Task 1: Principal Component Analysis (PCA) Normal Estimation

There are two parts in this task: (1) implement PCA function; (2) implement main function to compute the principal directions of the point cloud.

Description of code snippet:

(1) define point cloud data path.
(2) load point cloud data.
(3) get coordinates from input data.

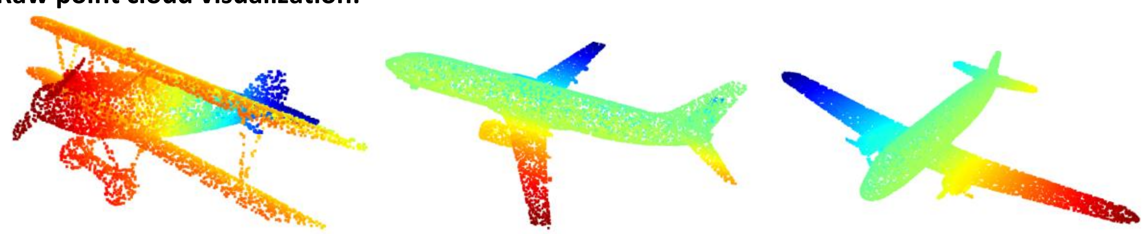
```python
""" ********************* load point cloud data ************************** """
# instance number, range: 0-39, 40 instances
category_index = 0
# path for point cloud dataset
root_dir = 'D:\PointCloud\modelnet40_normal_resampled'
category = os.listdir(root_dir)
point_cloud_name = '_0002.txt'
# default first point cloud data
file_name = os.path.join(root_dir,
                         category[category_index],
                         category[category_index] +
                         point_cloud_name)
print(file_name)
# six columns, the first three values are coordinates, the latter three are normals.
point_cloud_np = np.loadtxt(file_name, delimiter=',')
# get points from point cloud, we only process coordinates and ignore the normals
point_cloud_np = point_cloud_np[:, 0:3]
print('total points number is:', point_cloud_np.shape[0])
```

Description of code snippet:

(1) convert Numpy format data to Pandas DataFrame data.
(2) Convert DataFrame to point cloud format.
(3) Instantiate point cloud data and visualize the raw point cloud data.

```python
""" *************** raw point cloud visualization ************************* """
point_cloud_pd = pd.DataFrame(point_cloud_np)
point_cloud_pd.columns = ["x", "y", "z"]
point_cloud_pynt = PyntCloud(point_cloud_pd)
point_cloud_o3d = point_cloud_pynt.to_instance("open3d", mesh=False)
o3d.visualization.draw_geometries([point_cloud_o3d])
```

Raw point cloud visualization:



Description of code snippet (PCA implementation):

(1) Normalization:

$\tilde{X} = [\tilde{x}_1, \cdots, \tilde{x}_m], \tilde{x}_i = x_i - \bar{x}, i = 1, \cdots, m \qquad \bar{x} = \frac{1}{m}\sum_{i=1}^{m} x_i$

```python
# 1. normalize the data to be zero mean
data_mean = np.mean(data, axis=0)
data_normalized = data - data_mean
```

(2) Compute covariance matrix and eigenvalue and eigenvectors using SVD

Compute SVD $H = \tilde{X}\tilde{X}^T = U_r \Sigma^2 U_r^T$

```python
# 2. get covariance matrix
func = np.cov if not correlation else np.corrcoef
cov_matrix = func(data_normalized, rowvar=False, bias=True)
```

```python
# 3. method-1: singular value decomposition
eigenvectors, eigenvalues, eigenvectors_transpose = np.linalg.svd(cov_matrix, full_matrices=True)
print(eigenvectors)
```

(3) Decreasingly sort the eigenvalues and eigenvectors

The principle vectors are the columns of U_r (Eigenvector of X = Eigenvector of H)

```python
if sort:
    # argsort() is increasing sorting. with -1,
    # it becomes decreasing sorting.
    sort = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[sort]
    eigenvectors = eigenvectors[:, sort]
```

(4) Apply PCA to get principal direction of point cloud

```python
""" *************** apply PCA to get principal directions ************************** """
points = np.asarray(point_cloud_o3d.points)
eigenvalues, eigenvectors = PCA(points)
point_cloud_vector = eigenvectors[:, 2] # the vector in the principal direction of point cloud
print('the main orientation of this point cloud is: ', point_cloud_vector)
```

(5) Compute surface normal using KNN

Surface normal on 3D point cloud

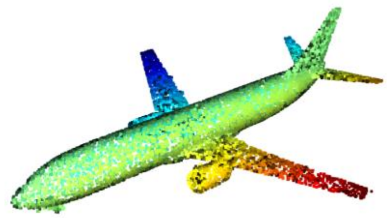
1. Select a point P
2. Find the neighborhood that defines the surface
3. PCA
4. Normal -> the least significant vector
5. Curvature -> ratio between eigen values $\lambda_3/(\lambda_1 + \lambda_2 + \lambda_3)$

```python
""" *************** compute the normal of each point iteratively ************************** """
# store raw point cloud data to KD tree
# prepare for applying nearest neighbor method to get points
pcd_tree = o3d.geometry.KDTreeFlann(point_cloud_o3d)
normals = []

# start the code
for i in range(points.shape[0]):
    """
    search_knn_vector_3d function
    input: [each point, the number of KNN]
    return: [int, open3d.utility.IntVector, open3d.utility.DoubleVector]

    find 10 KNN points for each point to get the fitting plane.
    apply PCA to get the eigenvector with minimum value as normal of that point
    """
    _, idx, _ = pcd_tree.search_knn_vector_3d(points[i], 10)
    k_nearest_point = points[idx, :]
    eigenvalues, eigenvectors = PCA(k_nearest_point)
    normals.append(eigenvectors[:, 2])
# end the code
# store the normals in the 3D point clouds
normals = np.array(normals, dtype=np.float64)
point_cloud_o3d.normals = o3d.utility.Vector3dVector(normals)
o3d.visualization.draw_geometries([point_cloud_o3d])
```

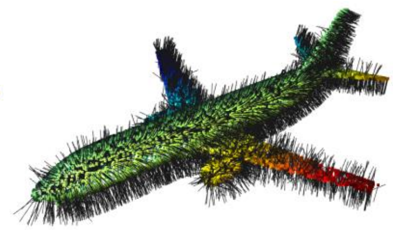
(6) Results visualization

```
""" *************** apply PCA to get principal directions ************************* """
points = np.asarray(point_cloud_o3d.points)
eigenvalues, eigenvectors = PCA(points)
point_cloud_vector1 = eigenvectors[:, 0]
point_cloud_vector2 = eigenvectors[:, 1]
point_cloud_vector3 = eigenvectors[:, 2]
print('the first component of this point cloud is: ', point_cloud_vector1)
print('the second component of this point cloud is: ', point_cloud_vector2)
print('the third component of this point cloud is: ', point_cloud_vector3)
```

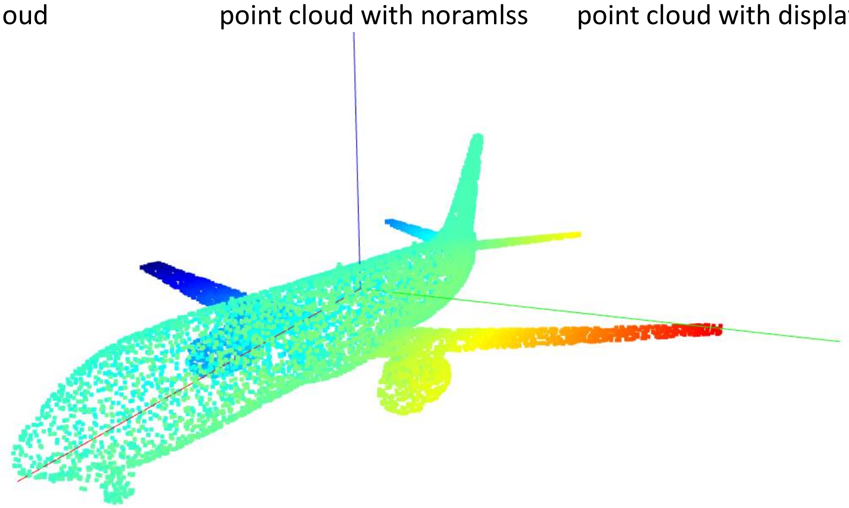
raw point cloud



point cloud with noramlss

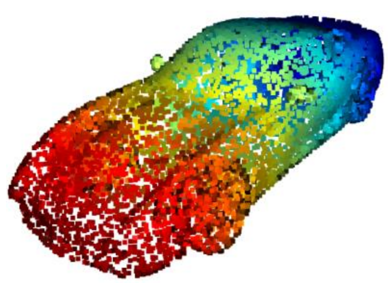


point cloud with displaying normals

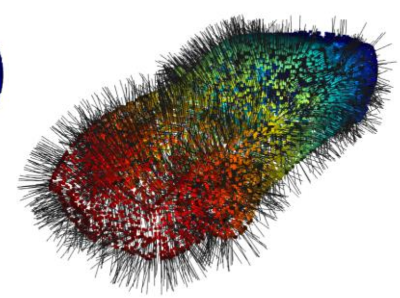


Point cloud with three principal components

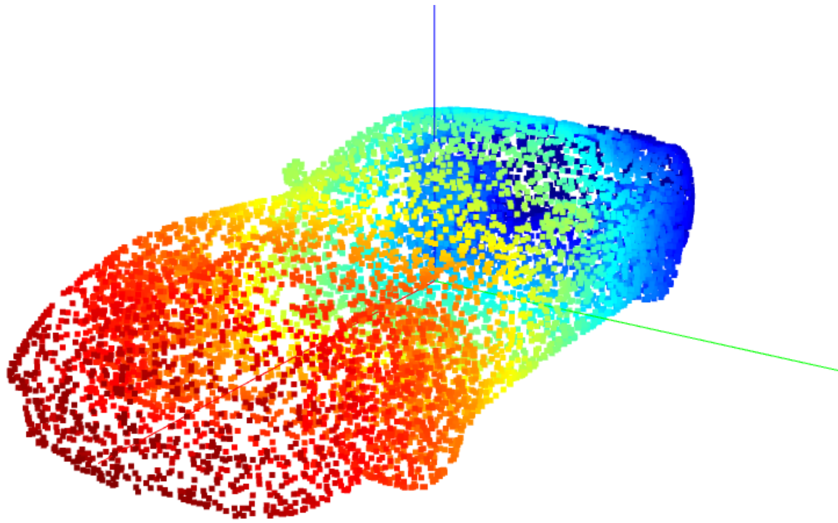
Raw point cloud



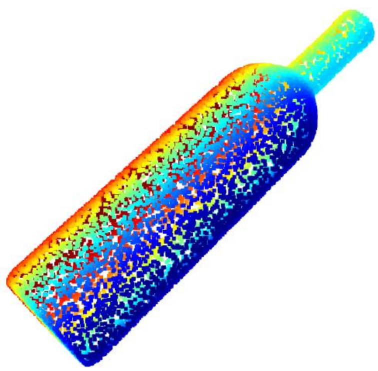
point cloud with noramlss



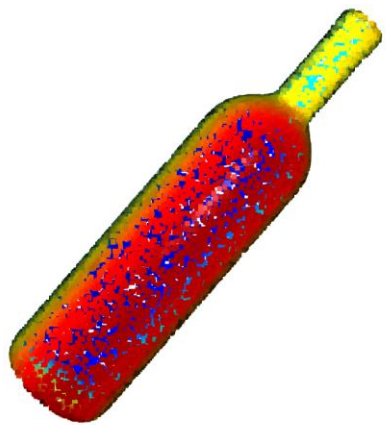
point cloud with displaying normals



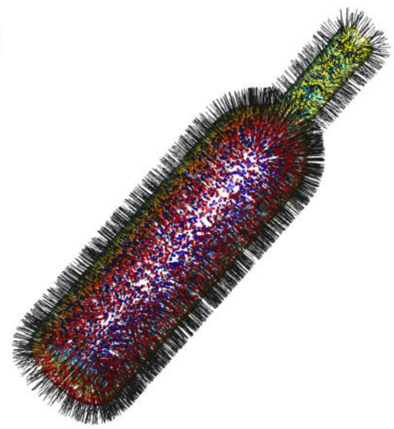
Point cloud with three principal components



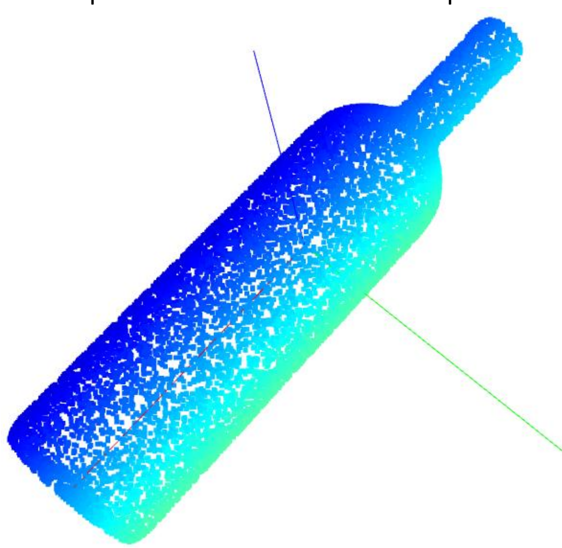
Raw point cloud



point cloud with noramlss



point cloud with displaying normals



Point cloud with three principal components

Task 2: Voxel Filtering Down-sampling

Algorithm logic:

1. Compute the min or max of the point set $\{p_1, p_2, \cdots p_N\}$
$$x_{max} = \max(x_1, x_2, \cdots, x_N), x_{min} = \min(x_1, x_2, \cdots, x_N), y_{max} = \cdots\cdots$$
2. Determine the voxel grid size r
3. Compute the dimension of the voxel grid
$$D_x = (x_{max} - x_{min})/r$$
$$D_y = (y_{max} - y_{min})/r$$
$$D_z = (z_{max} - z_{min})/r$$
4. Compute voxel index for each point
$$h_x = \lfloor (x - x_{min})/r \rfloor$$
$$h_y = \lfloor (y - y_{min})/r \rfloor$$
$$h_z = \lfloor (z - z_{min})/r \rfloor$$
$$h = h_x + h_y * D_x + h_z * D_x * D_y$$
5. Sort the points according to the index in Step 4
6. Iterate the sorted points, select points according to Centroid / Random method
0, 0, 0, 0, 3, 3, 3, 8, 8, 8, 8, 8, 8, 8, 8, ……

Code Snippets:

Step 1: Compute the minimum and maximum values of point cloud data

```python
def voxel_filter(point_cloud, leaf_size):
    filtered_points = []

    # hw3
    # start the code

    # step1: compute the min or max of the point
    x_max, y_max, z_max = point_cloud.max(axis=0)
    x_min, y_min, z_min = point_cloud.min(axis=0)
```

Step 2: assign voxel grid size from input

```python
# step2: determine the voxel grid size r
voxel_grid_size = leaf_size
```

Step 3: compute the dimension of voxel grid

```python
# step3: Compute the dimension of the voxel grid
Dx = (x_max - x_min) / voxel_grid_size
Dy = (y_max - y_min) / voxel_grid_size
Dz = (z_max - z_min) / voxel_grid_size
```

Step 4: compute voxel index for each point

```python
# step4: Compute voxel index for each point
point_cloud = np.asarray(point_cloud)
h = []
for i in range(point_cloud.shape[0]):
    hx = np.floor((point_cloud[i][0] - x_min) / voxel_grid_size)
    hy = np.floor((point_cloud[i][1] - x_min) / voxel_grid_size)
    hz = np.floor((point_cloud[i][2] - x_min) / voxel_grid_size)
    H = hx + hy * Dx + hz * Dx * Dy
    h.append(H)
h = np.asarray(h)
```

Step 5: Sor the points according to the index in Step 4

```python
# step5: Sort the points according increasingly to the index in step4
voxel_index = np.argsort(h)
h_sort = h[voxel_index]
```

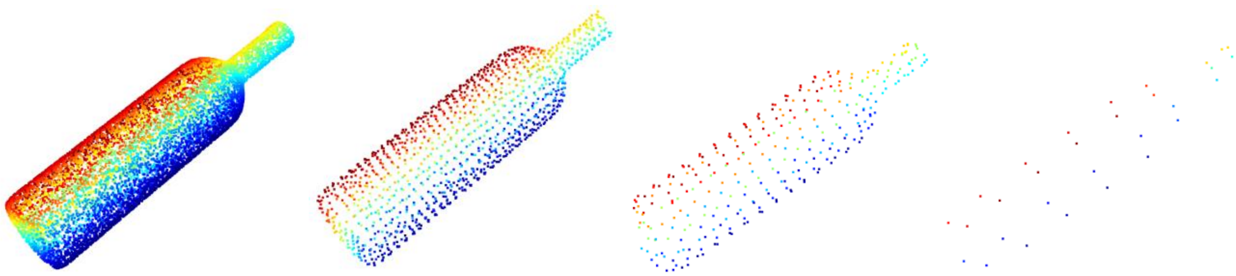
Step 6: Iterate the sorted points, select points according to Centroid / Random method

```python
# step6: Iterate the sorted points, select points according to Centroid / Random method
index_begin = 0
for i in range(len(voxel_index) - 1):
    if (h_sort[i] == h_sort[i + 1]):
        continue

    point_index = voxel_index[index_begin:(i + 1)]
    filtered_points.append(np.mean(point_cloud[point_index], axis=0))
    index_begin = i
```

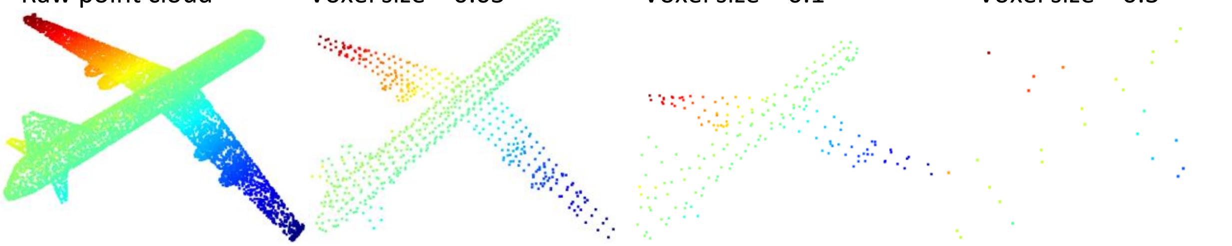
Return the result in float64 format which can avoid overflow.

```python
# change point cloud format to np.array
filtered_points = np.array(filtered_points, dtype=np.float64)
return filtered_points
```

Filtered Result Visualization:



Raw point cloud | Voxel size = 0.05 | Voxel size = 0.1 | Voxel size = 0.3



Raw point cloud | Voxel size = 0.05 | Voxel size = 0.1 | Voxel size = 0.3

Raw point cloud | Voxel size = 0.05 | Voxel size = 0.1 | Voxel size = 0.3