对使用 IMLS-ICP 算法进行激光匹配的代码解析

a) 先对代码的 ROS 架构进行整体分析:

首先在 main 函数内，我们创建一个 imlsDebug 的对象，

```
imlsDebug imls_debug;
```

该对象会一直等待接收来自激光雷达和轮式里程计的消息，对激光雷达前后两帧的点云进行帧间匹配，计算相对位姿，然后将激光里程计和轮子里程计的路径转发出去，便于稍后在 rviz 中显示。

进入 imlsDebug 类，我们在构造函数中，读取 bag 包 播放的数据，激光雷达消息的 topic 叫作/sick_scan，消息类型是我们自定义安装的消息类型 champion_nav_msgs::ChampionNavLaserScan 里程计消息的 topic 是/odom, 消息类型是 ros 里程计特有的消息类型 nav_msgs::Odometry。

一旦接受到消息，就执行回调函数:

```
championLaserScanCallback(scan);
odomCallback(odom);
```

我们重点关注对激光雷达点云进行帧间匹配的回调函数 championLaserScanCallback(scan)，该回调函数内，我们在对待第一帧激光点云数据，只将激光消息 msg 内的 ranges+angles 转换为激光坐标系下的二维点云并保存在 m_prevPointCloud, 并设置激光雷达里程计的初始位姿为(0,0,0)。如果不是第一帧激光点云数据，我们就将激光消息 msg 内的 ranges+angles 转换为激光坐标系下的二维点云并保存在 nowPts 内。

接下去，我们将前一帧二维激光点云数据 m_prevPointCloud 和当前帧激光点云数据 nowPts 使用 set 成员函数，传入 IMLSICPMatcher 类内，

```
//调用imls进行icp匹配，并输出结果.
m_imlsMatcher.setSourcePointCloud(nowPts);
m_imlsMatcher.setTargetPointCloud(m_prevPointCloud);
```

并使用该类的成员函数

```
m_imlsMatcher.Match(rPose,rCovariance)
```

来实现 IMLS-ICP 的点云匹配算法，变量 rPose 是当前帧激光雷达里程计在上一帧激光里程计坐标系下的相对位姿。我们在上一帧激光里程计的位姿基础上右乘该相对位姿，就构成了当前帧激光里程计的位姿:

```
Eigen::Matrix3d nowPose = lastPose * rPose;
```

之后，我们只需要将 nowPose 转化为李代数形式转发出去即可。

b) 现在重点关注 IMLSICPMatcher 类所在的源文件 imls_icp.cpp：

首先是无参构造函数 IMLSICPMatcher::IMLSICPMatcher( );我们需要初始化如下参数:

```
IMLSICPMatcher::IMLSICPMatcher(void )
{
    m_r = 0.03;
    m_h = 0.10;                                 //含义见论文. 用来计算权重.  m_r ~ 3*m_h
    m_Iterations = 10;                          // 迭代次数
    m_PtID = 0;                                 //点云的id
    m_pTargetKDTree = m_pSourceKDTree = NULL;   // 目标帧(参考帧)点云 和 当前点云的指针
}
```

在设置当前点云的成员函数内，我们将当前点云存入类成员变量 m_sourcePointCloud，并去除非法数据:

```
void IMLSICPMatcher::setSourcePointCloud(std::vector<Eigen::Vector2d> &_source_pcloud)
```

在设置目标点云的成员函数内，

```
void IMLSICPMatcher::setTargetPointCloud(std::vector<Eigen::Vector2d> &_target_pcloud)
```

我们将目标点云存入类成员变量 m_targetPointCloud，去除非法数据，并且初始化目标点云的 kd 树 m_pTargetKDTree。以下是创建目标点云 kd 树的代码:

```
//构建kd树.
if(m_pTargetKDTree == NULL)
{
    m_targetKDTreeDataBase.resize(2,m_targetPointCloud.size());    // 该矩阵用来保存二维点云数据
    for(int i = 0; i < m_targetPointCloud.size();i++)
    {
        m_targetKDTreeDataBase(0,i) = m_targetPointCloud[i](0);
        m_targetKDTreeDataBase(1,i) = m_targetPointCloud[i](1);
    }
    m_pTargetKDTree = Nabo::NNSearchD::createKDTreeLinearHeap(m_targetKDTreeDataBase);
}
```

首先将代表二维目标点云数据的类成员容器 targetPointCloud 的数据复制给矩阵 Eigen::MatrixXd m_targetKDTreeDataBase，并且使用该矩阵来设置目标点云的 kd 树 m_pTargetKDTree。这里使用了 libnabo 库, 一个在低维空间快速最近邻搜索库。当然，我们也可以使用 pcl 提供的 kd 树:

https://blog.csdn.net/luolaihua2018/article/details/117218724

两者在使用形式上是类似的，都需要先定义保存下标和保存距离平方的容器。既可以使用最近邻搜索 kdtree.nearestKSearch(), 也可以使用基于半径的搜索 kdtree.radiusSearch()。

c) 现在逐行关注 IMLSICPMatcher 类的成员函数 Match:

```
497    bool IMLSICPMatcher::Match(Eigen::Matrix3d& finalResult,
498                               Eigen::Matrix3d& covariance)
```

该成员函数串起了所有其他重要的成员函数来实现 IMLS-ICP 算法。

首先我们需要计算目标点云中每个点的法向量。该计算点云法向量的方法与 NICP 中使用特征分解协方差矩阵来计算法向量的方法一致。对目标点云中的每一个点，我们先使用我们之前已经构建好的目标点云的 kd 树来找到该点周围最近的 20 个点:

```
for(int i = 0; i < m_targetPointCloud.size();i++)
{
    Eigen::Vector2d xi = m_targetPointCloud[i];

    int K = 20;
    // Eigen数据类型定义: typedef Matrix<double, Dynamic, Dynamic> MatrixXd;   typedef Matrix<int, Dynamic, 1> VectorXi;
    Eigen::VectorXi indices(K);      // Vector<int, 20, 1>
    Eigen::VectorXd dist2(K);        // Vector<double, 20, 1>
    int num = m_pTargetKDTree->knn(xi,indices,dist2,K,0.0,
                                   Nabo::NNSearchD::SORT_RESULTS | Nabo::NNSearchD::ALLOW_SELF_MATCH|
                                   Nabo::NNSearchD::TOUCH_STATISTICS,
                                   0.15);

    std::vector<Eigen::Vector2d> nearPoints;
    for(int ix = 0; ix < K;ix++)
    {
        if(dist2(ix) < std::numeric_limits<double>::infinity() &&
                std::isinf(dist2(ix)) == false)
        {
            nearPoints.push_back(m_targetKDTreeDataBase.col(indices(ix)));
        }
        else break;
    }
```

Quit

然后调用成员函数 ComputeNormal 对该点的最近邻点计算均值和协方差，然后对协方差矩阵进行特征值分解，最小特征值对应的特征向量即为该点的法向量。成员函数 ComputeNormal 返回该法向量:

```
//根据周围的激光点计算法向量，参考ppt中NICP计算法向量的方法
// 计算均值 mu
Eigen::Vector2d mu = Eigen::Vector2d(0, 0);
for(auto it = nearPoints.cbegin(); it != nearPoints.cend(); ++it)
{
    mu += *it;
}
mu /= nearPoints.size();

// 计算协方差 cov
Eigen::Matrix2d cov = Eigen::Matrix2d::Zero();
for(int i = 0; i != nearPoints.size(); ++i)
{
    cov += (nearPoints[i] - mu) * (nearPoints[i] - mu).transpose();
}
cov /= nearPoints.size();

//对协方差矩阵进行特征分解，法向量定义为最小特征值对应的特征向量
// 计算特征值和特征向量，使用selfadjont按照对阵矩阵的算法去计算，可以让产生的vec和val按照有序排列（由小到大）
// 由于这里协方差矩阵为对称矩阵，所以可以使用Eigen::SelfAdjointEigenSolver，更方便
Eigen::SelfAdjointEigenSolver<Eigen::Matrix2d> es(cov);
Eigen::Matrix2d evs = es.eigenvectors();
normal = evs.col(0);
```

我们计算目标点云中的每一个点的法向量，并将其存入成员变量 std::vector<Eigen::Vector2d> m_targetPtCloudNormals 中，后面计算点面距离时，会用到该容器中的法向量。

然后接下来在成员函数 Match 中，我们迭代计算帧间匹配的相对位姿 result:

```
Eigen::Matrix3d result;
```

这里使用了一个 for 循环，只要没有达到终止条件，或未达到最大循环次数 m_Iterations 就会一直迭代计算下去。

```
for(int i = 0; i < m_Iterations;i++)
```

我们使用目标函数:

$$min \sum_{x_j \in S_k} \left| \vec{n_j} \cdot \left(\boldsymbol{R}\boldsymbol{x}_j + \boldsymbol{t} - \boldsymbol{y}_j \right) \right|^2$$

其中，y_j 为 x_j 在曲面上的投影点，满足如下约束:

$$\boldsymbol{y}_j = \boldsymbol{x}_j - I^{P_k}(\boldsymbol{x}_j)\vec{\boldsymbol{n}_j}$$

n_J 为 y_j 处的单位法向量，由于我们不知道投影点，也就无法求出该投影点的单位法向量。
所以我们只好拿曲面中离 x_j 最近的点的法向量代替 n_J。
其中，代表点面距离的隐函数为:

$$I^{P_k}(\boldsymbol{x}) = \frac{\sum_{\boldsymbol{p}_i \in P_k} \left(W_{\boldsymbol{p}_i}(x) \left((\boldsymbol{x} - \boldsymbol{p}_i) \cdot \vec{\boldsymbol{n}_i} \right) \right)}{\sum_{\boldsymbol{p}_j \in P_k} W_{\boldsymbol{p}_j}(x)}$$

这里 n_l 为点 p_i 处的法向量:
权重定义为:

$$W_{p_i}(x) = e^{-\|x-p_i\|^2/h^2}$$

根据以上目标函数，我们在 for 循环构建以下代码。我们先利用当前迭代步中估计的相对位姿 result, 将当前帧的点云变换到目标帧坐标系下:

```
//根据当前估计的位姿对原始点云进行转换.
std::vector<Eigen::Vector2d> in_cloud;
for(int ix = 0; ix < m_sourcePointCloud.size();ix++)
{
    Eigen::Vector3d origin_pt;
    origin_pt << m_sourcePointCloud[ix],1;

    Eigen::Vector3d now_pt = result * origin_pt;
    in_cloud.push_back(Eigen::Vector2d(now_pt(0),now_pt(1)));
}
```

然后把 sourceCloud 经过 result 变换后的点云 in_cloud 投影到 targetCloud 组成的平面上，对应的投影点为 sourceCloud 的匹配点，将其存到 ref_cloud 中，并计算该投影点的法向量。

```
std::vector<Eigen::Vector2d> ref_cloud;
std::vector<Eigen::Vector2d> ref_normal;
projSourcePtToSurface(in_cloud,
                      ref_cloud,
                      ref_normal);
```

这里调用成员函数 projSourcePtToSurface。在该成员函数内，对于 in_cloud 的每一个点，我们找到它在目标点云中最近的一个点，这里又使用了 kd 树的最近邻搜索:

```
//找到在target_cloud中的最近邻
//包括该点和下标.
int K = 1;
Eigen::VectorXi indices(K);
Eigen::VectorXd dist2(K);
m_pTargetKDTree->knn(xi,indices,dist2);

Eigen::Vector2d nearXi = m_targetKDTreeDataBase.col(indices(0));

//为最近邻计算法向量. ——进行投影的时候，使用统一的法向量.
Eigen::Vector2d nearNormal = m_targetPtCloudNormals[indices(0)];
```

如果 in_cloud 中的该点没有好的法向量定义，即法向量其中一个值无穷大或不存在定义，或者匹配点之间的平方距离 dist2 大于 m_h 的平方，即认为匹配点之间离得过远，是不好的匹配，都应当删除该匹配。使用该最近邻点处的法向量作为投影点的法向量。

```
//为最近邻计算法向量. ——进行投影的时候，使用统一的法向量.
Eigen::Vector2d nearNormal = m_targetPtCloudNormals[indices(0)];
```

接下来，我们调用函数 ImplicitMLSFunction(xi,height)，即定义隐函数来计算 in_cloud 中的点 xi 到平面的距离。在构建曲面时，我们注意到离 xi 越近的点权重越大，并且 $W_{pi}(xi)$ 在 xi 距离 pi 较远时,会迅速减小,当距离为 3h 时,权重基本为 0 了,所以我们只考虑离 xi 足够近的点来构建曲面即可。

代码中使用 kd 树来寻找目标点云中离点 xi 最近的 20 个点，构建容器 nearPoints 和 nearNormals 来保存最近点和最近点的法向量。
于是我们就可以利用之前给出的目标函数中对隐函数的定义公式来计算权重 weight 和点面距离 height:

```
//根据函数进行投影. 计算height, 即ppt中的I(x)
for(int i = 0; i < nearPoints.size(); ++i)
{
    double diff_norm = (x - nearPoints[i]).transpose() * (x - nearPoints[i]);
    double weight = std::exp(-diff_norm / m_h / m_h);
    double proj = weight * (x - nearPoints[i]).transpose() * nearNormals[i];

    weightSum += weight;
    projSum += proj;
}

height = projSum / (weightSum + 1e-5);
```

回到成员函数 projSourcePtToSurface 内，我们在已知投影点法向量 nearNormal，点面距离 height 情况下，就可以计算投影点 yi 了:

```
//计算yi.
yi = xi - height * nearNormal;
```

我们将当前点云对应在曲面上的投影点保存在 ref_cloud 内，将投影点的法向量保存在 ref_normal 内，作为参数传递给成员函数 SolveMotionEstimationProblem，用来求解帧间相对位姿的变化 deltaTrans:

```
//计算帧间位移. 从当前的source -> target
Eigen::Matrix3d deltaTrans;
bool flag = SolveMotionEstimationProblem(in_cloud,
                                         ref_cloud,
                                         ref_normal,
                                         deltaTrans);
```

在给定经估计的变换矩阵 result 变换后的点 in_cloud 和投影点 ref_cloud 情况下,最小化点面距离

$$\min_{T_k}\sum_i\left\|x_j^i - T_k p_i\right\|^2$$

来求出变换矩阵，这就和 PL-ICP 的算法流程一样了:

算法流程

1. 把当前帧的数据p_i根据初始位姿进行变换，得到$T_k p_i$；
2. 对于当前帧中的点 $T_k p_i$，在参考帧中找到最近的两个点 $x_{j_1}^i, x_{j_2}^i$，并计算直线法向量 n_i^T 和投影点 x_j^i;
3. 计算点$T_k p_i$与投影点x_j^i的距离，去除误差过大的点;
4. 最小化误差函数 $\min_{T_k}\sum_i\left\|x_j^i - T_k p_i\right\|^2$

我们可以使用 LM 等优化算法来解 deltaTrans, 就需要计算雅可比矩阵。但是在匹配关系已知的情况下，我们可以使用闭式解解得 deltaTrans。SolveMotionEstimationProblem 函数内使用了论文[2]中给出的利用拉格朗日算子算出的闭式解。解得闭式解 deltaTrans 后，我们更新相对位姿 result, 然后判断 deltadist 和 deltaAngle 是否满足迭代停止条件。若不满足，就再次进行新的一轮匹配，再次解出新的 deltaTrans，直到收敛:

```
//更新位姿.
result = deltaTrans * result;

//迭代条件是否满足.
double deltadist = std::sqrt( std::pow(deltaTrans(0,2),2) + std::pow(deltaTrans(1,2),2));
double deltaAngle = std::atan2(deltaTrans(1,0),deltaTrans(0,0));

//如果迭代条件允许，则直接退出.
if(deltadist < 0.001 && deltaAngle < (0.01/57.295))
{
    break;
}
```

至此，就是 IMLS-ICP 算法的全部代码实现。

文献参考:

[1] Least-Squares Fitting of Two 3-D Points Sets, K.S. ARUN, T.S. HUANG, S.D. BLOSTEIN.

[2] An ICP variant using a point-to-line metric, Andrea Censi

[3] NICP: Dense Normal Based Point Cloud Registration, Jacopo Serafin and Giorgio Grisetti

[4] Provably Good Moving Least Squares, Ravikrishna Kolluri

[5] IMLS-SLAM: scan-to-model matching based on 3D data, Jean-Emmanuel Deschaud