Lab 2 – 3D Obstacle Detection from Point Clouds

Name: Sushant Gautam; NetId: sg2223

Objective:

- To understand about the 3D point cloud.
- To perform obstacle detection in the point cloud data.
- To understand the Euclidean clustering algorithm.

Question1 (20 pts):

Implement the function filter_ground to remove points on the ground surface from the rest of the point cloud. This can be done by setting a height threshold and filtering out points with Z coordinate below the specified height threshold (e.g. 1 meter). Note that the Z-coordinate of the point cloud is relative to the center of the laser scanner.

```
# TODO: implementation function to filter out points with Z value below a specified threshold
def filter_ground(cloud, ground_level=0):
    mask = cloud[:,2] >= ground_level
    filtered_pts = cloud[mask]
    return filtered_pts
```

Output: See the video attached herewith.

Question 2(20 pts):

Implement the function filter_by_distance to remove points that are too far away so that we only consider obstacles that are close to the vehicle. This can be done by setting a horizontal distance threshold and filtering out points with horizontal distance greater than the specified distance threshold (e.g. 10 meters). The horizontal distance of each 3D point can be calculated using the X-coordinate and Y-coordinate of the 3D point. Note that the X-coordinate and Y- coordinate of the point cloud is relative to the center of the laser scanner.

```
# TODO: implementation function to filter out points further than a specified distance
def filter_by_distance(cloud, distance=10):
    # 4 dimension in the bag files; ignore last dimension
    # Reference point (origin in this case)
    ref = np.array([0, 0, 0, 0])
    # Calculate distance of each point from reference
    dist = np.linalg.norm(cloud - ref, axis=1)


    # Create mask for points within distance threshold
    mask = dist <= distance

    # Filter points
    filtered_pts = cloud[mask]
    return filtered_pts
```

Output: See the video attached herewith.

Question 3(40 pts):

Implement the function euclidean_clustering to organize the point cloud into clusters of points that correspond to individual obstacles. The Euclidean clustering algorithm works by iteratively selecting points and grouping them together with neighboring points that are within a specified Euclidean distance to form clusters. The clustering distance threshold (e.g. 0.5 meters) controls the size and number of output clusters. The function should output a list of integer cluster labels ranging from 1 to N where N is the total number of clusters. That is, the list should have a value of 1 for all the points from cluster #1 and a value of 2 for all the points from cluster #2 etc. The original pseudocode for Euclidean Clustering is given below:

```
# TODO: implementation function to perform Euclidean clustering at a specified threshold in meters
def euclidean_clustering(cloud, threshold=0.5):
    """
    Perform Euclidean clustering on 3D points within distance threshold
    """
    num_points = len(cloud)
    cluster_labels = np.zeros(num_points, dtype=int)
    cluster_id = 1  # Start with cluster ID 1

    # Use a KDTree for efficient neighbor search within the threshold
    kdtree = KDTree(cloud)
    for i in range(num_points):
        if cluster_labels[i] != 0:
            continue

        queue = [i]
        cluster_labels[i] = cluster_id

        while queue:
            point_idx = queue.pop(0)


            neighbors = kdtree.query_radius([cloud[point_idx]], r=threshold)[0]

            for neighbor_idx in neighbors:
                if cluster_labels[neighbor_idx] == 0:
                    queue.append(neighbor_idx)
                    cluster_labels[neighbor_idx] = cluster_id

        cluster_id += 1

    return cluster_labels

```

- I use scikit-learn kdTree to find the neighbors. It uses tree structure to find the neighbors for the Euclidean clustering. Other steps are the same as mentioned in the algorithm pseudocode in the lab guide.

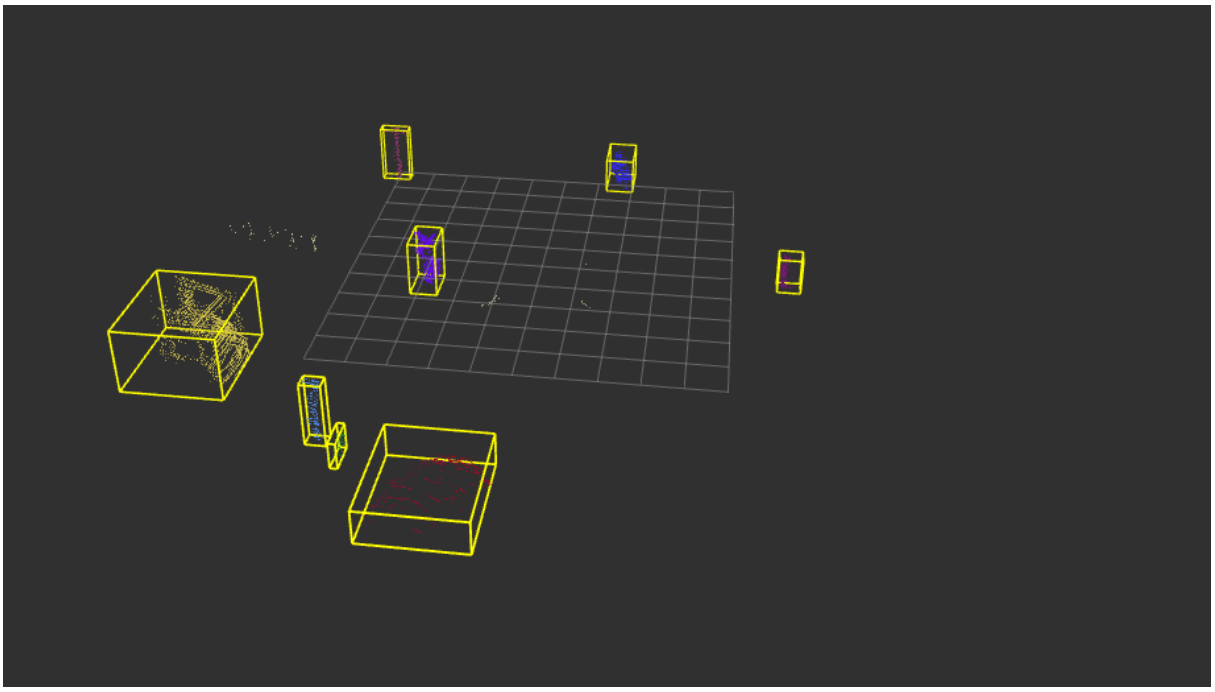
Output: See in the attached video herewith.

Question 4 (20 pts):

Implement the function get_bounding_boxes to compute the coordinates of the boundingbox around each detected obstacle from the previous step. Each obstacle should have acorresponding bounding box and each bounding box should be described by 8 points. The width of the bounding box should range from the minimum to maximum X coordinates in the corresponding point cloud cluster. The length of the bounding box should range from the minimum to maximum Y coordinates in the corresponding point cloud cluster. The height of the bounding box should range from the minimum to maximum Z coordinates in the corresponding point cloud cluster. The computed bounding boxes will automatically be visualized in the RVIZ window.

```python
# TODO: implementation function to compute bounding boxes from cluster labels
def get_bounding_boxes(cloud, cluster_labels):
    bounding_boxes = []
    unique_clusters = np.unique(cluster_labels)

    for cluster_id in unique_clusters:
        if cluster_id ==0: # 0 is not the label
            continue
        cluster_points = cloud[cluster_labels == cluster_id]
        min_coords = np.min(cluster_points, axis=0)
        max_coords = np.max(cluster_points, axis=0)

        corners = np.array([
            [min_coords[0], min_coords[1], min_coords[2]],
            [max_coords[0], min_coords[1], min_coords[2]],
            [min_coords[0], max_coords[1], min_coords[2]],
            [max_coords[0], max_coords[1], min_coords[2]],
            [min_coords[0], min_coords[1], max_coords[2]],
            [max_coords[0], min_coords[1], max_coords[2]],
            [min_coords[0], max_coords[1], max_coords[2]],
            [max_coords[0], max_coords[1], max_coords[2]]
        ])

        bounding_boxes.append(corners)


    return bounding_boxes
```

Output:



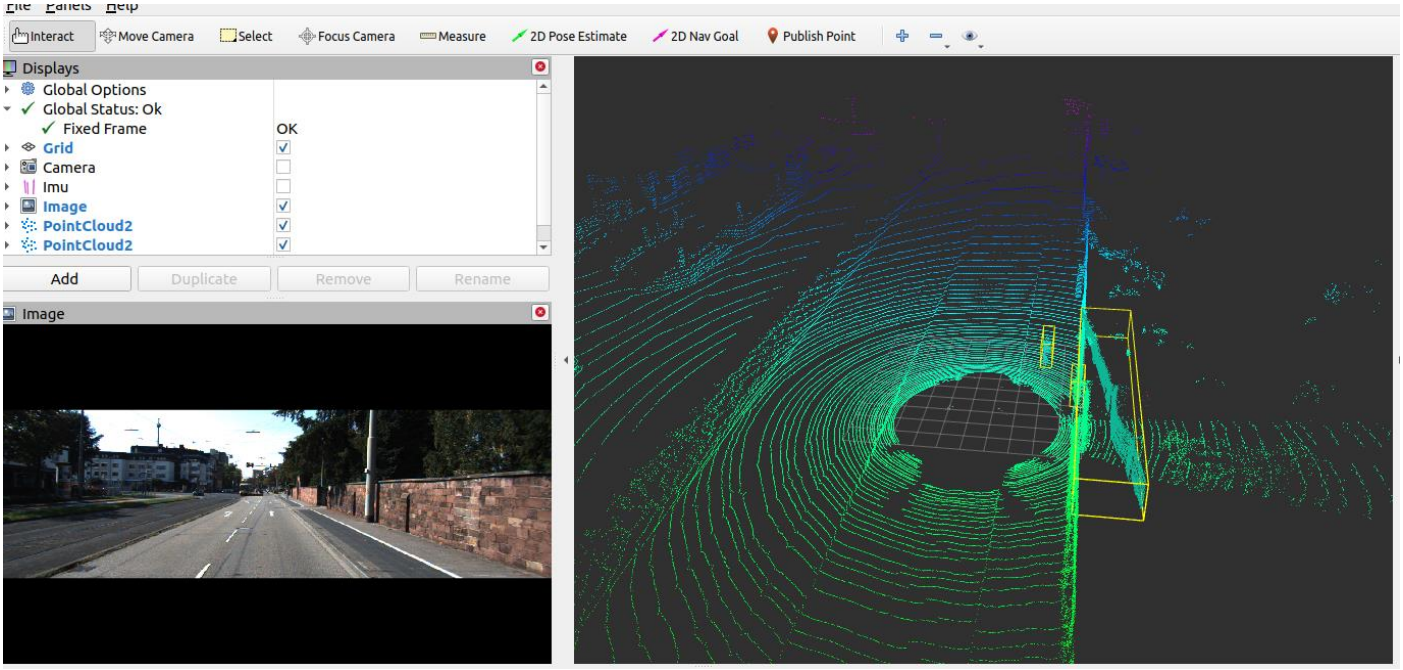
Part III: Extra Credit

Question5:

(10 pts) Run the obstacle detection code on a second dataset (kitti_2011_09_26_drive_0002_synced.bag) and record the results. Note that some algorithm parameters such as the ground level threshold and the filter-by-distance threshold may have to be changed.

- Need to change in the lab2.launch to run on a second dataset.

Output:



Question6:

(10 pts) The raw point cloud data is very noisy, and some obstacles could be falsely detected due to clutter or motion noise. To overcome this problem, implement the function filter_clusters to filter out clusters that are too small. Iterate through the point cloud clusters that were output by the Euclidean Clustering algorithm and remove clusters that have fewer than a certain number of points (e.g., 100 points). Tip: you may assign the cluster label as 0 to indicate that those points do not belong to any cluster.

```python
# TODO: (extra credit) implementation function to filter clusters by number of points
def filter_clusters(cluster_labels, min_cluster_size=100):
    """
    Filter out clusters that have fewer than a certain number of points.

    Parameters:
    cluster_labels (numpy.ndarray): Array containing cluster labels for each point.
    min_cluster_size (int): Minimum number of points a cluster should have to be considered valid.

    Returns:
    numpy.ndarray: Filtered cluster labels.
    """
    unique_clusters, cluster_counts = np.unique(cluster_labels, return_counts=True)

    # Iterate through unique clusters and filter out small clusters
    for cluster_id, count in zip(unique_clusters, cluster_counts):
        if cluster_id == 0:
            continue  # Skip unassigned points

        if count < min_cluster_size:
            # Set cluster label to 0 for small clusters
            cluster_labels[cluster_labels == cluster_id] = 0

    return cluster_labels

```

Question7:

(20 pts) With the basic implementation of Euclidean Clustering, the computation time is too high to be run in real-time on a robot. To overcome this problem, implement the function euclidean_clustering_accelerated where the code is optimized to run at a lower computational cost. One possible strategy is to use the FLANN library to speed up the process of searching for neighboring points within the Euclidean Clustering algorithm (https://pypi.org/project/flann/). Record the computation time in seconds and compare the performance between the accelerated and the non-accelerated functions for Euclidean Clustering. If you face any issues installing FLANN with Python3, you may refer to this Github issue. (https://github.com/primetang/pyflann/issues/1)

```python
def euclidean_clustering_accelerated(cloud, threshold=0.5):
    """
    Perform Accelerated Euclidean clustering on 3D points within distance threshold using FLANN.
    NumPy arrays to store cluster labels and visited points, which allows for efficient element-wise operations
    flann.nn_radius to find neighbors within the given threshold for each point.
    update cluster labels and mark visited points in a vectorized manner.
    """
    num_points = len(cloud)
    cluster_labels = np.zeros(num_points, dtype=int)
    cluster_id = 1  # Start with cluster ID 1

    # Create a FLANN index for efficient neighbor search within the threshold
    flann = FLANN()
    flann.build_index(cloud, algorithm='kdtree')

    # Initialize an array to keep track of visited points
    visited = np.zeros(num_points, dtype=bool)

    for i in range(num_points):
        if cluster_labels[i] != 0:
            continue

        # Initialize a queue for BFS traversal
        queue = [i]
        cluster_labels[i] = cluster_id

        while queue:
            point_idx = queue.pop(0) # Convert the queue to a NumPy array
            neighbors = flann.nn_radius(cloud[point_idx], threshold)[0]

            # Filter neighbors that have not been visited
            unvisited_neighbors = neighbors[~visited[neighbors]]

            # Mark all unvisited neighbors with the same cluster ID
            cluster_labels[unvisited_neighbors] = cluster_id

            # Mark all visited points
            visited[unvisited_neighbors] = True

            # Add unvisited neighbors to the queue
            queue.extend(unvisited_neighbors)

        cluster_id += 1

    return cluster_labels
```

- I use **pyflann** library which is the wrapper of **flann** library.
- Perform Accelerated Euclidean clustering on 3D points within distance threshold using **FLANN**.
- I used NumPy arrays to store cluster labels and visited points, which allows for efficient element-wise operations. This replaces the explicit loop over neighbors.
- I used **flann.nn_radius** to find neighbors within the given threshold for each point.
- I update cluster labels and mark visited points in a vectorized manner.
- Computation time reduce to ~0.2 seconds (on average) in compared to previous non-optimized euclidean clustering takes 2.2 seconds (on average) to run.

Question 8:

(max 20 pts) Open-ended. Implement any visualization or algorithmic improvements to the program. In your report, you justify the change and explain how advantageous it is compared to the original program.

- I use flann kdtree to index the point cloud points and to the neighbors which is more efficient method than the pseudocode provided in lab guide.
- I use vectorization operation to mark the visited neighbors without explicit use of for loop which simply reduce time computation from $O(n^3)$ to $O(n^2)$. And also optimize even further using Numpy array vectorization.
- The total computation time to find the cluster with accelerated algorithm takes ~ 0.2 seconds which seems fast during run time.