

PclCSharp

1 简介

该文档主要介绍PointCloudSharp命名空间和PclCSharp命名空间中的各个函数。

2 PointCloudSharp命名空间

PointCloudSharp中封装了pcl中存储点云的基本数据结构，包括

`pcl::PointCloud<pcl::PointXYZ>`、`vector<pcl::PointIndices>` 和 `pcl::PointCloud<pcl::Normal>` 等。目前封装好的只有 `pcl::PointCloud<pcl::PointXYZ>`、`vector<pcl::PointIndices>` 这两类，其他的数据结构后续逐步封装。

2.1 PointCloudXYZ

该类是对 `pcl::PointCloud<pcl::PointXYZ>` 的封装。

2.1.1 属性介绍

1.Width: 表示点云宽度（如果点云是有序的，类似图像结构,则代表点云的列数），即一行点云的数量。

2.Height:表示点云高度（如果点云是有序的，类似图像结构,则代表点云的行数），即一列点云的数量。

若为有序点云，height可以大于1，即多行点云；若为无序点云，height需要等于1，即一行点云，此时width的数量即为点云的数量。

大部分传感器的数据源都是无序点云，但也有某些传感器，如TOF相机采集的数据源是有序点云。有序点云的优势在于，通过知道两个相邻点的关系，最近邻点操作会变得更加有效，因此可以加速计算。而对于无序点云来说，需要使用Kdtree或者Octree算法来给点云有序化，以此加快运行速度，这也是为什么大部分点云处理算法需要设置近邻数的原因（设置近邻数实际上是设置Kdtree或者Octree的参数）。

3.Size:整个点云的数量。

4.PointCloudXYZPointer:最重要的属性！**该属性实际是PCL中** `pcl::PointCloud<pcl::PointXYZ>`

`*`。为了提高运行效率，使用数组不是合适的选择。所以必须要把C++中的

`pcl::PointCloud<pcl::PointXYZ> *` 封装给C#使用。该属性在C#中是一个**IntPtr类型**，里面包含了所有的点云信息。

2.1.2方法介绍

```
///  
///@brief 无参构造函数  
///@details 初始化一个空的点云对象  
public PointCloudXYZ()  
  
///  
///@brief 有参构造函数  
///@details 传入点云文件地址，加载对应的点云对象到_PointCloudPointer中  
///@param path 点云文件路径  
public PointCloudXYZ(string path)  
  
///  
///@brief 析构函数  
///@details 内存管理。当对象的生存期结束时，释放掉点云对象内存。这里容易出现野指针，万分注意！  
~PointCloudXYZ()
```

```

///@brief 获得某点的X值
///@param index 点的索引号
public double GetX(int index)

///@brief 获得某点的Y值
///@param index 点的索引号
public double GetY(int index)

///@brief 获得某点的Z值
///@param index 点的索引号
public double GetZ(int index)

///@brief 重置点云大小
///@param size 点云的大小
public void ReSize(int size)

///@brief 压入一个点，压入的位置在元素最后面
///@param x 压入点的x值
///@param y 压入点的y值
///@param z 压入点的z值
public void Push(double x, double y, double z)

///@brief 获得点云中的极值
///@param out_res 极值结果
///@details 使用一个double数组保存结果，大小为6.其中，
///第一第二个元素依次是x的最小值、最大值，第三第四个元素是y的最小最大值，后面依次类推
public void GetMinMaxXYZ(double[] out_res)
{
    getMinMaxXYZ(_PointCloudPointer, out_res);
}

///@brief 弹出一个点，类似于出栈，弹出的是元素中最后一个点
public void Pop()

///@brief 清空点云对象中所有点，对象的size属性变为0
public void Clear()

```

2.2 PointIndices

该类是对 `vector<pcl::PointIndices>` 的封装，**主要是为了解决存储分割结果的问题**。点云经过分割之后，会被分为n个点簇，每个点簇用点的索引表示，也就是pcl中的 `pcl::PointIndices`。整个结果用一个vector存储，也就是 `vector<pcl::PointIndices>`。C#中没有类似的结构，所以将其封装为 `PointIndices` 类供C#使用。

2.2.1 属性介绍

1.Size:分割出来的点簇数量

2.PointIndicesPointer:**该属性实际是PCL中** `vector<pcl::PointIndices> *`，在C#中是一个 `IntPtr` 类型，里面包含了所有的点簇信息。

2.2.2 方法介绍

```

///@brief 无参构造函数
///@details 初始化一个空的点云索引对象
public PointIndices()

///@brief 析构函数
///@details 内存管理。当对象的生存期结束时，释放掉点云对象内存。这里容易出现野指针，万分注意！
~PointIndices()

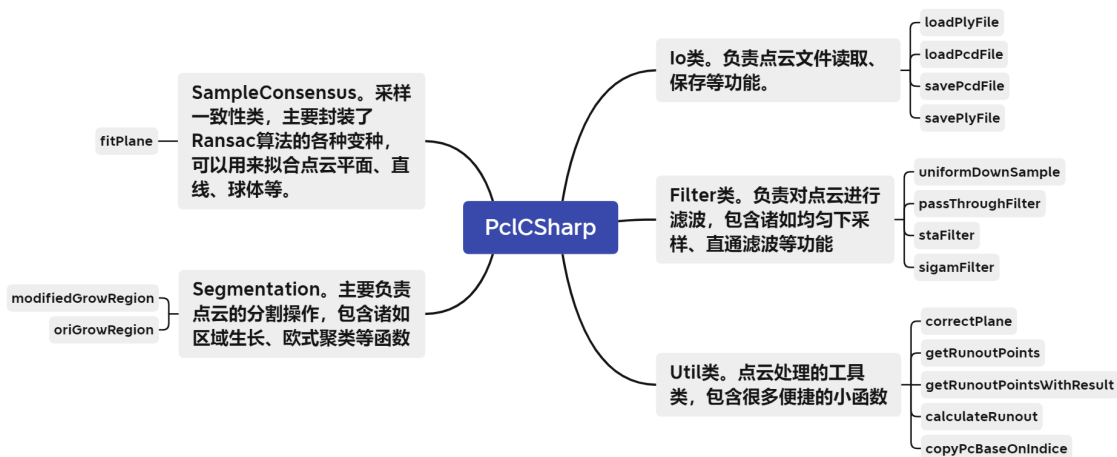
///@brief 根据索引获得对应的点簇指针，也就是pcl::PointIndices，一个点簇对应着一个分割结果
///@param index 索引号
public IntPtr GetPointIndex(int index)

///@brief 根据索引获得对应点簇索引的点数大小
///@param index 索引号
///@details 这个函数一般用于筛选点簇中。比如分割的结果中包含很多点簇，但我只想要点数最多或者最少的点簇，那就可以使用这个函数
///做一个判断，选出点数最多或最少的点簇
public int GetSizeOfIndex(int index)

```

3 PclCSharp命名空间

该命名空间中包含了pcl中点云处理的算法，暂时封装了**Io**、**Filter**、**Segmentation**、**SampleConsensus**和**Util**五个静态类，每个类大体对应着pcl的一个模块，后续有空会慢慢增加其他模块。各个类之间的依赖关系见下图



3.1 lo

Io模块中主要包含点云文件的**读取保存功能**，目前可加载pcd、ply、obj和stl格式的文件，**并可将这四种格式文件转成pcd或者ply格式文件**。

3.1.1 loadPlyFile

```
/// @brief 加载ply文件
/// @param path 文件路径
/// @param pc 点云对象指针，此处使用PointCloudSharp类的PointCloudPointer属性
public static extern int loadPlyFile([MarshalAs(UnmanagedType.LPStr)] string
path, IntPtr pc);
```

3.1.2 loadPcdFile

```
/// @brief 加载pcd文件
/// @param path 文件路径
/// @param pc 点云对象指针，此处使用PointCloudXYZ类的PointCloudXYZPointer属性
public static extern int loadPcdFile([MarshalAs(UnmanagedType.LPStr)] string
path, IntPtr pc);
```

3.1.3 loadObjFile

```
/// @brief 加载obj文件
/// @param path 文件路径
/// @param pc 点云对象指针，此处使用PointCloudSharp类的PointCloudPointer属性
public static extern int loadObjFile([MarshalAs(UnmanagedType.LPStr)] string
path, IntPtr pc);
```

3.1.4 stl2PointCloud

```
/// @brief stl文件转为点云对象
/// @param path 文件路径
/// @param pc 点云对象指针，此处使用PointCloudSharp类的PointCloudPointer属性
public static extern void stl2PointCloud([MarshalAs(UnmanagedType.LPStr)] string
path, IntPtr pc);
```

3.1.5 savePcdFile

```
/// @brief 保存pcd文件
/// @param path 保存的路径
/// @param pc 点云对象指针，此处使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @param binaryMode 是否保存为二进制文件，若参数大于0，则将点云文件保存为二进制文件，反之亦然
/// @note 如果需要查看保存的点云文件数据，则binaryMode参数应该小于或等于0；如果需要提高文件读取速度，binaryMode应大于0
public static extern void savePcdFile([MarshalAs(UnmanagedType.LPStr)] string
path, IntPtr pc, int binaryMode);
```

3.1.6 savePlyFile

```
/// @brief 保存ply文件
/// @param path 保存的路径
/// @param pc 点云对象指针，此处使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @param binaryMode 是否保存为二进制文件，若参数大于0，则将点云文件保存为二进制文件，反之亦然
/// @note 如果需要查看保存的点云文件数据，则binaryMode参数应该小于或等于0；如果需要提高文件读取速度，binaryMode应大于0
public static extern void savePlyFile([MarshalAs(UnmanagedType.LPStr)] string path, IntPtr pc, int binaryMode);
```

3.2 Filter

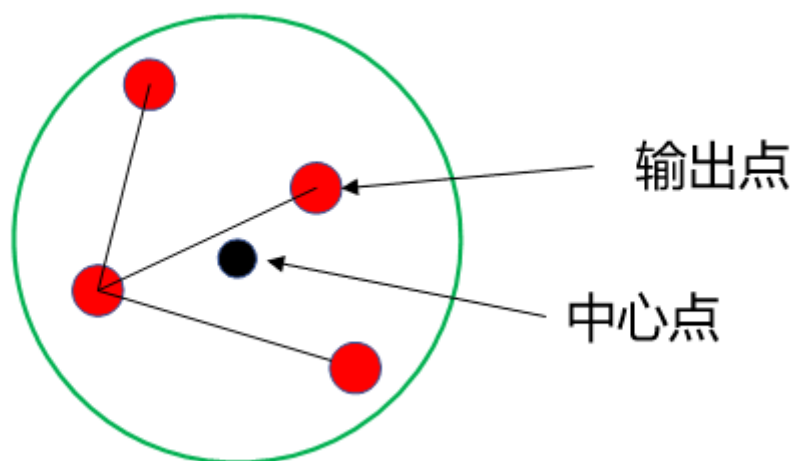
Filter模块主要实现点云的滤波功能，目前封装了均匀下采样、统计滤波、直通滤波、半径滤波等功能

3.2.1 uniformDownSample

```
/// @brief 对点云进行均匀下采样，可有效减少点云数量
/// @param in_pc 输入的点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @param radius 下采样半径，该值越大，采样后点云越稀疏
/// @param out_pc 采样后的点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @note 在DD马达端面跳动测量中，radius建议取100
public static extern void uniformDownSample(IntPtr in_pc, double radius, IntPtr out_pc);
```

算法原理：

通过构建指定半径的球体对点云进行下采样滤波，将每一个球内距离球体中心最近的点作为下采样之后的点输出。



3.2.2 staFilter

```

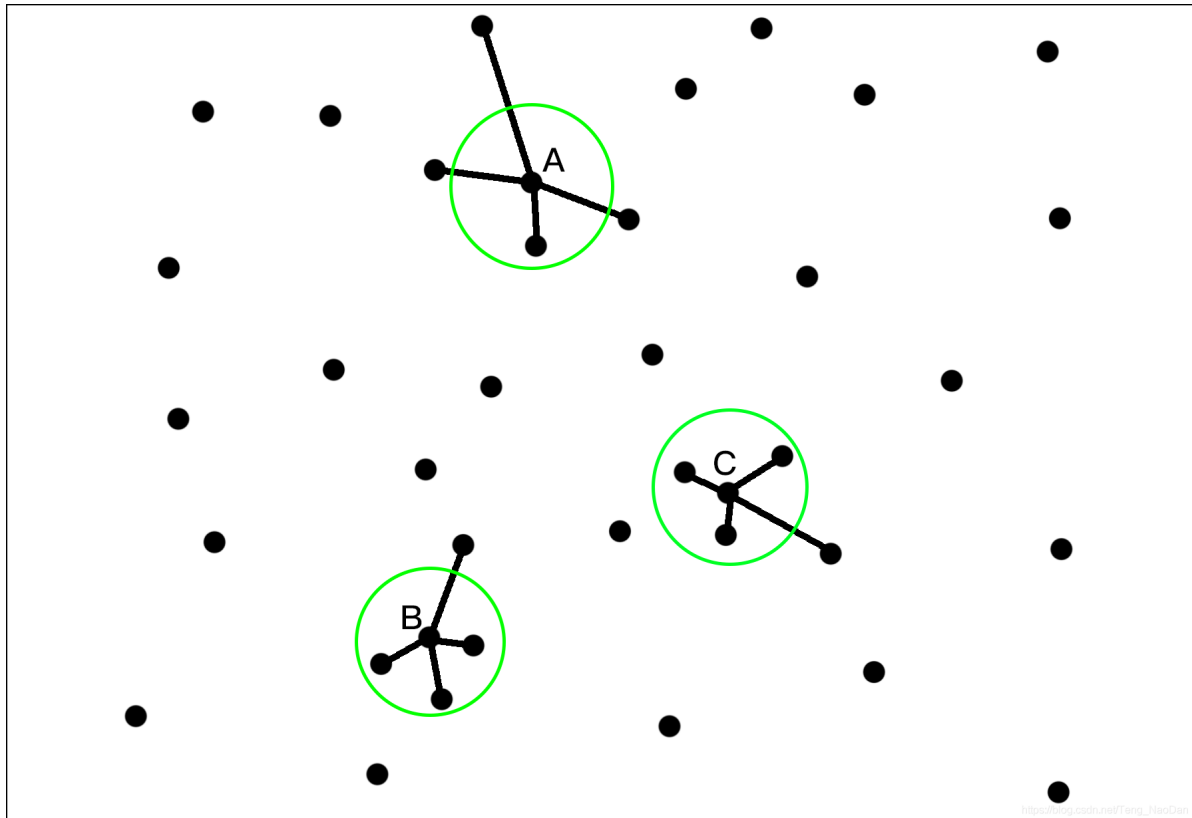
/// @brief 对点云进行统计滤波，可去除离群点
/// @param in_pc 输入的点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @param neighbor_num 指定一个点的邻居数目
/// @param thresh 统计滤波阈值
/// @param out_pc 结果点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @note 点的平均距离在 $[\mu - \alpha \times \sigma, \mu + \alpha \times \sigma]$ 之外的点被剔除，thresh即是 $\alpha$ ，默认为1。
///  $\mu$ 和 $\sigma$ 分别是整个点云距离的均值和标准差。在DD马达端面跳动测量中，neighbor_num建议取50，
thresh建议取1
public static extern void staFilter(IntPtr in_pc, int neighbor_num, float
thresh, IntPtr out_pc);

```

算法原理：

逐点计算统计点与邻域中neighbor_num个点的平均距离。假设得到的距离的分布为高斯分布，就可以得到均值 μ 和标准差 σ ，以 $\mu + \text{std_mul} \times \sigma$ 为距离阈值，距离在区间之外的点便视为离群点。其中std_mul是用户指定的标准差倍数的阈值，即参数中的thresh。

如下图所示，圆形的半径为距离阈值，则A点邻域有3个点被视为离群点，B、C点邻域各有1个点被视为离群点。



3.2.3 passThroughFilter

```

/// @brief 对点云进行直通滤波处理
/// @details 直通滤波是一种依赖于坐标的滤波方法。通过指定要滤波的轴，并设置要过滤的大小区间来进行滤波
/// @param in_pc 输入的点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @param axis_name 指定对哪个轴过滤，只有三个参数供选择，分别是'x','y'和'z'。字母应小写
/// @param limit_min 过滤区间的最小值
/// @param limit_max 过滤区间的最大值
/// @param negative 指定是否反向过滤。negative大于0时，说明是反向过滤，即剔除[limit_min,limit_max]区间之内的点，反之亦然
/// @param out_pc 结果点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @note 一般来说，建议将negative设置为0，不执行反向过滤，即只剔除[limit_min,limit_max]区间之外的点
public static extern void passThroughFilter(IntPtr in_pc,
[MarshalAs(UnmanagedType.LPStr)] string
axis_name,float limit_min, float limit_max, int negative, IntPtr out_pc);

```

3.2.4 sigamFilter

```

/// @brief sigam法则剔除异常值
/// @param in_pc 输入的点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @param sigam_thresh sigam系数值，一般为3
/// @param out_pc 结果点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @note 在DD马达端面跳动测量中，sigam_thresh建议取2
public static extern void sigamFilter(IntPtr in_pc, int sigam_thresh, IntPtr out_pc);

```

3.2.5 radiusFilter

```

/// @brief 半径滤波，可去除离群点
/// @param in_pc 输入的点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @param radius 指定半径范围
/// @param num_thresh 在指定半径范围内的个数阈值。
/// @param out_pc 结果点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @note 半径滤波思路很简单，首先指定某点的半径范围，然后计算在该半径范围内的近邻点数目
/// 若该数目小于指定的num_thresh，则剔除该点。该滤波剔除稀疏的点云效果好
public static extern void radiusFilter(IntPtr in_pc, double radius, int num_thresh, IntPtr out_pc);

```

3.3 Segmentation

Segmentation模块主要封装了点云的分割功能，目前主要封装了区域生长和欧式分割等功能。

3.3.1 oriGrowRegion

```

/// @brief 原始区域生长
/// @param in_pc 输入的点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @param neighbor_num 指定一个点的邻居数目
/// @param smooth_thresh 两法向量夹角阈值
/// @param curva_thresh 曲率阈值
/// @param MinClusterSize 成为一个点云簇的最小点数
/// @param MaxClusterSize 成为一个点云簇的最大点数
/// @param out_pc 点云簇向量指针，存储了各个点云簇的索引。使用PointIndices类的
PointIndicesPointer属性
/// @note 获得了out_pc指针之后，里面只是存储了点云簇的索引，并不是真实的点云，所以还需要使用
Util类中的
copyPcBaseOnIndice函数将对应的点云簇复制出来
public static extern void oriGrowRegion(IntPtr in_pc, int neighbor_num, float
smooth_thresh,
float curva_thresh, int MinClusterSize,
int MaxClusterSize,
IntPtr out_Indices);

```

算法概述：

该算法的目的是合并平滑约束条件下足够接近的点。因此，该算法的输出数据结构是由聚类组成的数组，其中每个聚类都是被认为是同一光滑表面的一部分的点的集合。该分割算法适合于小曲率的阶梯面分割。

算法流程：

首先将曲率最小的点作为初始种子点，然后开始区域生长，过程如下：

- \1. 选中的点添加到种子集合中
- \2. 对于每一个种子点，找到它的邻近点。这里需要设置的就是neighbor_num参数，设置一个点的近邻数
- \3. 算出每个相邻点的法线和当前种子点的法线之间的角度，如果角度小于smooth_thresh阈值，认为这两点在一个平面上，则将当前点添加到当前区域。
- \4. 然后计算每个邻居点的曲率值，如果曲率小于阈值curva_thresh，那这个点将被添加到种子集合中进行下一步测试。
- \5. 将当前的种子从种子列表中移除。

一直重复上述步骤，直到种子列表变成空的，则区域生长结束。

3.3.2 modifiedGrowRegion

```

/// @brief 修改了的区域生长，这个函数将会直接返回点数最多的点云簇
/// @details 这个函数内部做了一次点云簇筛选，只选取点云数最多的点云簇作为输出，所以它的输出是
一个点云对象指针
/// 即PointCloudXYZ类的PointCloudXYZPointer属性
/// @param in_pc 输入的点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @param neighbor_num 指定一个点的邻居数目
/// @param smooth_thresh 两法向量夹角阈值
/// @param curva_thresh 曲率阈值
/// @param MinClusterSize 成为一个点云簇的最小点数
/// @param MaxClusterSize 成为一个点云簇的最大点数
/// @param out_pc 结果点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @note 在DD马达端面跳动测量中，neighbor_num建议取50，smooth_thresh和curva_thresh建议
取1
/// MinClusterSize建议取100，MaxClusterSize建议取5000000

```



```

public static extern void modifiedGrowRegion(IntPtr in_pc, int neighbor_num,
float smooth_thresh,
float curva_thresh, int
MinClusterSize, int MaxClusterSize,
IntPtr out_pc);

```

3.3.3 euclideanCluster

```

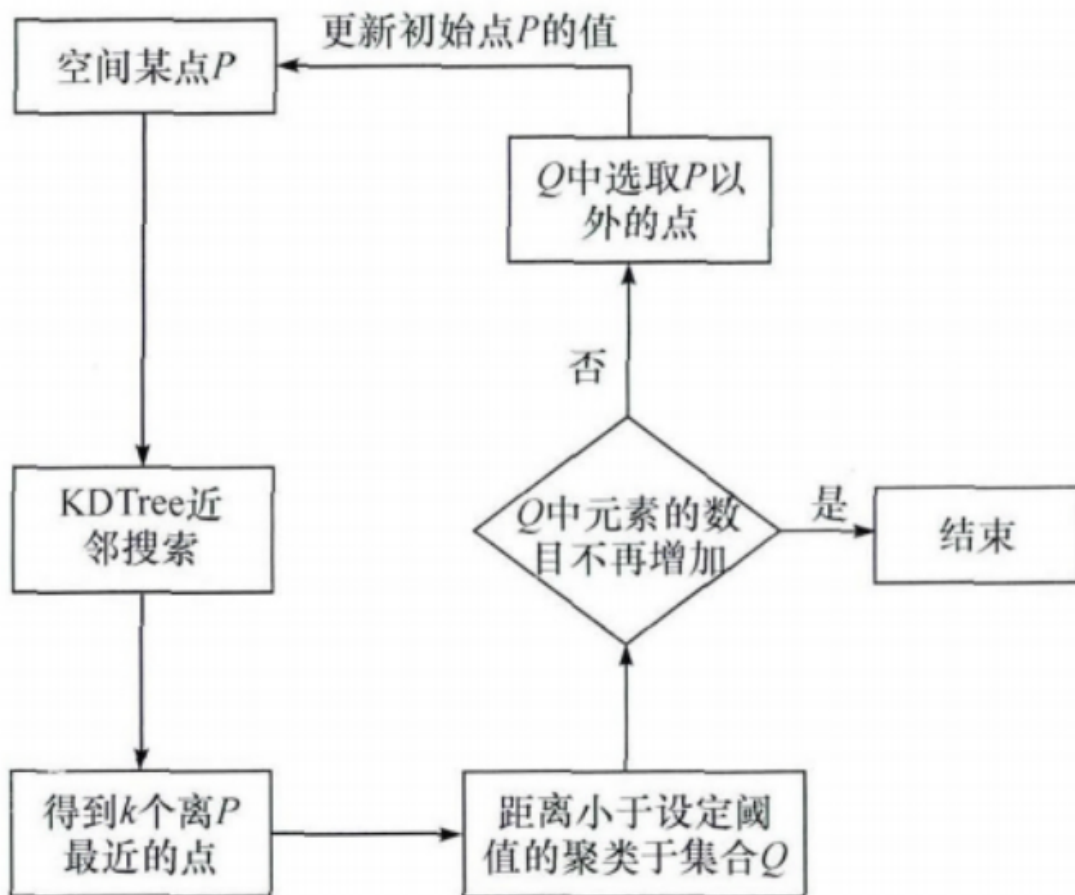
/// @brief 欧式聚类
/// @param in_pc 输入的点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @param distance_thresh 欧式聚类的距离阈值，欧式聚类小于这个距离的点被归为一类
/// @param MinClusterSize 成为一个点云簇的最小点数
/// @param MaxClusterSize 成为一个点云簇的最大点数
/// @param out_pc 点云簇向量指针，存储了各个点云簇的索引。使用PointIndices类的
PointIndicesPointer属性
/// @note 获得了out_pc指针之后，里面只是存储了点云簇的索引，并不是真实的点云，所以还需要使用
Util类中的
copyPcBaseOnIndices函数将对应的点云簇复制出来。
/// @attention 该算法时间复杂度有点高
public static extern void euclideanCluster(IntPtr in_pc, double distance_thresh,
int MinClusterSize,
int MaxClusterSize, IntPtr
out_Indices);

```

算法流程：

- 1 找到空间中某点p10，有kdTree找到离他最近的n个点，判断这n个点到p的距离。将距离小于阈值r的点p12,p13,p14...放入Q中；或者找距离他指定范围内的所有点，放入Q中。
- 2 在 Q(p10) 里找到一点p12,重复1
- 3 在 Q(p10,p12) 找到一点，重复1，找到p22,p23,p24...全部放进Q里

4 当 Q 再也不能有新点加入了，则完成搜索了；



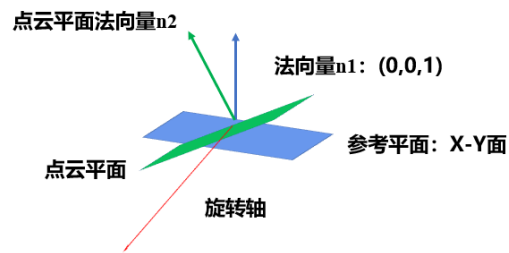
3.4 Util

Util模块主要封装了一些实用的工具函数，比如校正平面、根据点云索引复制点云

3.4.1 correctPlane

```
/// @brief 将点云校正到和法向量为 (0,0,1) 平面平行
/// @param in_pc 输入的点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @param normal 待校正平面的方程系数
/// @param out_pc 结果点云对象指针，使用PointCloudXYZ类的PointCloudXYZPointer属性
public static extern void correctPlane(IntPtr in_pc, float[] normal, IntPtr out_pc);
```

算法原理:



目的:

将点云平面旋转到和参考平面平行

步骤:

旋转前法向量为n2, 旋转后法向量为n1 (0,0,1)。

(1)把两个向量单位化, 再通过点积可直接求得向量的夹角 θ ;

(2)对n1、n2两向量叉积求得垂直于这两个向量所在平面的法向量, 即旋转轴, 将旋转轴单位化得到单位向量k。

(3)最后将夹角 θ 和旋转轴的单位向量k代入罗德里格斯旋转公式, 即可得到对应的旋转矩阵。

(4)最后利用得到的旋转矩阵对点云平面作旋转变换

$$T = \begin{vmatrix} \cos \theta + k_x^2(1 - \cos \theta) & k_x k_y(1 - \cos \theta) - k_z \sin \theta & k_y \sin \theta + k_x k_z(1 - \cos \theta) \\ k_z \sin \theta + k_x k_y(1 - \cos \theta) & \cos \theta + k_y^2(1 - \cos \theta) & -k_x \sin \theta + k_y k_z(1 - \cos \theta) \\ -k_y \sin \theta + k_x k_z(1 - \cos \theta) & k_x \sin \theta + k_y k_z(1 - \cos \theta) & \cos \theta + k_z^2(1 - \cos \theta) \end{vmatrix}$$

3.4.2 copyPcBaseOnIndice

```
/// @brief 根据点云的索引复制点云
/// @param in_pc 输入的目标点云
/// @param in_indice 输入的点云索引指针, 它应该是从PointIndices类中的GetPointIndices函数
/// 中获取。
/// 这指定了你要从in_pc中复制的点云索引
/// @param out_pc 返回复制好的点云
/// @note 这个函数应该与oriGrowRegion配合使用
public static extern void copyPcBaseOnIndice(IntPtr in_pc, IntPtr in_indice,
IntPtr out_pc);
```

3.5 SampleConsensus

SampleConsensus模块主要封装了一些基于随机抽样一致性的算法, 比如基于Ransac的点云平面拟合等

3.5.1 fitPlane

```
/// @brief 使用Ransac算法拟合点云平面
/// @param in_pc 输入的点云对象指针, 使用PointCloudXYZ类的PointCloudXYZPointer属性
/// @param distance_thresh 距离阈值
/// @param max_itera 最大迭代次数
/// @param normal 返回拟合平面的一般方程系数, 依次包含a、b、c、d四个值。方程形式为
ax+by+cz+d=0
/// @return 返回的是拟合的点云平面与法向量是 (0,0,1)平面的夹角
/// @note 在DD马达端面跳动测量中, distance_thresh建议取10, max_itera建议取100
public static extern float fitPlane(IntPtr in_pc, float distance_thresh, int
max_itera, float[] normal);
```

算法原理:

- (1) 首先, 随机选择两个点确定一个直线方程。
- (2) 通过这两个点, 我们可以计算出这两个点所表示的模型方程 $y=ax+b$ 。
- (3) 我们将所有的数据点套到这个模型中计算误差, 这里的误差可以是每个点到直线的距离之和。

(4) 我们找到所有小于误差阈值的点，也就是内点。从下图可以看到，蓝色的点是内点，红色的点是外点。

(5) 然后我们再重复 (1) ~ (4) 这个过程，直到达到一定迭代次数后，我们选出内点数目最多的模型，作为问题的解。