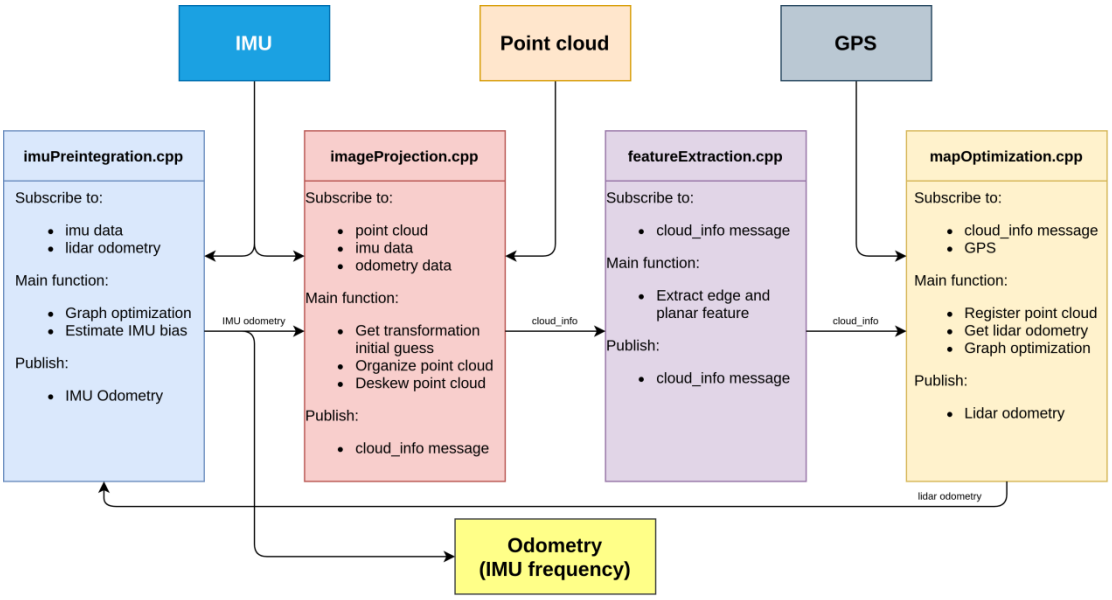
LIO-SAM 代码解析

徐胜攀

Github

https://github.com/TixiaoShan/LIO-SAM

LIO-SAM 的代码工程结构



系统所需要的 sensor 包括：

IMU、 Lidar (point cloud)、GPS（optional）

系统的输出：

Odometry（以 IMU 频率输出里程计位姿、速度等状态信息）

系统的 4 个模块：

- imuPreintegration：基于 IMU 量测和激光里程计构建 IMU 预积分约束，
- imageProjection：基于 imu odometry 进行点云去畸变
- featureExtraction：点云特征提取
- mapOptmization：基于因子图的回环检测和后端优化

1. imuPreintegration

主函数：

```C++
int main(int argc, char** argv)
{
    ros::init(argc, argv, "roboat_loam");

    IMUPreintegration ImuP;

    TransformFusion TF;

    ROS_INFO("\033[1;32m----> IMU Preintegration
Started.\033[0m");

    ros::MultiThreadedSpinner spinner(4);
    spinner.spin();

    return 0;
}
```

主要包含两个类，IMUPreintegration 和 TransformFusion。

IMUPreintegration 用于与预积分相关的计算和优化，TransformFusion 进行位姿的融合和发布。

IMUPreintegration

利用因子图 gtsam::ISAM2 进行预积分相关的计算和优化，这部分实际上是一个 IMU Odometry。

主要了解构造函数及两个重要的回调函数。

构造函数中，主要注册接收和发布对象，和 IMU 噪声、预积分噪声等相关的协方差矩阵的初始化。

订阅：

imuTopic

lio_sam/mapping/odometry_incremental

发布：

imu_incremental

优化器相关：

```C++
        gtsam::ISAM2Params optParameters;
```

```
        optParameters.relinearizeThreshold = 0.1;
        optParameters.relinearizeSkip = 1;
        optimizer = gtsam::ISAM2(optParameters);

        gtsam::NonlinearFactorGraph newGraphFactors;
        graphFactors = newGraphFactors;

        gtsam::Values NewGraphValues;
        graphValues = NewGraphValues;
```

其中 gtsam::ISAM2 为贝叶斯树，为主要优化器，提供优化方法；gtsam::NonlinearFactorGraph 为因子图，包含约束关系（factor）和优化变量（values）。

odometryHandler

是紧耦合的关键函数。注意 gtsam 进行预积分紧耦合优化的方法。

接收 mapOptimization 模块发送的 lidar 位姿，构建 IMU 预积分约束，并紧耦合 Lidar 和 IMU 进行优化。

紧耦合中待估计状态包括:

$$\mathbf{x} = [\,\mathbf{R}^{\mathbf{T}},\ \mathbf{p}^{\mathbf{T}},\ \mathbf{v}^{\mathbf{T}},\ \mathbf{b}^{\mathbf{T}}\,]^{\mathbf{T}}, \qquad (1)$$

这些状态都是 IMU 的状态，要获取 Lidar 状态，只需乘以 IMU 到 Lidar 的外参。

步骤:

1）获取初始 lidar pose，并记录当前帧时间

```
C++
gtsam::Pose3 lidarPose = gtsam::Pose3(gtsam::Rot3::Quaternion(r_w,
r_x, r_y, r_z), gtsam::Point3(p_x, p_y, p_z));
double currentCorrectionTime = ROS_TIME(odomMsg);
```

2）如果系统未初始化，进行初始化并返回

初始化实际上是在 gtsam 优化器中设置状态初值及其协方差，并初始化预积分对象的 IMU bias。

初始化完成后，记录关键帧的数量为 1: key = 1

3）当关键帧数量达到 100 帧时，对因子图进行重置

以最后一帧的状态及协方差作为先验状态重新初始化因子图，并记录关键帧数量 key=1

4) 对当前帧之前的 IMU 消息进行积分和退队

```C++
double dt = (lastImuT_opt < 0) ? (1.0 / 500.0) : (imuTime -
lastImuT_opt);
imuIntegratorOpt_->integrateMeasurement(gtsam::Vector3(thisImu-
>linear_acceleration.x,           thisImu->linear_acceleration.y,
thisImu->linear_acceleration.z),
gtsam::Vector3(thisImu->angular_velocity.x, thisImu-
>angular_velocity.y,
     thisImu->angular_velocity.z), dt);

lastImuT_opt = imuTime;
imuQueOpt.pop_front();
```

5）基于 IMU factor 进行因子图优化，获取预测的状态

具体步骤参见源码。其中有一个关键步骤是，将 IMU 积分对象转换成预积分对象，然后连同上一帧的状态和当前帧状态一起，通过构造函数构建一个 gtsam::ImuFactor。

6）检查优化是否失败，如果失败，则对系统进行重置并返回

重置会标识系统未初始化，重新开始。

7）退队旧的 IMU 量测，更新当前帧状态，重新计算积分（因为 bias 改变了）

8）帧号加 1，返回。

failureDetection

当速度大于 30m/s，或者 bias 大于 1 时，则认为优化失败。

imuHandler

订阅和处理 IMU 消息：将 IMU 消息加入消息队列，进行 IMU 状态积分，发布高频 lidar 里程计位姿和速度。

```C++
imuIntegratorImu_-
>integrateMeasurement(gtsam::Vector3(thisImu.linear_acceleration.x
, thisImu.linear_acceleration.y, thisImu.linear_acceleration.z),

gtsam::Vector3(thisImu.angular_velocity.x,
thisImu.angular_velocity.y,    thisImu.angular_velocity.z), dt);

// predict odometry
```

```
gtsam::NavState currentState = imuIntegratorImu_-
>predict(prevStateOdom, prevBiasOdom);

// publish odometry
nav_msgs::Odometry odometry;
odometry.header.stamp = thisImu.header.stamp;
odometry.header.frame_id = odometryFrame;
odometry.child_frame_id = "odom_imu";

pubImuOdometry.publish(odometry);
```

这里发布的是高频的 lidar 位姿。

核心是，在 odometryHandler 中，通过预积分约束，对因子图进行优化，得到优化后的 imu bias，并重新进行积分，得到 IMU 位姿，然后，利用 imu 到 lidar 外参，转换成 lidar 位姿，从而发布高频的 lidar 位姿（imu_incremental）。

TransformFusion

订阅:

lio_sam/mapping/odometry: lidarOdometryHandler

imu_incremental: imuOdometryHandler

发布:

lio_sam/imu/path

imu_odometry

这一部分是融合 lidar 后端优化位姿与 imu 里程计位姿，发布高频定位信息 （IMU frequency localization）。

lidarOdometryHandler

接收 lidar mapping 位姿，存入：lidarOdomAffine，lidarOdomTime

imuOdometryHandler

根据激光里程计的低频但更精确的输出(map 坐标系，即经过了回环检测约束和 GPS 约束后，得到的全局坐标系)，将 IMU 里程计高频但存在漂移的输出（odometry 坐标系）去累积漂移，转换到 map 坐标系，获得高频且更准确的高频 lidar 位姿输出，并输出 base 的位姿，发布 IMU 轨迹。

$$T_{back}^{m} = T_{front}^{m} T_{back}^{front}$$

$$T_{back}^{front} = \left(T_{front}^{o}\right)^{-1} T_{back}^{o}$$

最新时刻 lidar 位姿：

```
C++
Eigen::Affine3f imuOdomAffineFront =
odom2affine(imuOdomQueue.front());
Eigen::Affine3f imuOdomAffineBack =
odom2affine(imuOdomQueue.back());
Eigen::Affine3f imuOdomAffineIncre = imuOdomAffineFront.inverse()
* imuOdomAffineBack;
Eigen::Affine3f imuOdomAffineLast = lidarOdomAffine *
imuOdomAffineIncre;
```

最新时刻 baselink tf：

```
C++
tCur = tCur * lidar2Baselink;
tf::StampedTransform odom_2_baselink = tf::StampedTransform(tCur,
odomMsg->header.stamp, odometryFrame, baselinkFrame);
```

总结

本模块中，imuPreintegration 接收 imu 原始数据(imu_topic)，以及 lidar 里程计数据(lio_sam/mapping/odometry_incremental)，构建预积分约束进行优化，得到优化后的 imu bias 和优化后的 imu 位姿。

优化后的 imu 位姿为高频率低精度的位姿，实际处理中转化为 lidar pose 发布（imu_incremental）。

TransformationFusion 中，接收高频低精的 lidar pose（imu_incremental）和后端优化得到的低频高精的 lidar pose (mapping/odometry)，消除累积漂移，得到高频高精的 lidar pose 预测(imu_odometry)。

2. imageProjection

主函数

```
C++
int main(int argc, char** argv)
{
```

```
    ros::init(argc, argv, "lio_sam");

    ImageProjection IP;

    ROS_INFO("\033[1;32m----> Image Projection Started.\033[0m");

    ros::MultiThreadedSpinner spinner(3);
    spinner.spin();

    return 0;
}
```

ImageProjection

构造函数

订阅：

imu_topic: imuHandler　原始 imu topic

imu_incremental: odometryHandler　高频低精 lidar pose，来自于外参+imu 积分

pointCloudTopic：cloudHandler　原始点云

发布：

deskew/cloud_deskewed：　pubExtractedCloud 去畸变点云

deskew/cloud_info：pubLaserCloudInfo　结构化点云

imuHandler

接收 imu 信息并存入队列：imuQueue

odometryHandler

接收里程计信息存入队列：odomQueue

cloudHandler

```
C++
void cloudHandler(const sensor_msgs::PointCloud2ConstPtr&
```

```
laserCloudMsg)
    {
        if (!cachePointCloud(laserCloudMsg))
            return;
        if (!deskewInfo())
            return;
        projectPointCloud();
        cloudExtraction();
        publishClouds();
        resetParameters();
    }
```

bool cachePointCloud(const sensor_msgs::PointCloud2ConstPtr& laserCloudMsg)

点云格式标准化，记录点云起止时间，找到 ring 通道，检查是否有时间戳

1) 将点云数据加入队列尾

2）取队首点云，转化为 PointXYZIRT 格式的点云，存到 laserCloudIn

3）记录点云的起止时间：timeScanCur，timeScanEnd

4）找到 ring 通道 ringFlag

5) 检查是否有时间戳通道 deskewFlag

void imuDeskewInfo()

1）将旧的 imu 信息退出队列

2）imu 旋转积分，计算各帧 imu 的姿态：imuRotX，imuRotY，imuRotZ

void odomDeskewInfo()

1）获取帧起诉时刻位姿：startOdomMsg

2）起诉时刻位姿作为该帧点云的初始估计位姿：initialGuess： X Y Z Roll Pitch Yaw

3) 获取帧结束时刻位姿：endOdomMsg

4）计算结束时刻相对于起始时刻的运动：transBt = transBegin.inverse() * transEnd;

void projectPointCloud()

将点云投影到 rangeImage（并降采样），并对点云去畸变（补偿相对于第一个点的运动增量），记录 rangeImage 中各像素的 range 和各点在 rangeImage 中的像素位置。

点去畸变:

计算当前时刻相对于起始时刻的变换，然后将点变换到起始时刻的坐标系。

这里完全用 imu 量测进行点云去畸变。在去畸变时，仅补偿了旋转带来的畸变，这是因为，一方面，旋转造成的影响可能比平移大得多，另一方面，imu 旋转的量测（角速度）比平移的量测（线加速度）更可靠。

void cloudExtraction()

将点云结构化，得到结构化后的点云 extractedCloud 及其结构化信息 cloudInfo。

结构化信息包括：各行的起始点序号（ cloudInfo.startRingIndex[i]）和终止点序号（ cloudInfo.endRingIndex[i]），点所在的列号，点的距离(range)。

void publishClouds()

发布结构化点云及其结构化信息 cloud_info。

总结

本模块进行点云去畸变和点云结构化（整理 range image 各行各列对应的点），发布结构化点云。

3. featureExtraction

主函数

```
C++
int main(int argc, char** argv)
{
    ros::init(argc, argv, "lio_sam");

    FeatureExtraction FE;

    ROS_INFO("\033[1;32m----> Feature Extraction
Started.\033[0m");
```

```
    ros::spin();

    return 0;
}
```

本模块主要进行 loam 特征提取，主要是计算各点的曲率，提取线状点和平面点。

订阅：

deskew/cloud_info：laserCloudInfoHandler，来自 ImageProjection 模块发布的结构化点云

发布：

feature/cloud_info：pubLaserCloudInfo，特征点云信息

feature/cloud_corner：pubCornerPoints，线状特征点

feature/cloud_surface：pubSurfacePoints，平面特征点

void laserCloudInfoHandler(const lio_sam::cloud_infoConstPtr& msgIn)

```
C++
void laserCloudInfoHandler(const lio_sam::cloud_infoConstPtr&
msgIn)
    {
        cloudInfo = *msgIn; // new cloud info
        cloudHeader = msgIn->header; // new cloud header
        pcl::fromROSMsg(msgIn->cloud_deskewed, *extractedCloud);
// new cloud for extraction

        calculateSmoothness();  //计算曲率
        markOccludedPoints();   //排除遮挡点
        extractFeatures();      //提取特征点
        publishFeatureCloud();  //发布特征点云
    }
```

void calculateSmoothness()

计算曲率。

每个点的曲率取该扫描线上第 i 个点左右 2k 个点，即[i-k, i-(k-1), ..., i, ..., i+(k-1), i+k]，计算 diffRange 的平方。

void markOccludedPoints()

标记遮挡点和离群点：cloudNeighborPicked[i]=1

原理：

1）如果列号差别较小（小于 10），但深度差较大（大于 0.3），则可能存在遮挡，标记深度较大一侧的点为背景点并挑选出来；

2）计算转角，ratio=diff_depth/depth，如果两侧深度差异都大于 2%，表明当前点是遮挡物点。

void extractFeatures()

处理各行(scan line)，将各行点分成 6 份，对每份点云，按曲率从小到大排序，将曲率大于 edgeThreshold 的点标记为线状点（cloudLabel[ind] = 1），将曲率小于 surfThreshold 的点标记为面状点（cloudLabel[ind] = -1）。

每个区间内的特征点数量小于 20。同时，将特征点及特征点附近相邻的像素标记为已挑选（维持特征点的稀疏性并避免重复挑选）。

最后，对特征点进行降采样，flat points 为 0.4， edge points 为 0.2。

lio-sam 取消了 LOAM 中显著平面点和显著边界点的处理。

对 scan line i，第 j 份的起始点 id:

```C++
int sp = (cloudInfo.startRingIndex[i] * (6 - j) +
cloudInfo.endRingIndex[i] * j) / 6;
```

$$s = (r_e - r_s)/6 * j + r_s = \frac{jr_e + (6-j)r_s}{6}$$

$$e = s + (r_e - r_s)/6 - 1 = \frac{jr_e + (6-j)r_s + r_e - r_s}{6} - 1 = \frac{(j+1)r_e + (5-j)r_s}{6} - 1$$

总结

本模块主要进行特征提取，得到 flat points 和 edge points。为维持特征点的均衡性（分成 6 个区间）、稳定性（去除 Occluded Points）、稀疏性等做了相应的处理。

4. mapOptmization

主要进行当前帧到地图的匹配和回环检测优化。

主函数

```C++
int main(int argc, char** argv)
{
    ros::init(argc, argv, "lio_sam");

    mapOptimization MO;

    ROS_INFO("\033[1;32m----> Map Optimization Started.\033[0m");

    std::thread loopthread(&mapOptimization::loopClosureThread,
&MO);
    std::thread
visualizeMapThread(&mapOptimization::visualizeGlobalMapThread,
&MO);

    ros::spin();

    loopthread.join();
    visualizeMapThread.join();

    return 0;
}
```

构造函数

1）初始化 ISAM2

2）注册订阅者和发布者

3）初始化 down size filter，包括 downSizeFilterCorner，downSizeFilterSurf，downSizeFilterICP，downSizeFilterSurroundingKeyPoses

4）初始化点云容器、KDTree 等其他资源。

订阅

feature/cloud_info：特征点云，mapOptimization::laserCloudInfoHandler

gpsTopic：gps 消息，mapOptimization::gpsHandler

lio_loop/loop_closure_detection：回环检测消息，mapOptimization::loopInfoHandler

发布

mapping/odometry_incremental：里程计增量位姿，相当于帧间位姿，来自于帧到局部地图的匹

配，pubLaserOdometryIncremental

mapping/odometry：全局优化位姿，pubLaserOdometryGlobal

mapping/trajectory：pubKeyPoses

mapping/map_global：pubLaserCloudSurround

laserCloudInfoHandler

```
C++
void laserCloudInfoHandler(const lio_sam::cloud_infoConstPtr&
msgIn)
    {
        // extract time stamp
        timeLaserInfoStamp = msgIn->header.stamp;
        timeLaserInfoCur = msgIn->header.stamp.toSec();

        // extract info and feature cloud
        cloudInfo = *msgIn;
        pcl::fromROSMsg(msgIn->cloud_corner,
*laserCloudCornerLast);
        pcl::fromROSMsg(msgIn->cloud_surface,
*laserCloudSurfLast);

        std::lock_guard<std::mutex> lock(mtx);

        static double timeLastProcessing = -1;
        if (timeLaserInfoCur - timeLastProcessing >=
mappingProcessInterval)
        {
            timeLastProcessing = timeLaserInfoCur;
            updateInitialGuess();
            extractSurroundingKeyFrames();
            downsampleCurrentScan();
            scan2MapOptimization();
            saveKeyFramesAndFactor();
            correctPoses();
            publishOdometry();
            publishFrames();
        }
    }
```

1) 记录当前帧时间：timeLaserInfoCur

2）获取当前帧特征点云：laserCloudSurfLast，laserCloudCornerLast

3）updateInitialGuess：更新初始估计

4）extractSurroundingKeyFrames：提取附近的关键帧

5）downsampleCurrentScan：当前帧下采样

6）scan2MapOptimization：当前帧到局部地图的匹配

7）saveKeyFramesAndFactor：保留关键帧，并保留 odometry factor 到因子图

8）correctPoses：回环检测优化

9）publishOdometry：发布里程计

10）publishFrames：发布当前帧数据

void updateInitialGuess()

transformTobeMapped：上一帧到 global map 的变换，T_{i-1}^{m}

lastImuPreTransformation：上一帧到 odometry 的变换，T_{i-1}^{o}

lastImuTransformation：旋转部分等效于T_{i-1}^{o}，平移部分为 0

transBack：当前帧到 odometry 的变换，T_{i}^{o}

transIncre：上一帧到当前帧的运动增量，T_{i}^{i-1}

transFinal：当前帧到 global map 的变换，T_{i}^{m}

$$T_{i}^{m} = T_{i-1}^{m} T_{i}^{i-1}$$

更新后：

lastImuPreTransformation：T_{i}^{o}

transformTobeMapped：T_{i}^{m}

这一部分是要更新 transformTobeMapped，即根据上一帧的 global pose 和当前帧的运动增量，来计算当前帧的 global pose，并重新赋给 transformTobeMapped。

在当前帧的运动增量的计算时，如果当前帧有 lidar odometry pose，那么根据上一帧的 lidar odometry pose 计算运动增量；否则，如果有 imu odometry pose，那么根据 imu odometry pose 计算运动增量，但此时只考虑更新旋转部分。

lastImuTransformation 用于只有 IMU 时的估计。如果只有 IMU，那么计算 transFinal 时只对旋转部分进行更新，因为 IMU 旋转部分的量测更可靠。

extractSurroundingKeyFrames()

搜索当前帧附近的关键帧。搜索当前帧附近一定半径范围内(代码中是 50 米)的所有帧，并降采样（降采样分辨率是 1 米），然后将降采样后的各帧附近 10s 范围内的帧也加入 surroundingKeyPosesDS，然后再提取对应点云。

注意用 intensity 存储了点云的帧号。

这里对 SurroundingKeyFrames 的处理与 paper 中不太一样，paper 中是通过当前帧的前 n 帧构建局部地图，n=25。

> extract the n most recent keyframes, which we call the *sub-keyframes*, for estimation. The set of sub-keyframes $\{\mathbb{F}_{i-n}, ..., \mathbb{F}_i\}$ is then transformed into frame $\mathbf{W}$ using the transformations $\{\mathbf{T}_{i-n}, ..., \mathbf{T}_i\}$ associated with them. The transformed sub-keyframes are merged together into a voxel map $\mathbf{M}_i$. Since we extract two types of features in the

void extractCloud(pcl::PointCloud<PointType>::Ptr cloudToExtract)

PointType 里主要存了当前帧点云位置(x,y,z)和帧号（intensity，for KeyInd）。

将点云合并到 laserCloudCornerFromMap 和 laserCloudSurfFromMap，并降采样（降采样距离 surf 0.4, corner 0.2），存储到 laserCloudSurfFromMapDS、laserCloudCornerFromMapDS。

这里还用了一个 laserCloudMapContainer 避免点云重复转换。

注意：code 中用 cornerCloudKeyFrames、surfCloudKeyFrames 保留所有单帧数据，这是可能是一个问题，不可能通过内存保留所有点云数据，尽管是特征点云。

downsampleCurrentScan()

对当前帧下采样，存储到 laserCloudCornerLastDS、laserCloudSurfLastDS。

scan2MapOptimization()

```
C++
kdtreeCornerFromMap->setInputCloud(laserCloudCornerFromMapDS);
            kdtreeSurfFromMap-
>setInputCloud(laserCloudSurfFromMapDS);

            for (int iterCount = 0; iterCount < 30; iterCount++)
```

```
            {
                laserCloudOri->clear();
                coeffSel->clear();

                cornerOptimization();
                surfOptimization();

                combineOptimizationCoeffs();

                if (LMOptimization(iterCount) == true)
                    break;
            }

            transformUpdate();
```

迭代进行 2 步法 LM 优化，直到收敛，优化的方法是首先进行特征匹配、计算残差，计算雅克比，然后进行高斯牛顿法优化，完成优化后，更新当前帧位姿。

void cornerOptimization()

1）将 souce cloud（laserCloudCornerFromMapDS）内的点变换到地图坐标系

pointAssociateToMap(&pointOri, &pointSel);

2）在 map cloud 上，搜索距离 source point 最近的 5 个点；

3）如果第 4 个点距离小于 1 米，则表明存在正确的匹配，进一步计算协方差和进行特征分解，得到 target 直线的中心点和方向

4）计算残差(ld2)、雅克比（残差相对于变换后点的雅克比，点到直线的距离对应的单位向量，(la, lb, lc)）和权重（距离越大，权重越小）,结果存入 coeffSelCornerVec

void surfOptimization()

与 cornerOptimization 过程类似。

雅克比：残差相对于变换后点的雅克比，点到平面的距离对应的单位向量，即平面法向量。

void combineOptimizationCoeffs()

将 edge 点残差和雅克比与 surface 点残差和雅克比加入 laserCloudOriSurfFlag，后续进行非线性优化。

bool LMOptimization(int iterCount)

高斯牛顿非线性优化求解位姿。

$$J^T J \Delta x = -J^T e$$

matA：nx6 向量，对应 J

matB：nx1 向量，对应-e

在求解过程中要防止解退化（degenerate）情形。方法是在第一次迭代时，计算 matAtA 的特征值，如果某特征值小于 100，则表明该行存在退化，此时，将特征向量置为 0，重新求解协方差矩阵：

matP = matV.inv() * matV2

对解进行改正：

matX = matP * matX2;

这将使得退化的行不更新。

void transformUpdate()

对点云匹配结果进行改正，确保 pitch、roll、z（相对前一帧的位姿）不会超过一定范围。并将变换结果存入 incrementalOdometryAffineBack，得到增量位姿（这实际上就是当前帧与地图匹配的结果）。

void saveKeyFramesAndFactor()

这一步执行 isam 层面的大的优化。将里程计因子(odom factor)、GPS 因子(GPS factor)、回环因子（loop closure factor）加入因子图 gtsam，最后放入 isam 中进行全局优化。

最后，将全局优化后的位姿 latestEstimate 加入 cloudKeyPoses3D（用于后续 KDTree 搜索）和 cloudKeyPoses6D（最新的优化结果），记录该帧优化的边缘化残差（poseCovariance）。

更新当前帧的后验 pose：transformTobeMapped

GPS 因子通过订阅 GPS 信息构建；

回环因子通过订阅回环检测信息得到，有一个单独的线程进行回环检测。

```C++
gtSAMgraph.add(BetweenFactor<Pose3>(cloudKeyPoses3D->size()-1,
cloudKeyPoses3D->size(), poseFrom.between(poseTo),
odometryNoise));
initialEstimate.insert(cloudKeyPoses3D->size(), poseTo);

// GPS too noisy, skip
gtsam::GPSFactor gps_factor(cloudKeyPoses3D->size(),
```

```
gtsam::Point3(gps_x, gps_y, gps_z), gps_noise);
```

void correctPoses()

isam 优化后，更新所有 pose。

```
C++
laserCloudMapContainer.clear();
            // clear path
globalPath.poses.clear();
            // update key poses
int numPoses = isamCurrentEstimate.size();
for (int i = 0; i < numPoses; ++i)
{
   cloudKeyPoses3D->points[i].x =
isamCurrentEstimate.at<Pose3>(i).translation().x();
   cloudKeyPoses3D->points[i].y =
isamCurrentEstimate.at<Pose3>(i).translation().y();
   cloudKeyPoses3D->points[i].z =
isamCurrentEstimate.at<Pose3>(i).translation().z();

   cloudKeyPoses6D->points[i].x = cloudKeyPoses3D->points[i].x;
   cloudKeyPoses6D->points[i].y = cloudKeyPoses3D->points[i].y;
   cloudKeyPoses6D->points[i].z = cloudKeyPoses3D->points[i].z;
   cloudKeyPoses6D->points[i].roll  =
isamCurrentEstimate.at<Pose3>(i).rotation().roll();
   cloudKeyPoses6D->points[i].pitch =
isamCurrentEstimate.at<Pose3>(i).rotation().pitch();
   cloudKeyPoses6D->points[i].yaw   =
isamCurrentEstimate.at<Pose3>(i).rotation().yaw();

   updatePath(cloudKeyPoses6D->points[i]);
}
```

void loopClosureThread()

```
C++
void loopClosureThread()
    {
        if (loopClosureEnableFlag == false)
```

```
            return;

        ros::Rate rate(loopClosureFrequency);
        while (ros::ok())
        {
            rate.sleep();
            performLoopClosure();
            visualizeLoopClosure();
        }
    }
```

一个单独的线程进行回环检测。

void performLoopClosure()

1) 检测回环：loopKeyPre， loopKeyCur

2) 构建局部地图并进行匹配: correctionLidarFrame=icp(prevKeyframeCloud, cureKeyframeCloud)

注意 prevKeyframeCloud 是全局坐标系的点云，cureKeyframeCloud 是 odometry 附近全局坐标系的点云（odometry 系下的点云）

3) 通过 fitness score 检查是否为有效回环，如果 invalid 则返回;

4）得到当前帧改正位姿：$T_i^m = T_o^m T_i^o$

Eigen::Affine3f tCorrect = correctionLidarFrame * tWrong;

5）得到回环约束:

```
C++
gtsam::Pose3 poseFrom = tCorrect;
gtsam::Pose3 poseTo = copy_cloudKeyPoses6D->points[loopKeyPre];
gtsam::Vector Vector6(6);
float noiseScore = icp.getFitnessScore();
Vector6 << noiseScore, noiseScore, noiseScore, noiseScore,
noiseScore, noiseScore;
noiseModel::Diagonal::shared_ptr constraintNoise =
noiseModel::Diagonal::Variances(Vector6);

loopIndexQueue.push_back(make_pair(loopKeyCur, loopKeyPre));
loopPoseQueue.push_back(poseFrom.between(poseTo));
loopNoiseQueue.push_back(constraintNoise);

loopIndexContainer[loopKeyCur] = loopKeyPre;
```

detectLoopClosureDistance

参数:

```
C++
nh.param<bool>("lio_sam/loopClosureEnableFlag",
loopClosureEnableFlag, false);
nh.param<float>("lio_sam/loopClosureFrequency",
loopClosureFrequency, 1.0);
nh.param<int>("lio_sam/surroundingKeyframeSize",
surroundingKeyframeSize, 50);
nh.param<float>("lio_sam/historyKeyframeSearchRadius",
historyKeyframeSearchRadius, 10.0);
nh.param<float>("lio_sam/historyKeyframeSearchTimeDiff",
historyKeyframeSearchTimeDiff, 30.0);
nh.param<int>("lio_sam/historyKeyframeSearchNum",
historyKeyframeSearchNum, 25);
nh.param<float>("lio_sam/historyKeyframeFitnessScore",
historyKeyframeFitnessScore, 0.3);
```

逻辑:

查找满足条件的第一个回环。只找一个。

搜索 historyKeyframeSearchRadius（10 米）范围的历史 pose 点，如果存在时间间隔大于 30 秒的帧，则认为找到了回环，返回。

总结

本部分进行后端优化，即进行当前帧当局部地图的匹配和存在 gps 约束、回环检测约束时的全局优化。

大致过程为:

1）单独的线程进行回环检测，当发现回环时，添加回环约束到队列

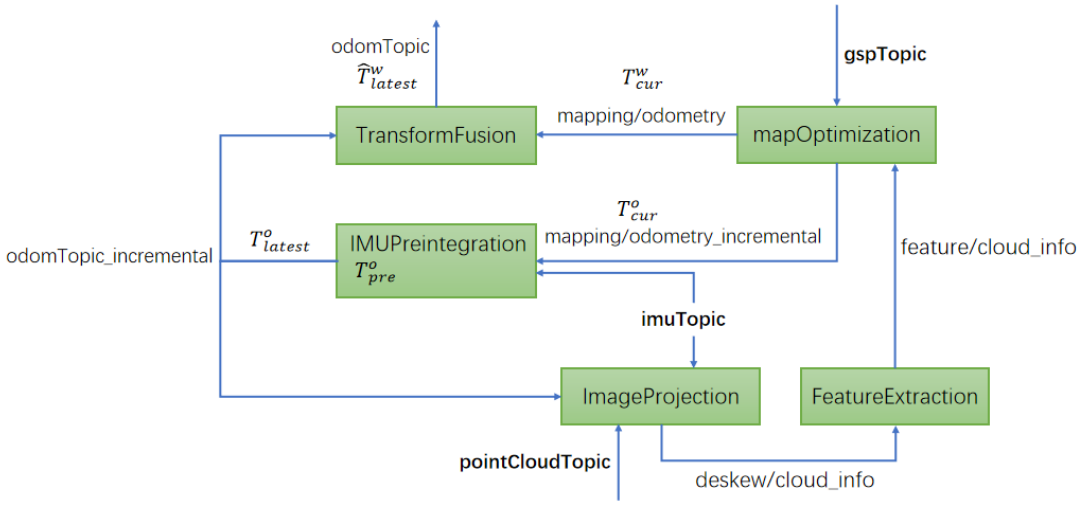
2）GPS 接收回调接收 GPS 信息并加入队列

3）lidar 点云接收回调接收 lidar 特征点云，进行当前帧到局部地图的匹配，得到初始全局位姿

4）全局因子图优化：将 odom factor、loop closure factor、gps factor 加入因子图并进行全局优化

5）改正所有历史帧位姿，并发布高精度的建图和定位结果。

总结



主要过程:

1）ImageProject 接收 IMU 量测和 Lidar 点云，进行点云去畸变，并进行点云结构化（将点云按行和列进行整理），同时接收 IMU 里程计结果得到点云初始位姿(initialGuess)；

2）FeatureExtraction 进行特征提取，处理过程中关注了特征点的均衡性（分成 6 个区间）、稳定性（去除 Occluded Points）、稀疏性等；

3）MapOptimization 模块接收特征点云，进行当前帧到局部地图的匹配，并进一步进行回环检测和 GPS 位置约束，进行全局因子图优化，得到精确的全局位姿；

4）IMUPreintegration 模块接收 IMU 和连续帧 lidar 位姿构建预积分约束，优化 IMU bias，并据此重新进行 IMU 积分，得到高频低精 pose；

5）TransformationFusion 中，接收高频低精的 lidar pose（imu_incremental）和后端优化得到的低频高精的 lidar pose (mapping/odometry)，消除累积漂移，得到高频高精的 lidar pose 预测 (imu_odometry)，从而持续发布高频高精的定位预测。