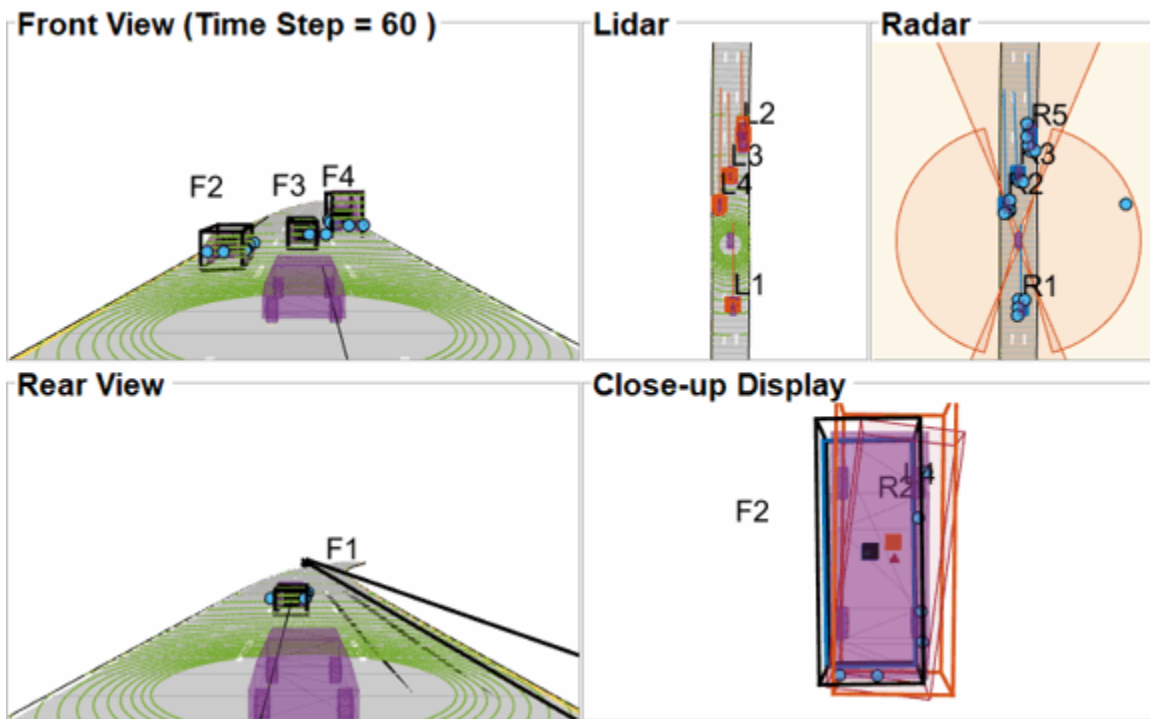
Note that the GOSPA metric for fused estimates is lower than the metric for individual sensor, which indicates that track accuracy increased after fusion of track estimates from each sensor.

```
% Plot GOSPA
figure; plot(gospa', 'LineWidth', 2); legend('Radar', 'Lidar', 'Fused');
title("GOSPA Metric"); xlabel('Time step'); ylabel('Metric'); grid on;
```

Closely-spaced targets

As mentioned earlier, this example uses a Euclidian-distance based clustering and bounding box fitting to feed the lidar data to a conventional tracking algorithm. Clustering algorithms typically suffer when objects are closely-spaced. With the detector configuration used in this example, when the passing vehicle approaches the vehicle in front of the ego vehicle, the detector clusters the point cloud from each vehicle into a bigger bounding box. You can notice in the animation below that the track drifted away from the vehicle center. Because the track was reported with higher certainty in its estimate for a few steps, the fused estimated was also affected initially. However, as the uncertainty increases, its association with the fused estimate becomes weaker. This is because the covariance intersection algorithm chooses a mixing weight for each assigned track based on the certainty of each estimate.

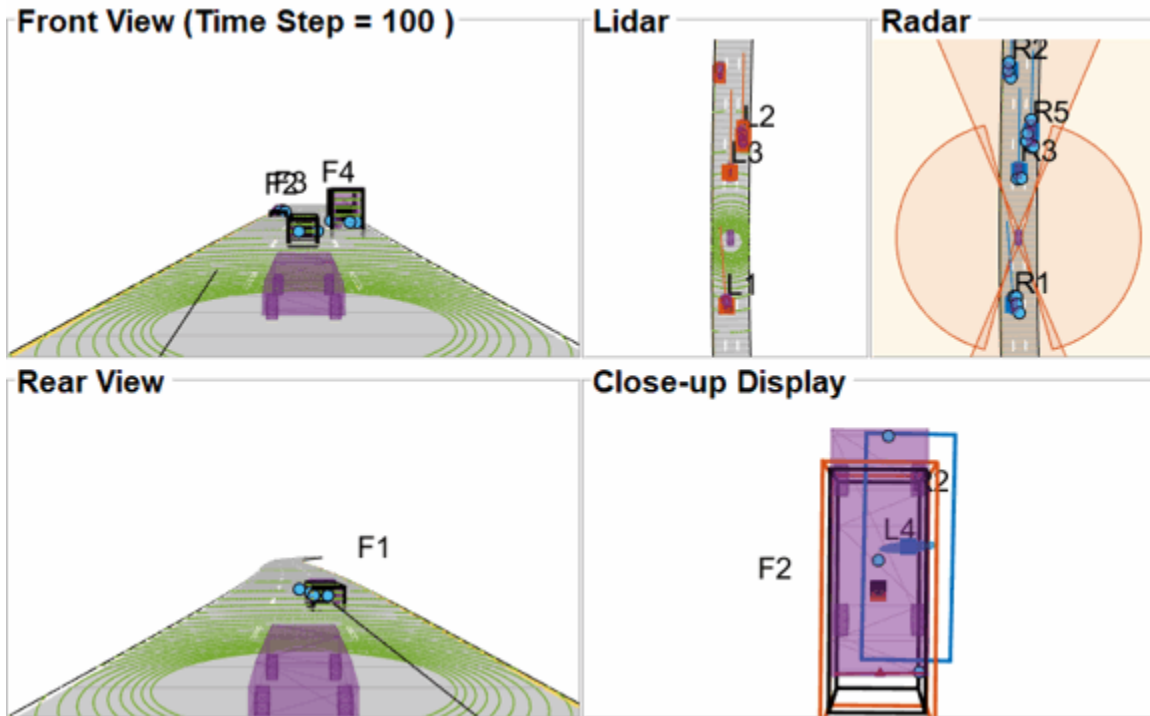


This effect is also captured in the GOSPA metric. You can notice in the GOSPA metric plot above that the lidar metric shows a peak around the 65th time step.

The radar tracks are not affected during this event because of two main reasons. Firstly, the radar sensor outputs range-rate information in each detection, which is different beyond noise-levels for the passing car as compared to the slower moving car. This results in an increased statistical distance between detections from individual cars. Secondly, extended object trackers evaluate multiple possible clustering hypothesis against predicted tracks, which results in rejection of improper clusters and acceptance of proper clusters. Note that for extended object trackers to properly choose the best clusters, the filter for the track must be robust to a degree that can capture the difference between two clusters. For example, a track with high process noise and highly uncertain dimensions may not be able to properly claim a cluster because of its premature age and higher flexibility to account for uncertain events.

Targets at long range

As targets recede away from the radar sensors, the accuracy of the measurements degrade because of reduced signal-to-noise ratio at the detector and the limited resolution of the sensor. This results in high uncertainty in the measurements, which in turn reduces the track accuracy. Notice in the close-up display below that the track estimate from the radar is further away from the ground truth for the radar sensor and is reported with a higher uncertainty. However, the lidar sensor reports enough measurements in the point cloud to generate a "shrunk" bounding box. The shrinkage effect modeled in the measurement model for lidar tracking algorithm allows the tracker to maintain a track with correct dimensions. In such situations, the lidar mixing weight is higher than the radar and allows the fused estimate to be more accurate than the radar estimate.



Summary

In this example, you learned how to setup a track-level fusion algorithm for fusing tracks from radar and lidar sensors. You also learned how to evaluate a tracking algorithm using the Generalized Optimal Subpattern Metric and its associated components.

Utility Functions

collectSensorData

A function to generate radar and lidar measurements at the current time-step.

```
function [radarDetections, ptCloud, egoPose] = helperCollectSensorData(egoVehicle, radars, lidar)
% Current poses of targets with respect to ego vehicle
tgtPoses = targetPoses(egoVehicle);

radarDetections = cell(0,1);
for i = 1:numel(radars)
    thisRadarDetections = step(radars{i},tgtPoses,time);
    radarDetections = [radarDetections;thisRadarDetections]; %#ok<AGROW>
end

% Generate point cloud from lidar
rdMesh = roadMesh(egoVehicle);
ptCloud = step(lidar, tgtPoses, rdMesh, time);

% Compute pose of ego vehicle to track in scenario frame. Typically
% obtained using an INS system. If unavailable, this can be set to
% "origin" to track in Ego vehicle's frame.
egoPose = pose(egoVehicle);
```

end

radar2central

A function to transform a track in the radar state-space to a track in the central state-space.

```
function centralTrack = radar2central(radarTrack)

% Initialize a track of the correct state size
centralTrack = objectTrack('State',zeros(10,1),...
    'StateCovariance',eye(10));

% Sync properties of radarTrack except State and StateCovariance with
% radarTrack See syncTrack defined below.
centralTrack = syncTrack(centralTrack,radarTrack);

xRadar = radarTrack.State;
PRadar = radarTrack.StateCovariance;

H = zeros(10,7); % Radar to central linear transformation matrix
H(1,1) = 1;
H(2,2) = 1;
H(3,3) = 1;
H(4,4) = 1;
H(5,5) = 1;
H(8,6) = 1;
H(9,7) = 1;

xCentral = H*xRadar; % Linear state transformation
PCentral = H*PRadar*H'; % Linear covariance transformation

PCentral([6 7 10],[6 7 10]) = eye(3); % Unobserved states

% Set state and covariance of central track
centralTrack.State = xCentral;
centralTrack.StateCovariance = PCentral;

end
```

central2radar

A function to transform a track in the central state-space to a track in the radar state-space.

```
function radarTrack = central2radar(centralTrack)

% Initialize a track of the correct state size
radarTrack = objectTrack('State',zeros(7,1),...
    'StateCovariance',eye(7));

% Sync properties of centralTrack except State and StateCovariance with
% radarTrack See syncTrack defined below.
radarTrack = syncTrack(radarTrack,centralTrack);

xCentral = centralTrack.State;
PCentral = centralTrack.StateCovariance;

H = zeros(7,10); % Central to radar linear transformation matrix
```

```

H(1,1) = 1;
H(2,2) = 1;
H(3,3) = 1;
H(4,4) = 1;
H(5,5) = 1;
H(6,8) = 1;
H(7,9) = 1;

xRadar = H*xCentral; % Linear state transformation
PRadar = H*PCentral*H'; % Linear covariance transformation

```

```

% Set state and covariance of radar track
radarTrack.State = xRadar;
radarTrack.StateCovariance = PRadar;
end

```

syncTrack

A function to syncs properties of one track with another except the "State" and "StateCovariance" property

```

function tr1 = syncTrack(tr1,tr2)
props = properties(tr1);
notState = ~strcmpi(props, 'State');
notCov = ~strcmpi(props, 'StateCovariance');

props = props(notState & notCov);
for i = 1:numel(props)
    tr1.(props{i}) = tr2.(props{i});
end
end

```

pose

A function to return pose of the ego vehicle as a structure.

```

function egoPose = pose(egoVehicle)
egoPose.Position = egoVehicle.Position;
egoPose.Velocity = egoVehicle.Velocity;
egoPose.Yaw = egoVehicle.Yaw;
egoPose.Pitch = egoVehicle.Pitch;
egoPose.Roll = egoVehicle.Roll;
end

```

helperLidarDistance

Function to calculate a normalized distance between the estimate of a track in radar state-space and the assigned ground truth.

```

function dist = helperLidarDistance(track, truth)

% Calculate the actual values of the states estimated by the tracker

% Center is different than origin and the trackers estimate the center
rOriginToCenter = -truth.OriginOffset(:) + [0;0;truth.Height/2];
rot = quaternion([truth.Yaw truth.Pitch truth.Roll], 'eulerd', 'ZYX', 'frame');
actPos = truth.Position(:) + rotatepoint(rot, rOriginToCenter)';

```

```
% Actual speed and z-rate
actVel = [norm(truth.Velocity(1:2));truth.Velocity(3)];

% Actual yaw
actYaw = truth.Yaw;

% Actual dimensions.
actDim = [truth.Length;truth.Width;truth.Height];

% Actual yaw rate
actYawRate = truth.AngularVelocity(3);

% Calculate error in each estimate weighted by the "requirements" of the
% system. The distance specified using Mahalanobis distance in each aspect
% of the estimate, where covariance is defined by the "requirements". This
% helps to avoid skewed distances when tracks under/over report their
% uncertainty because of inaccuracies in state/measurement models.

% Positional error.
estPos = track.State([1 2 6]);
reqPosCov = 0.1*eye(3);
e = estPos - actPos;
d1 = sqrt(e'/reqPosCov*e);

% Velocity error
estVel = track.State([3 7]);
reqVelCov = 5*eye(2);
e = estVel - actVel;
d2 = sqrt(e'/reqVelCov*e);

% Yaw error
estYaw = track.State(4);
reqYawCov = 5;
e = estYaw - actYaw;
d3 = sqrt(e'/reqYawCov*e);

% Yaw-rate error
estYawRate = track.State(5);
reqYawRateCov = 1;
e = estYawRate - actYawRate;
d4 = sqrt(e'/reqYawRateCov*e);

% Dimension error
estDim = track.State([8 9 10]);
reqDimCov = eye(3);
e = estDim - actDim;
d5 = sqrt(e'/reqDimCov*e);

% Total distance
dist = d1 + d2 + d3 + d4 + d5;

end
```

helperRadarDistance

Function to calculate a normalized distance between the estimate of a track in radar state-space and the assigned ground truth.

```

function dist = helperRadarDistance(track, truth)

% Calculate the actual values of the states estimated by the tracker

% Center is different than origin and the trackers estimate the center
rOriginToCenter = -truth.OriginOffset(:) + [0;0;truth.Height/2];
rot = quaternion([truth.Yaw truth.Pitch truth.Roll], 'eulerd', 'ZYX', 'frame');
actPos = truth.Position(:) + rotatepoint(rot,rOriginToCenter)';
actPos = actPos(1:2); % Only 2-D

% Actual speed
actVel = norm(truth.Velocity(1:2));

% Actual yaw
actYaw = truth.Yaw;

% Actual dimensions. Only 2-D for radar
actDim = [truth.Length;truth.Width];

% Actual yaw rate
actYawRate = truth.AngularVelocity(3);

% Calculate error in each estimate weighted by the "requirements" of the
% system. The distance specified using Mahalanobis distance in each aspect
% of the estimate, where covariance is defined by the "requirements". This
% helps to avoid skewed distances when tracks under/over report their
% uncertainty because of inaccuracies in state/measurement models.

% Positional error
estPos = track.State([1 2]);
reqPosCov = 0.1*eye(2);
e = estPos - actPos;
d1 = sqrt(e'/reqPosCov*e);

% Speed error
estVel = track.State(3);
reqVelCov = 5;
e = estVel - actVel;
d2 = sqrt(e'/reqVelCov*e);

% Yaw error
estYaw = track.State(4);
reqYawCov = 5;
e = estYaw - actYaw;
d3 = sqrt(e'/reqYawCov*e);

% Yaw-rate error
estYawRate = track.State(5);
reqYawRateCov = 1;
e = estYawRate - actYawRate;
d4 = sqrt(e'/reqYawRateCov*e);

% Dimension error
estDim = track.State([6 7]);
reqDimCov = eye(2);
e = estDim - actDim;
d5 = sqrt(e'/reqDimCov*e);

```

```
% Total distance
dist = d1 + d2 + d3 + d4 + d5;

% A constant penalty for not measuring 3-D state
dist = dist + 3;

end
```

helperRadarLidarFusionFcn

Function to fuse states and state covariances in central track state-space

```
function [x,P] = helperRadarLidarFusionFcn(xAll,PAll)
n = size(xAll,2);
dets = zeros(n,1);

% Initialize x and P
x = xAll(:,1);
P = PAll(:, :, 1);

onlyLidarStates = false(10,1);
onlyLidarStates([6 7 10]) = true;

% Only fuse this information with lidar
xOnlyLidar = xAll(onlyLidarStates,:);
POnlyLidar = PAll(onlyLidarStates,onlyLidarStates,:);

% States and covariances for intersection with radar and lidar both
xToFuse = xAll(~onlyLidarStates,:);
PToFuse = PAll(~onlyLidarStates,~onlyLidarStates,:);

% Sorted order of determinants. This helps to sequentially build the
% covariance with comparable determinations. For example, two large
% covariances may intersect to a smaller covariance, which is comparable to
% the third smallest covariance.
for i = 1:n
    dets(i) = det(PToFuse(1:2,1:2,i));
end
[~,idx] = sort(dets,'descend');
xToFuse = xToFuse(:,idx);
PToFuse = PToFuse(:, :, idx);

% Initialize fused estimate
thisX = xToFuse(:,1);
thisP = PToFuse(:, :, 1);

% Sequential fusion
for i = 2:n
    [thisX,thisP] = fusecovintUsingPos(thisX, thisP, xToFuse(:,i), PToFuse(:, :, i));
end

% Assign fused states from all sources
x(~onlyLidarStates) = thisX;
P(~onlyLidarStates,~onlyLidarStates,:) = thisP;

% Fuse some states only with lidar source
valid = any(abs(xOnlyLidar) > 1e-6,1);
```



```

xMerge = xOnlyLidar(:,valid);
PMerge = POnlyLidar(:, :, valid);

if sum(valid) > 1
    [xL,PL] = fusecovint(xMerge,PMerge);
elseif sum(valid) == 1
    xL = xMerge;
    PL = PMerge;
else
    xL = zeros(3,1);
    PL = eye(3);
end

x(onlyLidarStates) = xL;
P(onlyLidarStates,onlyLidarStates) = PL;

end

function [x,P] = fusecovintUsingPos(x1,P1,x2,P2)
% Covariance intersection in general is employed by the following
% equations:
%  $P^{-1} = w_1 P_1^{-1} + w_2 P_2^{-1}$ 
%  $x = P(w_1 P_1^{-1} x_1 + w_2 P_2^{-1} x_2)$ ;
% where  $w_1 + w_2 = 1$ 
% Usually a scalar representative of the covariance matrix like "det" or
% "trace" of P is minimized to compute w. This is offered by the function
% "fusecovint". However. in this case, the w are chosen by minimizing the
% determinants of "positional" covariances only.
n = size(x1,1);
idx = [1 2];
detP1pos = det(P1(idx,idx));
detP2pos = det(P2(idx,idx));
w1 = detP2pos/(detP1pos + detP2pos);
w2 = detP1pos/(detP1pos + detP2pos);
I = eye(n);

P1inv = I/P1;
P2inv = I/P2;

Pinv = w1*P1inv + w2*P2inv;
P = I/Pinv;

x = P*(w1*P1inv*x1 + w2*P2inv*x2);

end

```

References

- [1] Lang, Alex H., et al. "PointPillars: Fast encoders for object detection from point clouds." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2019.
- [2] Zhou, Yin, and Oncel Tuzel. "Voxelnet: End-to-end learning for point cloud based 3d object detection." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018.

[3] Yang, Bin, Wenjie Luo, and Raquel Urtasun. "Pixor: Real-time 3d object detection from point clouds." Proceedings of the IEEE conference on Computer Vision and Pattern Recognition. 2018.

Code Generation For Lidar Object Detection Using PointPillars Deep Learning

This example shows how to generate CUDA® MEX for a PointPillars object detector with custom layers. For more information, see “Lidar 3-D Object Detection Using PointPillars Deep Learning” on page 1-91 example from the Lidar Toolbox™.

Third-Party Prerequisites

Required

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

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

- NVIDIA CUDA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, 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);
```

Pretrained PointPillars Network

Download the PointPillars network trained in the *Lidar 3-D Object Detection Using PointPillars Deep Learning* example. To train the network yourself, see “Lidar 3-D Object Detection Using PointPillars Deep Learning” on page 1-91.

```
outputFolder = fullfile(tempdir, 'Pandaset');
pretrainedNetURL = 'https://ssd.mathworks.com/supportfiles/lidar/data/trainedPointPillarsPandaset';
matFile = helperDownloadPretrainedPointPillarsNet(outputFolder, pretrainedNetURL);
load(matFile, 'net');
```

pointpillarsDetect Entry-Point Function

The `pointpillarsDetect` entry-point function takes pillar features and pillar indices as input and passes them to a trained network for prediction through the `pointpillarPredict` function. The `pointpillarsDetect` function loads the network object from the MAT file into a persistent variable and reuses the persistent object for subsequent prediction calls. Specifically, the function uses the `dlnetwork` representation of the network trained in the “Lidar 3-D Object Detection Using PointPillars Deep Learning” on page 1-91.

```
type('pointpillarsDetect.m')
```

```
function [bboxes,scores,labels] = pointpillarsDetect(matFile, pillarFeatures, pillarIndices, gridParams)
% The pointpillarsDetect function detects the bounding boxes, scores, and
% labels in the point cloud.

coder.extrinsic('helpergeneratePointPillarDetections');

% Define Anchor Boxes
anchorBoxes = {{1.92, 4.5, 1.69, -1.78, 0}, {1.92, 4.5, 1.69, -1.78, pi/2}, {2.1575, 6.0081, 2.3091, 0, 0}};

% Predict the output
predictions = pointpillarPredict(matFile,pillarFeatures,pillarIndices,numOutputs);

% Generate Detections
[bboxes,scores,labels] = helpergeneratePointPillarDetections(predictions,gridParams,pillarIndices,
    anchorBoxes,confidenceThreshold,overlapThreshold,classNames);

end

function YPredCell = pointpillarPredict(matFile,pillarFeatures,pillarIndices,numOutputs)
% Predict the output of network and extract the following confidence,
% x, y, z, l, w, h, yaw and class.

% load the deep learning network for prediction
persistent net;

if isempty(net)
    net = coder.loadDeepLearningNetwork(matFile);
end

YPredCell = cell(1,numOutputs);
[YPredCell{:}] = predict(net,pillarIndices,pillarFeatures);
end
```

Evaluate the Entry-Point Function for Object Detection

Evaluate the entry-point function on a point cloud.

- Define the grid parameters and pillar extraction parameters.
- Read a point cloud and generate pillar features and indices using the `createPillars` helper function, attached to this example as a supporting file.
- Specify the confidence threshold as 0.5 to keep only detections with confidence scores greater than this value.
- Specify the overlap threshold as 0.1 to remove overlapping detections.
- Use the entry-point function `pointpillarsDetect` to get the predicted bounding boxes, confidence scores, and class labels.
- Display the point cloud with bounding boxes.

Define grid parameters.

```
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 the pseudo-image.

```
Xn = round((xMax - xMin) / xStep);
Yn = round((yMax - yMin) / yStep);
gridParams = {xMin,yMin,zMin},{xMax,yMax,zMax},{xStep,yStep,dsFactor},{Xn,Yn};
```

Define the pillar extraction parameters.

```
P = 12000; % Define number of prominent pillars.
N = 100; % Define number of points per pillar.
```

Calculate the number of network outputs.

```
networkOutputs = numel(net.OutputNames);
```

Read a point cloud from the Pandaset data set [2 on page 1-0] and convert it to *M*-by-4 format.

```
pc = pcread('pandasetDrivingData.pcd');
ptCloud = cat(2,pc.Location,pc.Intensity);
```

Create pillar features and pillar indices from the point clouds using the `createPillars` helper function, attached to this example as a supporting file.

```
processedPtCloud = createPillars({ptCloud, '', ''}, gridParams,P,N);
```

Extract the pillar features and pillar indices.

```
pillarFeatures = darray(processedPtCloud{1,1}, 'SSCB');
pillarIndices = darray(processedPtCloud{1,2}, 'SSCB');
```

Specify the class names.

```
classNames = {'car','truck'};
```

Define the desired thresholds.

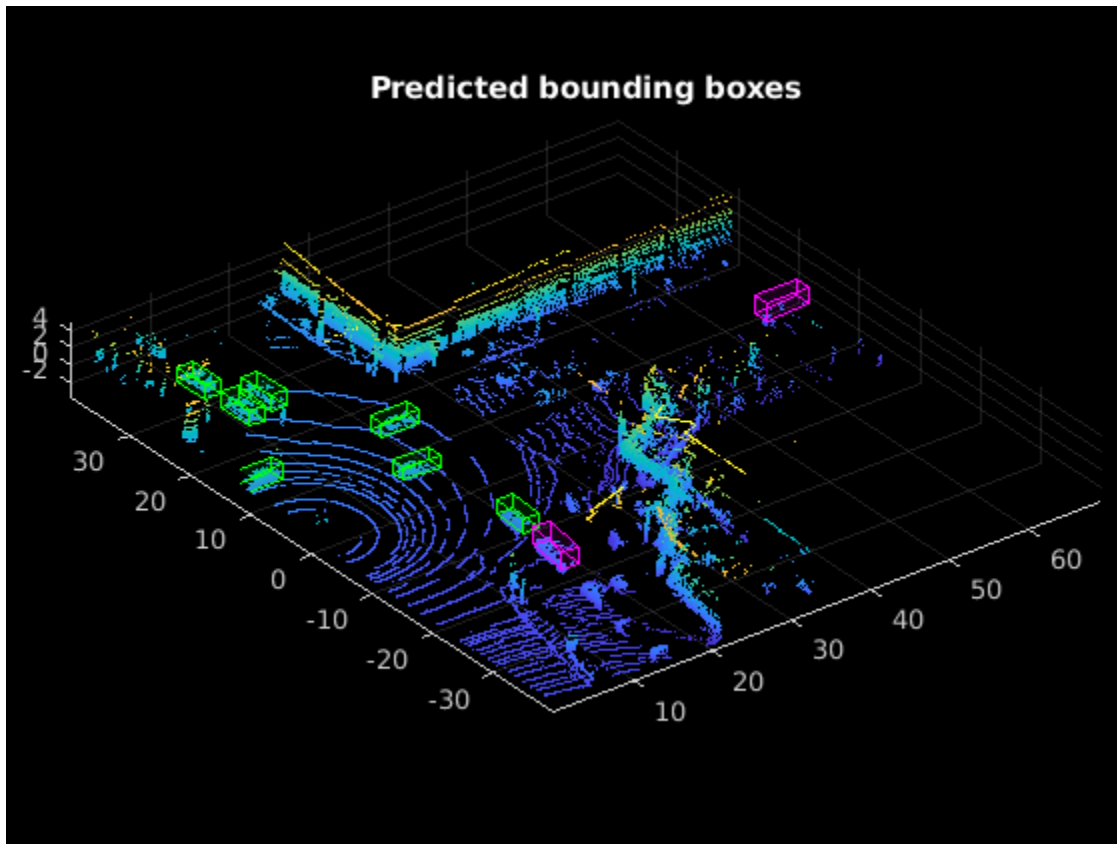
```
confidenceThreshold = 0.5;
overlapThreshold = 0.1;
```

Use the `detect` method on the PointPillars network and display the results.

```
[bboxes,~,labels] = pointpillarsDetect(matFile, pillarFeatures, pillarIndices, gridParams, network);
bboxesCar = bboxes(labels == 'car',:);
bboxesTruck = bboxes(labels == 'truck',:);
```

Display the detections on the point cloud.

```
helperDisplay3DBoxesOverlaidPointCloud(pc.Location, bboxesCar, 'green', bboxesTruck, 'magenta',
```



Generate CUDA MEX

To generate CUDA® code for the `pointpillarsDetect` 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.

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

args = {coder.Constant(matFile), pillarFeatures, pillarIndices, gridParams, coder.Constant(networkOutputs,
    confidenceThreshold, overlapThreshold, classNames);

codegen -config cfg pointpillarsDetect -args args -report
```

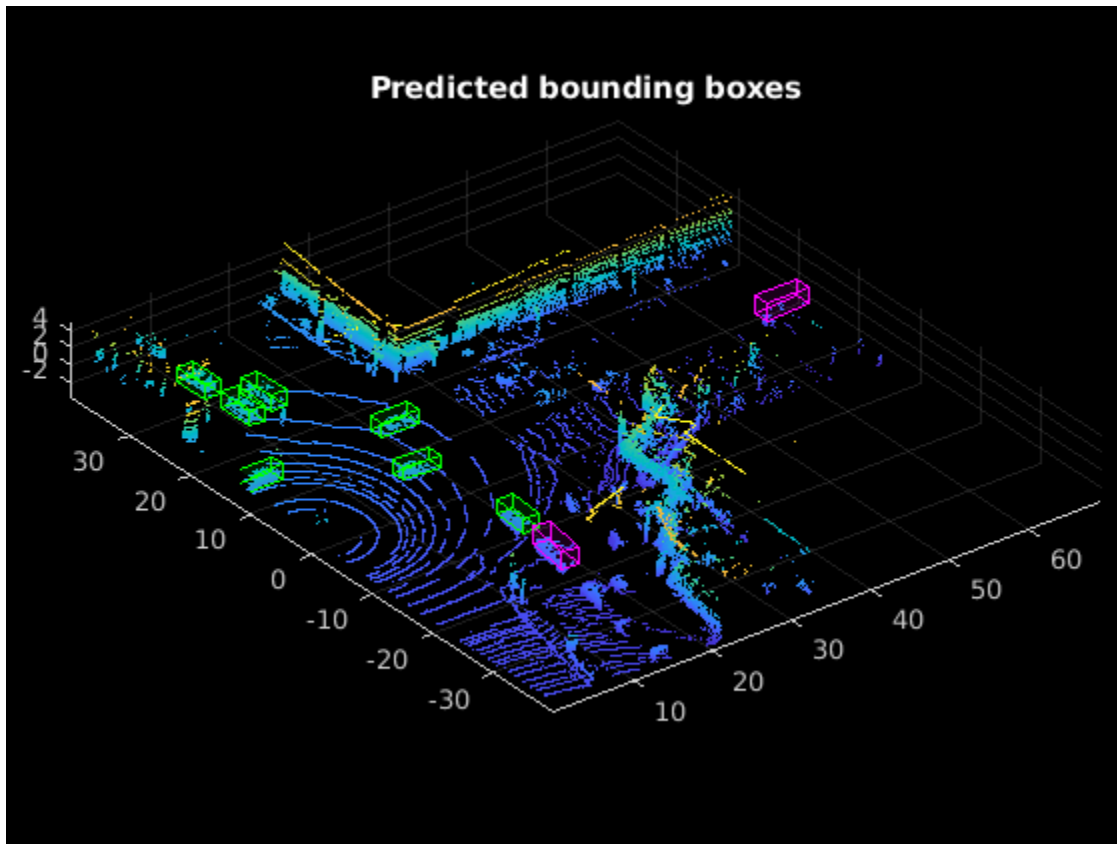
Code generation successful: [View report](#)

Run the Generated MEX

Call the generated CUDA MEX with the same pillar features and indices as before. Display the results.

```
[bboxes, ~, labels] = pointpillarsDetect_mex(matFile, pillarFeatures, pillarIndices, gridParams,
    networkOutputs, confidenceThreshold, overlapThreshold, classNames);
bboxesCar = bboxes(labels == 'car',:);
bboxesTruck = bboxes(labels == 'truck',:);
```

```
helperDisplay3DBoxesOverlaidPointCloud(pc.Location, bboxesCar, 'green', bboxesTruck, 'magenta',
```



Helper Functions

```
function preTrainedMATFile = helperDownloadPretrainedPointPillarsNet(outputFolder, pretrainedNetURL)
% Download the pretrained PointPillars network.
if ~exist(outputFolder, 'dir')
    mkdir(outputFolder);
end
preTrainedZipFile = fullfile(outputFolder, 'trainedPointPillarsPandasetNet.zip');
preTrainedMATFile = fullfile(outputFolder, 'trainedPointPillarsPandasetNet.mat');
if ~exist(preTrainedMATFile, 'file')
    if ~exist(preTrainedZipFile, 'file')
        disp('Downloading pretrained detector (8.4 MB)...');
        websave(preTrainedZipFile, pretrainedNetURL);
    end
    unzip(preTrainedZipFile, outputFolder);
end
end

function helperDisplay3DBoxesOverlaidPointCloud(ptCld, labelsCar, carColor, labelsTruck, truckColor)
% Display the point cloud with different colored bounding boxes for different
% classes
figure;
ax = pcshow(ptCld);
showShape('cuboid', labelsCar, 'Parent', ax, 'Opacity', 0.1, 'Color', carColor, 'LineWidth', 2);
hold on;
showShape('cuboid', labelsTruck, 'Parent', ax, 'Opacity', 0.1, 'Color', truckColor, 'LineWidth', 2);
end
```

```
    showShape('cuboid', labelsTruck, 'Parent', ax, 'Opacity', 0.1, 'Color', truckColor, 'LineWid  
    title(titleForFigure);  
    zoom(ax,1.5);  
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>.

[2] Hesai and Scale. PandaSet. <https://scale.com/open-datasets/pandaset>.

Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning

This example shows how to train a PointNet++ deep learning network to perform semantic segmentation on aerial lidar data.

Lidar data acquired from airborne laser scanning systems is increasingly used in applications such as topographic mapping, building and city modeling, biomass measurement, and disaster management. Extracting meaningful information from this data requires semantic segmentation, a process where each pixel in the point cloud is assigned a class label.

In this example, you train a PointNet++ network to perform semantic segmentation by using the Dayton annotated lidar earth scan (DALES) data set [1 on page 1-0], which contains scenes of dense, labeled lidar aerial data from urban, suburban, rural, and commercial settings. Out of 40 scenes, 29 scenes are used for training and the remaining 11 sequences are used for testing. The data set provides semantic segmentation labels for 8 such as buildings, cars, trucks, poles, power lines, fences, ground, and vegetation.

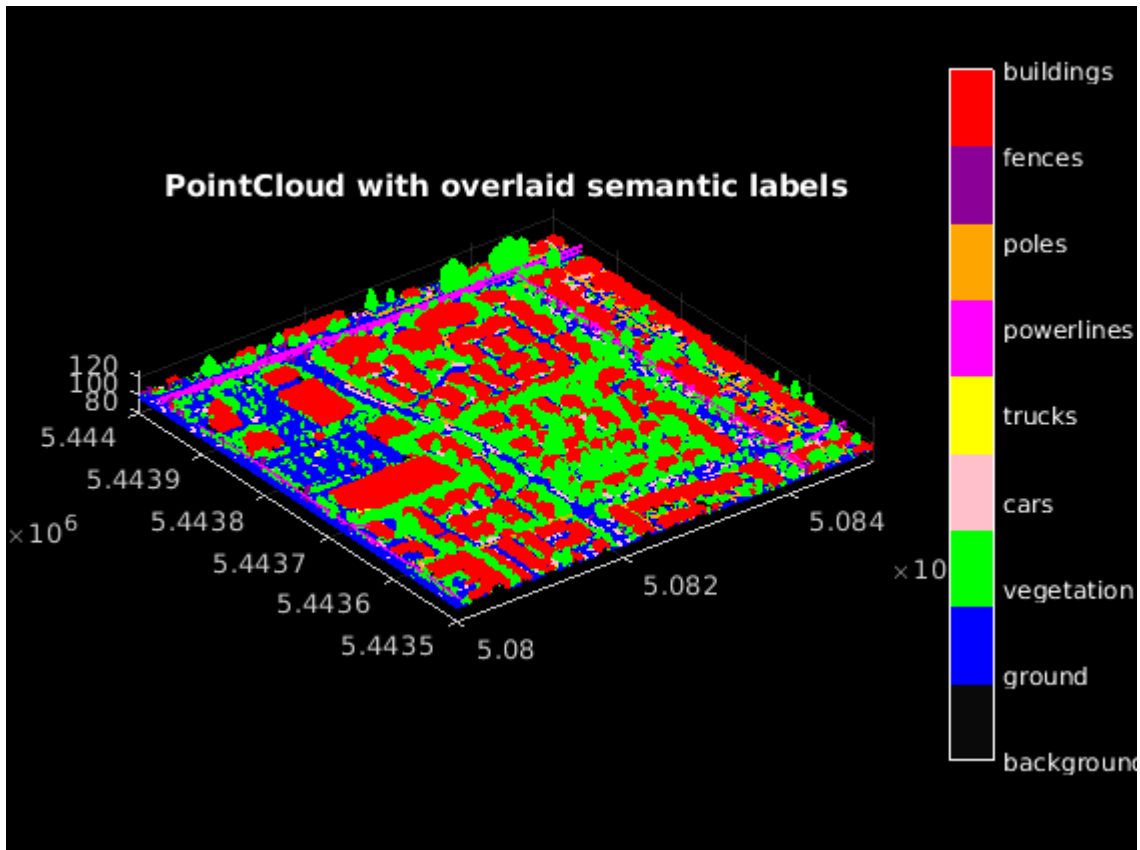
Load DALES Data

The DALES data set contains 40 scenes of aerial lidar data. The data set is already divided into 29 scenes for training and 11 scenes for testing. Each pixel in the data has a class label. Follow the instructions on the DALES website to download the data set to the folder specified by the `dataFolder` variable. Create folders to store training and test data.

```
dataFolder = fullfile(tempdir, 'DALES');
trainDataFolder = fullfile(dataFolder, 'dales_las', 'train');
testDataFolder = fullfile(dataFolder, 'dales_las', 'test');
```

Preview a point cloud from the training data.

```
lasReader = lasFileReader(fullfile(trainDataFolder, '5080_54435.las'));
[pc,attr] = readPointCloud(lasReader, "Attributes", "Classification");
labels = attr.Classification + 1;
classNames = [
    "background"
    "ground"
    "vegetation"
    "cars"
    "trucks"
    "powerlines"
    "poles"
    "fences"
    "buildings"
];
cmap = labelColorMap();
figure;
colormap(cmap);
ax = pcshow(pc.Location, labels);
labelColorbar(ax, cmap, classNames);
title("PointCloud with overlaid semantic labels");
```



Preprocess Data

Each point cloud in the DALES data set covers an area of 500-by-500 meters, which is much larger than the typical area covered by terrestrial rotating lidar point clouds. For efficient memory processing, divide the point cloud into small, non-overlapping grids. Additionally, focus on the three dominant classes—the ground, buildings, and vegetation—to avoid problems due to class imbalance.

Use the `helperCropPointCloudsAndMergeLabels` function, attached to this example as a supporting file, to:

- Crop the point clouds into non-overlapping grids of size 50-by-50 meters.
- Merge all classes other than the dominant classes (the ground, buildings, and vegetation) into a single background class so that the data contains only four classes.
- Save the cropped grids and semantic labels as PCD and PNG files, respectively.

Define the grid dimensions and set a fixed number of points per grid to enable faster training.

```
gridSize = [50,50];
numPoints = 115000;
```

If you use this workflow for your own data, set `writeFiles` to "false" if the training data is divided into grids. Please note that the training data must be in a format supported by the `pcread` function.

```
writeFiles = true;
[pcCropTrainPath,labelsCropTrainPath] = helperCropPointCloudsAndMergeLabels(gridSize,trainDataFolder);
[pcCropTestPath,labelsCropTestPath] = helperCropPointCloudsAndMergeLabels(gridSize,testDataFolder);
```

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

Create Datastore Objects for Training

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

```
ldsTrain = fileDatastore(pcCropTrainPath, 'ReadFcn', @(x) pcread(x));
```

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

```
classNames = classNames([1,2,3,9],:)
```

```
classNames = 4x1 string
    "background"
    "ground"
    "vegetation"
    "buildings"
```

```
numClasses = numel(classNames);
```

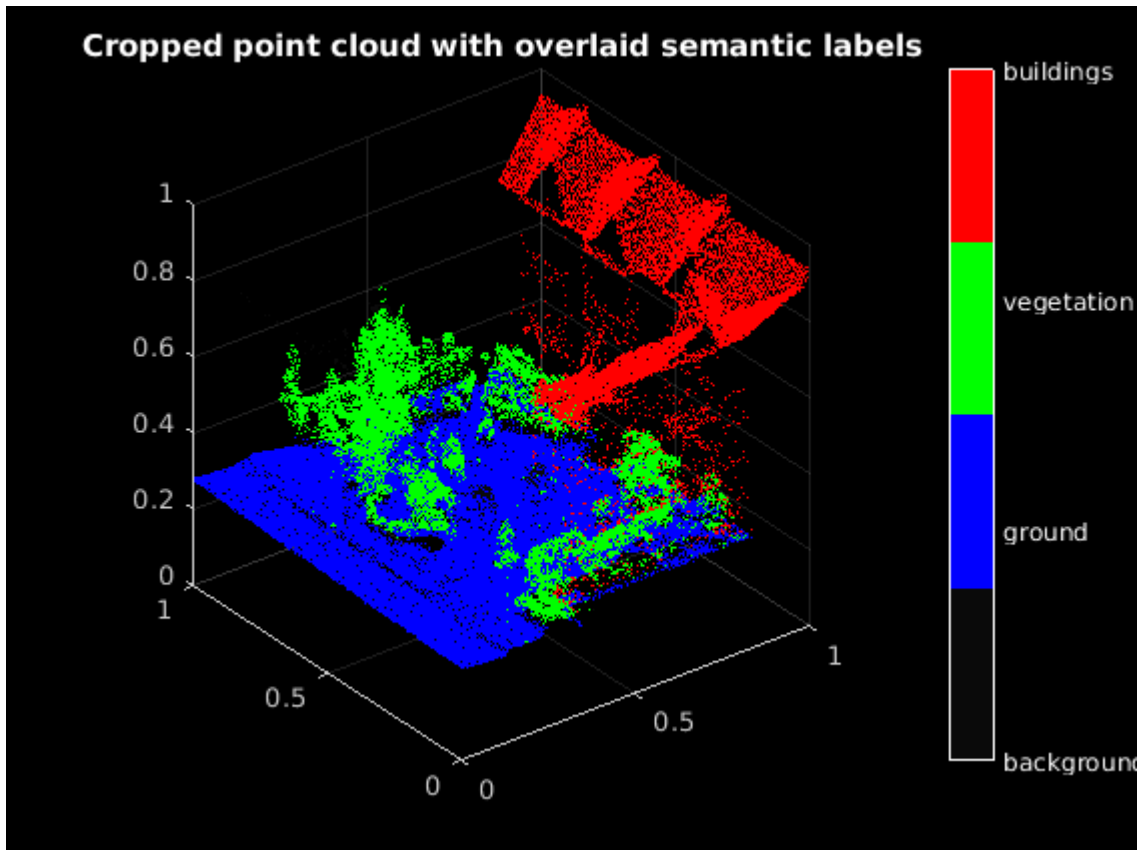
```
% Specify label IDs from 1 to the number of classes.
```

```
labelIDs = 1 : numClasses;
```

```
pxdsTrain = pixelLabelDatastore(labelsCropTrainPath,classNames,labelIDs);
```

Load and display the point clouds.

```
ptcld = preview(ldsTrain);
labels = preview(pxdsTrain);
cmap = cmap([1,2,3,9],:);
figure;
ax1 = pcshow(ptcld.Location,uint8(labels));
labelColorbar(ax1,cmap,classNames);
title("Cropped point cloud with overlaid semantic labels");
```



Use the `normalizePointCloud` function, defined at the end of the example, to normalize the point cloud between 0 and 1.

```
ldsTransformed = transform(ldsTrain,@(x) normalizePointCloud(x));
```

Use the `onehotEncodeLabels` function, defined at the end of the example, to convert the labels to one-hot-encoded representation for training the segmentation network using a custom training loop.

```
pxdsTransformed = transform(pxdsTrain,@(x) onehotEncodeLabels(x,classNames));
```

Use the `combine` function to combine the point clouds and pixel labels into a single datastore for training.

```
dsTrain = combine(ldsTransformed,pxdsTransformed);
```

Define PointNet++ Model

The PointNet++ [2 on page 1-0] segmentation model consists of two main components:

- Set abstraction modules
- Feature propagation modules

The series of set abstraction modules progressively subsamples points of interest by hierarchically grouping points, and uses a custom PointNet architecture to encode points into feature vectors. Because semantic segmentation tasks require point features for all original points, this example uses a series of feature propagation modules to hierarchically interpolate features to original points using an inverse- distance-based interpolation scheme.

The set abstraction module is implemented using the `sampleAndGroup` and `sharedMLP` supporting functions and the feature propagation module is implemented using the `featurePropagation` and `sharedMLP` supporting functions. The `sharedMLP` module is the only learnable module that consists of convolution and instance normalization layers. Use the supporting function `pointNetplusplusModel` to define the entire PointNet++ segmentation model. The supporting functions are listed at the end of this example.

Define PointNet++ Model Parameters

The `sampleAndGroup` function of the set abstraction module is parameterized by the number of clusters, number of samples per cluster, and radius of each cluster, whereas the `sharedMLP` function implements a custom PointNet architecture that is parameterized by using the number of input channels and the hidden channel sizes.

This example uses hyperparameter values tuned on the DALES data set. If you want to apply PointNet++ to a different data set, you must perform additional hyperparameter tuning.

Set the parameters of the first set abstraction module. Set the number of clusters to 1024, the number of samples in each cluster to 128, and the radius of each cluster to 0.1. Also set the PointNet model input channel size to three and the hidden channel sizes to 32, 32, and 64.

```
inputChannelSize = 3;
hiddenChannelSize = [32,32,64];
nontrainable.nClusters1 = 1024;
nontrainable.nSamples1 = 128;
nontrainable.radius1 = 0.1;
trainable.SharedMLP1 = initializeSharedMLP(inputChannelSize,hiddenChannelSize);
```

Set the parameters of the second set abstraction module. Set the number of clusters to 256, the number of samples in each cluster to 64, and the radius of each cluster to 0.2. Also set the PointNet model input channel size to 67 and the hidden channel sizes to 64, 64, and 128.

```
inputChannelSize = 67;
hiddenChannelSize = [64,64,128];
nontrainable.nClusters2 = 256;
nontrainable.nSamples2 = 64;
nontrainable.radius2 = 0.2;
trainable.SharedMLP2 = initializeSharedMLP(inputChannelSize,hiddenChannelSize);
```

Set the parameters of the third set abstraction module. Set the number of clusters to 64, the number of samples in each cluster to 32, and the radius of each cluster to 0.4. Also set the PointNet model input channel size to 131 and the hidden channel sizes to 128, 128, and 256.

```
inputChannelSize = 131;
hiddenChannelSize = [128,128,256];
nontrainable.nClusters3 = 64;
nontrainable.nSamples3 = 32;
nontrainable.radius3 = 0.4;
trainable.SharedMLP3 = initializeSharedMLP(inputChannelSize,hiddenChannelSize);
```

Set the parameters of the fourth set abstraction module. Set the number of clusters to 16, the number of samples in each cluster to 32, and the radius of each cluster to 0.8. Also set the PointNet model input channel size to 259 and the hidden channel sizes to 256, 256, and 512.

```
inputChannelSize = 259;
hiddenChannelSize = [256,256,512];
nontrainable.nClusters4 = 16;
```

```
nontrainable.nSamples4 = 32;
nontrainable.radius4 = 0.8;
trainable.SharedMLP4 = initializeSharedMLP(inputChannelSize,hiddenChannelSize);
```

Set the parameters of the first feature propagation module. Set the fifth shared MLP model input channel size to 768 and the hidden channel size to 256 and 256.

```
inputChannelSize = 768;
hiddenChannelSize = [256,256];
trainable.SharedMLP5 = initializeSharedMLP(inputChannelSize,hiddenChannelSize);
```

Set the parameters of the second feature propagation module. Set the shared MLP model input channel size to 384 and the hidden channel size to 256 and 256.

```
inputChannelSize = 384;
hiddenChannelSize = [256,256];
trainable.SharedMLP6 = initializeSharedMLP(inputChannelSize,hiddenChannelSize);
```

Set the parameters of the third feature propagation module. Set the shared MLP model input channel size to 320 and the hidden channel size to 256 and 128.

```
inputChannelSize = 320;
hiddenChannelSize = [256,128];
trainable.SharedMLP7 = initializeSharedMLP(inputChannelSize,hiddenChannelSize);
```

Set the parameters of the fourth feature propagation module. Set the shared MLP model input channel size to 128 and the hidden channel size to 128 and 128.

```
inputChannelSize = 128;
hiddenChannelSize = [128,128];
trainable.SharedMLP8 = initializeSharedMLP(inputChannelSize,hiddenChannelSize);
```

Set the final shared MLP model input channel size to 128 and the hidden channel size to 128 and numClasses.

```
inputChannelSize = 128;
hiddenChannelSize = [128,numClasses];
trainable.SharedMLP9 = initializeSharedMLP(inputChannelSize,hiddenChannelSize);
params.trainable = trainable;
params.nontrainable = nontrainable;
```

Specify Training Options

Train for one epoch. Set the initial learning rate to 2e-4 and the L2 regularization factor to 0.01.

```
numEpochs = 1;
miniBatchSize = 1;
learningRate = 2e-4;
l2Regularization = 0.01;
```

Initialize the options for Adam optimization.

```
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.999;
```

Initialize the moving average of the parameter gradients and the element-wise squares of the gradients used by the Adam optimizer.

```
trailingAvg = [];
trailingAvgSq = [];
```

Train Model

Train the model using a CPU or GPU. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). 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) object to load the data in batches of `miniBatchSize` during training.

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

mbq = minibatchqueue(dsTrain,2,...
    "MiniBatchSize",miniBatchSize,...
    "OutputEnvironment",executionEnvironment,...
    "MiniBatchFormat",["SSCB","SSCB"],...
    "DispatchInBackground",dispatchInBackground);
```

Train the model using a custom training loop. For each iteration:

- Read a batch of data.
- Evaluate the model gradients.
- Apply L2 weight regularization.
- Use `adamupdate` to update the model parameters.
- Update the training progress plotter using the helper function `configureTrainingProgressPlotter`, defined at the end of this example.

```
doTraining = false;
if doTraining

    % 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.
            [ptCld, ptLabels] = next(mbq);
```

```

% Evaluate the model gradients and loss using dlfeval and the modelGradients function.
[gradients, loss] = dlfeval(@modelGradients, ptCld, ptLabels, params);

% L2 regularization.
gradients = dlupdate(@(g,p) g + l2Regularization*p,gradients,params.trainable);

% Update the network parameters using the Adam optimizer.
[params.trainable, trailingAvg, trailingAvgSq] = adamupdate(params.trainable, gradients,
    trailingAvg, trailingAvgSq, iteration,...
    learningRate,gradientDecayFactor, squaredGradients);

% Update training plot with new points.
addpoints(lossPlotter, iteration,double(gather(extractdata(loss))));
title("Training Epoch " + epoch + " of " + numEpochs);
drawnow;
end
end
else
% Download pretrained model parameters.
pretrainedNetURL = 'https://www.mathworks.com/supportfiles/lidar/data/trainedPointNetplusplus';

preTrainedZipFile = fullfile(dataFolder,'trainedPointNetplusplusNet.zip');
preTrainedMATFile = fullfile(dataFolder,'trainedPointNetplusplusNet.mat');
if ~exist(preTrainedMATFile,'file')
    if ~exist(preTrainedZipFile,'file')
        disp('Downloading pretrained detector (3.2 MB)...');
        websave(preTrainedZipFile, pretrainedNetURL);
    end
    unzip(preTrainedZipFile,dataFolder);
end

% Load pretrained model.
load(preTrainedMATFile,'params');

% Move model parameters to the GPU if possible and convert to a dlarray.
params.trainable = prepareForPrediction(params.trainable,@(x)dlarray(toDevice(x,canUseGPU)))
end

```

Segment Aerial Point Cloud

Read the full test point cloud.

```

lasReader = lasFileReader(fullfile(testDataFolder,'5175_54395.las'));
[pc,attr] = readPointCloud(lasReader,"Attributes", "Classification");

```

Calculate the number of non-overlapping grid based on gridSize, XLimits, and YLimits of the point cloud.

```

numGridsX = round(diff(pc.XLimits)/gridSize(1));
numGridsY = round(diff(pc.YLimits)/gridSize(2));
xRange = linspace(pc.XLimits(1),pc.XLimits(2),numGridsX+1);
yRange = linspace(pc.YLimits(1),pc.YLimits(2),numGridsY+1);

```

Iterate over all the non-overlapping grids and predict the labels.

```

pcFinal = [];
for r = 1:numGridsX

```



```

for c = 1:numGridsY
    idx = (pc.Location(:,1) >= xRange(r)) & (pc.Location(:,1) < xRange(r+1))...
        & (pc.Location(:,2) >= yRange(c)) & (pc.Location(:,2) < yRange(c+1));
    ptCloudGrid = select(pc,idx);

    % Select fixed number of points and labels from the given point cloud.
    ptCloudGrid = selectPoints(ptCloudGrid,numPoints);

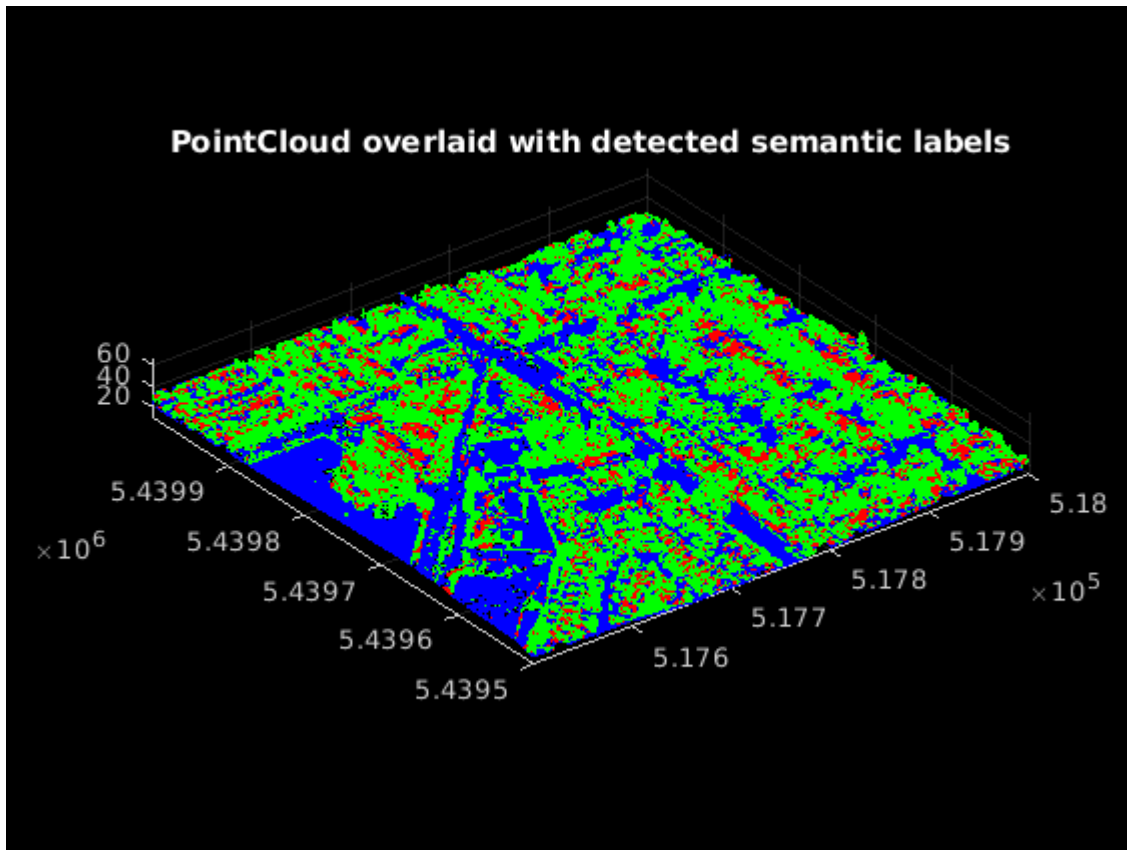
    % Apply the preprocessing.
    ptCloud = normalizePointCloud(ptCloudGrid);
    loc = ptCloud{1,1};
    X1 = dlarray(loc,'SSCB');
    if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
        X1 = gpuArray(X1);
    end

    % Get the output predictions.
    op = pointNetplusModel(X1,params.trainable,params.nontrainable);
    [~,opLabels] = max(gather(extractdata(op)),[],3);
    color = cmap(opLabels,:);
    ptCloudGrid.Color = uint8(color);

    pcFinal = [pcFinal;ptCloudGrid];
end
end

ptCloudOut = pccat(pcFinal);
color = 255.*ptCloudOut.Color;
figure;
pcshow(ptCloudOut.Location,color);
title('PointCloud overlaid with detected semantic labels');

```



Evaluate Network

Evaluate the network performance on the test data. Use the `evaluateSemanticSegmentation` function to compute the semantic segmentation metrics from the test set results.

```
confusionMatrix = {};

% Define the number of samples to evaluate the network. Set numSamplesToTest to
% 1100 to evaluate the model on the entire test data set.
numSamplesToTest = 50;

for i = 1:numSamplesToTest
    ptCldPath = fullfile(pcCropTestPath,sprintf("%03d.pcd",i));
    ptCloud = pcread(ptCldPath);
    ptCloud = normalizePointCloud(ptCloud);
    loc = ptCloud{1,1};

    X1 = dldarray(loc,'SSCB');
    if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
        X1 = gpuArray(X1);
    end

    % Get the output predictions.
    op = pointNetplusModel(X1,params.trainable,params.nontrainable);
    [~,predictionLabels] = max(gather(extractdata(op)),[],3);

    targetLabelPath = fullfile(labelsCropTestPath,sprintf("%03d.png",i));
```

```

targetLabels = imread(targetLabelPath);
targetLabels = squeeze(targetLabels);

% Calculate the confusion matrix.
confMatrix = segmentationConfusionMatrix(double(predictionLabels),...
    double(targetLabels),"Classes",1:numClasses);
if size(confMatrix,1) ~= numel(classNames)
    continue
end
confusionMatrix{i} = confMatrix;
end

confusionMatrix = confusionMatrix';
metrics = evaluateSemanticSegmentation(confusionMatrix,classNames)

Evaluating semantic segmentation results
-----
* Selected metrics: global accuracy, class accuracy, IoU, weighted IoU.
* Processed 50 images.
* Finalizing... Done.
* Data set metrics:

    GlobalAccuracy    MeanAccuracy    MeanIoU    WeightedIoU
    -----
    0.76168          0.53687        0.42486    0.60851

metrics =
    semanticSegmentationMetrics with properties:

        ConfusionMatrix: [4x4 table]
    NormalizedConfusionMatrix: [4x4 table]
        DataSetMetrics: [1x4 table]
        ClassMetrics: [4x2 table]
        ImageMetrics: [50x4 table]

```

Model Gradients

The `modelGradients` function takes as input a mini-batch of data `X`, the corresponding target `yTarget`, and the learnable parameters, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss. To compute the gradients, evaluate the `modelGradients` function using the `dlfeval` function in the training loop.

```

function [gradients, loss] = modelGradients(X,yTarget,params)

% Execute the model function.
yPred = pointNetplusModel(X,params.trainable,params.nontrainable);

% Compute the loss.
loss = focalCrossEntropy(yPred,yTarget,'TargetCategories','independent');

% Compute the parameter gradients with respect to the loss.
gradients = dlgradient(loss, params.trainable);
end

```

PointNet++ Model Function

The model function takes as input the model parameters `parameters` and the input data `dLX`. The network outputs the predictions for the labels.

```
function dLY = pointNetplusplusModel(dLX,trainable,nontrainable)

    % Set abstraction module 1
    [points1, pointFeatures1] = sampleAndGroup(dLX,[],nontrainable.nClusters1,...
        nontrainable.nSamples1,nontrainable.radius1);
    dLY = sharedMLP(pointFeatures1,trainable.SharedMLP1.Perceptron);
    pointNetFeatures1 = max(dLY,[],2);

    % Set abstraction module 2
    [points2, pointFeatures2] = sampleAndGroup(points1,pointNetFeatures1,nontrainable.nClusters2
        nontrainable.nSamples2,nontrainable.radius2);
    dLY = sharedMLP(pointFeatures2,trainable.SharedMLP2.Perceptron);
    pointNetFeatures2 = max(dLY,[],2);

    % Set abstraction module 3
    [points3, pointFeatures3] = sampleAndGroup(points2,pointNetFeatures2,nontrainable.nClusters3
        nontrainable.nSamples3,nontrainable.radius3);
    dLY = sharedMLP(pointFeatures3,trainable.SharedMLP3.Perceptron);
    pointNetFeatures3 = max(dLY,[],2);

    % Set abstraction module 4
    [points4, pointFeatures4] = sampleAndGroup(points3,pointNetFeatures3,nontrainable.nClusters4,
        nontrainable.nSamples4,nontrainable.radius4);
    dLY = sharedMLP(pointFeatures4,trainable.SharedMLP4.Perceptron);
    pointNetFeatures4 = max(dLY,[],2);

    % Feature propagation module 1
    pointsFP1 = featurePropagation(points3,points4,pointNetFeatures3,pointNetFeatures4);
    pointNetFP1 = sharedMLP(pointsFP1,trainable.SharedMLP5.Perceptron);

    % Feature propagation module 2
    pointsFP2 = featurePropagation(points2,points3,pointNetFeatures2,pointNetFP1);
    pointNetFP2 = sharedMLP(pointsFP2,trainable.SharedMLP6.Perceptron);

    % Feature propagation module 3
    pointsFP3 = featurePropagation(points1,points2,pointNetFeatures1,pointNetFP2);
    pointNetFP3 = sharedMLP(pointsFP3,trainable.SharedMLP7.Perceptron);

    % Feature propagation module 4
    pointsFP4 = featurePropagation(dLX,points1,[],pointNetFP3);
    dLY = sharedMLP(pointsFP4,trainable.SharedMLP8.Perceptron);

    % Shared MLP
    dLY = sharedMLP(dLY,trainable.SharedMLP9.Perceptron);
    dLY = softmax(dLY);
    dLY = dLarray(dLY,'SSCB');
end
```

Model Parameter Initialization Functions

Initialize Shared Multilayer Perceptron Function

The `initializeSharedMLP` function takes as input the channel size and the hidden channel size, and returns the initialized parameters in a structure. The parameters are initialized using He weight initialization.

```
function params = initializeSharedMLP(inputChannelSize,hiddenChannelSize)
weights = initializeWeightsHe([1 1 inputChannelSize hiddenChannelSize(1)]);
bias = zeros(hiddenChannelSize(1),1,"single");
p.Conv.Weights = darray(weights);
p.Conv.Bias = darray(bias);

p.InstanceNorm.Offset = darray(zeros(hiddenChannelSize(1),1,"single"));
p.InstanceNorm.Scale = darray(ones(hiddenChannelSize(1),1,"single"));

params.Perceptron(1) = p;

for k = 2:numel(hiddenChannelSize)
    weights = initializeWeightsHe([1 1 hiddenChannelSize(k-1) hiddenChannelSize(k)]);
    bias = zeros(hiddenChannelSize(k),1,"single");
    p.Conv.Weights = darray(weights);
    p.Conv.Bias = darray(bias);

    p.InstanceNorm.Offset = darray(zeros(hiddenChannelSize(k),1,"single"));
    p.InstanceNorm.Scale = darray(ones(hiddenChannelSize(k),1,"single"));

    params.Perceptron(k) = p;
end
end

function x = initializeWeightsHe(sz)
fanIn = prod(sz(1:3));
stddev = sqrt(2/fanIn);
x = stddev .* randn(sz);
end

function x = initializeWeightsGaussian(sz)
x = randn(sz,"single") .* 0.01;
end
```

Sampling and Grouping

```
function [newClusters,newpointFeatures] = sampleAndGroup(points,pointFeatures,nClusters,nSamples
% The sampleAndGroup layer first samples the point cloud to a given number of
% clusters and then constructs local region sets by finding neighboring
% points around the centroids using the queryBallPoint function.

    points = extractdata(squeeze(points));
    if ~isempty(pointFeatures)
        pointFeatures = extractdata(squeeze(pointFeatures));
    end

    % Find the centroids for nClusters - nClusters*3.
    centroids = farthestPointSampling(points,nClusters);
    newClusters = points(centroids,:);
```

```
% Find the neareset nSamples for nClusters - nClusters*nSamples*3.
idx = queryBallPoint(points,newClusters,nClusters,nSamples,radius);
newPoints = reshape(points(idx,:),[nClusters,nSamples,3]);

% Normalize the points in relation to the cluster center.
newpointFeatures = newPoints - reshape(newClusters,nClusters,1,3);

if ~isempty(pointFeatures)
    groupFeatures = reshape(pointFeatures(idx,:),...
        [nClusters,nSamples,size(pointFeatures,2)]);
    newpointFeatures = cat(3,newPoints,groupFeatures);
end

newpointFeatures = darray(newpointFeatures,'SSC');
newClusters = darray(newClusters,'SSC');
end
```

Farthest Point Sampling

```
function centroids = farthestPointSampling(pointLocations,numPoints)
% The farthestPointSampling function selects a set of points from input
% points, which defines the centroids of local regions.
% pointLocations - PointCloud locations N-by-3.
% numPoints - Number of clusters to find.
% centroids - centroids of each farthest cluster.

% Initialize initial indices as zeros.
centroids = zeros(numPoints,1);

% Distance from centroid to each point.
distance = ones(size(pointLocations,1),1) .* 1e10;

% Random Initialization of the first point.
farthest = randi([1,size(pointLocations,1)],1);

for i = 1:numPoints
    centroids(i) = farthest;
    centroid = pointLocations(farthest,:);
    dist = sum(((pointLocations - centroid).^2),2);
    mask = dist < distance;
    distance(mask) = dist(mask);
    [~,farthest] = max(distance,[],1);
end
end
```

Query Ball Point

```
function groupIdx = queryBallPoint(XYZ,newXYZ,nClusters,nSamples,radius)
% Given the cluster center, the queryBallPoint finds all points that are
% within a radius to the query point.

N = size(XYZ,1);
groupIdx = reshape(1:N,[1,N]);
groupIdx = repmat(groupIdx,[nClusters,1]);

% Find the distance between the centroids and given points.
sqDist = squareDistance(newXYZ,XYZ);
```

```

% Find the points that are inside the given radius.
groupIdx(sqDist > (radius)^2) = N;
groupIdx = sort(groupIdx,2,"ascend");

% Find the closest nSamples points within the given radius.
groupIdx = groupIdx(:,1:nSamples);
groupFirst = repmat(groupIdx(:,1),1,nSamples);
mask = (groupIdx == N);
groupIdx(mask) = groupFirst(mask);
end

Feature Propagation

function newPoints = featurePropagation(points1,points2,pointNetFeatures1,pointNetFeatures2)
% Use the inverse distance weighted average based on the k nearest neighbors to
% interpolate features.

points1 = extractdata(squeeze(points1));
points2 = extractdata(squeeze(points2));
if ~isempty(pointNetFeatures1)
pointNetFeatures1 = extractdata(squeeze(pointNetFeatures1));
end
pointNetFeatures2 = extractdata(squeeze(pointNetFeatures2));

% Find the K nearest neighbors for each point.
dists = squareDistance(points1,points2);
[dists,idx] = sort(dists,2,"ascend");
dists = dists(:,1:3);
idx = idx(:,1:3);

% Calculate the weights for interpolation.
dist_recip = 1./(dists+1e-8);
normFactor = sum(dist_recip,2);
weights = dist_recip./normFactor;

% Perform weighted interpolation.
interpolatedPoints = pointNetFeatures2(idx,:);
interpolatedPoints = reshape(interpolatedPoints,[size(idx,1),size(idx,2),size(pointNetFeatures2,3)]);

interpolatedPoints = interpolatedPoints .* weights;
interpolatedPoints = squeeze(sum(interpolatedPoints,2));

if ~isempty(pointNetFeatures1)
% Calculate the new points.
newPoints = cat(2,pointNetFeatures1,interpolatedPoints);
else
newPoints = interpolatedPoints;
end

newPoints = darray(newPoints,'SCS');
end

% Find the squared distance.
function dist = squareDistance(src,dst)
dist = -2 * (src*permute(dst,[2,1,3]));
tmp1 = sum(src.^2,2);
tmp1 = reshape(tmp1,[size(src,1),1]);
tmp2 = sum(dst.^2,2);

```

```

    tmp2 = reshape(tmp2,[1,size(dst,1)]);
    dist = dist + tmp1 + tmp2;
end

```

Shared Multilayer Perceptron Function

```

function dLY= sharedMLP(dlX,parameters)
% The shared multilayer perceptron function processes the input dlX using a
% series of perceptron operations and returns the result of the last
% perceptron.
    dLY = dlX;
    for k = 1:numel(parameters)
        dLY = perceptron(dLY,parameters(k));
    end
end

```

Perceptron Function

```

function dLY = perceptron(dlX,parameters)
% The perceptron function processes the input dlX using a convolution, a
% instance normalization, and a ReLU operation and returns the output of the
% ReLU operation.

% Convolution.
W = parameters.Conv.Weights;
B = parameters.Conv.Bias;
dLY = dlconv(dlX,W,B);

% Instance normalization.
offset = parameters.InstanceNorm.Offset;
scale = parameters.InstanceNorm.Scale;
dLY = instancenorm(dLY,offset,scale);

% ReLU.
dLY = relu(dLY);
end

```

Supporting Functions

The `normalizePointCloud` function extracts the X, Y, Z point data from the input data and normalizes the data between 0 and 1. The function returns the normalized X, Y, Z data.

```

function data = normalizePointCloud(data)
    if ~iscell(data)
        data = {data};
    end

    numObservations = size(data,1);
    for i = 1:numObservations
        % Scale points between 0 and 1.
        xlim = data{i,1}.XLimits;
        ylim = data{i,1}.YLimits;
        zlim = data{i,1}.ZLimits;

        xyzMin = [xlim(1) ylim(1) zlim(1)];
        xyzDiff = [diff(xlim) diff(ylim) diff(zlim)];

        tmp = (data{i,1}.Location - xyzMin) ./ xyzDiff;
        % Convert the data to SCSB format.
    end
end

```



```

        data{i,1} = permute(tmp,[1,3,2]);
    end
end

function data = onehotEncodeLabels(data,classNames)
    numObservations = size(data,1);
    for i = 1:numObservations
        labels = data{i,1}';
        encodedLabels = onehotencode(labels,2,'ClassNames',classNames);
        data{i,1} = permute(encodedLabels,[1,3,2]);
    end
end

```

prepareForPrediction Function

The `prepareForPrediction` function is used to apply a user-defined function to nested structure data. Use this function to move model parameter and state data to the GPU.

```

function p = prepareForPrediction(p,fcn)

for i = 1:numel(p)
    p(i) = structfun(@(x)invoke(fcn,x),p(i),'UniformOutput',0);
end

    function data = invoke(fcn,data)
        if isstruct(data)
            data = prepareForPrediction(data,fcn);
        else
            data = fcn(data);
        end
    end
end

% Move data to the GPU.
function x = toDevice(x,useGPU)
if useGPU
    x = gpuArray(x);
end
end

```

selectPoints Function

The `selectPoints` function samples the desired number of points. When the point cloud contains more than the desired number of points, the function uses `pcdsample` to randomly select points. Otherwise, the function replicates data to produce the desired number of points.

```

function ptCloudOut = selectPoints(ptCloud,numPoints)
    if ptCloud.Count > numPoints
        ind = 1:ptCloud.Count;
    else
        replicationFactor = ceil(numPoints/ptCloud.Count);
        ind = repmat(1:ptCloud.Count,1,replicationFactor);
    end

    ptCloudOut = select(ptCloud,ind(1:numPoints));
end

```

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

```
function labelColorbar(ax, cmap, classNames)
    colormap(ax, cmap);

    % Add colorbar to current figure.
    c = colorbar(ax);
    c.Color = 'w';

    % Center tick labels and use class names for tick marks.
    numClasses = size(classNames, 1);
    c.Ticks = 1:1:numClasses;
    c.TickLabels = classNames;

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

function cmap = labelColorMap()
% Colormap for the original classes.
cmap = [[10,10,10];
        [0,0,255];
        [0,255,0];
        [255,192,203];
        [255,255,0];
        [255,0,255];
        [255,165,0];
        [139,0,150];
        [255,0,0]];
cmap = cmap./255;
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
```

References

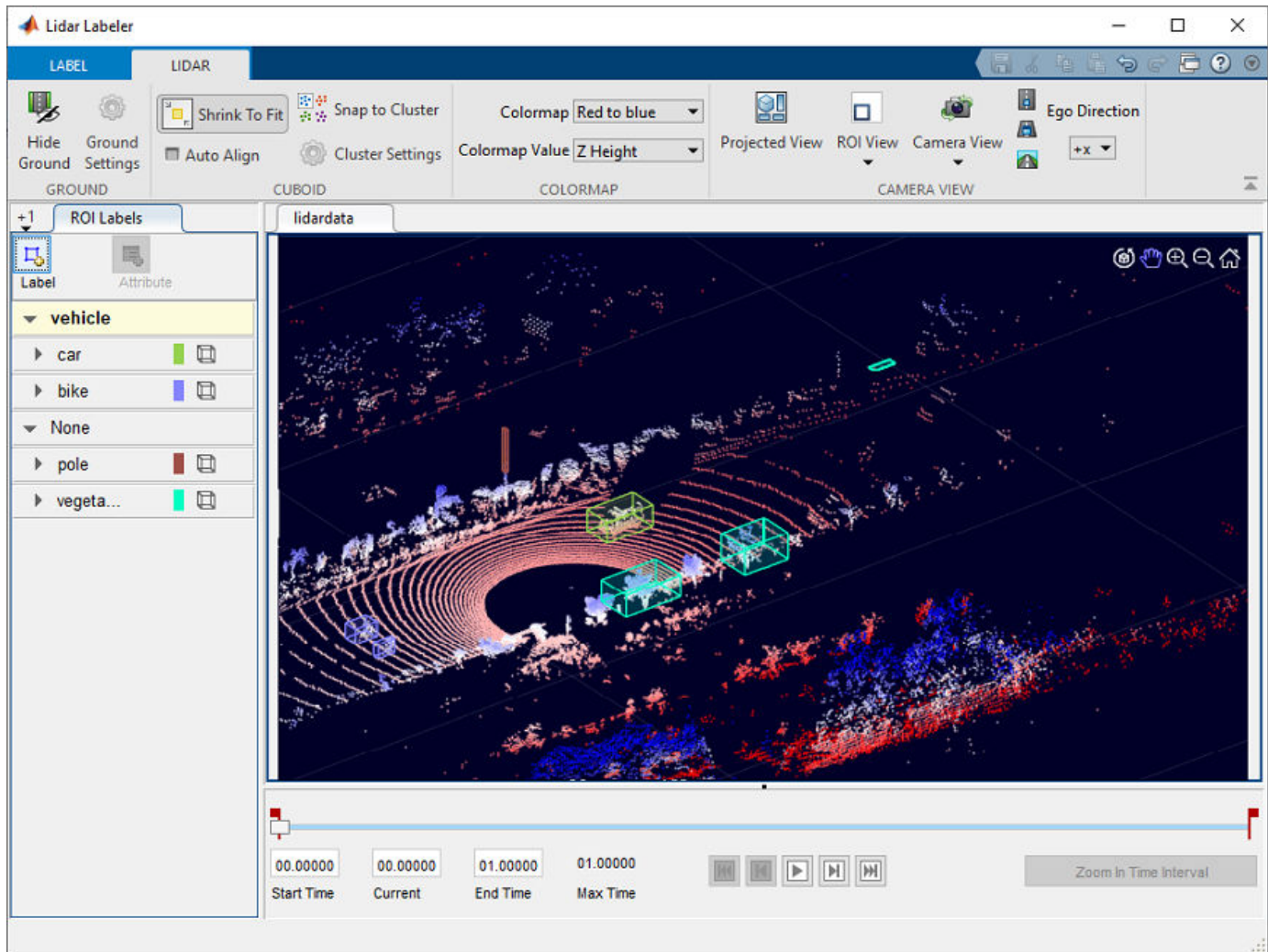
- [1] Varney, Nina, Vijayan K. Asari, and Quinn Graehling. "DALES: A Large-Scale Aerial LiDAR Data Set for Semantic Segmentation." *ArXiv:2004.11985 [Cs, Stat]*, April 14, 2020. <https://arxiv.org/abs/2004.11985>.
- [2] Qi, Charles R., Li Yi, Hao Su, and Leonidas J. Guibas. "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space." *ArXiv:1706.02413 [Cs]*, June 7, 2017. <https://arxiv.org/abs/1706.02413>.

Lidar Labeling

- “Get Started with the Lidar Labeler” on page 2-2
- “Keyboard Shortcuts and Mouse Actions for Lidar Labeler” on page 2-13
- “Use Custom Point Cloud Source Reader for Labeling” on page 2-16

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
```

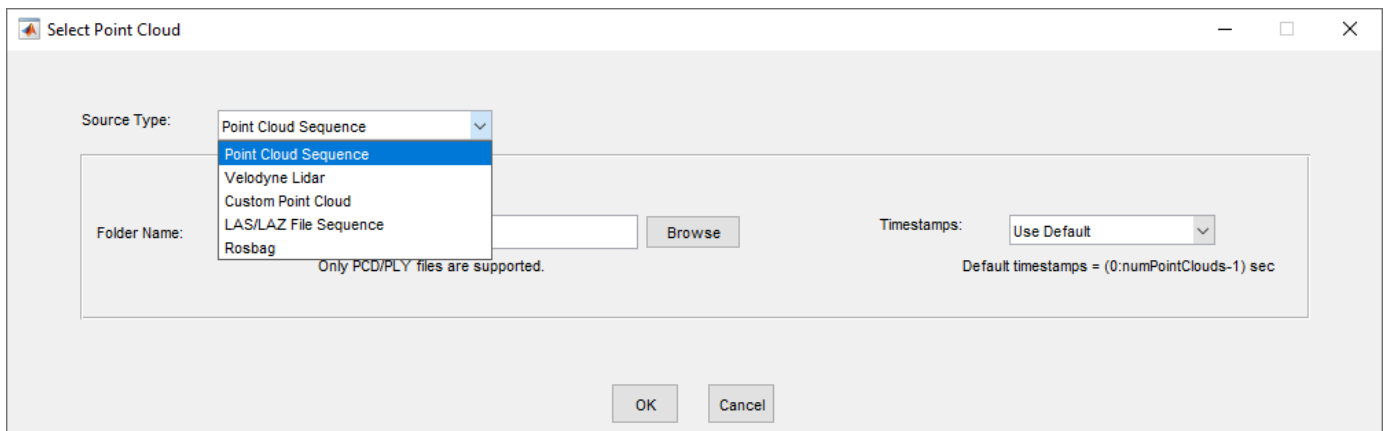
The app opens to an empty session.

Alternatively, you can open the app from the **Apps** tab, under **Image Processing and Computer Vision**.

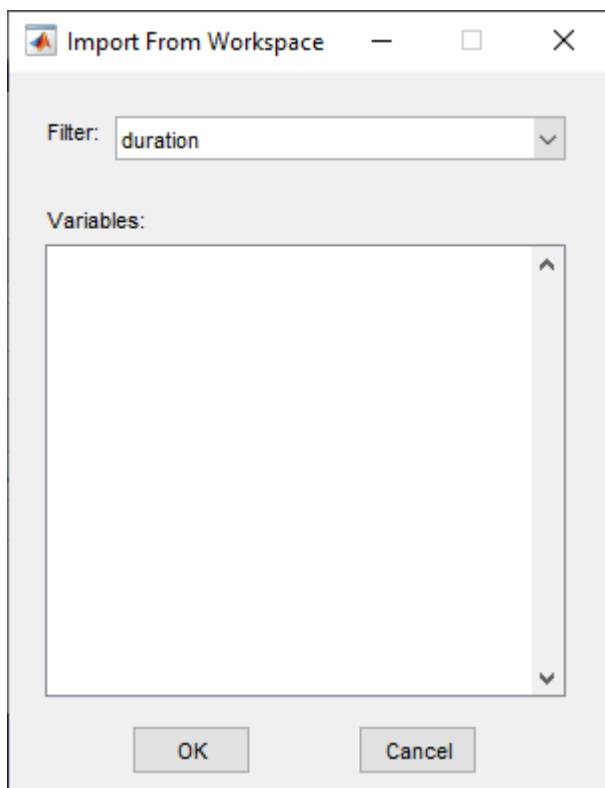
Load Signals from Data Sources

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 **Import > Add Point Cloud**. The **Select Point Cloud** 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 Point Cloud** window, click **OK**. The point cloud sequence loads into the app.

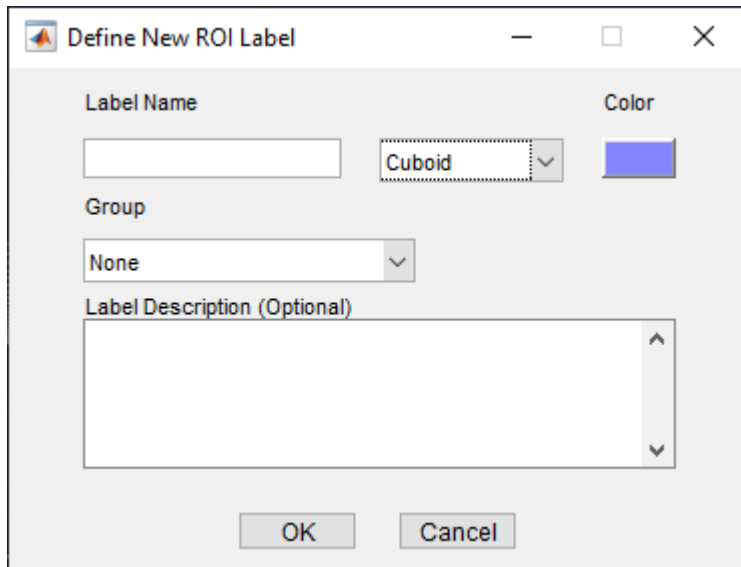
Create Labels and Attributes

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”.

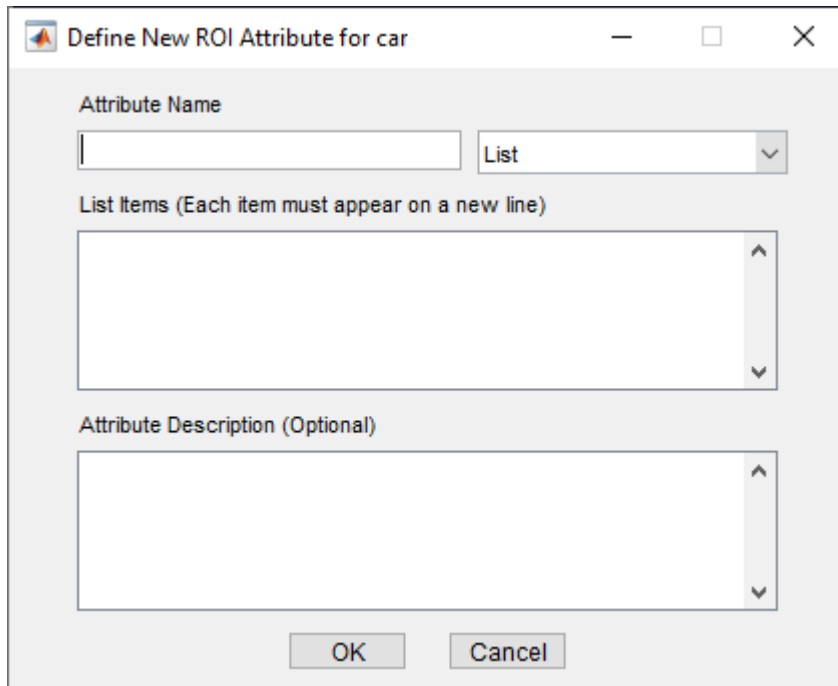
Create an ROI Attribute

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**.



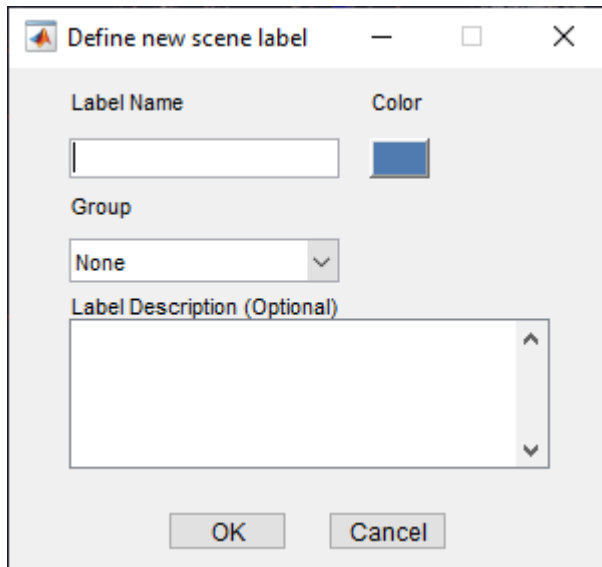
For more details about these attributes, see “ROI Labels and Attributes”.

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.



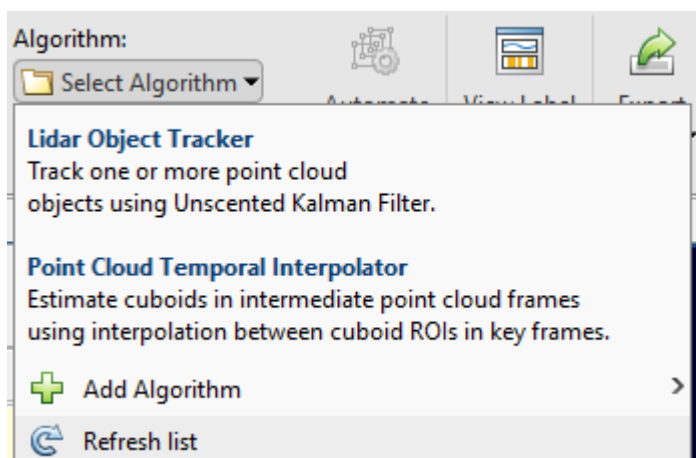
- 4 From the Group list, select New Group and provide a name for the group. Adding labels to groups is optional.
- 5 The **Scene Labels** pane shows the scene label definition.

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 toolbar, in the **Automate Labeling** section, click **Select Algorithm**.



- 3 Select an algorithm for automation.

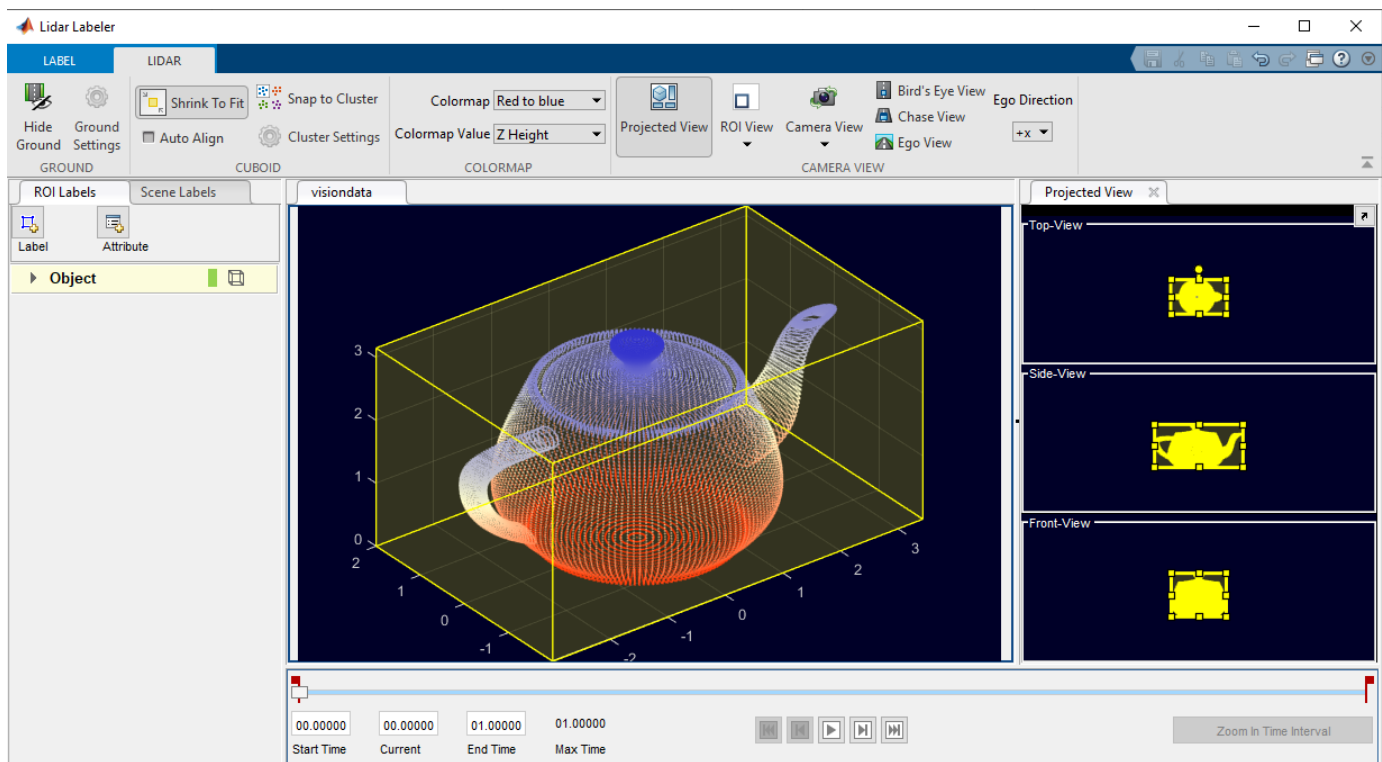
- Click **Automate** and then follow the automation instructions in the right pane of the app.

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.

Projected View

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
--	-------------------------------------

Camera View

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.

ROI View

You can define and view a region of interest (ROI) in the point cloud using the **ROI View**, and then select **Select ROI**.

The app opens the **Adjust ROI Limits** dialog box, which contains the ROI parameters.

To specify x -, y -, and z -axes limits for the ROI, drag the corresponding minimum and maximum value sliders. Alternatively, you can type new minimum or maximum values in the corresponding text boxes. You can also adjust the displayed point size of the point cloud using the **Point Size** parameter. Use this to improve visualizations of sparse point clouds by increasing their point size. If you want to return to the full view of the point cloud, click **ROI View** and select **Full View**.

Sync Image Viewer

Connect an external tool to the application to display time-synchronized images for use as reference while labeling. See the `lidar.syncImageViewer.SyncImageViewer` class. The following example shows how to connect an external image display to the **Lidar Labeler**.

Connect Image Display to Lidar Labeler

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

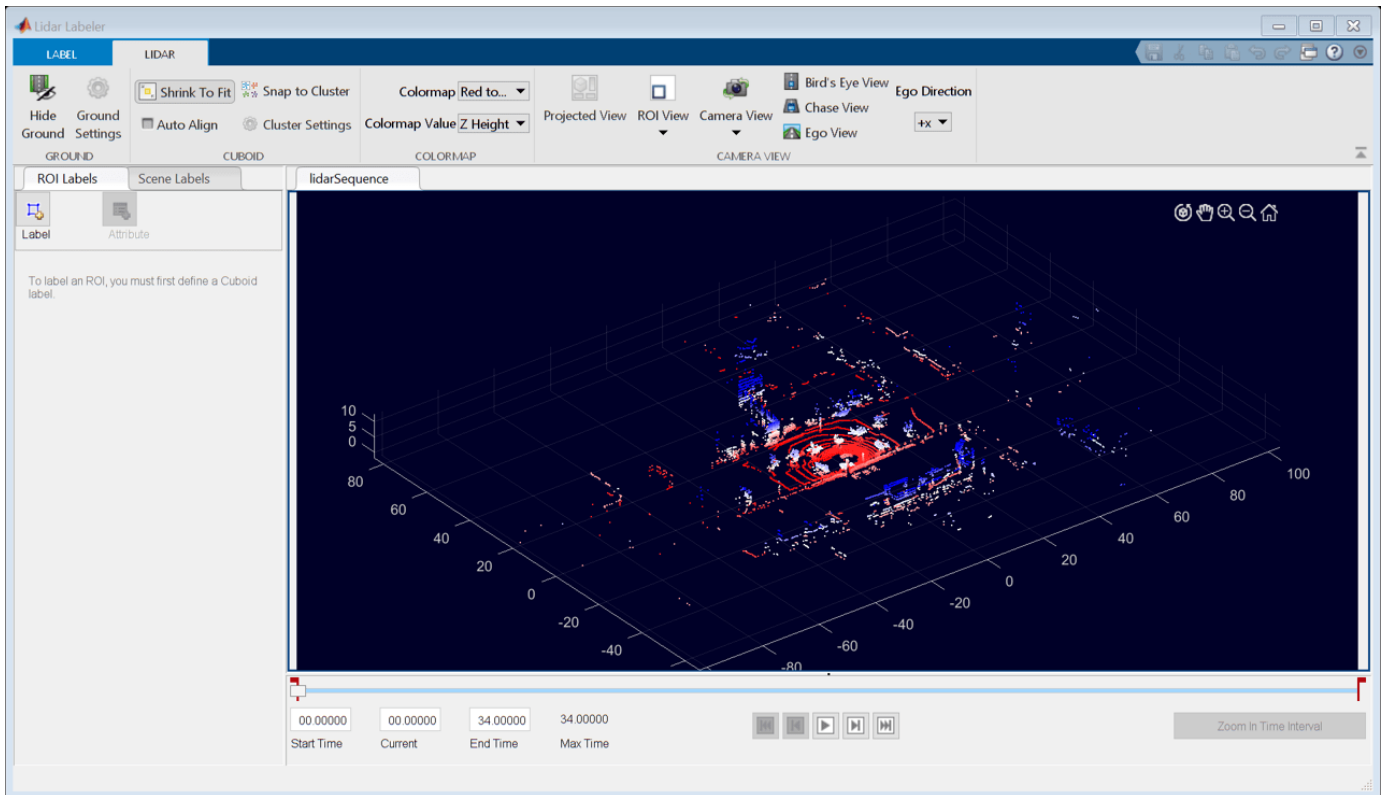
Specify the name of the lidar data to load into the app.

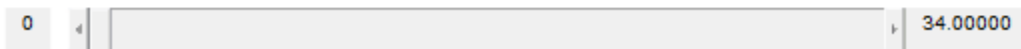
```
sourceName = fullfile('lidarSequence');
```

Connect the video display to the app and display synchronized data.

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

2 Lidar Labeling





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 2-13

Keyboard Shortcuts and Mouse Actions for Lidar Labeler

Note On Macintosh platforms, use the **Command (⌘)** key instead of **Ctrl**.

Label Definitions

Task	Action
Navigate through ROI labels and their groups in the ROI Label Definition pane.	Up or Down arrow
Navigate through scene labels and their groups in the Scene Label Definition pane,	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	<ul style="list-style-type: none"> PC: End Mac: Hold Fn and press the right arrow
Go to the first frame	<ul style="list-style-type: none"> 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: <ol style="list-style-type: none"> 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

Use Custom Point Cloud Source Reader for Labeling

The **Lidar Labeler** app enables you to label ground truth data in point clouds. You can use a custom reader to import the data. First, create a custom reader function. Then, load the custom reader function and corresponding point cloud data source into the **Lidar Labeler**.

Create Custom Reader Function

Specify a custom reader as a function handle. The custom reader must have this syntax.

```
outputPointCloud = readerFcn(sourceName,currentTimestamp)
```

In this example, `readerFcn` is the name of the custom reader function.

The custom reader function loads a point cloud from `sourceName`, which corresponds to the current timestamp specified by `currentTimestamp`. For example, suppose you want to load the point cloud at the third timestamp for a timestamp duration vector that runs from 1 to 5 seconds. To specify `currentTimestamp`, at the MATLAB command prompt, enter this code.

```
timestamps = seconds(1:5);  
currIdx = 3;  
currentTimestamp = timestamps(currIdx);
```

`outputPointCloud` from the custom reader function must be a `pointCloud` object. `currentTimestamp` is a scalar value that corresponds to the current frame that the function is executing.

Import Data Source into Lidar Labeler App

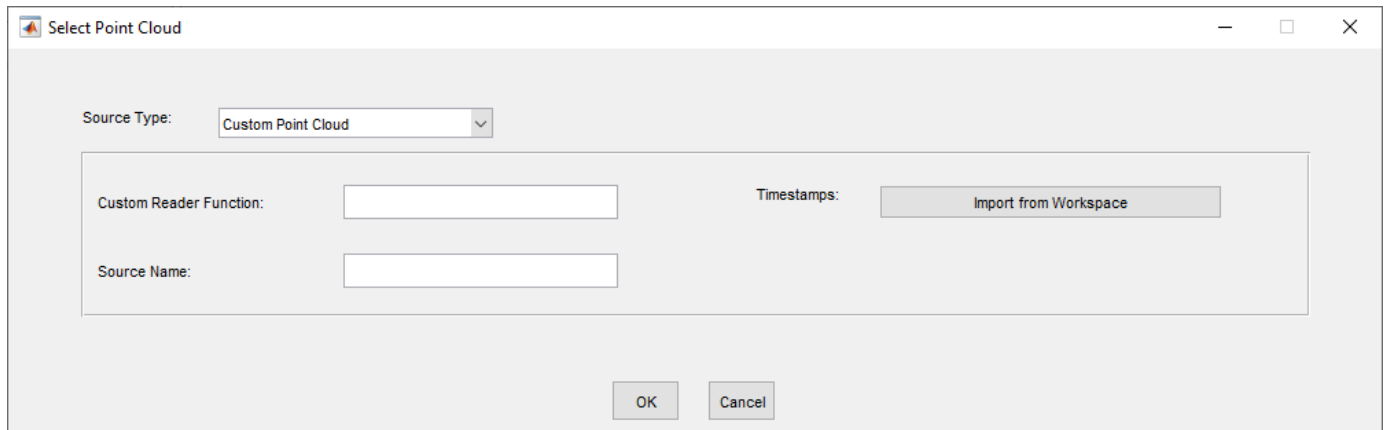
To import a custom data source into the app, first create a structure. This structure stores the function handle and timestamps. Specify the custom reader function handle that reads the data, and the timestamps.

```
sourceParams = struct();  
sourceParams.FunctionHandle = readerFcn;  
sourceParams.Timestamps = timestamps;
```

To load this structure into the app, at the MATLAB command prompt, enter this code.

```
lidarLabeler(sourceName,@sourceParams.FunctionHandle,sourceParams.Timestamps);
```

Alternatively, on the toolstrip of the **Lidar Labeler** app, select **Import > Add Point Cloud**. Then, in the Select Point Cloud dialog box, choose **Custom Point Cloud** as the **Source Type**. Specify **Custom Reader Function** as the function handle and also specify **Source Name**. In addition, you must import corresponding timestamps from the MATLAB workspace.



See Also

Apps

Lidar Labeler

Classes

`lidar.labeler.loading.CustomPointCloudSource` |
`vision.labeler.loading.MultiSignalSource`

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
- “Calibration Guidelines and Procedure” on page 3-10

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 “I/O”. For more information about streaming data from Velodyne® 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 “Preprocessing”.

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 convert unorganized point clouds to organized point clouds with the “Unorganized to Organized Conversion of Point Clouds Using Spherical Projection” on page 1-173 workflow.

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 1-75.
- **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 1-41.
- **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 SLAM Overview”
- “Getting Started with Point Clouds Using Deep Learning”

Coordinate Systems in Lidar Toolbox

Lidar Toolbox uses these coordinate systems:

- **World** — 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.

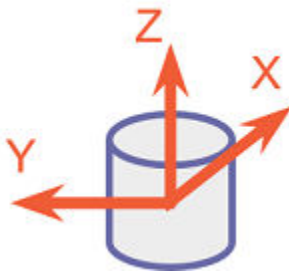
World Coordinate System

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.

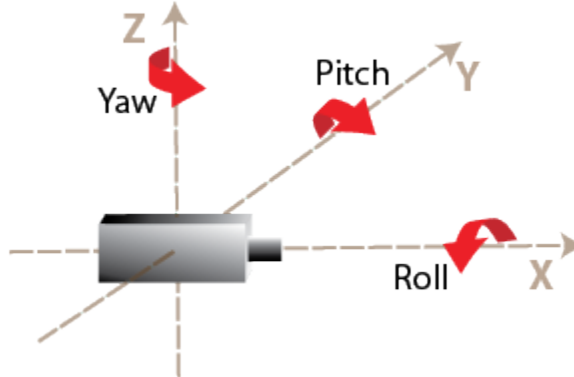
Sensor Coordinate System

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.

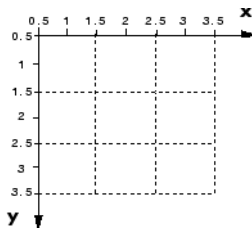


The yaw, pitch, and roll angles of sensors follow an ISO convention. These angles are clockwise-positive when looking in the positive direction of the z -, y -, and x -axes, respectively.



Spatial Coordinate System

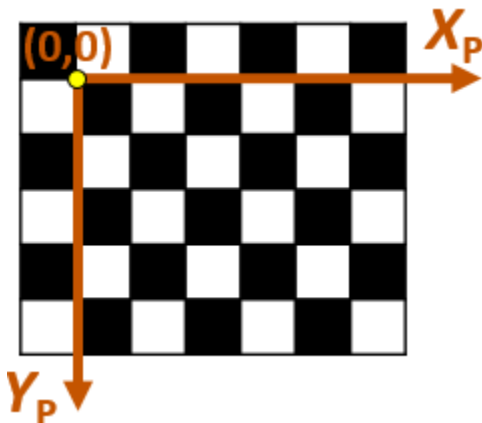
Spatial coordinates enable you to specify a location in an image with greater granularity than pixel coordinates. In the pixel coordinate system, each pixel is treated as a discrete unit, uniquely identified by an integer row and column pair, such as (3,4). In the spatial coordinate system, locations 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”

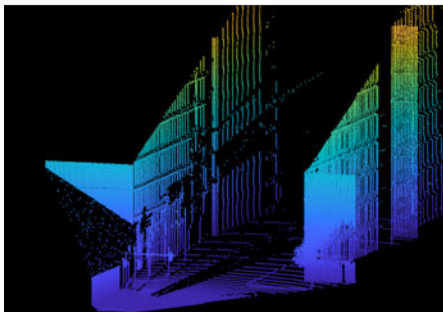
What Is Lidar Camera Calibration?

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:

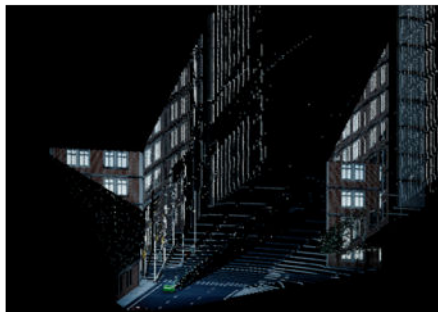
- Calibration for intrinsic parameters
- Calibration for extrinsic parameters between the lidar and camera

The intrinsic parameters of the lidar sensors are calibrated in advance by the manufacturers.

Lidar point cloud data

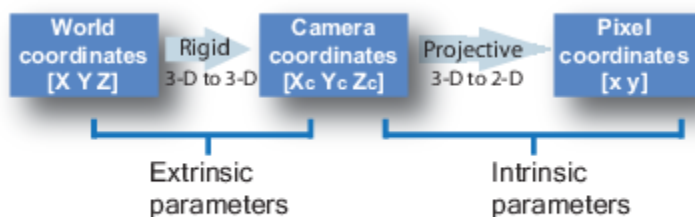


Fused lidar and camera data

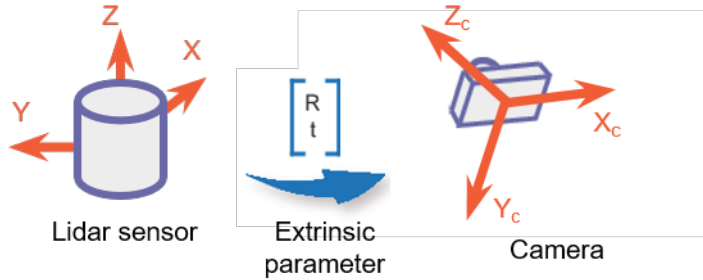


Extrinsic Calibration of Lidar and Camera

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.

$$\underbrace{[x \ y \ 1]}_{\text{Image points}} = \underbrace{[X \ Y \ Z \ 1]}_{\text{World points}} \begin{bmatrix} R \\ t \end{bmatrix} K$$

↓ Extrinsic
↓ Intrinsic matrix
Rotation and translation

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.

$f_x = F/p_x$

$f_y = F/p_y$

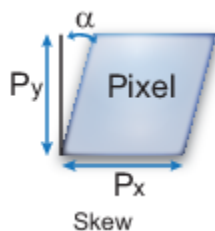
F — Focal length in world units, typically expressed in millimeters.

(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 1-21
- "Detect Vehicles in Lidar Using Image Labels" on page 1-65

More About

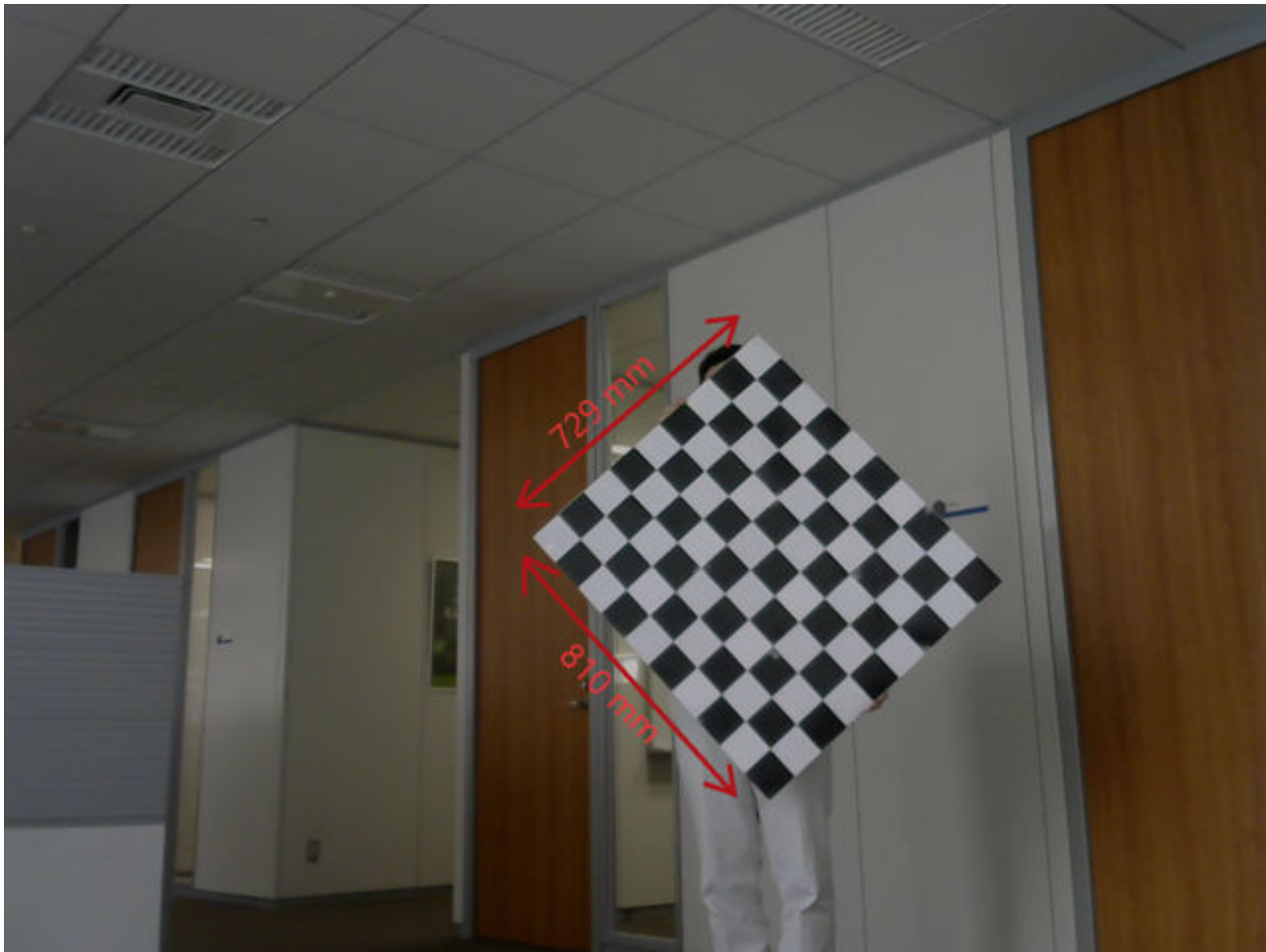
- "Calibration Guidelines and Procedure" on page 3-10

Calibration Guidelines and Procedure

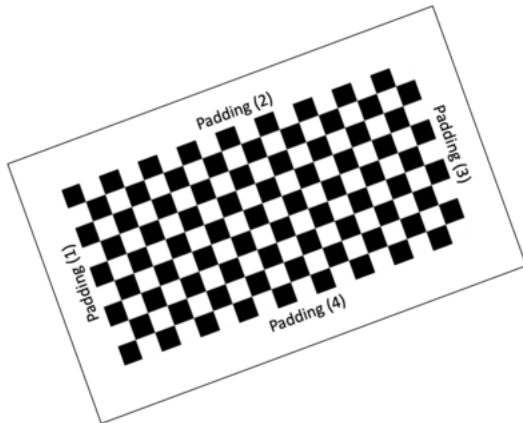
These guidelines and procedures apply to lidar-camera calibration. Follow these guidelines for best results when using the “Lidar and Camera Calibration” on page 1-21 workflow.

Checkerboard Guidelines

- Use the checkerboard function to create a checkerboard image. The checkerboard must be a rectangle.
- Use a checkerboard that contains an even number of squares along one edge and an odd number of squares along the other edge. A square pattern can produce unexpected results for camera extrinsics. A non-square pattern contains two black corners along one side and two white corners along the opposite side. Use these features to determine the orientation of the pattern and the origin point of the checkerboard. The calibrator assigns the x-direction to the longer side.
- Print the checkerboard from end-to-end on a foam board to avoid any measurement errors.



- If you add padding along each side of the checkerboard, measure the values accurately. The values must be specified as a vector to the **Lidar Camera Calibrator** app or `estimateCheckerboardCorners3d` function. The figure shows how the elements of the vector pad the sides.



Checkerboard Padding

Guidelines for Capturing Data

- Capture data from both sensors simultaneously. Capture data with no motion blur effects. Motion blur can degrade the accuracy of the calibration. If you are working with a video recording, carefully capture the point clouds corresponding to those frames.
- Hold the checkerboard target with your arms fully extended, rather than close to your body. Otherwise, parts of your body may appear to be planar with the target. This could lead to inaccurate checkerboard detection. Hold the target from behind, rather than on the edges, for sparse lidar sensors.
- Remove other items on the same plane that the calibrator might cluster with the target.
- The figure below shows many different ways to hold the checkerboard for capturing data. Tilt the checkerboard to expose more of its area to the scan lines from the lidar sensor. Capture at least 10 frames for a proper calibration.
- For high-resolution lidar sensors like the HDL-64 and Ouster OS1-64, you can hold the checkerboard horizontally or vertically while capturing data. However, tilt the checkerboard to a 45 degree angle while capturing data for best results.
- Point cloud data must be saved in the PCD or PLY format.

See Also

Lidar Camera Calibrator | `estimateCheckerboardCorners3d`

Related Examples

- “Lidar and Camera Calibration” on page 1-21
- “Read Lidar and Camera Data from Rosbag File” on page 4-8

More About

- “What Is Lidar Camera Calibration?” on page 3-7

Tutorials

- “Estimate Transformation Between Two Point Clouds Using Features” on page 4-2
- “Match and Visualize Corresponding Features in Point Clouds” on page 4-5
- “Read Lidar and Camera Data from Rosbag File” on page 4-8

Estimate Transformation Between Two Point Clouds Using Features

This example shows how to estimate rigid transformation between two point clouds. In the example, you use feature extraction and matching to significantly reduce the number of points required for estimation. After you use the `extractFPFHFeatures` function to extract fast point feature histogram (FPFH) features from the point clouds, you use the `pcmatchfeatures` function to search for matches in the extracted features. Finally, you use the `estimateGeometricTransform3D` function and the matching features to estimate the rigid transformation.

Preprocessing

Create two point clouds by applying rigid transformation to an input point cloud.

Read the point cloud data into the workspace.

```
rng("default")
ptCld = pcread("highwayScene.pcd");
ptCld.Count
```

```
ans = 65536
```

Downsample the point cloud to improve the computation speed, as it contains around 65,000 points.

```
ptCloud = pcdsample(ptCld,"gridAverage",0.2);
ptCloud.Count
```

```
ans = 24596
```

Create a rigid transformation matrix with a 30-degree rotation and translation of 5 units in x- and y-axes.

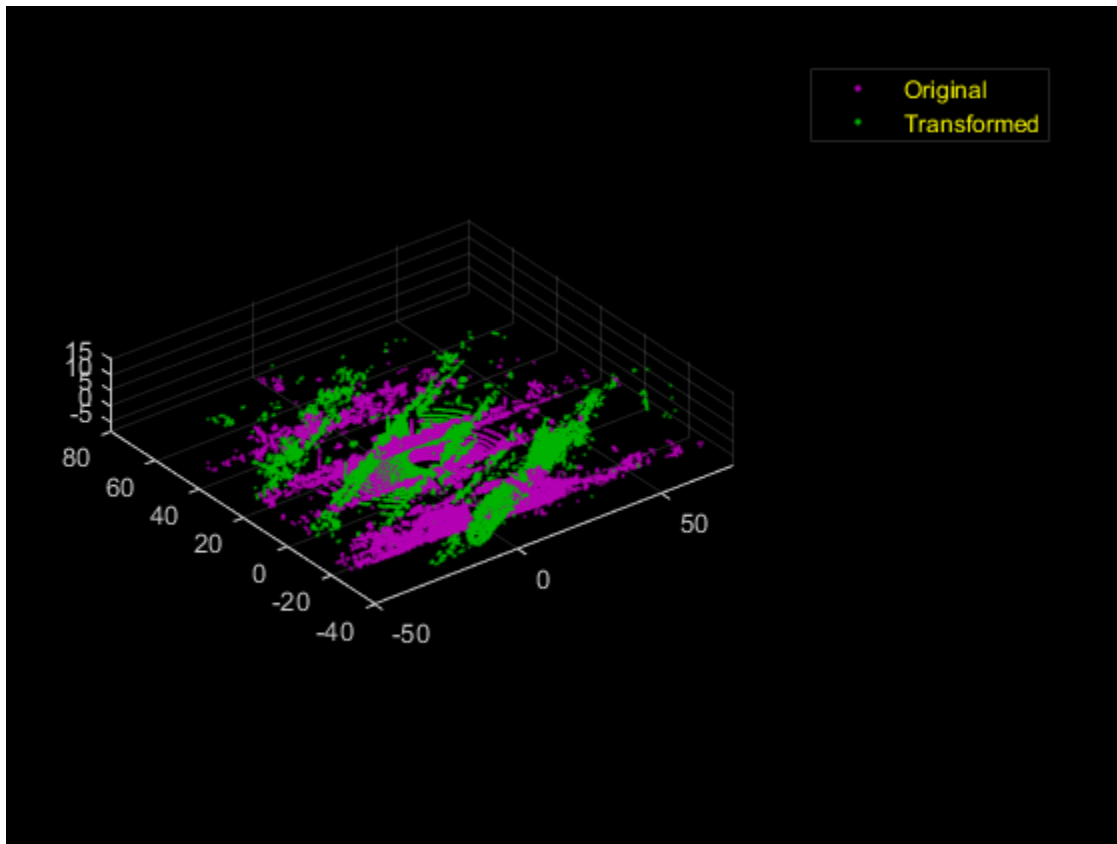
```
rotAngle = 30;
rot = [cosd(rotAngle) sind(rotAngle) 0; ...
      -sind(rotAngle) cosd(rotAngle) 0; ...
      0 0 1];
trans = [5 5 0];
tform = rigid3d(rot,trans);
```

Transform the input point cloud.

```
ptCloudTformed = pctransform(ptCloud,tform);
```

Visualize the two point clouds.

```
pcshowpair(ptCloud,ptCloudTformed)
xlim([-50 75])
ylim([-40 80])
legend("Original", "Transformed", "TextColor", [1 1 0])
```



Feature Extraction and Registration

Extract features from both the point clouds using the `extractFPFHFeatures` function.

```
fixedFeature = extractFPFHFeatures(ptCloud);
movingFeature = extractFPFHFeatures(ptCloudTformed);
```

Find matching features and display the number of matching pairs.

```
[matchingPairs,scores] = pcmatchfeatures(fixedFeature,movingFeature, ...
    ptCloud,ptCloudTformed,"Method","Exhaustive");
length(matchingPairs)
```

```
ans = 1814
```

Select matching points from the point clouds.

```
fixedPts = select(ptCloud,matchingPairs(:,1));
matchingPts = select(ptCloudTformed,matchingPairs(:,2));
```

Estimate the transformation matrix using the matching points.

```
estimatedTform = estimateGeometricTransform3D(fixedPts.Location, ...
    matchingPts.Location,"rigid");
disp(estimatedTform.T)
```

```
    0.8660    0.5000    0.0002    0
   -0.5000    0.8660   -0.0002    0
```

```
-0.0003    0.0000    1.0000    0
 4.9995    5.0022    0.0020    1.0000
```

Display the defined transformation matrix.

```
disp(tform.T)
```

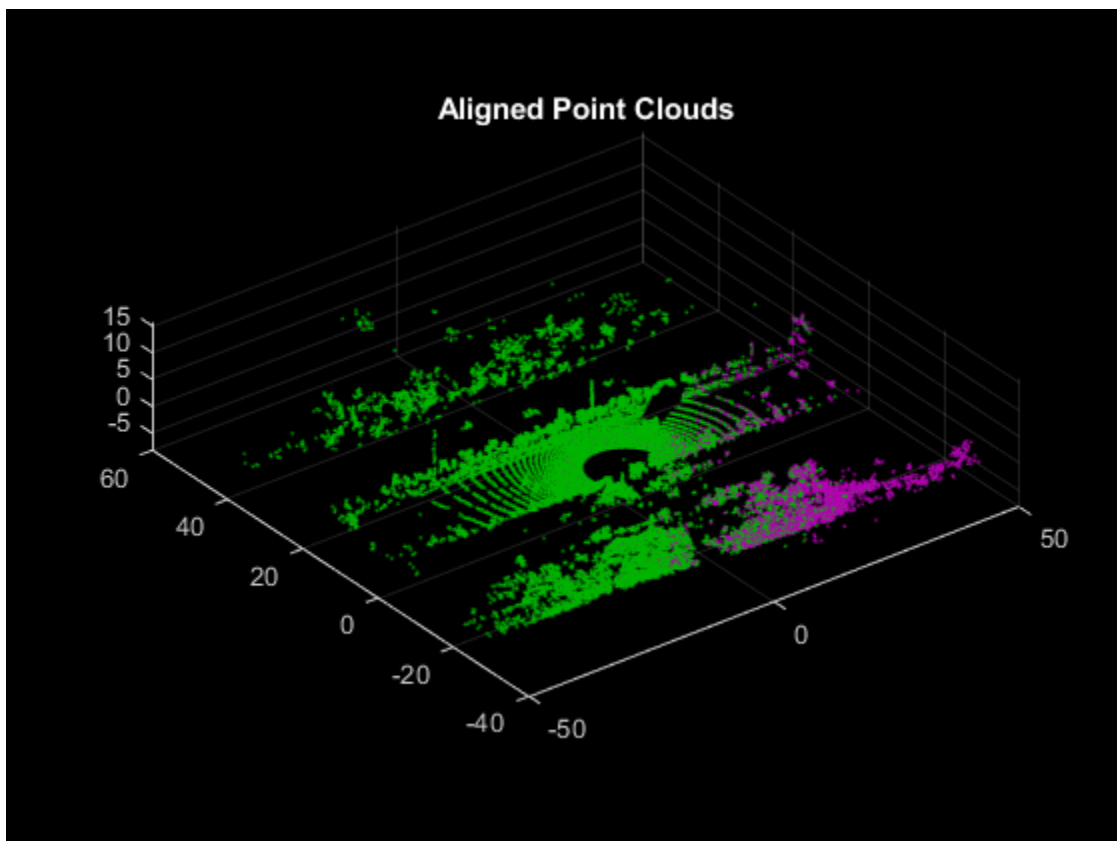
```
 0.8660    0.5000         0         0
-0.5000    0.8660         0         0
         0         0    1.0000         0
 5.0000    5.0000         0    1.0000
```

Use the estimated transformation to retransform ptCloudTformed back to the initial point cloud.

```
ptCloudTformed = pctransform(ptCloudTformed,invert(estimatedTform));
```

Visualize the two point clouds.

```
pcshowpair(ptCloud,ptCloudTformed)
xlim([-50 50])
ylim([-40 60])
title("Aligned Point Clouds")
```



Match and Visualize Corresponding Features in Point Clouds

This example shows how to corresponding features between point clouds using the `pcmatchfeatures` function and visualize them using the `pcshowMatchedFeatures` function.

Create a `velodyneFileReader` object.

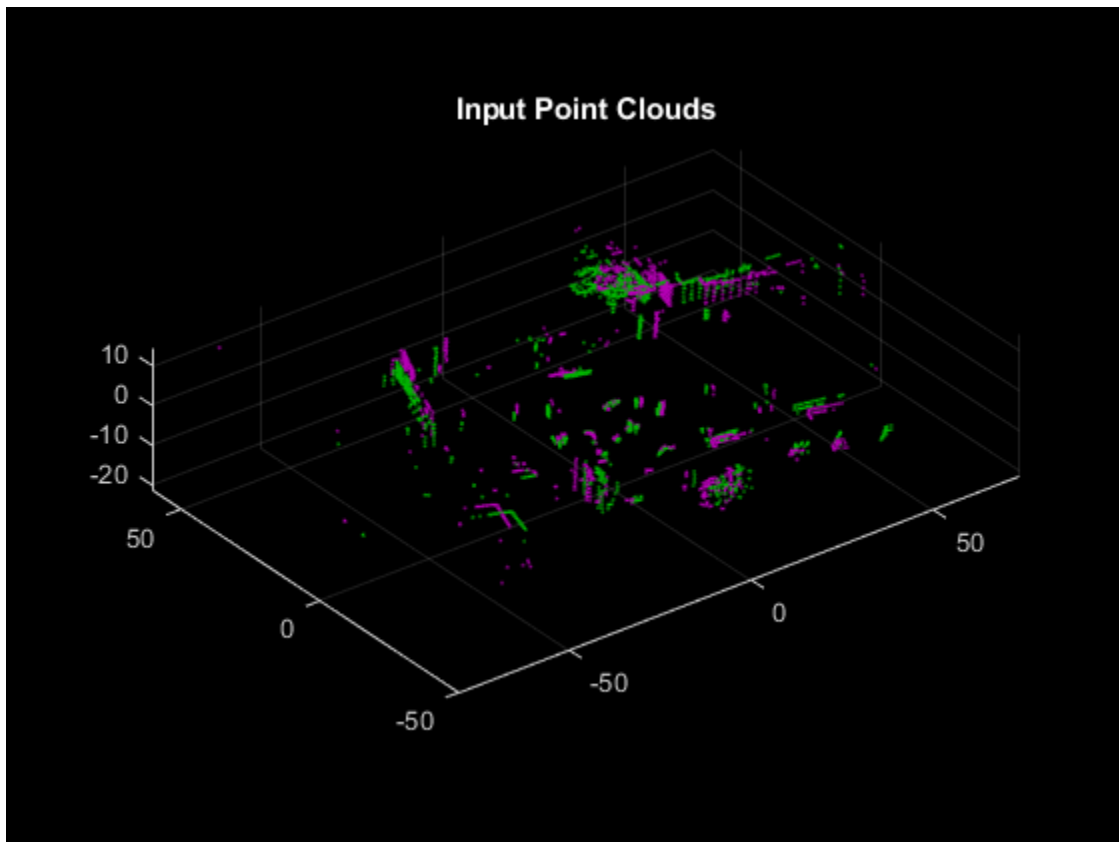
```
veloReader = velodyneFileReader('lidarData_ConstructionRoad.pcap', 'HDL32E');
```

Read two point clouds from the `velodyneFileReader` object into the workspace.

```
frameNumber = 1;
skipFrame = 5;
fixed = readFrame(veloReader, frameNumber);
moving = readFrame(veloReader, frameNumber + skipFrame);
```

Segment and remove the ground plane from the fixed point cloud and moving point cloud.

```
groundPtsIdxFixed = segmentGroundSMRF(fixed);
groundPtsIdxMoving = segmentGroundSMRF(moving);
fixedSeg = select(fixed, ~groundPtsIdxFixed, 'OutputSize', 'full');
movingSeg = select(moving, ~groundPtsIdxMoving, 'OutputSize', 'full');
figure
pcshowpair(movingSeg, fixedSeg)
ylim([-50 60])
title('Input Point Clouds')
```



The superimposed input point clouds are color coded:

- Magenta — Moving point cloud
- Green — Fixed point cloud

Downsample the point clouds to reduce the computation time. Downsampling reduces the number of points to process.

```
fixedDownsampled = pcdsample(fixedSeg, 'gridAverage', 0.2);  
movingDownsampled = pcdsample(movingSeg, 'gridAverage', 0.2);
```

Extract features from the point clouds using the `extractFPFHFeatures` function. The function returns valid indices in both the point clouds. Select the valid points and create new reference point clouds.

```
[fixedFeature, fixedValidInds] = extractFPFHFeatures(fixedDownsampled);  
[movingFeature, movingValidInds] = extractFPFHFeatures(movingDownsampled);  
fixedValidPts = select(fixedDownsampled, fixedValidInds);  
movingValidPts = select(movingDownsampled, movingValidInds);
```

Match features between the point clouds using the extracted features and reference point clouds.

```
indexPairs = pcmatchfeatures(movingFeature, fixedFeature, movingValidPts, ...  
    fixedValidPts);
```

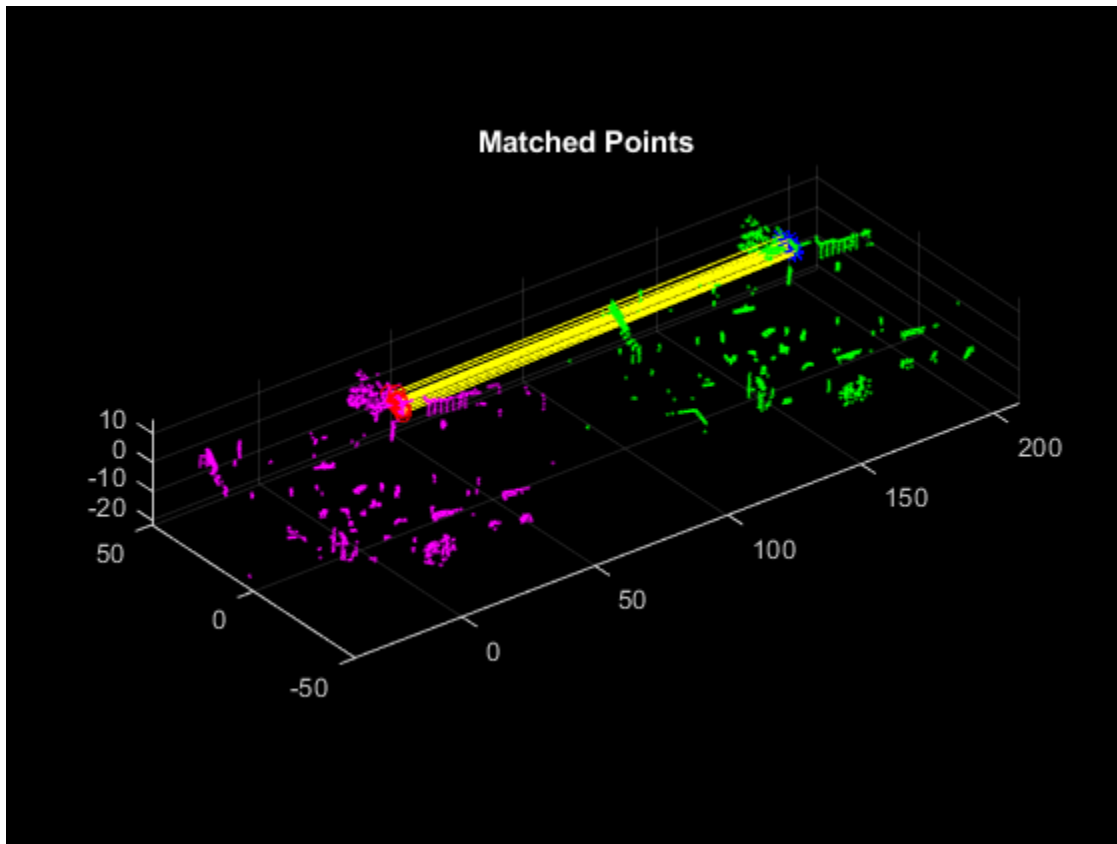
If you do not have the corresponding point cloud data, you can use the two feature sets by themselves. The `pcmatchfeatures` function uses point cloud data to estimate the spatial relation between the points associated with potential feature matches and reject matches based on a spatial relation threshold.

Create point clouds of only the points in each point cloud with matching features in the other point cloud.

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

Visualize the matches.

```
figure  
pcshowMatchedFeatures(movingSeg, fixedSeg, matchedMovingPts, matchedFixedPts, ...  
    'Method', 'montage')  
xlim([-40 210])  
ylim([-50 50])  
title('Matched Points')
```



The matched features and point clouds are color coded to improve visualization:

- Magenta — Moving point cloud
- Green — Fixed point cloud
- Red circle — Matched points in the moving point cloud
- Blue asterisk — Matched points in the fixed point cloud
- Yellow — Line connecting the matched features

Read Lidar and Camera Data from Rosbag File

This example shows how to read and save images and point cloud data from a rosbag file. This example also shows how to prepare the data for lidar camera calibration.

Download the rosbag file from the given URL using the `helperDownloadRosbag` helper function, defined at the end of this example.

```
outputFolder = fullfile(tempdir, 'RosbagFile');
rosbagURL = ['https://ssd.mathworks.com/supportfiles/lidar/data/' ...
    'lccSample.zip'];
helperDownloadRosbag(outputFolder, rosbagURL);
```

Retrieve information from the bag file.

```
path = fullfile(outputFolder, 'lccSample.bag');
bag = rosbag(path);
```

Select image and point cloud messages from the rosbag and select a subset of messages from the file by using the appropriate topic names. You can filter the data by using timestamps as well. For more information, see the `select` (ROS Toolbox) function.

```
imageBag = select(bag, 'Topic', '/camera/image/compressed');
pcBag = select(bag, 'Topic', '/points');
```

Read all the messages.

```
imageMsgs = readMessages(imageBag);
pcMsgs = readMessages(pcBag);
```

To prepare data for lidar camera calibration, the data across both the sensors must be time-synchronized. Create `timeseries` (ROS Toolbox) objects for the selected topics and extract the timestamps.

```
ts1 = timeseries(imageBag);
ts2 = timeseries(pcBag);
t1 = ts1.Time;
t2 = ts2.Time;
```

For accurate calibration, images and point clouds must be captured with the same timestamps. Match the corresponding data from both the sensors according to their timestamps. To account for uncertainty, use a margin of 0.1 seconds.

```
k = 1;
if size(t2,1) > size(t1,1)
for i = 1:size(t1,1)
    [val,indx] = min(abs(t1(i) - t2));
    if val <= 0.1
        idx(k,:) = [i indx];
        k = k + 1;
    end
end
else
for i = 1:size(t2,1)
    [val,indx] = min(abs(t2(i) - t1));
    if val <= 0.1
        idx(k,:) = [indx i];
```



```

        k = k + 1;
    end
end
end

```

Create directories to save the valid images and point clouds.

```

imageFilePath = fullfile(tempdir, 'PointClouds');
pcFilePath = fullfile(tempdir, 'Images');
if ~exist(imageFilePath, 'dir')
    mkdir(imageFilePath);
end
if ~exist(pcFilePath, 'dir')
    mkdir(pcFilePath);
end

```

Extract the images and point clouds. Name and save the files in their respective folders. Save corresponding image and point clouds under the same number.

```

for i = 1:length(idx)
    I = readImage(imageMsgs{idx(i,1)});
    pc = pointCloud(readXYZ(pcMsgs{idx(i,2)}));
    n_strPadded = sprintf( '%04d', i );
    pcFileName = strcat(imageFilePath, '/', n_strPadded, '.pcd');
    imageFileName = strcat(pcFilePath, '/', n_strPadded, '.png');
    imwrite(I, imageFileName);
    pcwrite(pc, pcFileName);
end

```

Launch the Lidar Camera Calibrator app and use the interface to load the data into the app. You can also load the data and launch the app from the MATLAB® command line.

```

checkerSize = 200; %millimeters
padding = [0 0 0 0];
lidarCameraCalibrator(imageFilePath, pcFilePath, checkerSize, padding)

```

Supporting Function

```

function helperDownloadRosbag(outputFolder, rosbagURL)
% Download the data set from the given URL to the output folder.
rosbagFile = fullfile(outputFolder, 'lccSample.bag');
rosbagZipFile = fullfile(outputFolder, 'lccSample.zip');
    if ~exist(rosbagFile, 'file')
        if ~exist(rosbagZipFile, 'file')
            mkdir(outputFolder);
            disp('Downloading the rosbag file (8.5 MB)...');
            websave(rosbagFile, rosbagURL);
        end
        unzip(rosbagZipFile, outputFolder);
    end
end

```

