1滤波器在ROS中的实现

size_t是为了适应不同的系统设计的，在32位的系统中，size_t是4个字节的，但是在64位的系统中，size_t是8个字节的

pcl_conversions.h中提供了从pcl数据类型到ros的消息类型的转换

makeShared()函数：

一般用于点云的复制时，如果直接使用=，那么就会出现两个指针指向同一个目标，`makeShared()`的说明为：该函数返回的虽然是一个智能指针，但是该智能指针指向的是一个深度复制的原对象，因此需要尽量避免对非空点云进行该操作。并且因为是深度复制，对新指针的操作都不会对原点云产生影响。指向原来的目标的智能指针的个数不会发生变化。

统计滤波器

```
统计离群值算法滤波
需要头文件：pcl/filters/statistical_outlier_removal.h
定义：pcl::StatisticalOutlierRemoval<pcl::PointXYZ> statFilter;
statFilter.setInputCloud(cloud.makeShared());
statFilter.setMeanK(40);//设置均值
statFilter.setStddevMulThresh(4);//方差
statFilter.filter(cloud_filtered);//输出结果到点云
```

体素栅格滤波器

```
需要头文件pcl/filters/voxel_grid.h
pcl::VoxelGrid<pcl::PointXYZ> voxelSampler;
voxelSampler.setInputCloud(cloud.makeShared());
voxelSampler.setLeafSize(10.f,10.f,10.f);
voxelSampler.filter(cloud_downsampled);//输出点云结果
```

半径滤波器

```
pcl::RadiusOutlierRemoval<pcl::PointXYZ> radiusfilter;
radiusfilter.setInputCloud(cloud.makeShared());
radiusfilter.setRadiusSearch(5);
radiusfilter.setMinNeighborsInRadius(20);
radiusfilter.filter(cloud_radius);
```

双边滤波器

```
    pcl::PointCloud<pcl::PointXYZI>::Ptr cloud_filtered(new
pcl::PointCloud<pcl::PointXYZI>);

    // ----------------------建立kdtree----------------------
    pcl::search::KdTree<pcl::PointXYZI>::Ptr tree(new
pcl::search::KdTree<pcl::PointXYZI>);//用一个子类作为形参传入
    //----------------------双边滤波----------------------
    pcl::BilateralFilter<pcl::PointXYZI> bf;
    bf.setInputCloud(cloud);
    bf.setSearchMethod(tree);
    bf.setHalfSize(0.1);   // 设置高斯双边滤波窗口的一半大小。
    bf.setStdDev(0.03);  // 设置标准差参数
    bf.filter(*cloud_filtered);
```

2 点云的配准

定义：由于三维扫描仪设备受到测量方式和被测物体形状的条件限制，一次扫描往往只能获取到局部的点云信息，进而需要进行多次扫描，然后每次扫描时得到的点云都有独立的坐标系，不可以直接进行拼接。在逆向工程、计算机视觉、文物数字化等领域中，由于点云的不完整、旋转错位、平移错位等，使得要得到完整点云就需要对多个局部点云进行配准。为了得到被测物体的完整数据模型,需要确定一个合适的坐标变换，将从各个视角得到的点集合并到一个统一的坐标系下形成一个完整的数据点云，然后就可以方便地进行可视化等操作，这就是点云数据的配准。

整个包在**registration**中

2.1 点云的配准方法

点云配准步骤上可以分为粗配准（Coarse Registration）和精配准（Fine Registration）两个阶段。

粗配准是指在点云相对位姿完全未知的情况下对点云进行配准，找到一个可以让两块点云相对近似的旋转平移变换矩阵，进而将待配准点云数据转换到统一的坐标系内，可以为精配准提供良好的初始值。常见粗配准算法：

- 粗配准
  - PCL 4PCS算法实现点云配准
  - PCL K4PCS算法实现点云配准
  - PCL 改进的RANSAC算法实现点云粗配准
  - PCL SAC-IA 初始配准算法
  - PCL 刚性目标的鲁棒姿态估计
  - PCA 实现点云粗配准(C++版)

精配准是指在粗配准的基础上，让点云之间的空间位置差异最小化，得到一个更加精准的旋转平移变换矩阵。该算法的运行速度以及向全局最优化的收敛性却在很大程度上依赖于给定的**初始变换估计**以及在迭代过程中**对应关系的确立**。所以需要各种粗配准技术为ICP算法提供较好的位置，在迭代过程中确立正确对应点集能避免迭代陷入局部极值，决定了算法的收敛速度和最终的配准精度。最常见的精配准算法是ICP及其变种。

- ICP 迭代最近点算法（Iterative Cloest Point）
  - GICP
  - NICP
  - MBICP
- NDT 正态分布变换算法（Normal Distributions Transform）

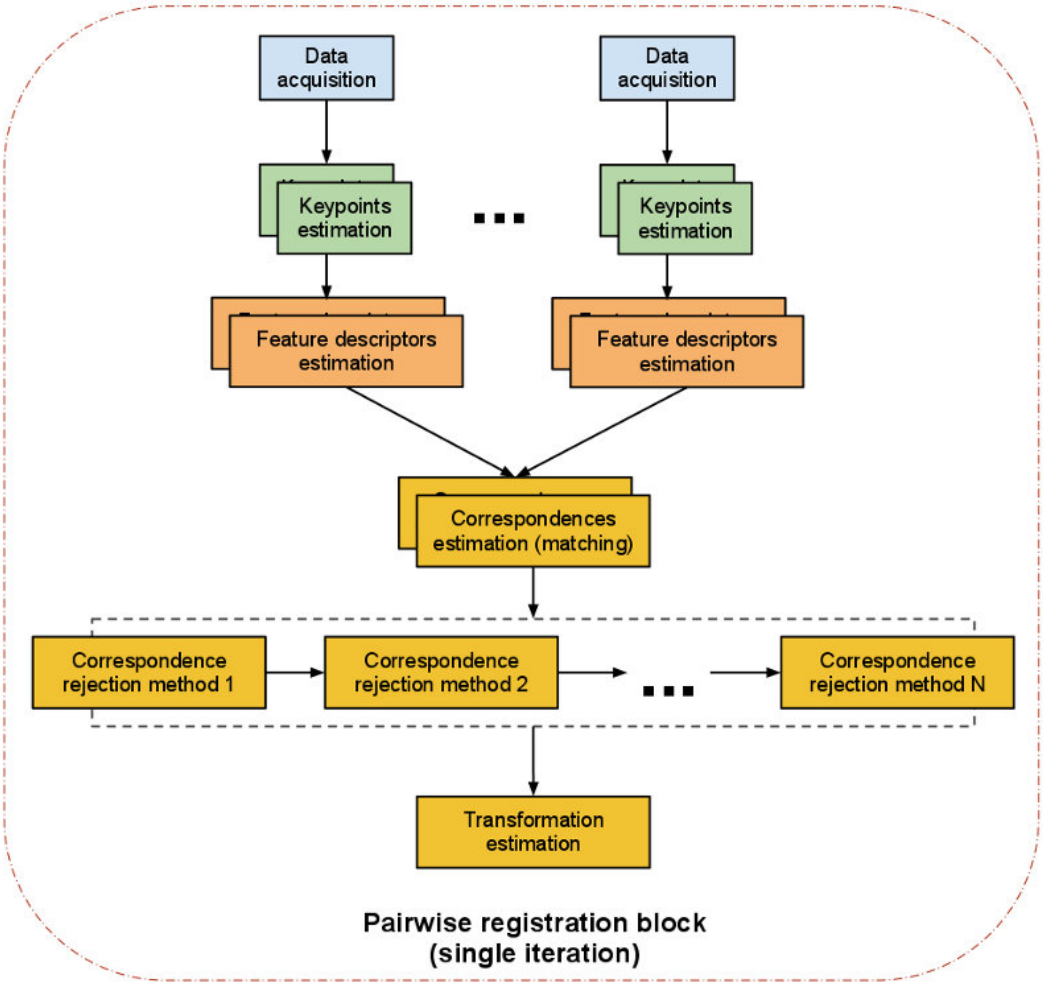
2.2 自动配准技术

通常所说的点云配准就是指自动配准，点云自动配准技术是通过一定的算法或者统计学规律，利用计算机计算两块点云之间的错位，从而达到把两片点云自动配准的效果。本质上就是把不同坐标系中测量得到的数据点云进行坐标变换，从而得到整体的数据模型。

即求得坐标变换参数 R（旋转矩阵）和 T（平移向量）,使得两视角下测得的三维数据经坐标变换后的距离最小。配准算法按照实现过程可以分为整体配准和局部配准。

两两配准¶

两两配准（pairwise registration）：我们称一对点云数据集的配准问题为两两配准（pairwise registration）。通常通过应用一个估算得到的表示平移和旋转的 4 × 4 刚体变换矩阵来使一个点云数据集精确地与另一个点云数据集(目标数据集)进行完美配准。



1. 首先从两个数据集中按照同样的关键点选取标准，提取关键点。

2. 对选择的所有关键点分别计算其特征描述子。
3. 结合特征描述子在两个数据集中的坐标的位置，以两者之间特征和位置的相似度为基础，估算它们的对应关系，初步估计对应点对。
4. 假定数据是有噪声的，除去对配准有影响的错误的对应点对。
5. 利用剩余的正确对应关系来估算刚体变换，完成配准。

整个配准过程最重要的是关键点的提取以及关键点的特征描述，以确保对应估计的准确性和效率，这样才能保证后续流程中的刚体变换矩阵估计的无误性。接下来我们对单次迭代的每一步进行解读：

3 点云的分割

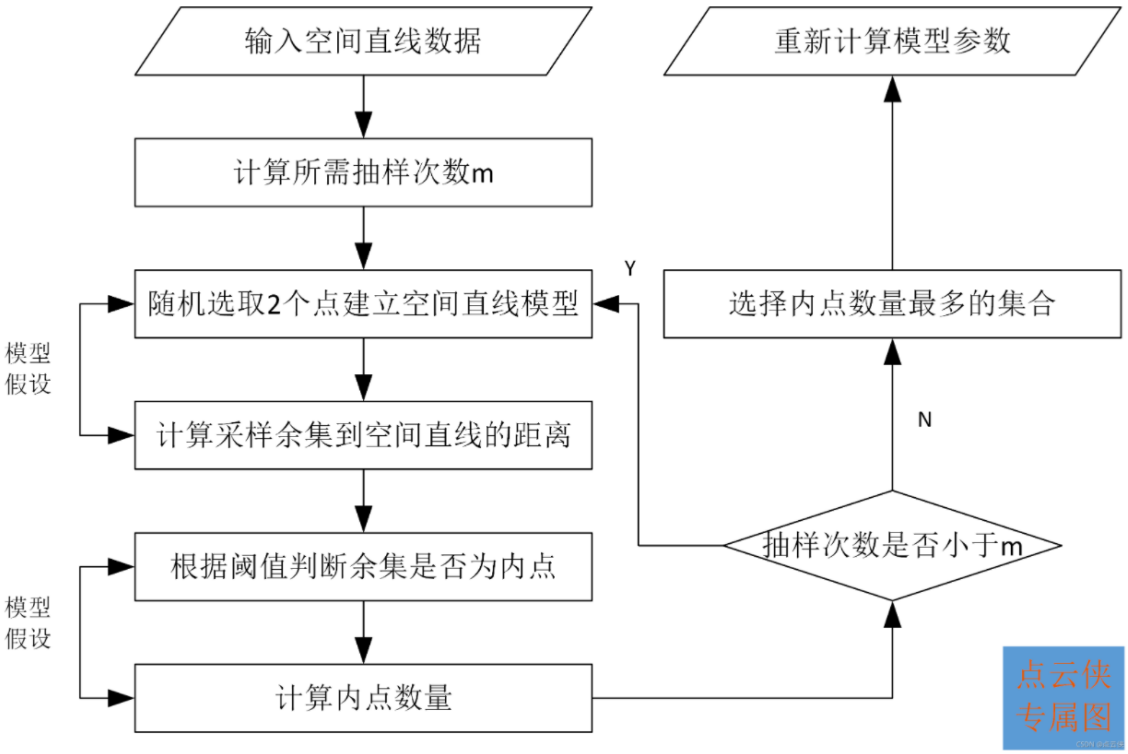
点云分割是根据空间、几何和纹理等特征对点云进行划分，使得同一划分区域内的点云拥有相似的特征。 点云的有效分割往往是许多应用的前提。例如，在逆向工程CAD/CAM 领域，对零件的不同扫描表面进行分割，然后才能更好地进行孔洞修复、曲面重建、特征描述和提取，进而进行基于 3D内容的检索、组合重用等。在激光遥感领域，同样需要对地面、物体首先进行分类处理，然后才能进行后期地物的识别、重建 。

总之，分割采用分而治之的思想，在点云处理中和滤波一样属于重要的基础操作，在PCL 中目前实现了进行分割的基础架构，为后期更多的扩展奠定了基础，现有实现的分割算法是鲁棒性比较好的**Cluster聚类分割**和RANSAC**基于随机采样一致性**的分割。

3.1 RANSAC

3.1.1 RANSAC拟合直线

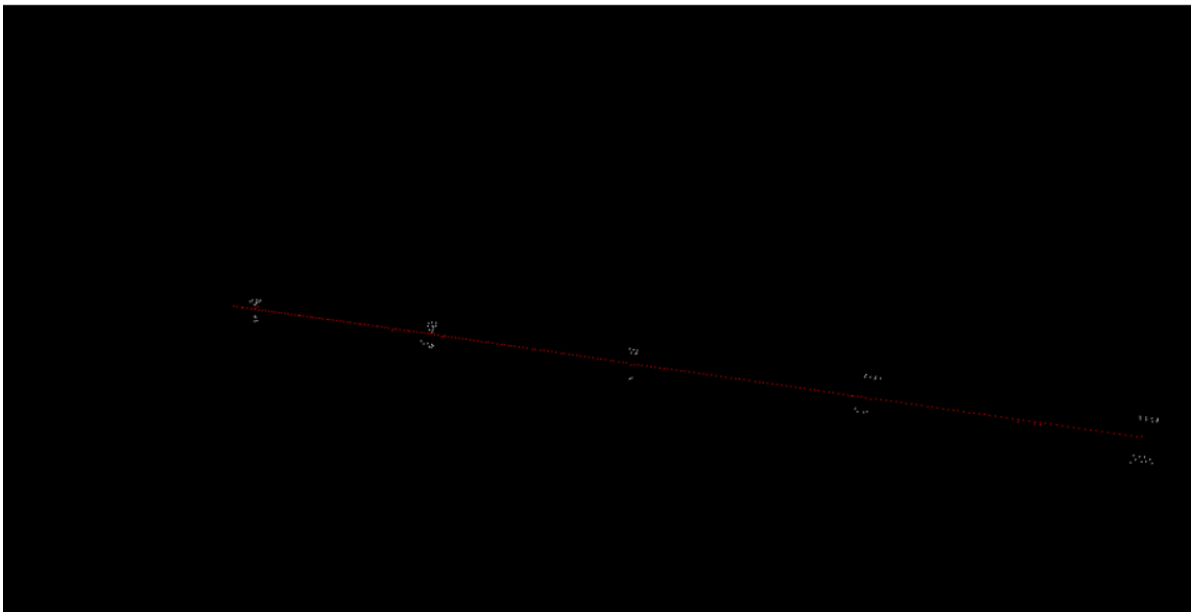
RANSAC算法由Fischler和Bolles于1981年提出，是一种从数据集合中迭代稳健估计模型参数的方法。该算法的基本思想是：不断地从数据集合中随机抽取样本集，寻求支持更多局内点的模型参数；利用模型余集检验获得的模型参数；通过一定次数的迭代，当采样样本集与合理解的一致性概率为最大时，将该采样样本集作为合理解的样本集，且参数解的正确性由样本余集检验支撑。其中数据集合中包含正确数据（内点inliers）和异常数据（外点outliers）。算法计算过程的实质为假设和检验：假设随机采样数据都为内点，利用随机采样数据计算模型参数；通过其他点对估计的模型参数进行检验。



RANSAC算法在空间直线拟合中的应用

- `point_on_line.x` ： 直线上一点的X坐标——> values[0]
- `point_on_line.y` ： 直线上一点的Y坐标——> values[1]
- `point_on_line.z` ： 直线上一点的Z坐标——> values[2]
- `line_direction.x` ： 直线方向的X坐标——> values[3]
- `line_direction.y` ： 直线方向的Y坐标——> values[4]
- `line_direction.z` ： 直线方向的Z坐标——> values[5]

- PCL RANSAC 拟合直线
- PCL RANSAC拟合分割多条直线
- PCL 计算三维空间中点到直线的距离
- PCL 角度约束的RANSAC拟合直线



3.1.2 RANSAC拟合平面

1、概述

随机抽样一致性算法RANSAC(Random sample consensus)是一种迭代的方法来从一系列包含有离异值的数据中计算数学模型参数的方法。

RANSAC算法本质上由两步组成，不断进行循环：

(1)从输入数据中随机选出能组成数学模型的最小数目的元素，使用这些元素计算出相应模型的参数。选出的这些元素数目是能决定模型参数的最少的。

(2)检查所有数据中有哪些元素能符合第一步得到的模型。超过错误阈值的元素认为是离群值（outlier),小于错误阈值的元素认为是内部点（inlier）。

这个过程重复多次，选出包含点最多的模型即得到最后的结果。

2、拟合平面

> **RANSAC具体到空间点云中拟合平面：**
> 1、从 点云 中随机选取三个点。
> 2、由这三个点组成一个平面。
> 3、计算所有其他点到该平面的距离，如果小于阈值T，就认为是处在同一个平面的点。
> 3、如果处在同一个平面的点超过n个，就保存下这个平面，并将处在这个平面上的点都标记为已匹配。
> 4、终止的条件是迭代N次后找到的平面小于n个点，或者找不到三个未标记的点。

目前，拟合平面最常见且最简单的方法是最小二乘拟合，但最小二乘拟合的精度容易受到噪点的影响。而随机采样一致算法(RANSAC)则通过迭代拟合的方法可以排除噪点的影响，拟合精度大大提高。随机采样一致算法流程如图1所示：

1. 由数学知识可知，对平面进行拟合至少需要三个点，因此首先随机选取三个点，然后根据平面方程(1)对平面模型参数A, B, C, D进行计算。

$$A \cdot x + B \cdot y + C \cdot z = D \tag{1}$$

2. 用余下的数据点去检验(1)中估计的平面模型，计算结果误差，将误差与设定的误差阈值进行比较，如果小于设定的阈值，则将该点确定为内点，统计该参数模型下内点的个数并记录。

3. 继续进行第1、2步，若当前模型的内点数量大于已经保存的最大内点数量，则更新模型参数，保留的模型参数始终是内点数量最多的模型参数。

4. 重复1~3步，不断迭代，直到达到迭代阈值，找到内点个数最多的模型参数，最后用内点再次对模型参数进行估计，从而得到最终的模型参数。

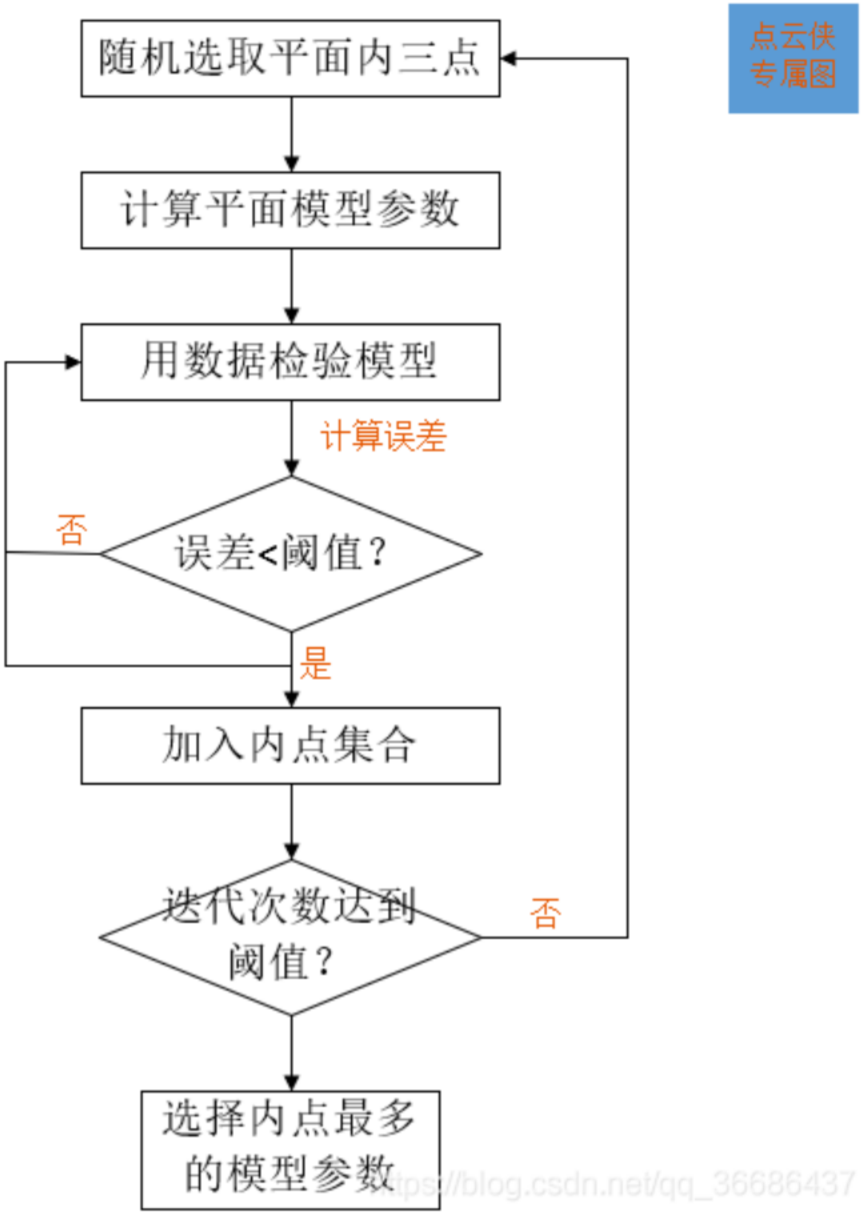
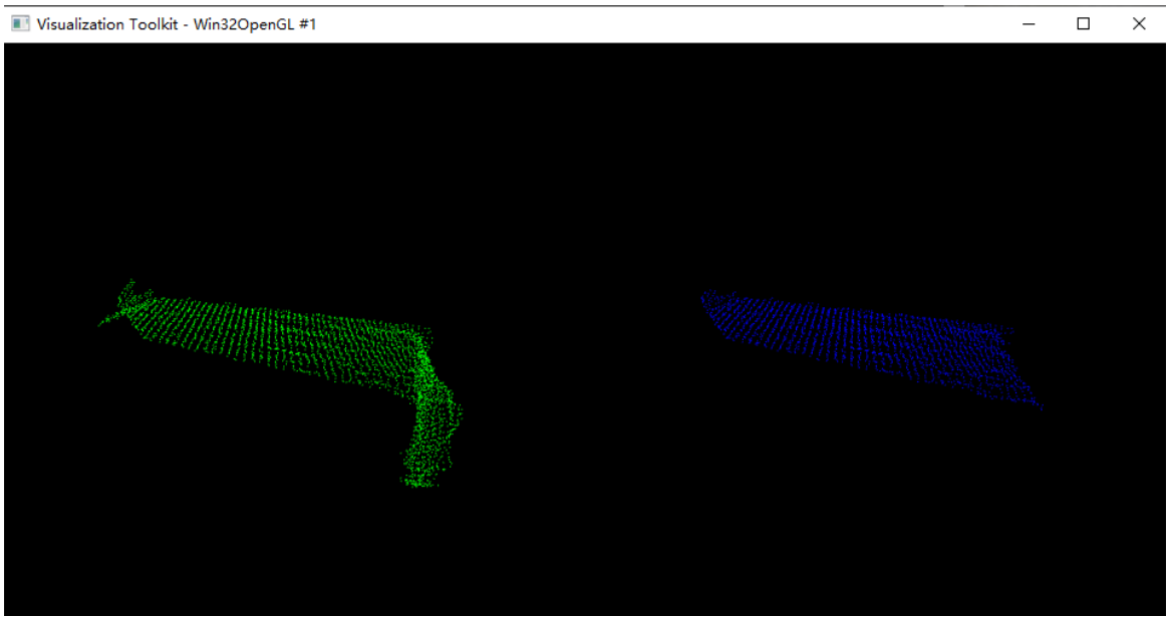


图1平面拟合算法流程图

平面模型系数定义为:

- a： 平面法线归一化的X坐标——> values[0]
- b： 平面法线归一化的Y坐标——> values[1]
- c： 平面法线归一化的Z坐标——> values[2]
- d： 平面方程的第四个黑森分量——> values[3]

> 左侧为原始点云，右侧为提取出来的平面点云



3.1.3 RANSAC拟合2D圆

2、拟合平面2D圆

圆的标准方程$(x-a)^2+(y-b)^2=r^2$中，有三个参数a、b、r，即圆心坐标为$(a，b)$，只要求出a、b、r，这时圆的方程就被确定，因此确定圆方程，须三个独立条件，其中圆心坐标是圆的定位条件，半径是圆的定形条件。

SampleConsensusModelCircle2D 定义了X-Y平面上二维圆的RANSAC拟合分割模型。

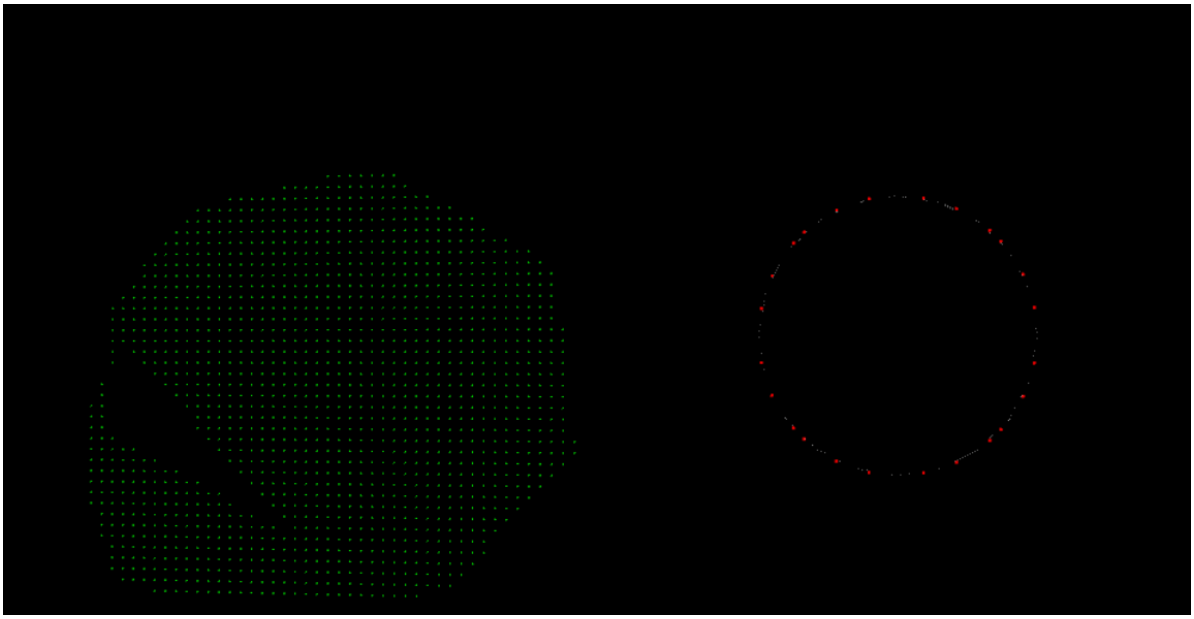
3、模型系数

二维圆模型系数定义为:

- `center.x`：圆心的X坐标——> values[0]
- `center.y`：圆心的Y坐标——> values[1]
- `radius`：圆的半径——> values[2]

下面可以用提取出的内点索引将拟合后的平面独立画出来，存在另一个点云里面

```
pcl::copyPointCloud<pcl::PointXYZ>(*cloud,inliers,*circle_cloud);
```



3.1.4 3D圆拟合

2、拟合3D圆

当球面与平面相交时，其交线是圆。 反之，空间的任何圆都可以表示成为一个球面与一个平面的交线。所以，空间圆的直角坐标方程为：

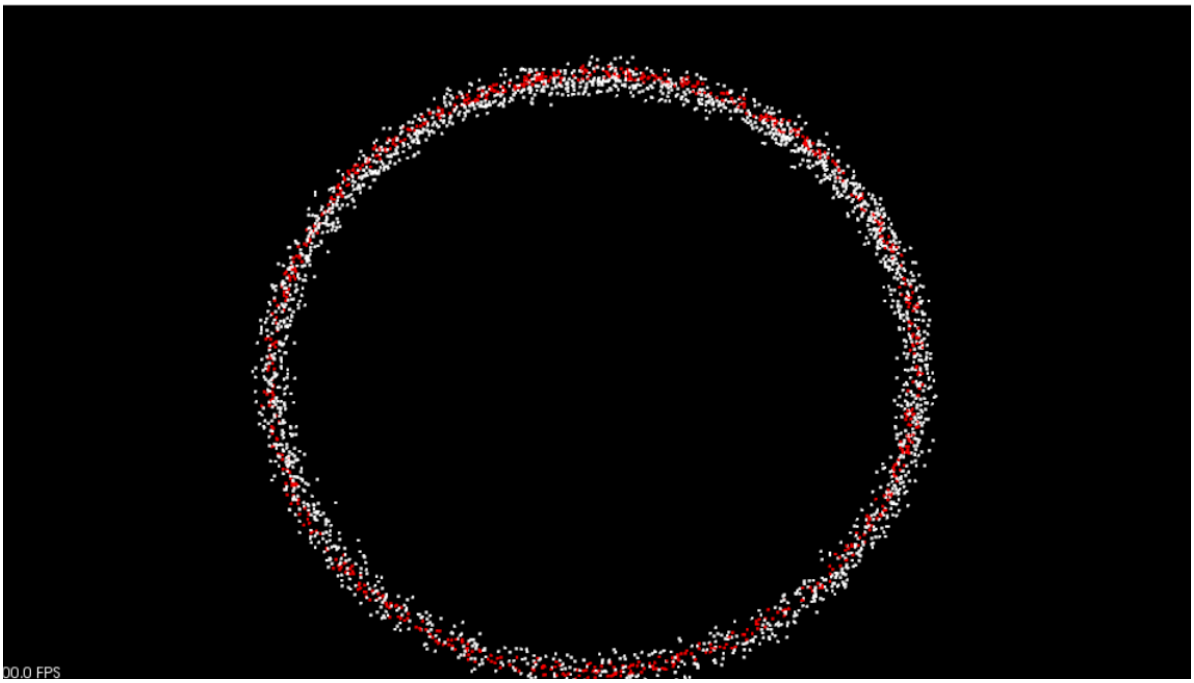
$$y = \begin{cases} (x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2 = R^2, \\ Ax + By + Cz + D = 0, \ (A^2 + B^2 + C^2 \neq 0) \end{cases} \tag{1}$$

pcl::SampleConsensusModelCircle3D定义了三维圆的RANSAC拟合分割模型。

3、模型系数

三维圆模型系数定义为:

- `center.x`： 圆心的X坐标——> values[0]
- `center.y`： 圆心的Y坐标——> values[1]
- `center.z`： 圆心的Z坐标——> values[2]
- `radius`： 圆的半径——> values[3]
- `normal.x`： 法线方向的X坐标——> values[4]
- `normal.y`： 法线方向的Y坐标——> values[5]
- `normal.z`： 法线方向的Z坐标——> values[6]



3.1.5 3D球体拟合

2、拟合球

球面的标准方程

$$(x-a)^2+(y-b)^2+(z-c)^2=r^2(r>0)$$

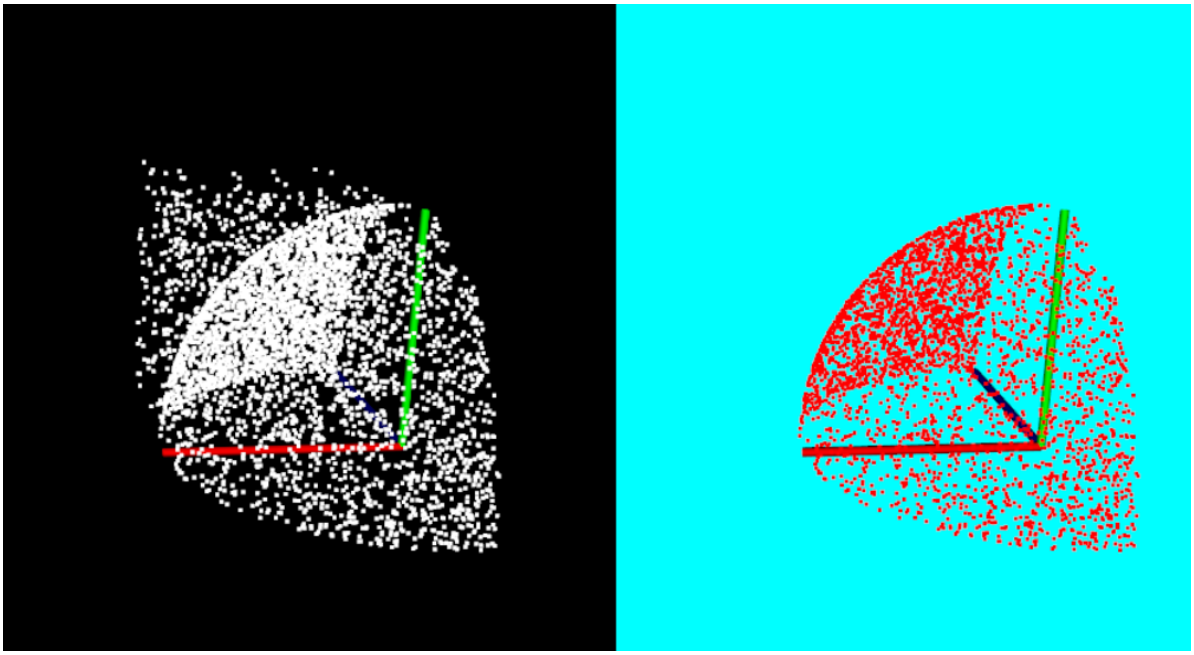
表示的球面的球心是(a,b,c),半径是r

SampleConsensusModelSphere 定义了三维球的RANSAC拟合分割模型。

3、模型系数

三纬球模型系数定义为:

- `center.x`：球心的X坐标——> values[0]
- `center.y`：球心的Y坐标——> values[1]
- `center.z`：球心的Y坐标——> values[2]
- `radius`：球的半径——> values[3]



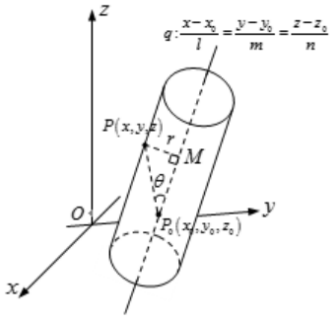
3.1.6 圆柱体拟合

1、圆柱方程参数估计

圆柱方程可以表示为：

$$(x-x_0)^2+(y-y_0)^2+(z-z_0)^2-r^2=\frac{[l(x-x_0)+m(y-y_0)+n(z-z_0)]^2}{l^2+m^2+n^2}$$

下图为空间圆柱示意图。



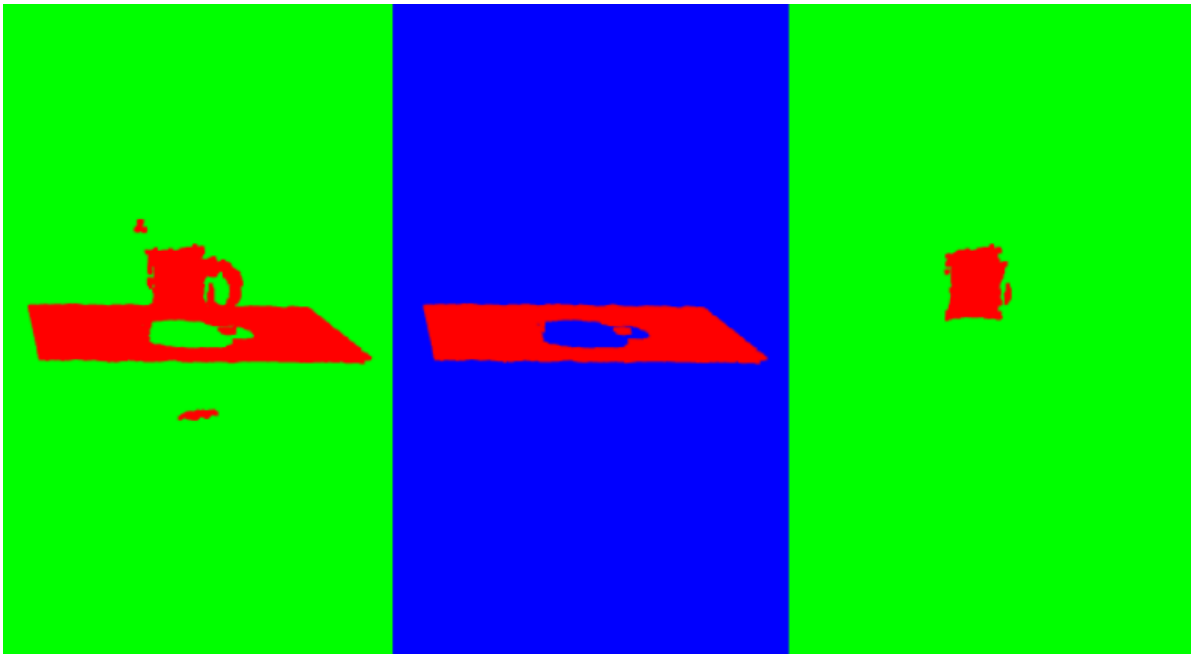
空间圆柱示意图

其中，(x_0,y_0,z_0)为圆柱轴线L上一点，(l,m,n)为圆柱轴线L方向向量，r为圆柱的半径，这七个参数可以确定一个圆柱方程。

2、模型系数

模型系数定义为:

- `point_on_axis.x`：位于圆柱轴上的点的X坐标——> values[0]
- `point_on_axis.y`：位于圆柱轴上的点的Y坐标——> values[1]
- `point_on_axis.z`：位于圆柱轴上的点的Z坐标——> values[2]
- `axis_direction.x`：圆柱的轴向X坐标——> values[3]
- `axis_direction.y`：圆柱的轴向Y坐标——> values[4]
- `axis_direction.z`：圆柱的轴向Z坐标——> values[5]
- `radius`：圆柱体的半径——> values[6]

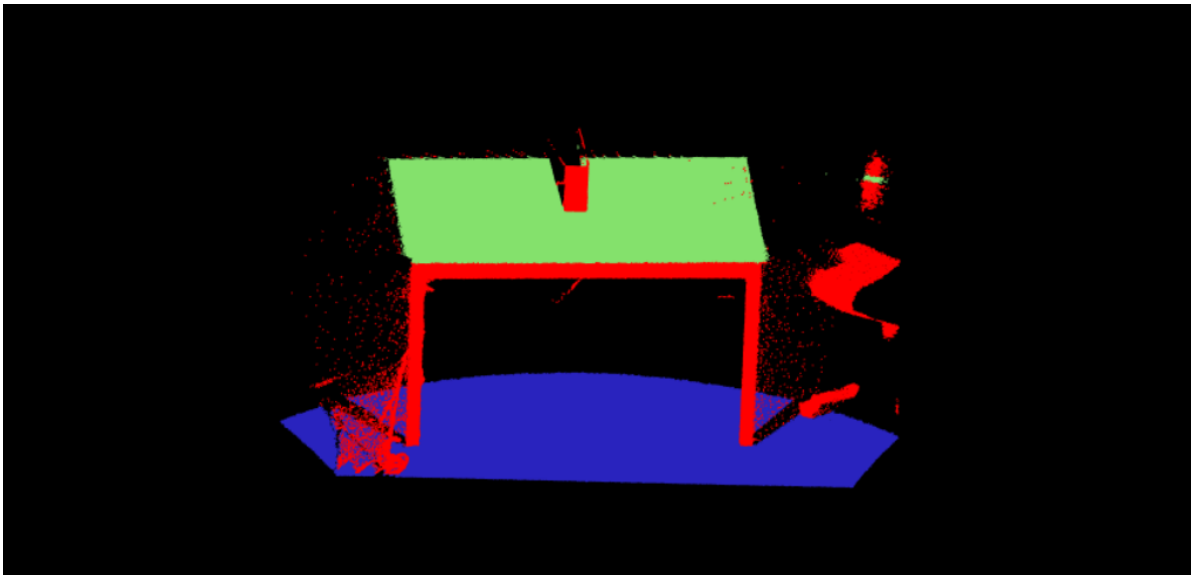


3.1.7 平面模型分割

`pcl::SACSegmentation`核心原理就是RANSAC拟合平面，换了一种代码书写方式从而实现模型提取，实际上和RANSAC拟合平面是一样的

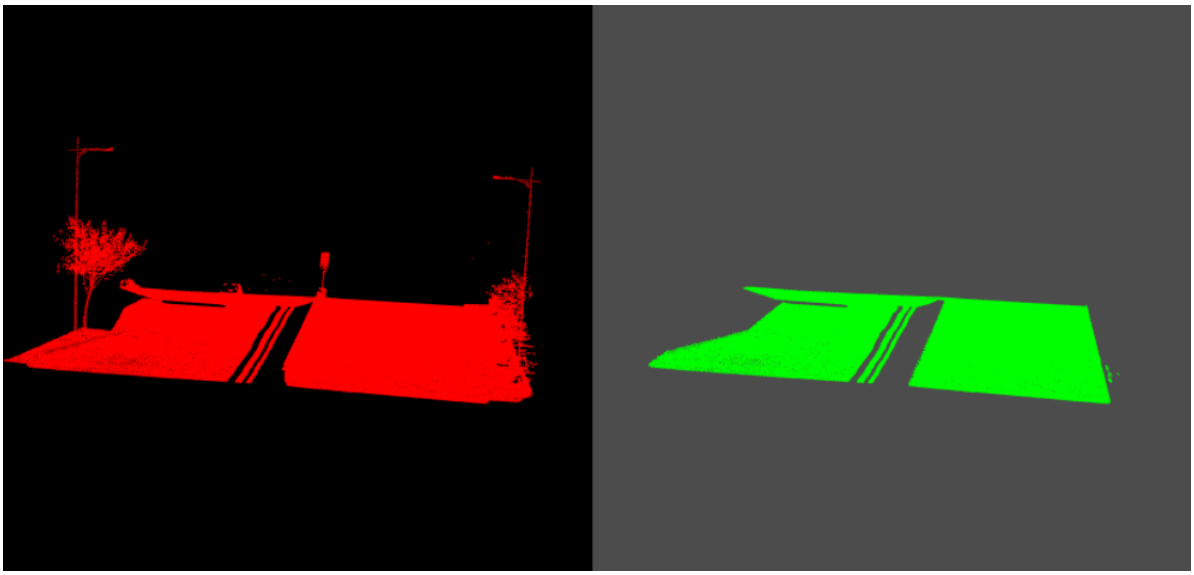
3.1.8 RANSAC分割多个平面

原理还是RANSAC拟合平面



3.1.9 分割指定阈值内的平面

拟合平面，然后计算点到平面的距离，设置阈值，将点到平面的距离在阈值范围外的点删除。



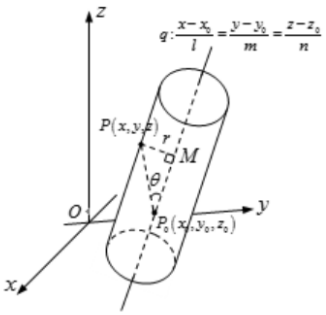
3.1.10 圆柱体模型拟合分割

1、圆柱方程参数估计

圆柱方程可以表示为：

$$(x-x_0)^2+(y-y_0)^2+(z-z_0)^2-r^2=\frac{[l(x-x_0)+m(y-y_0)+n(z-z_0)]^2}{l^2+m^2+n^2}$$

下图为空间圆柱示意图。



空间圆柱示意图

其中，(x_0,y_0,z_0)为圆柱轴线L上一点，(l,m,n)为圆柱轴线L方向向量，r为圆柱的半径，这七个参数可以确定一个圆柱方程。

2、模型系数

模型系数定义为:

- `point_on_axis.x`：位于圆柱轴上的点的X坐标——> values[0]
- `point_on_axis.y`：位于圆柱轴上的点的Y坐标——> values[1]
- `point_on_axis.z`：位于圆柱轴上的点的Z坐标——> values[2]
- `axis_direction.x`：圆柱的轴向X坐标——> values[3]
- `axis_direction.y`：圆柱的轴向Y坐标——> values[4]
- `axis_direction.z`：圆柱的轴向Z坐标——> values[5]
- `radius`：圆柱体的半径——> values[6]

3.2 常用拟合模型及使用方法

- 随机采样一致性算法（RANSAC）
- 加权采样一致性算法（MSAC）
- 最大似然一致性算法（MLESAC）
- 最小中值方差一致性算法（LMEDS）
- 渐进采样一致性（PROSAC）

3.3 最小二乘拟合平面

对于得到的n个点云数据，设拟合出的平面方程为：

$$ax + by + cz + d = 0 \tag{1}$$

约束条件为：

$$a^2 + b^2 + c^2 = 1 \tag{2}$$

可以得到平面参数 a、b、c、d。此时，要使获得的拟合平面是最佳的，就是使得k个邻近点到该平面的距离的平方和最小，即满足：

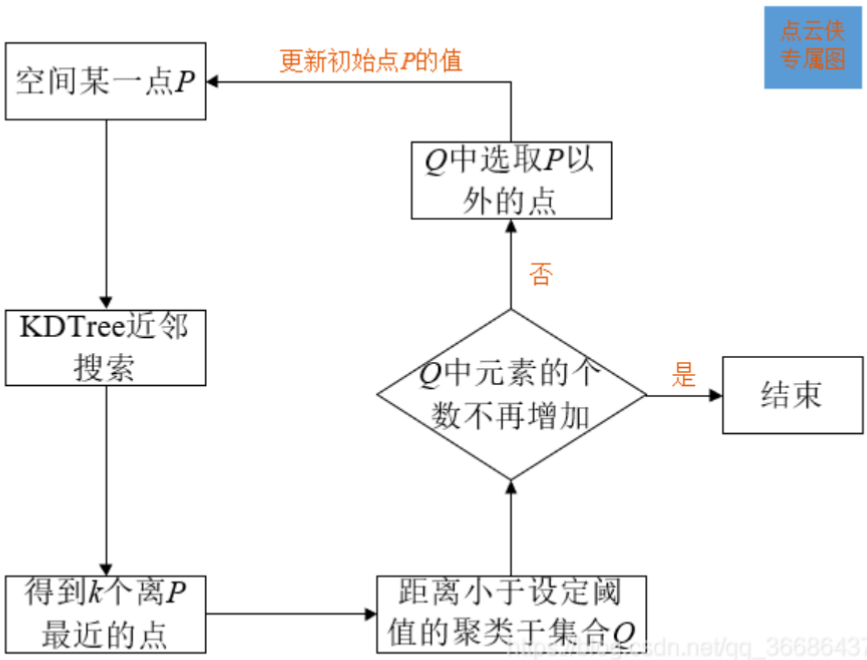
$$e = \sum_{i=1}^{n} d_i^2 \rightarrow min \tag{3}$$

式中，d_i是点云数据中的任一点$p_i(x_i, y_i, z_i)$到这个平面的距离$d_i = |ax_i + by_i + cz_i + d|$。要使$e \rightarrow min$，可以用$SVD$矩阵分解得到。

3.4 其他几何分割

3.4.1 欧式聚类分割

欧式聚类是一种基于欧氏距离度量的聚类算法。基于KD-Tree的近邻查询算法是加速欧式聚类算法的重要预处理方法。流程如下：



对于欧式聚类来说，距离判断准则为前文提到的欧氏距离。对于空间某点P，通过KD-Tree近邻搜索算法找到k个离p点最近的点，这些点中距离小于设定阈值的便聚类到集合Q中。如果Q中元素的数目不在增加，整个聚类过程便结束；否则须在集合Q中选取p点以外的点，重复上述过程，直到Q中元素的数目不在增加为止。

2、实现方法

具体的实现方法（原理是将一个点云团聚合成一类）：

- 1 找到空间中某点 p10，用kdTree找到离他最近的n个点，判断这n个点到 p10的距离。
- 将距离小于阈值 r的点p12,p13,p14…放在类Q里
- 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。

```
pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;// 欧式聚类对象
    ec.setClusterTolerance(0.02);                     // 设置近邻搜索的搜索半径为2cm
    ec.setMinClusterSize(100);                        // 设置一个聚类需要的最少的点数
目为100
    ec.setMaxClusterSize(25000);                      // 设置一个聚类需要的最大点数目
为25000
    ec.setSearchMethod(tree);                         // 设置点云的搜索机制
    ec.setInputCloud(cloud_filtered);
    ec.extract(cluster_indices);                      // 从点云中提取聚类，并将点云索
引保存在cluster_indices中
```

3.4.2 区域生长分割

1、算法流程

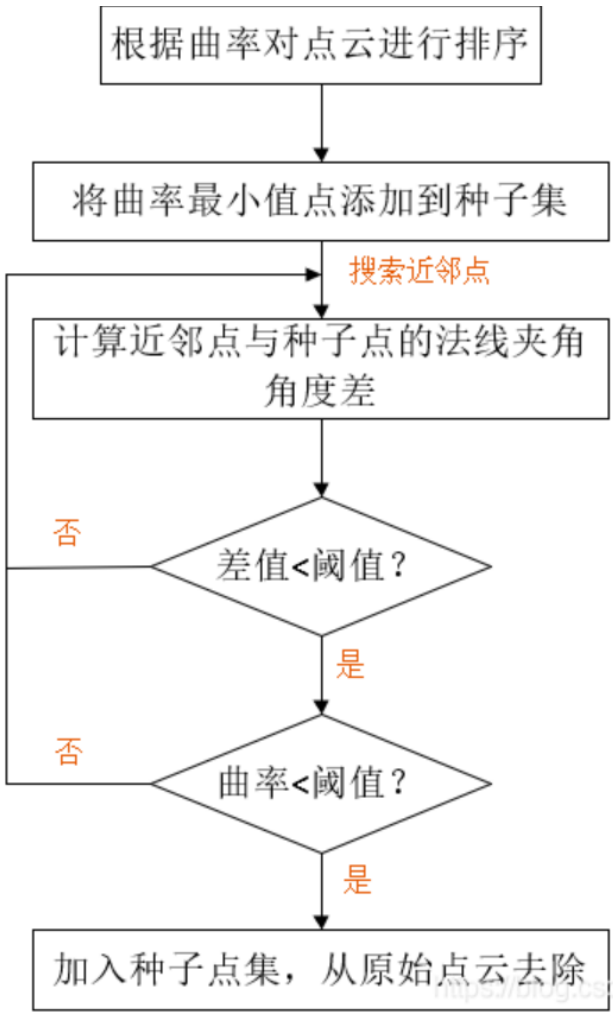
根据点的曲率值对 点云 进行排序，曲率最小的点叫做初始种子点，

（1）区域生长算法从曲率最小的种子点开始生长，初始种子点所在区域为最平滑区域，从初始种子点所在的区域开始生长可减小分割片段的总数，从而提高算法的效率。

（2）设置一空的聚类区域C和空的种子点序列Q，聚类数组L。

（3）选好初始种子点，将其加入种子点序列Q中，并搜索该种子点的领域点，计算每一个领域点法线与种子点法线之间的夹角，小于设定的平滑阀值时，将领域点加入到C中，同时判断该领域点的曲率值是否小于曲率阀值，将小于曲率阈值的领域点加入种子点序列Q中，在邻域点都判断完成后，删除当前种子点，在Q中重新选择新的种子点重复上述步骤，直到Q中序列为空，一个区域生长完成，将其加入聚类数组L中。

（4）利用曲率值从小到大排序，顺序选择输入点集 的点作为种子点加入到种子点序列中，重复以上生长的步骤。



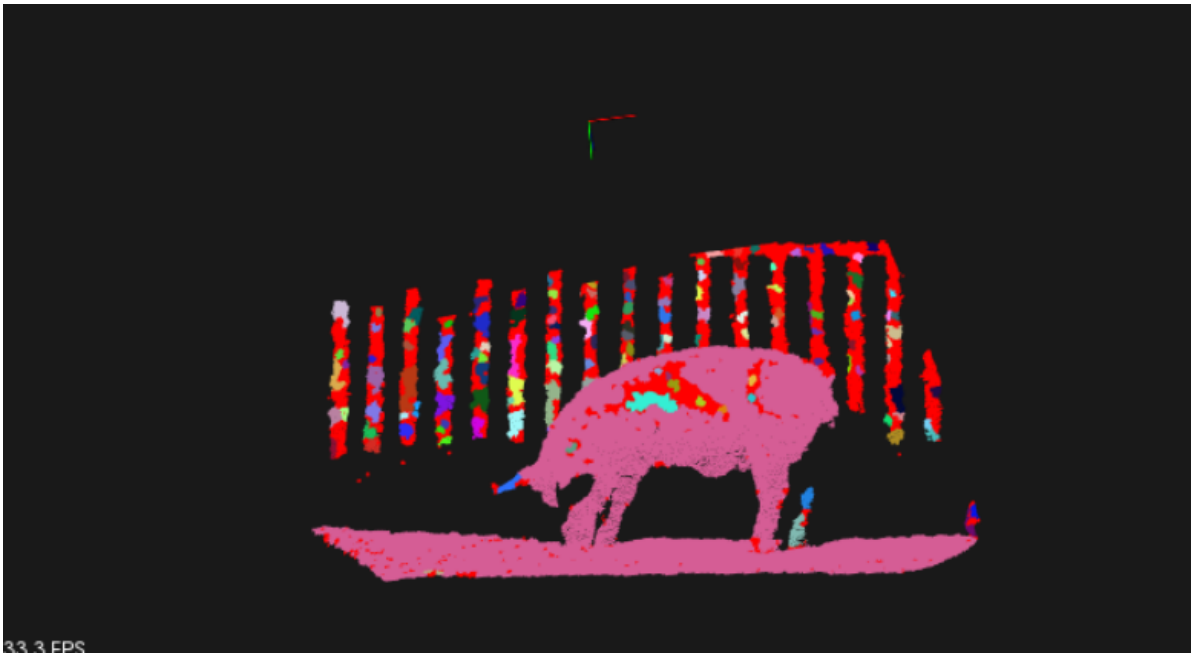
区域生长算法流程图

区域生长算法(法线差值与曲率差值)

具体实现方法：

0. 计算法线normal和曲率curvatures，依据曲率升序排序；

1. 选择曲率最低的为初始种子点，种子点周围的临近点和种子点相比较；
2. 法线的方向足够相近（法线夹角足够平滑），法线夹角阈值；
3. 曲率是否足够小（表面处在同一个弯曲程度），曲率差值阈值；
4. 如果满足2、3则该点可作做种子点；
5. 如果只满足2，则归类而不做种子。




```
//区域生长聚类分割对象 <点，法线>
    pcl::RegionGrowing<pcl::PointXYZ, pcl::Normal> reg;
    reg.setMinClusterSize(50);                              // 最小的聚类的点数
    reg.setMaxClusterSize(1000000);                         // 最大的聚类的点数
    reg.setSearchMethod(tree);                              // 搜索方式
    reg.setNumberOfNeighbours(30);                          // 设置搜索的邻域点的个数
    reg.setInputCloud(cloud);                               // 输入点云
    if (Bool_Cuting)reg.setIndices(indices);                // 通过输入参数设置，确定是否
输入点云索引
    reg.setInputNormals(normals);                           // 输入的法线
    reg.setSmoothnessThreshold(SmoothnessThreshold / 180.0 * M_PI);// 设置平滑阈
值，法线差值阈值
    reg.setCurvatureThreshold(CurvatureThreshold);      // 设置曲率的阈值

    std::vector <pcl::PointIndices> clusters;
    reg.extract(clusters);                                  // 获取聚类的结果，分割结果保
存在点云索引的向量中
```

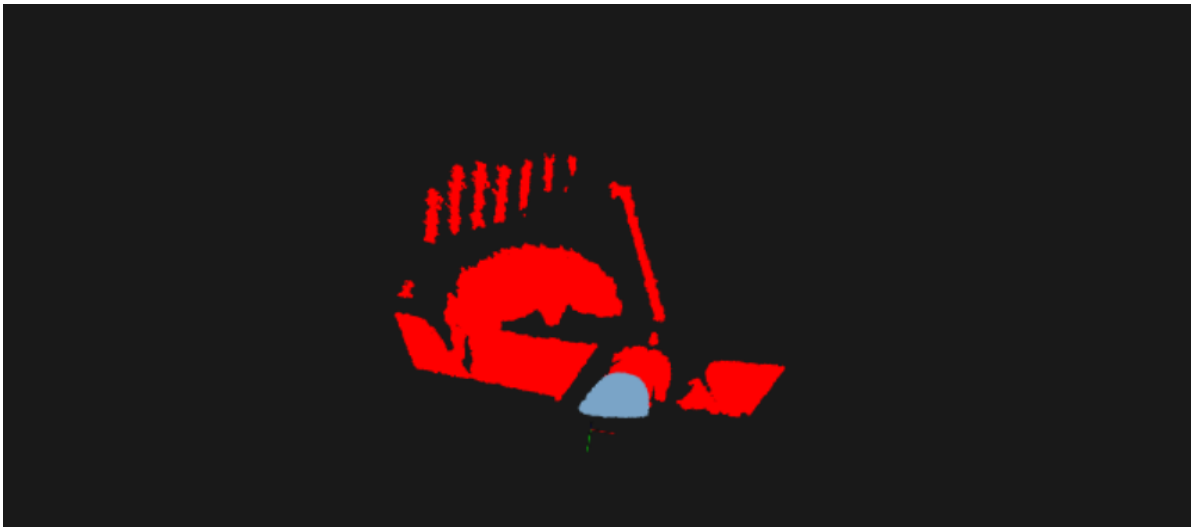
3.4.3 基于颜色的区域生长分割

1、原理概述

该算法与区域生长算法一样，是基于同一策略之上的，与区域生长相比，该算法主要有两处不同：第一，该算法用颜色代替了法线，去掉了点云规模上 限的限制。可以认为，同一个颜色且挨得近，是一类的可能性很大，不需要上限来限制。所以这种方式比较适合用于室内场景分割，尤其是复杂室内场景，颜色分割可以轻松的将连续的场景点云变成不同的物体。哪怕是高低不平的地面，没法用采样一致分割器抽掉，颜色分割算法同样能完成分割任务。第二，利用合并算法来控制过分割或欠分割。分割过程中，若两个相邻聚类的平均颜色相差较少，则将这两个聚类合并。然后进行第二步合并，在此步骤中，检查每一个聚类所包含的点的数量，如果这个数量小于用户定义的值，则当前这个聚类与其相近邻聚类合并在一起。

2、算法步骤

（1）分割，当前种子点和领域点之间色差小于色差阀值的视为一个聚类；
（2）合并，聚类之间的色差小于色差阀值和并为一个聚类，且当前聚类中点的数量小于聚类点数量的与最近的聚类合并在一起。



```
pcl::search::Search <pcl::PointXYZRGB>::Ptr tree(new pcl::search::KdTree<pcl::PointXYZR
pcl::RegionGrowingRGB<pcl::PointXYZRGB> reg;
reg.setInputCloud(cloud);
reg.setIndices(indices);
reg.setSearchMethod(tree);
reg.setDistanceThreshold(10);//设置距离阈值，用于聚类相邻点搜索
reg.setPointColorThreshold(6);//设置两点颜色阈值
reg.setRegionColorThreshold(5);//设置两类区域颜色阈值
reg.setMinClusterSize(600);//设置一个聚类的最少点数目
```

3.4.4 最小图割分割

头文件min_cut_segmentation.h

论文：https://gfx.cs.princeton.edu/pubs/Golovinskiy_2009_MBS/paper_small.pdf

1、理论基础

该算法的思想如下：

1. 建图：对于给定的点云，算法将包含点云中每一个点的图构造为一组普通顶点和另外两个称为源点和汇点的顶点。与该点对应的图的每个普通顶点都与源点和汇点相连接形成边。除此之外，每个普通顶点都有边缘，将对应的点与其最近的邻居连接起来。
2. 算法为每条边缘分配权重。有三种不同的重量：
   首先，它将权重分配到云点之间的边缘。这个权重称为平滑成本，由公式计算：

$$smoothCost = e^{-(\frac{dist}{\sigma})^2}$$

   这里dist是点之间的距离。距离点越远，边被切割的可能性就越大。
   下一步，算法设置数据成本。它包括前景和背景惩罚。第一个是将云点与源顶点连接起来并具有用户定义的常量值的边缘的权重。第二种方法被分配给连接点与汇顶点的边缘，并由公式计算：

$$backgroundPenalty = (\frac{distanceToCenter}{radius})$$

   这里distanceToCenter与水平平面中物体的预期中心的距离：

$$distanceToCenter = \sqrt{(x - centerX)^2 + (y - centerY)^2}$$

3. 在做了所有的准备工作之后，就开始寻找最小的切割。在此分析的基础上，对点云的前景点和背景点进行划分。

2、实现流程

Aleksey Golovinskiy 等人提出了一种基于最小割的点云分割方法，在三维点云数据中，根据对象的一个大致位置，建立一个 K-最近邻图，强制添加前景约束或者可选的背景约束，然后找到最小割来完成前景和背景的分割。在三维点云中进行对象分割具有挑战性，真实世界的数据比较嘈杂，前景点和背景点往往纠缠在一起，而且扫描获取的点云数据分布不均。现有的点云分割方法主要集中在提取几何图元和局部点云数据，很少有进行对象分割。基于图切的方法最初应用在图像前景和背景分割中，现将该方法扩展到处理三维点云数据中，但缺少了颜色或纹理等信息供参考，不存在平滑的表面模型而且还包含噪声点。

给定一个对象的大致位置，分割算法的目的是返回属于该对象的前景点，另外为了提高效率和精度，根据对象的尺寸，还需要附加一个属于前景范围的水平半径参数。该算法分为五个步骤：

1. 输入场景点云数据
2. 建立最近邻图，以使临近点具有相同的标签
3. 输入一个预设的前景水平半径，创建一个背景惩罚函数，使远距离点归属到背景范围中
4. 交互式地添加对象的大致位置作为前景约束
5. 输出对象点数据

需要用户提供前景点云（目标位置）的中心点，对分割结果影响很大

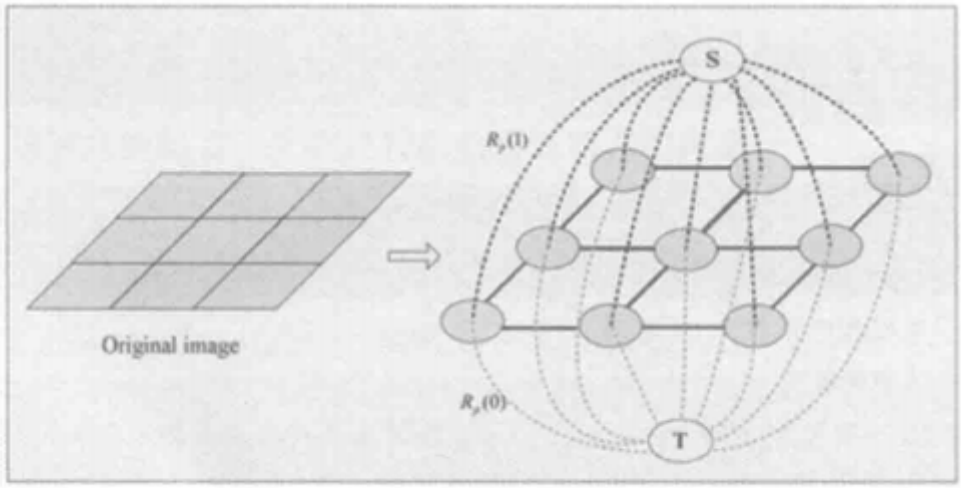
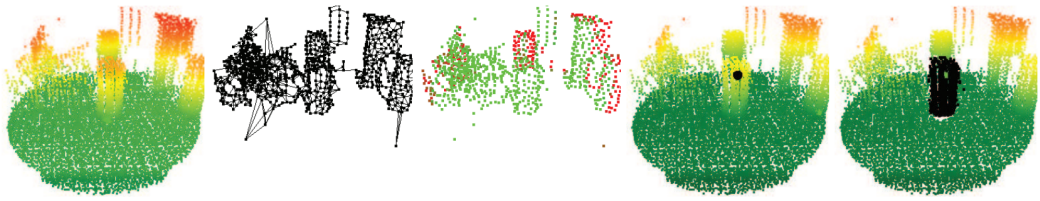


图 12-12　图割示意图




```
#include <iostream>
#include <vector>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/visualization/cloud_viewer.h>
#include <pcl/filters/passthrough.h>
#include <pcl/segmentation/min_cut_segmentation.h>//最小图割头文件

using namespace std;
int main (int argc, char** argv)
{
```

```
  pcl::PointCloud <pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud
<pcl::PointXYZ>);
  if ( pcl::io::loadPCDFile <pcl::PointXYZ>
("min_cut_segmentation_tutorial.pcd", *cloud) == -1 )
  {
    cout << "Cloud reading failed." << endl;
    return (-1);
  }
  pcl::IndicesPtr indices (new vector <int>);//创建一组索引
  //--------------------直通滤波-----------------------
  pcl::PassThrough<pcl::PointXYZ> pass;
  pass.setInputCloud (cloud);
  pass.setFilterFieldName ("z");
  pass.setFilterLimits (0.0, 1.0);
  pass.filter (*indices);
  //-------------------最小图割-----------------------
  pcl::MinCutSegmentation<pcl::PointXYZ> seg;//创建PointXYZ类型的最小图割对象
  seg.setInputCloud (cloud);
  seg.setIndices (indices);
  //提供前景点云（目标物体）的中心点，该中心点需要用户自己设置，对分割结果影响较大
  pcl::PointCloud<pcl::PointXYZ>::Ptr foreground_points(new
pcl::PointCloud<pcl::PointXYZ>);
  pcl::PointXYZ point;
  point.x = 68.97;
  point.y = -18.55;
  point.z = 0.57;
  foreground_points->points.push_back(point);
  seg.setForegroundPoints (foreground_points);//输入前景点云的中心点
  seg.setSigma (0.25);//设置平滑成本的Sigma值
  seg.setRadius (3.0433856);//设置背景惩罚权重的半径
  seg.setNumberOfNeighbours (14);//设置临近点数目
  seg.setSourceWeight (0.8);//设置前景惩罚权重

  vector <pcl::PointIndices> clusters;
  seg.extract (clusters);//获取分割结果

  cout << "Maximum flow is " << seg.getMaxFlow () << endl;
  //----------------------可视化分割结果---------------------------------

  pcl::PointCloud <pcl::PointXYZRGB>::Ptr colored_cloud =
seg.getColoredCloud();//对前景点赋予红色，对背景点赋予白色。
  pcl::visualization::PCLVisualizer viewer("最小割分割方法");
  viewer.addPointCloud(colored_cloud);
  viewer.addSphere(point, 5, 100, 100, 0, "sphere");//圆心坐标，半径，RGB
  viewer.setShapeRenderingProperties(pcl::visualization::PCL_VISUALIZER_OPACITY,
0.2, "sphere");

  while (!viewer.wasStopped())
  {
      viewer.spin();
  }

  return (0);
}
```

4 三维重建