0.作业代码的执行方法

①首先安装依赖库，`Eigen`和`PCL`

②然后进行编译

```
cd lecture_1
mkdir build
cd build
cmake ..
make -j4
```

③运行

```
cd build
./test_read_model_data # 第一题
./test_pca #第二题
./test_normal_vector # 第三题
./test_voxel # 第四题
```

> 以下所有代码都是采用的C++编写，运行的系统是Ubuntu18.04，PCL版本是1.8，矩阵运算的库是Eigen

1.Build dataset for Lecture 1

a. Download ModelNet40 dataset
b. Select one point cloud from each category

对于第一题设计统一的数据点格式，为了方便后续的作业完成

```
class ModelData{
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
    ModelData() = default;

    unsigned int vertex_number_ = 0;
    unsigned int face_number_ = 0;
    unsigned int edge_number_ = 0;

```

```
  typedef typename std::vector<Eigen::Vector3d,
      Eigen::aligned_allocator<Eigen::Vector3d>> TypeVertexVector;
  typedef typename std::vector<Eigen::Matrix<unsigned int, 4, 1>,
      Eigen::aligned_allocator<Eigen::Matrix<unsigned int, 4, 1>>>
TypeFaceVector;

  TypeVertexVector vertices_;
  TypeFaceVector faces_;
};
```

2.Perform PCA for the 40 objects, visualize it.

①首先，编写读取`off`文件的代码，将其保存成如下形式：完整代码，参考这里

②编写主成分分析的代码：完整代码，参考这里

在计算主成分时，使用Eigen库中提供的svd分解函数，其部分代码如下：

```
void PrincipleComponentAnalysis::CalculatePrincipleVector() {
  Eigen::Vector3d center = X_.rowwise().mean();
  Eigen::MatrixXd normalized_X = X_.colwise() - center;

  Eigen::JacobiSVD<Eigen::MatrixXd> svd(normalized_X,
Eigen::ComputeFullU);
  principle_vector_ = svd.matrixU();
}
```

③执行Encoder操作，通过函数的接口中的`dim`变量控制使用主向量的个数：

```
Eigen::MatrixXd PrincipleComponentAnalysis::Encoder(unsigned int
dim) {
  if (dim > principle_vector_.cols()){
    std::cerr << "dimension greater than number of principle vector!"
<< std::endl;
  }

```

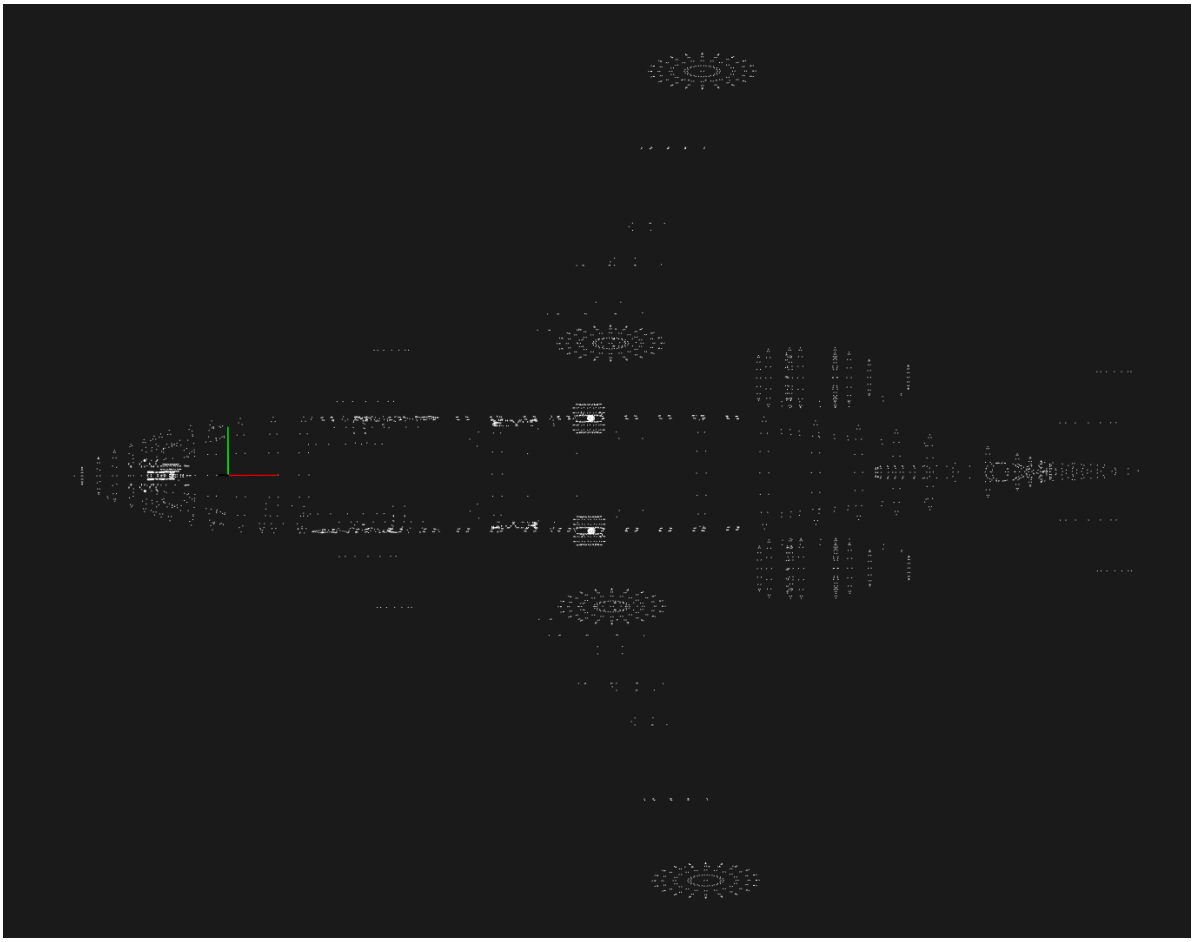
```
    Eigen::MatrixXd compressed_X;
    compressed_X.resize(3, X_.cols());
    compressed_X.setZero();

    Eigen::Matrix3d part_principle_vector = Eigen::Matrix3d::Zero();

    for (int i = 0; i < dim; ++i) {
      part_principle_vector.row(i) = principle_vector_.col(i).transpose();
    }

    return part_principle_vector * X_;
}
```

④实验结果

- airplane_0627.off

a.当选取主成分中的三个时，得到的点云与原来的点云没有区别，效果如下：

b.当选取主成分中的前两个时，飞机高度方向的信息被压缩了，效果如下：



c.当选取主向量中最大的那一个时，飞机的宽度和高度被压缩，只剩下长度信息

时间：对于airplane0627数据集，其中有一万多个点，但是作业中实现的代码，只需要0.000779902s就能完成主向量的计算。

实验了另外的数据集，获得与上面同样的结论，主成分分析对于提炼数据中的重要部分还是非常有用和高效的。

3. Perform surface normal estimation for each point of each object, visualize it.

①首先编写函数接口，主要函数如下：[完整代码，参考这里]

```
1  Vector3ds SurfaceNormalEstimation::CalculateNormalVector(const
   ModelData::TypeVertexVector &points, const double radius_threshold
```

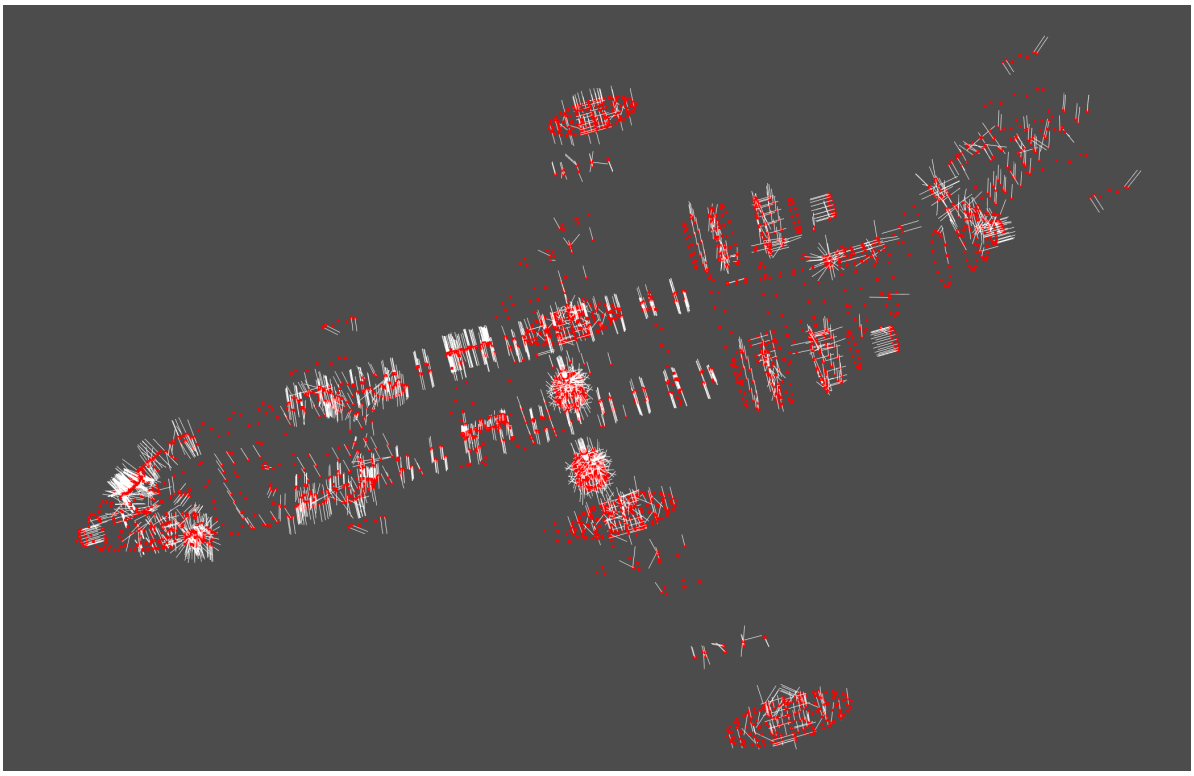
```
) {
  Vector3ds normal_vectors;
  normal_vectors.reserve(points.size());

  for (int i = 0; i < points.size(); ++i) {
    PrincipleComponentAnalysis principle_component_analysis;
    ModelData::TypeVertexVector part_points;
    part_points.emplace_back(points[i]);
    for (int j = 0; j < points.size(); ++j) {
      if (i == j){
        continue;
      }

      double radius = (points[i] - points[j]).norm();

      if (radius <= radius_threshold){
        part_points.emplace_back(points[j]);
      }
    }

    if (part_points.size() < 3){
      normal_vectors.emplace_back(Eigen::Vector3d(0,0,0));
      continue;
    }

    principle_component_analysis.InputData(part_points);

 normal_vectors.emplace_back(principle_component_analysis.Calcul
ateNormalVector());
  }

  return normal_vectors;
}
```

主要思路是借用第一题实现的pca代码，这其中最近邻代码的搜索是一个重要问题，它决定了大部分的计算效率，由于第二章才会讲kdtree搜索，所以这里暂时先采用暴力搜索，为了在第二章与kdtree的时间进行对比。

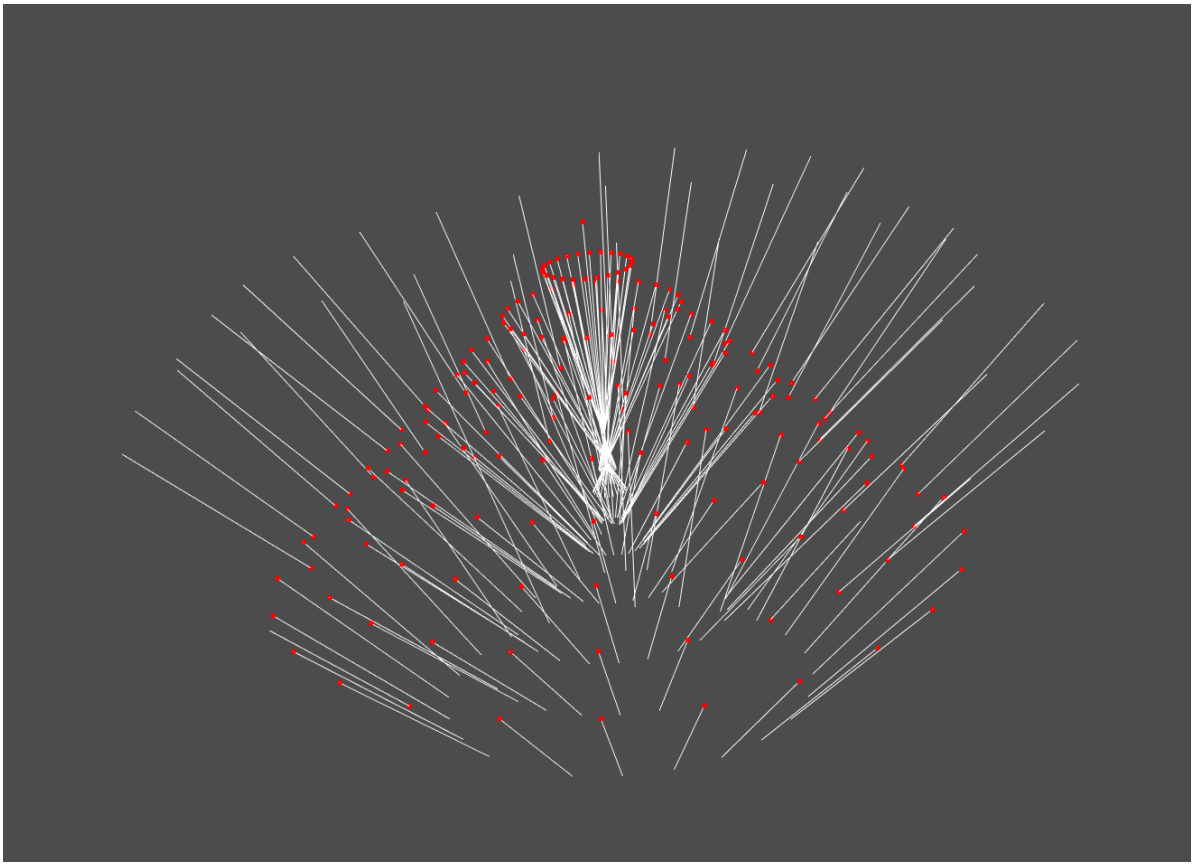
> 作业中的最近邻判断是根据距离，满足一定距离的点都算是邻居，实际上这里还可以采用距离最近的前几个点作为邻居，这种方法会降低算法对人工设定半径的依赖性。

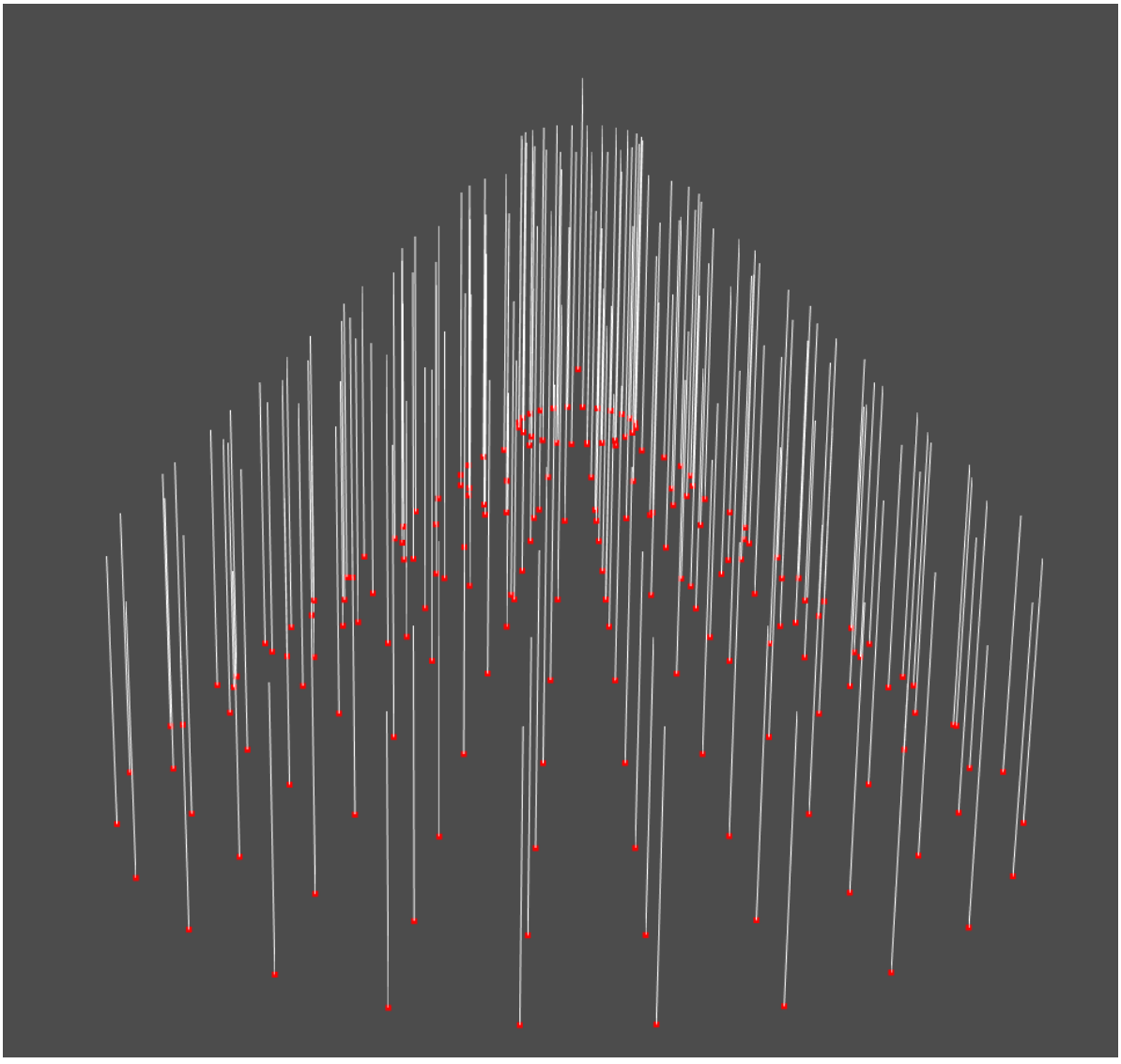
②：实验结果

时间：对于airplane0627数据集，其中有一万多个点，在采用暴力搜索的方案中，将半径设为5m，大于需要7.34671可以求解出整个飞机的法向量，效果如下：



问题：在求解法向量时，作业中实现的方法会过分依赖于数据点的形态，以及搜索半径，不同的搜索半径可能得到的结果大不相同。以下两张图分别将搜索半径设置为5m和1m，可以看到设置为1m时法向量的计算结果很差。





4. Downsample each object using voxel grid downsampling (exact, both centroid &random). Visualize the results.

①对于voxel filter算法的实现，可以看考代码中的`voxel_filter.cpp`，方法完全是按照ppt中的讲解进行的实现。

在构造函数中，输入`voxel grid`的大小

```
VoxelFilter::VoxelFilter(const Eigen::Vector3d &voxel_grid_size) {
  voxel_grid_size_ = voxel_grid_size;
}
```

②然后输入原始点数据：

```
void VoxelFilter::InputPoints(const ModelData::TypeVertexVector
&source_points) {
  source_points_ = source_points;
}
```

③执行voxel滤波

```
void VoxelFilter::FilterByCentroid(Vector3ds &target_points) {
  double x_min = std::numeric_limits<double>::max();
  double x_max = std::numeric_limits<double>::min();
  double y_min = std::numeric_limits<double>::max();
  double y_max = std::numeric_limits<double>::min();
  double z_min = std::numeric_limits<double>::max();
  double z_max = std::numeric_limits<double>::min();

  for (int i = 0; i < source_points_.size(); ++i) {
    const Eigen::Vector3d &point = source_points_.at(i);

    if (point.x() < x_min) {
      x_min = point.x();
    }

    if (point.x() > x_max) {
      x_max = point.x();
```

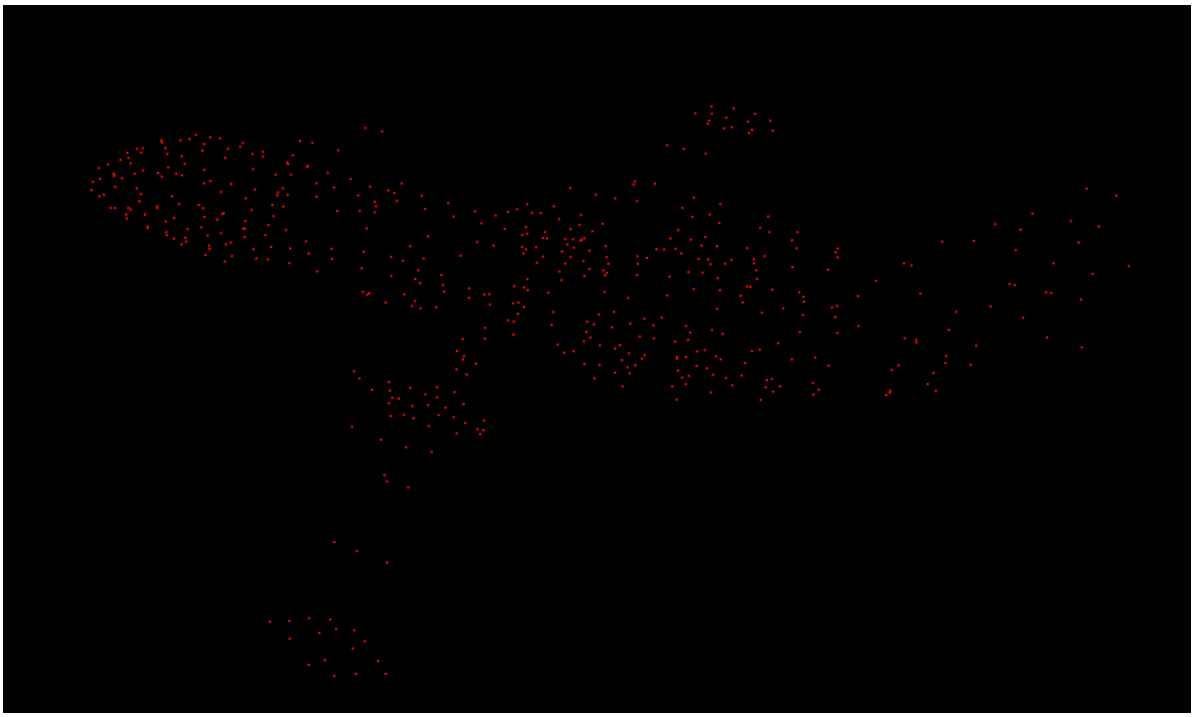
```
18        }
19
20        if (point.y() < y_min) {
21          y_min = point.y();
22        }
23
24        if (point.y() > y_max) {
25          y_max = point.y();
26        }
27
28        if (point.z() < z_min) {
29          z_min = point.z();
30        }
31
32        if (point.z() > z_max) {
33          z_max = point.z();
34        }
35      }
36
37      if (voxel_grid_size_.x() == 0 ||
38        voxel_grid_size_.y() == 0 ||
39        voxel_grid_size_.z() == 0) {
40        std::cerr << "voxel grid size equal 0" << std::endl;
41      }
42
43      int D_x = std::ceil((x_max - x_min) / voxel_grid_size_.x());
44      int D_y = std::ceil((y_max - y_min) / voxel_grid_size_.y());
45      int D_z = std::ceil((z_max - z_min) / voxel_grid_size_.z());
46
47      std::vector<SearchIndex> search_indices(source_points_.size());
48      for (unsigned int i = 0; i < source_points_.size(); ++i) {
49        const Eigen::Vector3d &point = source_points_.at(i);
50        unsigned int h_x = std::floor((point.x() - x_min) /
voxel_grid_size_.x());
51        unsigned int h_y = std::floor((point.y() - y_min) /
voxel_grid_size_.y());
52        unsigned int h_z = std::floor((point.z() - z_min) /
voxel_grid_size_.z());
53
54        unsigned h = h_x + h_y * D_x + h_z * D_x * D_y;
55
```

```
56        search_indices.at(i).point_index = i;
57        search_indices.at(i).voxel_index = h;
58      }
59
60      std::stable_sort(search_indices.begin(), search_indices.end(),
    myCmp);
61
62      Eigen::Vector3d point(0,0,0);
63      int count = 0;
64      for (unsigned int i = 1; i < search_indices.size(); ++i) {
65        point += source_points_.at(search_indices.at(i-1).point_index);
66        count++;
67
68        if (search_indices.at(i - 1).voxel_index !=
    search_indices.at(i).voxel_index ) {
69          point = point / (count * 1.0);
70          target_points.emplace_back(point);
71
72          if (i == search_indices.size() - 1){
73
     target_points.emplace_back(source_points_.at(search_indices.at(i).p
    oint_index));
74            break;
75          }
76
77          count = 0;
78          point.setZero();
79        }
80
81        if (search_indices.at(i-1).voxel_index ==
    search_indices.at(i).voxel_index
82          && i == search_indices.size())
83        {
84          target_points.back() = (target_points.back() +
    source_points_.at(search_indices.at(i).point_index)) / 2.0;
85        }
86      }
87  }
```

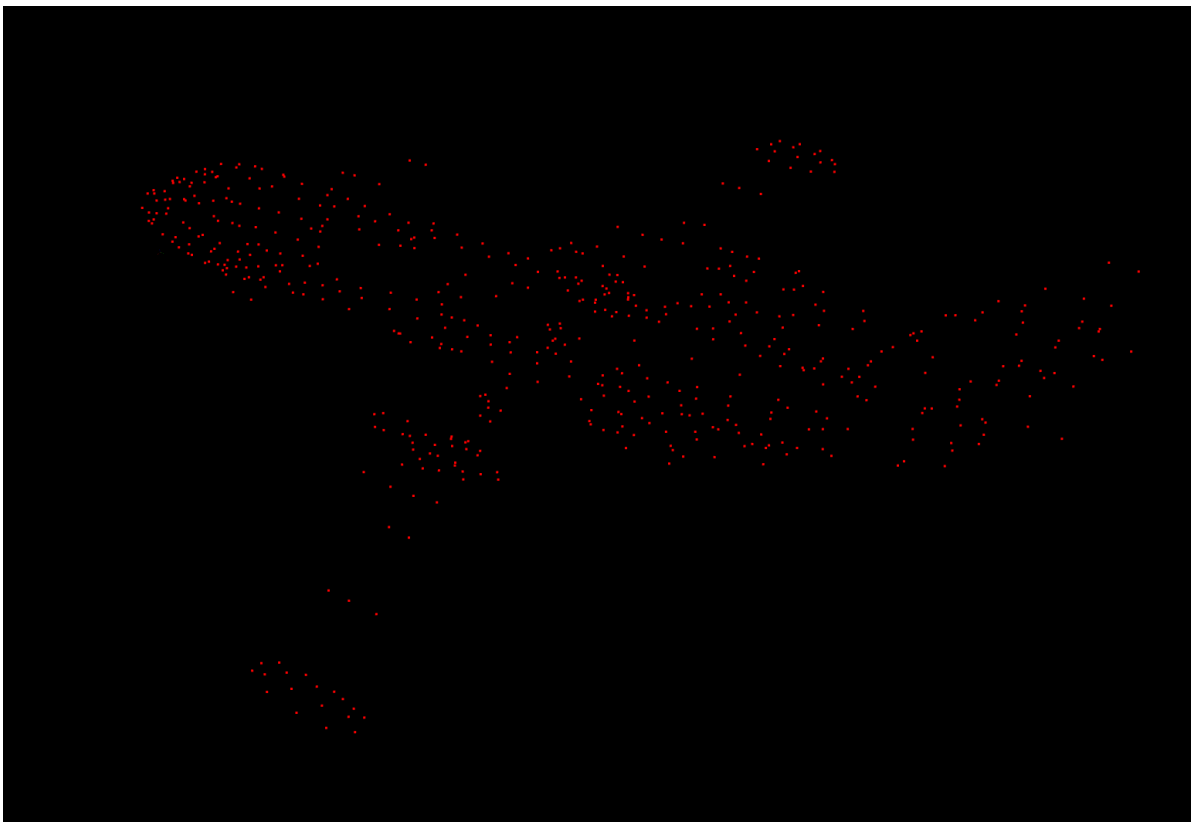
④ 实验结果：

通过与pcl库中的voxel filter算法进行对比发现，在结果上作业题中实现的效果与其没有什么大的差异，效果基本一致。以下是将voxel grid的大小设为[30.0,30.0,30.0]之后的滤波效果对比：

pcl库自带voxel filter算法



作业中实现的算法：



结论：

对于voxel filter算法而言，其滤波效果总体不错，效果很高，在选点策略上会对最终的效果造成一定的影响，对于栅格尺寸较小时，选择两种差别并不大。当栅格尺寸很大时，需要根据实际的效果进行选择。