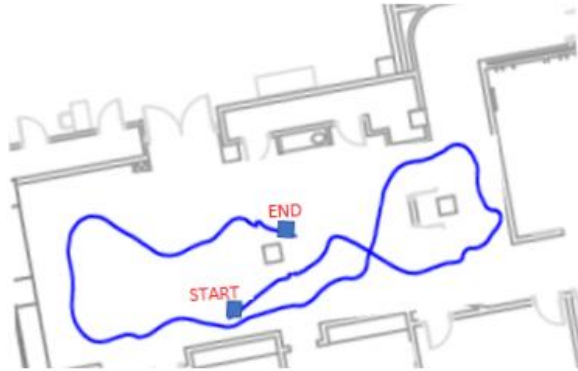


Lidar-SLAM

(Simultaneous Localization And Mapping)



关于SLAM

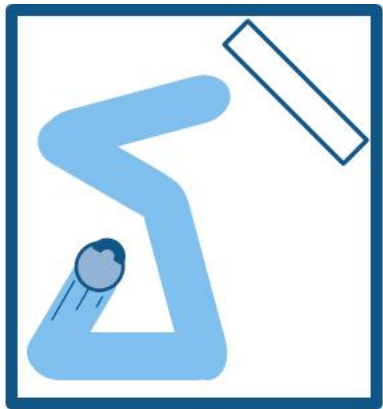
同步定位与地图构建 (**SLAM**) 是自动驾驶汽车所用的一种技术，不仅可以用它构建地图，还可同时在该地图上定位车辆。**SLAM** 算法让汽车能够构建未知环境的地图。工程师们使用地图信息执行路径规划和避障等任务。

SLAM 为何重要

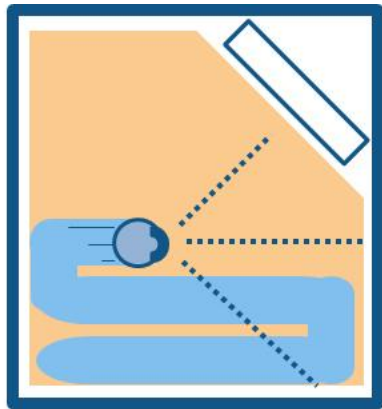
早在多年前，人们就已开始对 **SLAM** 开展技术研究。如今，随着计算机处理速度显著提升，且相机和激光测距仪等低成本传感器大为普及，**SLAM** 更是在越来越多的领域投入实际应用。

家用扫地机器人是代表性的 **SLAM** 应用，没有 **SLAM** 功能的机器人只会在房间里随机移动，无法打扫整个地面空间。此外，这种方法会消耗更多功率，因此电池会更快耗尽。相反，采用 **SLAM** 的机器人可以使用滚轮转数等信息以及来自相机和其他成像传感器的数据，确定所需的移动量。这称为定位。机器人还可以同步使用相机和其他传感器创建其周围障碍物的地图，避免同一区域清洁两次。这称为建图。

SLAM 为何重要



Without SLAM:
Cleaning a room randomly.

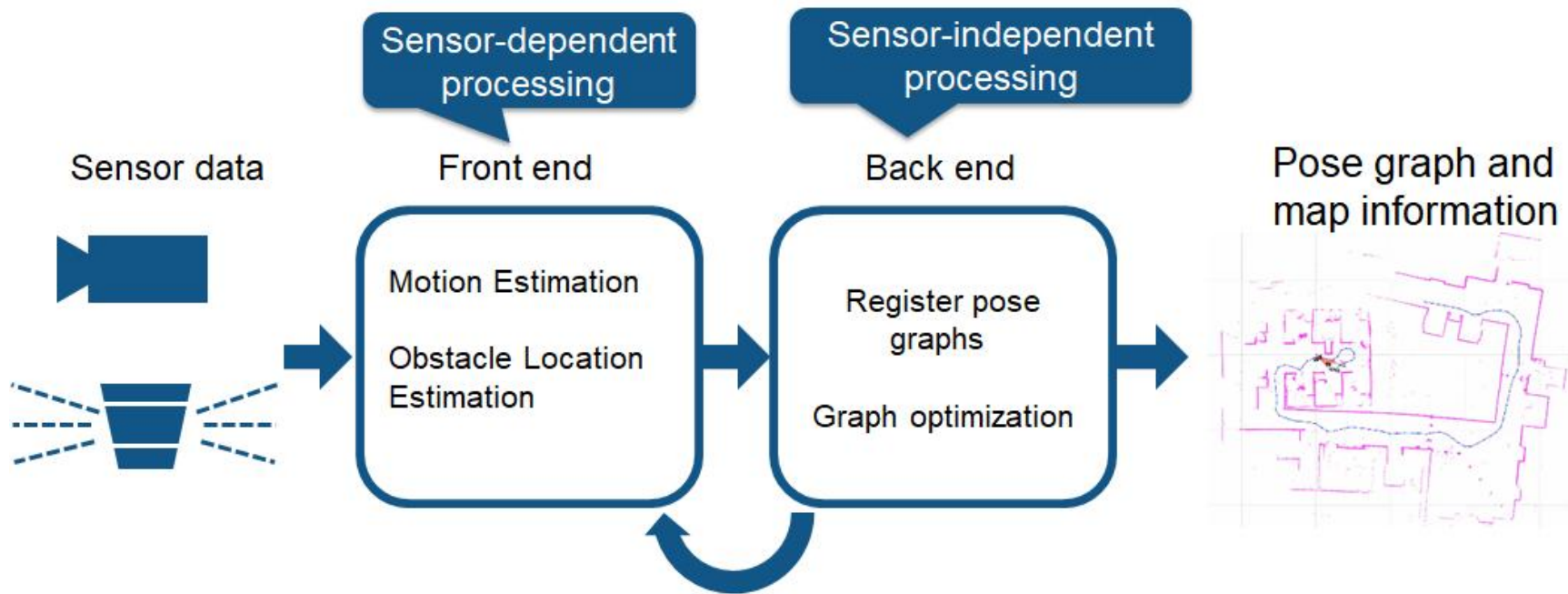


With SLAM:
Cleaning while understanding the room's layout.

SLAM 还可用于许多其他应用场景中，例如让一队移动机器人在仓库中移动并整理货架，让自动驾驶汽车停泊到空车位，或者让无人机在未知环境中完成送货。**MATLAB**提供了 **SLAM** 算法、函数和分析工具来开发各种应用。可以在实现同步定位与地图构建的同时，完成传感器融合、目标跟踪、路径规划和路径跟随等其他任务。

SLAM 工作原理

大致说来，实现 SLAM 需要两类技术。一类技术是传感器信号处理（包括前端处理），这类技术在很大程度上取决于所用的传感器。另一类技术是位姿图优化（包括后端处理），这类技术与传感器无关。



激光雷达 SLAM

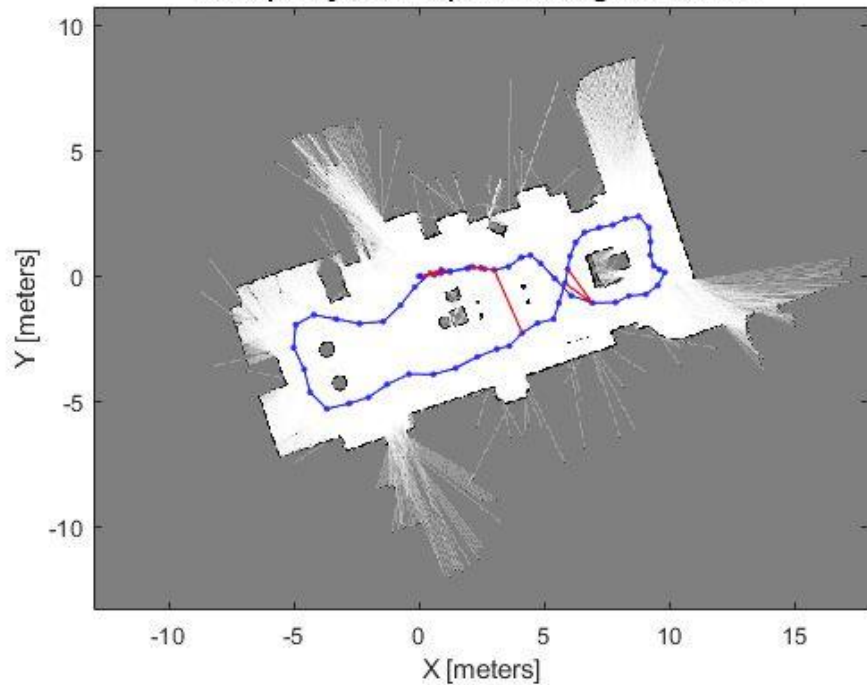
光探测与测距（激光雷达）方法主要使用激光传感器（或距离传感器）。对比相机、ToF 和其他传感器，激光可以使精确度大大提高，常用于自动驾驶汽车和无人机等高速移动运载设备的相关应用。激光传感器的输出值一般是二维 (x, y) 或三维 (x, y, z) 点云数据。激光传感器点云提供了高精度距离测度数据，特别适用于 SLAM 建图。一般来说，首先通过点云匹配来连续估计移动。然后，使用计算得出的移动数据（移动距离）进行车辆定位。对于激光点云匹配，会使用迭代最近点 (ICP) 和正态分布变换 (NDT) 等配准算法。二维或三维点云地图可以用栅格地图或体素地图表示。

激光雷达 SLAM

就密度而言，点云不及图像精细，因此并不总能提供充足的特征来进行匹配。例如，在障碍物较少的地方，将难以进行点云匹配，因此可能导致跟丢车辆。此外，点云匹配通常需要高处理能力，因此必须优化流程来提高速度。鉴于存在这些挑战，自动驾驶汽车定位可能需要融合轮式测距、全球导航卫星系统 (GNSS) 和 IMU 数据等其他测量结果。仓储机器人等应用场景通常采用二维激光雷达 SLAM，而三维激光雷达点云 SLAM 则可用于无人机和自动驾驶。

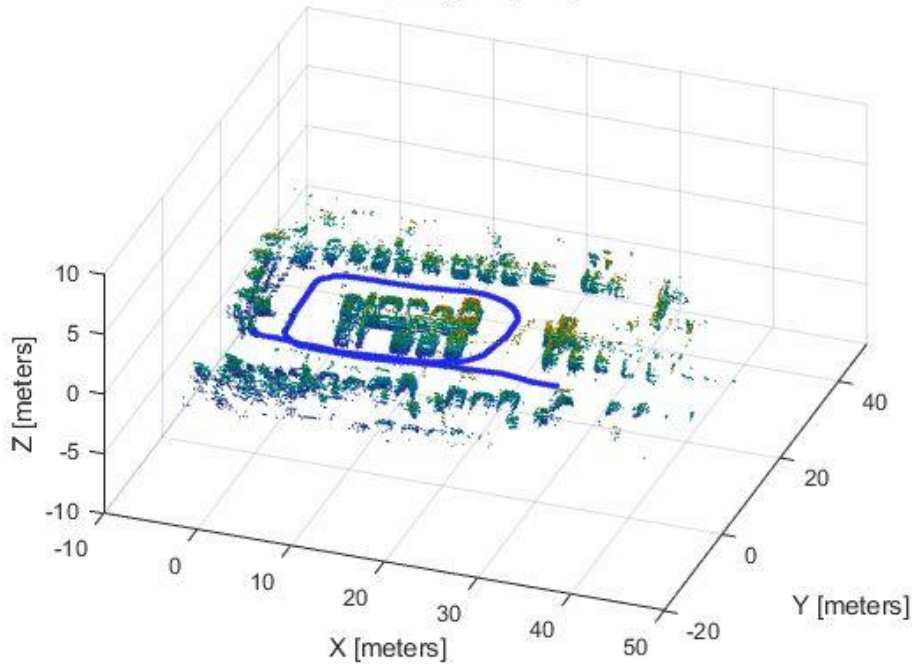
激光雷达 SLAM

Occupancy Grid Map Built Using Lidar SLAM



二维激光雷达 SLAM

Occupancy Map



三维激光雷达 SLAM

视觉 SLAM

视觉 SLAM（又称 **vSLAM**）使用从相机和其他图像传感器采集的图像。视觉 SLAM 可以使用普通相机（广角、鱼眼和球形相机）、复眼相机（立体相机和多相机）和 **RGB-D** 相机（深度相机和 **ToF** 相机）。

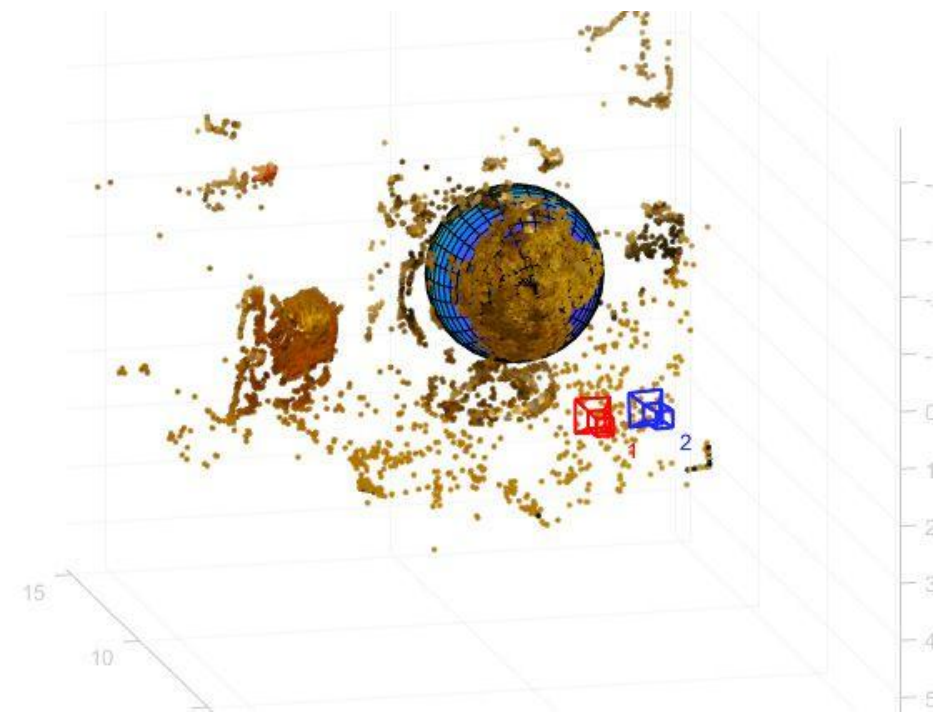
视觉 SLAM 所需的相机价格相对低廉，因此实现成本较低。此外，相机可以提供大量信息，因此还可以用来检测路标（即之前测量过的位置）。路标检测还可以与基于图的优化结合使用，这有助于灵活实现 SLAM。

视觉 SLAM

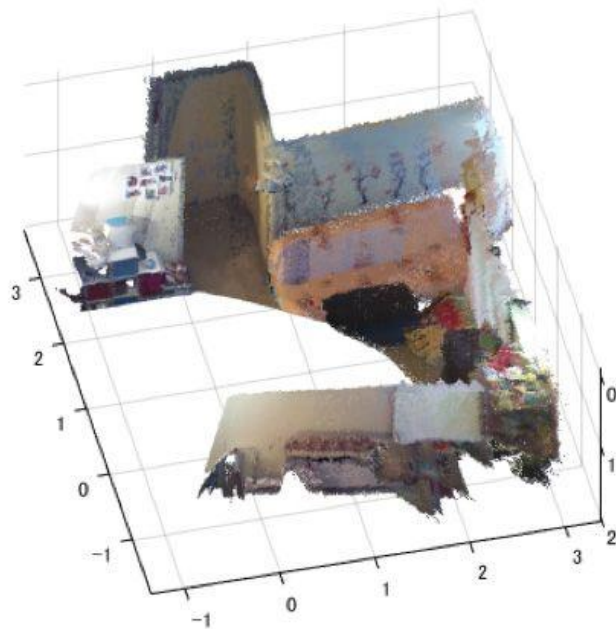
使用单个相机作为唯一传感器的 **vSLAM** 称为单目 **SLAM**，此时难以定义深度。这个问题可以通过以下方式解决：检测待定位图像中的 **AR** 标记、棋盘格或其他已知目标，或者将相机信息与其他传感器信息融合，例如测量速度和方向等物理量的惯性测量单元 (**IMU**) 信息。**vSLAM** 相关的技术包括运动重建 (**SfM**)、视觉测距和捆绑调整。

视觉 **SLAM** 算法可以大致分为两类。稀疏方法：匹配图像的特征点并使用 **PTAM** 和 **ORB-SLAM** 等算法。稠密方法：使用图像的总亮度以及 **DTAM**、**LSD-SLAM**、**DSO** 和 **SVO** 等算法。

视觉 SLAM



运动重建



RGB-D SLAM 点云配准

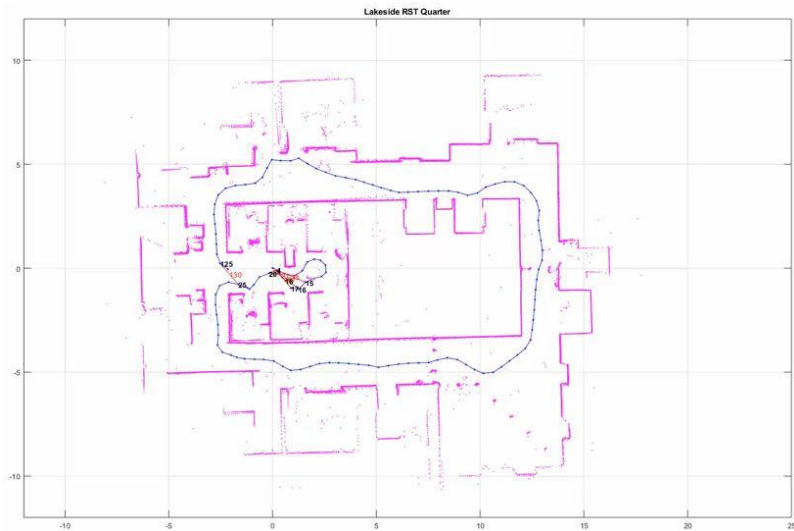
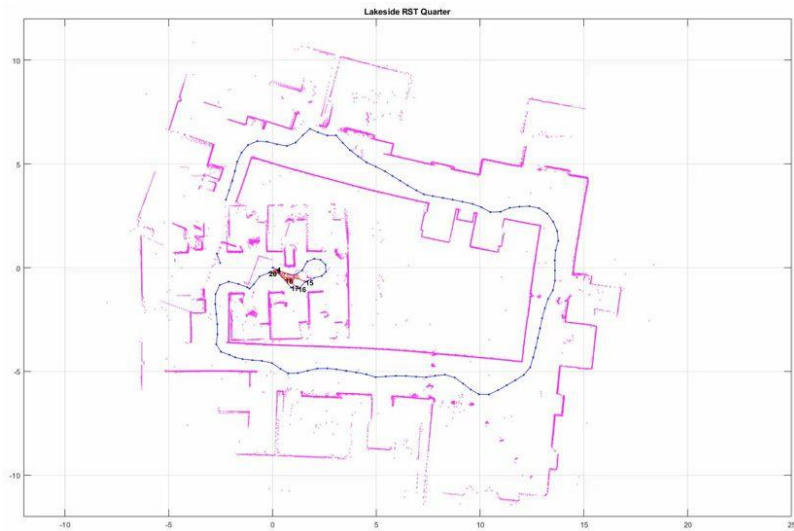
SLAM 面临的常见挑战

虽然 **SLAM** 已在某些场景下投入实际应用，但是仍面临诸多技术挑战，因此难以得到更为广泛的应用。不过，每项挑战都可以凭借特定的对策加以克服。

1. 定位误差累积，导致与实际值产生偏差

SLAM 会估计连续移动，并容许一定的误差。但是误差会随着时间累积，导致与实际值产生明显偏差。误差还会导致地图数据瓦解或失真，让后续搜索变得困难。我们来看一个绕正方形通道行驶的例子。随着误差累积，机器人的起点和终点对不上了。这称为闭环问题。这类位姿估计误差不可避免。我们必须设法检测到闭环，并确定如何修正或抵消累积的误差。

SLAM 面临的常见挑战



对策之一是记住之前到过的某处的某些特征，将其作为路标，从而最小化定位误差。构建位姿图有助于修正误差。将误差最小化问题视为优化问题进行求解，以生成更准确的地图数据。这种优化在视觉 **SLAM** 中称为捆绑调整。

SLAM 面临的常见挑战

2.定位失败，地图上的位置丢失。

图像和点云建图不考虑机器人的移动特征。在某些情况下，这种方法会生成不连续的位置估计。例如，可能会有计算结果显示，以 1 米/秒速度移动的机器人突然向前“瞬移”了 10 米。避免这种定位失败的办法有两种：一是使用恢复算法；二是将运动模型与多个传感器融合，基于传感器数据计算。有多种方法可以实现运动模型的传感器融合。一些常用传感器是惯性测量装置，例如惯性测量单元 (IMU)、航姿参考系统 (AHRS)、惯性导航系统 (INS)、加速度计传感器、陀螺仪传感器和磁力传感器。安装到车辆的轮式编码器通常用于测距。

SLAM 面临的常见挑战

定位失败时，一种恢复对策是记住之前经过的某个位置的关键帧，将其作为路标。搜索路标时，会以特定方法进行特征提取以便高速扫描。有些方法基于图像特征，例如特征袋 (BoF) 和视觉词袋 (BoVW)。近年来，人们也使用深度学习来比较特征距离。

SLAM 面临的常见挑战

3. 图像处理、点云处理和优化带来高计算成本

在车辆硬件上实现 **SLAM** 时，计算成本是个问题。计算通常在处理能力有限的紧凑型低功耗嵌入式微处理器上执行。为了实现准确定位，必须高频率执行图像处理和点云匹配。此外，闭环等优化计算都是高成本计算流程。此处的挑战在于如何在嵌入式微处理器上执行这种高成本处理。

对策之一是并行运行多个不同流程。例如，用于匹配流程前处理的特征提取就相对适合并行运行。使用多核 **CPU** 进行处理时，单指令多数据 (**SIMD**) 计算和嵌入式 **GPU** 在某些情况下可以进一步提升速度。而且，由于位姿图优化可以在相对长的周期里执行，降低其优先级并以规律间隔执行也能提高性能。

实现 SLAM

SLAM 前端的传感器信号和图像处理

- 二维/三维激光雷达处理和扫描匹配
- 三维点云处理和点云配准
- 使用特征袋和视觉词袋进行闭环检测
- 使用深度学习进行目标检测和语义分割

SLAM 后端的二维/三维位姿图

- 生成二维/三维位姿图
- 基于节点和边约束优化位姿图

使用来自 SLAM 算法的输出地图进行路径规划和控制

- 实现 RRT 或 Hybrid A* 等路径规划算法
- 发送控制指令以跟随规划路径并避开障碍

激光SLAM 示例

- Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans
- Perform SLAM Using 3-D Lidar Point Clouds
- Lidar 3-D Object Detection Using PointPillars Deep Learning

The lidarSLAM algorithm

Simultaneous localization and mapping (SLAM) is a general concept for algorithms correlating different sensor readings to build a map of a vehicle environment and track pose estimates. Different algorithms use different types of sensors and methods for correlating data.

The lidarSLAM algorithm uses lidar scans and odometry information as sensor inputs. The lidar scans map the environment and are correlated between each other to build an underlying pose graph of the vehicle trajectory. Odometry information is an optional input that gives an initial pose estimate for the scans to aid in the correlation. Scan matching algorithms correlate scans to previously added scans to estimate the relative pose between them and add them to an underlying pose graph.

The lidarSLAM algorithm

The pose graph contains nodes connected by edges that represent the relative poses of the vehicle. Edges specify constraints on the node as an information matrix. To correct for drifting pose estimates, the algorithm optimizes over the whole pose graph whenever it detects loop closures.

The algorithm assumes that data comes from a vehicle navigating an environment and incrementally getting laser scans along its path. Therefore, scans are first compared to the most recent scan to identify relative poses and are added to the pose graph incrementally. However, the algorithm also searches for loop closures, which identify when the vehicle scans an area that was previously visited.

SLAM with Lidar Scans

This example demonstrates how to implement the Simultaneous Localization And Mapping (SLAM) algorithm on a collected series of lidar scans using pose graph optimization. The goal of this example is to build a map of the environment using the lidar scans and retrieve the trajectory of the robot.

SLAM with Lidar Scans

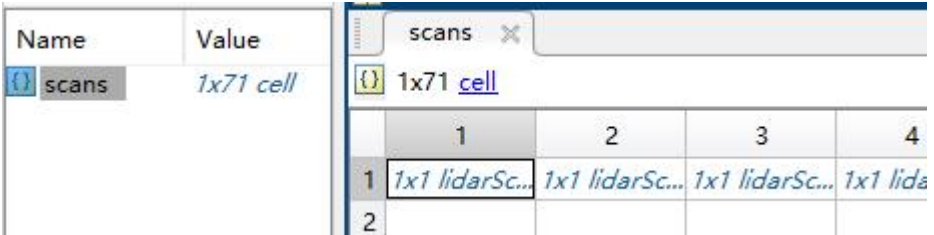
To build the map of the environment, the SLAM algorithm incrementally processes the lidar scans and builds a pose graph that links these scans. The robot recognizes a previously-visited place through scan matching and may establish one or more loop closures along its moving path. The SLAM algorithm utilizes the loop closure information to update the map and adjust the estimated robot trajectory.

Load Laser Scan Data from File

Load a down-sampled data set consisting of laser scans collected from a mobile robot in an indoor environment. The average displacement between every two scans is around 0.6 meters.

The offlineSlamData.mat file contains the scans variable, which contains all the laser scans used in this example.

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



Name	Value
scans	1x71 cell

scans				
	1	2	3	4
1	1x1 lidarSc...	1x1 lidarSc...	1x1 lidarSc...	1x1 lida
2				



Property	Value	Size
Ranges	271x1 double	271x1
Angles	271x1 double	271x1
Cartesian	271x2 double	271x2
Count	271	1x1

Load Laser Scan Data from File

A floor plan and approximate path of the robot are provided for illustrative purposes. This image shows the relative environment being mapped and the approximate trajectory of the robot.



Run SLAM Algorithm, Construct Optimized Map and Plot Trajectory of the Robot

Create a lidarSLAM object and set the map resolution and the max lidar range. This example uses a Jackal™ robot from Clearpath Robotics™. The robot is equipped with a SICK™ TiM-511 laser scanner with a max range of 10 meters. Set the max lidar range slightly smaller than the max scan range (8m), as the laser readings are less accurate near max range. Set the grid map resolution to 20 cells per meter, which gives a 5cm precision.

```
maxLidarRange = 8;  
mapResolution = 20;  
slamAlg = lidarSLAM(mapResolution, maxLidarRange);
```

Name	Value
scans	1x71 cell
mapReso...	20
maxLidar...	8
slamAlg	1x1 lidarSLAM

Property	Value	Size
PoseGraph	1x1 poseGraph	1x1
MapResolution	20	1x1
MaxLidarRange	8	1x1
OptimizationFcn	@optimizePoseGraph	1x1
LoopClosureThreshold	100	1x1
LoopClosureSearchRadius	8	1x1
LoopClosureMaxAttempts	1	1x1
LoopClosureAutoRollback	1	1x1
OptimizationInterval	1	1x1
MovementThreshold	[0,0]	1x2
ScanRegistrationMethod	'BranchAndBound'	1x14
TranslationSearchRange	[4,4]	1x2
RotationSearchRange	1.5708	1x1

Run SLAM Algorithm, Construct Optimized Map and Plot Trajectory of the Robot

The following loop closure parameters are set empirically. Using higher loop closure threshold helps reject false positives in loop closure identification process. However, keep in mind that a high-score match may still be a bad match. For example, scans collected in an environment that has similar or repeated features are more likely to produce false positives. Using a higher loop closure search radius allows the algorithm to search a wider range of the map around current pose estimate for loop closures.

```
slamAlg.LoopClosureThreshold = 210;  
slamAlg.LoopClosureSearchRadius = 8;
```

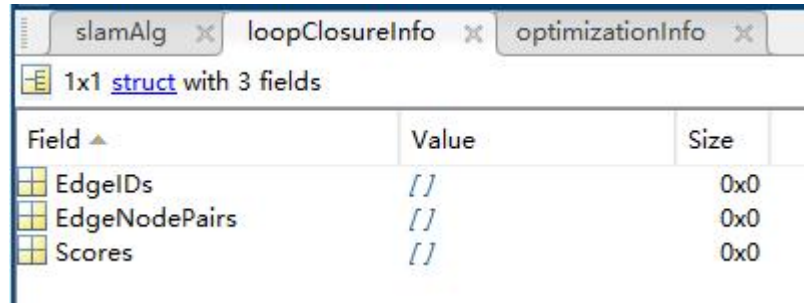
Name	Value
scans	1x71 cell
mapReso...	20
maxLidar...	8
slamAlg	1x1 lidarSLAM

slamAlg		
1x1 lidarSLAM		
Property	Value	Size
PoseGraph	1x1 poseGraph	1x1
MapResolution	20	1x1
MaxLidarRange	8	1x1
OptimizationFcn	@optimizePoseGraph	1x1
LoopClosureThreshold	210	1x1
LoopClosureSearchRadius	8	1x1
LoopClosureMaxAttempts	1	1x1
LoopClosureAutoRollback	1	1x1
OptimizationInterval	1	1x1
MovementThreshold	[0,0]	1x2
ScanRegistrationMethod	'BranchAndBound'	1x14
TranslationSearchRange	[4,4]	1x2
RotationSearchRange	1.5708	1x1

Observe the Map Building Process with Initial 10 Scans

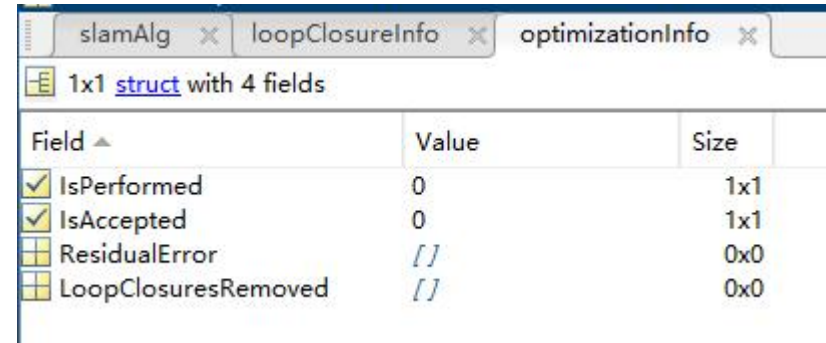
Incrementally add scans to the slamAlg object. Scan numbers are printed if added to the map. The object rejects scans if the distance between scans is too small. Add the first 10 scans first to test your algorithm.

```
for i=1:10
    [isScanAccepted, loopClosureInfo, optimizationInfo] = addScan(slamAlg, scans{i});
    if isScanAccepted
        fprintf('Added scan %d \n', i);
    end
end
```



The image shows the MATLAB variable viewer for the 'slamAlg' variable. It is a 1x1 struct with 3 fields: EdgeIDs, EdgeNodePairs, and Scores. All three fields have empty array values '[]' and a size of 0x0.

Field	Value	Size
EdgeIDs	[]	0x0
EdgeNodePairs	[]	0x0
Scores	[]	0x0



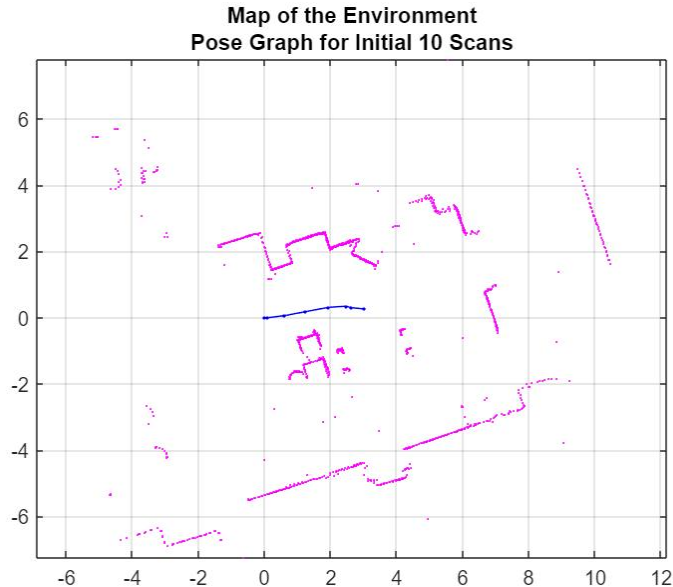
The image shows the MATLAB variable viewer for the 'loopClosureInfo' variable. It is a 1x1 struct with 4 fields: IsPerformed, IsAccepted, ResidualError, and LoopClosuresRemoved. IsPerformed and IsAccepted have values of 0 and size 1x1. ResidualError and LoopClosuresRemoved have empty array values '[]' and size 0x0.

Field	Value	Size
IsPerformed	0	1x1
IsAccepted	0	1x1
ResidualError	[]	0x0
LoopClosuresRemoved	[]	0x0

Observe the Map Building Process with Initial 10 Scans

Reconstruct the scene by plotting the scans and poses tracked by the slamAlg.

```
figure;  
show(slamAlg);  
title({'Map of the Environment','Pose Graph for Initial 10 Scans'});
```



Observe the Effect of Loop Closures and the Optimization Process

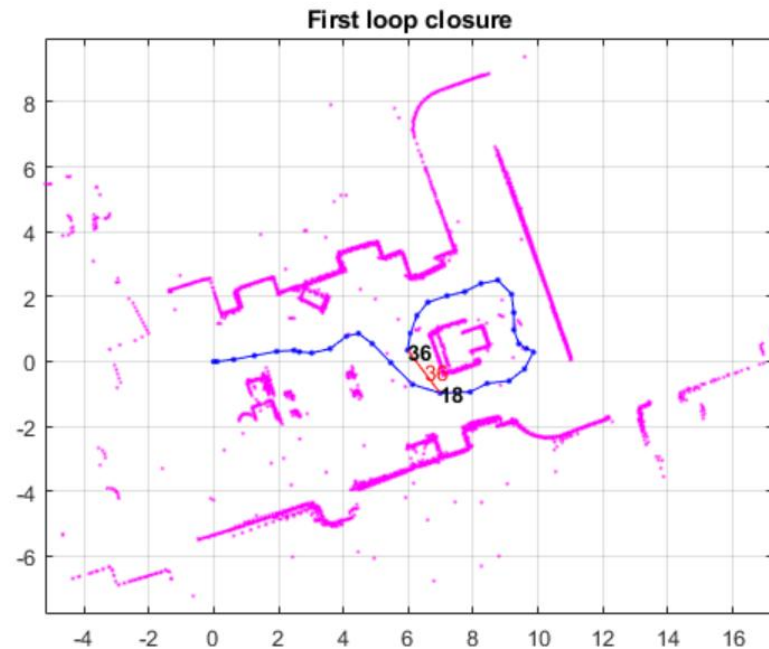
Continue to add scans in a loop. Loop closures should be automatically detected as the robot moves. Pose graph optimization is performed whenever a loop closure is identified. The output `optimizationInfo` has a field, `IsPerformed`, that indicates when pose graph optimization occurs.

Plot the scans and poses whenever a loop closure is identified and verify the results visually. This plot shows overlaid scans and an optimized pose graph for the first loop closure. A loop closure edge is added as a red link.

Observe the Effect of Loop Closures and the Optimization Process

```
firstTimeLCDetected = false;

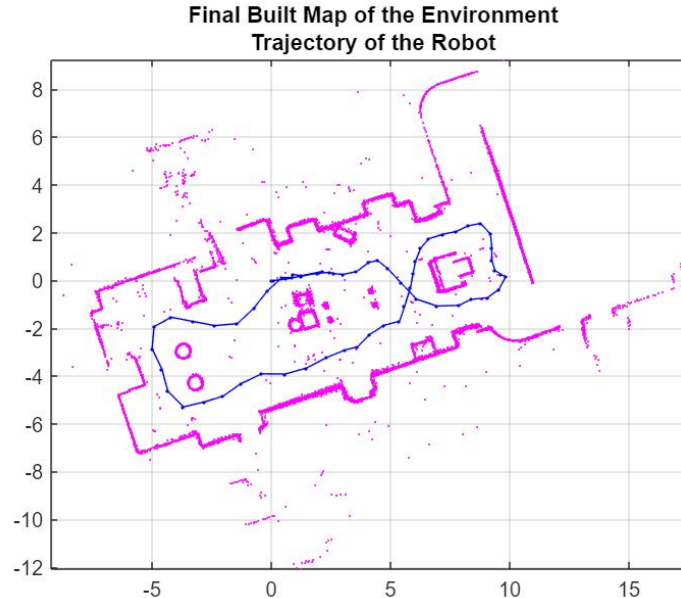
figure;
for i=10:length(scans)
    [isScanAccepted, loopClosureInfo, optimizationInfo] = addScan(slamAlg, scans{i});
    if ~isScanAccepted
        continue;
    end
    % visualize the first detected loop closure, if you want to see the
    % complete map building process, remove the if condition below
    if optimizationInfo.IsPerformed && ~firstTimeLCDetected
        show(slamAlg, 'Poses', 'off');
        hold on;
        show(slamAlg.PoseGraph);
        hold off;
        firstTimeLCDetected = true;
        drawnow
    end
end
title('First loop closure');
```



Visualize the Constructed Map and Trajectory of the Robot

Plot the final built map after all scans are added to the slamAlg object. Though the previous for loop only plotted the initial closure, all the scans were added.

```
figure  
show(slamAlg);  
title({'Final Built Map of the Environment', 'Trajectory of the Robot'});
```



Visually Inspect the Built Map Compared to the Original Floor Plan

An image of the scans and pose graph is overlaid on the original floorplan. You can see that the map matches the original floor plan well after adding all the scans and optimizing the pose graph.



Build Occupancy Grid Map

The optimized scans and poses can be used to generate a occupancyMap, which represents the environment as a probabilistic occupancy grid.

```
#Extract scans and corresponding poses
```

```
[scans, optimizedPoses] = scansAndPoses(slamAlg);
```

```
map = buildMap(scans, optimizedPoses, mapResolution, maxLidarRange);
```

Visualize the occupancy grid map populated with the laser scans and the optimized pose graph.

```
figure;
```

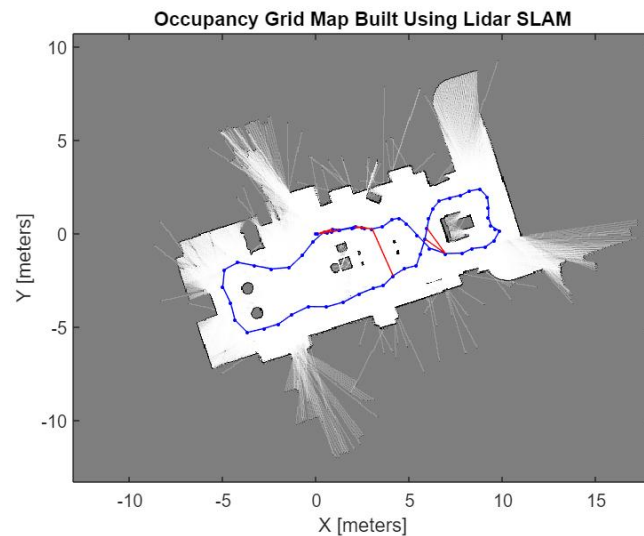
```
show(map);
```

```
hold on
```

```
show(slamAlg.PoseGraph, 'IDs', 'off');
```

```
hold off
```

```
title('Occupancy Grid Map Built Using Lidar SLAM');
```



SLAM Using 3-D Lidar Point Clouds

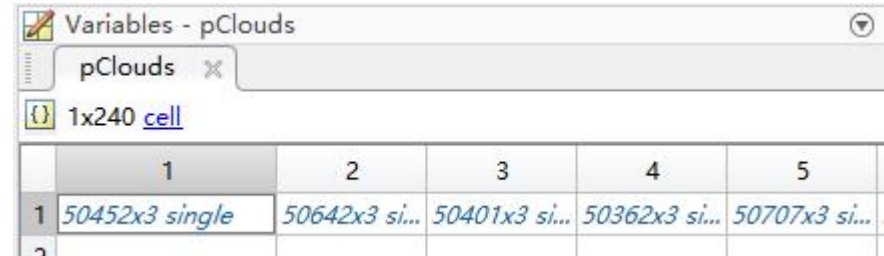
This example demonstrates how to implement the simultaneous localization and mapping (SLAM) algorithm on collected 3-D lidar sensor data using point cloud processing algorithms and pose graph optimization. The goal of this example is to estimate the trajectory of the robot and create a 3-D occupancy map of the environment from the 3-D lidar point clouds and estimated trajectory.

The demonstrated SLAM algorithm estimates a trajectory using a Normal Distribution Transform (NDT) based point cloud registration algorithm and reduces the drift using SE3 pose graph optimization using trust-region solver whenever a robot revisits a place.

Load Data And Set Up Tunable Parameters

Load the 3-D lidar data collected from a Clearpath™ Husky robot in a parking garage. The lidar data contains a cell array of n-by-3 matrices, where n is the number 3-D points in the captured lidar data, and the columns represent xyz-coordinates associated with each captured point.

load pClouds.mat



The image shows a screenshot of the MATLAB 'Variables' window. The window title is 'Variables - pClouds'. Inside, there is a tab for 'pClouds' with a close button. Below the tab, it displays '1x240 cell'. A scrollable table below shows the first five elements of the cell array. Each element is a matrix of size n-by-3, where n is the number of 3-D points in that specific lidar capture. The first row of the table shows the dimensions for the first five elements: 50452x3, 50642x3, 50401x3, 50362x3, and 50707x3. The second row shows the first few elements of each matrix, which are single values.

	1	2	3	4	5
1	50452x3 single	50642x3 si...	50401x3 si...	50362x3 si...	50707x3 si...
2					

Parameters For Point Cloud Registration Algorithm

Specify the parameters for estimating the trajectory using the point cloud registration algorithm. `maxLidarRange` specifies the maximum range of the 3-D laser scanner.

```
maxLidarRange = 20;
```

The point cloud data captured in an indoor environment contains points lying on the ground and ceiling planes, which confuses the point cloud registration algorithms.

Some points are removed from the point cloud with these parameters:

- `referenceVector` - Normal to the ground plane.
- `maxDistance` - Maximum distance for inliers when removing the ground and ceiling planes.
- `maxAngularDistance` - Maximum angle deviation from the reference vector when fitting the ground and ceiling planes.

Parameters For Point Cloud Registration Algorithm

```
referenceVector = [0 0 1];
```

```
maxDistance = 0.5;
```

```
maxAngularDistance = 15;
```

To improve the efficiency and accuracy of the registration algorithm, the point clouds are downsampled using random sampling with a sample ratio specified by randomSampleRatio.

```
randomSampleRatio = 0.25;
```

gridStep specifies the voxel grid sizes used in the NDT registration algorithm. A scan is accepted only after the robot moves by a distance greater than distanceMovedThreshold.

```
gridStep = 2.5;
```

```
distanceMovedThreshold = 0.3;
```

Parameters For Loop Closure Estimation Algorithm

Specify the parameters for the loop closure estimation algorithm. Loop closures are only searched within a radius around the current robot location specified by

`loopClosureSearchRadius`.

`loopClosureSearchRadius = 3;`

The loop closure algorithm is based on 2-D submap and scan matching. A submap is created after every `nScansPerSubmap` (Number of Scans per submap) accepted scans. The loop closure algorithm disregards the most recent `subMapThresh` scans while searching for loop candidates.

`nScansPerSubmap = 3;`

`subMapThresh = 50;`

An annular region with z-limits specified by `annularRegionLimits` is extracted from the point clouds. Points outside these limits on the floor and ceiling are removed after the point cloud plane fit algorithms identify the region of interest.

`annularRegionLimits = [-0.75 0.75];`

Parameters For Loop Closure Estimation Algorithm

The maximum acceptable Root Mean Squared Error (RMSE) in estimating relative pose between loop candidates is specified by `rmseThreshold`. Choose a lower value for estimating accurate loop closure edges, which has a high impact on pose graph optimization.

`rmseThreshold = 0.26;`

The threshold over scan matching score to accept a loop closure is specified by `loopClosureThreshold`. Pose Graph Optimization is called after inserting `optimizationInterval` loop closure edges into the pose graph.

`loopClosureThreshold = 150;`

`optimizationInterval = 2;`

Initialize Variables

Set up a pose graph, occupancy map, and necessary variables.

% 3D Posegraph object for storing estimated relative poses

```
pGraph = poseGraph3D;
```

% Default serialized upper-right triangle of 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];
```

% Number of loop closure edges added since last pose graph optimization and map refinement

```
numLoopClosuresSinceLastOptimization = 0;
```

% True after pose graph optimization until the next scan

```
mapUpdated = false;
```

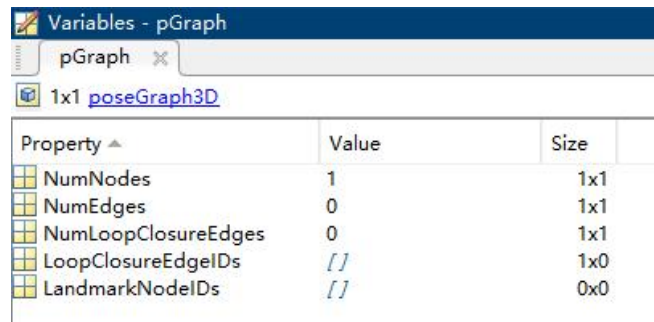
% Equals to 1 if the scan is accepted

```
scanAccepted = 0;
```

% 3D Occupancy grid object for creating and visualizing 3D map

```
mapResolution = 8; % cells per meter
```

```
omap = occupancyMap3D(mapResolution);
```

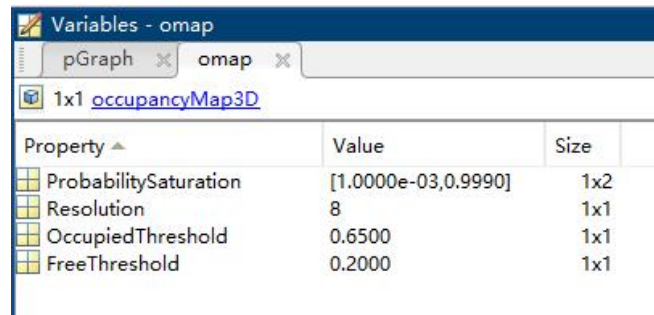


Variables - pGraph

pGraph

1x1 [poseGraph3D](#)

Property ^	Value	Size
NumNodes	1	1x1
NumEdges	0	1x1
NumLoopClosureEdges	0	1x1
LoopClosureEdgeIDs	[]	1x0
LandmarkNodeIDs	[]	0x0



Variables - omap

pGraph omap

1x1 [occupancyMap3D](#)

Property ^	Value	Size
ProbabilitySaturation	[1.0000e-03,0.9990]	1x2
Resolution	8	1x1
OccupiedThreshold	0.6500	1x1
FreeThreshold	0.2000	1x1

Initialize Variables

Preallocate variables for the processed point clouds, lidar scans, and submaps. Create a downsampled set of point clouds for quickly visualizing the map.

```
pcProcessed = cell(1,length(pClouds));  
lidarScans2d = cell(1,length(pClouds));  
submaps = cell(1,length(pClouds)/nScansPerSubmap);
```

```
pcsToView = cell(1,length(pClouds));  
Create variables for display purposes.  
% Set to 1 to visualize created map and posegraph during build process  
viewMap = 1;  
% Set to 1 to visualize processed point clouds during build process  
viewPC = 0;
```

Name	Value
 pClouds	1x240 cell
 lidarScans2d	1x240 cell
 pcProcessed	1x240 cell
 pcsToView	1x240 cell
 submaps	1x80 cell

Initialize Variables

Set random seed to guarantee consistent random sampling.

```
rng(0);
```

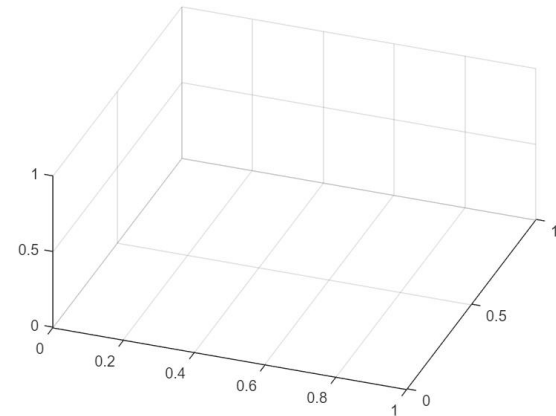
Initialize figure windows if desired.

```
% If you want to view the point clouds while processing them sequentially  
if viewPC==1
```

```
    pplayer = pcplayer([-50 50],[-50 50],[-10 10],'MarkerSize',10);  
end
```

```
% If you want to view the created map and posegraph during build process  
if viewMap==1
```

```
    ax = newplot; % Figure axis handle  
    view(20,50);  
    grid on;  
end
```



Trajectory Estimation And Refinement Using Pose Graph Optimization

The trajectory of the robot is a collection of robot poses (location and orientation in 3-D space). A robot pose is estimated at every 3-D lidar scan acquisition instance using the 3-D lidar SLAM algorithm. The 3-D lidar SLAM algorithm has the following steps:

- Point cloud filtering
- Point cloud downsampling
- Point cloud registration
- Loop closure query
- Pose graph optimization

Trajectory Estimation And Refinement Using Pose Graph Optimization

Iteratively process the point clouds to estimate the trajectory.

```
count = 0; % Counter to track number of scans added
disp('Estimating robot trajectory...');
for i=1:length(pClouds)
    % Read point clouds in sequence
    pc = pClouds{i};
```

Point Cloud Filtering

Point cloud filtering is done to extract the region of interest from the acquired scan. In this example, the region of interest is the annular region with ground and ceiling removed. Remove invalid points outside the max range and unnecessary points behind the robot corresponding to the human driver.

```
ind = (-maxLidarRange < pc(:,1) & pc(:,1) < maxLidarRange ...  
      & -maxLidarRange < pc(:,2) & pc(:,2) < maxLidarRange ...  
      & (abs(pc(:,2))>abs(0.5*pc(:,1)) | pc(:,1)>0));
```

```
K>> size(pc(ind,:))  
ans =  
      38248      3  
  
K>> size(pc)  
ans =  
      50452      3
```

```
pcl = pointCloud(pc(ind,:));
```



A screenshot of the MATLAB variable inspector window showing the properties of a 1x1 pointCloud object. The table lists properties such as Location, Count, XLimits, YLimits, ZLimits, Color, Normal, and Intensity, along with their values and sizes.

Property	Value	Size
Location	38248x3 single	38248x3
Count	38248	1x1
XLimits	[-19.9787,19.9911]	1x2
YLimits	[-19.1078,19.9648]	1x2
ZLimits	[-1.3908,6.9565]	1x2
Color	[]	0x0
Normal	[]	0x0
Intensity	[]	0x0

Remove points on the ground plane.

```
[~, ~, outliers] = ...  
    pcfitplane(pcl, maxDistance,referenceVector,maxAngularDistance);  
pcl_wogrd = select(pcl,outliers,'OutputSize','full');
```

Point Cloud Filtering

Remove points on the ceiling plane.

```
[~, ~, outliers] = ...  
    pcfitplane(pcl_wogrd, 0.2, referenceVector, maxAngularDistance);  
pcl_wogrd = select(pcl_wogrd, outliers, 'OutputSize', 'full');
```

Select points in annular region.

```
ind =  
(pcl_wogrd.Location(:, 3) < annularRegionLimits(2)) & (pcl_wogrd.Location(:, 3) > annularRegionLimits(1));  
pcl_wogrd = select(pcl_wogrd, ind, 'OutputSize', 'full');
```

Point Cloud Downsampling

Point cloud downsampling improves the speed and accuracy of the point cloud registration algorithm. Down sampling should be tuned for specific needs. The random sampling algorithm is chosen empirically from down sampling variants below for the current scenario.

```
pcl_wogrd_sampled = pcdownsampling(pcl_wogrd,'random',randomSampleRatio);
```

```
if viewPC==1
    % Visualize down sampled point cloud
    view(pplayer,pcl_wogrd_sampled);
    pause(0.001)
end
```



1x1 pointCloud			
Property	Value	Size	
Location	9562x3 single	9562x3	
Count	9562	1x1	
XLimits	[-19.3392,18.4305]	1x2	
YLimits	[-17.3567,18.6787]	1x2	
ZLimits	[-0.5131,0.7395]	1x2	
Color	[]	0x0	
Normal	[]	0x0	
Intensity	[]	0x0	

Loop Closure Query

Loop closure query determines whether or not the current robot location has previously been visited. The search is performed by matching the current scan with the previous scans within a small radius around the current robot location specified by `loopClosureSearchRadius`. Searching within the small radius is sufficient because of the low-drift in lidar odometry, as searching against all previous scans is time consuming. Loop closure query consists of the following steps:

- Create submaps from `nScansPerSubmap` consecutive scans.
- Match the current scan with the submaps within the `loopClosureSearchRadius`.
- Accept the matches if the match score is greater than the `loopClosureThreshold`. All the scans representing accepted submap are considered as probable loop candidates.
- Estimate the relative pose between probable loop candidates and the current scan. A relative pose is accepted as a loop closure constraint only when the RMSE is less than the `rmseThreshold`.

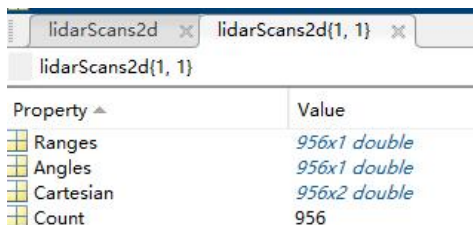
Loop Closure Query

```
if scanAccepted == 1
    count = count + 1;
    pcProcessed{count} = pcl_wogrd_sampled;
    lidarScans2d{count} = exampleHelperCreate2DScan(pcl_wogrd_sampled);

% Submaps are created for faster loop closure query.
if rem(count,nScansPerSubmap)==0
    submaps{count/nScansPerSubmap} = exampleHelperCreateSubmap(lidarScans2d, pGraph,count,nScansPerSubmap,maxLidarRange);
end

% loopSubmapIds contains matching submap ids if any otherwise empty.
if (floor(count/nScansPerSubmap)>subMapThresh)
    [loopSubmapIds,~] = exampleHelperEstimateLoopCandidates(pGraph, count,submaps,lidarScans2d{count},nScansPerSubmap, ...
        loopClosureSearchRadius,loopClosureThreshold,subMapThresh);

    if ~isempty(loopSubmapIds)
        rmseMin = inf;
```



The image shows a MATLAB variable viewer window for the variable `lidarScans2d{1,1}`. The window has a title bar with the variable name and a close button. Below the title bar, the variable name is repeated. The main area of the window is divided into two columns: 'Property' and 'Value'. There are four rows of properties: 'Ranges', 'Angles', 'Cartesian', and 'Count'. Each row has a small icon to the left of the property name. The values for these properties are: 'Ranges' is '956x1 double', 'Angles' is '956x1 double', 'Cartesian' is '956x2 double', and 'Count' is '956'.

Property ▲	Value
Ranges	956x1 double
Angles	956x1 double
Cartesian	956x2 double
Count	956

```

if ~isempty(loopSubmapIds)
    rmseMin = inf;

    % Estimate best match to the current scan
    for k = 1:length(loopSubmapIds)
        % For every scan within the submap
        for j = 1:nScansPerSubmap
            probableLoopCandidate = ...
                loopSubmapIds(k)*nScansPerSubmap - j + 1;
            [loopTform,~,rmse] = pcregisterndt(pcl_wogrd_sampled, pcProcessed{probableLoopCandidate},gridStep);
            % Update best Loop Closure Candidate
            if rmse < rmseMin
                loopCandidate = probableLoopCandidate;
                rmseMin = rmse;
            end
            if rmseMin < rmseThreshold
                break;
            end
        end
    end
end

% Check if loop candidate is valid
if rmseMin < rmseThreshold
    % loop closure constraint
    relPose = [tform2trvec(loopTform.T') tform2quat(loopTform.T')];

    addRelativePose(pGraph,relPose,infoMat, loopCandidate,count);
    numLoopClosuresSinceLastOptimization = numLoopClosuresSinceLastOptimization + 1;
end
end
end

```

Loop Closure Query

```
function lidarScan2d = exampleHelperCreate2DScan(ptCloud)
```

```
% This helper function is used to create 2D Lidar scans from 3D Lidar  
% Scans. These 2D scans are required for 2D submap creation which are  
% useful for loop closure query. 3D Point Cloud Down sampling is done  
% before creating a 2D scan to reduce the computations.  
% annularSamplingRatio specifies the sample ratio to uniform sample the  
% annular region. The sampling ratio is empirically chosen for this  
% example.
```

```
function [submap] = exampleHelperCreateSubmap(lidarScans,poseGraph,currentId,nScansPerSubmap,maxRange)
```

```
% This helper function is used to create 2D submaps from 2D Lidar Scans which  
% have been created by slicing the point clouds along an annular region.  
% These Submaps are required for detecting loop closures. Each submap  
% represents multiple scans.
```

```
function [loopSubmapIds,loopScores] = exampleHelperEstimateLoopCandidates(pGraph,currentScanId,submaps,currScan,...  
    nScansPerSubmap,loopClosureSearchRadius,loopClosureThreshold,subMapThresh)
```

```
% This helper function to returns submap ids which lie within a radius from  
% current scan and match with the current scan. Instead of matching the  
% current scan with all the previously accepted scans for faster query  
% current scan is matched against a submap (group of scans). Due to this  
% the number of matching operation reduces significantly. The submaps are  
% said to be matching with the current scan when the submap and scan match  
% score is greater than loopClosureThreshold. Most recent subMapThresh  
% submaps are not considered while estimating a loop sub map.
```

NDT算法

NDT全称为Normal Distributions Transform，正态分布转换，属于用非线性优化方法解决Slam中帧间匹配的一种算法。算法主要思想是为前一帧激光点划分栅格，并假设每一块栅格的激光点分布符合正态分布，把当前帧激光点数据通过转换矩阵转换到前一帧上来，求和计算转换之后的激光点的概率之和，可以用最优化方法最大化这个概率之和，从而解得转换矩阵，达到帧间匹配的作用。

主要步骤

- 1.计算NDT过程
- 2.帧间匹配
- 3.牛顿算法迭代收敛

NDT算法

计算NDT过程

对于每一个小的栅格， 需要执行下面的三步：

1.在每一个栅格内， 有激光点云集合 $x_{i=1,..n}$ ；

2.计算激光点云集合的均值 $q = \frac{1}{n} \sum x_i$ ；

3.计算激光点云集合的协方差 $\Sigma = \frac{1}{n} \sum (x_i - q) (x_i - q)^t$ ；

这里的协方差主要用于描述激光点的分散程度。

那么对于得到的一个激光点在此栅格中的概率可以用如下公式计算：

$$p(x) \sim \exp\left(-\frac{(x - q)^t \Sigma^{-1} (x - q)}{2}\right)$$

NDT算法

帧间匹配过程

$$T : \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

上公式中 $\begin{pmatrix} x \\ y \end{pmatrix}$ 是当前帧激光点云数据, $\begin{pmatrix} x' \\ y' \end{pmatrix}$ 是上一帧激光点云数据, T 是两帧激光点云的变换矩阵, 计算 T 的过程如下:

1. 建立第一帧(前一帧)激光点云的NDT(概率值);
2. 初始化两帧(前一帧和当前帧)之间的位姿, 对于移动机器人, 可以用里程计来初始化;
3. 根据初始化的位姿, 完成第二帧(当前)激光的坐标转换, 即根据把当前帧的激光点, 通过转换矩阵, 转到前一阵激光的坐标下;
4. 计算第二帧激光经过坐标转换后的概率分布情况, 并计算总的分数;
5. 换一个位姿, 按照3, 4步骤计算匹配分数, 直到达到收敛准则, 这一步用到了牛顿方法;

Pose Graph Optimization

Pose graph optimization runs after a sufficient number of loop edges are accepted to reduce the drift in trajectory estimation. After every loop closure optimization the loop closure search radius is reduced due to the fact that the uncertainty in the pose estimation reduces after optimization.


```

if (numLoopClosuresSinceLastOptimization == optimizationInterval)||...
    ((numLoopClosuresSinceLastOptimization>0)&&(i==length(pClouds)))
    if loopClosureSearchRadius ~1
        disp('Doing Pose Graph Optimization to reduce drift.');
```

end

```

% pose graph optimization
pGraph = optimizePoseGraph(pGraph);
loopClosureSearchRadius = 1;
if viewMap == 1
    position = pGraph.nodes;
    % Rebuild map after pose graph optimization
    omap = occupancyMap3D(mapResolution);
    for n = 1:(pGraph.NumNodes-1)
        insertPointCloud(omap,position(n,:),pcsToView{n}.removeInvalidPoints,maxLidarRange);
    end
    mapUpdated = true;
    ax = newplot;
    grid on;
end
numLoopClosuresSinceLastOptimization = 0;
% Reduce the frequency of optimization after optimizing
% the trajectory
optimizationInterval = optimizationInterval*7;
end

```

Pose Graph Optimization

Visualize the map and pose graph during the build process. This visualization is costly, so enable it only when necessary by setting viewMap to 1. If visualization is enabled then the plot is updated after every 15 added scans.

```
pcToView = pcdownsampling(pcl_wogrd_sampled, 'random', 0.5);
pcsToView{count} = pcToView;

if viewMap==1
    % Insert point cloud to the occupancy map in the right position
    position = pGraph.nodes(count);
    insertPointCloud(omap,position,pcToView.removeInvalidPoints,maxLidarRange);

    if (rem(count-1,15)==0) || mapUpdated
        exampleHelperVisualizeMapAndPoseGraph(omap, pGraph, ax);
    end
    mapUpdated = false;
else
    % Give feedback to know that example is running
    if (rem(count-1,15)==0)
        fprintf('.');
    end
end
end
```

Pose Graph Optimization

Update previous relative pose estimate and point cloud.

```
    prevPc = pcl_wogrd_sampled;  
    prevTform = tform;  
end  
end
```

Build and Visualize 3-D Occupancy Map

The point clouds are inserted into occupancyMap3D using the estimated global poses. After iterating through all the nodes, the full map and estimated vehicle trajectory is shown.

```
if (viewMap ~= 1) || (numLoopClosuresSinceLastOptimization > 0)
    nodesPositions = nodes(pGraph);
    % Create 3D Occupancy grid
    omapToView = occupancyMap3D(mapResolution);

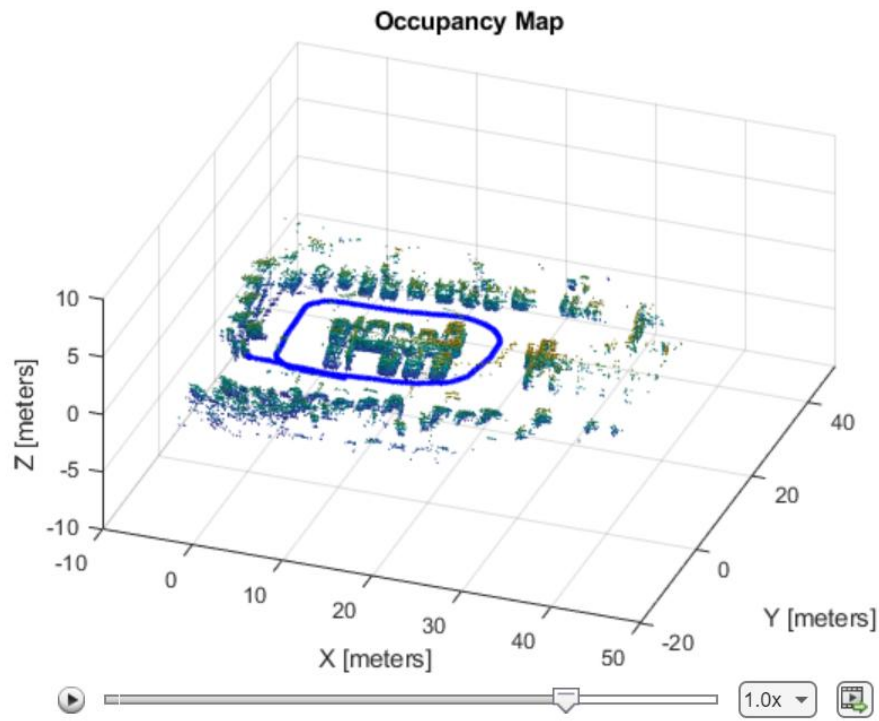
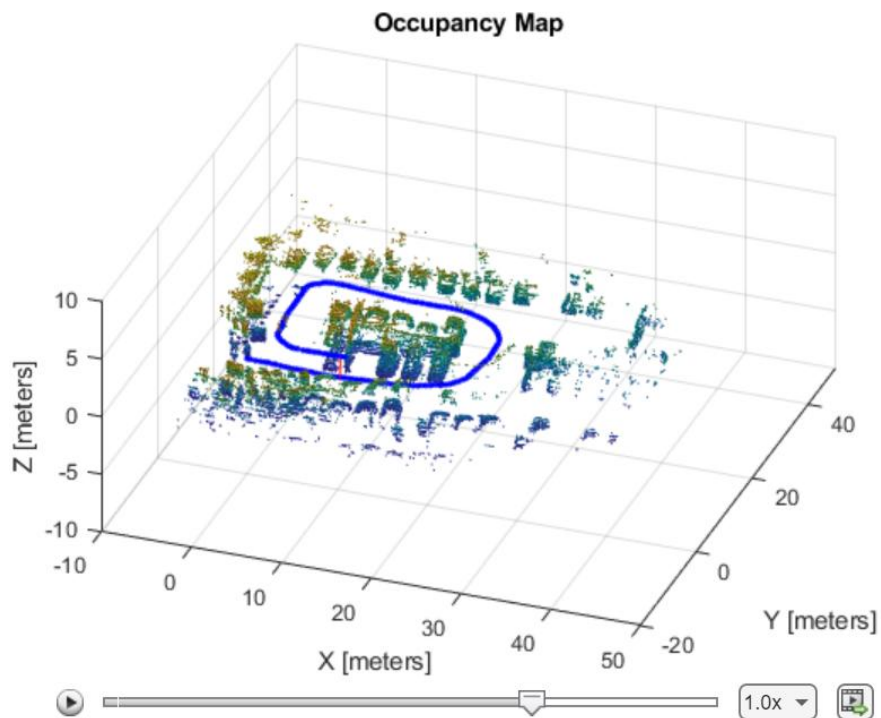
    for i = 1:(size(nodesPositions,1)-1)
        pc = pcsToView{i};
        position = nodesPositions(i,:);

        % Insert point cloud to the occupancy map in the right position
        insertPointCloud(omapToView, position, pc.removeInvalidPoints, maxLidarRange);
    end

    figure;
    axisFinal = newplot;
    exampleHelperVisualizeMapAndPoseGraph(omapToView, pGraph, axisFinal);
end
```

Build and Visualize 3-D Occupancy Map

The point clouds are inserted into occupancyMap3D using the estimated global poses. After iterating through all the nodes, the full map and estimated vehicle trajectory is shown.



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. 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.

Download Lidar Data Set

This example uses a subset of PandaSet [2] that contains 2560 preprocessed organized point clouds. Each point cloud covers 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. The downloaded file contains Lidar, Cuboids and semanticLabels folders that holds the point clouds, cuboid label and semantic label info respectively

```
doTraining = false;
```

```
outputFolder = fullfile(pwd,'Pandaset');
```

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

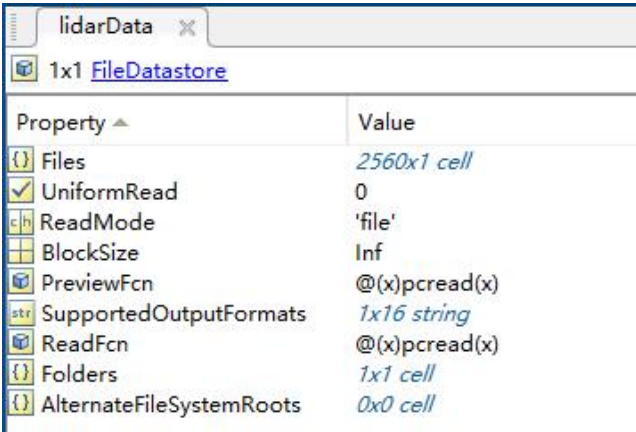

Load Data

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));
```

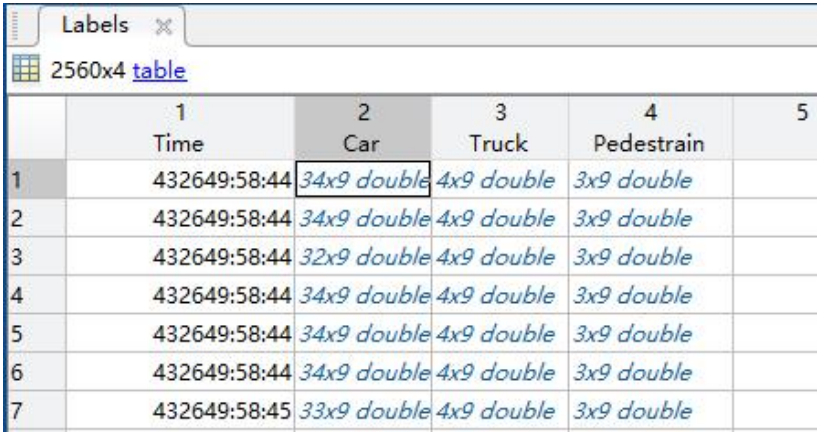
Load the 3-D bounding box labels of the car and truck objects.

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



The screenshot shows the MATLAB Variable Inspector for the variable 'lidarData'. It is a 1x1 FileDatastore. The properties listed are: Files (2560x1 cell), UniformRead (0), ReadMode ('file'), BlockSize (Inf), PreviewFcn (@(x)pcread(x)), SupportedOutputFormats (1x16 string), ReadFcn (@(x)pcread(x)), Folders (1x1 cell), and AlternateFileSystemRoots (0x0 cell).

Property	Value
Files	2560x1 cell
UniformRead	0
ReadMode	'file'
BlockSize	Inf
PreviewFcn	@(x)pcread(x)
SupportedOutputFormats	1x16 string
ReadFcn	@(x)pcread(x)
Folders	1x1 cell
AlternateFileSystemRoots	0x0 cell



The screenshot shows the MATLAB Variable Inspector for the variable 'Labels'. It is a 2560x4 table. The columns are: 1 Time, 2 Car, 3 Truck, 4 Pedestrian, and 5. The data is shown for rows 1 through 7.

	1 Time	2 Car	3 Truck	4 Pedestrian	5
1	432649:58:44	34x9 double	4x9 double	3x9 double	
2	432649:58:44	34x9 double	4x9 double	3x9 double	
3	432649:58:44	32x9 double	4x9 double	3x9 double	
4	432649:58:44	34x9 double	4x9 double	3x9 double	
5	432649:58:44	34x9 double	4x9 double	3x9 double	
6	432649:58:44	34x9 double	4x9 double	3x9 double	
7	432649:58:45	33x9 double	4x9 double	3x9 double	

Load Data

Display the full-view point cloud.

figure

```
ptCld = read(lidarData);
```

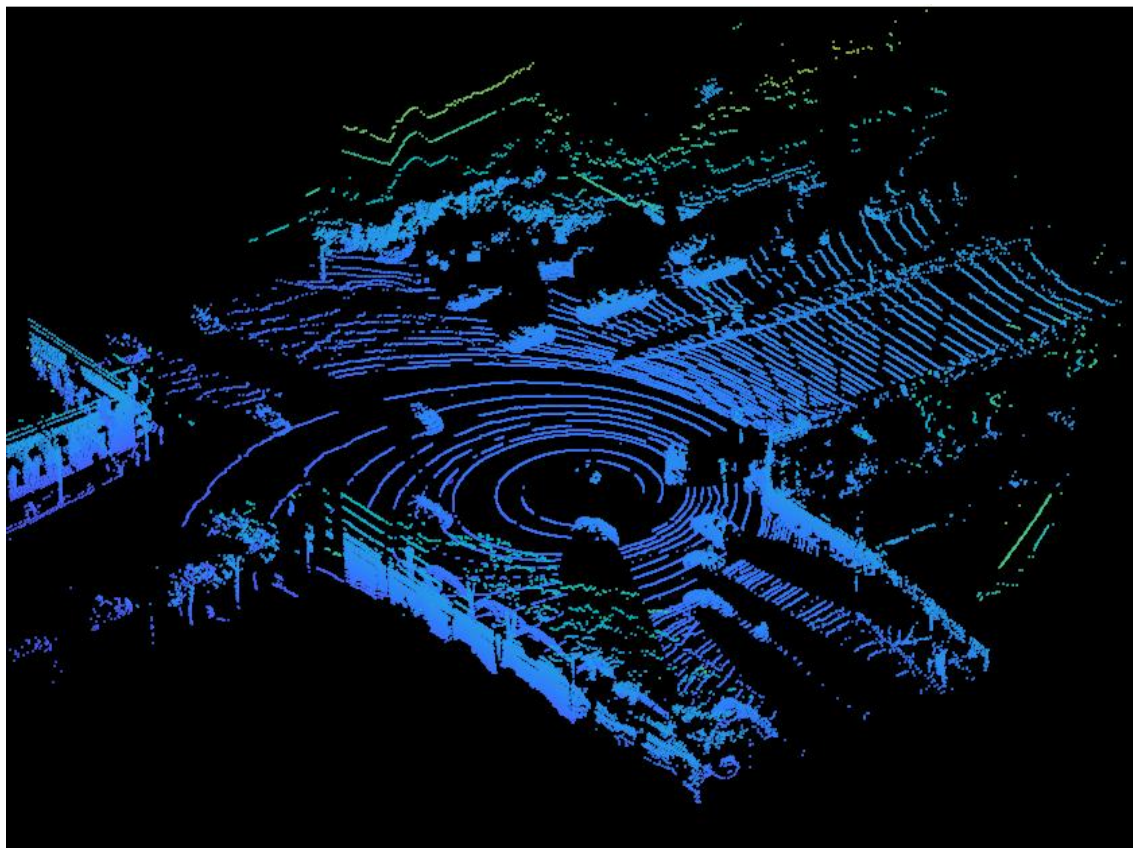
```
ax = pcshow(ptCld.Location);
```

```
set(ax,'XLim',[-50 50],'YLim',[-40 40]);
```

```
zoom(ax,2.5);
```

```
axis off;
```

ptCld	
1x1 pointCloud	
Property	Value
Location	64x1856x3 double
Count	118784
XLimits	[-165.4897,117.0436]
YLimits	[-182.2586,196.7349]
ZLimits	[-7.7812,17.9046]
Color	[]
Normal	[]
Intensity	64x1856 double



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. These parameters determine the size of the input passed to the network. Select a smaller point cloud range along the x, y, and z-axis to detect objects closer to origin.

```
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 point cloud parameters.
```

```
pointCloudRange = [xMin xMax yMin yMax zMin zMax];
```

```
voxelSize = [xStep yStep];
```

Preprocess Data

Use the `cropFrontViewFromLidarData` helper function 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,pointCloudRange);
```

Preprocess Data

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};
```

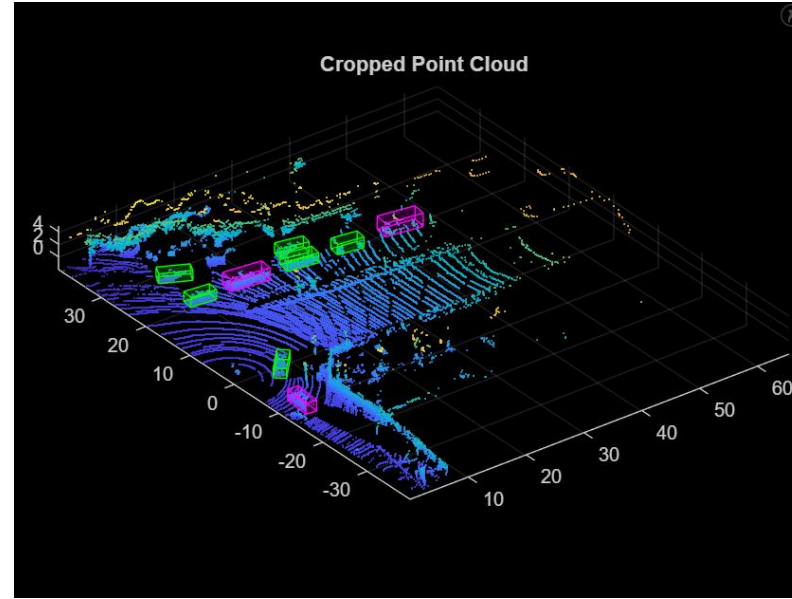
```
gtLabelsCar = processedLabels.Car{1};
```

```
gtLabelsTruck = processedLabels.Truck{1};
```

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

	gtLabelsCar
	gtLabelsTruck

6x9 double
3x9 double



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),:);
```

trainData			
1792x1 cell			
	1	2	
1	1x1 pointCloud		
2	1x1 pointCloud		
3	1x1 pointCloud		
4	1x1 pointCloud		
5	1x1 pointCloud		
6	1x1 pointCloud		
7	1x1 pointCloud		
8	1x1 pointCloud		
9	1x1 pointCloud		
10	1x1 pointCloud		
11	1x1 pointCloud		
12	1x1 pointCloud		
13	1x1 pointCloud		

trainLabels			
1792x2 table			
	1 Car	2 Truck	3
1	6x9 double	3x9 double	
2	8x9 double	[2.9820,23...	
3	10x9 double	0	
4	12x9 double	0	
5	18x9 double	0	
6	13x9 double	2x9 double	
7	9x9 double	0	
8	12x9 double	[6.0352,-3...	
9	11x9 double	0	
10	12x9 double	[6.9823,-2...	
11	6x9 double	0	
12	12x9 double	0	

Create Datastore Objects for Training

So that you can easily access the datastores, save the training data as PCD files by using the `saveptCldToPCD` helper function. 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);
```

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

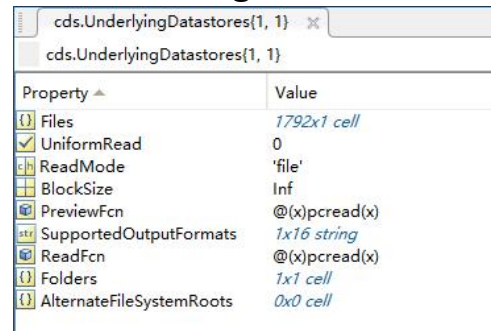
```
lds = fileDatastore(dataLocation,'ReadFcn',@(x) pcread(x));
```

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);
```



cds.UnderlyingDatastores{1, 1}	
cds.UnderlyingDatastores{1, 1}	
Property	Value
Files	1792x1 cell
UniformRead	0
ReadMode	'file'
BlockSize	Inf
PreviewFcn	@(x)pcread(x)
SupportedOutputFormats	1x16 string
ReadFcn	@(x)pcread(x)
Folders	1x1 cell
AlternateFileSystemRoots	0x0 cell

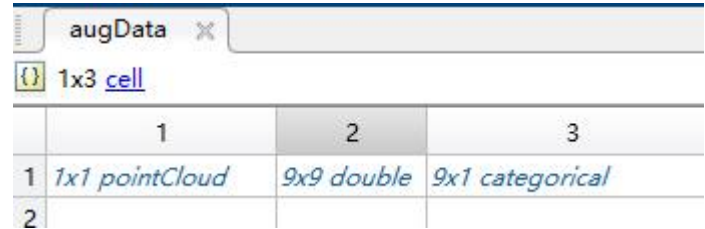
Data Augmentation

This example uses ground truth data augmentation and several other global data augmentation techniques to add more variety to the training data and corresponding boxes. Read and display a point cloud before augmentation using the `helperDisplay3DBoxesOverlaidPointCloud` helper function.

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

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

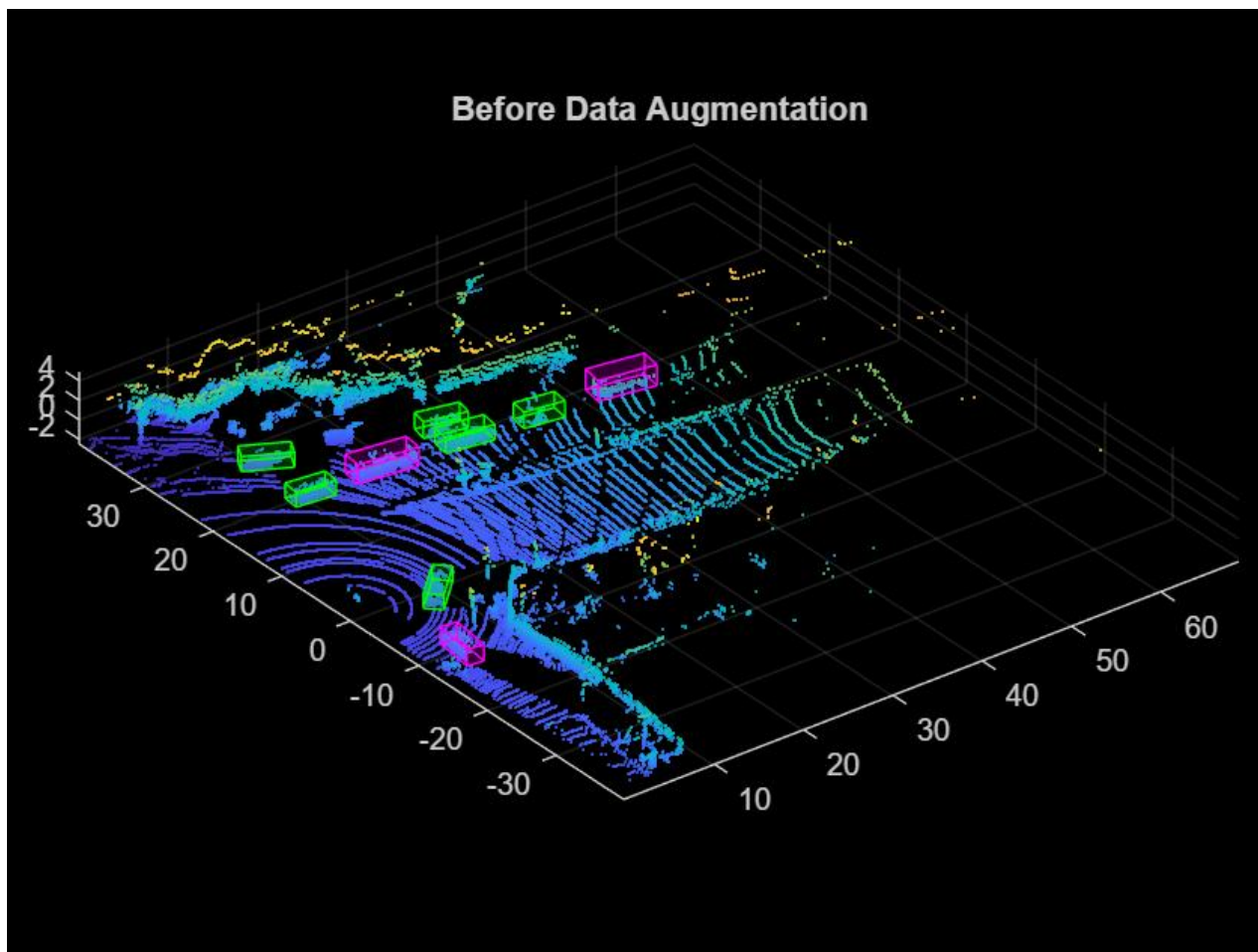
```
helperDisplay3DBoxesOverlaidPointCloud(augptCld.Location,labelsCar,'green',...  
    labelsTruck,'magenta','Before Data Augmentation');
```



The image shows a MATLAB variable viewer window titled 'augData'. It displays a 1x3 cell array. The first cell contains a 1x1 pointCloud, the second cell contains a 9x9 double, and the third cell contains a 9x1 categorical.

	1	2	3
1	1x1 pointCloud	9x9 double	9x1 categorical
2			

Data Augmentation



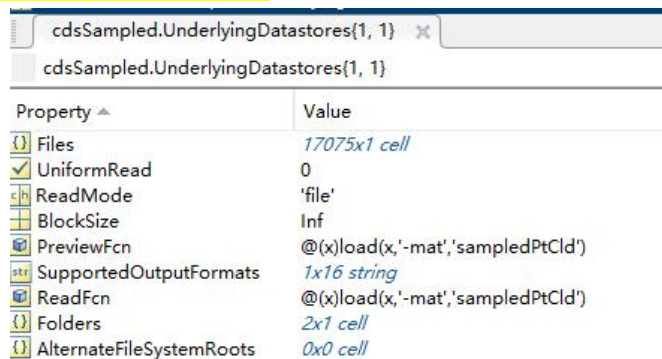
Data Augmentation

Use the `sampleLidarData` function to sample 3-D bounding boxes and their corresponding points from the training data.

```
classNames = {'Car','Truck'};  
sampleLocation = fullfile(outputFolder,'GTsamples');  
[IdsSampled,bdsSampled] = sampleLidarData(cds,classNames,'MinPoints',20,...  
    'Verbose',false,'WriteLocation',sampleLocation);  
cdsSampled = combine(IdsSampled,bdsSampled);
```

Use the `pcBboxOversample` function 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.

```
numObjects = [10 10];  
cdsAugmented = transform(cds,@(x)pcBboxOversample(x,cdsSampled,classNames,numObjects));
```



The image shows a screenshot of the MATLAB Variable Editor. The variable being viewed is `cdsSampled.UnderlyingDatastores{1, 1}`. The editor displays a table of properties and their values for this variable.

Property	Value
Files	17075x1 cell
UniformRead	0
ReadMode	'file'
BlockSize	Inf
PreviewFcn	@(x)load(x,'-mat','sampledPtCld')
SupportedOutputFormats	1x16 string
ReadFcn	@(x)load(x,'-mat','sampledPtCld')
Folders	2x1 cell
AlternateFileSystemRoots	0x0 cell

Data Augmentation

Apply these additional data augmentation techniques 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 the ground truth augmented boxes using the `helperDisplay3DBoxesOverlaidPointCloud` helper function, defined at the end of the example.

```
augData = read(cdsAugmented);
```

```
augptCld = augData{1,1};
```

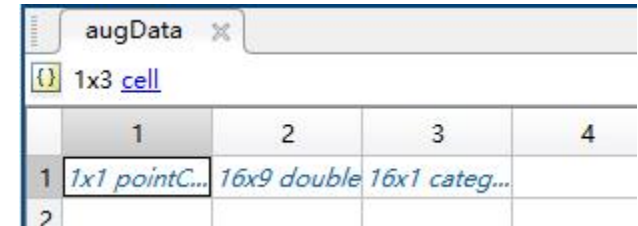
```
augLabels = augData{1,2};
```

```
augClass = augData{1,3};
```

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

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

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



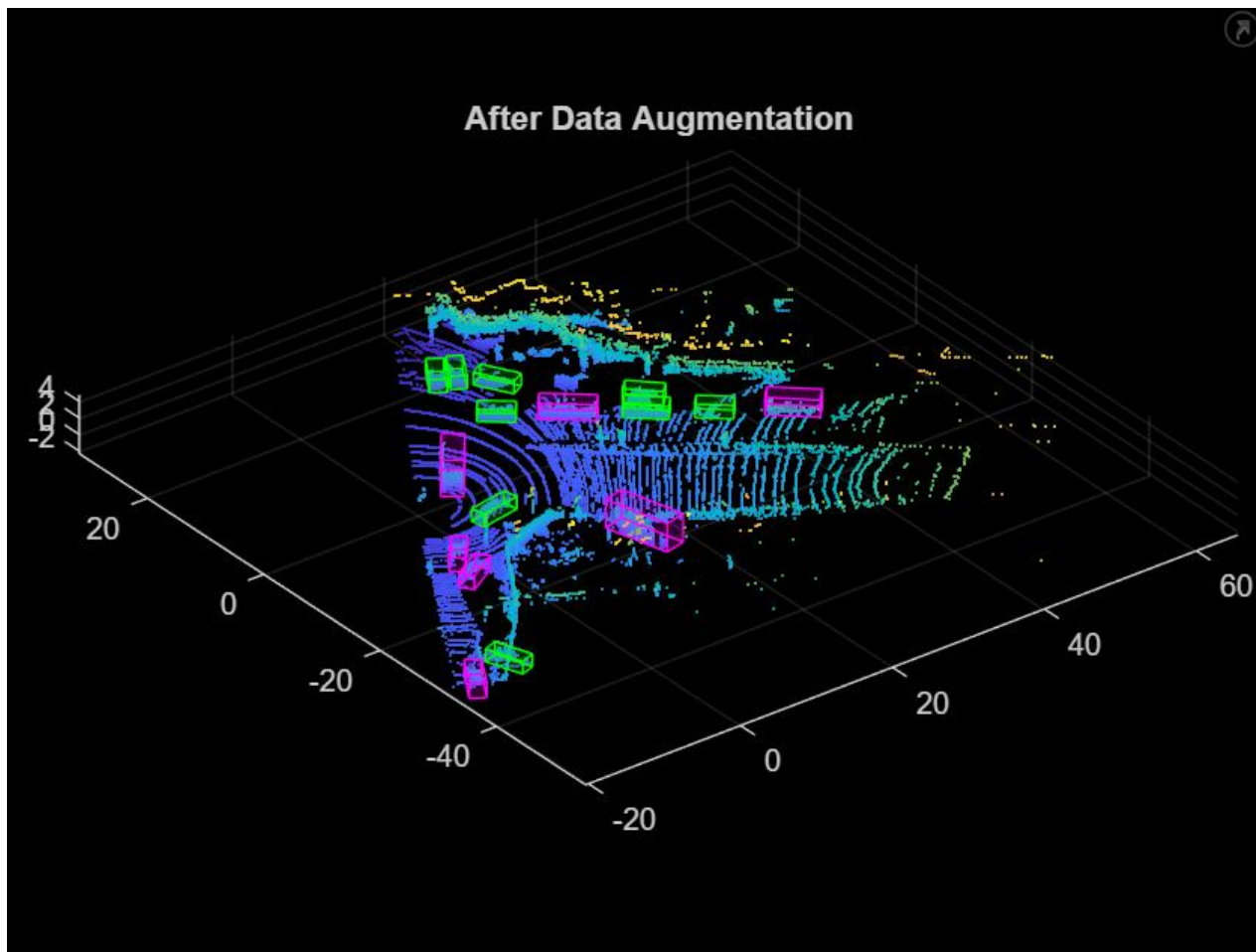
The image shows a MATLAB variable viewer window titled 'augData'. It displays a 1x3 cell array. The first cell contains a 1x1 point cloud, the second cell contains a 16x9 double array, and the third cell contains a 16x1 categorical array. The table below represents the structure of the data.

	1	2	3	4
1	1x1 pointC...	16x9 double	16x1 categ...	
2				

Data Augmentation

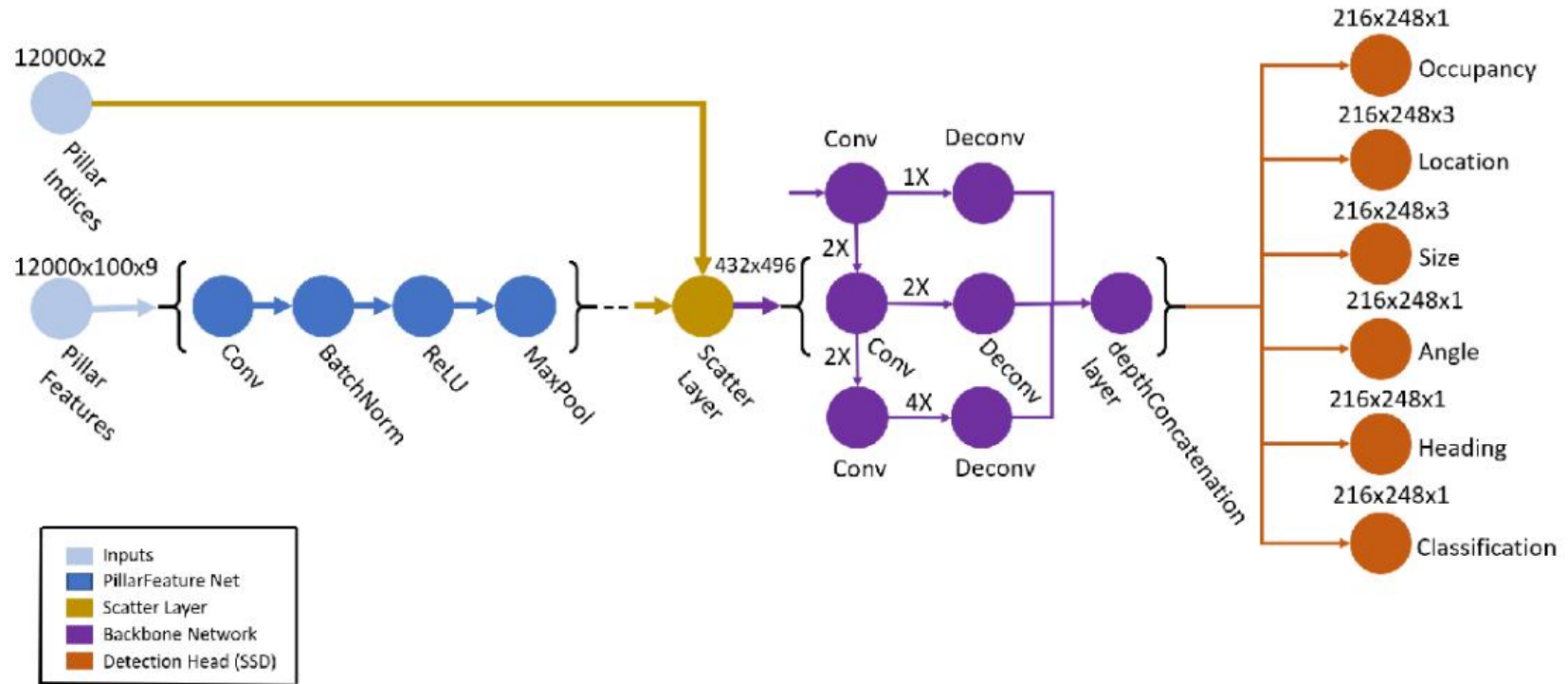
```
function helperDisplay3DBBoxesOverlaidPointCloud(ptCld,labelsCar,carColor,...
    labelsTruck,truckColor,titleForFigure)
% Display the point cloud with different colored bounding boxes for different
% classes.
    figure;
    ax = pcshow(ptCld);
    showShape('cuboid',labelsCar,'Parent',ax,'Opacity',0.1,...
        '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
```

Data Augmentation



Create PointPillars Object Detector

Use the pointPillarsObjectDetector function to create a PointPillars object detection network. The diagram shows the network architecture of a PointPillars object detector. You can use the Deep Network Designer App to create a PointPillars network.



Create PointPillars Object Detector

The `pointPillarsObjectDetector` function requires you to specify several inputs that parameterize the PointPillars network:

- Class names
- Anchor boxes
- Point cloud range
- Voxel size
- Number of prominent pillars
- Number of points per pillar

Create PointPillars Object Detector

% Define the number of prominent pillars.

```
P = 12000;
```

% Define the number of points per pillar.

```
N = 100;
```

Estimate the anchor boxes from training data using calculateAnchorsPointPillars helper function.

```
anchorBoxes = calculateAnchorsPointPillars(trainLabels);  
classNames = trainLabels.Properties.VariableNames;
```

Define the PointPillars detector.

```
detector = pointPillarsObjectDetector(pointCloudRange,classNames,anchorBoxes,...  
    'VoxelSize',voxelSize,'NumPillars',P,'NumPointsPerPillar',N);
```



ISSUES

ANALYSIS RESULT

	Name	Type	Activations	Learnable Prope...	
1	pillars indices reshape 12000×2×1 images	Image Input	12000(S) × 2(S) × 1(C) × 1(B)	-	
2	pillars input 12000×100×9 images	Image Input	12000(S) × 100(S) × 9(C) × 1(B)	-	
3	pillars conv2d 64 1×1 convolutions with stride [1 1] and...	Convolution	12000(S) × 100(S) × 64(C) × 1(B)	Weg... 1 × 1 × 9 ... Bias 1 × 1 × 64	
4	pillars batchnorm Batch normalization	Batch Normalization	12000(S) × 100(S) × 64(C) × 1(B)	Offset 1 × 1 × 64 Scale 1 × 1 × 64	
5	pillars relu ReLU	ReLU	12000(S) × 100(S) × 64(C) × 1(B)	-	
6	pillars reshape 1×100 max pooling with stride [1 100] a...	Max Pooling	12000(S) × 1(S) × 64(C) × 1(B)	-	
7	pillars scatter_nd Scatter features back to original pillar lo...	Function	-	-	
8	cnn block1 conv2d0 64 3×3 convolutions with stride [2 2] and...	Convolution	-	Weights Bias	
9	cnn block1 conv2d0_relu ReLU	ReLU	-	-	
10	cnn block1 bn0 Batch normalization	Batch Normalization	-	Offset Scale	
11	cnn block1 conv2d1 64 3×3 convolutions with stride [1 1] and...	Convolution	-	Weights Bias	
12	cnn block1 conv2d1_relu ReLU	ReLU	-	-	
13	cnn block1 bn1 Batch normalization	Batch Normalization	-	Offset Scale	
14	cnn block1 conv2d2 64 3×3 convolutions with stride [1 1] and...	Convolution	-	Weights Bias	
15	cnn block1 conv2d2_relu ReLU	ReLU	-	-	
16	cnn block1 bn2 Batch normalization	Batch Normalization	-	Offset Scale	
17	cnn block1 conv2d3 64 3×3 convolutions with stride [1 1] and...	Convolution	-	Weights Bias	

Train Pointpillars Object Detector

Specify the network training parameters using the trainingOptions function.

```
options = trainingOptions('adam',...  
    'Plots','none',...  
    'MaxEpochs',60,...  
    'MiniBatchSize',1,...  
    'GradientDecayFactor',0.9,...  
    'SquaredGradientDecayFactor',0.999,...  
    'LearnRateSchedule','piecewise',...  
    'InitialLearnRate',0.0002,...  
    'LearnRateDropPeriod',15,...  
    'LearnRateDropFactor',0.8,...  
    'ExecutionEnvironment','auto',...  
    'DispatchInBackground',true,...  
    'BatchNormalizationStatistics','moving',...  
    'ResetInputNormalization',false,...  
    'CheckpointPath',tempdir);
```

Train Pointpillars Object Detector

Use the `trainPointPillarsObjectDetector` function to train the PointPillars object detector if `doTraining` is "true". Otherwise, load a pretrained detector.

```
if doTraining
    [detector,info] = trainPointPillarsObjectDetector(cdsAugmented,detector,options);
else
    pretrainedDetector = load('pretrainedPointPillarsDetector.mat','detector');
    detector = pretrainedDetector.detector;
end
```

Generate Detections

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

Read the point cloud from the test data.

Run the detector on the test point cloud to get the predicted bounding boxes and confidence scores. Display the point cloud with bounding boxes using the `helperDisplay3DBoxesOverlaidPointCloud` function.

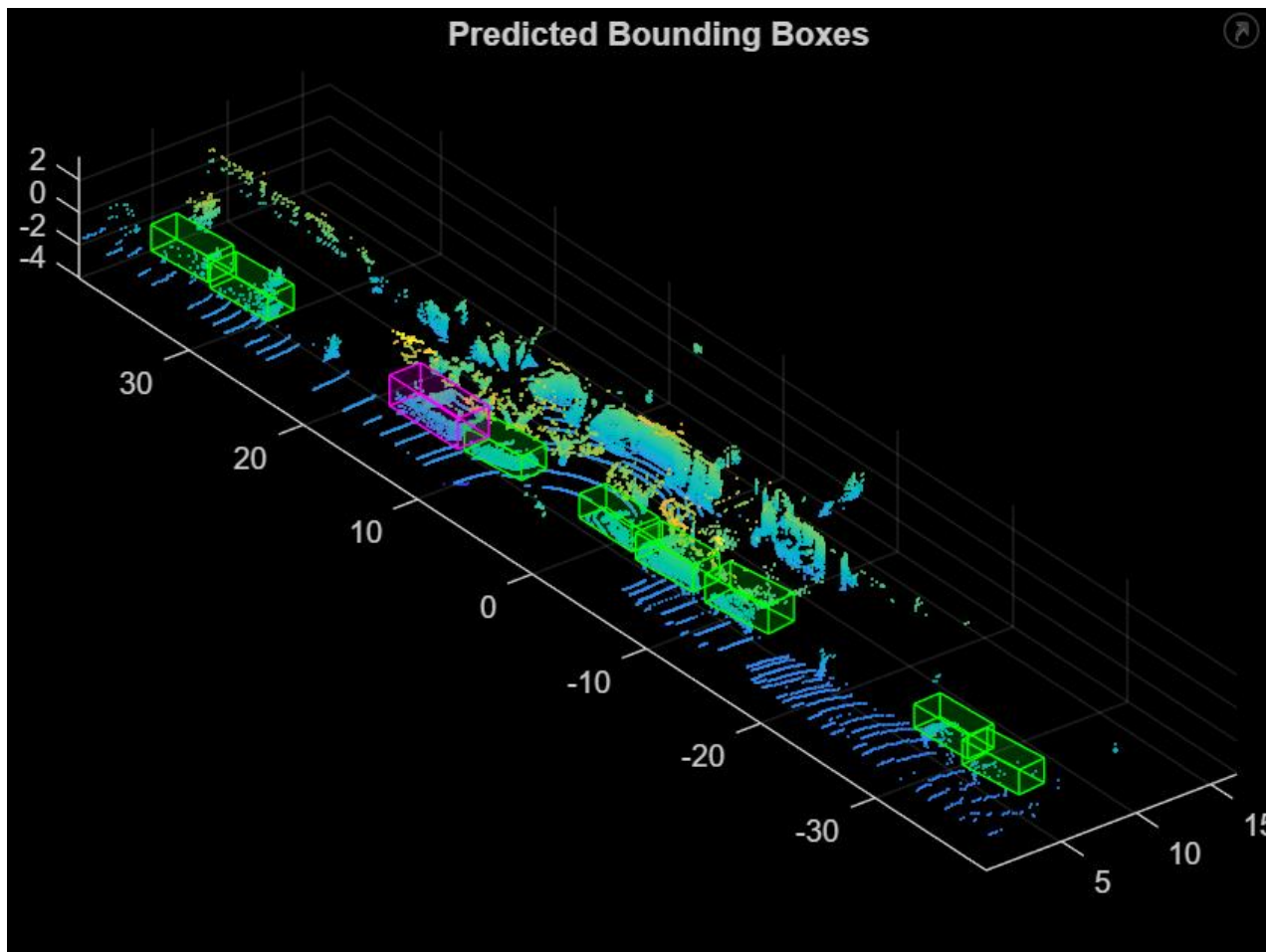
```
ptCloud = testData{45,1};  
gtLabels = testLabels(45,:);
```

```
% Specify the confidence threshold to use only detections with confidence scores above this value.  
confidenceThreshold = 0.5;  
[box,score,labels] = detect(detector,ptCloud,'Threshold',confidenceThreshold);
```

```
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');
```

Generate Detections



Evaluate Detector Using Test Set

Evaluate the trained object detector on a large set of point cloud data to measure the performance.

```
numInputs = 50;
% Generate rotated rectangles from the cuboid labels.
bds = boxLabelDatastore(testLabels(1:numInputs,:));
groundTruthData = transform(bds,@(x)createRotRect(x));
% Set the threshold values.
nmsPositiveIoUThreshold = 0.5;
confidenceThreshold = 0.25;
detectionResults = detect(detector,testData(1:numInputs,:),...
    'Threshold',confidenceThreshold);
% Convert the bounding boxes to rotated rectangles format and calculate
% the evaluation metrics.
for i = 1:height(detectionResults)
    box = detectionResults.Boxes{i};
    detectionResults.Boxes{i} = box(:,[1,2,4,5,7]);
end
metrics = evaluateDetectionAOS(detectionResults,groundTruthData, nmsPositiveIoUThreshold);
disp(metrics(:,1:2))
```

	AOS	AP
Car	0.89791	0.89791
Truck	0.75879	0.75879

SLAM 示例代码

```
%Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans  
openExample('nav/OfflineSLAMExample')
```

```
%Perform SLAM Using 3-D Lidar Point Clouds  
openExample('nav/PerformSLAMUsing3DLidarPointCloudsExample')
```

```
%Lidar 3-D Object Detection Using PointPillars Deep Learning  
openExample('deeplearning_shared/Lidar3DObjectDetectionUsingPointPillarsExample')
```