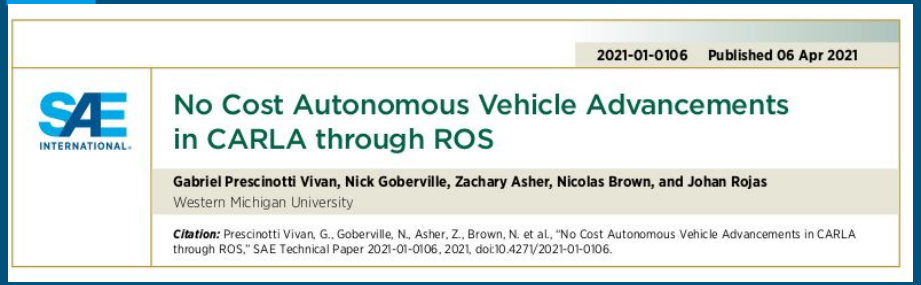
Intern Report

A report by Aarjan Budathoki

Overview

- ★ ADAS & ROS Study
- ★ ROS + Gazebo: Basic Simulation of Robotic Arm
- ★ VLP 16: Data acquisition & processing
- ★ PCL Library: Analyzing, processing, mapping indoor and outdoor environments
- ★ Lidar Dataset: Generation of a person dataset
- ★ Clustering
- ★ Kd-Tree
- ★ RANSAC Scratch Implementation

ADAS & ROS Study



2021-01-0106 Published 06 Apr 2021

SAE INTERNATIONAL

No Cost Autonomous Vehicle Advancements in CARLA through ROS

Gabriel Prescinotti Vivan, Nick Goberville, Zachary Asher, Nicolas Brown, and Johan Rojas
Western Michigan University

Citation: Prescinotti Vivan, G., Goberville, N., Asher, Z., Brown, N. et al., "No Cost Autonomous Vehicle Advancements in CARLA through ROS," SAE Technical Paper 2021-01-0106, 2021, doi:10.4271/2021-01-0106.

Transition from a simulation environment to a real environment not more challenging than developing algorithm in first place.

Conclusion

An autonomous vehicle algorithm was successfully developed. It utilized and combined different features, such as data collection from multiple sensors, information processing through computer vision and machine learning, and autonomous control via waypoint navigation and a PID feedback controller. The vehicle is also able to stop for obstacles, such as stop signs, other cars and pedestrians. Overall, it can achieve high velocities without losing control or producing excessive jerky movements. Based on this, we believe that this algorithm would be a viable solution to trial in a physical vehicle.

other existing solutions. The key takeaway of this research paper is that open-source simulation platforms are reliable enough in order to test low-cost, simple, and user-friendly solutions from research groups that do not have access to expensive, state-of-the-art hardware or software.

Physical Data collection = slower and time consuming, expensive

1. **Waymo = 20 million miles driven since 2009**
2. **Cruise = 0.5 million miles driven**

ROS2 & Gazebo Revise

Understanding ros concepts:

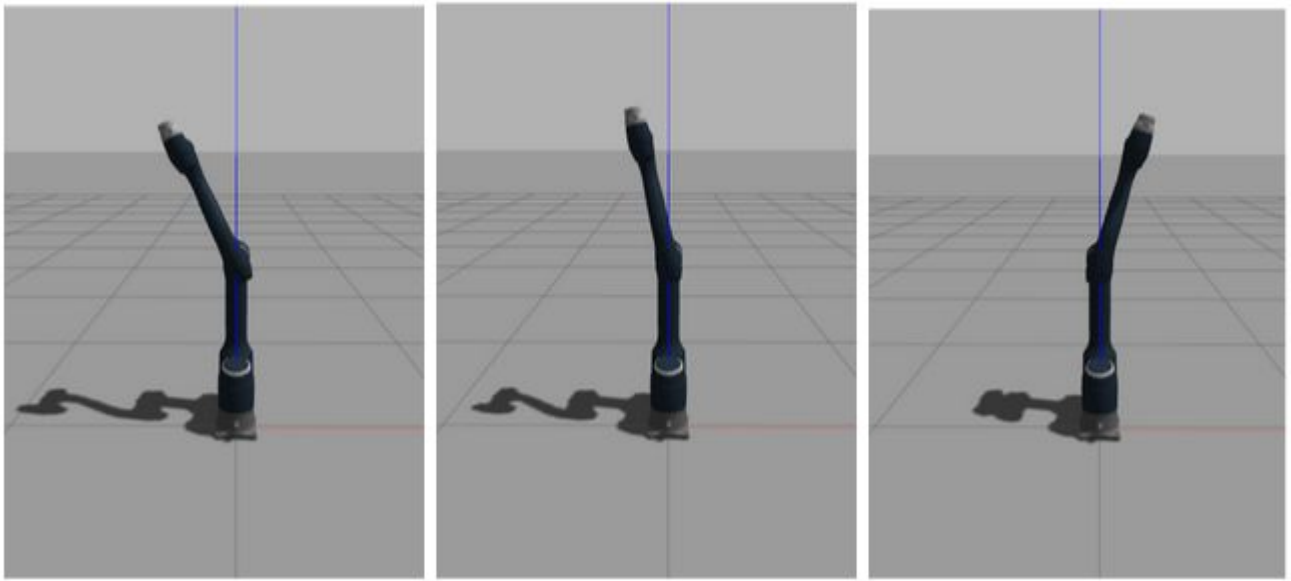
Services

Actions

Various Industrial Gripper and Gripper Interfacing Commands

ROS2 & Gazebo Revise

Simulation of Doosan Robotic Arm without Gripper in Gazebo world



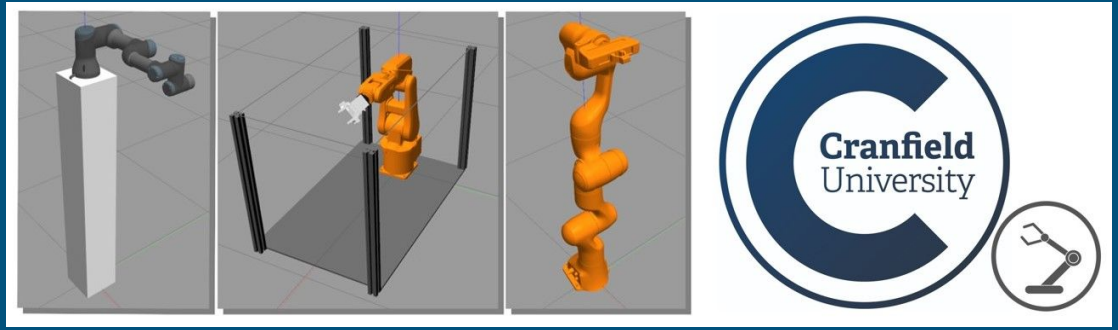
Carla Installation

./CarlaUE4.sh -prefernvidia

Robotic Arm & Motion Planning (moveit2)

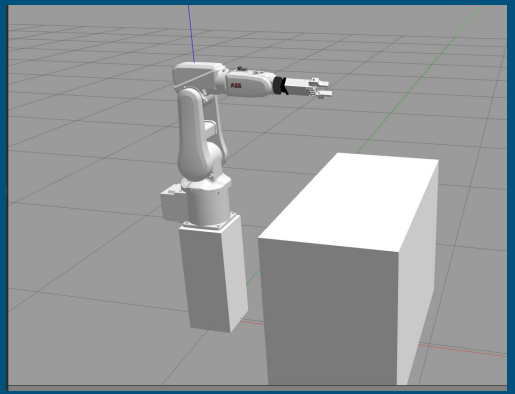
Robots like

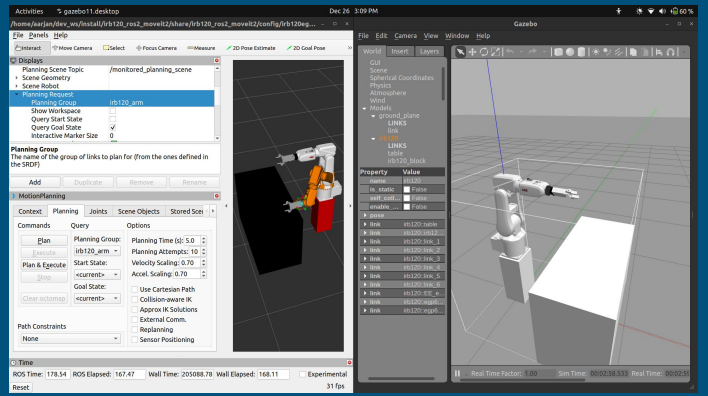
1. ABB Robot
2. Universal Robot
3. FANUC
4. KUKA
5. Panda



Features

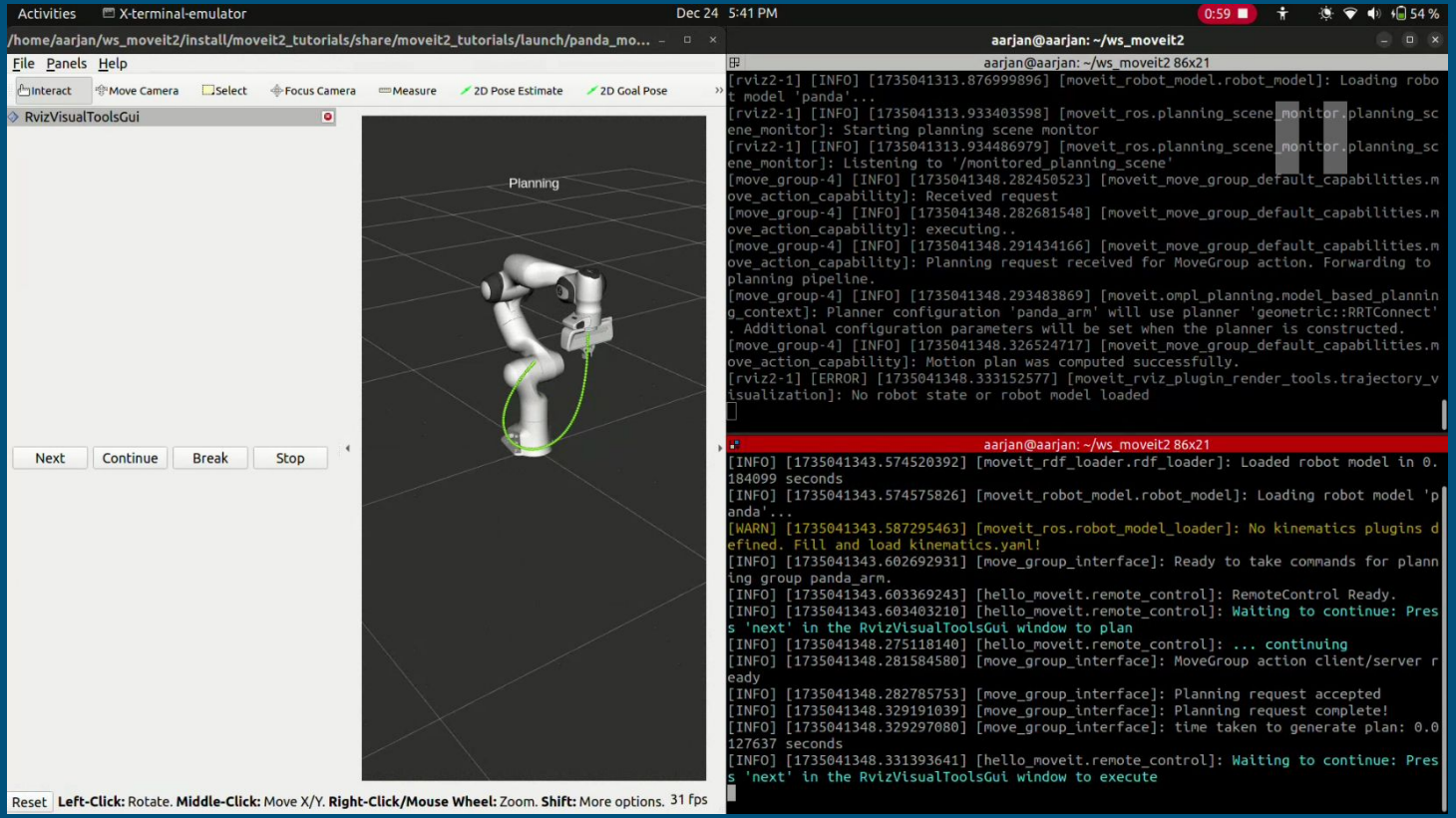
1. ros2_grasping
2. ros2_execution
3. ros2_data
4. ros2_actions





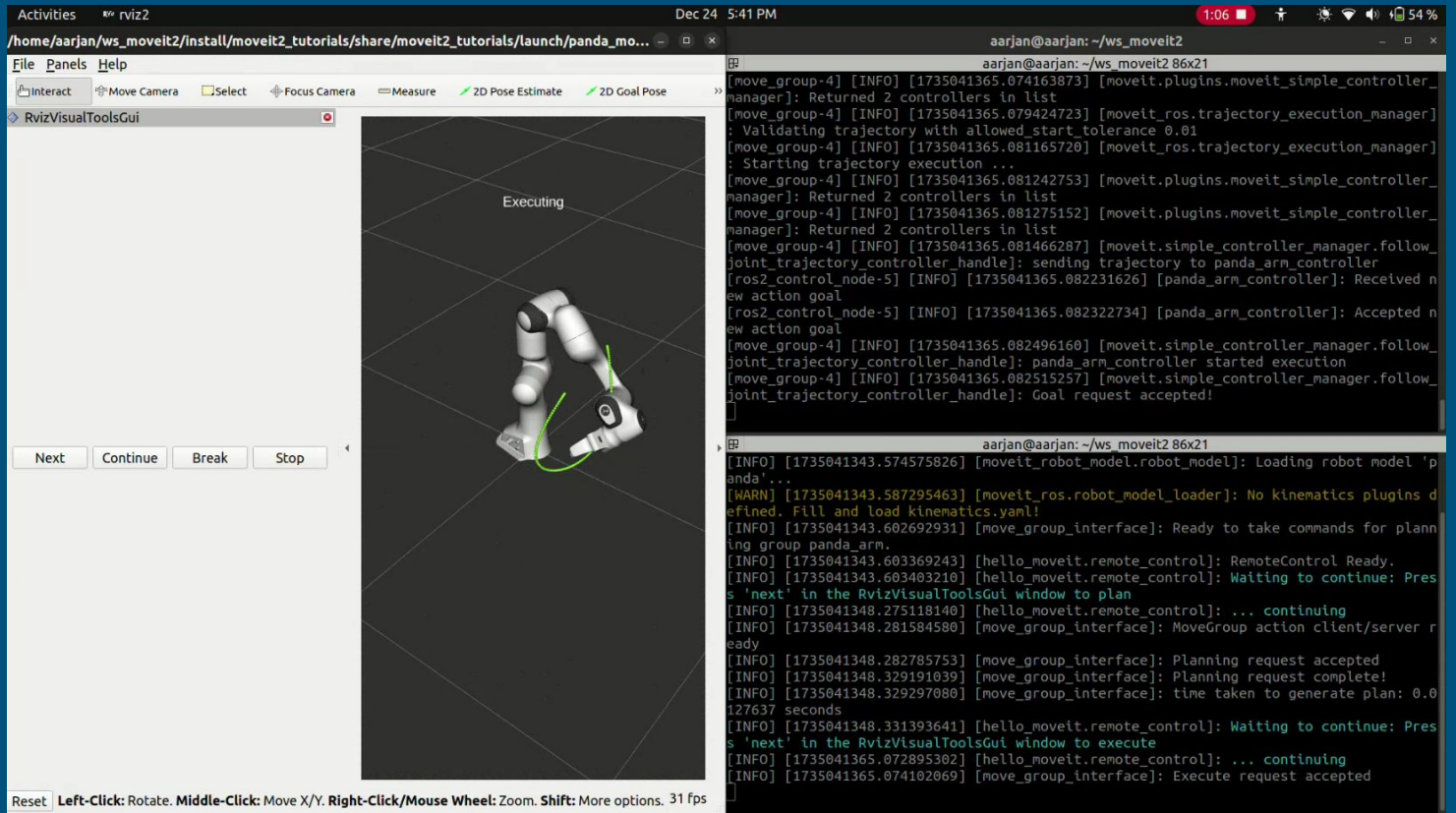
Motion Planning of Industrial Robotic Arms

- ros2 humble
- ubuntu 22.04
- moveit2

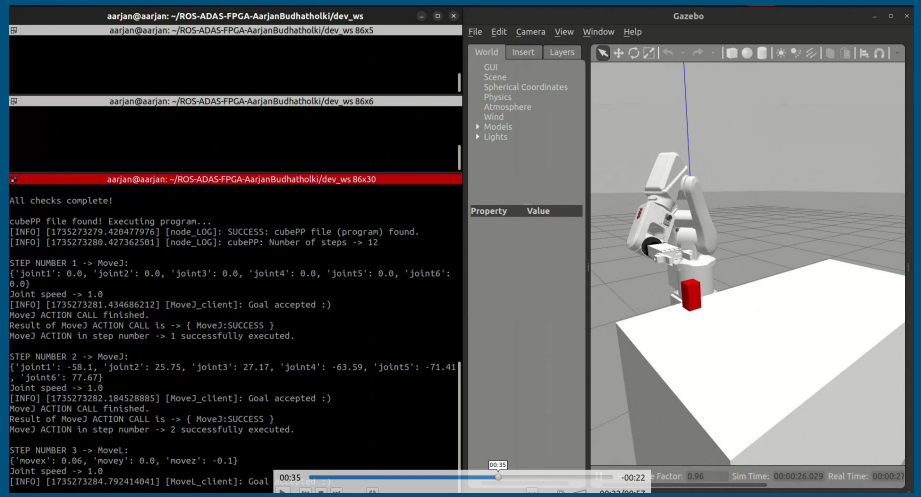


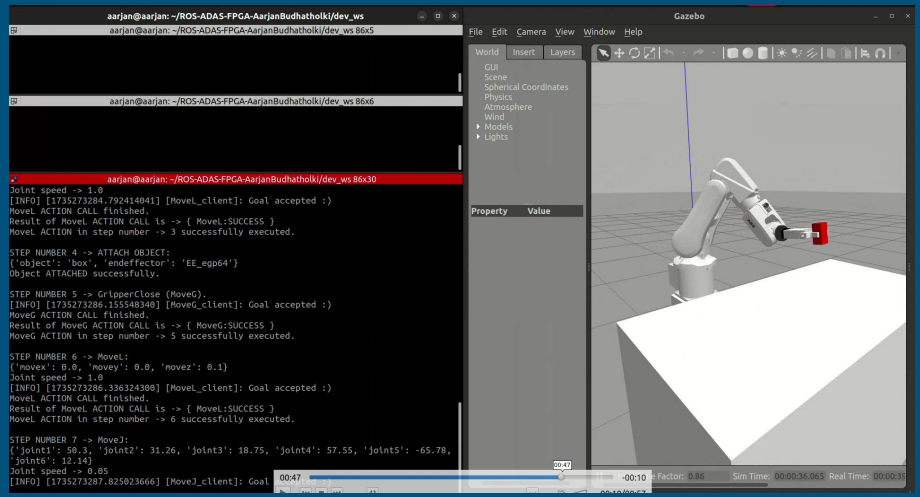
Motion Planning of Industrial Robotic Arms

- ros2 humble
- ubuntu 22.04
- moveit2



Pick & Place ABB Robot



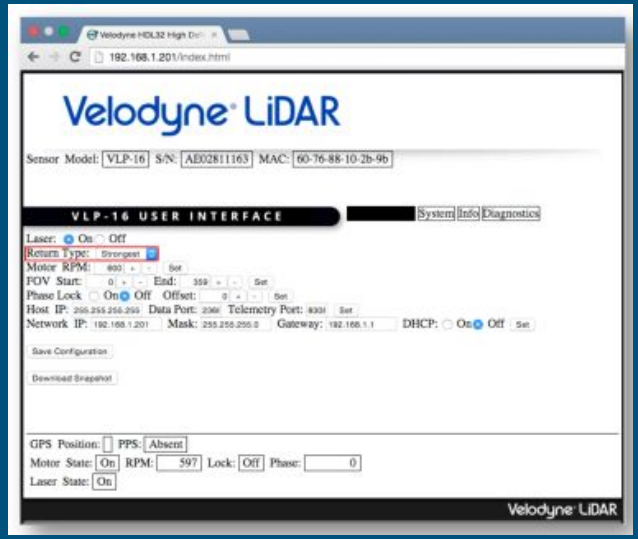


VLP 16: Lidar Datasheet

- 16 laser/detector pairs mounted in a compact housing
- Lasers fire thousands of times pers second, giving 3d pcd in real time
- 5-20 hz = rotations pers second = adjustable
- Vertical fov = 30
- Range: 100m
- Lidar Specifications:
  - Range Precision & Accuracy
  - Scan Pattern

Mode	Packets/Second	Megabits/Second
Strongest	754	~ 8
Last	754	~ 8
Dual	1508	~ 16

Table 1. Dual Return Data Rates



VLP 16: Lidar Datasheet

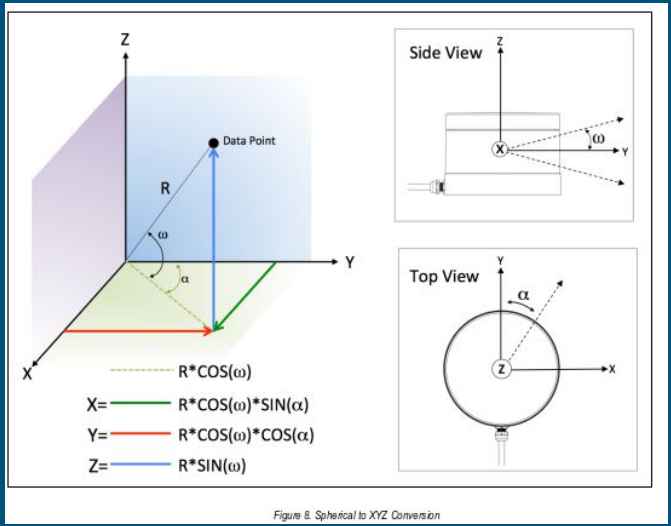


Figure 8. Spherical to XYZ Conversion

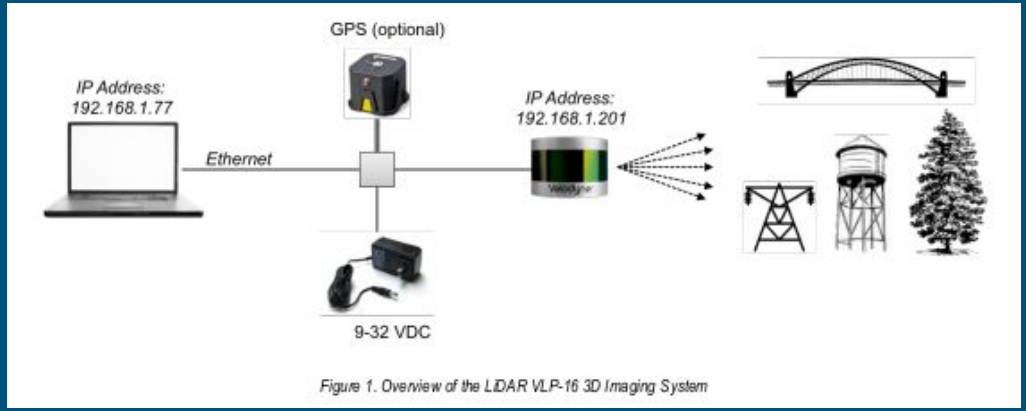


Figure 1. Overview of the LIDAR VLP-16 3D Imaging System

VLP 16: Data Format

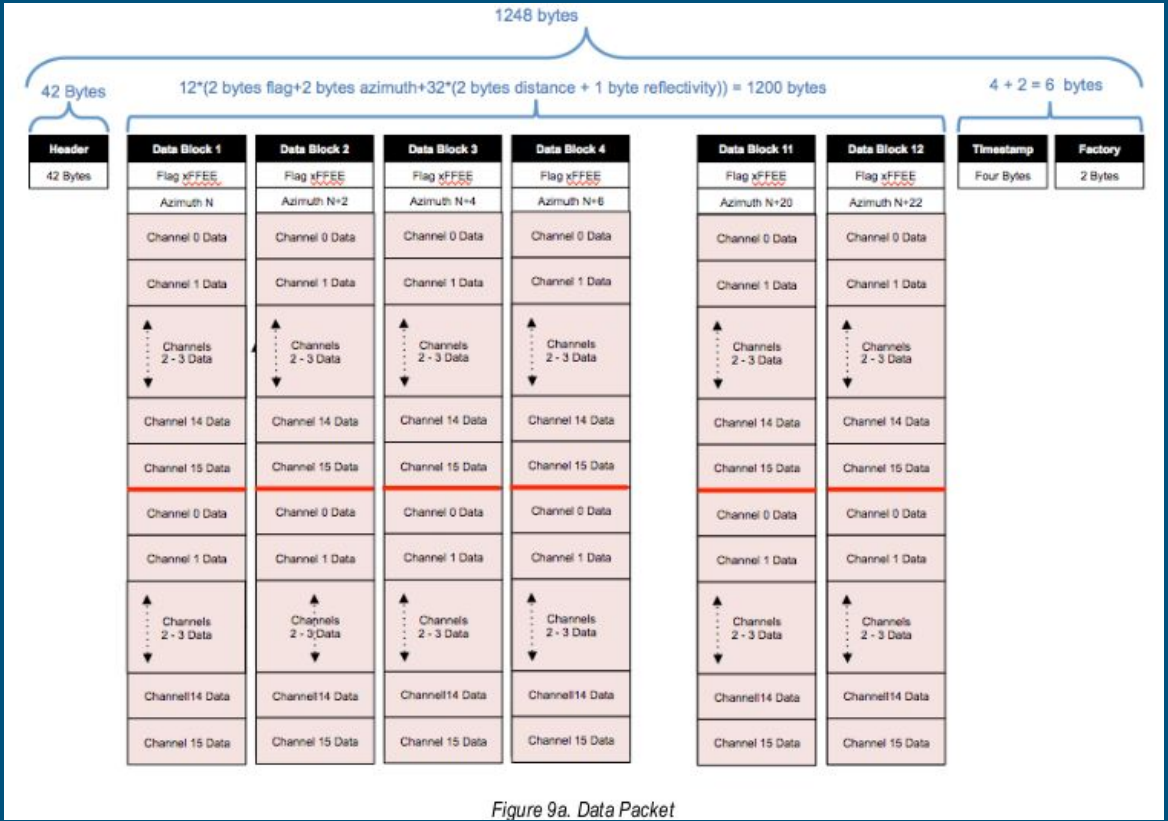
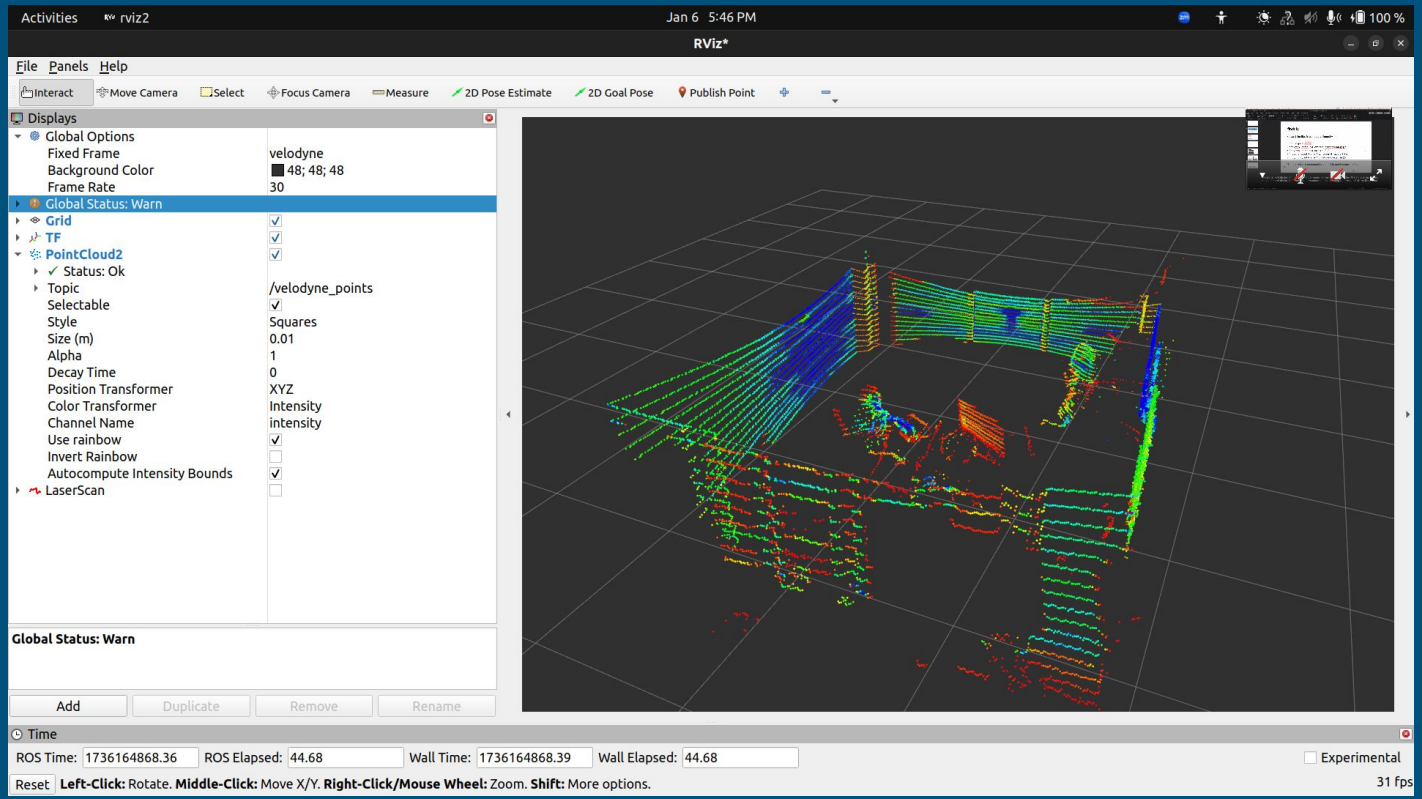


Figure 9a. Data Packet

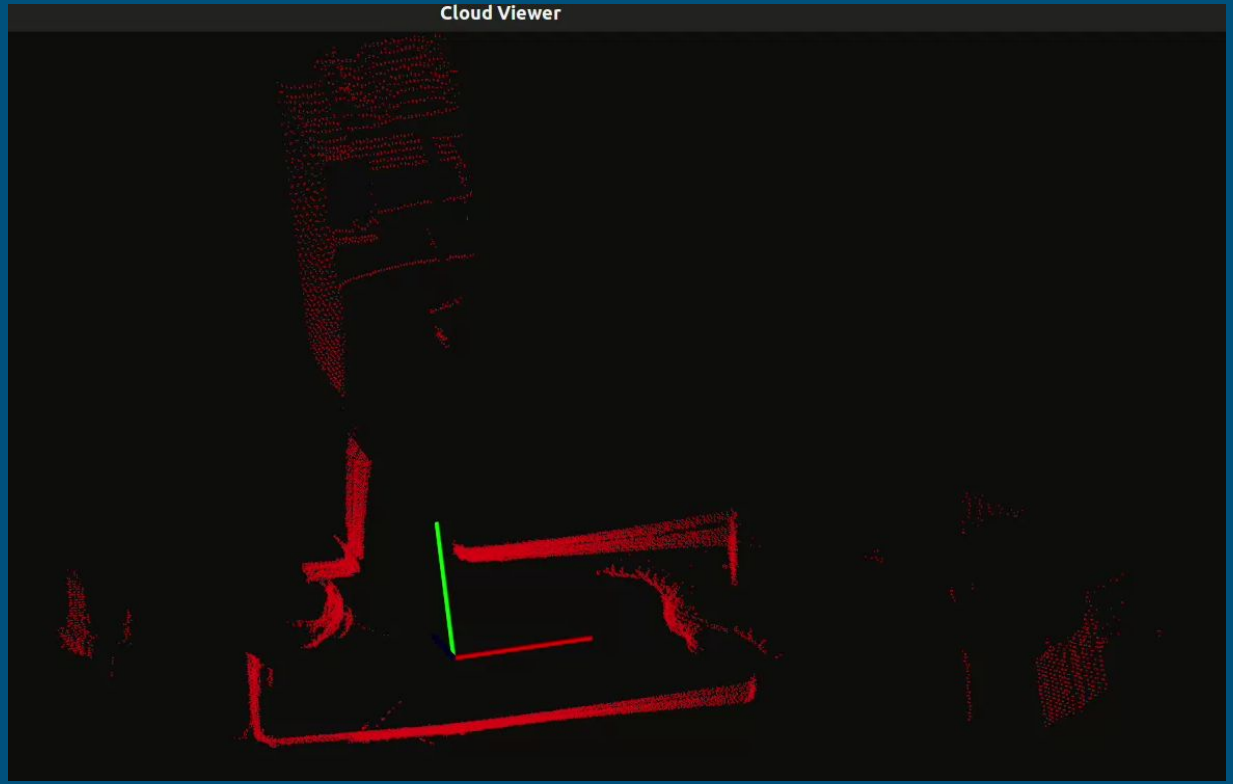
VLP 16: Data Visualization in RVIZ



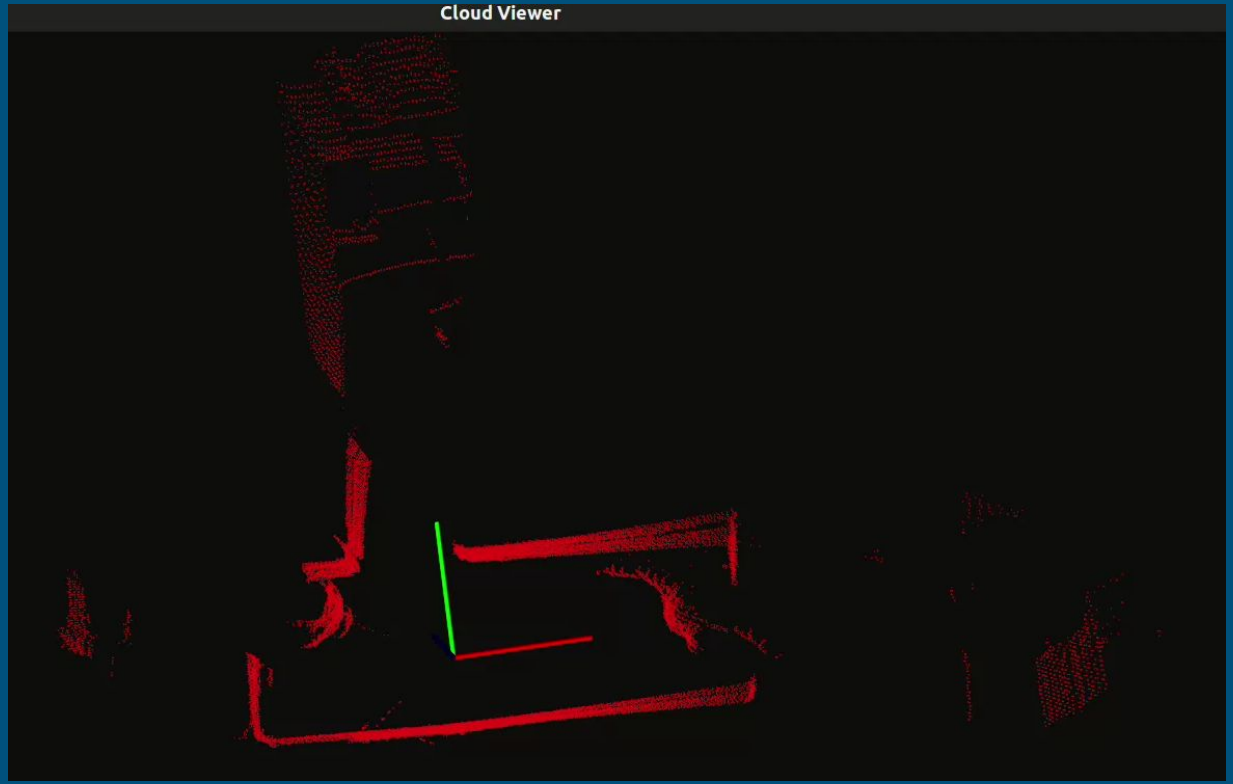
VLP 16: Data Acquisition, Processing & Visualization

- Lidar sends data through ethernet to laptop/pc.
- And sends data through UDP port.
- Lidar Driver
- Lidar Decoder
- Top view Visualization

VLP 16: Data acquisition & processing



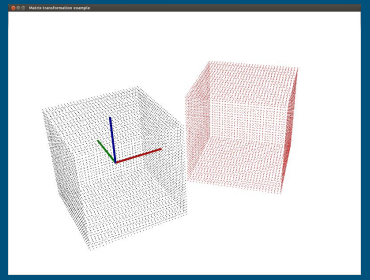
VLP 16: Data acquisition & processing

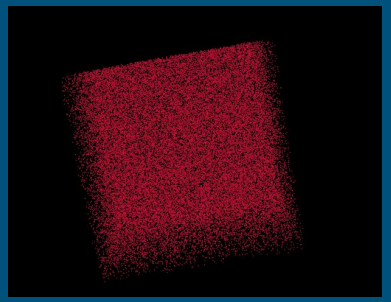


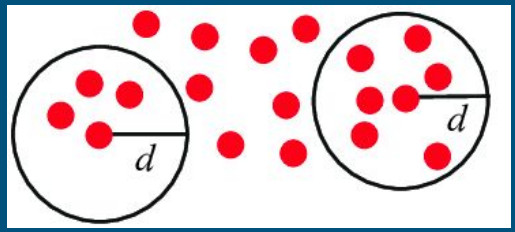
PCL Library

Followed tutorial of pcl library official website:

- Matrix Transform
- PassThrough Filter
- Voxel Grid Filter
- Statistical Outlier Removal
- Read/Write point cloud data from/to .pcd files
- Understanding of .pcd and .bin file formats.
- HDL Grabber
- Plane Model Segmentation
- Euclidean Cluster Extraction





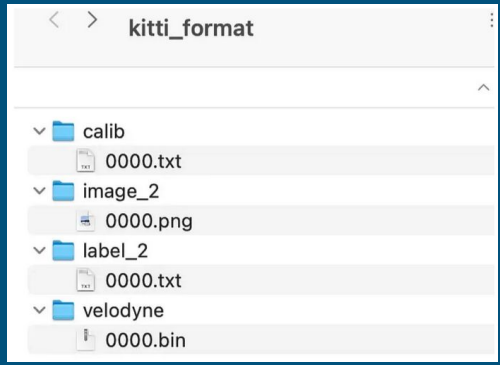


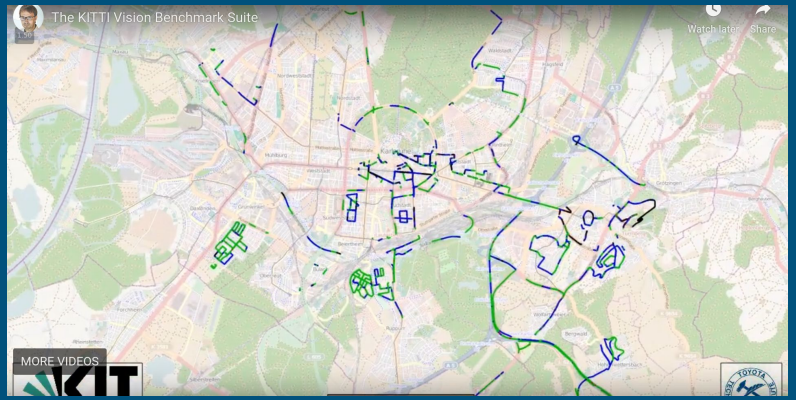
KiTTi Dataset

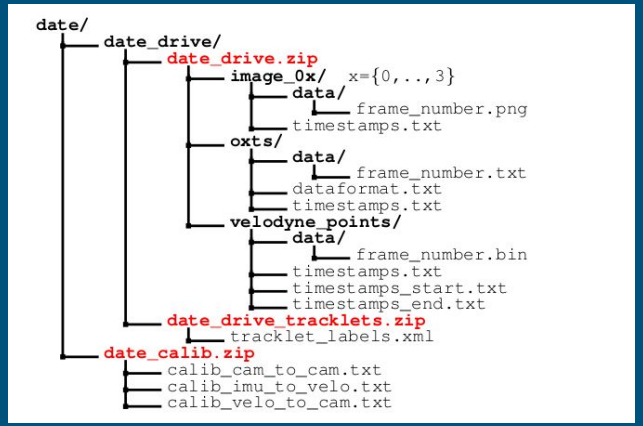
Welcome to the KITTI Vision Benchmark Suite!

We take advantage of our autonomous driving platform Annieway to develop novel challenging real-world computer vision benchmarks. Our tasks of interest are: stereo, optical flow, visual odometry, 3D object detection and 3D tracking. For this purpose, we equipped a standard station wagon with two high-resolution color and grayscale video cameras. Accurate ground truth is provided by a Velodyne laser scanner and a GPS localization system. Our datsets are captured by driving around the mid-size city of Karlsruhe, in rural areas and on highways. Up to 15 cars and 30 pedestrians are visible per image. Besides providing all data in raw format, we extract benchmarks for each task. For each of our benchmarks, we also provide an evaluation metric and this evaluation website. Preliminary experiments show that methods ranking high on established benchmarks such as Middlebury perform below average when being moved outside the laboratory to the real world. Our goal is to reduce this bias and complement existing benchmarks by providing real-world benchmarks with novel difficulties to the community.

Share







Lidar & Camera Frame Synchronization

LIDAR Rotational Speed and Data Packets

300 rpm 19sec 14318 packets packets/second 753.57
600 rpm 11 sec 8290 packets packets/second 753.63
1200 rpm 19sec 14318 packets packets/second 753.75

Results: hence independent of rotational speed of lidar, we always get 754 packets per second.

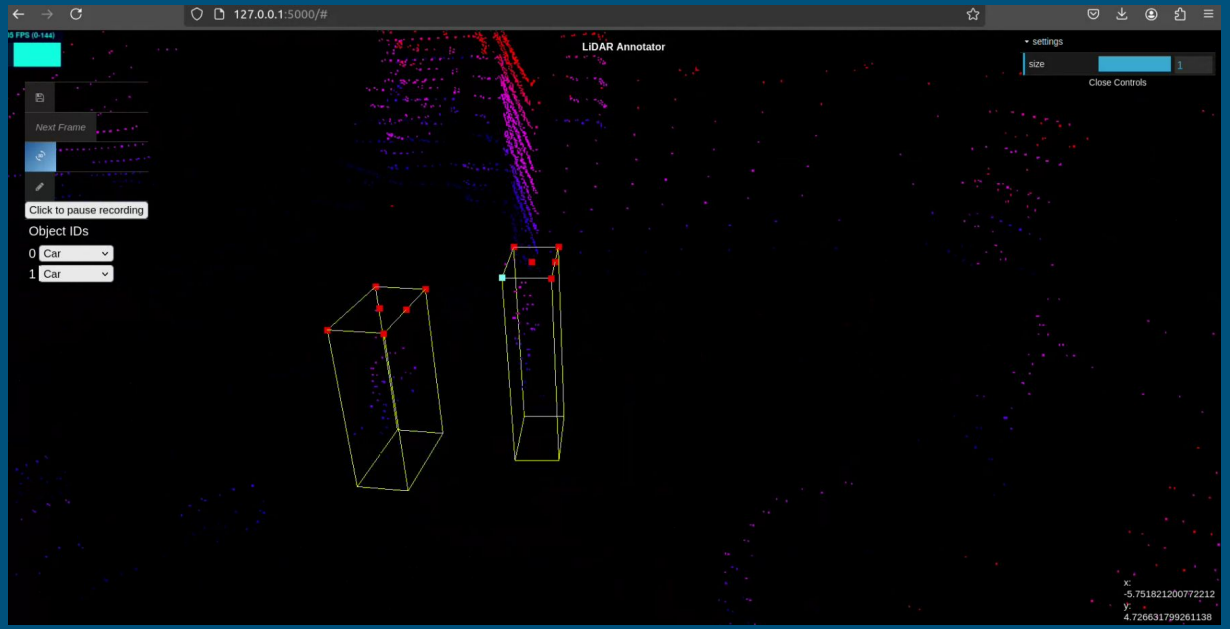
Create Lidar Person Dataset

Lidar Data Annotation Tools Review:

1. Label Cloud
2. BasicAi
3. Latte
4. Kitti own developed software not open source

Latte

Web browser based tool

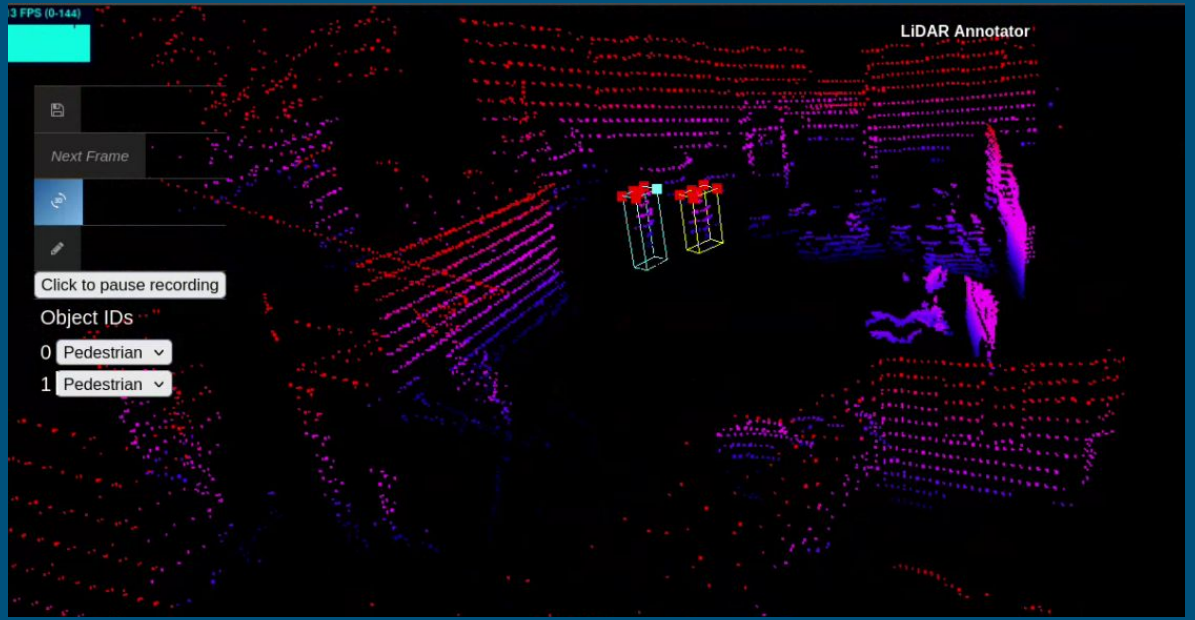


Latte

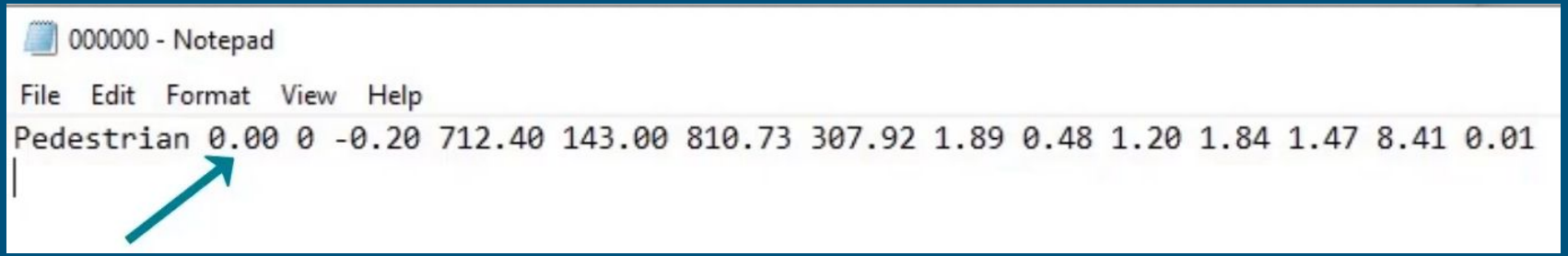
JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON


```
bounding boxes:
  0:
    center:
      x:        -2.9580141270457387
      y:        1.6981649866766613
      z:        -0.13765254616737366
    width:      0.7602598100272502
    length:     0.4186926976810761
    height:     1.6109680533409119
    angle:      0
    object_id:  "pedestrian"
```

Labeled Object file



<class names> <truncation> <occlusion> <alpha> <bbox coordinates> <3D dimensions> <location> <rotation_y> <score>

Obstacle Clustering

- Read/load pcd file
- Downsampling
  - Voxel downsampling
- Segmentation(Road and other obstacles segmentation)
  - RANSAC/MSAC
- Clustering
  - Euclidean clustering

Clustering Algorithm Review

- Clustering LiDAR Data with K-means and DBSCAN: paper review
  - DBSCAN is preferable for segmenting LiDAR data due to its ability to handle arbitrary cluster shapes.
- OPTICS Clustering
  - High computation not good
- Euclidean Clustering: PCL

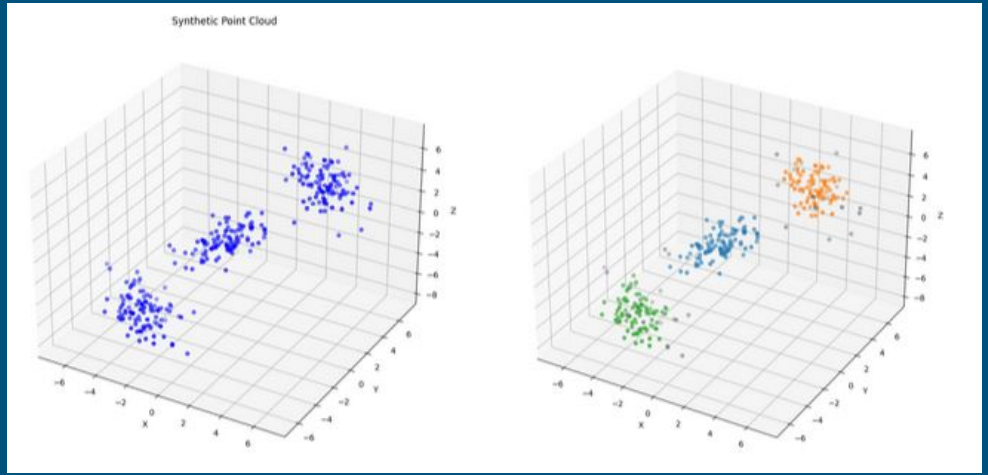
Ground Removal using M-SAC

Before the application of the clustering algorithms, it was necessary to pre-process the data to remove the ground. This is an important step because firstly, the goal is not detecting the ground plane, and secondly, the ground points connect separate objects, such as cars and pedestrians, so it can affect the clustering process. By removing the ground plane, there will be fewer points to allocate, resulting in better-separated objects. Consequently, it becomes easier to identify obstacles and visualize clusters.

The method used for ground extraction was the M-estimator SAmple Consensus (MSAC) algorithm which is a variant of the RANdom SAmple Consensus (RANSAC) algorithm (Azam et al., 2018). The objective is to fit a plane to the point cloud data that will contain the ground points denominated as inlier points. The outliers are the points that do not belong to the plane, thus the interest points for clustering. As an example, Figure 3 shows the original point cloud data for scenario 1 (top) and the points remaining after the ground removal process (bottom), indicating that the ground removal procedure was successful.

Density Based Spatial Clustering DBSCAN

- Groups points based on density, allowing clusters of arbitrary shapes.
- Requires tuning parameters
  - ε (radius)
  - m (minimum points per cluster).
- Can effectively identify outliers as noise.



RANSAC Algorithm

- Start with the input data
- Pick random samples
- Fit a mathematical model (line, curve, 3d shape, ...)
- Compute a cost by checking how many points fit the model
- Repeat until you found the model with the lowest cost

Parameters:

Estimator, min_samples, maxIterations, stop_score, Loss

RANSAC is a resampling technique that generates candidate solutions by using the minimum number observations (data points) required to estimate the underlying model parameters. As pointed out by Fischler and Bolles [1], unlike conventional sampling techniques that use as much of the data as possible to obtain an initial solution and then proceed to prune outliers, RANSAC uses the smallest set possible and proceeds to enlarge this set with consistent data points [1].

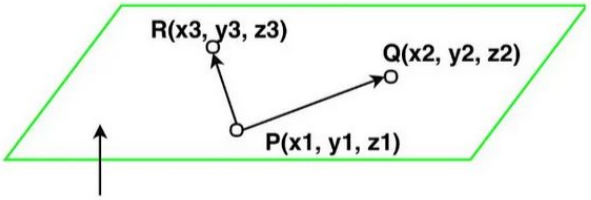
The basic algorithm is summarized as follows:

Algorithm 1 RANSAC

1: Select randomly the minimum number of points required to determine the model parameters.
2: Solve for the parameters of the model.
3: Determine how many points from the set of all points fit with a predefined tolerance ϵ.
4: If the fraction of the number of inliers over the total number points in the set exceeds a predefined threshold τ, re-estimate the model parameters using all the identified inliers and terminate.
5: Otherwise, repeat steps 1 through 4 (maximum of N times).

RANSAC PseudoCode

Step 1 :: Select a random set of points(3 points for a forming a plane)



Plane Equation in Cartesian form

$$ax + by + cz + d = 0$$

Step 2 :: Calculate the parameters required for the plane equation.

Plane Equation in Cartesian form

$$ax + by + cz + d = 0$$

If 3 points are provided, then the constants a, b, c, and d can be derived using the following formulas :

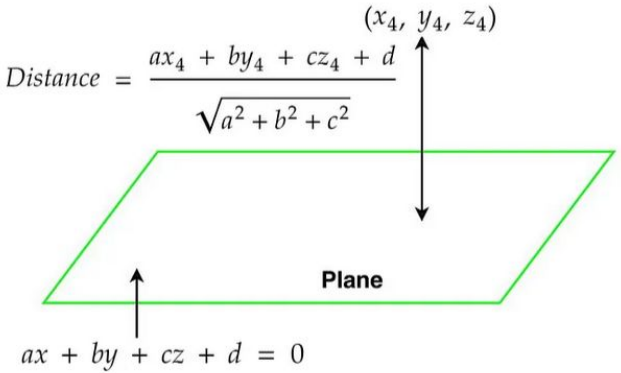
$$a = [(y_2 - y_1)(z_3 - z_1) - (z_2 - z_1)(y_3 - y_2)]$$

$$b = [(z_2 - z_1)(x_3 - x_1) - (x_2 - x_1)(z_3 - z_2)]$$

$$c = [(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_2)]$$

$$d = -(ax + by + cz)$$

Step 3 :: Calculate the deviation of all the points in the point cloud from the plane using a distance estimate.



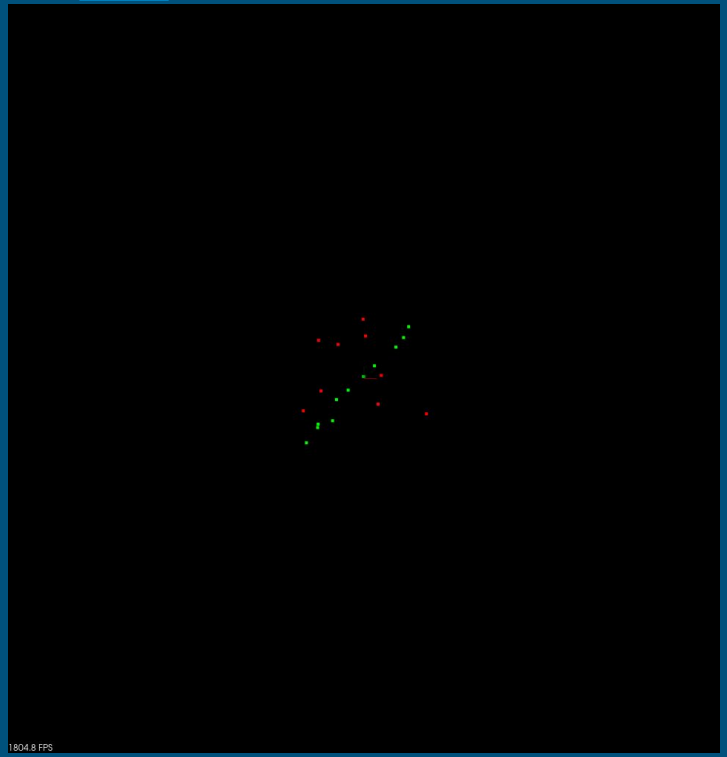
Step 4 :: If the distance is within the THRESHOLD then add the point as an inlier.

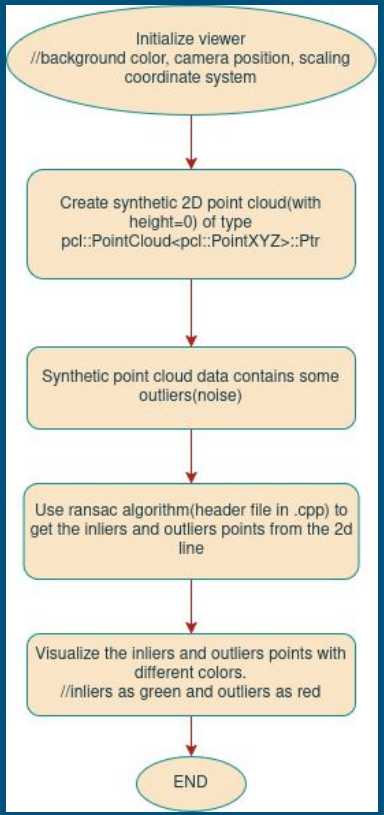
Step 5 :: Store the plane points and points having the maximum number of inliers.

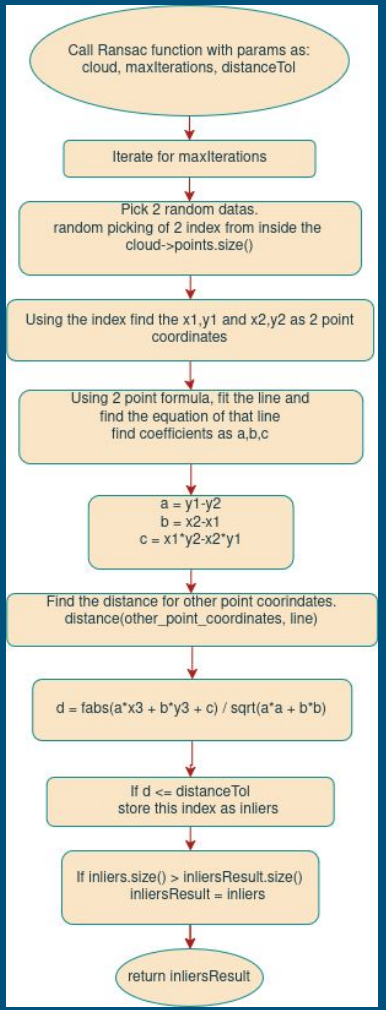
Step6:: Repeat the process again for MAX_ITERATIONS

After the RANSAC processing is complete, the plane having the maximum number of inliers is the best estimate of the ground plane. Any points cluster above this ground plane can be classified as an object, obstacle or traffic signs.

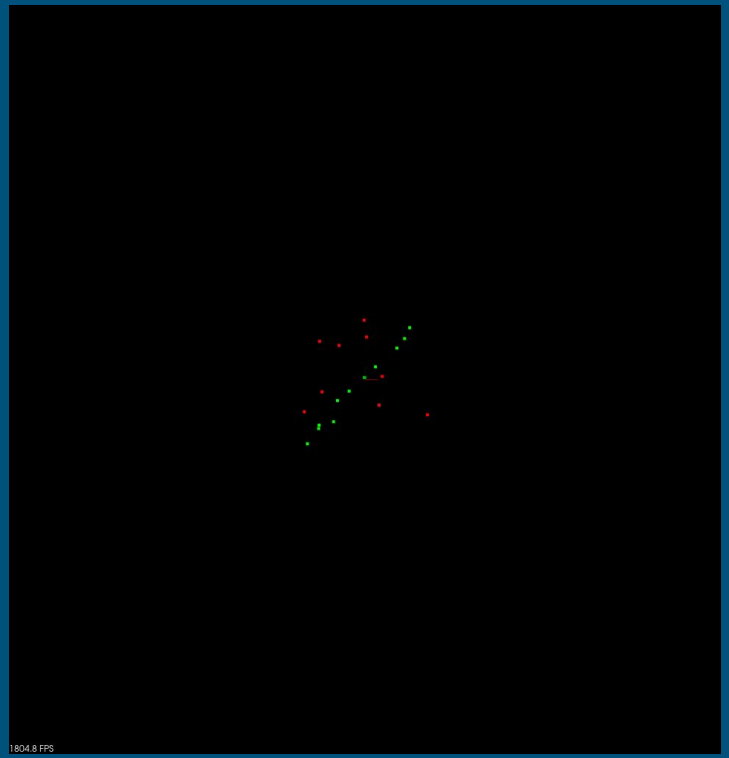
Ransac for 2D lines

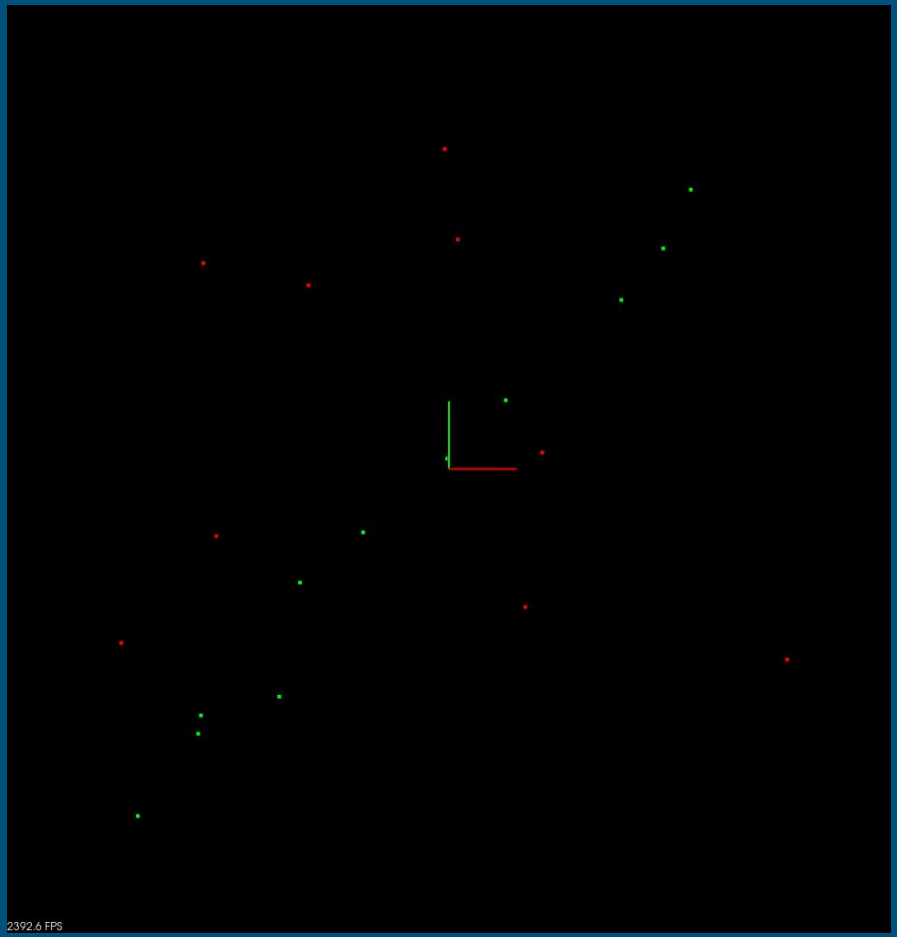






Ransac for 2D lines

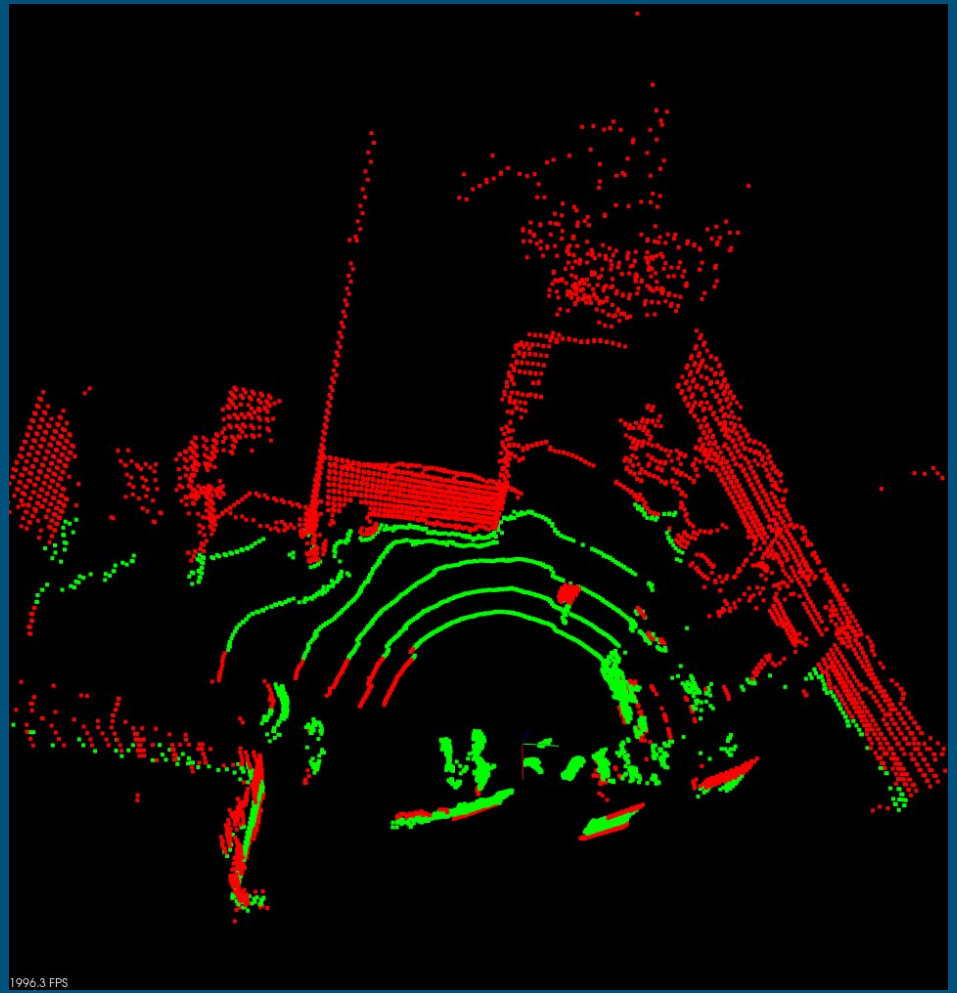




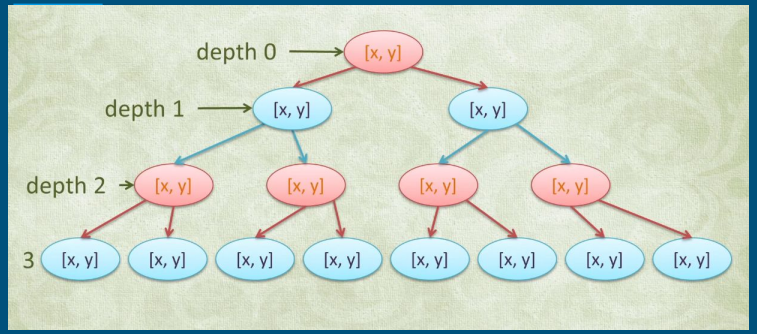
Ransac for Planes

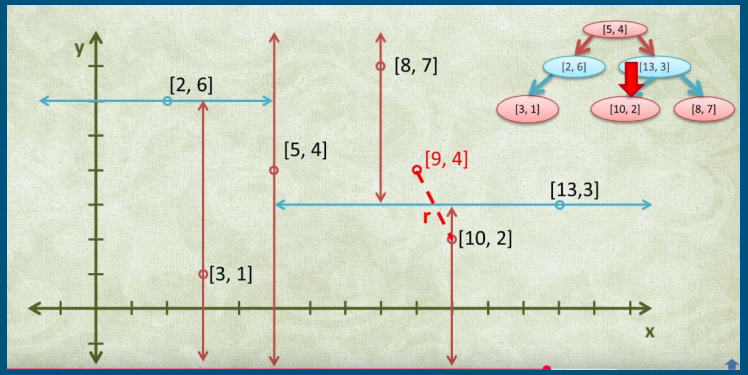
- Red = obstacles
  - Pedestrian, cyclists, cars

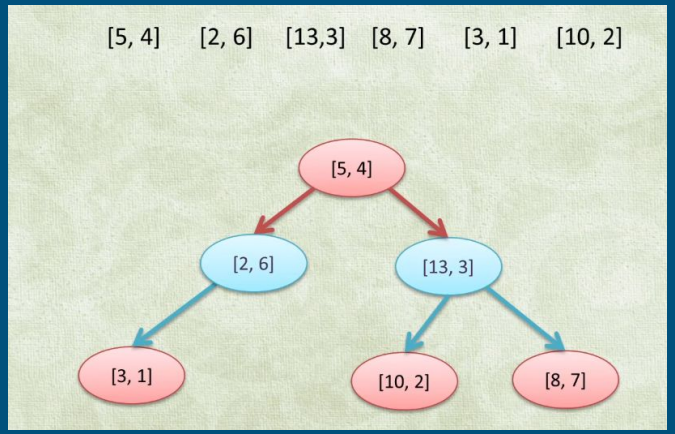
- Green = plane
  - Road



Kd-Tree

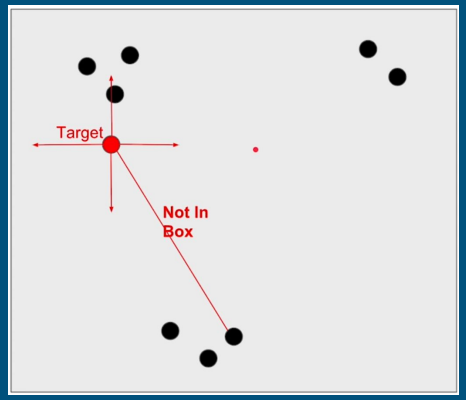


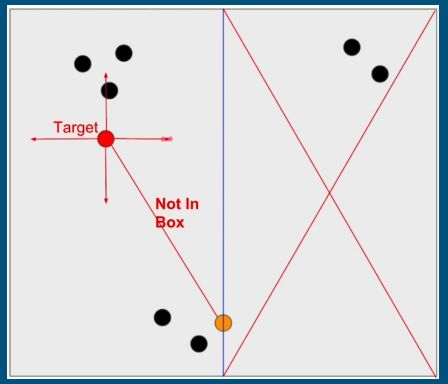


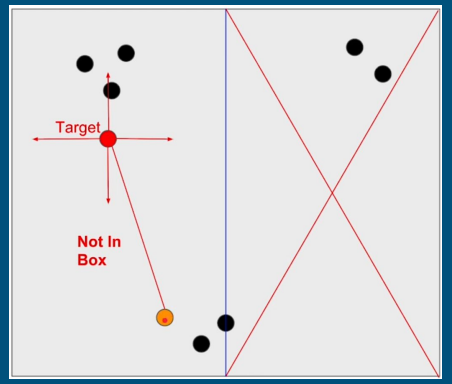


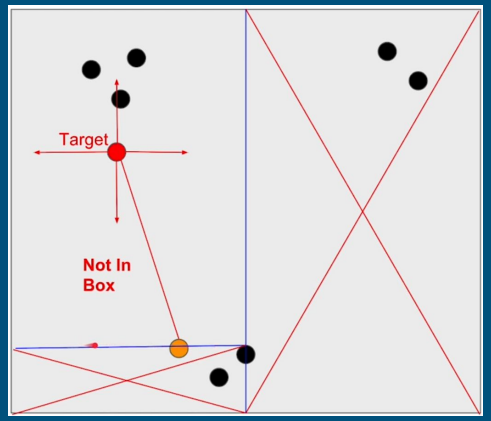
Optimized data structure for storing the points of the point cloud data(pcd) for easy search and retrieval

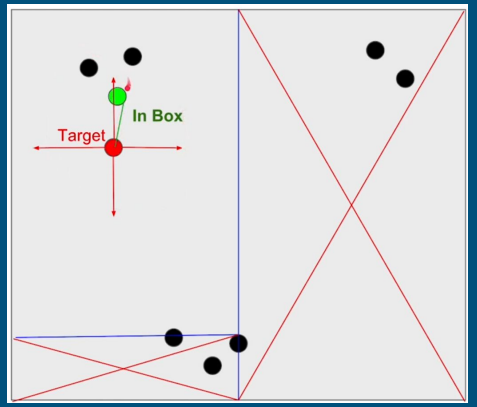
Fast Search using Kd-Tree

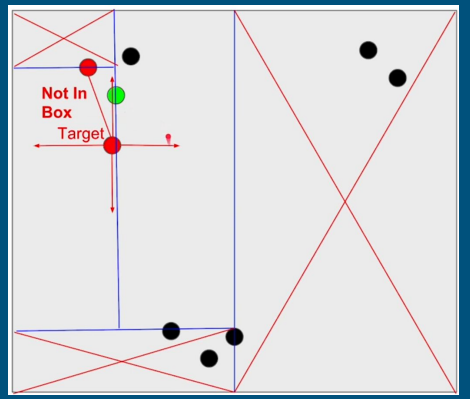






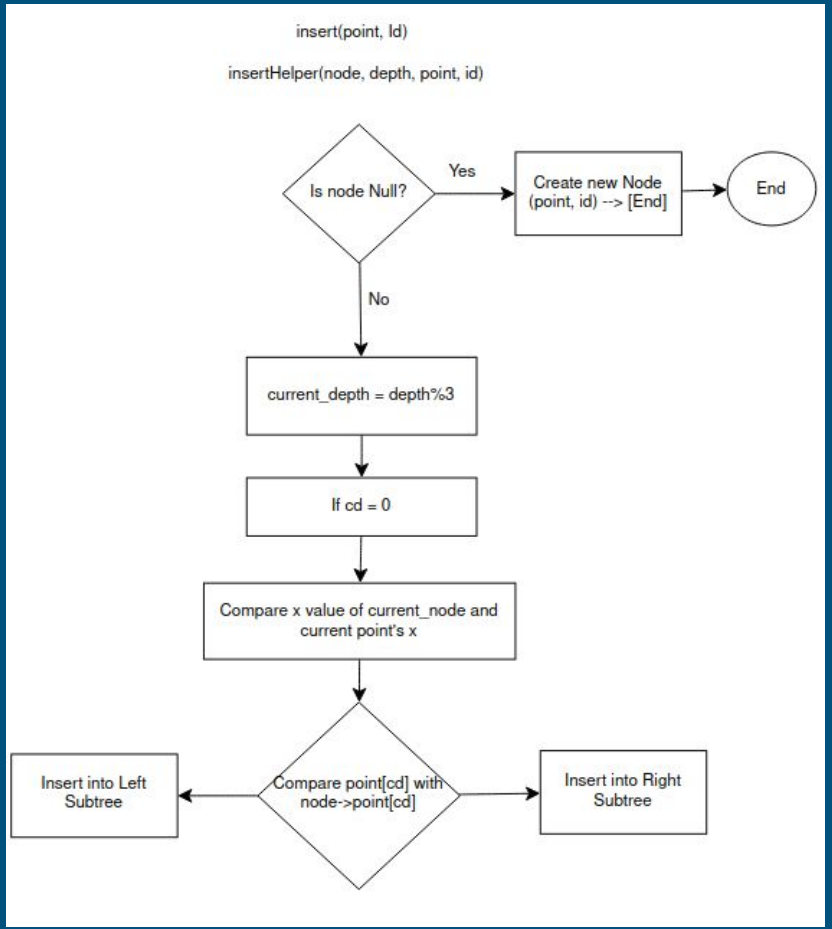




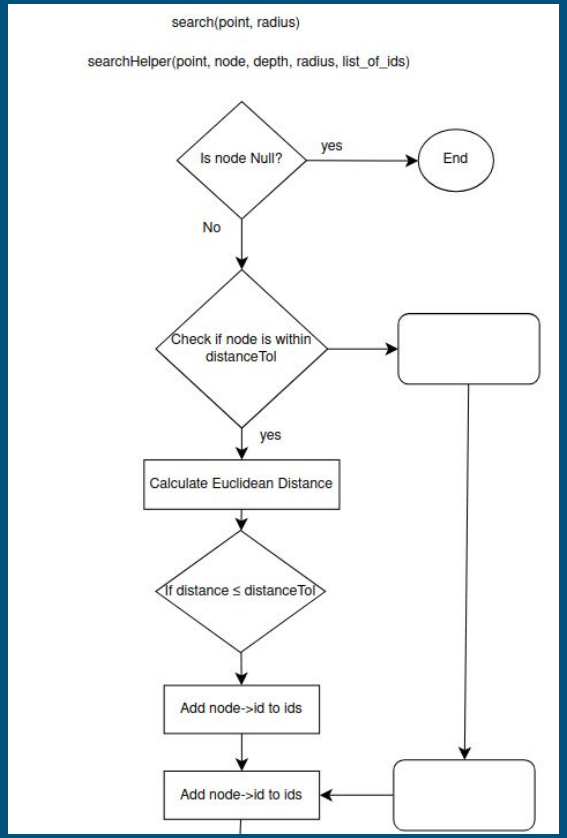


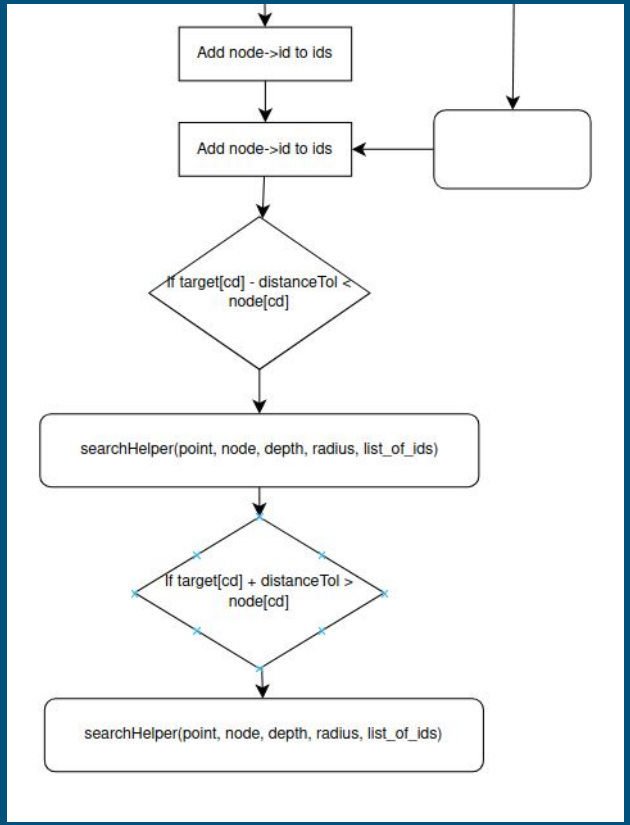
Continued

Kd-Tree Points Insertion



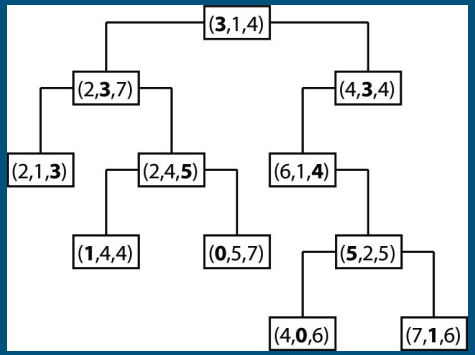
Radius Search in KdTree





Kd-Tree Implementation

For floating points x, y, z



Output

```
Test Search
0,2,7,10,11,12,
```

```
// Create data
std::vector<std::vector<float>> points = {{3.0, 1.0, 4.0},  // 0 d = sqrt(9)
                                          {2.0, 3.0, 7.0},  // 1
                                          {4.0, 3.0, 4.0},  // 2 d = sqrt(3)
                                          {2.0, 1.0, 3.0},  // 3
                                          {2.0, 2.0, 3.0},  // 4
                                          {2.0, 2.0, 4.0},  // 5
                                          {2.0, 4.0, 5.0},  // 6
                                          {6.0, 1.0, 4.0},  // 7 d = sqrt(3)
                                          {1.0, 4.0, 4.0},  // 8
                                          {0.0, 5.0, 7.0},  // 9
                                          {5.0, 2.0, 5.0},  // 10 d = 0
                                          {4.0, 0.0, 6.0},  // 11 d = sqrt(6)
                                          {7.0, 1.0, 6.0}}; // 12 d = sqrt(6)
```

```
KdTree *tree = new KdTree;

for (int i = 0; i < points.size(); i++)
    tree->insert(points[i], i);

int it = 0;

std::cout << "Test Search" << std::endl;
std::vector<int> nearby = tree->search({5.0, 2.0, 5.0}, 3.0);

for (int index : nearby)
    std::cout << index << ",";
std::cout << std::endl;
```

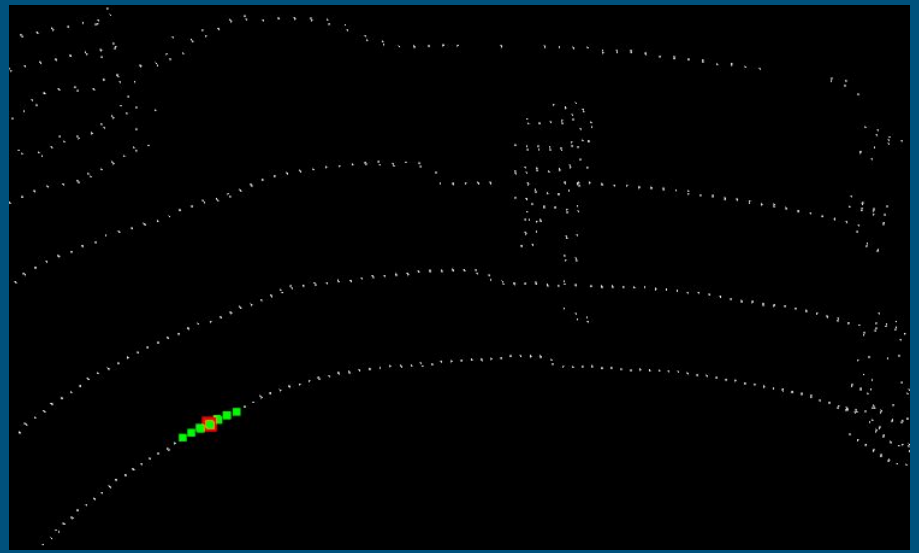
Kd-Tree Implementation

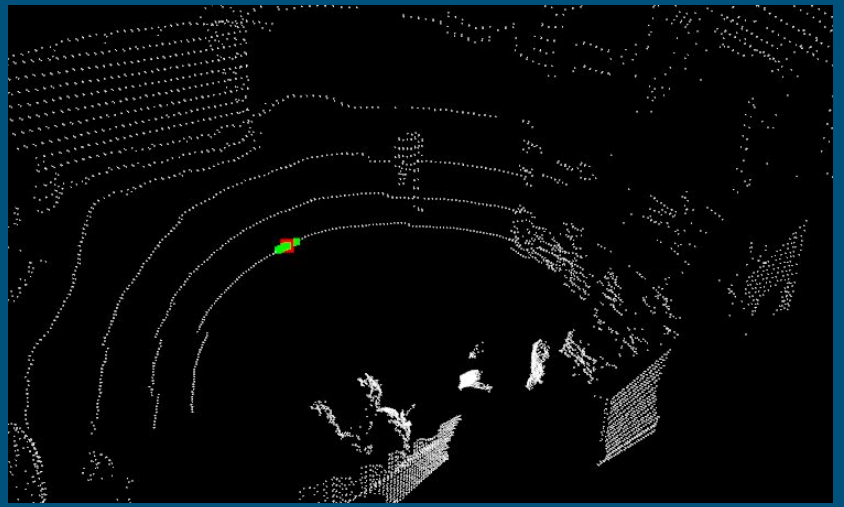
Load pcd file format in pcl::PointCloud<pcl::PointXYZI>::Ptr cloud

Select any 1 random point from the cloud

```
int i = 0;
for (const auto &point : inputCloud->points)
{
    tree.insert(point, i++);
}

int randomIdx;
pcl::PointXYZI searchPoint;
float radius = 0.2;
std::vector<int> pointIdxRadiusSearch;
std::vector<float> c;

for (int i = 0; i < 5; i++)
{
    randomIdx = rand() % inputCloud->points.size();
    searchPoint = inputCloud->points[randomIdx];
    std::cout << "index: " << randomIdx << std::endl;

    // scratch kd-tree
    pointIdxRadiusSearch = tree.search(searchPoint, radius);

    // pcl kdtree
    // kdtree.radiusSearch(searchPoint, radius, pointIdxRadiusSearch, pointRadiusSquaredDistance);

    visualizeKdTreeSearch(inputCloud, searchPoint, pointIdxRadiusSearch);
}
```

Kd-Tree Implementation

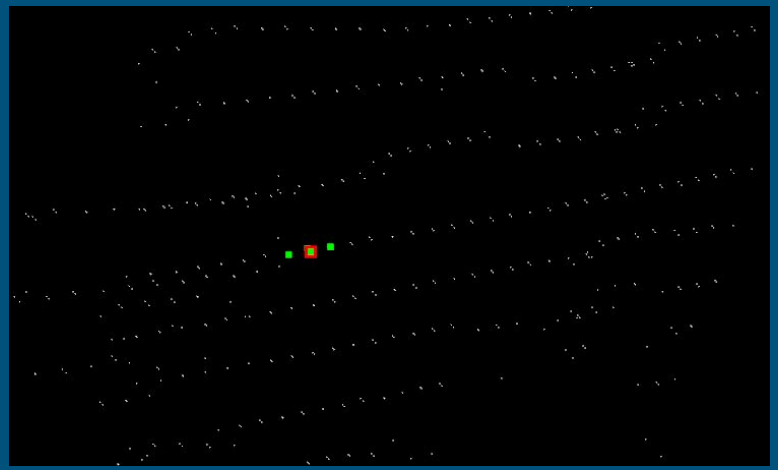


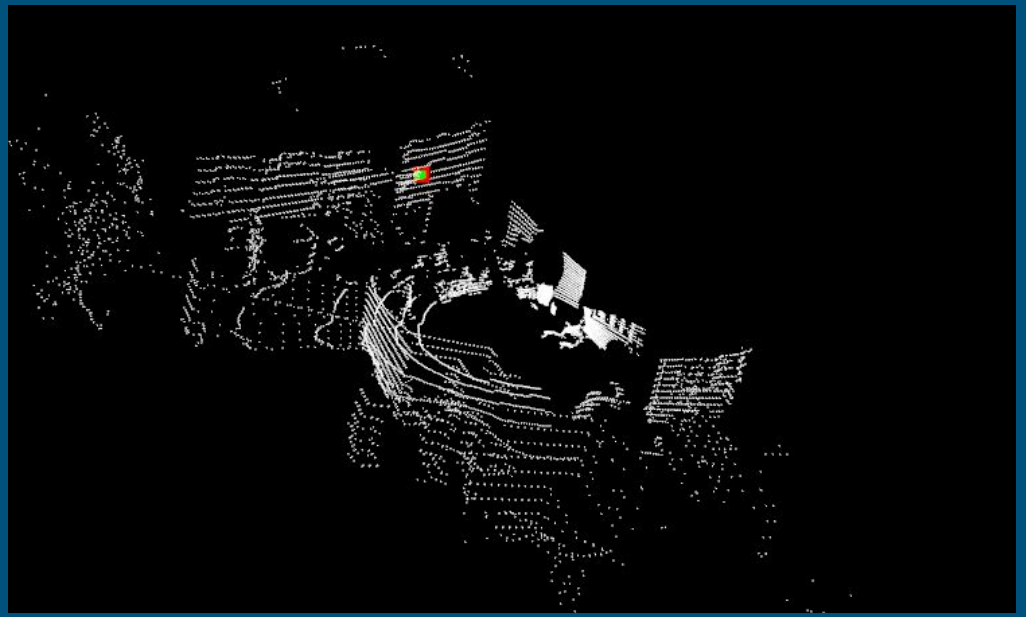


Select any 1 random point from the cloud

Searched point Red
Neighbor points Green

Kd-Tree Implementation



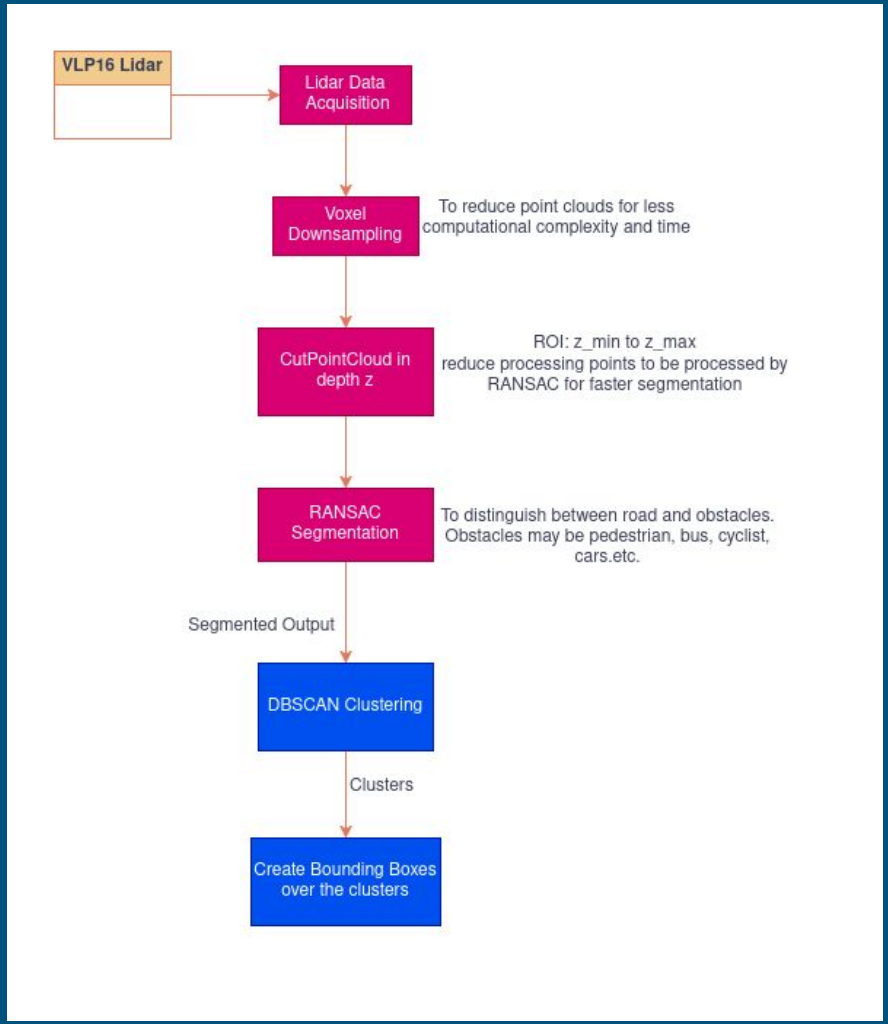


Select any 1 random point from the cloud

Visualization:
Search point in red
Neighbor point in green

Continued

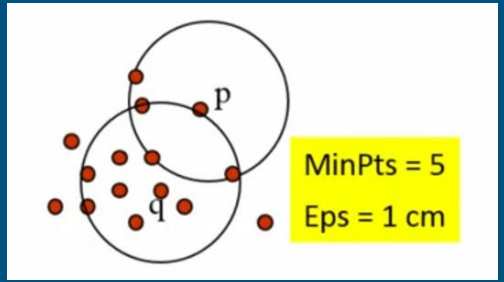
Flow:


VLP16 Lidar
Lidar Data Acquisition
Voxel Downsampling
To reduce point clouds for less computational complexity and time
CutPointCloud in depth z
ROI: z_min to z_max
reduce processing points to be processed by RANSAC for faster segmentation
RANSAC Segmentation
To distinguish between road and obstacles. Obstacles may be pedestrian, bus, cyclist, cars.etc.
Segmented Output
DBSCAN Clustering
Clusters
Create Bounding Boxes over the clusters

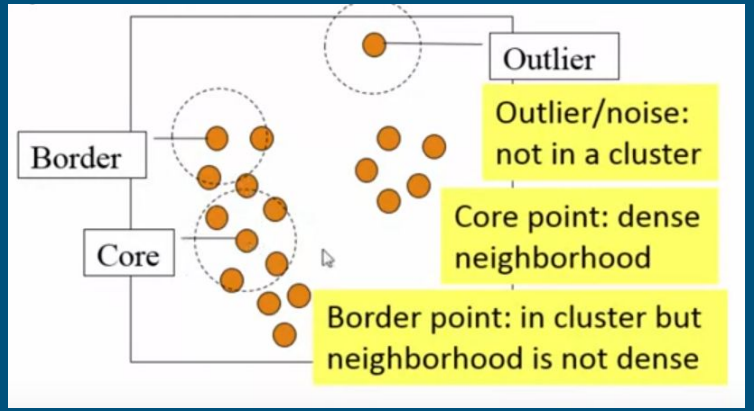
Density-based Spatial Clustering of Applications with Noise

Data clustering algorithm proposed in 1996.

Eps: max radius of the neighborhood
minPts: min no of points in the neighborhood of a point



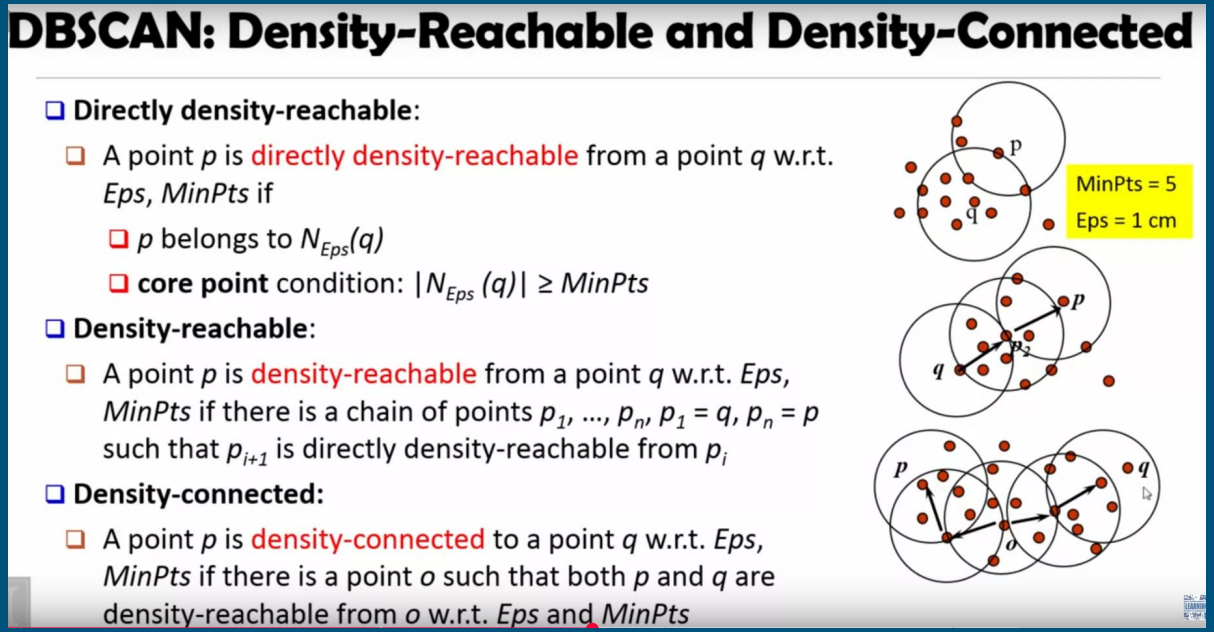
Robust to noise points away from the density core
No need to know the number of clusters.
Clusters of arbitrary shape can be found.



1. CorePoint: q such that it includes minPts neighbors
2. BorderPoint: q such that it doesn't have enough minPts neighbors
3. NoisePoint: cannot form a cluster with enough minPts and cannot be reached by any core points

Density-based Spatial Clustering of Applications with Noise: Algorithm

1. Arbitrarily select a point p
2. Retrieve all points density-reachable from p wrt eps and minPts.
3. If p is a corePoint, a cluster is formed
4. If p is a borderPoint, no points are density reachable from p, then DBSCAN visits the next point
5. Continue the process until all of the points have been processed.



DBSCAN Algorithm cpp implementation github review

https://github.com/james-yoo/DBSCAN

DBSCAN / benchmark_hepta.dat

james-yoo Add files via upload

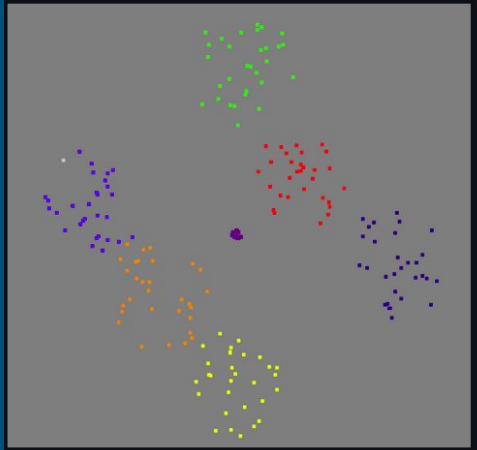
Code | Blame | 213 lines (213 loc) · 6.46 KB

```
1   212
2   -0.063274,0.027734,0.022683,1
3   -0.000731,0.048211,0.069198,1
4   -0.060767,-0.00908,0.053085,1
5   0.013252,-0.011876,0.055324,1
6   -0.054508,-0.003813,0.001738,1
7   0.02418,0.068275,0.033462,1
8   -0.029308,0.059849,-0.06326,1
```

benchmark dataset

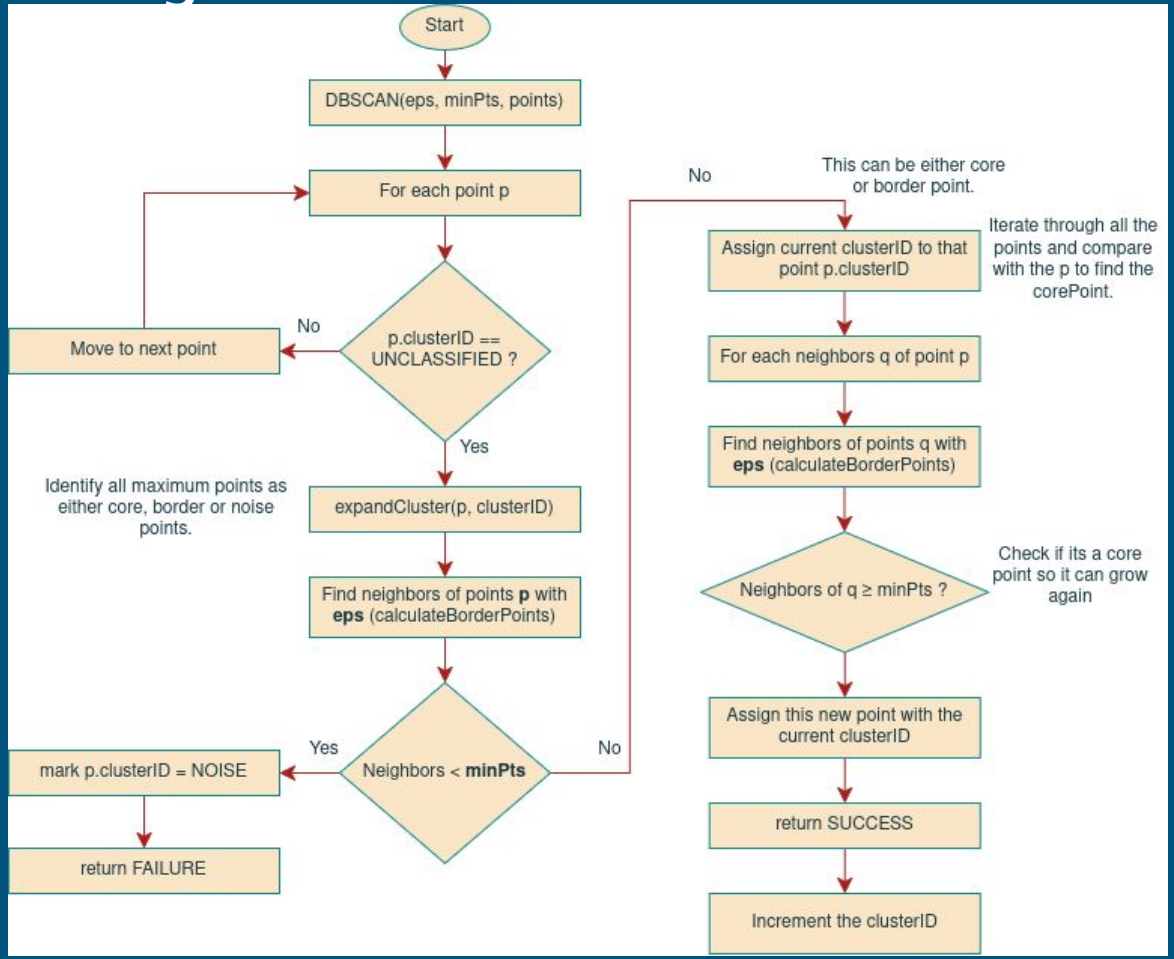
Artificial dataset was used to test clustering algorithm which can be obtained here. Following parameters were used:

1. Minimum number of points: 4
2. Epsilon: 0.75



Source: Hepta (Total number of cluster: 7)

DBSCAN Algorithm



DBSCAN: Complexity

Computational Complexity

O(n^2)

If spatial index used to store the objects (point clouds) it is
O(nlogn)

n: number of database objects.

Complexity [edit]

DBSCAN visits each point of the database, possibly multiple times (e.g., as candidates to different clusters). For practical considerations, however, the time complexity is mostly governed by the number of regionQuery invocations DBSCAN executes exactly one such query for each point, and if an indexing structure is used that executes a neighborhood query in $O(\log n)$, an overall average runtime complexity of $O(n \log n)$ is obtained (if parameter ε is chosen in a meaningful way, i.e. such that on average only $O(\log n)$ points are returned). Without the use of an accelerating index structure, or on degenerated data (e.g. all points within a distance less than ε), the worst case run time complexity remains $O(n^2)$. The $\binom{n}{2}$ - n = $(n^2-n)/2$-sized upper triangle of the distance matrix can be materialized to avoid distance recomputations, but this needs $O(n^2)$ memory, whereas a non-matrix based implementation of DBSCAN only needs $O(n)$ memory.

Kitti Data

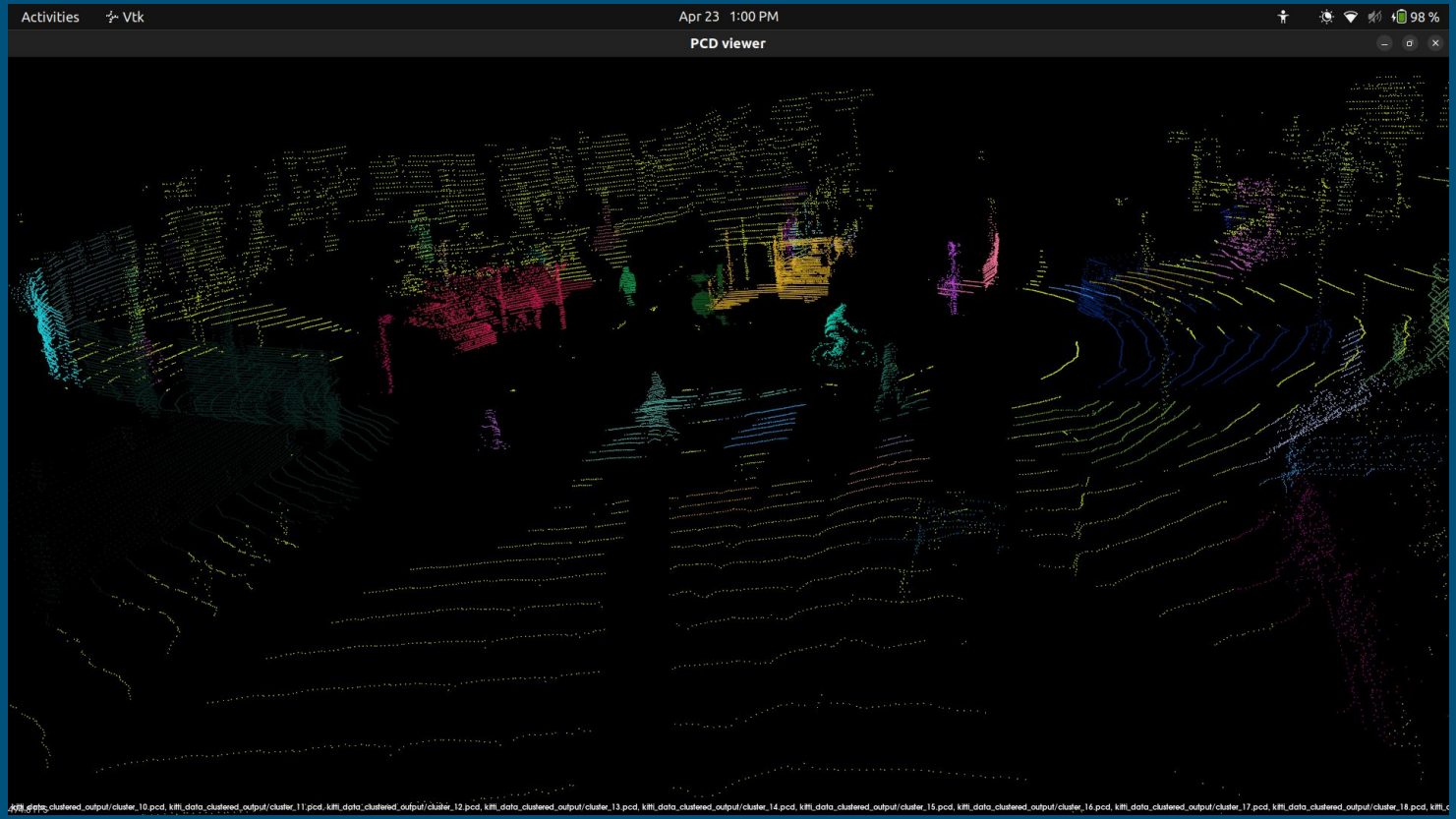
With dbscan

no use of kdtree

no preprocessing of the data

around 123000 points

took around 12-20 mins.



Our Environment Data

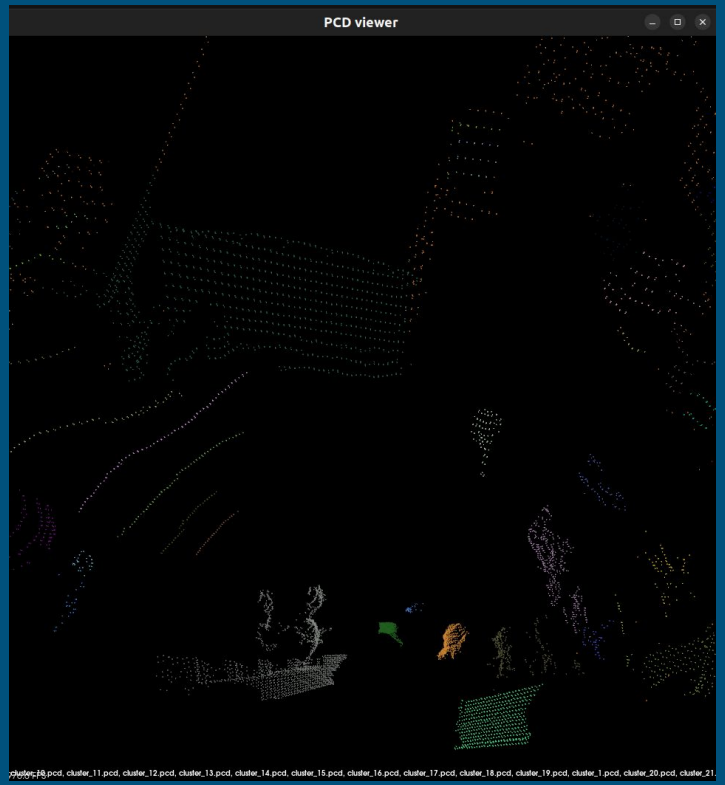
With dbscan

No use of kdtree

No preprocessing of the data

around 12000 points

took 8 second.



Preprocess the data

Voxel Downsampling
Cut the z data of the point cloud data for ransac ground plane segmentation
Do ransac segmentation for the roi frame
Remove the inliers i.e road
Add the outliers and the remaining z data

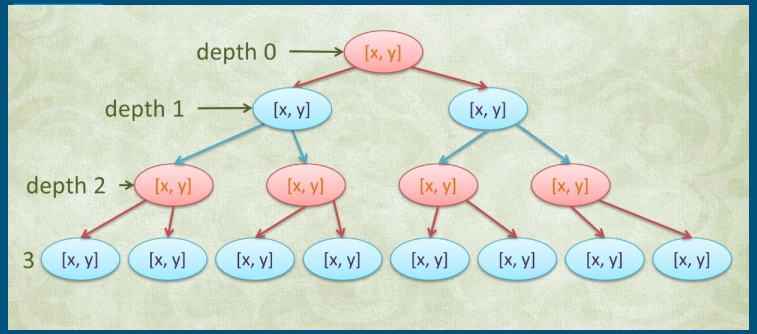
kitti_data.pcd reduced from 123000 -> 67000-75000 points by 45-50% points reduction.
logictronix_enc.pcd reduced from 12000 points to around 10500-11000 points.

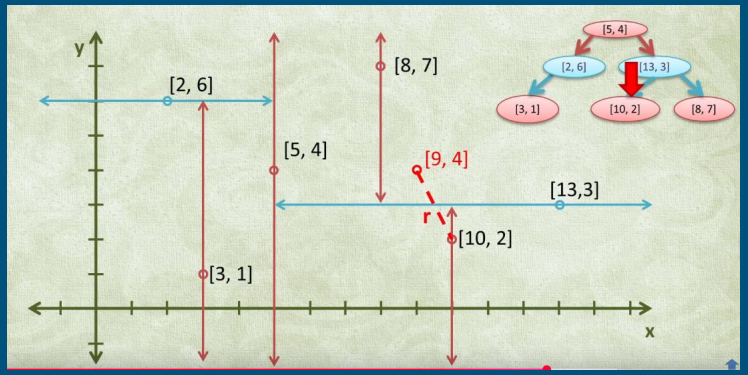
Time taken

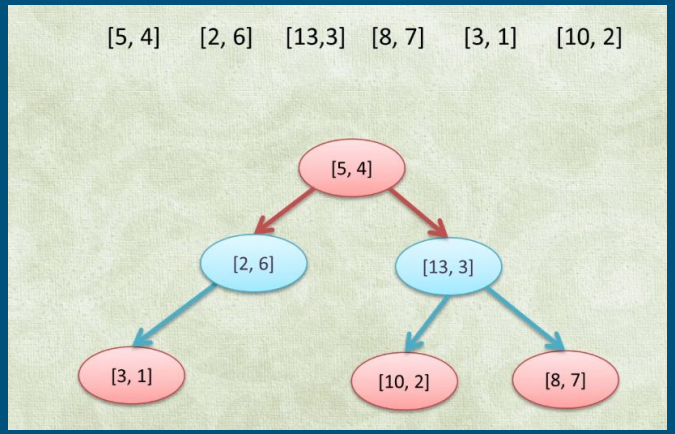
kitti_data_filtered.pcd clustering time took 4-7 mins reduced from 10-15 mins.

logictronix_env_filtered.pcd data clustering took 6-7 sec.

Kd-Tree

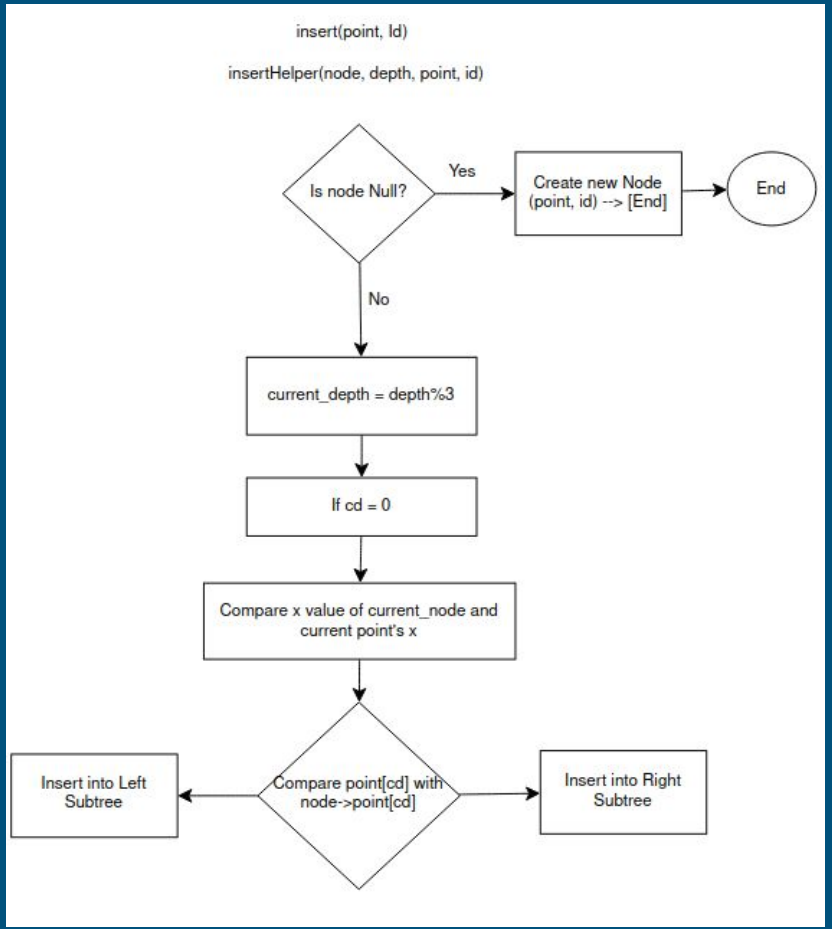




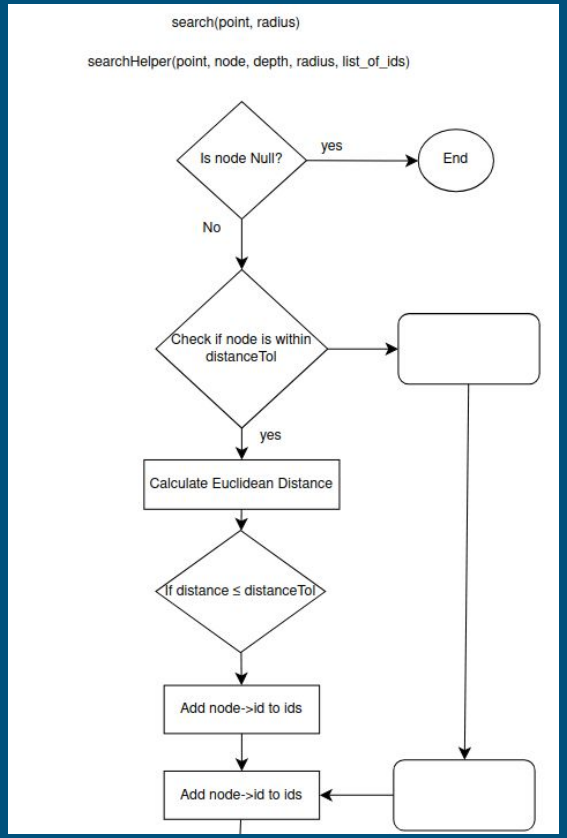


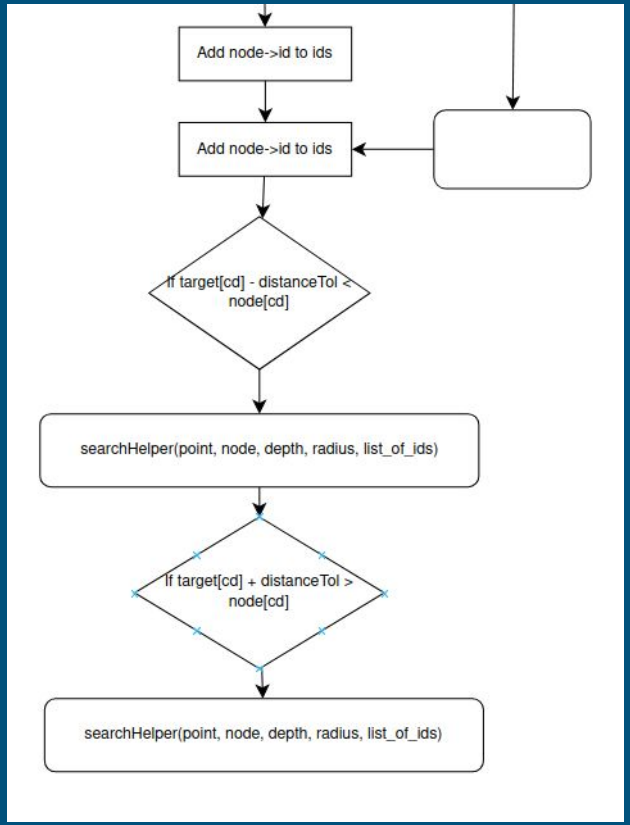
Optimized data structure for storing the points of the point cloud data(pcd) for easy search and retrieval

Kd-Tree Points Insertion

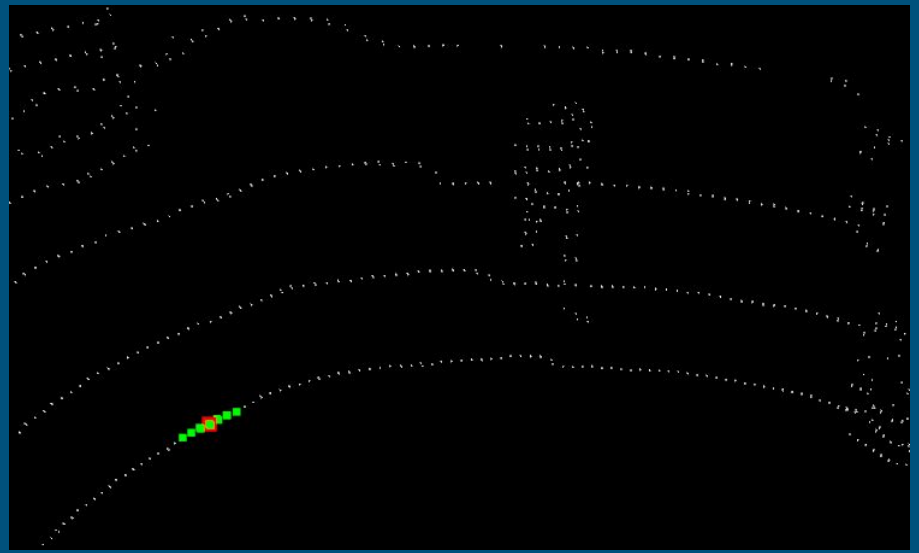


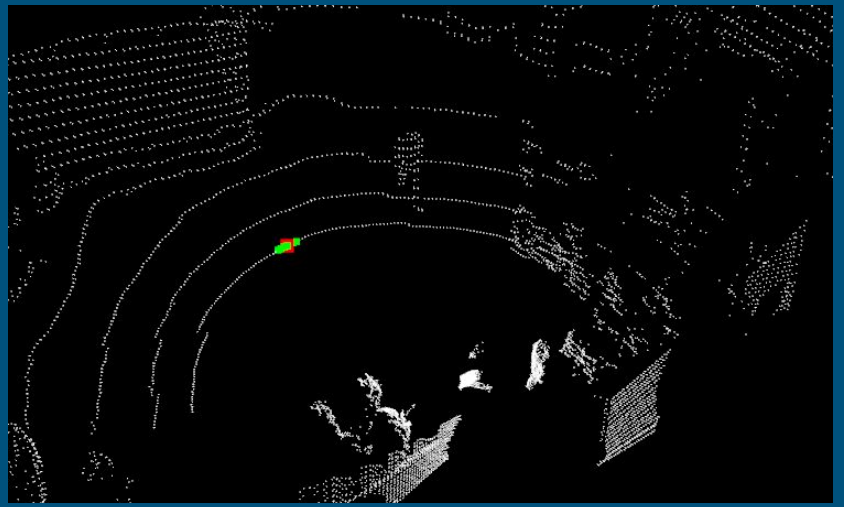
Radius Search in KdTree





Kd-Tree Implementation

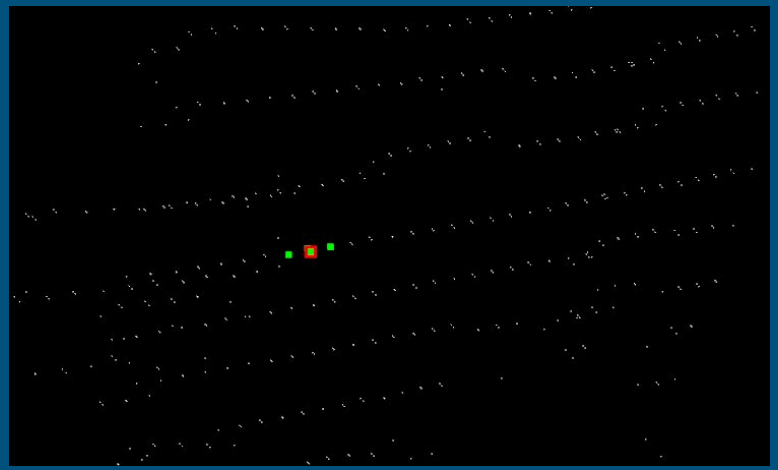


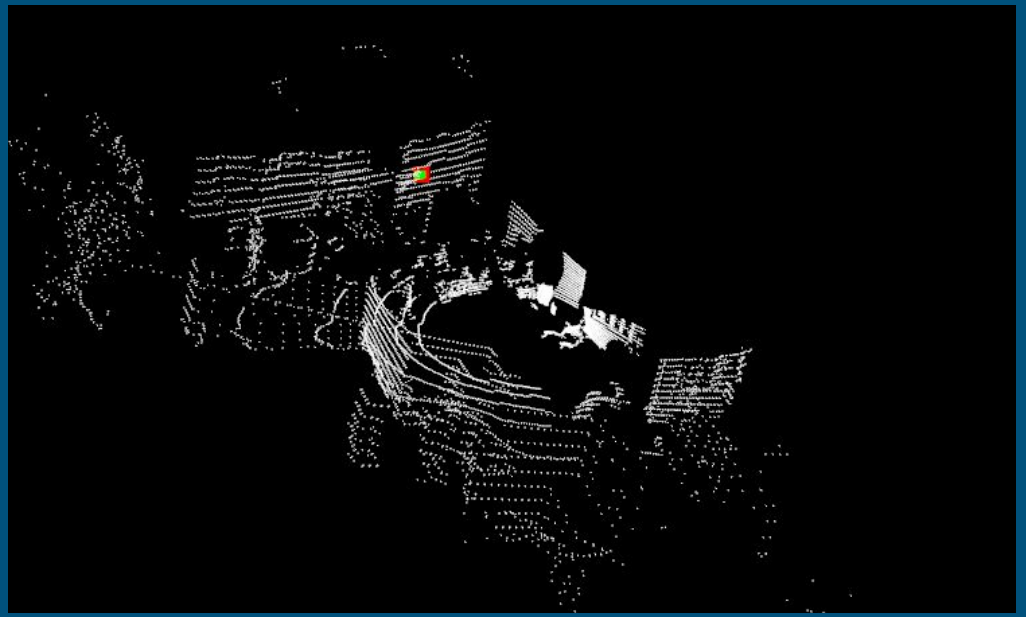


Select any 1 random point from the cloud

Searched point Red
Neighbor points Green

Kd-Tree Implementation





Select any 1 random point from the cloud

Visualization:
Search point in red
Neighbor point in green

Dbscan optimization using scratch Kd-Tree

Read the .pcd files

Convert the pcd file to std::vector<LidarPoint>
LidarPoint{
 double x,y,z;
int clusterID;
};

kitti_filtered_data.pcd took 3-6 second which was 4-7 mins before.

logictronix_env_filtered_data.pcd took 1-1.3 second which was 6-7 seconds before.

Dbscan optimization using pcl Kd-Tree

Read the .pcd files

kitti_filtered_data.pcd took 450-700ms.

logictronix_env_filtered_data.pcd took 500ms.

DBSCAN Clustering performance summary

- **Without Preprocessing, No KD-Tree**:
  - **KITTI data (~123,000 points)**:
    - Clustering time: **12–20 minutes**.
  - **Logictronix_env.data (~12,000 points)**:
    - Clustering time: **8 seconds**.

After Preprocessing:

- **KITTI data**:
  - Points reduced from **123,000 → 67,000–75,000** (~45–50% reduction).
- **Logictronix_env.data**:
  - Points reduced from **12,000 → 10,500–11,000**.

Clustering Time After Preprocessing (Still No KD-Tree):

- **KITTI data (filtered)**:
  - Clustering time reduced to **4–7 minutes** (from 10–15 minutes).
- **Logictronix_env.data (filtered)**:
  - Clustering time reduced to **6–7 seconds**.

After DBSCAN Optimization with Scratch KD-Tree Implementation:

- **KITTI filtered data**:
  - Clustering time: **3–6 seconds** (was 4–7 minutes).
- **Logictronix_env filtered data**:
  - Clustering time: **1–1.3 seconds** (was 6–7 seconds).

After DBSCAN Optimization with PCL KD-Tree:

- **KITTI filtered data**:
  - Clustering time: **450–700 ms**.
- **Logictronix_env filtered data**:
  - Clustering time: **~500 ms**.

DBSCAN Clustering further optimization steps?

1. Parallelize the operation
    a. For distance calculation
    b. For neighbor search operation

2. Cluster expansion steps avoiding race condition when updating clusterID

Clustering LiDar Data with K-Means and DBSCAN

2.2). Since the dataset is 3-dimensional, the suitable value for m is 6. Figure 4 illustrates how to select the appropriate ε, which for scenario 1 was found to be 0.7 since that is the value where the curvature of the graph occurs. From that point onwards, the slope of the graph becomes very high (near vertical).

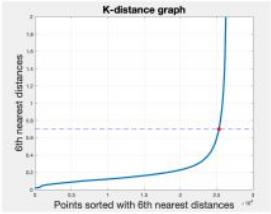


Figure 4: Selection of the best value for ε (0.7), with m=6 for scenario 1.

DBSCAN:

1. Suitable choice for the parameter minPts is 2 times the dimension of the data.
    a. Since the dataset is 3 dimensional, suitable value for m = 6.
2. For estimation of eps value: K distance graph for the data is designed where k is the parameter m.
3. Goal is to compute the distance from each point in the data to its m-th nearest point and then plot the sorted points in ascending order of their m-distance value against those distances.
4. Ideal value for eps will be the point of maximum curvature.
    a. For scenario 1 eps = 0.7 found since i.e the value where the curvature of the graph occurs.

2.2 DBSCAN Clustering

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) (Ester et al., 1996) is a data clustering algorithm that given a set of points in some space, groups together points based on a specified distance (e.g. Euclidean), that are close to each other. A cluster only forms when a minimum number of points (m), specified by the user, is assigned. In low-density regions, some data points do not have a neighborhood large enough to create a new cluster, so the algorithm marks them as outliers. This method is helpful whenever the dataset has arbitrary shapes and densities, so it can be useful for segmenting LiDAR data. The algorithm has two parameters: the minimum number of points that a cluster needs to be created (m) and a fixed radius for the search of neighbors (ε).

A suitable choice for the parameter m is 2 times the dimension of the data (Sander et al., 1998). For the estimation of the ε value, a k-distance graph for the data is designed where k is the parameter m (Ester et al., 1996). The goal is to compute the distance from each point in the data to its m-th nearest point and then plot the sorted points in ascending order of their m-distance value against those distances. The ideal value for ε will be the point of maximum curvature. The process is illustrated for a test scenario in section 5.1 (Figure 4).

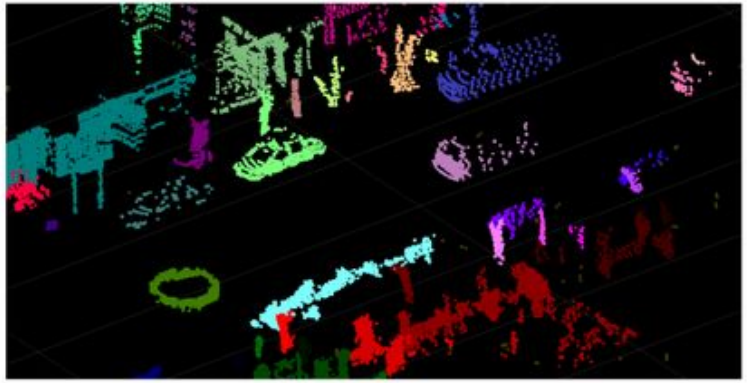


Figure 5: DBSCAN clusters for scenario 1.

Observing fig 5: method efficiently forms the clusters according to the objects observed by the Lidar.

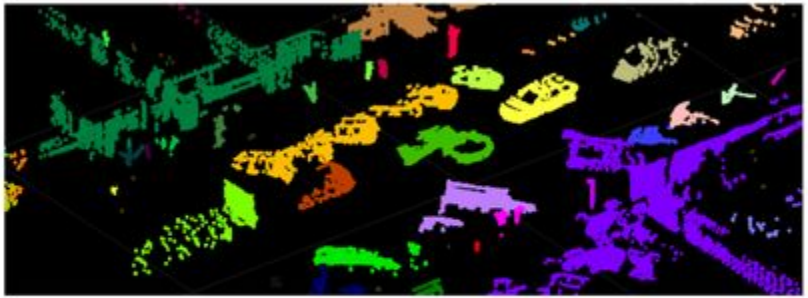
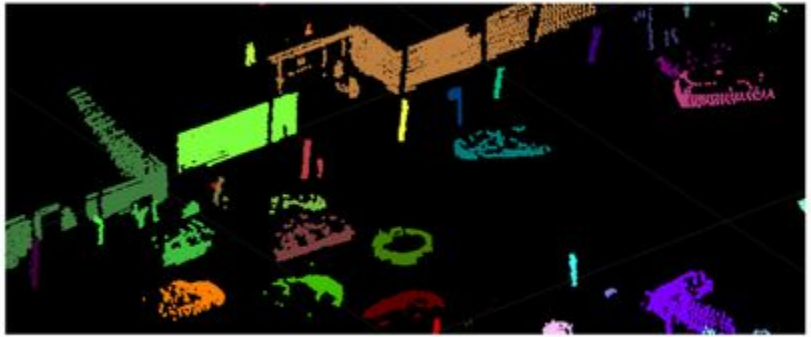


Figure 6: DBSCAN clusters for scenario 4.

Obstacles very close to each other were not always divided into separate groups.

dbscan grouped the parked cars in a single cluster when it should have separated the cluster accordingly to the number of parked vehicles.



Objects on the road are more distant from the sidewalk(such as scenario 2) the clusters were better established.

Figure 7: DBSCAN clusters for scenario 2.

Conclusion

1. The experiment with 5 H3D test scenarios leads to the conclusion that **DBSCAN performs better when the objects in the scene are rather distant**. **For objects very close to each other, DBSCAN clustered different objects into the same cluster.**
2. The ground removal procedure proved to be an essential requirement to separate the interest objects in all scenarios tested.
3. Analysis of H3D and BOSCH datasets led to the conclusion that DBSCAN and Kmeans are more useful in urban scenarios than in highway scenes since there are moore objects to cluster.
4. Clustering method are useful techniques for segmenting LiDar data.

Dbscan Parameters understanding & estimation

Minimum Samples ("MinPts")

There is no automatic way to determine the MinPts value for DBSCAN. Ultimately, the MinPts value should be set using domain knowledge and familiarity with the data set. From some research I've done, here are a few rules of thumb for selecting the MinPts value:

- The larger the data set, the larger the value of MinPts should be
- If the data set is noisier, choose a larger value of MinPts
- Generally, MinPts should be greater than or equal to the dimensionality of the data set
- For 2-dimensional data, use DBSCAN's default value of MinPts = 4 (Ester et al., 1996).
- If your data has more than 2 dimensions, choose MinPts = 2*dim, where dim= the dimensions of your data set (Sander et al., 1998).

Dbscan Parameters understanding & estimation

Epsilon

1. Calculate the avg distance between each point and its k nearest neighbors, where k = the minPts value you selected.
2. Average k-distances are then plotted in ascending order on a k distance graph.

3. Example:

Dimension of dataset:10
So minpts = 2*10 = 20

1. Used NearestNeighbors from Scikit-Learn,
2. calculate average distance between each point and its found 20th nearest neigbors.
3. define n_neighbors as minPts(20).
4. Sort distance values by ascending value and plot.
5. Sorted distances produces a k-distance elbow plot.
6. Choose eps from crook of the elbow.

Adaptive Eps(d)

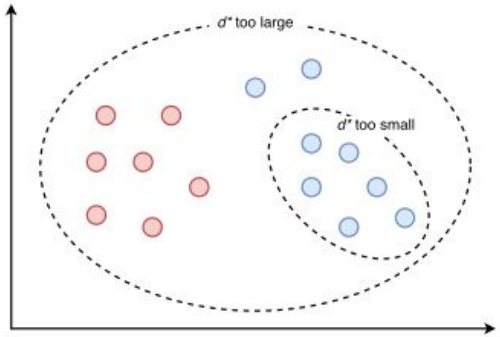


Fig. 3 Point cloud of two distinct objects (red and blue). Different values of d^* lead to different clustering results.

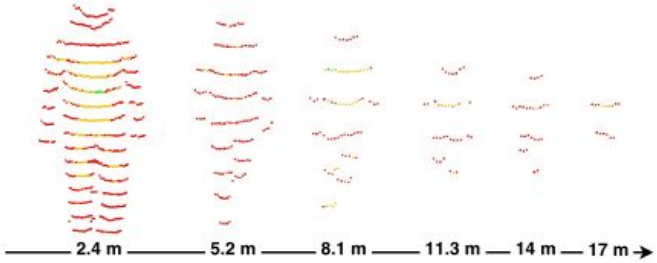


Fig. 4 Example of 3D LiDAR human point clouds at different distances. The further the person is from the sensor, the sparser is the relative point cloud.

be split into multiple clusters. If too large, multiple objects could be merged into a single cluster (see Fig. 3).

In our case, the shape of the point clouds formed by 3D LiDAR scans on the human body can vary significantly with the distance of the person from the sensor. In particular, due to its vertical angular resolution ($\Theta = 2°$), the vertical distance between two consecutive points can be very large, as shown in Fig. 4. We therefore adapt the threshold d^* to the actual scan range r as follows:

$$d^* = 2 \cdot r \cdot \tan\frac{\Theta}{2} \quad (2)$$

This equation simply computes the expected vertical distance between two adjacent scan channels for some straight vertical object.

d* = 2. r. tan(theta/2)

For velodyne vlp16 puck sensor, theta = vertical angular resolution = 2deg

1. Object a t 2.4 meters
    a. d = 2 * 2.4 * tan(2/2) = 8.38 cm
2. Object at 11.3 meters
    a. d = 2 * 11.3 * tan(2/2) = 39.5 cm

Farther the object is the less dense the point cloud becomes. This effects distance based clustering and detection of humans using 3D lidar.

In fig 3:

1. d too large:
    a. Under segmentation
    b. Algo treats both objects as one.
2. d too small
    a. Over segmentation
    b. Now even the blue object gets broken into multiple clusters.

Adaptive Eps(d)

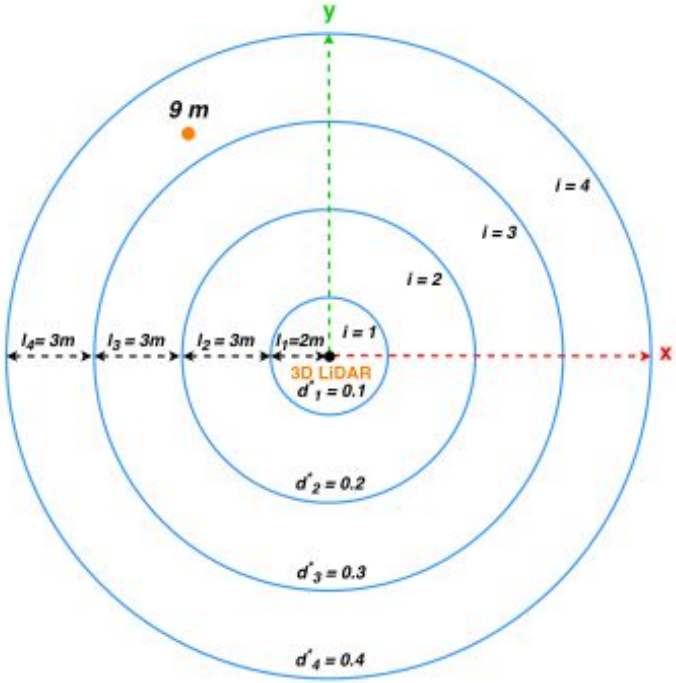


Fig. 5 Different values of d^* correspond to different nested regions. In this example, a cluster (red dot) at 9m from the sensor lies on the 4th circular region, where the clustering threshold is $d^*_4 = 0.4$m.

our application, we define circular regions 2-3m wide using $\Delta d = 0.1$m, which is a good resolution to detect potential human clusters.

Further Flow:

1. Apply bounding boxes around clusters
2. Fusion of Lidar and USB camera data for clustering, tracking
    a. Manipulation images using opencv library
    b. Collision detection system based on motion models, lidar, camera measurements.
    c. Feature Tracking
        i. Detect features from images to track objects over time using binary descriptors
    d. Projection of 3D lidar points into camera frame data
    e. Using DL to detect vehicles in camera images
        i. CNN, yolo, RNN, transformer vision models(attention mechanism)
    f. Creating a 3D object from lidar and camera data
3. SFA3D like neural network, Point Pillar Models for ML based implementation to generate 3D object detection or segmentation
4. HIL & SIL in ADAS with ROS
5. Kalman Filter
6. Vehicle Kinematics
7. Image Segmentation
    a. Find contours, and edges of multiple objects in an image
    b. Background subtraction for video

Kd-Tree Implementation in RANSAC

```
// find the distance for other points from the line
for (int index = 0; index < cloud->points.size(); index++)
{
    if (inliers.count(index) > 0)
    {
        continue;
    }

    pcl::PointXYZI point = cloud->points[index];

    float x3, y3, distance;
    x3 = point.x;
    y3 = point.y;

    distance = fabs(a * x3 + b * y3 + c) / sqrt(a * a + b * b);

    if (distance <= distanceTol)
    {
        inliers.insert(index);
    }
}

// record the line with highest number of inliers and select the data
if (inliers.size() > inliersResult.size())
{
    inliersResult = inliers;
}
```

```
for (int idx = 0; idx < cloud->points.size(); idx++)
{
    pcl::PointXYZI p = cloud->points[idx];
    std::vector<int> nearby_points1 = tree->search(p, 0.2);

    for (int i : nearby_points1)
    {
        pcl::PointXYZI pt = cloud->points[i];
        float dist = fabs(A * pt.x + B * pt.y + C * pt.z + D) / norm;
        if (dist <= distanceTol)
        {
            inliers.insert(idx);
        }
        // inliers.insert(idx);
    }
}
```

Continued

PointCloud Reduction

Thank You!