

# hdlgraph slam 算法解析

该算法主要分为 prefiltering; floor\_detection; scan\_matching; hdl\_graph\_slam

## prefiltering.cpp

- 这部分主要是对点云实现预处理功能，主要包括点云网格滤波，点云离群点outlier的去除（这里可以选择statistic或者raduis两种方式进行滤波），以及距离滤波（只要点云在指定距离范围内的点）

## floor\_detection.cpp

- 该cpp 主要接收 prefiltering .cpp 滤波后的点云，并完成地面检测
- 主要是 boost::optional floor = detect(cloud); 函数功能
  - 可以设置tilt\_matrix对Y轴方向的激光倾斜角进行补偿
  - 紧接着，调用 filtered = plane\_clip(filtered, Eigen::Vector4f(0.0f, 0.0f, 1.0f, sensor\_height + height\_clip\_range), false); 函数，输入是滤波后的点云，输出是平面点云，vector 4f 对应的是想要的平面约束，plane clip 后续会根据平面约束滤出相关的点云
  - normal\_filtering(filtered); filtered表示平面点云，normal\_filtering 主要通过非垂直方向对点云进行滤波
  - pcl::RandomSampleConsensus ransac(model\_p); 通过ransac 函数根据输入的准平面点云进行再一次滤波，主要滤出外点，同时保留内点，并计算平面内点所对应的平面方程系数

## scan\_matching\_odometry.cpp

- 该函数主要计算相邻两帧之间的匹配，输入为prefiltering.cpp滤波后得到的/filtered\_points，紧接着进入 matching 函数

```
matching (const ros::Time& stamp, const pcl::PointCloud<PointT>::ConstPtr& cloud)
```

- step1: 首先判断是否是关键帧
- step2: 这里主要调用PCL里面现成的库函数，主要有 (ICP, GICP, NDT, GICP\_OMP, NDT\_OMP) ,值得关注的是NDT\_OMP，该算法通过采用多线程方式去完成NDT算法，因此效率是传统NDT算法的10倍，精度基本差不多；这里如果选择KDRTREE，那么和传统的NDT算法一致，剩下的速度比较： DIRECT1》 DIRECT7，但是DIRECT7更稳定同时速度比KDRTREE 快
- 选择完上述方法之后就开始完成匹配，这里主要是当前帧点云与Keyframe 进行匹配；
- 有两个参数比较重要：**keyframe\_delta\_trans** **keyframe\_delta\_angle**

```
* The minimum translational distance and rotation angle between keyframes. If this value is zero, fram
```

## hdl\_graph\_slam.cpp

- cloud\_callback(const nav\_msgs::OdometryConstPtr& odom\_msg, const sensor\_msgs::PointCloud2::ConstPtr& cloud\_msg) 该函数主要是odom信息与cloud信息的同步，同步之后检查关键帧是否更新
  - 关键帧判断 这里主要看关键帧设置的这两个阈值keyframe\_delta\_trans; keyframe\_delta\_angle; 作者认为关键帧不能设置的太近，同时也不能太远；太近的关键帧会被抛弃，如果 keyframe\_delta\_trans=0 表示关键帧之间的距离为0，因此每一帧都是关键帧，scan\_matching 函数就会变成相邻帧匹配；变成关键帧的要求就是：

```
// calculate the delta transformation from the previous keyframe
Eigen::Isometry3d delta = prev_keypose.inverse() * pose;
double dx = delta.translation().norm();
double da = std::acos(Eigen::Quaterniond(delta.linear()).w());
```

  

```
// too close to the previous frame
if(dx < keyframe_delta_trans && da < keyframe_delta_angle) {
    return false;
}
```

- optimization\_timer\_callback(const ros::TimerEvent& event) 将所有的位姿放在posegraph 中开始优化
  - loop detection 函数： 主要就是将当前帧和历史帧遍历，寻找loop
    - 寻找潜在闭环帧

```
find_candidates(const std::vector<KeyFrame::Ptr>& keyframes, const KeyFrame::Ptr& new_keyframe) const
```

- 由于每个关键帧都建立累计距离，当前帧对应的累计距离-边的累计距离 $<\text{distance\_from\_last\_edge\_thresh}$  那么就不能建立闭环，意思就是相邻两个闭环帧不能建立的过于密集
- accum\_distance\_thresh 这里指的是当前帧与临近帧的距离阈值，假设距离阈值为5m, 那么当前帧距离5米范围内关键帧不再考虑
- distance\_thresh 小于该阈值范围内两个关键帧为潜在闭环帧，将所有满足条件的都存起来作为 candidates

- 潜在闭环完成匹配 (matching 函数)

```
Loop::Ptr matching(const std::vector<KeyFrame::Ptr>& candidate_keyframes, const KeyFrame::Ptr& new_keyframe, h
```

主要是当前帧与潜在闭环帧完成点云匹配，通过统计得分寻找最优匹配，并且只将最优匹配对应的transform作为最为闭环约束，这里对应的就是fitness\_score\_thresh

- 全场亮点:计算不同loop的信息矩阵
- calc\_information\_matrix(loop->key1->cloud, loop->key2->cloud, relpose) 这里relpose 表示闭环帧之间的位姿约束
  - 首先判断是否使用常值信息矩阵，一般都是单位阵 $1/0.5=50$  作为位姿， $1/0.1=10$ 作为角度；可以看出我们更相信位姿，而非角度；这里对应变量分别是：const\_stddev\_x const\_stddev\_q
  - 这里的fitness\_socre 的计算非常朴素，主要是通过建立KD-TREE 在目标点云里面寻找source 点云每个点对应的最近点并记录两点间的距离  $\text{fitness\_score} += \text{nn\_dists}[i]; \text{return } (\text{fitness\_score} / n);$  上面这两个式子实际是记录所有点的距离，并将所有点的距离/n=平均距离； 通过求所有闭环帧KD-TREE查找对应点的平均距离来作为fitness\_score
  -