第一章 多传感器融合基础知识

第一节 什么是多传感器融合

什么是slam

SLAM全称是Simultaneous Localization And Mapping，及同时定位和建图

多传感器融合的目的就是为了做SLAM

具体一点，就是更**好**的去定位和建图

这里再把好具体一点，就理解为更精确的定位和建图精度，以及更优的鲁棒性

多传感器融合

slam领域中的多传感器融合简单来说就是融合多个传感器的数据来获得更准确的定位信息和建图结果

第二节 为什么需要多传感器融合

有哪些传感器

跟slam相关的传感器有 相机，激光雷达，GPS，轮速里程记，惯性测量单元（IMU）等

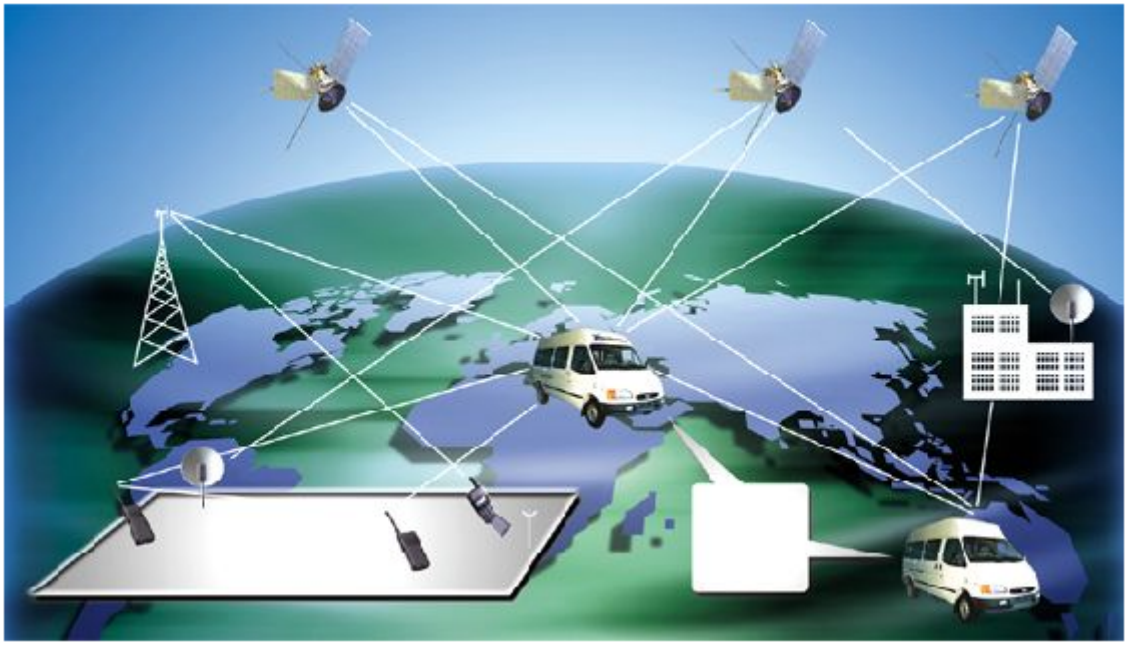
这些传感器基本上是自动驾驶车辆必备的传感器

他们分别的特点有哪些

相机：优点是便宜，轻量，包含信息丰富，可以参照人眼，人工智能的巅峰估计就是只利用相机可以在任何场景下实现实时建图、定位、图像分割、目标检测和分类、导航、避障等等高级功能，缺点就是现有的通用的视觉slam技术依赖图像的纹理来进行特征点的提取，没有纹理或者黑夜图像就很难被很好的利用起来，其次，图像中缺乏3d信息，通常建模slam问题需要同时优化位姿和地图点，这给优化问题的计算和精度带来了挑战，另一方面，单目图像缺乏尺度信息

激光雷达：优点包含了3d信息，探测距离远，同时不受光照条件和环境纹理等影响，缺点是几何特点会在部分场景下如长走廊，空旷的草原等失效，同时价格昂贵，当然16线的已经降价很多了，但是自动驾驶的乘用车基本不可能使用这么低线束的激光雷达，改装成本高，算法上限有限

GNSS：优点是提供全局定位信息，如果在卫星信号好同时附近有基站的情况下，配合RTK（实时动态差分技术）可以实现厘米级定位，不受光照以及环境几何特征的影响，缺点是基站作用范围有限，部署成本高，同时城市内容易发生多路径效应，定位结果不稳定，地下等场景完全没用



轮速里程记：优点是可以提供比较准确的车轮信息，对一定时间内的状态变化估计有非常好的作用，同时提供了尺度信息，缺点是基本只是一个2d运动模型，打滑算法就受影响，同时无法进行全局定位

IMU：优点是对一段时间的运动状态尤其是旋转有着比较好的估计，可以感受到重力，使得roll和pitch全局能观，不受任何环境因素的影响，缺点是积分轨迹时间一长容易发散，重力估计不正确会使得平移的估计偏差较大

我们可以发现基本上每一个传感器都用着自身无可比拟的优点以及无法规避的缺点，因此如果把所有的鸡蛋放到一个篮子里，无论是精度还是鲁棒性都会遇到非常大的问题，这也是自动驾驶车辆所无法接受的，因此，我们需要扬长避短，取各家所长，形成一个稳定的鲁棒的方案。

多传感器融合的方向

根据主传感器的类型分别视觉为主的方案以及激光为主的方案，由于视觉激光两个传感器的关联性以及互补性有限，因此这两种传感器紧耦合的方案并不多，比较多的方案要么以激光为主，要么以视觉为主同时融合其他传感器的方案。

视觉slam方向：常见的方式是一个视觉特征点前端（当然还有基于直接法的前端，如DSO），通过光流或者描述子建立不同帧特征点之间的关联，后端根据前端特征关联的结果和其他传感器数据进行融合，根据融合的方式分为基于优化的后端（ORBSLAM2、3, VINS-MONO，VINS-FUSION）以及基于滤波的后端（MSCKF），视觉通常会提供一个重投影误差作为约束或者更新量

激光slam方向：目前性能最好使用最广的激光slam方案是基于LOAM的系列方案，LOAM主要是为多线激光雷达设计的lidar定位和建图的方案，当然，由于现在其他一些lidar硬件的推出，一些LOAM的改进版本也是适当推出，如（Livox LOAM）。

基于LOAM方案通常前端是对当前帧激光雷达提取特征（通常是面特征和线特征），通常后端结合其他传感器信息给当前帧到地图中的匹配提供一个良好的初值（激光slam中最重要的事情就是给scan matching提供一个更准确的init guess）

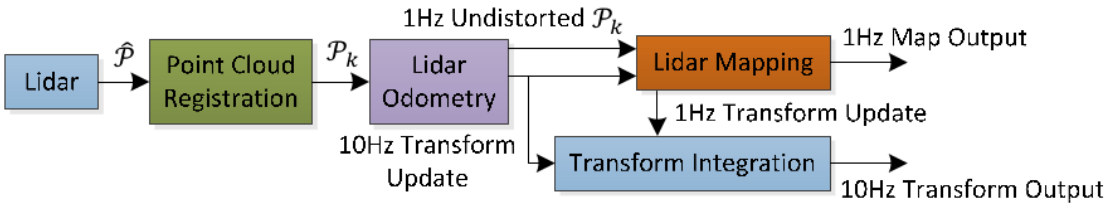
第三节 传感器融合方案介绍（A-LOAM，LeGO-LOAM，LIO-SAM）

本次课程主要讲解激光雷达为主传感器方案的多传感器融合

为什么选择激光雷达

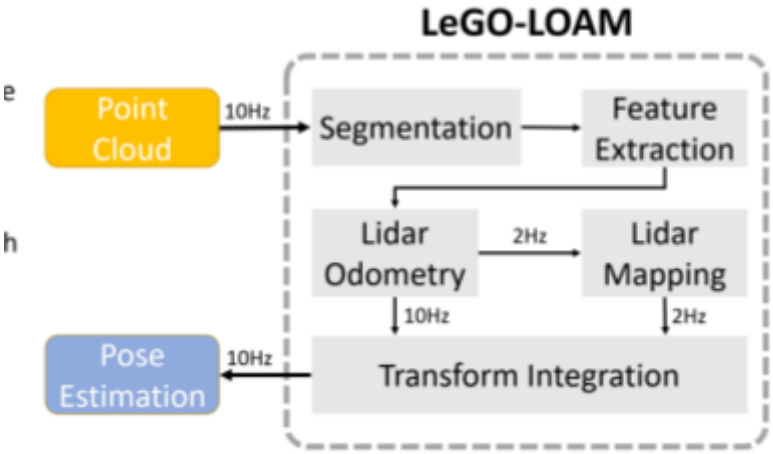
激光雷达相比图像有着对光照、纹理不敏感的优点，激光雷达地图相比通用的视觉特征点+描述子地图有着更好的稳定性，在安全性至上的自动驾驶领域，激光雷达方案比视觉方案鲁棒性更优，几乎所有L4级别的自动驾驶解决方案都会带有激光雷达（像特斯拉这样的纯视觉方案应用并不多），因此，从实用性上来讲，激光雷达有着视觉难以比拟的优点

LOAM (A-LOAM)



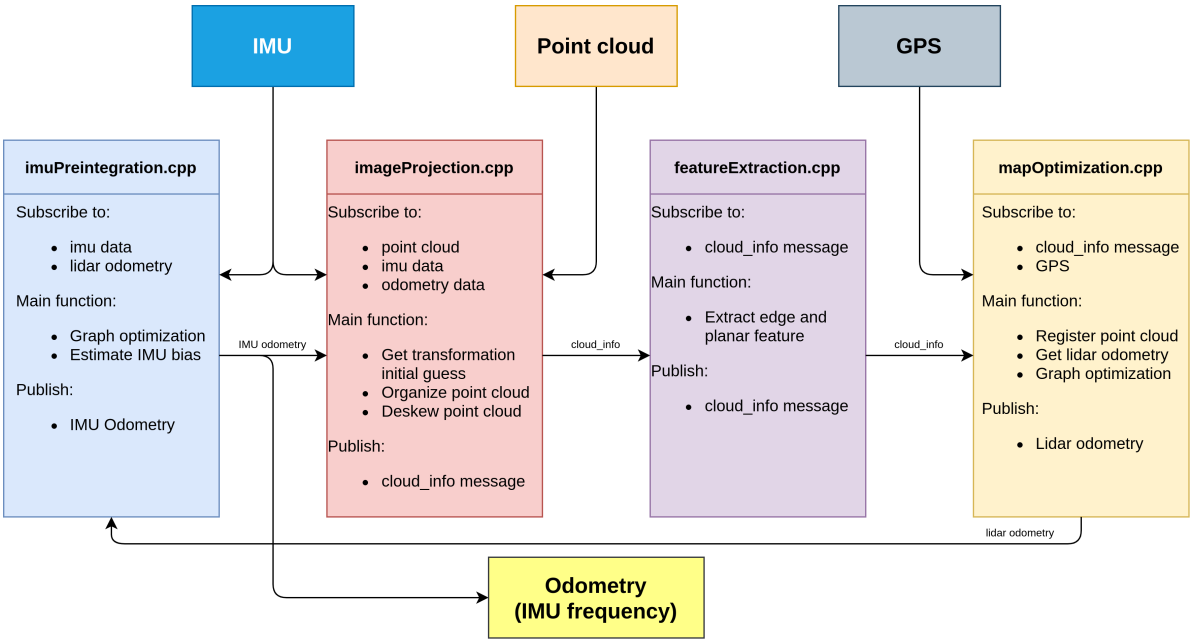
非常经典的激光里程记和建图方案，也是其他LOAM方案的鼻祖，LOAM只基于激光雷达（可选IMU），通过把SLAM拆分成一个高频低精的前端以及一个低频高精的后端来实现lidar里程记的实时性。

LeGO-LOAM



在原有LOAM基础上，在前端增加了地面点提取，并且根据嵌入式平台的计算特点，将前端做的更加轻量级，以便于在算力有限的平台上可以实时运行，后端将其使用slam中关键帧的定义进行重构，同时增加回环检测和回环位姿图优化，使得地图的全局一致性更优

LIO-SAM



在LOAM基础上采用紧耦合的imu使用方式，放弃了帧间里程记作为前端里程记，而使用紧耦合后imu的预测值作为前端里程记，后端沿用LeGO-LOAM，同时加入了对GPS信号的适配，使得其全局一致性更出色。

第二章 LOAM介绍及运行

第一节 LOAM论文带读

详见视频

第二节 A-LOAM简介

什么是A-LOAM

Advanced implementation of LOAM

A-LOAM is an Advanced implementation of LOAM (J. Zhang and S. Singh. LOAM: Lidar Odometry and Mapping in Real-time), which uses Eigen and Ceres Solver to simplify code structure. This code is modified from LOAM and LOAM_NOTED. This code is clean and simple without complicated mathematical derivation and redundant operations. It is a good learning material for SLAM beginners.

A-LOAM是港科大秦通博士使用Eigen以及Ceres-Solver对原始LOAM进行重构，在保证算法原理不变的前提下，对代码框架进行优化，使其更加容易被读懂，同时对初学者学习LOAM原理和算法也是极有帮助的。

github链接：GitHub - HKUST-Aerial-Robotics/A-LOAM: Advanced implementation of LOAM

原始loam代码有什么缺点

1、代码没有比较晦涩难懂，充斥着大量欧拉角的计算

我们这里就一个函数进行一个比较

在mapping模块中，我们需要把lidar odometry的增量更新到mapping中，其实现在loam中如下

```
void transformAssociateToMap()
{
    float x1 = cos(transformSum[1]) * (transformBefMapped[3] - transformSum[3])
             - sin(transformSum[1]) * (transformBefMapped[5] - transformSum[5]);
    float y1 = transformBefMapped[4] - transformSum[4];
    float z1 = sin(transformSum[1]) * (transformBefMapped[3] - transformSum[3])
             + cos(transformSum[1]) * (transformBefMapped[5] - transformSum[5]);

    float x2 = x1;
    float y2 = cos(transformSum[0]) * y1 + sin(transformSum[0]) * z1;
    float z2 = -sin(transformSum[0]) * y1 + cos(transformSum[0]) * z1;

    transformIncre[3] = cos(transformSum[2]) * x2 + sin(transformSum[2]) * y2;
    transformIncre[4] = -sin(transformSum[2]) * x2 + cos(transformSum[2]) * y2;
    transformIncre[5] = z2;

    float sbcx = sin(transformSum[0]);
    float cbcx = cos(transformSum[0]);
    float sbcy = sin(transformSum[1]);
    float cbcy = cos(transformSum[1]);
    float sbcz = sin(transformSum[2]);
    float cbcz = cos(transformSum[2]);

    float sblx = sin(transformBefMapped[0]);
    float cblx = cos(transformBefMapped[0]);
    float sbly = sin(transformBefMapped[1]);
    float cbly = cos(transformBefMapped[1]);
    float sblz = sin(transformBefMapped[2]);
    float cblz = cos(transformBefMapped[2]);

    float salx = sin(transformAftMapped[0]);
    float calx = cos(transformAftMapped[0]);
    float saly = sin(transformAftMapped[1]);
    float caly = cos(transformAftMapped[1]);
    float salz = sin(transformAftMapped[2]);
    float calz = cos(transformAftMapped[2]);
```

```
float srx = -sbcx*(salx*sblx + calx*cblx*salz*sblz + calx*calz*cblx*cblz)
          - cbcx*sbcy*(calx*calz*(cbly*sblz - cblz*sblx*sbly)
          - calx*salz*(cbly*cblz + sblx*sbly*sblz) + cblx*salx*sbly)
          - cbcx*cbcy*(calx*salz*(cblz*sbly - cbly*sblx*sblz)
          - calx*calz*(sbly*sblz + cbly*cblz*sblx) + cblx*cbly*salx);
transformTobeMapped[0] = -asin(srx);

float srycrx = sbcx*(cblx*cblz*(caly*salz - calz*salx*saly)
             - cblx*sblz*(caly*calz + salx*saly*salz) + calx*saly*sblx)
             - cbcx*cbcy*((caly*calz + salx*saly*salz)*(cblz*sbly - cbly*sblx*sblz)
             + (caly*salz - calz*salx*saly)*(sbly*sblz + cbly*cblz*sblx) - calx*cblx*cbly*saly)
             + cbcx*sbcy*((caly*calz + salx*saly*salz)*(cbly*cblz + sblx*sbly*sblz)
             + (caly*salz - calz*salx*saly)*(cbly*sblz - cblz*sblx*sbly) + calx*cblx*saly*sbly);
float crycrx = sbcx*(cblx*sblz*(calz*saly - caly*salx*salz)
             - cblx*cblz*(saly*salz + caly*calz*salx) + calx*caly*sblx)
             + cbcx*cbcy*((saly*salz + caly*calz*salx)*(sbly*sblz + cbly*cblz*sblx)
             + (calz*saly - caly*salx*salz)*(cblz*sbly - cbly*sblx*sblz) + calx*caly*cblx*cbly)
             - cbcx*sbcy*((saly*salz + caly*calz*salx)*(cbly*sblz - cblz*sblx*sbly)
             + (calz*saly - caly*salx*salz)*(cbly*cblz + sblx*sbly*sblz) - calx*caly*cblx*sbly);
transformTobeMapped[1] = atan2(srycrx / cos(transformTobeMapped[0]),
                               crycrx / cos(transformTobeMapped[0]));

float srzcrx = (cbcz*sbcy - cbcy*sbcx*sbcz)*(calx*salz*(cblz*sbly - cbly*sblx*sblz)
             - calx*calz*(sbly*sblz + cbly*cblz*sblx) + cblx*cbly*salx)
             - (cbcy*cbcz + sbcx*sbcy*sbcz)*(calx*calz*(cbly*sblz - cblz*sblx*sbly)
             - calx*salz*(cbly*cblz + sblx*sbly*sblz) + cblx*salx*sbly)
             + cbcx*sbcz*(salx*sblx + calx*cblx*salz*sblz + calx*calz*cblx*cblz);
float crzcrx = (cbcy*sbcz - cbcz*sbcx*sbcy)*(calx*calz*(cbly*sblz - cblz*sblx*sbly)
             - calx*salz*(cbly*cblz + sblx*sbly*sblz) + cblx*salx*sbly)
             - (sbcy*sbcz + cbcy*cbcz*sbcx)*(calx*salz*(cblz*sbly - cbly*sblx*sblz)
             - calx*calz*(sbly*sblz + cbly*cblz*sblx) + cblx*cbly*salx)
             + cbcx*cbcz*(salx*sblx + calx*cblx*salz*sblz + calx*calz*cblx*cblz);
transformTobeMapped[2] = atan2(srzcrx / cos(transformTobeMapped[0]),
                               crzcrx / cos(transformTobeMapped[0]));

x1 = cos(transformTobeMapped[2]) * transformIncre[3] - sin(transformTobeMapped[2]) * transformIncre[4];
y1 = sin(transformTobeMapped[2]) * transformIncre[3] + cos(transformTobeMapped[2]) * transformIncre[4];
z1 = transformIncre[5];
```

两张图都没有完全截完，不仅让人看起来头大无比，而且这种代码的可读性和可修改性都非常低，如果在公司中，略微严格的code reviewer都**不会允许这种代码合入项目主分支**，我们看一下A LOAM中是如何实现的

```
void transformAssociateToMap()
{
        q_w_curr = q_wmap_wodom * q_wodom_curr;
        t_w_curr = q_wmap_wodom * t_wodom_curr + t_wmap_wodom;
}
```

一共就两行代码，实际上，上面那一大段的代码所表述的意思确实就是这两行代码所描述的事情。由此可见，A-LOAM的重构使得代码变得十分简洁。

2、原始LOAM由于一些原因被作者闭源，后续不再维护

我们在github上是找不到作者loam源码的，通常是一些loam的注释(比如GitHub - cuitaixiang/LOAM_NOTED: loam code noted in Chinese（loam中文注解版）)或者fork作者代码之后的一些改进工作

（GitHub - laboshinl/loam_velodyne: Laser Odometry and Mapping (Loam) is a realtime method for state estimation and mapping using a 3D lidar.）

实际slamer也需要使用一些三方库

A-LOAM中使用的Eigen和Ceres-Solver是slamer常用的三方库，我们开发算法几乎很难规避这两种库，因此，A-LOAM的架构也更符合现代slam工程开发的需求，对A-LOAM的学习也可以迅速掌握这些常见三方库的用法。

第三节 A-LOAM编译及安装

详见视频

第四节 A-LOAM运行示例及可视化分析

详见视频

第五节 ROS及相关坐标系介绍

ROS简介

ROS（Robot Operation System）中文名称机器人操作系统，ROS并不是传统的操作系统（Windows，Linux，Android），相反，ROS依赖现有操作系统。

使用ROS之前需要安装诸如Ubuntu的Linux发行版，而且不同的Ubuntu发行版是绑定不同的ROS版本，如（Ubuntu16.04配ROS Kinetic，Ubuntu18.04配ROS Melodic），目的就是为了依赖操作系统中的进程管理、文件系统、用户界面等

因此，ROS实际上是一个中间件

ROS优点

分布式进程：它以可执行进程的最小单位（节点，Node）的形式进行编程，每个进程独立运行，并有机地收发数据。（通过rosmaster来管理所有node）
这样每个模块独立开发，定义好接口，剩下的交互交给ros
无论在线离线算法模块都不需要任何调整

配套的开发套件：非常方便的可视化rviz，rosbag的录包播包功能
你只需要把你想可视化的当作一个topic发出去，rviz就会非常方便的帮助你可视化
所以有的数据集直接提供ros包

公共存储库：很多功能包（gmapping，amcl等）都是开源的，大大降低了开发和二次开发成本。
你只需要根据功能包的消息定义给他喂数据，他就会把结果以消息格式发出来。

API类型：使用ROS开发程序时，ROS被设计为可以简单地通过调用API将其加载到其使用的代码中。你会发现ros编程跟调用其他第三方库一样方便。

ROS有一个庞大的生态和开源社区

ROS一些基本概念

功能包（package）：功能包是构成ROS的基本单元。ROS应用程序是以功能包为单位开发的

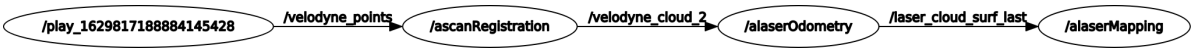
节点（node）：一个功能包至少有一个节点，一个节点可以理解成一个进程

消息（message）：节点之间通过消息来发送和接收数据，消息的类型可以自己定义，也可以使用ros定义好的消息类型

话题（topic）：最常见的消息通信方式，发布者负责发，订阅者负责收，通过主节点将订阅者节点直接连接到发布者节点

Launch文件：可以同时启动多个节点，（否则一个节点占用一个终端），加载参数到参数服务器（就是充当配置文件作用）

ROS各个节点状态示意图（以A-LOAM为例）



SLAM中一些坐标系

无论是描述一个点的坐标或者一个物体的位姿都是基于某个坐标系的，脱离坐标系的坐标描述是不现实的。

1、map坐标系

给定了一个地图，通常是点云地图或者视觉地图，他们的坐标系一般是固定的，通过点云匹配或者视觉特征匹配可以获得在地图坐标系下的位姿

2、odom坐标系

通常是以机器人上电时刻的位置作为原点，然后通过机器人对自身运动的估计来获取odom坐标系下的位姿，可以理解为里程记构建的坐标系，通常会遭遇累计漂移的问题

3、baselink坐标系

车体坐标系，通常以车上某个位置作为车体坐标系的原点，一般来讲，如果多个传感器需要都标定到车体系才能进行多传感器融合

4、lidar坐标系

我们获得的lidar点云都是在lidar坐标系下的，都是相对lidar中心的坐标，这也是从驱动拿到的lidar数据的所在的坐标系

第三章 A-LOAM前端源码分析

第一节 多线激光雷达性质

激光雷达测距原理

激光雷达发射脉冲打到物体后返回，根据激光测距原理，通过计算发射时间和接受时间的时间差就能计算出该物体离激光雷达中心的距离，由于其获得真实距离，所以lidar slam中不会存在视觉slam中的尺度问题。

多线激光雷达

多线激光雷达即有多个激光发射器同时工作，如常见的Velodyne16,就是共有16个激光发射器，一般这些发射器竖排排列，然后一起水平旋转。

激光雷达在一定的时间内旋转一圈，即一帧的点云数据。值得注意的是，在一帧时间内激光雷达的载体也会运动，因此，一帧数据内会包括不同时间的点云，这也就是激光雷达的运动畸变。

第二节 特征提取及均匀化

详见视频

第三节 异常点的剔除机制

详见视频

第四节 ceres-solver简介及自动求导方式介绍

slam中的优化问题一般形式

通常来讲，对一个slam问题建模成一个优化问题就是需要迭代去求解形如

$$J^TJ\delta x = -J^Te$$

这样一个问题，也就是说需要我们去求出此时的雅克比矩阵J以及残差e才可以求出优化变量的增量，进而迭代求解最终的优化变量

ceres-solver介绍

求解slam中的优化问题通常比较复杂，一般来说我们会借助一些现有的优化器来帮助我们实现，如ceres-slover，g2o，gtsam等，A-LOAM中选用了ceres-solver作为优化器，而ceres-solver是谷歌开源的一个优化工具，其具有文档齐全，接口简单等优点，广受开发者欢迎，谷歌也在其开源的激光slam算法cartographer中使用ceres-solver。

ceres-solver自动求导介绍

通常一个优化器会帮助我们解决优化问题中大部分内容，但是残差的计算以及残差对优化变量的雅克比矩阵通常是需要用户来定义的，而雅克比矩阵通常比较复杂，因此有的优化库如g2o，gtsam会预先定义好一些常见的优化问题所涉及到的残差及雅克比计算方式，但是并不能覆盖到所有场景，一旦有些问题不是优化器所事先建模好的问题，那就需要用户自行定义残差和雅克比的计算方式，这往往会非常麻烦，而ceres-solver通过引用自动求导功能，无论什么优化问题，用户只需要定义残差的计算方式，自动求导功能会帮助用户计算对应的雅克比矩阵来实现优化问题求解，对于用户来讲是一个非常方便而友好的功能。

自动求导接口使用

需要定义一个类，其中必须使用模板函数重载（）运算符，（）运算符函数第几个参数是参数块的起始指针，最后一个参数是残差块的指针，（）运算符负责计算残差，

定义完之后，其他交给ceres-solver就好啦。

参考http://www.ceres-solver.org/automatic_derivatives.html

第五节 激光雷达的运动畸变以及补偿方式

什么是运动畸变

我们知道激光雷达的一帧数据是过去**一段时间**而非某个时刻的数据，因此在这一帧时间内的激光雷达或者其载体通常会发生运动，因此，这一帧数据的原点都不一致，会导致一些问题，比如

这个代表一辆车往墙的方向运动，原则上这个墙应该是水平的，但是由于上述原因，最终该帧的数据如下

这个和实际场景就不相符合，因此，运动去畸就非常重要。

如何进行运动补偿

运动补偿的目的就是把所有的点云补偿到某一个时刻，这样就可以把本身在过去100ms内收集的点云统一到一个时间点上去

比如一种的做法是补偿到起始时刻

$$P_{start} = T_{\_start\_current} * P_{current}$$

因此运动补偿需要知道每个点该时刻对应的位姿$T_{_start_current}$,通常有几种做法

1、如果有高频里程记，可以比较方便的获取每个点相对起始扫描时刻的位姿

2、如果有imu，可以方便的求出每个点相对起始点的旋转

3、如果没有其他传感器，可以使用匀速模型假设，使用上一个帧间里程记的结果作为当前两帧之间的运动，同时假设当前帧也是匀速运动，也可以估计出每个点相对起始时刻的位姿

第六节 帧间lidar里程记

详见视频

第四章 A-LOAM后端源码分析

第一节 基于珊格点云地图构建

不同于前端的scan-to-scan的过程，LOAM的后端是scan-to-map的算法，具体来说就是把当前帧和地图进行匹配，得到更准的位姿同时也可以构建更好的地图。由于是scan-to-map的算法，因此计算量会明显高于scan-to-scan的前端，所以，后端通常处于一个低频的运行频率，但是由于scan-to-map的精度往往优于scan-to-scan，因此后端也有着比起前端来说更高的精度。

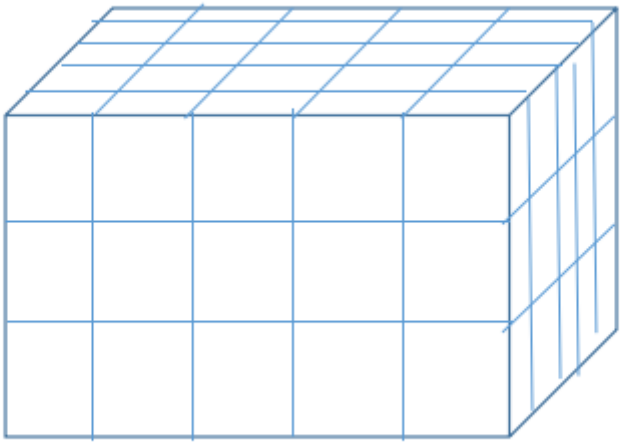
初始位姿更新

前端里程记会定期向后端发送位姿$T_{odom_current}$，但是在mapping模块中，我们需要的得到的位姿是$T_{map_current}$，因此，mapping模块就是需要估计出odom坐标系和map坐标系之间的相对位姿变换T_{map_odom}

如此 $T_{map\_current} = T_{map\_odom} \times T_{odom\_current}$

地图的构成

首先需要了解的一件事情就是地图的构成，地图通常是当前帧通过匹配得到在地图坐标系下的准确位姿之后拼接而成，如果我们保留所有的拼接的点云，此时随着时间的运行，内存很容易就吃不消了，因此考虑存储离当前帧比较近的部分地图，同时，为了便于地图更新和调整，在原始LOAM中，使用的是基于珊格的地图存储方式。具体来说，将整个地图分成21×21×11个珊格，每个珊格是一个边长50m的正方体，当地图逐渐累加时，珊格之外的部分就被舍弃，这样可以保证内存空间不会随着程序的运行而爆炸。

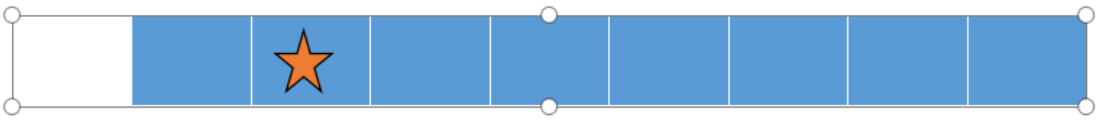


但是，我们注意到，如果当前位姿远离的珊格覆盖范围，则地图也就没有意义了，因此，珊格地图也需要随着当前位姿**动态调整**，从而保证我们可以从珊格地图中取出离当前位姿比较近的点云来进行scan-to-map算法，借以获得最优位姿估计。

珊格地图的调整

当当前位姿即将到达地图边界的时候，当前珊格地图就必须动态调整，如下图，我们以一维case为例

当前帧即将抵达地图的左边界时，我们需要把当前帧和地图整体右移一部分，保证当前帧处于一个安全的位置



这样相对移动之前，当前帧就处在一个“安全的位置”，然后左边会空出来一个珊格。

第二节 地图中线特征和面特征的提取

在前端里程记部分，我们通过当前帧的线特征和面特征分别和上一帧的线特征和面特征进行匹配，构建约束，然后进行优化求解。由于机械式激光雷达的性质，我们在寻找匹配的过程中需要注意线束相关的约束，以免构建的约束不符合实际。

在后端的当前帧和地图匹配的时候，我们就需要从地图里寻找线特征和面特征的约束对，此时，由于没有了线束信息，我们就需要采取额外的操作来判断其是否符合线特征和面特征的给定约束。

线特征的提取

通过kdtree在地图中找到5个最近的线特征，为了判断他们是否符合线特征的特性，我们需要对其进行特征值分解，通常来说，当上述5个点都在一条直线上时，他们只有一个主方向，也就是特征值是一个大特征值，以及两个小特征值，最大特征值对应的特征向量就对应着直线的方向向量。

面特征的提取

同样首先通过kdtree在地图中找到最近的面特征，原则上面特征也可以使用特征值分解的方式，选出最小特征值对应的特征向量及平面的法向量，不过代码里选用的是平面拟合的方式：

我们知道平面方程为$Ax+By+Cz+D=0$，考虑到等式的形式，可以进一步写成，$Ax+By+Cz+1=0$，也就是三个未知数，五个方程，写成矩阵的形式就是一个5×3大小的矩阵，求出结果之后，我们还需要对结果进行校验，来观察其是否符合平面约束，具体就是分别求出5个点到求出平面的距离，如果太远，则说明该平面拟合不成功。

第三节 构建优化问题求解位姿

详见视频

第四节 地图位姿更新

我们通过第三节的地图优化，求出了当前帧在地图坐标系下的最优位姿T_{map_curr}，如本章第一节所提到的，mapping模块需要实时估计出map坐标系和odom坐标系之间的位姿变换，因此，我们这儿更新T_{map_odom}如下

$$T_{map\_odom} = T_{map\_curr} * T_{odom\_curr}^{-1}$$

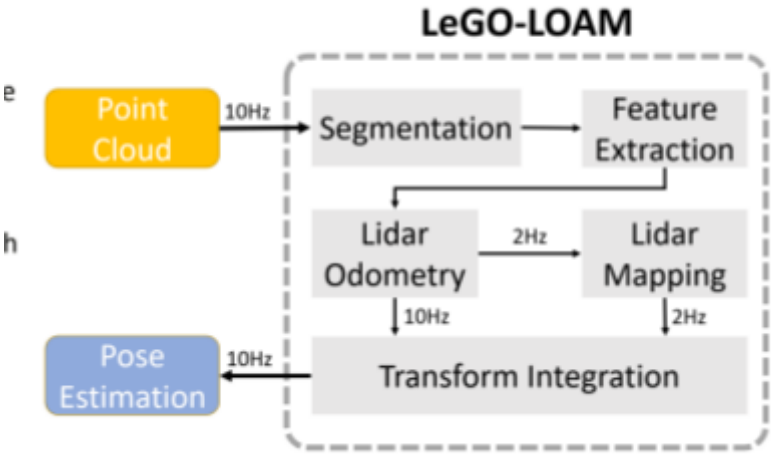
第五章 LeGO-LOAM介绍及运行

第一节 LeGO-LOAM框架简介

LeGO-LOAM是Tixiao Shan提出的一种基于LOAM的改进版本，其主要是为了实现小车在多变地形下的定位和建图，其针对前端和后端都做了一系列的改进，具体来说：

前端

1. 对地面点进行分类和提取，避免一些一场边缘点的提取
2. 应用了一个简单的点云聚类算法，剔除了一些可能的outlier
3. 两步迭代求解前端帧间里程记，不影响精度的情况下减轻计算负载，保障了嵌入式平台的实时性



后端

1. 使用slam中关键帧的概念对后端部分进行了重构
2. 引入回环检测和位姿图优化概念，使得地图的全局一致性更好

第二节 LeGO-LOAM论文解读

详见视频

第三节 代码编译安装及常见问题解决

详见视频

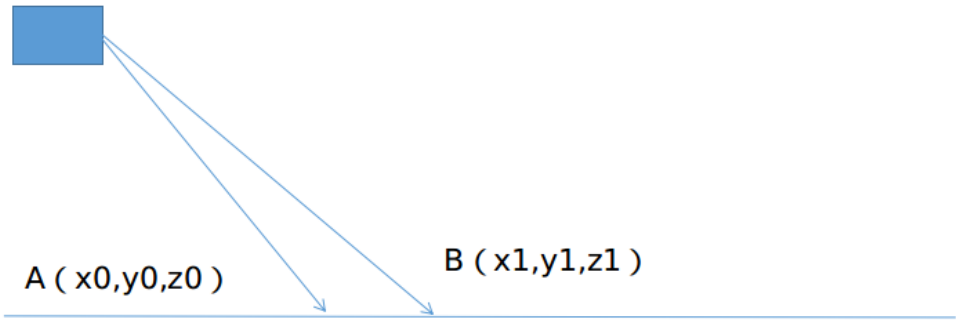
第四节 代码运行实例及可视化分析

详见视频

第六章 LeGO-LOAM相对LOAM的改进

第一节 地面分离方法

LeGO-LOAM中前端改进中很重要的一点就是充分利用了地面点，那首先自然是提取对地面点的提取



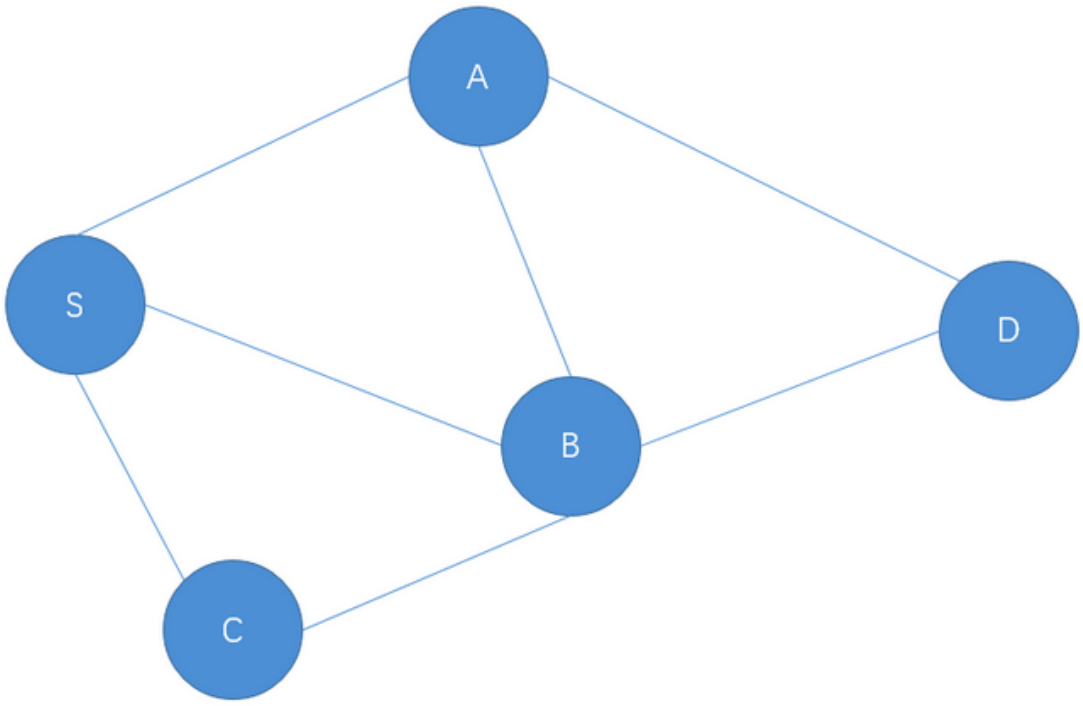
如上图，相邻的两个扫描线束的同一列打在地面上如AB点所示，他们的垂直高度差$h = |z_0 - z_1|$，水平距离差$d = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$，计算垂直高度差和水平高度差的角度，$\theta = atan2(h, d)$，理想情况下，$\theta$应该接近0,考虑到一方面激光雷达安装也无法做到绝对水平，另一方面，地面也不是绝对水平，因此，这个角度会略微大于0,考虑到作者实际在草坪之类的场景下运动，因此这个值被设置成10度。

实际上此种地面分离算法有一些简单，我们可以结合激光雷达安装高度等其他先验信息进行优化。

第二节 广度优先遍历算法介绍

广度优先遍历（BFS）和深度优先遍历（DFS）同属于两种经典的图遍历的算法。

具体到广度优先遍历算法来说，首先从某个节点出发，一层一层地遍历，下一层必须等到上一层节点全部遍历完成之后才会开始遍历



比如在上面这个无向图中，如果我们从A节点开始遍历，那么首先访问和A节点相邻的节点，这里就是S、B、D，然后在访问和S、B、D相邻的其他节点，这里就是C，因此，遍历的顺序是A->S->B->D->C;如果我们从S开始遍历，则顺序就是S->A->B->C->D；可以看到，不同的起始点对应的遍历顺序是不同的。

通常我们使用BFS遍历图结构的时候，会借助一个队列std::queue来帮助我们实现，其基本步骤如下

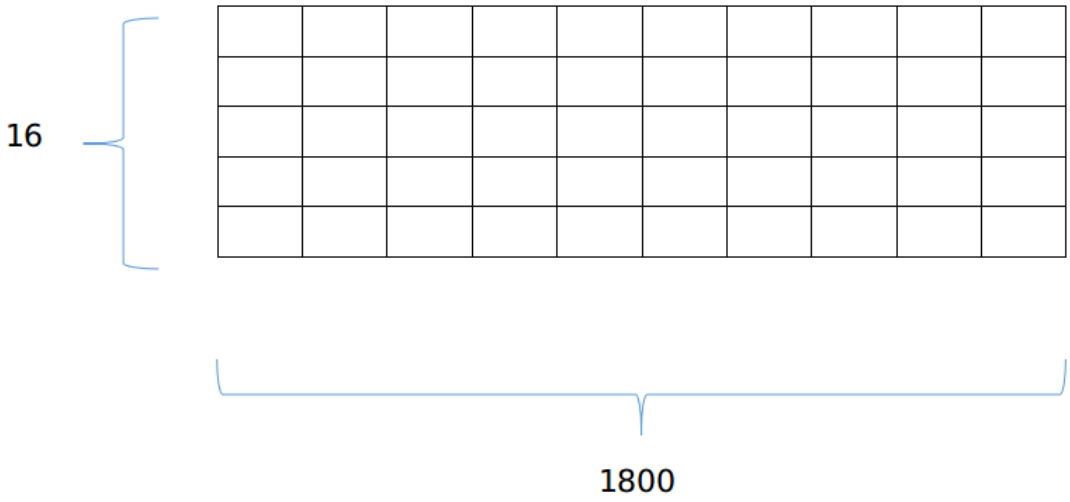
1. 将起始顶点S加入队列
2. 访问S，同时把S标记为已访问

3. 循环从队列中取出节点
    1. 如果队列为空，则访问结束
    2. 否则类似S，将该节点标记为已访问，同时将其子节点加入队列。

在LeGO-LOAM中，BFS算法用于实现一个简单的点云聚类功能。

第三节 基于BFS的点云聚类和外点剔除

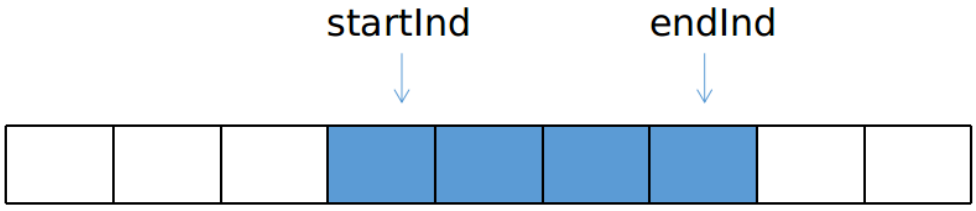
BFS算法适用于图数据结构，为了把单帧lidar点云运用上BFS算法，首先需要将其建模成一个图模型，一个很简单有效的办法就是将其投影到一个平面图上，以velodyne-16为例，我们将其投影到一个16×1800大小的图上（这里16是一共有16跟线束，1800是因为水平分辨率是0.2度，一个扫描周期有1800个点）如图



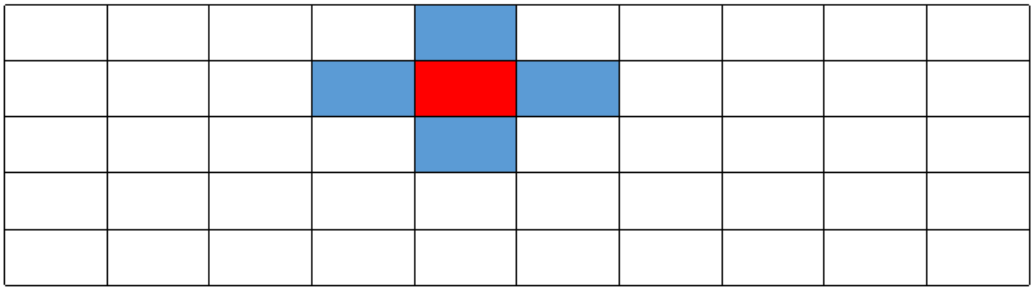
对于任何一个珊格点，其上下左右四个相邻点视为图结构中的邻接节点，这里要注意的是，左右边界的点和边界另一侧也构成邻接，因为水平方向是同一个扫描周期，具有物理意义上的连续性。我们可以以任意一个点开始执行BFS搜索，直到遍历完这部分近邻点，聚类点数过少的就认为是outlier，可以被剔除。

具体实现

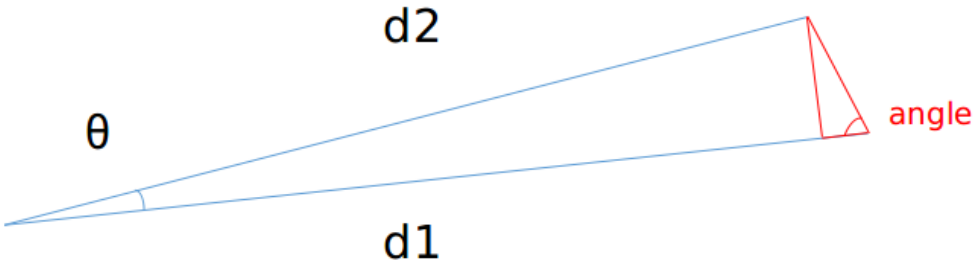
1. 遍历每个点，如果该点已经被处理过了就不再处理
2. 如果没有被处理就说明这是一个新的聚类，然后执行BFS的步骤
    1. 将队列里的首元素弹出，然后将该元素近邻塞入队列末尾（这里没有使用std::queue，使用的普通数组，所以就使用双指针来替代）



这里的近邻就是上下左右



2. 分别判断近邻和自身距离是否足够近



angle越大则认为两点越可能是同一个聚类物体上的点，则打上同样的label

3. 如此循环

第四节 两步优化的帧间里程记

和原始LOAM（或者A-LOAM）一样，通过前后两帧点云来估计两帧之间的运动，从而累加得到前端里程记的输出，和上述方法使用线面约束同时优化六自由度帧间位姿不同，LeGO-LOAM的前端分成两个步骤，每个步骤估计三自由度的变量

第一步 利用地面点优化

地面点更符合面特征的性质，因此，地面点的优化问题就使用点到面的约束来构建，同时我们注意到，地面点之间的约束对x，y和yaw这三个自由度是不能观的，换句话说，当这三个自由度的值发生变化时，点到面的残差不会发生显著变化，所以，地面点之间的优化只会对pitch，roll以及z进行约束和优化

第二步 利用角点优化

第一部优化完pitch、roll以及z之后，我们仍需对另外三个自由度的变量进行估计，此时，我们选用提取的角点进行优化，由于多线激光雷达提取的角点通常是垂直的边缘特征，因此，这些特征对x、y以及yaw有着比较好的能观性，通过角点的优化结合上地面点的结果可以得到六自由度的帧间优化结果。

第七章 LIO-SAM介绍及运行

详见视频

第八章 惯性测量单元简介及预积分

第一节 IMU器件介绍及选型建议

IMU的全称是惯性测量单元，包括一个三轴的加速度计以及一个三轴的陀螺仪，分别测量出物体的加速度和角速度信息，不受周围环境结构，光照等外界因素影响;同时，通常IMU的输出频率在100-1000hz之间，远高于相机或者激光雷达的输出频率，一方面可以提高整体系统的输出频率，另一方面，可以在视觉或者激光短期失效的时候提供一段时间的位姿推算。

在大多数的LIO或者VIO中，关于IMU输出的建模方式为

$$a = R_{bw}(a_t - g) + b_a + n_a$$

$$w = w_t + b_w + n_w$$

即输出的加速度计和陀螺仪的数据受零偏以及高斯白噪声的影响，因此，紧耦合的LIO或者VIO都会实时估计IMU的零偏，以实现IMU数据的最大利用率。

然而，实际的IMU数据并没有这里理想，除了零偏和高斯白噪声，还有可能受到刻度误差，尺度因子，轴向偏差等影响，然而，如果把这些因素都通过建模的方式考虑进来，就显得过于复杂，因此，通常的做法是在IMU选型的时候就最大化避免上述误差的影响。也就是说，我们选择IMU型号时除了关注价格（当然价格很多时候也是非常重要甚至是决定性的因素），还需要关心其出厂标定情况，是否做过温度补偿之类。

关于更多IMU相关的内容可以参考秦永元的《惯性导航》

第二节 IMU状态传递方程

IMU可以获得当前时刻的角速度和加速度值，通过该值可以对系统状态（位置，速度，姿态）进行推算

连续时间两个关键帧b_k, b_{k+1}之间的状态传递公式如下

$$\mathbf{p}_{b_{k+1}}^{w} = \mathbf{p}_{b_k}^{w} + \mathbf{v}_{b_k}^{w}\Delta t_k + \iint_{t\in[t_k,t_{k+1}]}\left(\mathbf{R}_t^w\left(\hat{\mathbf{a}}_t - \mathbf{b}_{a_t} - \mathbf{n}_a\right) - \mathbf{g}^w\right)dt^2$$

$$\mathbf{v}_{b_{k+1}}^{w} = \mathbf{v}_{b_k}^{w} + \int_{t\in[t_k,t_{k+1}]}\left(\mathbf{R}_t^w\left(\hat{\mathbf{a}}_t - \mathbf{b}_{a_t} - \mathbf{n}_a\right) - \mathbf{g}^w\right)dt$$

$$\mathbf{q}_{b_{k+1}}^{w} = \mathbf{q}_{b_k}^{w} \otimes \int_{t\in[t_k,t_{k+1}]}\frac{1}{2}\boldsymbol{\Omega}\left(\hat{\boldsymbol{\omega}}_t - \mathbf{b}_{w_t} - \mathbf{n}_w\right)\mathbf{q}_t^{b_k}dt$$

其中

$$\boldsymbol{\Omega}(\boldsymbol{\omega}) = \begin{bmatrix} -\lfloor\boldsymbol{\omega}\rfloor_\times & \boldsymbol{\omega} \\ -\boldsymbol{\omega}^T & 0 \end{bmatrix}, \lfloor\boldsymbol{\omega}\rfloor_\times = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$

考虑到我们接受的传感器数据都是离散形式的，因此，我们实际应用时是使用离散形式进行状态传播的，

每当收到一帧新的imu数据后，系统状态变化为：

$$p_{b_{i+1}}^{w} = p_{b_i}^{w} + v_{b_i}^{w}\delta t + \frac{1}{2}\bar{\hat{a}}_i\delta t^2$$
$$v_{b_{i+1}}^{w} = v_{b_i}^{w} + \bar{a}_i\delta t$$
$$q_{b_{i+1}}^{w} = q_{b_i}^{w} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\widehat{\bar{\omega}}_i\delta t \end{bmatrix}$$

第三节 IMU预积分

为什么需要预积分

从第二节可以发现，当k时刻的状态发生变化时，则通过imu积分得到的k+1时刻的状态也会发生相应的变化，而在基于滑窗的后端优化或者因子图的优化中，对一些状态量进行调整是必然发生的，此时，如果每次状态发生调整时都imu的积分过程重新执行一遍，则实时性必然无法得到保证，因此，预积分理论就是为解决这个问题而提出的，其核心思想就是对IMU积分的结果和上一时刻系统的状态无关，这样，当系统状态在优化过程中发生调整的时候，就不需要对下一时刻的系统状态重新积分。

怎样进行预积分

参考上一节连续时间IMU积分的公式，等号两边同时乘上$R_w^{b_k}$即可，即

$$
\begin{aligned}
R_w^{b_k} p_{b_{k+1}}^w &= R_w^{b_k}\left(p_{b_k}^w + v_{b_k}^w \Delta t_k - \frac{1}{2} g^w \Delta t_k^2\right) + \alpha_{b_{k+1}}^{b_k} \\
R_w^{b_k} v_{b_{k+1}}^w &= R_w^{b_k}\left(v_{b_k}^w - g^w \Delta t_k\right) + \beta_{b_{k+1}}^{b_k} \\
q_w^{b_k} \otimes q_{b_{k+1}}^w &= \gamma_{b_{k+1}}^{b_k}
\end{aligned}
$$

其中

$$
\begin{aligned}
\alpha_{b_{k+1}}^{b_k} &= \iint_{t \in [k, k+1]} \left[R_t^{b_k}\left(\hat{a}_t - b_{a_t}\right)\right] dt^2 \\
\beta_{b_{k+1}}^{b_k} &= \int_{t \in [k, k+1]} \left[R_t^{b_k}\left(\hat{a}_t - b_{a_t}\right)\right] dt \\
\gamma_{b_{k+1}}^{b_k} &= \int_{t \in [k, k+1]} \frac{1}{2} \Omega\left(\widehat{\omega}_t - b_{\omega_t}\right) \gamma_t^{b_k} dt
\end{aligned}
$$

上面三个变量即预积分量，我们可以发现这三个预积分量都和k时刻或k+1时刻状态无关，因此当k时刻状态发生变化时，我们不需要将IMU的数据重新积分。

关于零偏的建模

通常来说，IMU的零偏会随着时间的变化而偏移，因此为了系统的准确性，零偏也是系统的优化变量之一，此时我们注意到预积分量虽然和两帧的具体位姿和速度等状态量无关，但是和零偏相关，因此，当零偏作为优化变量被优化后，预积分量也会发生相应的变化，那么此时我们是否需要重新积分呢？如果重新积分，预积分的意义不就失去了吗？

为了避免零偏的变化导致预积分量重新积分，考虑到通常零偏的变化在短时间（100ms）非常小，因此，我们可以使用一阶泰勒展开来进行近似，具体为

$$
\begin{aligned}
\alpha_{b_{k+1}}^{b_k} &\approx \hat{\alpha}_{b_{k+1}}^{b_k} + J_{b_a}^{\alpha} \delta b_a + J_{b_\omega}^{\alpha} \delta b_\omega \\
\beta_{b_{k+1}}^{b_k} &\approx \hat{\beta}_{b_{k+1}}^{b_k} + J_{b_a}^{\beta} \delta b_a + J_{b_\omega}^{\beta} \delta b_\omega \\
\gamma_{b_{k+1}}^{b_k} &\approx \hat{\gamma}_{b_{k+1}}^{b_k} \otimes \begin{bmatrix} 1 \\ \frac{1}{2} J_{b_\omega}^{\gamma} \delta b_\omega \end{bmatrix}
\end{aligned}
$$

这里预积分量关于零偏的雅克比矩阵会在预积分计算的时候一并计算，因此，当零偏被优化调整之后，只需要根据事先计算好的雅克比矩阵对预积分量进行更新即可。

离散时间的预积分更新

同样，实际系统是离散的IMU数据，我们的目的是得到两个关键帧（视觉 or lidar）之间的预积分结果，而我们获得的IMU数据是离散的，因此，常见的做法就是每收到一帧新的IMU数据更新一次预积分量，同样，这是一个求和而非连续积分的过程

当收到新的IMU数据后，预积分量更新公式如下

$$\hat{\alpha}_{i+1}^{b_k} = \hat{\alpha}_i^{b_k} + \hat{\beta}_i^{b_k}\delta t + \frac{1}{2}R\left(\hat{\gamma}_i^{b_k}\right)\left(\hat{a}_i - b_{a_i}\right)\delta t^2$$
$$\hat{\beta}_{i+1}^{b_k} = \hat{\beta}_i^{b_k} + R\left(\hat{\gamma}_i^{b_k}\right)\left(\hat{a}_i - b_{a_i}\right)\delta t$$
$$\hat{\gamma}_{i+1}^{b_k} = \hat{\gamma}_i^{b_k} \otimes \hat{\gamma}_{i+1}^{i} = \hat{\gamma}_i^{b_k} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\left(\widehat{\omega}_i - b_{\omega_i}\right)\delta t \end{bmatrix}$$

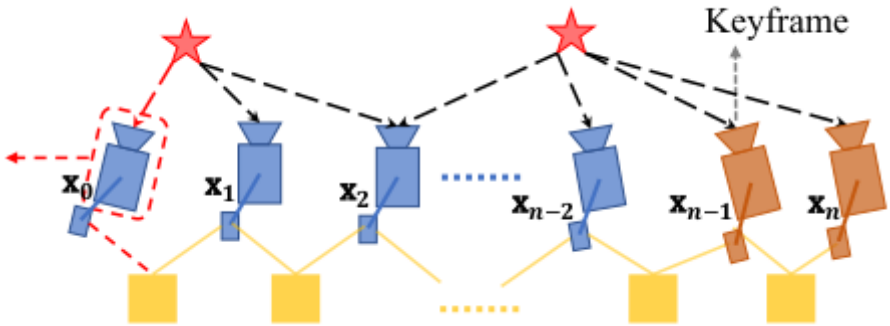
更新方程同样遵循基本的经典物理运动学公式

第四节 预积分在优化问题中的建模

由前面几节内容可知，预积分量约束相邻两帧的状态量（位置、速度、姿态），同时考虑到IMU的零偏的性质，即短时间内变换速率比较缓慢，因此可以认为两帧之间的零偏不变，也就是还可以约束两帧的零偏变化。

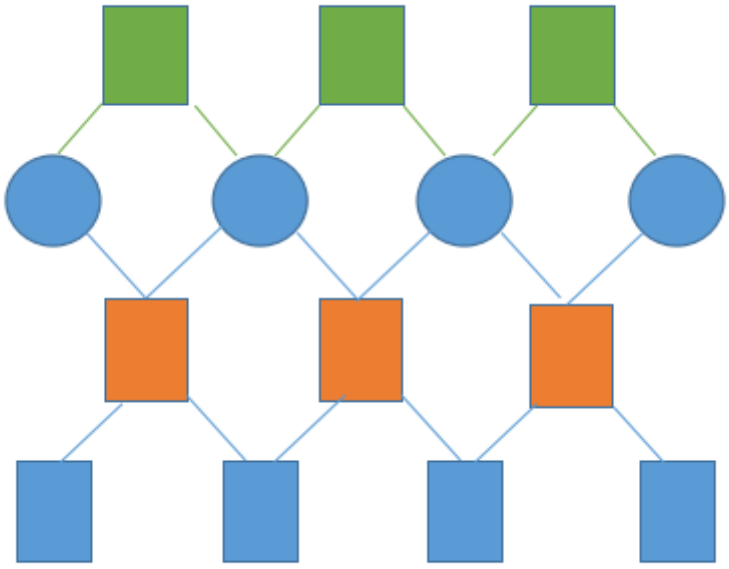
同时，在slam问题中，IMU预积分约束通常会和其他传感器的约束一起构建一个优化问题

视觉VIO中预积分和视觉的优化建模



这里黄色代表IMU预积分，可以看到其只对相邻帧发生约束，五角星代表视觉地图点，同一个地图点可以通过重投影误差对多个关键帧的位姿形成约束

LIO-SAM中预积分和lidar位姿的优化建模



如图，蓝色圆圈代表关键帧位姿，蓝色矩形代表关键帧速度和零偏，橙色矩形代表IMU预积分约束，可以看到，它可以约束相邻帧的位姿、速度和零偏，绿色矩形代表lidar里程记的帧间约束，其约束相邻两帧的位置和姿态。

这里涉及到论文中所提及的lidar odom因子和预积分因子，也就是预积分节点因子图的优化模型。

第五节 IMU标定方法简介

无论是VIO还是LIO，IMU和其他传感器的标定结果往往对最终的里程记和建图性能有着显著的影响，同样，在其他多传感器融合算法中，传感器之间的标定结果的精度对多传感器融合的效果也是有着非常大的影响。

在lidar-imu融合的算法中，lidar和IMU之间的外参标定也是非常重要，在一些数据集上往往有着经过良好标定的结果，然而，绝大多数情况下，传感器之间的外参需要我们自己去标定。

LIO-SAM的工程中，作者推荐了一个Lidar-IMU标定的功能包https://github.com/chennuo0125-HIT/lidar_imu_calib，来实现了一个简单的lidar和IMU之间的标定，通常来讲，lidar和IMU之间的旋转外参想比于平移外参对里程记的性能有着更显著的影响，因此，条件有限的情况下，可以只标定lidar和IMU之间的旋转外参，平移外参则可以通过手工丈量等方式获取。

该旋转标定算法本质上是一个手眼标定算法，即使先计算出两帧lidar之间的旋转（通过ICP、NDT等点云配准算法），然后计算出两帧IMU之间的旋转（IMU陀螺仪积分），然后通过手眼标定的方式求解出外参，其和VINS-Mono中相机IMU旋转外参初始化的过程一致，具体方式为

$$\mathbf{q}_{b_{k+1}}^{b_k} \otimes \mathbf{q}_l^b = \mathbf{q}_l^b \otimes \mathbf{q}_{l_{k+1}}^{l_k}$$

这里我们使用四元数来表示旋转，此时我们注意到四元数的乘法可以通过一下公式转成矩阵的乘法

$$\mathbf{p} \otimes \mathbf{q} = \begin{bmatrix} p_w q_w - p_x q_x - p_y q_y - p_z q_z \\ p_w q_x + p_x q_w + p_y q_z - p_z q_y \\ p_w q_y - p_x q_z + p_y q_w + p_z q_x \\ p_w q_z + p_x q_y - p_y q_x + p_z q_w \end{bmatrix}$$

$$\mathbf{q}_1 \otimes \mathbf{q}_2 = [\mathbf{q}_1]_L \mathbf{q}_2 \quad \text{and} \quad \mathbf{q}_1 \otimes \mathbf{q}_2 = [\mathbf{q}_2]_R \mathbf{q}_1$$

$$[\mathbf{q}]_L = \begin{bmatrix} q_w & -q_x & -q_y & -q_z \\ q_x & q_w & -q_z & q_y \\ q_y & q_z & q_w & -q_x \\ q_z & -q_y & q_x & q_w \end{bmatrix}, \quad [\mathbf{q}]_R = \begin{bmatrix} q_w & -q_x & -q_y & -q_z \\ q_x & q_w & q_z & -q_y \\ q_y & -q_z & q_w & q_x \\ q_z & q_y & -q_x & q_w \end{bmatrix}$$

化简一下，即

$$[\mathbf{q}]_L = q_w \mathbf{I} + \begin{bmatrix} 0 & -\mathbf{q}_v^\top \\ \mathbf{q}_v & [\mathbf{q}_v]_\times \end{bmatrix}, \quad [\mathbf{q}]_R = q_w \mathbf{I} + \begin{bmatrix} 0 & -\mathbf{q}_v^\top \\ \mathbf{q}_v & -[\mathbf{q}_v]_\times \end{bmatrix}$$

其中

$$[\mathbf{a}]_\times \triangleq \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}$$

由上述关系可以对原式进行转换，将四元数的乘法转成矩阵的乘法

$$\mathbf{Q}_{b_{k+1}}^{b_k L} \mathbf{q}_l^b = \mathbf{Q}_{l_{k+1}}^{l_k R} \mathbf{q}_l^b$$

合并之后为

$$\left(\mathbf{Q}_{b_{k+1}}^{b_k L} - \mathbf{Q}_{l_{k+1}}^{l_k R} \right) \mathbf{q}_l^b = \mathbf{0}$$

通常来说，我们会收集若干组IMU和lidar的相对旋转和平移，则可以联立如下

$$\begin{bmatrix} \mathbf{Q}_{b_1}^{b_0 L} - \mathbf{Q}_{l_1}^{l_0 R} \\ \mathbf{Q}_{b_2}^{b_1 L} - \mathbf{Q}_{l_2}^{l_1 R} \\ \mathbf{Q}_{b_n}^{b_{n-1} L} - \mathbf{Q}_{l_n}^{l_{n-1} R} \end{bmatrix}_{4n \times 4} \quad \mathbf{q}_l^b = \mathbf{A}_{4n \times 4} \mathbf{q}_l^b = \mathbf{0}$$

相当于已知一个4n×4大小的矩阵，求出一个4×1向量的最优解，通常n>4，因此，这是一个基本的超定方程求解问题，通常使用SVD方法求解

即将A矩阵进行SVD分解，得

$UDV^T x = 0$

这里U矩阵和V矩阵都是正交矩阵，D是奇异值由大到小的对角矩阵，因此等价求解

$DV^T x = 0$

然后，我们令$V^T x = y$，$||y|| = 1$

当且仅当$y = [0, 0, 0, 1]^T$时，Dy取得最小值，

此时对应的x即为V矩阵中最小奇异值对应的列向量，然后将其转换成四元数即为所求的旋转。

第九章 LIO-SAM前端源码分析

第一节 因子图优化及GTSAM库介绍

因子图

在slam的后端优化问题中，我们通常会通过一些传感器的观测，比如视觉特征点，IMU预积分量，Lidar面点和边缘点的约束去构建一个优化问题，求解状态量（如位姿、速度等），这个时候我们考虑一个问题，当给这个系统新增一个约束时，我们就会重新对所有的约束对状态的优化问题进行求解，当图优化模型增大时，显然进行一次优化的时间也会增加很多，一方面实时性遭遇了挑战，另一方面，很久之前的状态似乎也没有继续更新的必要。为了解决这个问题，一种方式是使用滑动窗口来控制优化问题的规模，通常来讲滑动窗口需要好处理边缘化的问题，另一方面，我们可以使用因子图的模型来解决这个问题。

Kaess等科研人员提出iSAM，即增量平滑和建图，使其可以自动增量处理大规模优化问题，具体来说，其内部使用一种基于概率的贝叶斯树，使得每次给因子图增加一个约束时，其会根据贝叶斯树的连接关系，调整和当前结点“关系比较密切”的结点，如此，既保障了优化问题的求解精度，也使得耗时不会随着优化问题的增大而增大。

关于因子图优化理论可以参考iSAM，iSAM2相关论文等文献。

因子图中一些概念

变量结点：类似g2O中的顶点或者ceres中的参数块，代表需要被优化的变量

因子结点：类似g2O中的边或者ceres中的cost function，代表约束，如预积分约束、位姿先验约束、帧间位姿约束等

GTSAM库的介绍

GTSAM全称是 Georgia Tech Smoothing and Mapping library，是佐治亚理工学院的科研人员基于因子图和贝叶斯网络推出的一个C++库文件，如果你想在你的工程里使用因子图优化的相关算法，那么最常用的方式就是借助GTSAM这个库来实现，因为其内部已经封装了关于因子图优化以及iSAM相关的算法实现，因此，我们只需要像调用其他第三方库的方式（如openCV，PCL等）调用GTSAM库即可

关于GTSAM库的详细介绍可以参考其官方文档https://gtsam.org/

第二节 GTSAM关于IMU预积分相关接口介绍

通过第八章相关介绍可知，我们对两个关键帧之间的若干帧IMU进行预积分，以形成预积分约束，对两帧之间的位置、速度、姿态以及零偏进行约束。GTSAM从4.0版本开始就在内部增加了IMU预积分相关的接口，因此，为了实现把预积分因子加入到图优化框架中，我们有必要熟悉一下GTSAM中跟IMU预积分相关的接口定义。

```
gtsam::PreintegrationParams
```

预积分相关参数，我们对IMU数据进行预积分之前通常需要事先知道IMU的噪声，重力方向等参数

```
gtsam::PreintegratedImuMeasurements
```

跟预积分相关的计算就在这个类中实现

这个类有一些重要的接口

```
(1) resetIntegrationAndSetBias
```

将预积分量复位，也就是说清空刚刚的预积分量，重新开始一个新的预积分量，因为预积分的计算依赖一个初始的IMU零偏，因此，在复位之后需要输入零偏值，所以这里复位和重设零偏在一个接口里。

```
(2) integrateMeasurement
```

输入IMU的测量值，其内部会自动实现预积分量的更新以及协方差矩阵的更新

```
(3) deltaTij
```

预积分量跨越的时间长度

```
(4) predict
```

预积分量可以计算出两帧之间的相对位置、速度、姿态的变化量，那结合上一帧的状态量就可以计算出下一关键帧根据预积分结果的推算值

第三节 预积分前端代码讲解

该模块涉及到的变量结点

```
gtsam::Pose3 // 表示六自由度位姿
gtsam::Vector3 // 表示三自由度速度
gtsam::imuBias::ConstantBias // 表示IMU零偏
```

以上也是预积分模型中涉及到的三种状态变量

涉及到的因子结点

```
gtsam::PriorFactor<T>
```

先验因子，表示对某个状态量T的一个先验估计，约束某个状态变量的状态不会离该先验值过远

```
gtsam::ImuFactor
```

imu因子，通过IMU预积分量构造出IMU因子，即IMU约束

```
gtsam::BetweenFactor
```

状态量间的约束，约束相邻两状态量之间的差值不会距离该约束过远

第十章 LIO-SAM后端源码分析

详见视频

第十一章 多传感器融合工程实践经验与技巧

第一节 A-LOAM LeGO-LOAM LIO-SAM 不同算法的对比

A-LOAM(LOAM)也即原始LOAM，后续无论是LeGO-LOAM还是LIO-SAM都是基于LOAM做的改进版本，在LOAM中，作者根据多线激光雷达的性质，提出了激光雷达的线特征和面特征的提取，并以此为基础实现了一个高频率低精度的前端以及低频率高精度的后端，这一点和PTAM以来的视觉slam将tracking和mapping分为两个线程有异曲同工之妙，借此完成了一个高精度的里程记和建图功能。

LeGO-LOAM在LOAM的基础上进行了一些改进，如

1、利用车载激光大多水平安装的特征，提取出地面点

2、使用聚类算法，使得前端特征更为干净

3、前端使用两步优化方法，减少运算负载，使其在嵌入式平台上也能运行

4、后端引入关键帧概念，同时加入了回环检测

LIO-SAM在上述基础上的改进

1、由于其支持手持设备，因此没有对地面点进行特殊处理

2、紧耦合的lidar+IMU融合模块，使得其充分利用IMU的数据，对快速旋转等场景有着更好的鲁棒性

3、融合GPS，使得全局地图可以在没有回环的情况下有着更好的全局一致性

4、易于扩展的框架，方便我们将其他传感器融合进来

总体来说，这三种框架随着时间顺序都是在前有基础进行的改造，因此，都是在吸收现在框架基础上进行的改进

第二节 算法中工程化技巧总结

在上述一些框架中，我们可以看到一些工程话的技巧，比如

1、LeGO-LOAM前端，对地面点的提取，利用地面点的一些性质对roll，pitch以及z进行一些约束和优化

2、通过BFS算法对前端特征进行过滤，使得更干净的特征留存了下来

3、后端滑动窗口的引入，使得构建局部地图更加方便，也更加方便实现纯里程记功能

4、对GPS的融合的谨慎的使用方式，使得既引入了全局观测，又不会对当前里程记的平滑性产生负面影响

第三节 多传感器融合算法改进落地建议

多传感器融合的目的是取长补短，比如在以激光雷达为主的融合方案中，我们需要明确激光雷达有什么缺陷，以此确定使用什么传感器进行融合

1、激光雷达需要运动补偿，我们需要短期内可靠的运动观测源，IMU以及轮速就可以被充分利用

2、激光雷达匹配本质上是一个优化问题，需要提供一个很好的初始值，和1一样，也是需要可靠的短期运动观测源，紧耦合的IMU融合或者轮速也是非常好的处理方式

3、激光雷达频率不高，为了提高频率，我们需要高频的其他传感器以获得高频输出，此时IMU和轮速又可以成为备选

4、里程记会有累计漂移的问题，全局观测是解决里程记该问题的非常好的方式，GPS作为常见的全局定位传感器，提供了修正累计漂移的功能

等等

第四节 多传感器融合未来发展趋势

人类需要多个传感器（眼睛鼻子耳朵等）来处理信息，SLAM也是一样，对传感器的正确使用会使得整个系统的鲁棒性越强，因此，在未来一段时间内，我们会尝试将更多新的传感器应用到SLAM中来，比如最近几年推出的LIVOX激光雷达等，和现有传感器一起取长补短，来提升整体性能的鲁棒性。

同时基于已有的传感器组合，我们也会探索更多紧耦合的融合方式，使得传感器之间的融合更有效率，做出精度更高的SLAM系统。

第十二章 项目实战大作业

第一节 企业对多传感器人才具体需求

我们认为，企业对多传感器融合或者slam要求有以下几个

1、对常见传感器特点，优缺点要足够了解，掌握在不同需求不同场景下的多传感器融合方案

2、扎实的slam基本功，对常用的视觉/激光slam理论及常见的开源框架熟练掌握

3、扎实的编程能力，对常见的数据结构和算法有足够的了解

4、相关度较高的项目/实习/工作经历

第二节 项目实战大作业

使用开源数据集（如liosam作者提供的数据包），实现

1、lidar和IMU旋转外参标定

2、使用LIO-SAM建好的地图，使用原数据包实现一个基于已知地图的定位

思路推荐

（1）可以参考LOAM建图过程，保存边缘特征和面特征地图，然后使用类似的配准方式实现定位

（2）保存完成的点云地图，使用ICP或NDT实现当前帧到地图的配准

（3）初始化是一个棘手的问题，为简化问题可以直接初始化到地图的原点