Lab 2 – 3D Obstacle Detection from Point Clouds
100 points

Overview:

In this lab, we will process a series of 3D point clouds acquired by a laser scanner to perform obstacle detection for an autonomous vehicle. To complete the lab you must submit via Canvas a zip file containing your source code, a video recording of your program execution, and a write-up describing your results and any extra credit problems you attempted.

Part I: Setup

1. Download the .zip file from Canvas for Lab 2/
2. First, we will copy everything over into our existing lab folder.
    - Unzip files from Canvas into the ai_labs folder
    - Then copy the python scripts into the scripts folder (e.g. by running:)
        - mv *.py scripts/
    - Make sure the scripts are executable
        - chmod +x scripts/*
    - Move the .bag files into the data/ folder
    - Move the .rviz file into the rviz/ folder

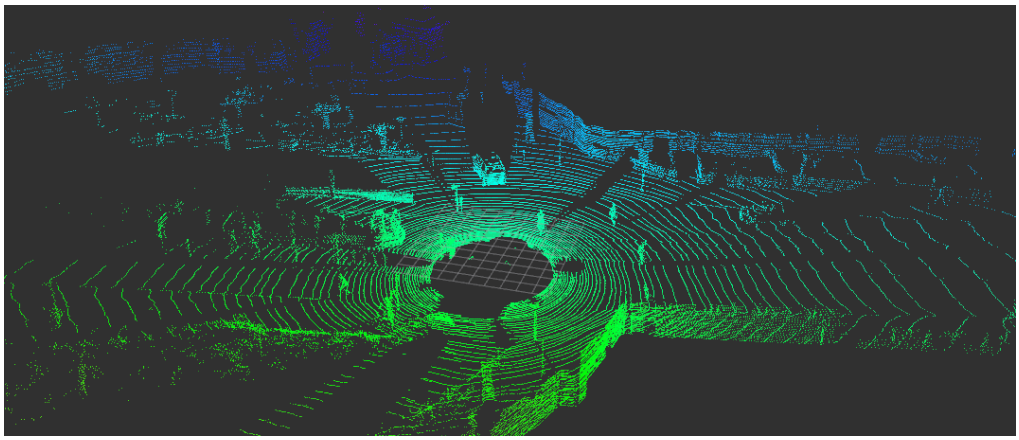
Part II: Point Cloud Processing

In this lab, we will be using point cloud data from the KITTI autonomous driving dataset http://www.cvlibs.net/datasets/kitti/. The 3D point cloud is processed to detect obstacles on the road such as cars, pedestrians, and cyclists.

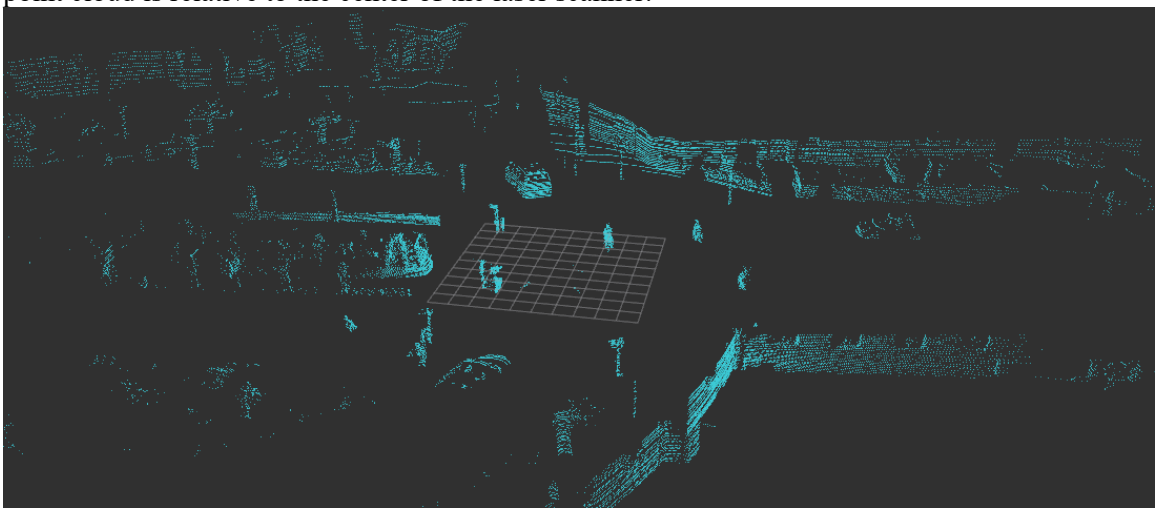
1. First, test that the ROS bag file (*kitti_2011_09_26_drive_0005_synced.bag*) and the visualization system are working. Run the following command from the terminal. You should see the RVIZ window pop up and show a visualization of the series of images and point clouds collected from the vehicle. The image data is not needed for this lab but is useful for visualization.

a. roslaunch ai_labs lab2.launch

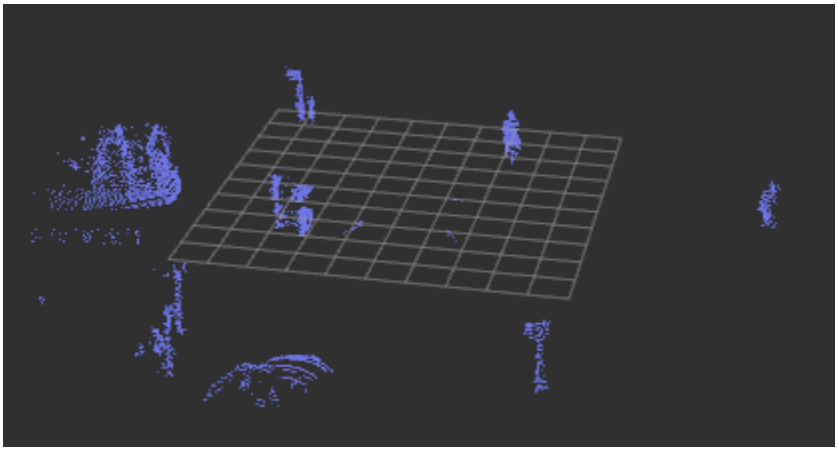
b. If it doesn't work, you can try to run catkin_make from the catkin_ws folder or to run source catkin_ws/devel/setup.bash



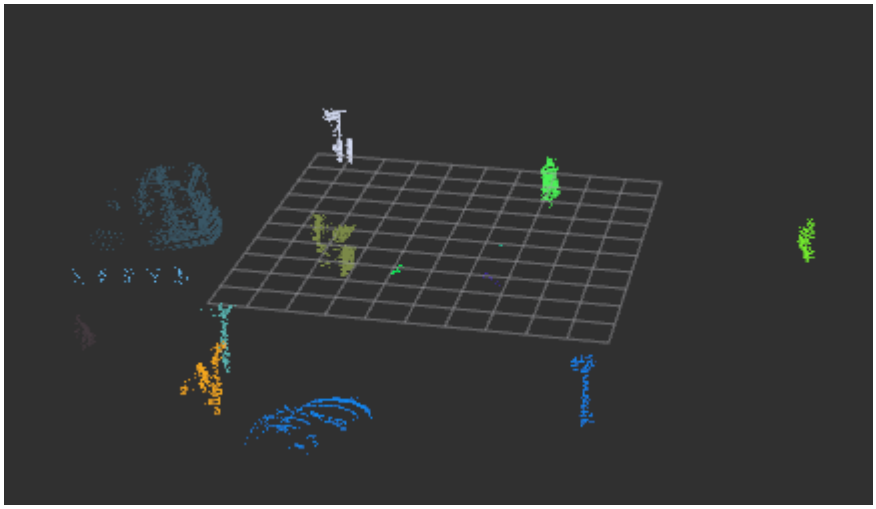
2. A 3D point cloud is a data structure consisting of an unordered list of 3D (*x, y, z*) points. In the dataset that we are using in this lab, the 3D point clouds are acquired using a Velodyne laser scanner mounted on top of the vehicle that emits a series of laser beams and uses those measurements to determine the 3D geometry of the surrounding environment. The next series of steps constitute a pipeline to process the 3D point cloud and reconstruct a bounding box for each obstacle detected in the point cloud.
3. **(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.



4. **(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.



5. **(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:



set up an empty list of clusters C, and a queue of the points that need to be checked Q;

then for every point $\boldsymbol{p}_i \in \mathcal{P}$, perform the following steps:

- add $\boldsymbol{p}_i$ to the current queue Q;
- for every point $\boldsymbol{p}_i \in Q$ do:
  - search for the set $\mathcal{P}_i^k$ of point neighbors of $\boldsymbol{p}_i$ in a sphere with radius $r < d_{th}$;
  - for every neighbor $\boldsymbol{p}_i^k \in \mathcal{P}_i^k$, check if the point has already been processed, and if not add it to Q;
- when the list of all points in Q has been processed, add Q to the list of clusters C, and reset Q to an empty list

the algorithm terminates when all points $\boldsymbol{p}_i \in \mathcal{P}$ have been processed and are now part of the list of point clusters C.

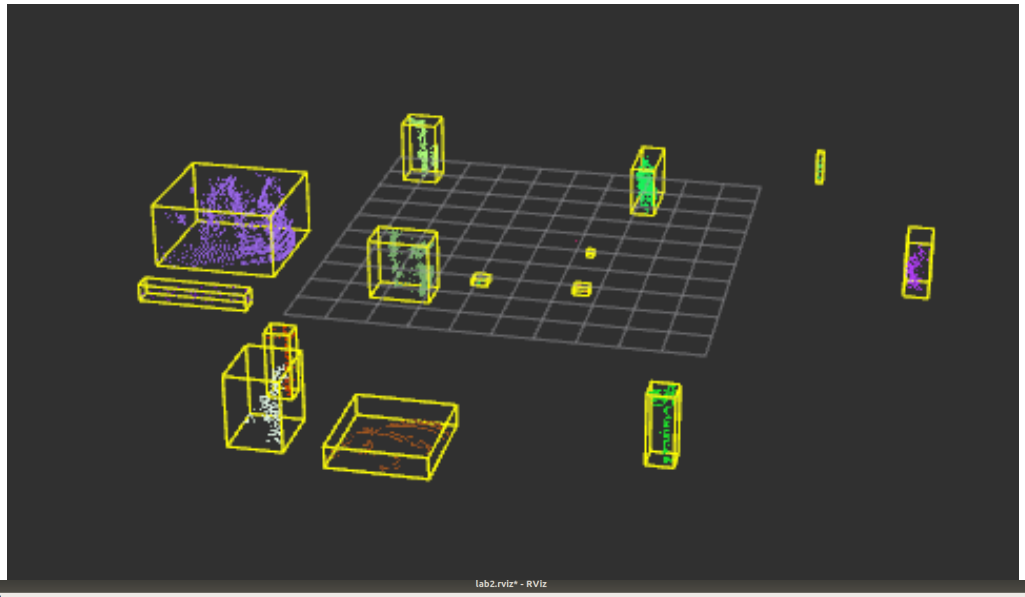
The detailed pseudocode for Euclidean clustering using a cluster label array is given below:

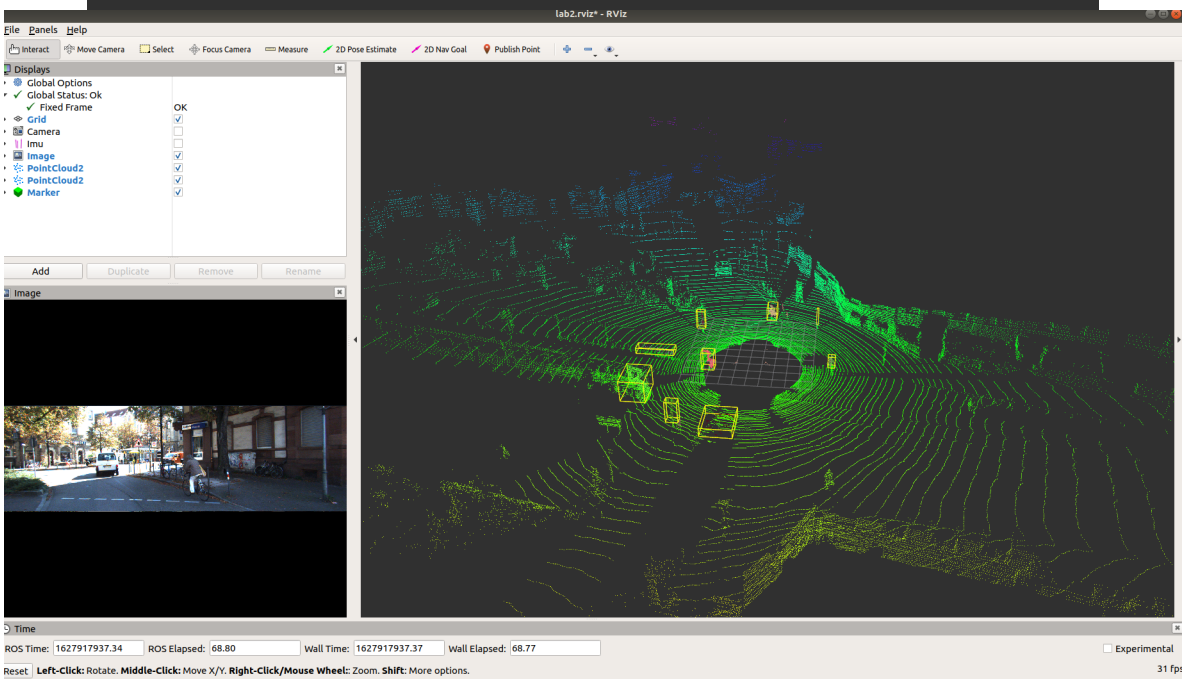
```
Set up an array of cluster labels, C
Set up a counter for the cluster ID, m
Set up a queue of the points that need to be checked Q
For every point p in the point cloud, P:
        • If p already has a label:
            ○ Continue
        • Add p to the current queue, Q
        • Assign the label of p to be m
        • While Q is not empty:
            ○ Pop the first point, q from Q
            ○ Search for the set of point neighbors N of q, i.e.
              points that have Euclidean distance < threshold
            ○ For every neighbor n in N:
                ▪ If n does not already have a label:
                    • Add n to Q
                    • Assign the label of n to be m
        • Reset Q to an empty list
        • Increment the counter, m by 1

Return the cluster labels, C
```

6. **(20 pts)** Implement the function *get_bounding_boxes* to compute the coordinates of the bounding box around each detected obstacle from the previous step. Each obstacle should have a

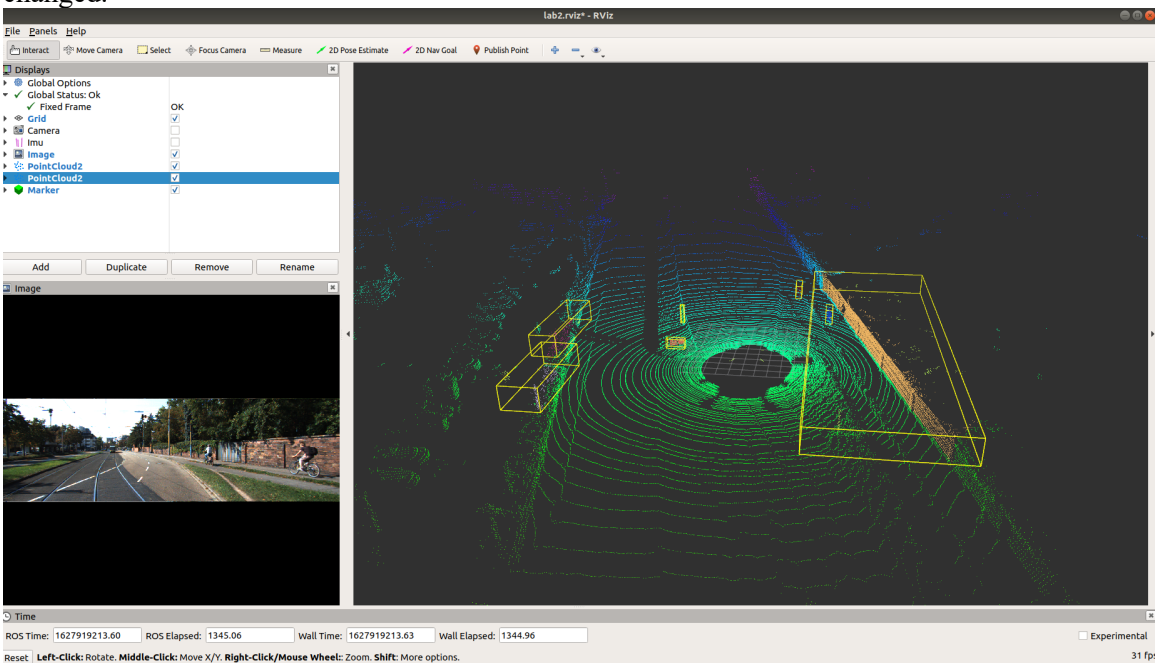
corresponding bounding box and each bounding box should be described by 8 points. The width of the bounding box should range from the minimum to maximum X coordinates in the corresponding point cloud cluster. The length of the bounding box should range from the minimum to maximum Y coordinates in the 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.





Part III: Extra Credit

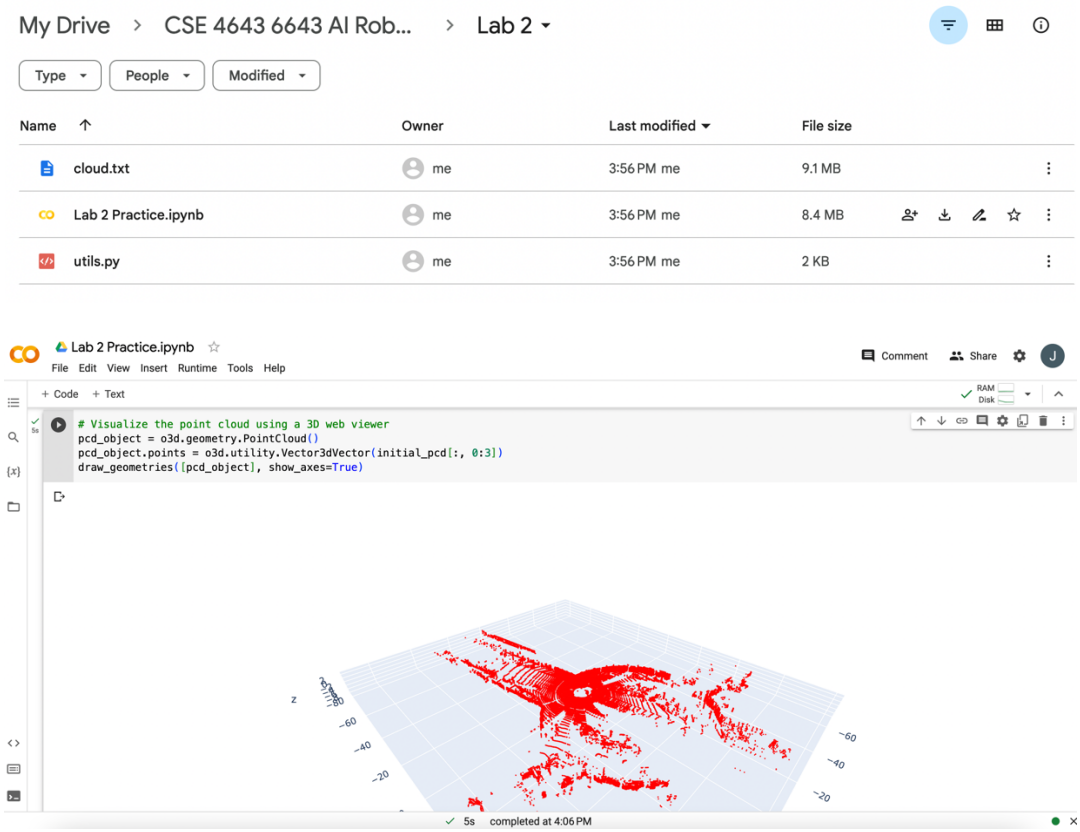
1. **(10 pts)** Run the obstacle detection code on a second dataset (*kitti_2011_09_26_drive_0002_synced.bag*) and record the results. Note that some algorithm parameters such as the ground level threshold and the filter-by-distance threshold may have to be changed.



2. **(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.
3. **(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)
4. **(max 20 pts)** Open-ended. Implement any visualization or algorithmic improvements to the program. In your report, justify the change and explain how it is advantageous compared to the original program.

Practice Notebook (optional)

Before running the whole system using Ubuntu and ROS, it might be a good idea to implement the relevant Python functions in a Jupyter notebook, make sure they work, then finally test the code with the full point cloud data sequence. You may use the provided files in the `lab_2_practice` folder in Google Colab or other Jupyter Notebook environments for practice (refer screenshots below).



Vectorization

When working with Python code, especially when dealing with robotics code that need to run as efficiently as possible, it is important to make use of a technique called vectorization. The NumPy library is essential to this because it provides a way of vectorizing code when performing calculations with numerical data. In short, vectorization refers to using of optimized, pre-compiled code (usually accessed through a single library function call) to perform mathematical operations over a sequence/vector of data instead of using an explicit iteration written in a compiled language code (i.e. using a "for-loop" written in Python).

For example, suppose that we are given the task of calculating the element-wise product of two arrays, "a" and "b", each containing 50000 numbers. There are two methods of performing this calculation:

Method 1: Non-vectorized code

```
for i in range(len(a)):
      vector[i]=a[i]*b[i]
```

Not efficient: takes 23.5ms to run!

Method 2: Vectorized code

```
vector =numpy.multiply(a, b)
```

More efficient: takes 0.2ms to run!

In summary, it is important to make use of vectorization whenever possible. Refer to the following link for more examples of vectorization using NumPy:
https://www.pythonlikeyoumeanit.com/Module3_IntroducingNumpy/VectorizedOperations.html

Frequently-asked Questions
Q: Why is the point cloud not updating after I implement my code?
A: You may have to uncheck the input point cloud message in RVIZ so that the output point cloud message is visible.

Q: How to implement a queue for Euclidean clustering?
A: You may either use a native Python list or the built-in queue library https://www.geeksforgeeks.org/queue-in-python/.

Q: What if my code runs too slowly?
A: Use vectorization whenever possible. The assignment will only be graded on correctness and not speed, so it is okay if your code is not optimal, since you are allowed to change the rosbag playback rate so that the algorithm can keep up.

Q: Are library functions allowed?
A: Numpy functions (np.sum, np.sqrt) and nearest neighbor functions are allowed (sklearn.neighbors, flann.nn_radius). Functions that implement an entire clustering algorithm are not allowed (scipy.cluster.hierarchy.fcluster, sklearn.cluster.DBSCAN).