点云匹配与识别

点云匹配与识别技术应用于3D重建，点云拼接，位姿估计，目标识别等场景中。一般来说，传统的点云匹配通过提取点云的局部描述符，寻找对应关系后进行位姿估计，而传统的点云识别通过点云聚类，提取各类点云的全局描述符，比较描述符距离实现。

该项目基于PCL库完成传统方法的点云匹配与识别，共实现8种局部描述符和6种全局描述符，并分析各个描述符的参数含义，在不同数据集中测试描述符精度与效果，**方便读者在不同场景的点云匹配与识别中快速调参，比较效果**。

项目地址：https://github.com/sally-203/object_location

一. 基本流程

点云匹配和识别的pipeline如图1所示 [1]

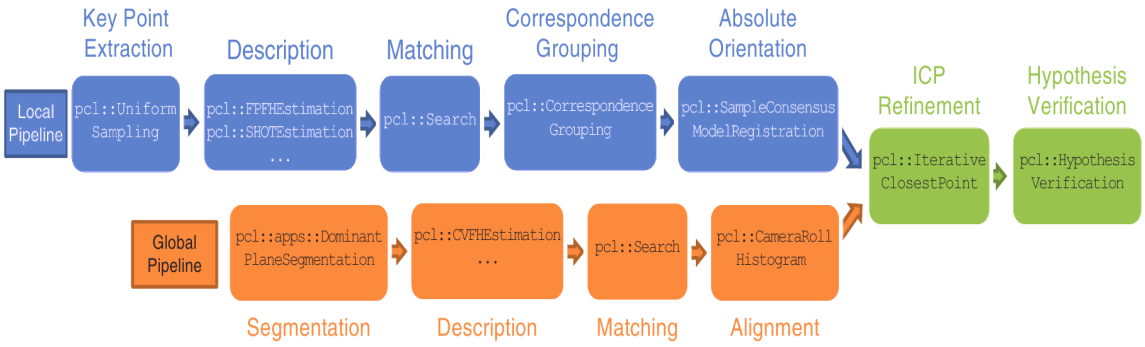


图1. 点云匹配与识别流程图

- 点云匹配流程：关键点提取，局部描述符计算，匹配对应关系，位姿估计；
- 点云识别流程：点云分割聚类，全局描述符计算，匹配对应关系；

二. 基于局部描述符的点云匹配

2.1 关键点提取

在点云匹配过程中，我们往往不会对所有的点计算描述符，所以一般都选取一些关键点计算描述符，降低计算量。而好的关键点一般需要具备2个特性：

- 可重复性：即使从不同角度拍摄场景，该点仍然为选择到的关键点；
- 独特性：需要有极高的特征描述性，可以很容易匹配到；

该项目共实现2种关键点提取算法，分别为ISS关键点与降采样方法。

2.1.1 ISS关键点提取

原理： 参考《Intrinsic shape signatures: A shape descriptor for 3D object recognition》[2]可知计算ISS的关键步骤如下：

- 选取点p_i，计算球状半径$r_{density}$邻域内每个点的权重$w_i = \frac{1}{|p_j:|p_j-p_i|<r_{density}|}$

- 计算r_{frame}邻域内，p_i带权重的协方差矩阵$cov(p_i)$，

$$COV(p_i) = \sum_{|p_j - p_i| < r_{frame}} w_j (p_j - p_i)(p_j - p_i)^T / \sum_{|p_j - p_i| < r_{frame}} w_j$$

- 计算矩阵特征值$\lambda_i^1, \lambda_i^2, \lambda_i^3$，以上特征值按照顺序递减，对应的特征向量是$e_i^1, e_i^2, e_i^3$
- 如果

$$\frac{\lambda_i^2}{\lambda_i^1} < \gamma_{21}, \frac{\lambda_i^3}{\lambda_i^2} < \gamma_{32}$$

且$\lambda_i^1 > \lambda_i^2 > \lambda_i^3$，那么$p_i$是关键点.

$\lambda_i^1 = \lambda_i^2 > \lambda_i^3$: p_i是平面上的点

$\lambda_i^1 > \lambda_i^2 = \lambda_i^3$: p_i是直线上的点

- 对λ_i^3做一次Non-Maximum Suppression

核心代码&参数详解：

```
void LocalMatcher::ExtractISSKeypoints(bool flag, const IssParameters&
iss_param)
{
    PXYZS::Ptr keypoints(new PXYZS);
    PXYZS::Ptr cloud(new PXYZS);
    pcl::ISSKeypoint3D<PXYZ, PXYZ> detector;

    detector.setInputCloud(cloud);
    pcl::search::KdTree<PXYZ>::Ptr KdTree(new pcl::search::KdTree<PXYZ>);
    detector.setSearchMethod(KdTree);
    double resolution = ComputeCloudResolution(cloud);

    // Set the radius of the spherical neighborhood used to compute the
scatter matrix.
    detector.setSalientRadius(iss_param.salient_radius * resolution);
    // Set the radius for the application of the non maxima supression
algorithm.
    detector.setNonMaxRadius(iss_param.nonmax_radius * resolution);
    // Set the minimum number of neighbors that has to be found while
applying the non maxima suppression algorithm.
    detector.setMinNeighbors(iss_param.min_neighbors);
    // Set the upper bound on the ratio between the second and the first
eigenvalue.
    detector.setThreshold21(iss_param.threshold21);
    // Set the upper bound on the ratio between the third and the second
eigenvalue.
    detector.setThreshold32(iss_param.threshold32);
    // Set the number of prpcessing threads to use. 0 sets it to automatic.
    detector.setNumberOfThreads(iss_param.num_threads);
```

```
    detector.compute(*keypoints);
}
```

首先考虑到点云的尺寸密度等参数，为了让参数更加鲁棒，需要先计算点云精度resolution。

- *salient_radius*是计算权重矩阵的点云半径r_{frame}；
- *nonmax_radius*是最后对选出来的关键点进行非极大值抑制选取的半径；
- *min_neighbors*是进行非极大值抑制方法所需要的最少的点;
- *Threshold21*是$\frac{\lambda_i^2}{\lambda_i^1}$;
- *Threshold32*是$\frac{\lambda_i^3}{\lambda_i^2}$;
- *num_threads*是运行线程数;

2.1.2 降采样

降采样方法包括体素下采样，均匀下采样，曲率下采样，随机降采样方法[3]，该项目采样体素下采样方法。

2.2 局部描述符计算

2.2.1 PFH (Point Feature Histogram) 原理

通过使用多维直方图概括点周围的平均曲率，对点的k邻域几何属性进行编码。下图表示查询点p_q以及其距离小于半径***pfh_radius***的k个邻域点的关系。

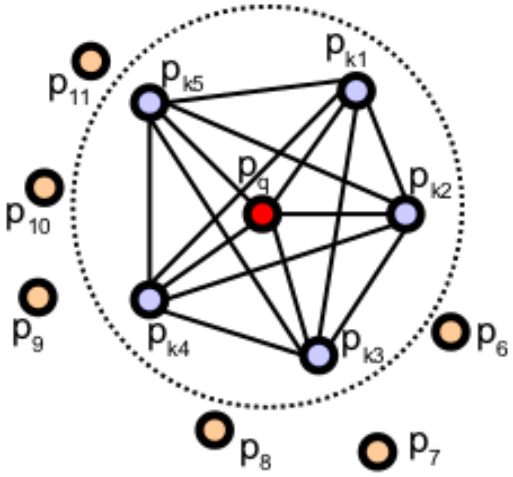


图2. PFH邻域关系

计算两点p_i和p_j的法线n_i和n_j，并定义这两个点的局部参考坐标系LRF (Local Reference Frame)，如下图所示

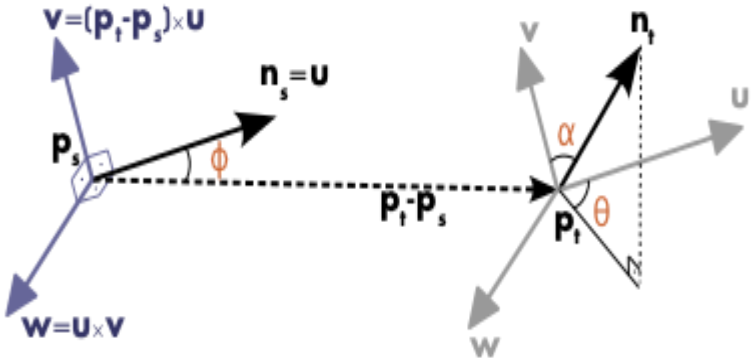


图3. LRF示意图

将两点共12个特征（x,y,z,r,g,b）减少到4个特征表示<α, ϕ, θ, d>，其中d为两点之间的欧式距离。同理计算邻域内所有两两结合的点对特征，形成直方图。

核心代码&参数详解：

```
void LocalMatcher::CalculatePfhDescri(
    const PXYZS::Ptr cloud,
    double pfh_radius, double normal_radius,
    const pcl::PointCloud<pcl::PFHSignature125>::Ptr& descriptors)
{
    pcl::PointCloud<PN>::Ptr normals(new pcl::PointCloud<PN>());
    pcl::search::KdTree<PXYZ>::Ptr kdtree(new pcl::search::KdTree<PXYZ>);

    EstimateNormalsByK(cloud, normals, 10);
    pcl::PFHEstimation<PXYZ, PN, pcl::PFHSignature125> pfh;
    pfh.setInputCloud(cloud);
    pfh.setInputNormals(normals);
    pfh.setSearchMethod(kdtree);
    pfh.setRadiusSearch(pfh_radius);

    pfh.compute(*descriptors);
}
```

这里计算描述符的主要参数是*pfh_radius*，主要影响的是图1邻域内的点数。*pfh_radius*越大，描述信息越全，但是计算量也越大。一般根据点云大小和实际场景确定*pfh_radius*。**值得注意的是**，local_parameters.h中的默认参数都是根据零件点云确定的，更换场景及点云，参数也需要调整。

2.2.2 FPFH (Fast Point Feature Histogram) 原理

FPFH是在PH基础上改进的快速描述符，如下图所示，主要按照2步进行计算：

1.对于查询点p_q，计算其自身与k个邻域点之间的PFH描述符，记为简化的点特征直方图SPFH（红色直线表示）；

2.遍历每个邻域点，同样按照第1步计算SPFH；

3.对以上计算得到的SPFH进行加权计算:

$$FPFH(p_q) = SPFH(p_q) + \frac{1}{k}\sum_{i=1}^{k}\frac{1}{w_i} \times SPFH(p_i)$$

其中权重w_i表示查询点和邻域点之间的距离，在1，2步计算中可能个会有多遍描述符计算的过程，用加粗直线表示。

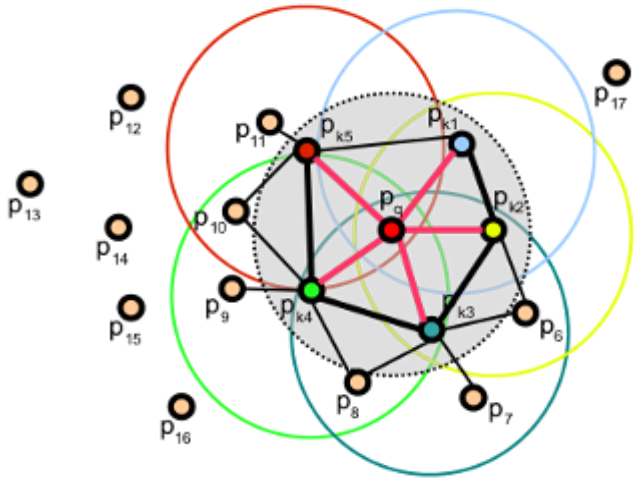


图4. FPFH示意图

核心代码&参数详解：

```
void LocalMatcher::CalculateFpfhDescri(
    const PXYZS::Ptr cloud,
    double fpfh_radius,
    const pcl::PointCloud<pcl::FPFHSignature33>::Ptr& descriptors)
{
    pcl::PointCloud<pcl::Normal>::Ptr normals(new
pcl::PointCloud<pcl::Normal>());
    pcl::search::KdTree<PXYZ>::Ptr kdtree(new pcl::search::KdTree<PXYZ>());

    EstimateNormalsByK(cloud, normals, 10);
    pcl::FPFHEstimation<PXYZ, PN, pcl::FPFHSignature33> fpfh;
    fpfh.setInputCloud(cloud);
    fpfh.setInputNormals(normals);
    fpfh.setSearchMethod(kdtree);
    fpfh.setRadiusSearch(fpfh_radius);
    fpfh.compute(*descriptors);
}
```

与PFH类似，这里计算描述符的主要参数是*fpfh_radius*，主要影响的是图3邻域内的点数。*fpfh_radius*越大，描述信息越全，但是计算量也越大。一般根据点云大小和实际场景确定*fpfh_radius*。

2.2.3 RSD (Radius-based Surface Descriptor) 原理

迭代关键点集合：

1. 选择以p_i为中心，r为半径的球体内的所有相邻点，相邻点集合记为p_{ik}；

2. 迭代p_{ik}集合，计算p_i与当前邻域点之间的距离以及它们法线之间的角度α。这些值组成p_i处曲率的直方图；
3. 可以通过两个点拟合具有近似半径r_c的圆，如果两点位于同一平面上时，则半径将变为无限大；
4. 由于查询点p_i可以是多个圆及其邻居的一部分，所以仅保留最小和最大的半径。该算法设置一个最大半径参数，超过该参数则被视为平面。

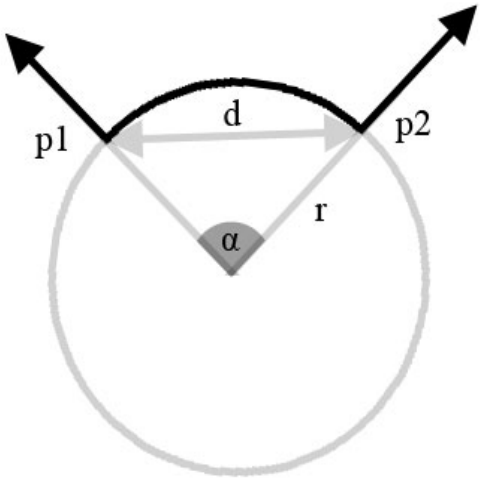


图5. RSD示意图

核心代码&参数详解：

```
void LocalMatcher::CalculateRsdDescri(
    const PXYZS::Ptr cloud,
    double rsd_radius, double plane_radius,
    const pcl::PointCloud<pcl::PrincipalRadiiRSD>::Ptr& descriptors)
{
    pcl::PointCloud<pcl::Normal>::Ptr normals(new
pcl::PointCloud<pcl::Normal>());
    pcl::search::KdTree<PXYZ>::Ptr kdtree(new pcl::search::KdTree<PXYZ>());

    EstimateNormalsByK(cloud, normals, 10);
    pcl::RSDEstimation<PXYZ, PN, pcl::PrincipalRadiiRSD> rsd;
    rsd.setInputCloud(cloud);
    rsd.setInputNormals(normals);
    rsd.setSearchMethod(kdtree);
    rsd.setRadiusSearch(rsd_radius);
    rsd.setPlaneRadius(plane_radius);
    rsd.setSaveHistograms(false);

    rsd.compute(*descriptors);
}
```

- *rsd_radius*是搜索半径r;
- *plane_radius*是平面半径，任何大于该参数的半径都被认为是无限大的平面；

2.2.4 3DSC (3D Shape Context) 原理

如下图所示，3DSC是2DSC的扩展[4]。以基础点p为例，它的北极方向为法向量方向，建立以p为中心的支撑域，并在各个方位角划分等分的bins。按照半径分为J+1份，记半径集合$R = R_0, \cdots, R_J$。按照俯仰角分为K+1份，记俯仰角集合$\theta = \theta_0, \cdots, \theta_K$。按照方位角分为L+1份，记方位角集合$\phi = \phi_0, \cdots, \phi_L$。俯仰角和方位角方向都是按照对数计算划分格子的。

$$R_0 = r_{min}, R_J = r_{max}, R_j = exp(ln(r_{min}) + \frac{j}{J} ln(\frac{r_{max}}{r_{min}}))$$

对数采样使得描述符对于距离更加鲁棒。靠近中心的bin在所有三个维度上都较小，所以这些bin对微小差异很敏感，因此我们使用最小半径 (r_{min} > 0) 参数，不计算小于R<r_{min}的bin。

θ最大为180度，ϕ最大为360度。另外，bin(j,k,l)对应权重计算公式如下：

$$w_{p_i} = \frac{1}{\rho_i \sqrt[3]{V(j,k,l)}}$$

$V(j,k,l)$ 是bin的体积，ρ_i是当前bin里面点的密度，所有bin构成3DSC描述符。

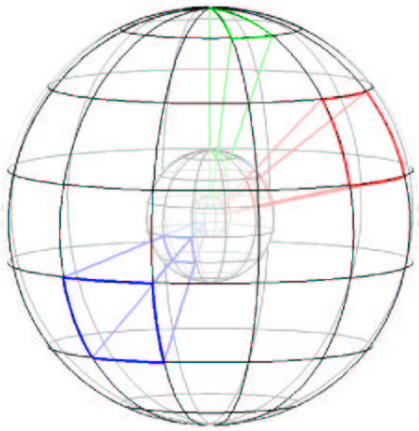


图6. 3DSC示意图

核心代码&参数详解：

```
void LocalMatcher::CalculateDscDescri(
    const PXYZS::Ptr cloud,
    double dsc_radius, double minimal_radius,
    double point_density_raidus,
    const pcl::PointCloud<pcl::ShapeContext1980>::Ptr& descriptors)
{
    pcl::PointCloud<PN>::Ptr normals(new pcl::PointCloud<pcl::Normal>());
    pcl::search::KdTree<PXYZ>::Ptr kdtree(new pcl::search::KdTree<PXYZ>());

    EstimateNormalsByK(cloud, normals, 10);
    pcl::ShapeContext3DEstimation<PXYZ, PN, pcl::ShapeContext1980> sc3d;
    sc3d.setInputCloud(cloud);
    sc3d.setInputNormals(normals);
    sc3d.setSearchMethod(kdtree);
    sc3d.setRadiusSearch(dsc_radius);
    sc3d.setMinimalRadius(minimal_radius);
    sc3d.setPointDensityRadius(point_density_raidus);
```

```
    sc3d.compute(*descriptors);
}
```

- *dsc_radius*是支撑球体半径，也是搜索半径R_{max}；
- *minimal_radius*支撑球体的最小半径r_{min}，避免在靠近球体中心的bin对噪声过于敏感；
- *point_density_radius*是计算邻域的局部点密度半径，就是在这个半径内计算ρ_i。

2.2.4 USC (Unique Shape Context) 原理

USC是在3DSC的基础上进行改进，USC研究者认为3DSC缺少可重复性的局部参考坐标系(Local Reference Frame)，因此他们提出建立一个LRF[5]。考虑当前特征点p，和半径R内的球状邻域，计算权重矩阵M

$$M = \frac{1}{Z} \sum_{i:d_i \le R} (R - d_i)(p_i - p)(p_i - p)^T$$

其中，$d_{i}=|p_{i}-p|*{2}*，$Z$是归一化因子，*Z=|sum*{i:d_{i}\le R}(R-d_{i})$。接着对M进行特征向量分解，最小特征值对应的特征向量为当前坐标系的法向量方向。

核心代码&参数详解：

```
void LocalMatcher::CalculateUscDescri(
    const PXYZS::Ptr cloud,
    double usc_radius, double minimal_radius,
    double point_density_raidus, double local_radius,
    const pcl::PointCloud<pcl::UniqueShapeContext1960>::Ptr& descriptors)
{
    pcl::search::KdTree<PXYZ>::Ptr kdtree(new pcl::search::KdTree<PXYZ>());

    pcl::UniqueShapeContext<PXYZ, pcl::UniqueShapeContext1960,
pcl::ReferenceFrame> usc;
    usc.setInputCloud(cloud);
    usc.setRadiusSearch(usc_radius);
    usc.setMinimalRadius(minimal_radius);
    usc.setPointDensityRadius(point_density_raidus);
    usc.setLocalRadius(local_radius);

    usc.compute(*descriptors);
}
```

因为是在3DSC基础上改进的，所以前三个参数都和3DSC一样

- *usc_radius*是支撑球体半径，也是搜索半径；
- *minimal_radius*是支撑球体的最小半径r_{min}，避免在靠近球体中心的bin对噪声过于敏感；
- *point_density_radius*是计算邻域的局部点密度半径，就是在这个半径内计算ρ_i;
- *local_radius*表示计算LRF的半径范围R。

2.2.4 SHOT (Signature of Histograms of Orientations) 原理

SHOT是一种基于局部特征的描述子，在特征点处建立局部坐标系，将邻域点的空间位置信息和几何特征统计信息结合起来描述特征点。Tombari等人[6]将3D局部特征描述方法分为两类，即基于特征的描述方法与基于直方图的描述方法，并分析了两种方法的优势，提出基于特征的局部特征描述方法要比后者在特征的描述能力上更强，而基于直方图的局部特征描述方法在特征的鲁棒性上比前者更胜一筹。计算步骤如下：

1. 按照USC相同的原理，在特征点邻域半径R内建立参考坐标系LRF，对特征点的球邻域分别沿径向（内外球），经度（时区），和纬度（南北半球）方向进行区域划分。通常径向划分为2，经度划分为8，纬度划分为2，总共32个区域；
2. 计算LRF中的每个划分区间内的每一点和坐标系原点（特征点）的角度θ的余弦值$cos\theta$，按其值保存在直方图中，这样初步得到了该特征点的直方图表达。每一个$cos\theta$可以划分11个区间（实验表示这个区间数量最好），所以总共有32*11个特征向量；
3. 因为描述符是基于局部直方图的，不可避免会受到边缘效应的影响，因此Tombari等人采用了四线性插值[7]方法。

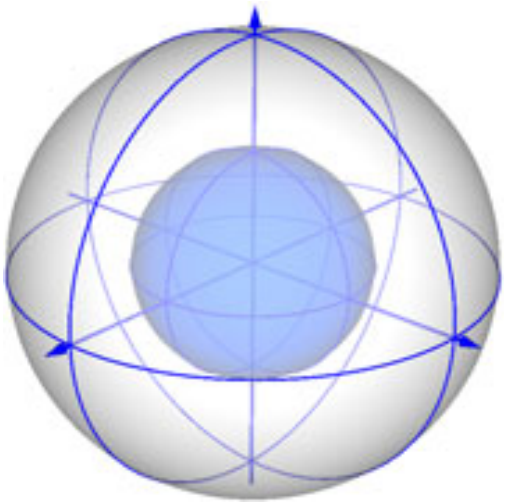


图7. SHOT示意图

核心代码&参数详解：

```
void LocalMatcher::CalculateShotDescri(
    const PXYZS::Ptr cloud,
    double shot_radius,
    const pcl::PointCloud<pcl::SHOT352>::Ptr& descriptors)
{
    pcl::PointCloud<PN>::Ptr normals(new pcl::PointCloud<pcl::Normal>());
    pcl::search::KdTree<PXYZ>::Ptr kdtree(new pcl::search::KdTree<PXYZ>());

    EstimateNormalsByK(cloud, normals, 10);

    pcl::SHOTEstimation<PXYZ, PN, pcl::SHOT352> shot;
    shot.setInputCloud(cloud);
    shot.setInputNormals(normals);
    shot.setRadiusSearch(shot_radius);

    shot.compute(*descriptors);
}
```

- *shot_radius*是关键点邻域半径，搜索半径

2.2.5 SI (Spin image) 原理

参考论文[8]，可知Spin Image是基于点云空间分布的最经典的特征描述方法，其思想是将一定区域的点云分布转换成二维的Spin Image，然后对场景和模型的Spin Image进行相似性度量。

参考博客[9]：

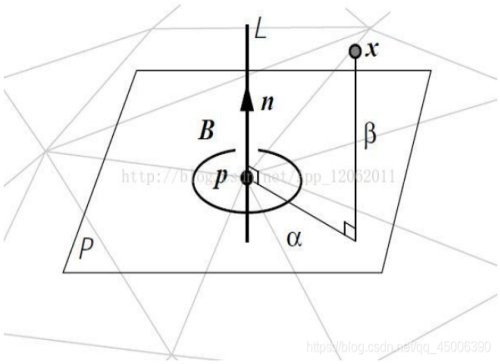


图8. SI特征点示意图

- P：三维网格某顶点P的切面
- n：p点的单位法向量
- x：p附近的三维网格上的另一个顶点
- α：x点在P上的投影与p的距离
- β：x点与P点的垂直距离
- p和n统一称为带法向的有向点

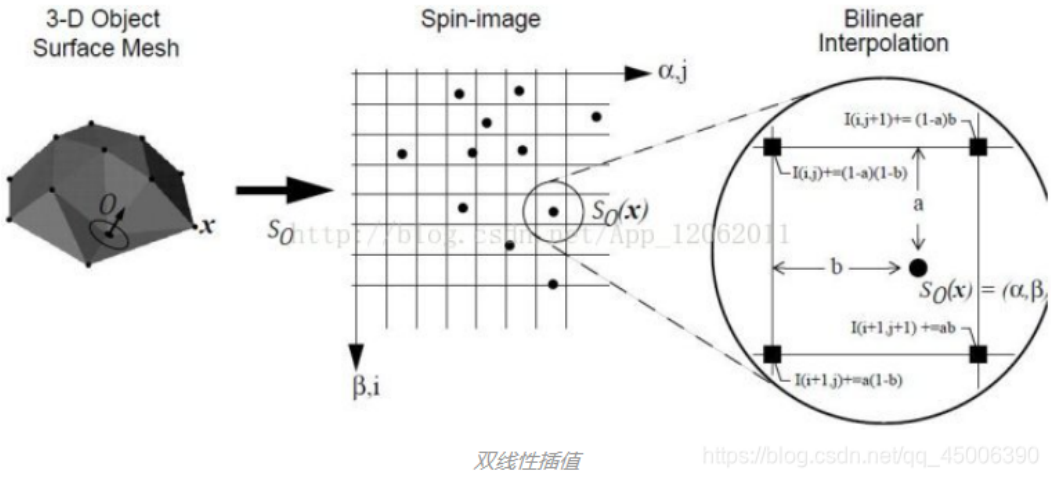
生成Spin Image的步骤如下所示：

1. 以有向点p的法向为轴生成一个圆柱坐标系；
2. 定义Spin Image参数，它是一个具有一定大小（行数列数），分辨率（二维网格大小）的二维图像；
3. 将圆柱体内的三维坐标投影到二维的Spin Image，这一过程可以理解为一个Spin Image绕着法向量n旋转360度，Spin Image扫到的三维空间的点会落到Spin Image的网格中：

$$S_O : R^3 \longrightarrow R^2$$

$$S_O(x) \longrightarrow (\alpha, \beta) = (\sqrt{|x-p|^2 - (n \cdot (x-p))^2}, n \cdot (x-p))$$

4. 根据Spin Image中的每个网格中落入的点的不同，计算每个网格的强度I。为了降低对位置的敏感度和降低噪声，使用双线性插值将1个点分布到4个像素中。



双线性插值

图9. 双线性插值示意图

5. 一般使用以下公式进行相似性度量：

$$C(P,Q) = (atanh(R(P,Q)))^2 - \lambda(\frac{1}{N-3})$$

其中

$$R(P,Q) = \frac{N\sum p_i q_i - \sum p_i \sum q_i}{\sqrt{(N\sum p_i^2 - (\sum p_i)^2(N\sum q_i^2 - (\sum q_i)^2)}}$$

其中， N是每个Spin Image的像素数，atanh为反双曲正切函数，其输入的范围是[-1,1]，R的范围是[-1,1]，两个Spin Image越相似，R越接近于1，完全一样时R的值为1。

核心代码&参数详解：

```
void LocalMatcher::CalculateSiDescri(
    const PXYZS::Ptr cloud,
    double si_radius, int image_width,
    double normal_radius,
    const pcl::PointCloud<pcl::Histogram<153>>::Ptr& descriptors)
{
    pcl::PointCloud<PN>::Ptr normals(new pcl::PointCloud<PN>());
    pcl::search::KdTree<PXYZ>::Ptr kdtree(new pcl::search::KdTree<PXYZ>());
    EstimateNormalsByK(cloud, normals, 10);

    pcl::SpinImageEstimation<PXYZ, PN, pcl::Histogram<153>> si;
    si.setInputCloud(cloud);
    si.setInputNormals(normals);
    si.setRadiusSearch(si_radius);
    si.setImageWidth(image_width);

    si.compute(*descriptors);
}
```

- image_width是spin_image的分辨率，即一个维度的bins数量
- si_radius是关键点邻域半径，也就是搜索半径

2.2.6 ROPS (Rotational Projection Statistics) 原理

ROPS是Yulan Guo等人[10]提出的一种通过计算位于局部表面上的所有点的散射矩阵来定义LRF的新技术。ROPS特征描述符是通过将特征点的邻近点旋转投影到2D平面上并计算这些投影点分布的统计数据获得的。局部曲面由给定支撑半径内的点和三角形组成，对于给定的局部表面，计算LRF(局部参考系)。

1. 绕当前轴，以给定角度旋转局部表面；
2. 组成旋转的局部表面的点投影到XY，XZ，YZ平面；
3. 建立投影分布矩阵，矩阵反映了每个bin中点的数量分布，bins数量表示矩阵维度；
4. 计算矩阵的中心矩和香农熵：M11，M12，M21，M22，E；
5. 这些值组成子特征；
6. 按照不同的旋转角度迭代重复步骤1-5，连接子特征构建最后的ROPS特征描述符。

核心代码&参数详解：

```
void LocalMatcher::CalculateRopsDescri(
    const PXYZS::Ptr cloud, double rops_radius,
    int num_partions_bins, int num_rotations,
    double support_radius, double normal_radius,
    const pcl::PointCloud<pcl::Histogram<135>>::Ptr& descriptors)
{
    pcl::PointCloud<PN>::Ptr normals(new pcl::PointCloud<PN>());
    pcl::search::KdTree<PXYZ>::Ptr kdtree(new pcl::search::KdTree<PXYZ>());
    pcl::PointCloud<pcl::PointNormal>::Ptr cloudNormals(
        new pcl::PointCloud<pcl::PointNormal>);

    EstimateNormalsByK(cloud, normals, 10);

    // perform triangulation
    pcl::concatenateFields(*cloud, *normals, *cloudNormals);
    pcl::search::KdTree<pcl::PointNormal>::Ptr kdtree2(
        new pcl::search::KdTree<pcl::PointNormal>);
    kdtree2->setInputCloud(cloudNormals);

    pcl::GreedyProjectionTriangulation<pcl::PointNormal> triangulation;
    pcl::PolygonMesh triangles;
    triangulation.setSearchRadius(0.1);
    triangulation.setMu(2.5);
    triangulation.setMaximumNearestNeighbors(10);
    triangulation.setMaximumSurfaceAngle(M_PI / 4);
    triangulation.setNormalConsistency(false);
    triangulation.setMinimumAngle(M_PI / 18);
    triangulation.setMaximumAngle(2 * M_PI / 3);
    triangulation.setInputCloud(cloudNormals);
    triangulation.setSearchMethod(kdtree2);
    triangulation.reconstruct(triangles);

    // rops estimation object
    pcl::ROPSEstimation<PXYZ, pcl::Histogram<135>> rops;
    rops.setInputCloud(cloud);
    rops.setSearchMethod(kdtree);
```

```
    rops.setRadiusSearch(rops_radius);
    rops.setTriangles(triangles.polygons);
    rops.setNumberOfPartitionBins(num_partions_bins);
    rops.setNumberOfRotations(num_rotations);
    rops.setSupportRadius(support_radius);

    rops.compute(*descriptors);
}
```

三角化算法是为了得到点云多边形，该项目使用GreedyProjectionTriangulation方法实现3D点云的三角剖分。

- SearchRadius: 搜索半径
- Mu: 设置最近邻距离的乘数以获得每个点的最终搜索半径（这将使算法适应不同的点云密度）
- MaximumNearestNeighbors: 设置搜索的最大数量的最近邻点
- MaximumSurfaceAngle: 如果点的法线与查询点法线的偏差超过此值，则不要考虑进行三角测量
- MinimumAngle: 每个三角形的最小角度
- MaximumAngle: 每个三角形的最大角度

三角化之后，进行rops描述符估计

- num_partions_bins: 允许设置用于分布矩阵计算的bins数量
- num_rotations: 旋转次数越多，得到的描述符就越大
- support_radius: 允许设置用于裁剪点的局部表面的支撑半径
- rops_radius: 邻域半径，也就是搜索半径

2.3 匹配对应关系

已经得到scene和model点云的描述符集合，接下来可以根据最近邻搜索得到两个点云对应匹配的描述符。

1. 首先，根据model点云描述符建立搜索树；
2. 接着，遍历scene点云描述符，在搜索树中寻找1个与它距离最接近的model描述符
3. 最后，记录找到的匹配描述符

核心代码&&参数详解（以PFH举例）：

```
pcl::KdTreeFLANN<pcl::PFHSignature125> matching;
matching.setInputCloud(model_descriptors);

pcl::CorrespondencesPtr correspondences(new pcl::Correspondences());

for (size_t i = 0; i < scene_descriptors->size(); ++i) {
    std::vector<int> neighbors(1);
    std::vector<float> squaredDistances(1);
    if (std::isfinite(scene_descriptors->at(i).histogram[0])) {
        int neighborCount = matching.nearestKSearch(scene_descriptors-
>at(i), 1,
            neighbors, squaredDistances);
        // std::cout << squaredDistances[0] << std::endl;
        if (neighborCount == 1 && squaredDistances[0] <
pfh_param.distance_thre) {
```

```
                pcl::Correspondence correspondence(neighbors[0],
static_cast<int>(i),
                    squaredDistances[0]);
                correspondences->push_back(correspondence);
            }
        }
    }
```

这边需要注意的是**distance_thre**，这是搜索的描述符距离阈值，不同的描述符，阈值不同，可以参考local_paramters.h

2.4 位姿估计

在标准RANSAC位姿估计循环中插入了一个简单但有效的“预拒绝”步骤，以避免验证可能错误的位姿假设。这是通过局部姿态不变的几何约束来实现的方法。

核心代码&&参数详解（以PFH举例）：[11]

```
pcl::SampleConsensusPrerejective<PXYZ, PXYZ, pcl::PFHSignature125> pose;
PXYZS::Ptr alignedModel(new PXYZS);

pose.setInputSource(model_keypoints_);
pose.setInputTarget(scene_keypoints_);
pose.setSourceFeatures(model_descriptors);
pose.setTargetFeatures(scene_descriptors);

pose.setCorrespondenceRandomness(pfh_param.common_params.randomness);
pose.setInlierFraction(pfh_param.common_params.inlier_fraction);
pose.setNumberOfSamples(pfh_param.common_params.num_samples);
pose.setSimilarityThreshold(pfh_param.common_params.similiar_thre);
pose.setMaxCorrespondenceDistance(pfh_param.common_params.corres_distance);
pose.setMaximumIterations(pfh_param.common_params.nr_iterations);

pose.align(*alignedModel);
if (pose.hasConverged()) {
    transformations_ = pose.getFinalTransformation();
    print();
} else {
    std::cout << "Did not converge." << std::endl;
}
```

- InputSource: 源点云关键点
- InputTatget: 目标点云关键点
- SourceFeatures: 源点云描述符
- TargetFeatures: 目标点云描述符
- CorrespondenceRandomness: 随机度。构造对应点对时，并非直接选择特征匹配距离最小的，而是在若干个最佳匹配点之间进行随机选择，这样虽然增加了迭代次数，但使得该算法对包含离群点的数据具有更好的鲁棒性（default=3）

- InlierFraction: 内点数量占比。在许多实际应用中，观察到的目标对象的大部分在其所在场景中都是不可见的。在这种情况下，对于场景中我们不需要点云数据中的所有点都配准，只要内点数目占所有点云数目的占比较高，则该变换假设就是有效的（default=0.01）
- NumberOfSamples: 采样数目。在目标物体和场景之间，对应点对的样本数目，为了正常位姿估计至少需要3个点（default=3）
- SimilarityThreshold: 相似性阈值。这个值越接近1，该算法通过减少迭代次数而变得更快速。但是，当噪声存在时，也增加了排除正确位置的风险（default=0.4）
- MaxCorrespondenceDistance: 内点阈值。源点云与目标点云的最近点距离小于阈值，则认为该对应点对为内点（default=1.0）
- MaximumIterations: 最大迭代次数。(default=20000)

三. 基于全局描述符的点云识别

3.1 点云分割聚类

分割聚类方法需要将点云P划分到更小的部分，减少整体的处理时间。欧几里得意义上的简单数据聚类方法可以通过使用固定宽度框或更一般地八叉树数据结构来利用空间的3D网格细分来实现。

1. 为输入点云数据集P创建Kd-tree表示
2. 建立聚类的空列表C和需要核对的点队列Q
3. 遍历$p_i \in P$，执行以下步骤：

- 给当前的队列Q增加p_i
- 针对每个点$p_i \in Q$，搜索$r < d_{th}$的邻域点组成集合P_i^k，并判断该邻域点是否已经被处理，如果没有就加入Q
- 当Q中的所有点都被处理过了，增加Q到C中，并且重设Q为一个空列表

4. 当$p_i \in P$都被处理过之后，算法终止

核心代码&&参数详解：

```
void GlobalMatcher::ClusterPointCloud(const bool flag, const PXYZS::Ptr
cloud,
    const ClusterParameters& cluster_param)
{
    pcl::search::KdTree<PXYZ>::Ptr kdtree(new pcl::search::KdTree<PXYZ>);
    kdtree->setInputCloud(cloud);

    pcl::EuclideanClusterExtraction<PXYZ> ec;
    ec.setClusterTolerance(cluster_param.cluster_tolerance);
    ec.setMinClusterSize(cluster_param.min_cluster_size);
    ec.setMaxClusterSize(cluster_param.max_cluster_size);
    ec.setSearchMethod(kdtree);
    ec.setInputCloud(cloud);

    std::vector<pcl::PointIndices> cluster_indices;
    ec.extract(cluster_indices);

    for (const auto& indices : cluster_indices) {
        PXYZS::Ptr cluster(new PXYZS);
```

```
            for (const auto& index : indices.indices) {
                cluster->points.push_back(cloud->points[index]);
            }
            cluster->width = cluster->points.size();
            cluster->height = 1;
            cluster->is_dense = true;
            if (flag) {
                scene_clusters_.push_back(cluster);
            } else {
                model_clusters_.push_back(cluster);
            }
        }
        if (flag) {
            std::cout << "scene cluster size: " << scene_clusters_.size() <<
std::endl;
            if (cluster_param.show_flag) {
                VisualizeClusters(scene_clusters_);
            }
        } else {
            std::cout << "model cluster size: " << model_clusters_.size() <<
std::endl;
            if (cluster_param.show_flag) {
                VisualizeClusters(model_clusters_);
            }
        }
    }
```

- ClusterTolerance: 聚类容差。如果取一个非常小的值，则可能会发生一个实际对象可以被视为多个簇的情况。另一方面，如果将该值设置得太高，则可能会发生多个对象被视为一个簇的情况。 因此，需要测试并尝试哪个值适合对应的数据集。
- MinClusterSize: 最小的聚类size。
- MaxClusterSize: 最大的聚类size。

3.2 全局描述符计算

全局描述符不是针对单个点计算的，而是针对代表对象的整个簇计算的，因此需要预处理步骤（cluster）。全局描述符用于对象识别，分类和几何分析。许多局部描述符也可以作为全局描述符，这可以通过将半径设置为任意两点之间的最大可能距离（因此簇中的所有点被视为邻居）来实现。

3.2.1 VFH(Viewpoint Feature Histogram)原理

视点特征直方图VFH是在FPFH的基础上，保持尺度不变性的同时添加视点方差，扩展到对整个对象簇的FPFH估计，并计算视点方向和每个点估计的法线之间的附加统计数据。

VFH是由两部分组成：视点方向分量和扩展FPFH分量。

1. 计算视点方向分量：首先，需要找到对象的质心，该质心是对所有点的X，Y，Z坐标进行平均而得到的点；然后，计算视点（传感器位置）和该质心之间的矢量并进行归一化；最后，对于簇中的所有点，计算该向量与其法线之间的角度，并将结果合并到直方图中。在计算角度时，矢量会平移到每个点。

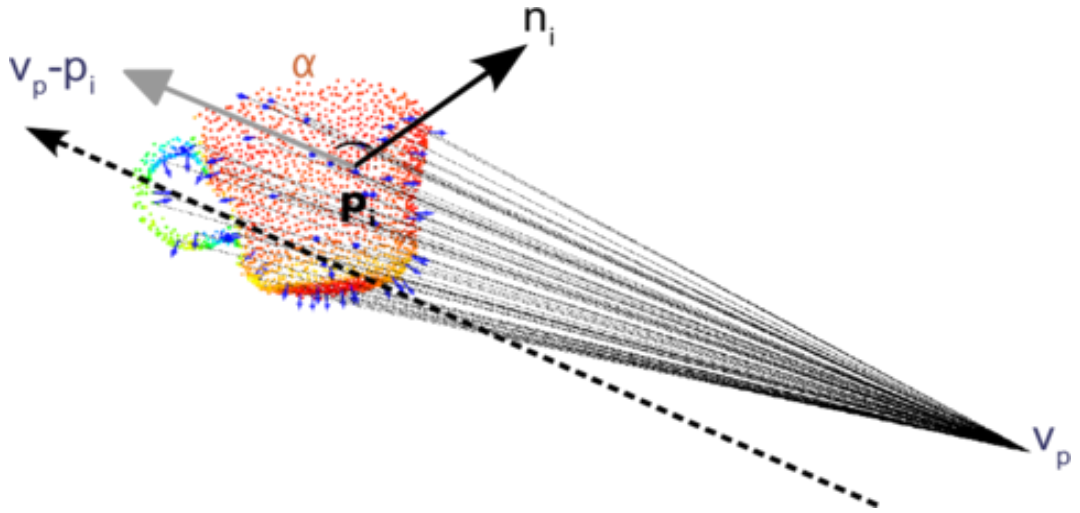


图10. 视点方向分量

2. 计算扩展FPFH分量：类似于FPFH计算，唯一的区别时，它仅针对质心计算，使用计算的视点方向向量作为其法线，并将所有簇的点设置为邻域点。

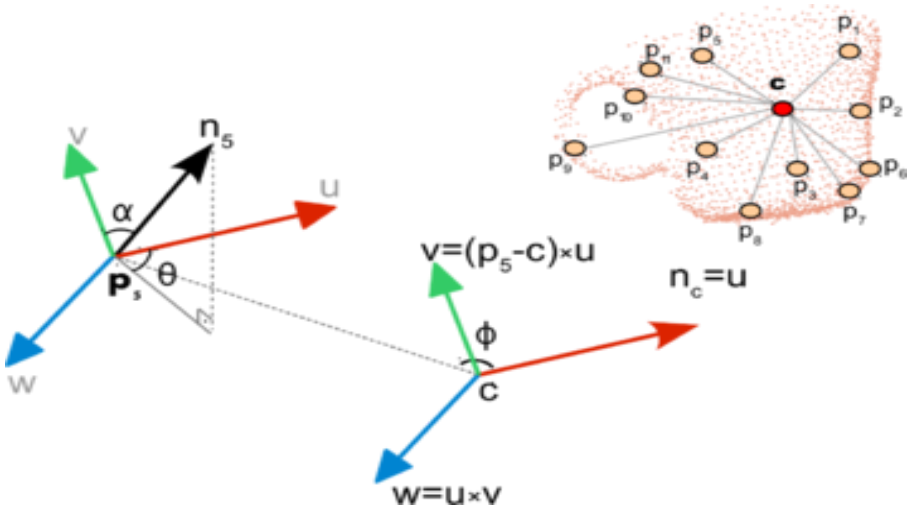


图11. 扩展FPFH分量

3. 将生成的4个直方图（1个用于视点分量，3个用于扩展的FPFH分量）连接起来以构建最终的VFH描述符。默认情况下，bins使用簇中的总点数进行标准化，这使得VFH描述符不随比例变化。

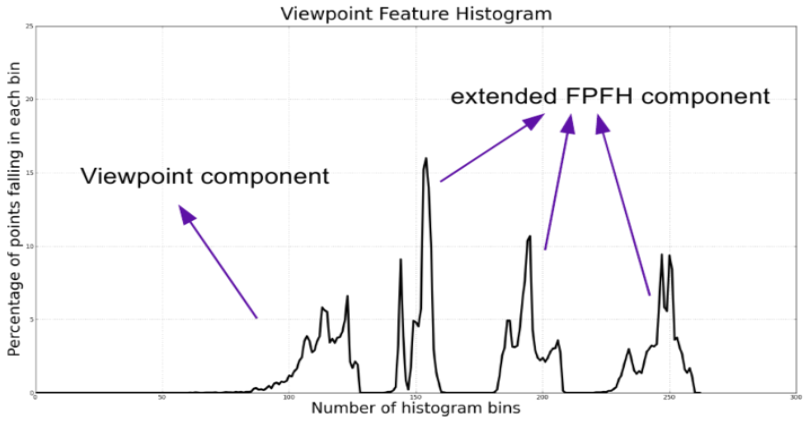


图12. VFH直方图拼接

PCL实现的VFH计算增加了第5个直方图，包含簇点到质心的距离（SDC, 可以参考CVFH)

核心代码&&参数详解：

```
void GlobalMatcher::CalculateVfhDescri(
    const PXYZS::Ptr cloud,
    pcl::PointCloud<pcl::VFHSignature308>::Ptr descriptors)
{
    pcl::PointCloud<pcl::Normal>::Ptr normals(new pcl::PointCloud<PN>());
    pcl::search::KdTree<PXYZ>::Ptr kdtree(new pcl::search::KdTree<PXYZ>());
    EstimateNormalsByK(cloud, normals, 10);

    pcl::VFHEstimation<PXYZ, PN, pcl::VFHSignature308> vfh;
    vfh.setInputCloud(cloud);
    vfh.setInputNormals(normals);
    vfh.setSearchMethod(kdtree);
    vfh.setNormalizeBins(true);
    vfh.setNormalizeDistance(false);

    vfh.compute(*descriptors);
}
```

- NormalizeBins: 使用总点数对结果直方图的bins进行归一化
- NormalizeDistance: 使用质心和任何簇点之间找到的最大尺寸来标准化SDC

3.2.2 CVFH(Clustered Viewpoint Feature Histogram)原理

VFH描述符对于遮挡，其他传感器伪影和测量误差并不稳健。如果对象簇丢失了许多点，则计算出的质心将与原始质心不同，从而改变最终描述符，不能找到正确的匹配。因此，引入聚类视点特征直方图CVFH。

相比于VFH计算整个簇的单个VFH直方图，CVFH首先使用**区域生长分割**将对象划分为稳定，平滑的区域，然后在每个区域都计算VFH描述符。 此外，CVFH还会计算形状分布分量(SDC)，这对有关区域质心周围的点分布的信息进行编码，并测量距离。SDC可以区分具有相似特征（大小和正态分布）的对象，例如彼此的两个平面。CVFH相对相机滚动角度是不变的，围绕该相机轴的旋转不会改变计算描述符。

图13. 原始点云

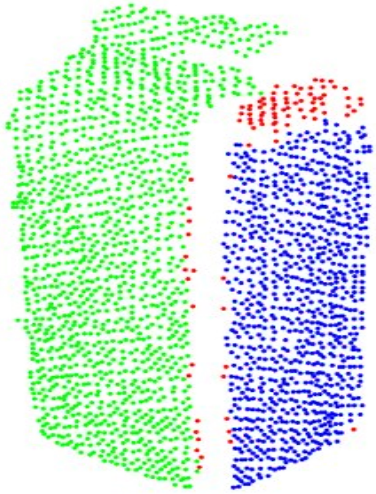


图14. 分割点云

核心代码&&参数详解：

```
void GlobalMatcher::CalculateCvfhDescri(
    const PXYZS::Ptr cloud,
    double eps_angle, double curv_thre,
    const pcl::PointCloud<pcl::VFHSignature308>::Ptr descriptors)
{
    pcl::PointCloud<pcl::Normal>::Ptr normals(new pcl::PointCloud<PN>());
    pcl::search::KdTree<PXYZ>::Ptr kdtree(new pcl::search::KdTree<PXYZ>());
    EstimateNormalsByK(cloud, normals, 10);

    pcl::CVFHEstimation<PXYZ, PN, pcl::VFHSignature308> cvfh;
    cvfh.setInputCloud(cloud);
    cvfh.setInputNormals(normals);
    cvfh.setEPSAngleThreshold(eps_angle);
    cvfh.setCurvatureThreshold(curv_thre);
    cvfh.setNormalizeBins(false);

    cvfh.compute(*descriptors);
}
```

- EPSAngleThreshold: 在区域分割步骤中法线的最大允许偏差
- CurvatureThreshold: 区域分割步骤的曲率阈值（曲率之间的最大差异）
- NormalizeBins: 是否使用所有点对结果直方图进行归一化，Note: 如果设置为true，则CVFH是尺度不变的。

3.2.3 OUR-CVFH(The Oriented, Unique and Repeatable CVFH)原理

OUR-CVFH扩展了之前的描述符，添加了唯一参考系的计算以使其更加鲁棒。

OUR-CVFH依赖半全局唯一参考系(Semi-Global Unique Reference Frames, SGURF)的使用，是为每个区域计算的可重复坐标系。它们不仅消除了相机胶卷的不变性并允许直接提取6DOF位姿而不需要额外的步骤，而且还提高了空间描述性。

核心代码&&参数详解：

```
void GlobalMatcher::CalculateOurcvfhDescri(
    const PXYZS::Ptr cloud,
    double eps_angle, double curv_thre, double axis_ratio,
    const pcl::PointCloud<pcl::VFHSignature308>::Ptr descriptions)
{
    pcl::PointCloud<pcl::Normal>::Ptr normals(new pcl::PointCloud<PN>());
    pcl::search::KdTree<PXYZ>::Ptr kdtree(new pcl::search::KdTree<PXYZ>());
    EstimateNormalsByK(cloud, normals, 10);

    pcl::OURCVFHEstimation<PXYZ, PN, pcl::VFHSignature308> ourcvfh;
    ourcvfh.setInputCloud(cloud);
    ourcvfh.setInputNormals(normals);
    ourcvfh.setSearchMethod(kdtree);
    ourcvfh.setEPSAngleThreshold(eps_angle);
    ourcvfh.setCurvatureThreshold(curv_thre);
    ourcvfh.setNormalizeBins(false);
    ourcvfh.setAxisRatio(axis_ratio);

    ourcvfh.compute(*descriptions);
}
```

- EPSAngleThreshold，CurvatureThreshold和NormalizeBins参数与CVFH算法中的参数一样
- AxisRatio: 设置SGURF轴之间的最小轴比率。

3.2.4 ESF(Ensemble of Shape Functions)原理

形状函数集合（ESF）是3个不同形状函数的组合，描述点云的某些属性：距离，角度和面积。这些对噪声和不完整表面具有鲁棒性。该算法迭代点云中的所有点，每次迭代都会随机选择3个点，对于这些点，计算形状函数：

- D2：计算3个点对之间的距离。针对每一个点对，检查连接两个点的线是否完全位于曲面内部，完全位于曲面外部或穿过曲面。分别合并到三类直方图中：IN，OUT或MIXED；
- D2比率： 曲面内部线条部分与外部线条部分之间比率的直方图。如果线完全位于外部，则该值为0；如果完全位于内部，则该值为1；如果混合，则为中间的某个值；
- D3：计算3个点形成的三角形面积的平方根。与D2一样，结果也分为IN，OUT，或MIXED三个类别的直方图；
- A3：计算点形成的角度。然后，根据与D2比率相反的方式对该值建立直方图。

循环结束后，一共计算了10个子直方图（D2，D3，A3每个都有3个直方图，以及D2比率的直方图，一共有10个），每个直方图有64bins，因此最终ESF描述符大小为640.

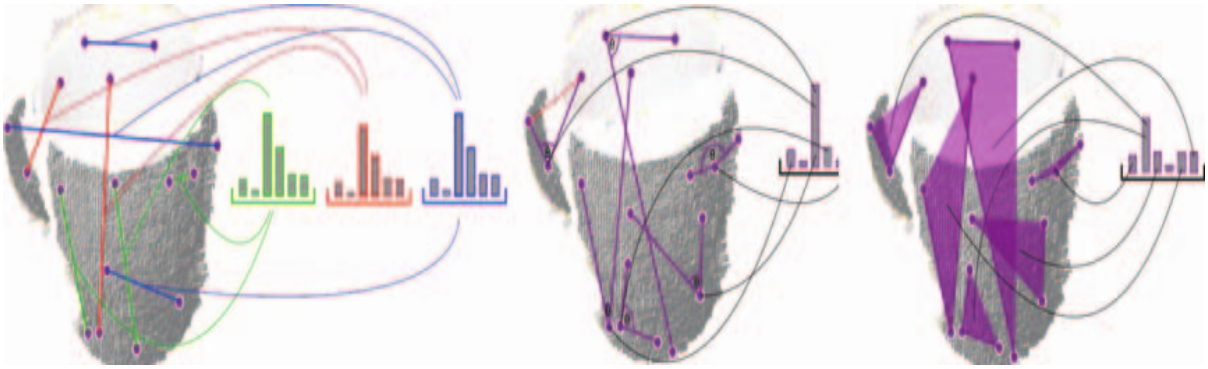


图15. ESF描述符

核心代码&&参数详解：

```
void GlobalMatcher::CalculateEsfDescri(
    const PXYZS::Ptr cloud,
    pcl::PointCloud<pcl::ESFSignature640>::Ptr descriptions)
{
    pcl::PointCloud<pcl::Normal>::Ptr normals(new pcl::PointCloud<PN>());
    pcl::search::KdTree<PXYZ>::Ptr kdtree(new pcl::search::KdTree<PXYZ>());
    EstimateNormalsByK(cloud, normals, 10);

    pcl::ESFEstimation<PXYZ, pcl::ESFSignature640> esf;
    esf.setInputCloud(cloud);
    esf.compute(*descriptions);
}
```

3.2.5 GFPFH(Global Fast Point Feature Histogram)原理

GFPFH表示全局FPFH描述符，设计目的是机器人的环境导航。

1. 表面分类。创建一组物体类别，假如一个咖啡杯，它可以有三个类别：手柄，杯子外表面，杯子内表面。
2. 计算FPFH描述符，并和类别一起输入CRF（条件随机场）算法。它可以标记每个表面，其中每个点都根据其所属的对象的类型进行分类。
3. 计算GFPFH描述符。设置八叉数，保存每个点属于某个类别的概率，将所有点的概率组成直方图。

核心代码&&参数分析：

```
void GlobalMatcher::GFPFHMatch(const GfpfhParameters& gfpfh_params)
{
  // label the point cloud
  pcl::PointCloud<pcl::GFPFHSignature16>::Ptr model_descriptors(new
pcl::PointCloud<pcl::GFPFHSignature16>);
      for (int i = 0; i < model_clusters_.size(); ++i) {
          auto cluster = model_clusters_[i];
          pcl::PointCloud<pcl::PointXYZL>::Ptr object(new
pcl::PointCloud<pcl::PointXYZL>());
          for (int j = 0; j < cluster->points.size(); ++j) {
              pcl::PointXYZL point;
```

```
                point.x = cluster->points[j].x;
                point.y = cluster->points[j].y;
                point.z = cluster->points[j].z;
                // point.label = 1 + j % scene_clusters_.size();
                point.label = 1 + i % model_clusters_.size();
                object->push_back(point);
            }
        }
        .......
}3.2.5 GFPFH(Global Fast Point Feature Histogram)原理3.2.5 GFPFH(Global
Fast Point Feature Histogram)原理

void GlobalMatcher::CalculateGfpfhDescri(const
pcl::PointCloud<pcl::PointXYZL>::Ptr cloud,
    double octree_leaf_size, double num_classes,
    const pcl::PointCloud<pcl::GFPFHSignature16>::Ptr descriptions)
{
    pcl::GFPFHEstimation<pcl::PointXYZL, pcl::PointXYZL,
pcl::GFPFHSignature16> gfpfh;
    gfpfh.setInputCloud(cloud);
    gfpfh.setInputLabels(cloud);
    gfpfh.setOctreeLeafSize(octree_leaf_size);
    gfpfh.setNumberOfClasses(num_classes);

    gfpfh.compute(*descriptions);
}
```

- octree_leaf_size: 八叉数叶子的大小
- num_classes: 类别数量

3.2.6 GRSD(Global Radius-based Surface Descriptor)原理

GRSD表示全局的RSD描述符。与GFPFH类似，预先执行体素化和表面分类步骤，使用RSD根据几何类别标记所有表面块。然后，将整个物体进行分类，并据此计算GRSD描述符。

核心代码&&参数分析：

```
void GlobalMatcher::CalculateGrsdDescri(
    const PXYZS::Ptr cloud,
    double grsd_radius,
    const pcl::PointCloud<pcl::GRSDSignature21>::Ptr descriptors)
{
    pcl::PointCloud<pcl::Normal>::Ptr normals(new pcl::PointCloud<PN>());
    pcl::search::KdTree<PXYZ>::Ptr kdtree(new pcl::search::KdTree<PXYZ>());
    EstimateNormalsByK(cloud, normals, 10);

    pcl::GRSDEstimation<PXYZ, PN, pcl::GRSDSignature21> grsd;
    grsd.setInputCloud(cloud);
    grsd.setInputNormals(normals);
    grsd.setSearchMethod(kdtree);
    grsd.setRadiusSearch(grsd_radius);
```

```
    grsd.compute(*descriptors);
}
```

- grsd_radius: 搜索半径，邻域半径。

3.3 匹配对应关系

已经得到model和scene的聚类点云，针对所有的聚类点云，计算对应的描述符。然后按照和局部描述符的匹配流程一样，在kdtree中搜索最近邻对应识别对，构成识别到的点云聚类correspondence，并显示识别结果。

核心代码&&参数分析（以VFH为例）：

```
void GlobalMatcher::VFHMatch(const VfhParameters& vfh_params)
{
    pcl::PointCloud<pcl::VFHSignature308>::Ptr model_descriptors(
        new pcl::PointCloud<pcl::VFHSignature308>());
    for (const auto cluster : model_clusters_) {
        pcl::PointCloud<pcl::VFHSignature308>::Ptr descriptors(
            new pcl::PointCloud<pcl::VFHSignature308>());
        CalculateVfhDescri(cluster, descriptors);
        if (descriptors->size() == 1) {
            model_descriptors->push_back(descriptors->at(0));
        } else {
            std::cout << "The VFH size in cluster cloud is not 1" <<
std::endl;
            return;
        }
    }

    pcl::PointCloud<pcl::VFHSignature308>::Ptr scene_descriptors(
        new pcl::PointCloud<pcl::VFHSignature308>());
    for (const auto cluster : scene_clusters_) {
        pcl::PointCloud<pcl::VFHSignature308>::Ptr descriptors(
            new pcl::PointCloud<pcl::VFHSignature308>());
        CalculateVfhDescri(cluster, descriptors);
        if (descriptors->size() == 1) {
            scene_descriptors->push_back(descriptors->at(0));
        } else {
            std::cout << "The VFH size in cluster cloud is not 1" <<
std::endl;
            return;
        }
    }

    pcl::KdTreeFLANN<pcl::VFHSignature308> kdtree;
    kdtree.setInputCloud(scene_descriptors);
    pcl::CorrespondencesPtr correspondences(new pcl::Correspondences());

    for (size_t i = 0; i < model_descriptors->size(); ++i) {
        std::vector<int> neighbors(1);
```

```
        std::vector<float> squaredDistances(1);

        int neighborCount = kdtree.nearestKSearch(model_descriptors->at(i),
1,
            neighbors, squaredDistances);

        if (neighborCount == 1) {
            pcl::Correspondence correspondence(
                neighbors[0], static_cast<int>(i),
                squaredDistances[0]); // [scene_index, model_index,
distance]
            correspondences->push_back(correspondence);
        }
    }

    correspondences_ = correspondences;
    CorrespondenceViewer(vfh_params.show_flag);

    return;
}
```

四. 实验测试与评估

根据局部描述符和全局描述符，分别测试局部匹配的速度与精度，全局识别的速度与精度。

4.1 数据集和实验平台

- 数据集：

  -- 1. our dataset:

MODEL:

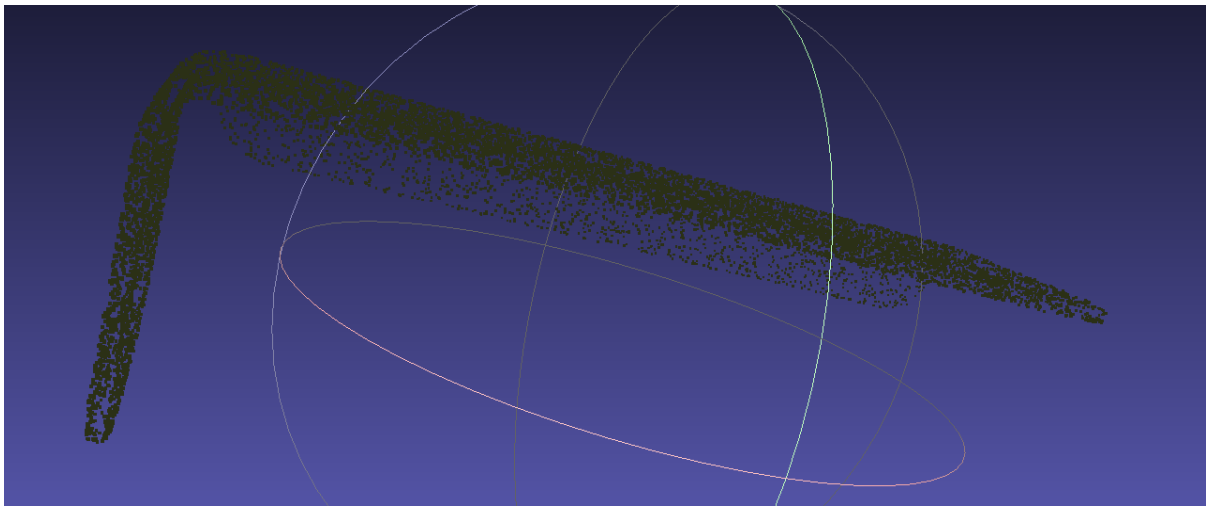


图16. L-shaped零件

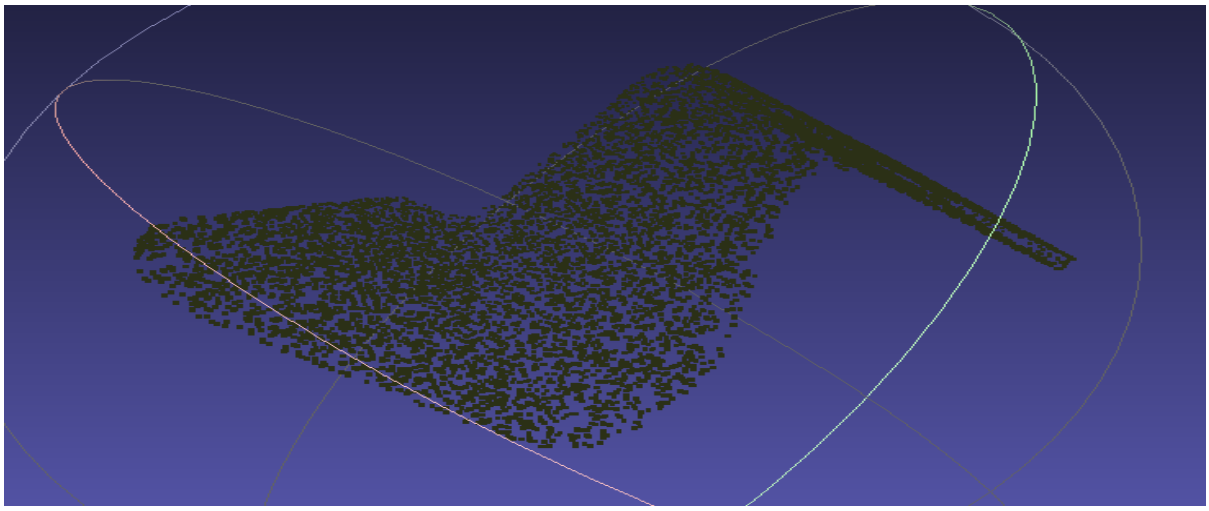


图17. S-shaped零件

SCENE:

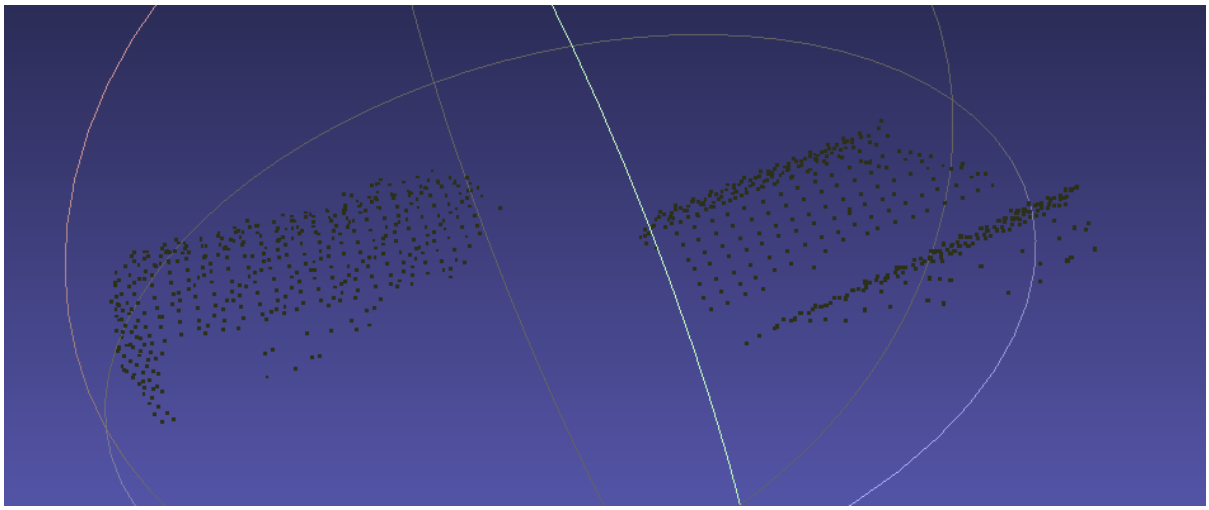


图18. 待匹配识别点云

-- 2.

-- 3.

- 实验平台：

  -- 操作系统： 基于ubuntu 18.04系统

  -- CPU： 第11代英特尔酷睿i9-11950H，基本时钟频率为2.6GHz, 8核, 16个线程

  -- RAM： 64001MB

  -- GPU： NVIDIA T1200

4.2 基于局部描述符的点云匹配

4.2.1 速度

数据集：点云点数 以计算时间为测量单位（秒s）

Descriptors / Datasets	1	2

Descriptors / Datasets	1	2
PFH	0.856	
FPFH	0.772	
RSD		
3DSC		
USC		
SHOT		
Spin Image		
RoPS		

4.2.2 精度

add: “未知变换”的实验评价

Descriptors / Datasets	1	2
PFH		
FPFH		
RSD		
3DSC		
USC		
SHOT		
Spin Image		
RoPS		

AccuracyEstimate()

4.3 基于全局描述符的点云识别

4.3.1 速度

4.3.2 精度

五. 问题与改进

estimateNormalsByK && EstimateNormals 区别

六. 总结:

该项目优势：

- 本文提供对应的参数分析，指导算法调参/提供每个方法的论文资料/参数文件global_parameters.h and local_parameters.h 方便调参匹配，拿来即用 / 比较各类匹配算法和识别算法的速度和精度
- 算法适用性
- 数据集: 更换传感器(realsense), object(size大一点), 噪声, 官方匹配数据集
- 无法判断错误识别

七. 参考

- [1] [PCL/OpenNI tutorial] (https://robotica.unileon.es/index.php?title=PCL/OpenNI_tutorial_5:_3D_object_recognition_(pipeline)#Matching)
- [2] Zhong Y. Intrinsic shape signatures: A shape descriptor for 3D object recognition[C]//2009 IEEE 12th international conference on computer vision workshops, ICCV Workshops. IEEE, 2009: 689-696.
- [3] [点云降采样] (https://blog.csdn.net/qq_39784672/article/details/125987962)
- [4] Frome, Andrea, et al. "Recognizing objects in range data using regional point descriptors." Computer Vision-ECCV 2004: 8th European Conference on Computer Vision, Prague, Czech Republic, May 11-14, 2004. Proceedings, Part III 8. Springer Berlin Heidelberg, 2004.
- [5] Tombari, Federico, Samuele Salti, and Luigi Di Stefano. "Unique shape context for 3D data description." Proceedings of the ACM workshop on 3D object retrieval. 2010.
- [6] Tombari, Federico, Samuele Salti, and Luigi Di Stefano. "Unique signatures of histograms for local surface description." Computer Vision–ECCV 2010: 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part III 11. Springer Berlin Heidelberg, 2010.
- [7] [SHOT特征描述子](https://blog.csdn.net/weixin_42192493/article/details/105900835)
- [8] Johnson, Andrew E., and Martial Hebert. "Using spin images for efficient object recognition in cluttered 3D scenes." IEEE Transactions on pattern analysis and machine intelligence 21.5 (1999): 433-449.
- [9] [特征描述 Spin Image] (https://blog.csdn.net/qq_45006390/article/details/118404128)
- [10] Guo Y, Sohel F, Bennamoun M, et al. Rotational projection statistics for 3D local surface description and object recognition[J]. International journal of computer vision, 2013, 105: 63-86.
- [11][点云配准](https://zhuanlan.zhihu.com/p/371518695)