KITTI 3D Point Cloud: Ground Filtering & Objects Clustering

0. Main Procedures

(1) Use RANSAC to fit the ground.

(2) Remove ground and down-sample the non-ground point cloud

(3) cluster the non-ground points by using DBSCAN

1. Ground Filtering

RANSAC is used to detect the ground. There are four tunable parameters used in RANSAC, which are the number of randomly selected samples, the number of iterations, the threshold of the distance between two points, the probability of non-ground points over all the points. In each iteration, we will perform the following procedures:

(1) From all the data points, randomly select three samples.
(2) Get the normal of the plane consisting of above three samples.
(3) Calculate the distances of all points to the above plane.
(4) Based on distance threshold and above distances result, we can pick the points that are smaller than the distance threshold, which means that these points are probably inside the selected plane. They are the possible ground points as the ground is a large plane consisting of lots of in-plane points.
(5) Through iteratively update the maximum number of ground points, we can gradually divide all the points into two parts: ground points and non-ground points.
(6) If the percentage of ground points is larger than the probability we set, then we break the loop.

```
"""
    Function: Remove ground from point cloud dataset

    Input:
            data: point cloud, one frame, N*3
            sample: the number of randomly selected samples
            max_iter: maximum iterations
            dist_thre: distance threshold, used to determine whether a data point belongs to inlier
            prob: the probability of non-ground data points over all the data points

    return:
            non_ground_point_idx: indices of non-ground data points
            ground_point_idx: indices of ground data points

"""
def ground_segmentation(data, sample=3, max_iter=100, dist_thre=0.35, prob=0.75):
    N, D = data.shape # N: the number of sample, D: dimension
    num_ground = 0 # the number of ground data points
    max_num_ground = 0 # the maximum number of ground data points
    non_ground_points = []
    ground_points = []
```

```
""" ******************** Using RANSAC to detect the ground  ******************** """
for i in range(max_iter) :
    # ----------------- Step 1: Randomly Select Samples (Plane detection: 3 samples selected) --------
    selected_samples = random.sample(data.tolist(), sample)
    selected_samples = np.array(selected_samples)

    # ----------------- Step 2: Get Plane Coefficients -----------------
    normal = estimate_normal(selected_samples, normalize=True).reshape(-1, 1)

    # ----------------- Step 3: Compute the distances of all the points to the plane -----------------
    distances = np.abs(np.dot(data, normal[:3]) + normal[3])

    # ----------------- Step 4: Compute the number of points in this plane -----------------
    points_idx_in_plane = [idx for idx in range(len(distances)) if distances[idx] < dist_thre]
    num_ground = len(points_idx_in_plane)  # update the number of ground points[i] < dist_thre]

    # ----------------- Step 5: Select the plane with most inlier points -----------------
    if num_ground > max_num_ground:
        max_num_ground = num_ground
        ground_points = points_idx_in_plane
        non_ground_points = [idx for idx in range(len(distances)) if distances[idx] >= dist_thre]

    # ----------------- Step 6: Stop iteration if the number of ground points reaches the prob -------
    if (max_num_ground / N) > (1 - prob):
        break

print('origin data points num:', N)
print('segmented data points num (non-ground):', len(non_ground_points))
return non_ground_points, ground_points
```

The purpose of computing normal of a plane is to find out the final plane of the ground. Thus, we start with three randomly selected samples and fit a plane and get corresponding plane equation/normal. At some probability, we will select three points on the ground. In this case, lots of ground points can be identified by comparing their distances to the fitting plane. Gradually, we will find out a plane with largest number of ground points and determine the ground. Therefore, in this part, based on the idea of three points forming a plane, we can calculate the plane equation for the normal estimation.

```
"""
    Function: Normal Estimation. Three points in the plane are known.
              Compute the plane equation and return the coefficient.

    Input:
          points: data points, 3*3 array
          normalize: bool
    return:
          coefficient: 1*4 array
"""
def estimate_normal(points, normalize=True):
    """
        plane: ax + by + cz + d = 0
        Normal:
            a = (y2 - y1) * (z3 - z1) - (y3 - y1) * (z2 - z1)
            b = (z2 - z1) * (x3 - x1) - (z3 - z1) * (x2 - x1)
            c = (x2 - x1) * (y3 - y1) - (x3 - x1) * (y2 - y1)
            d = -a * x1 - b * y1 - c * z1
    """
    p21 = points[1, :] - points[0, :]
    p31 = points[2, :] - points[0, :]
    a = (p21[1] * p31[2]) - (p31[1] * p21[2])
    b = (p21[2] * p31[0]) - (p31[2] * p21[0])
    c = (p21[0] * p31[1]) - (p31[0] * p21[1])

    if normalize:
        r = math.sqrt(a ** 2 + b ** 2 + c ** 2)
        a /= r
        b /= r
        c /= r

    d = -a * points[0, 0] - b * points[0, 1] - c * points[0, 2]

    return np.array([a, b, c, d])
```

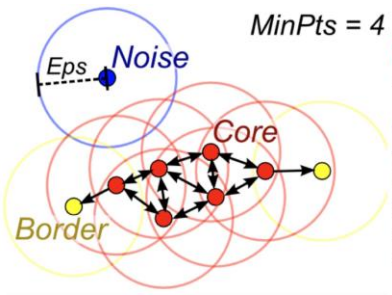
2. DBSCAN (non-ground point clouds clustering)

DBSCAN is short for Density-based Spatial Clustering of Application with Noise. The reason we select DBSCAN instead of K-Means, GMM, Spatial Clustering, Mean Shift, is because (1) DBSCAN can determine the number of clusters automatically; (2) it is robust to outliers (noise); (3) the cluster is determined by point density or Euclidean distance; (4) time complexity is low compared to others (O(NlogN)). Below is the procedures and representation of DBSCAN. However, if there exists a tiny connection between two clusters, then these two clusters will be identified as one cluster. This is the reason why we want to remove the ground point cloud first and later using DBSCAN as we can cut off the connection between different objects on the ground and the ground itself. Thus, after removing the ground, the other objects can be regarded as different separate objects. This will also decrease the probability of combining two clusters with small connection to a bigger cluster.

Preparation: all points labeled as unvisited

Parameters: distance r, min_samples

1. Randomly select a unvisited point p, find its neighborhood within r
2. Number of points within $r \geq \min\_samples$?
   - Yes. p is a *core point*, Create a cluster C, go to step 3, mark p as *visited*.
   - No. Mark p as *noise* and *visited*.
3. Go through points within its r-neighborhood, label it as C
   - If it is a *core point*, set it as the "new p", repeat step-3
4. Remove cluster C from the database, go to step-1
5. Terminate when all points are visited.



Red: Core points. point number within circle ≥ 4

Yellow: Border points. Still part of the cluster because it is within r of a core point, but does not meet the min_points criteria

Blue: Noise point. Not assigned to a cluster.

How to select core points and their corresponding neighbors?

For selecting the nearest neighbors, we can use brute force search or a faster KDTree search. For the brute force searching for the neighbors, we can simply compute the distances between a specific point and all the data points. Then we can include all the points that have distances smaller than the setting search range (radius). After considering neighbors for each point in the dataset, we will pick some points that meets the specific requirements as the core points. Here, we can set the core points requirement with the constraint of minimum number of samples. As long as the number of neighbors of a point is larger than this minimum number of samples, then we can specify it as the core point and store it. The tunable parameters in this method are radius and minimum number of samples.

```
"""
    Function: Search for core points from point clouds.
              Constraints: (1) in the range of radius; (2) the number of points >= min_num_samples
              Euclidean space can be used to search for the points. kdtree can also be used for faster search.

    Input:
            data: point clouds
            radius: search range
            min_num_samples: the minimum number of samples
            method: searching strategy
    Return:
            core_points(set): set of core point indices
            neighbor_points(dict): nearest neighbor points around the core point (include core point)
"""
def getCore(data, radius, min_num_samples, method='kdtree'):
```

```
"""
    Function: Get the neighbor points of a specific point in the range of radius based on the dataset
    Input:
            point: target point, 1*3
            data: point clouds, N*3
            radius: search range
    Return:
            point_to_neighbors: indices of neighbor points
"""
def getNeighbors(point, data, radius):
    distances = np.sum((point - data) ** 2, axis=1) # distances square of all points to the center point
    neighbor_points = [idx for idx in range(distances.shape[0]) if distances[idx] < radius * radius]
    return neighbor_points

core_points = set()
point_to_neighbors = {}

if method == 'euclidean':
    for i in range(data.shape[0]):
        neighbors = getNeighbors(data[i], data, radius)
        if len(neighbors) >= min_num_samples:
            core_points.add(i)
            point_to_neighbors[i] = neighbors
```

However, if the amount of data points is huge, the time complexity of this brute force will definitely high. Thus, we can use KDTree search to make the search process faster. The tunable parameter in this KDTree is the leaf size, radius, minimum number of samples.

```
if method == 'kdtree':
    kdtree = KDTree(data, leaf_size=1) # leaf_size can be adjusted
    neighbors = kdtree.query_radius(data, radius) # indices of nearest neighbors in the range of radius
    for i in range(neighbors.shape[0]):
        if len(neighbors[i]) >= min_num_samples:
            core_points.add(i)
            point_to_neighbors[i] = neighbors[i]

return core_points, point_to_neighbors
```

DBSCAN implementation:

```
"""
    Function: Cluster the point clouds using DBSCAN.
    Input:
            data: point clouds (non-ground points removed)
            radius: search range
            min_num_samples: the minimum number of samples around the core points
    return:
            clusters_index: cluster labels of all non-ground points, N*1, N does not include ground points.

"""
def clustering(data, radius=0.4, min_num_samples=5):
    N, _ = data.shape

    # --------------------- Step 1: Initialization ---------------------
    core_points = set()
    point_to_neighbors = {}
    cluster = set()
    clusters_index = np.zeros(N, dtype=int)
    # noise = []
    points_not_visit = set(range(N))
    k = 0 # the kth cluster
```

```
# --------------------- Step 2: Search for all the core points ---------------------
core_points, point_to_neighbors = getCore(data, radius, min_num_samples, method='kdtree')

# traverse all the core points
while len(core_points):
    points_old = points_not_visit
    idx = np.random.randint(0, len(core_points)) # randomly select a core point index
    core_point = list(core_points)[idx]
    points_not_visit = points_not_visit - {core_point} # delete current core point from unvisited points set
```

```
    # --------------------- Step 3: BFS ---------------------
    queue = []
    queue.append(core_point)
    while len(queue):
        cur_point = queue[0]
        if cur_point in core_points:
            neighbors = set(point_to_neighbors[cur_point]) # find the neighbors of current core point
            neighbors = neighbors & points_not_visit # exclude the core point in neighbors set
            queue += list(neighbors) # add the neighbors to the queue
            points_not_visit = points_not_visit - neighbors # mark the neighbors as visited points
        queue.remove(cur_point)

    # --------------------- Step 4: Get clusters ---------------------
    cluster = points_old - points_not_visit
    core_points = core_points - cluster
    k += 1
    clusters_index[list(cluster)] = k

return clusters_index
```

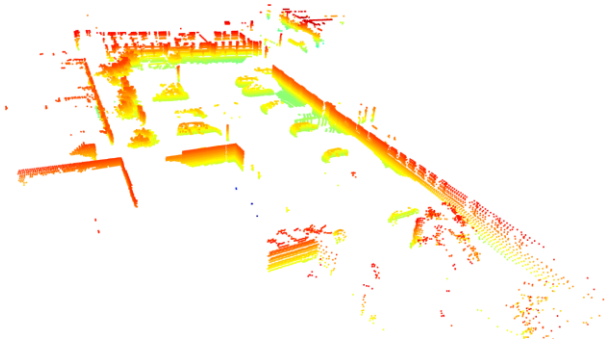
3. Results and analysis

	007476.bin	007477.bin	007478.bin	007479.bin
Raw point cloud	119356	115030	120796	123658
Non-ground	60534	58287	46765	60439
Non-ground (filtered)	21463	18437	18197	28248
Clustering time	70.03s	67.94s	42.49	66.45
Clustering time (filtered)	7.92s	5.30s	3.94	13.96

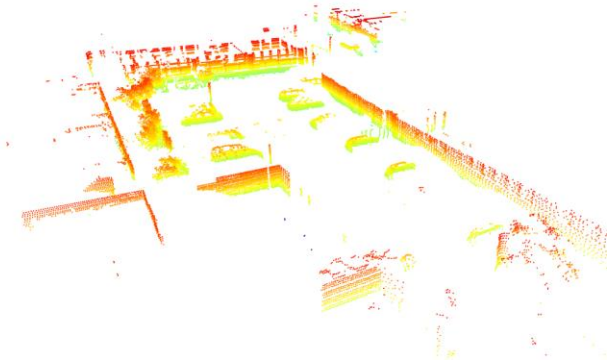
As we can see in the above table, each frame of point cloud has a huge amount of data points. If we apply the clustering in the raw data points, the efficiency is low although the results are good. Therefore, in this experiment, RANSAC is first used to detect the ground so that the ground can be removed before applying clustering algorithm. After removing the ground data points, the non-ground data points are also filtered by using voxel filtering method, which can reduce the size of data points and enhance the efficiency. After we get the filtered non-ground data points, we can apply DBSCAN algorithm to cluster the remaining data points. In DBSCAN, we used KDTree to search for all the core points instead of using Euclidean distance as the matrix will be huge and hence increase the computation cost. As shown in the following results, the filtering does not exert obvious effect on the clustering results but increases the clustering speed.

(1) 007476.bin

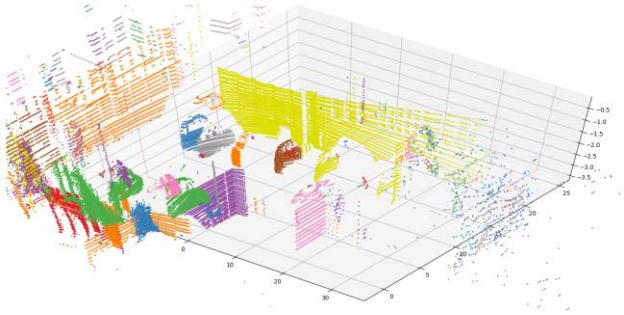
Raw point cloud Ground point cloud



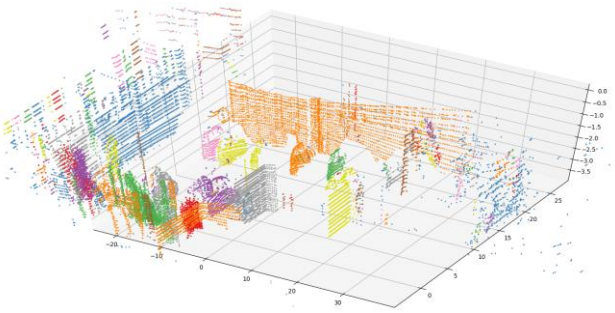
Non-ground point cloud



Non-ground point cloud after voxel filtering

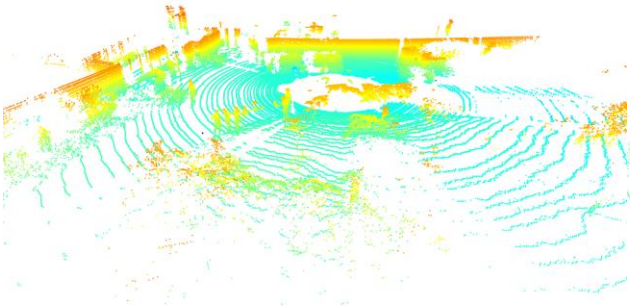


Clusters over non-ground points

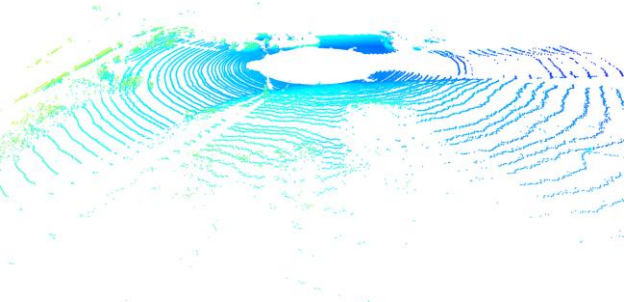


Clusters over non-ground points after filtering

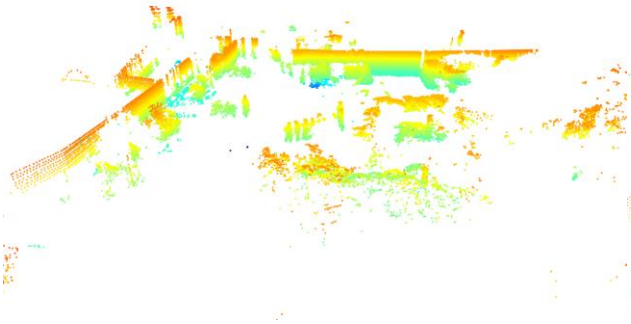
(2) 007477.bin



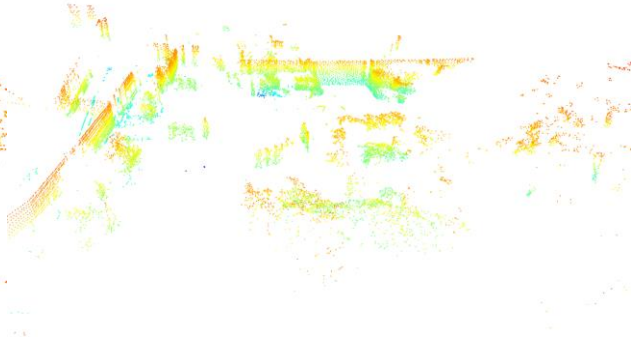
Raw point cloud



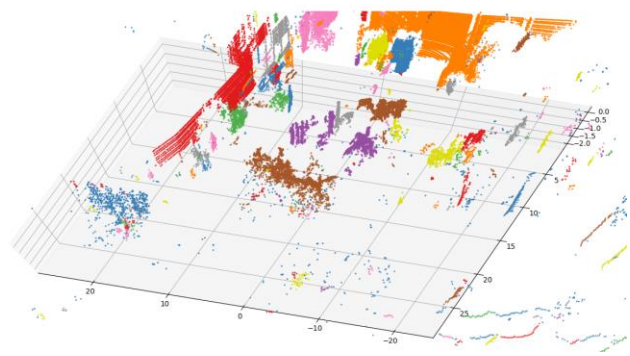
Ground point cloud



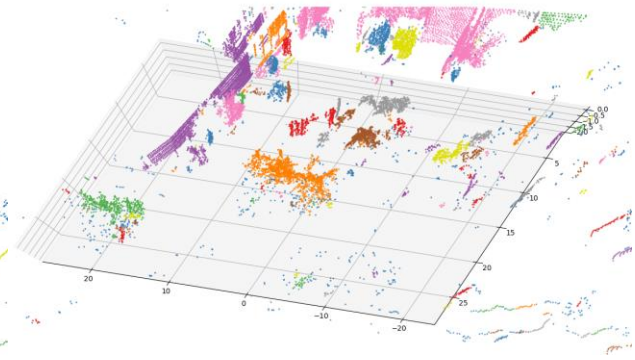
Non-ground point cloud



Non-ground point cloud after voxel filtering

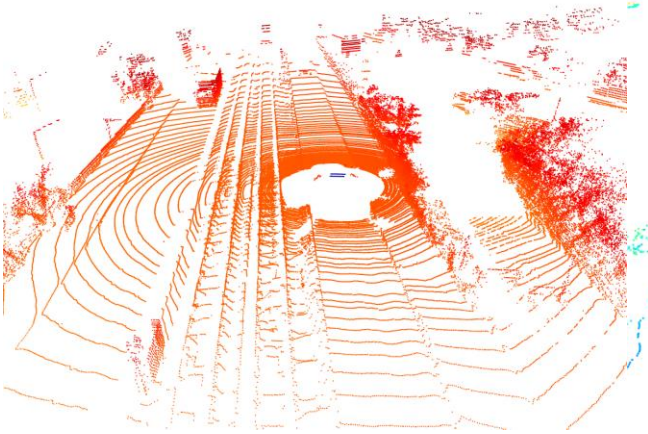


Clusters over non-ground points

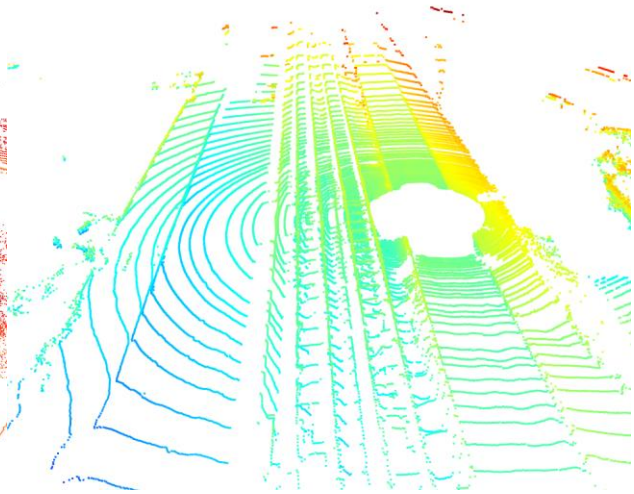


Clusters over non-ground points after filtering

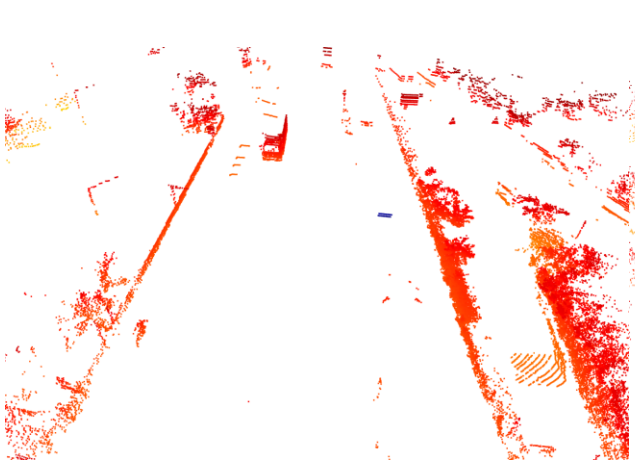
(3) 007478.bin



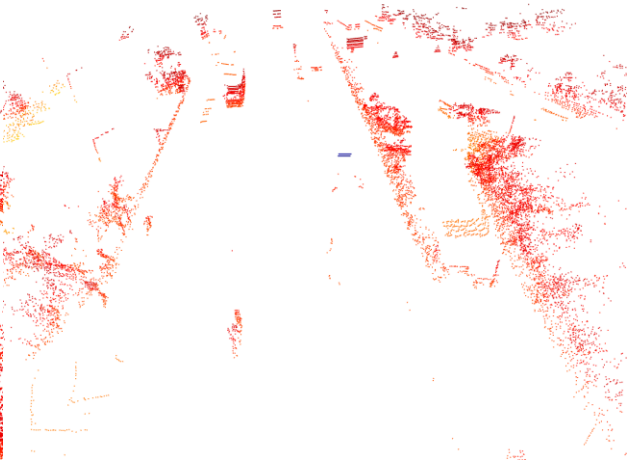
Raw point cloud



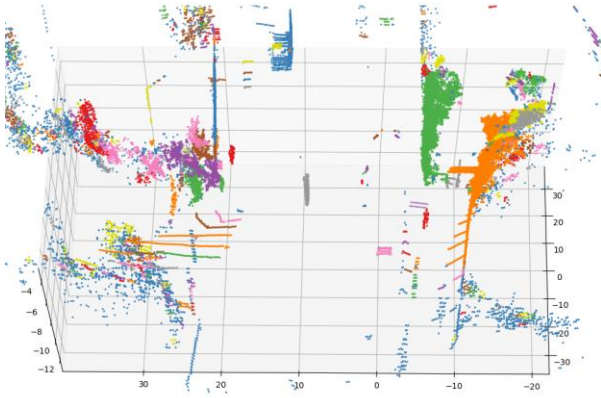
Ground point cloud



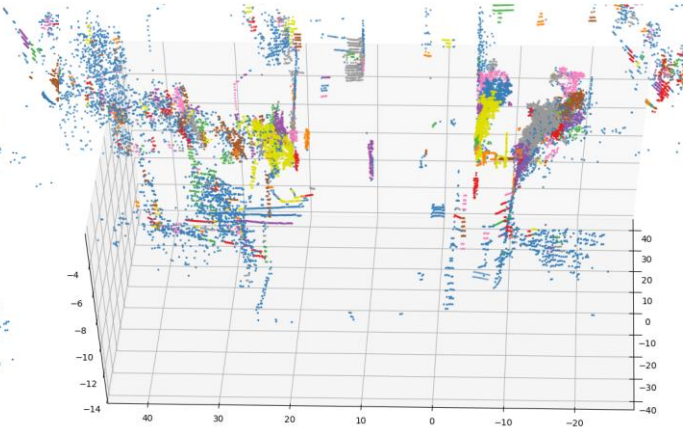
Non-ground point cloud



Non-ground point cloud after voxel filtering

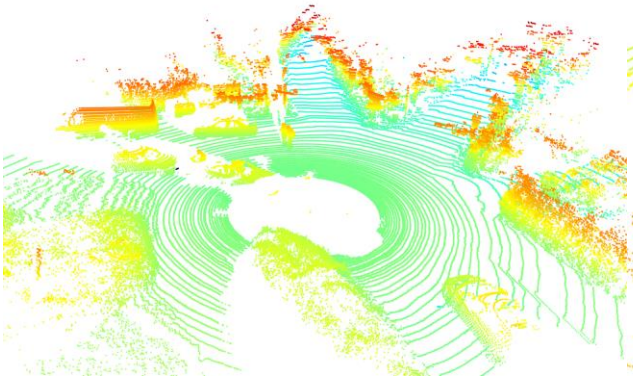


Clusters over non-ground points

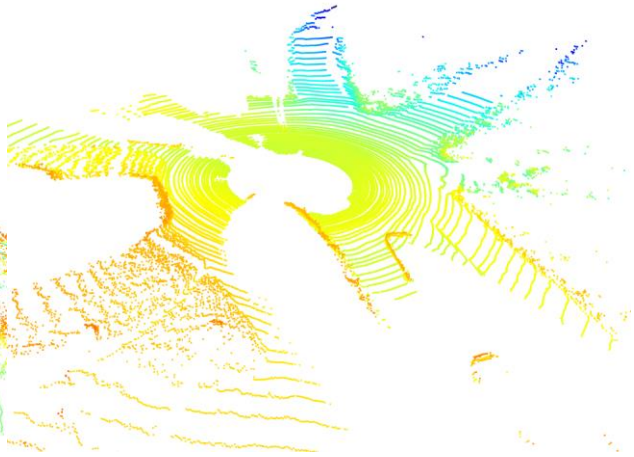


Clusters over non-ground points after filtering

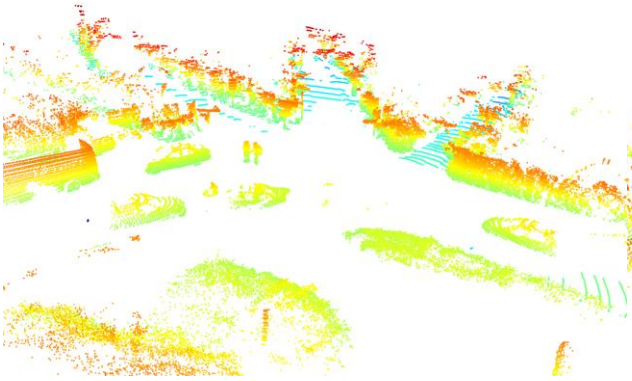
(4) 007479.bin



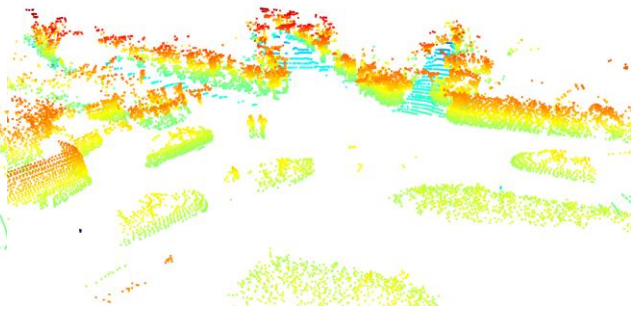
Raw point cloud



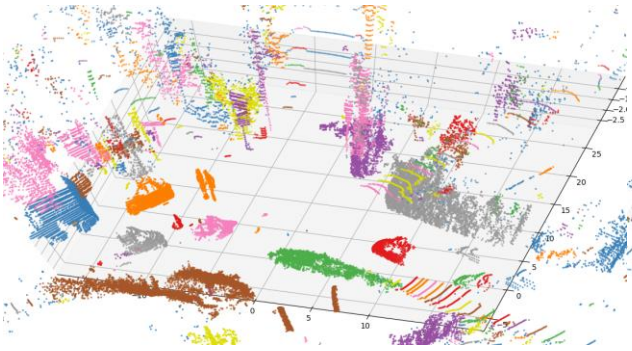
Ground point cloud



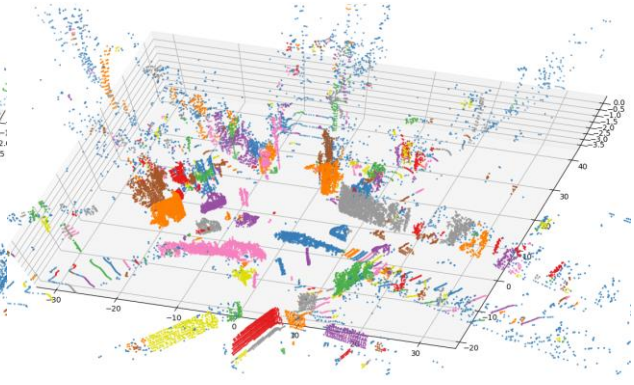
Non-ground point cloud



Non-ground point cloud after voxel filtering



Clusters over non-ground points



Clusters over non-ground points after filtering

4. Challenges

(1) Current algorithm cannot well detect the slopes on the road. If we use a large distance threshold, then we cannot detect the ground effectively. A better idea is to detect the ground in different sections, but this will definitely increase the computation cost.

(2) How to balance the computation cost and performance. If we set large parameters, it can decrease the false detection rate. However, it increases the computation cost as well. If we use larger leaf size in KDTree, we can reduce the computation cost but it will filter more data points.

(3) Based on the characteristics of different dataset, the fixed parameters cannot perform well in all the dataset.