该内容是公众号 计算机视觉life 旗下课程**《(第2期) VINS:Mono+Fusion | 白板手推公式+源码逐行精讲》** 的课件编纂成的课程手册。

更新时间：2022年4月7日

扫描学习SLAM、三维重建

0 编译运行代码

0.0 安装ROS

大家都会了吧。
安装教程参考 http://wiki.ros.org/cn/ROS/Installation
安装不成功?
ubuntu 20.04尝试:

```
sudo apt-get install curl && curl http://fishros.com/tools/install/ros-noetic | bash
```

ubuntu 18.0尝试

```
sudo apt-get install curl && curl http://fishros.com/tools/install/ros-melodic | bash
```

ubuntu 16.04尝试

```
sudo apt-get install curl && curl http://fishros.com/tools/install/ros-kinetic | bash
```

还是不成功？来群里询问吧。

0.1 安装VINS

```
git clone https://github.com/HKUST-Aerial-Robotics/VINS-Mono.git
```

把代码复制粘贴到你的catkin_ws/src下，或者直接在这个目录下用git clone.

随后还需要安装一些依赖项如ros-cv-bridge。发型版号就是melodic或indigo，这和你安装的ROS版本有关系。

```
sudo apt-get install ros-发型版号-cv-bridge
```

具体教程可以参考VINS官网

如果不能编译成功和cv mat相关错误，则修改CMakeList.txt中对OpenCV版本的修改

```
find . -name "CMakeList.txt" | xargs sed -i "s/OpenCV REQUIRED/OpenCV 3.4.5 REQUIRED"
```

当然还可能存在其他错误。记得source当前工作空间的devel.setup.bash。这一步是为了让roslaunch 能正确找到对应的包。

0.1 SLAM 中的滤波VS优化

0.1.1 卡尔曼滤波

google翻译Filter: a porous device for removing impurities or solid particles from a liquid or gas passed through it.
换一个翻译，勉强叫"提纯"也说的过去。

信号与系统里的滤波：过滤掉杂质（不需要的信号），对结果提纯。
SLAM中的滤波：喂进去多个定位信息，或者传感器信息，只出来一个定位结果。

卡尔曼滤波分为预测、更新的部分。
卡尔曼滤波一共有5个重要的公式，分别如下

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1}$$
$$z_k = \boldsymbol{H}x_k + \boldsymbol{v}_k$$

$$\hat{x}_{\bar{k}} = A\hat{x}_{k-1} + Bu_{k-1}$$
$$P_{\bar{k}} = AP_{k-1}A^T + Q$$
$$K_k = \frac{P_k^T H^T}{HP_k^T H^T + R}$$
$$\hat{x}_k = \hat{x}_{\bar{k}} + K_k\left(z_k - H\hat{x}_{\bar{k}}\right)$$
$$P_k = (I - K_k H)P_{\bar{k}}$$

卡尔曼滤波器首先假设转移是线性运算的，同样所有的噪声都是符合高斯分布的。
卡尔曼滤波只维护当前状态，去预测下一个状态。不会用到“上上个状态”。

很多系统不都是线性，就需要对卡尔曼滤波进行改进——“扩展卡尔曼滤波”。

0.1.2 扩展卡尔曼滤波

这里可以在**估计点处**利用一阶导的方式进行线性近似。

0.2 SLAM中优化概述

利用一些观测的结果，去估计最终结果。
估计的结果，“尽量”满足所有的观测。
用核函数、ransac去去除一些离群点(outlier)。

0.2 ROS系统简介

ROS不是传统的操作系统，是利用现有的系统。
使用ROS前小安装ubuntu的linux操作系统，再安装ROS，这样可以调用linux系统的各种资源。
ROS实际上是一种中间件。

ROS有很多优点，可以分布式进程，每个进程独立运行，有机的的收发数据。这样每个模块可以独立开发，定义好接口，剩下的交给ROS。
无论在线离线模块都不需要任何调整。有大量的配套开发套件，比如rviz，rosbag等。很多开源数据集提供ROS包，方便大家开发。
ROS的使用者很多，很多人的代码都提供了ROS接口的支持。

0.2.1 ROS基本概念

0.2.1.1 ROS的介绍

- 功能包
  构成ROS的基本单元。ROS应用程序以功能包为单位开发的。
- 节点
  一个功能包至少一个节点，一个节点可以理解为一个进程。
- 消息
  节点之间通过消息来发送和接受数据，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
  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（否则启动多个节点很麻烦），加载参数到参数服务器（充当配置文件）。

0.2.1.2 ROS缺点

数据copy次数太多了。

- 节点到用户动态内存copy
- 发送方到内核态copy
- tcp链接后从内核态到接收节点copy
- 接受节点反序列化转化成结构化消息copy

改进为共享内存通讯

- 发送节点序列化成流式数据copy
- 接受节点直接把共享内存数据反序列化copy

0.3 坐标系的定义和符号说明

坐标系有右手和左手系。大多数SLAM是基于右手系。
相机坐标系：y向下，z向前，x向右。
IMU坐标系：x向前，y向左，z向上。
每个点都在某个坐标系下，slam很重要的坐标系叫做世界系。一般把SLAM的第一帧认为是世界原点。
变换矩阵一般记做

$$T = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}$$

符号一般用$\mathbf{T}_{wc}$,

$$\mathbf{T}_{wc} = \begin{pmatrix} \mathbf{R}_{wc} & \mathbf{t}_{wc} \\ 0 & 1 \end{pmatrix}$$

通常$\mathbf{t}$向量会记一下在哪个系下的向量。比如 $^w\mathbf{t}$表示在w系下。

1 VINS前端

对于视觉SLAM系统，通常分为前端和后端两个部分。

前端在VINS的框架中对应的节点名称是featrue_tracker。主要功能是解析图像，从图像中获取有效信息（如特征点），把有效信息汇总发给后端进行处理。
本章主要涉及的内容有去畸变、归一化、特征点提取、光流追踪、相机矫正等知识。

1.1 参数读取与设置

对于VINS系统，里面有大量的参数，和前端有关系的参数是通过readParameters(n);函数进行读取的。
里面是通过调用opencv的接口来读取yaml文件（提前准备好的参数文件）。
配置文件包括相机畸变参数，外参，以及各种算法的参数（比如featrue tracer），或者各种阈值（ransac求解的inlier的阈值等），以及相机模型（针孔，或鱼眼等）。

1.2 img_callback

主要包涵有3个部分，
1、通过时间戳来判断光流追踪是否成功。
2、控制给后端发送信息的频率，控制不要超过10Hz，给后端节约运算资源。
注意，这里被舍弃的帧还是需要做光流追踪的工作。因为光流需要假设2帧之间的变化特别小。
3、图像均衡化
如果失败的话，就要模块重启。

这个函数里工程上有一些技巧：
1、需要要check自己的一些特征相关工作是否是正确运行。
2、后端不一定要处理所有的数据，避免后端处理不过来。这里需要用一定的策略来实现调度。

1.3 readImage函数

大致的思路是做图像预处理，然后光流追踪，提取新的特征点，最后特征点去畸变，计算速度。

首先是判断是否图像均衡化，如果图像太暗或者太亮，提取特征点会比较难。那么就需要提升图像对比度，来提取角点。这里直接用的是opencv的函数createCLAHE来实现均衡化。

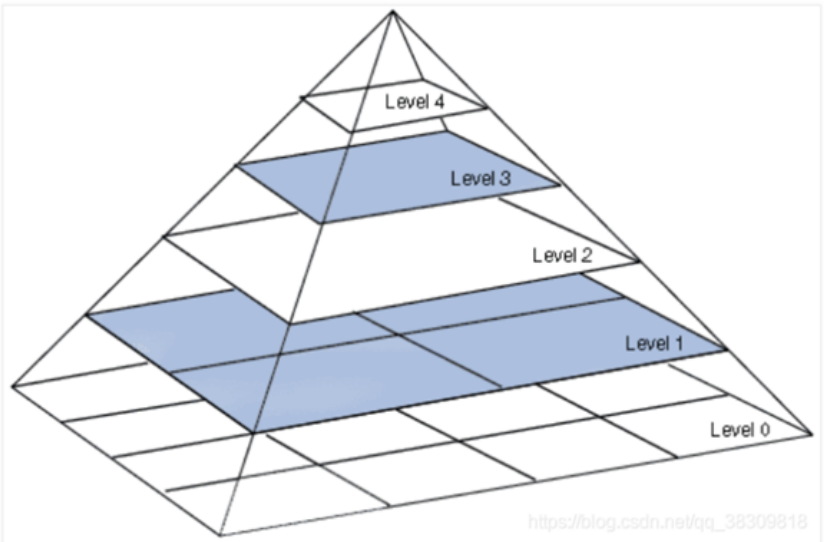
特别要注意，在这个函数里，这里prev是上一次的图像，forw是当前图像的意思。

用上一帧的特征点消息，就可以和当前帧进行光流追踪了。随后调用opencv的函数calcOpticalFlowPyrLK直接光流追踪。

1.4光流追踪 最小光度误差

在光流的领域中，有一个一个基本假设，“相邻两帧灰度不变”。

在一个图里找一个特征点，这个特征点附近的一块区域的灰度，和另一个图里一个区域的灰度要尽量相同。



通过图像金字塔，可以提升光流追踪的稳定性。通过图像金字塔的缩放，可以拉进2个图里像素的距离，从而提升光流追踪的稳定性。

我们把上一层金字塔的结果，作为一下层金字塔追踪的初值，一直追踪到Level0。图上是把一次的光流追踪，变为了多次光流追踪，准确度提升了，但是花费更多的时间。并不是金字塔越多越好，但是花费时间多，所以要权衡时间开销和精度的需求。

光流追踪函数的说明如下

calcOpticalFlowPyrLK()

```
void cv::calcOpticalFlowPyrLK ( InputArray         prevImg,
                                InputArray         nextImg,
                                InputArray         prevPts,
                                InputOutputArray   nextPts,
                                OutputArray        status,
                                OutputArray        err,
                                Size               winSize = Size(21, 21),
                                int                maxLevel = 3,
                                TermCriteria       criteria = TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, 0.01),
                                int                flags = 0,
                                double             minEigThreshold = 1e-4
                              )
```

Python:

```
cv.calcOpticalFlowPyrLK( prevImg, nextImg, prevPts, nextPts[, status[, err[, winSize[, maxLevel[, criteria[, flags[, minEigThreshold]]]]]]] ) -> nextPts, status, err
```

第一个参数：第一帧图像
第二个参数：第二帧图像
第三个参数：第一帧已经提取的特征
第四个参数：在第二帧图像的坐标
第五个参数：状态位，1表示特征点被成功追踪。否则位特征点追踪失败。

OpenCV官方的光流教程 点击这里

追踪光流的时候，还需要判断光流是否检测成功。简单的方法就是判断这个结果是否在图像范围内。

1.5 特征点的筛选

程序中有一个reduceVector函数，他作用是取出有用的数据。
随后要便利track_cnt数组，里面保存的是被追踪的次数，给追踪次数加1。

此时我们已经实现了光流追踪，和简单去除outlier的操作。如果当前帧要被publish出去，那么需要再做一次oulier的剔除。
具体过程如下：
先进行一次对极约束的剔除，首先对上一次的点进行去畸变（把像素坐标系放到相机坐标系下），让相机坐标系下的z等于1。去畸变的做法在后面会介绍。

用归一化相机坐标系，是为了避免不同相机参数不同的问题。比如480P和960P相机之间关于像素的参数，需要设置不同的值，归一化后就可以解决这个问题。
这里通过求解本质矩阵的方式，来去除outlier。因为求解过程比较费时间，所以只在向后端发送数据的时候，才做这个过程。

接下来剔除ouler的方法，是特征点均匀化。
对所有的特征点排序，按照追踪的次数排序。显然，被追踪的次数多的点，通常比较好。因为被追踪的次数多，说明这个特征点比较稳定，就更应该保留下来（也更小概率是奇葩点）。

排序完后，就进行均匀化的操作，不同于orb slam四叉树的均匀化的操作。具体的过程，是在被选中的重要点为中心，周围画一个圈，这个圈内不允许有别的特征点的存在，避免特征点过于集中。

现在我们已经剔除掉很多特征点了，我们需要再提取一些特征点来保证特征点的总量够用。特征点的提取用的函数是opencv的函数：goodfeaturetotrack。他是在灰度图行进行提取特征用的函数。这个函数里本身也带有特征点均匀化的功能。

最后调用addPoints函数，把新提取的特征点，加入进容器了。当然，新的特征点的追踪次数也设置为1，因为都是刚刚追踪到的。

回顾一下这一小节的内容，这部分首先是对极约束筛除一些特征点，设置mask让特征点均匀，最后再提取一些新的特征点。

1.6 计算特征点速度

特征点之间的匹配关系是已知的，用能匹配的上的特征点，根据变化的时间求解特征点变化的像素速度（在x，y方向上的速度）。

1.7 给后端发数据

给后端发数据，首先记录发送的数据数量pub_count数量加1.
把去畸变后的像素坐标、速度都保存起来，一起发送给后端。当然，只给后端发可以三角化的的点（被追踪数量大于1）。

1.8 光流VS描述子匹配

提取描述子，再用光流进行像素匹配。要提取的特征点较少。
但是描述子匹配略不同，是每个图像都要进行特征点匹配，随后对特征点进行匹配。描述子的匹配的各种算法比如FLANN并不快。主要每一张图都要进行特征子提取，和匹配，就比较慢。

总的来说，**LK光流还是更鲁棒。**
但是LK光流的缺点是如果一个点在某个时候被挡住了，随后再次出现的时候，会被认为是全新的点。

1.9 去畸变

输入一个像素坐标系的像素坐标，通过LiftProjective得到归一化坐标，这里会详细讲解。
直接调用opencv去畸变的方法也挺好，但是作者没有这么做。 TA的经验：

简单的说，不考虑畸变的世界3D点投到2D相机平面，就是相机的K矩阵乘以世界点坐标，随后把xy坐标除以z，就得到了相机坐标了。

畸变会让图像的像素变化，一般去畸变前后的像素变化大小，和图像中心的距离有关系。距离图像中心越远，畸变越明显。

根据畸变模型的公式，

$$x_{ditorted} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 xy + p_2(r^2 + 2x^2)$$
$$y_{dstorted} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y^2) + 2p_2 xy$$

这是一个输入畸变前的点，得到畸变后的点坐标的公式。
畸变后的A‘，如何求畸变前的点的坐标？用迭代的方式，我们先假设A‘点是B点，然后用B‘表示畸变后的点，这样就可以求出BB'之间的位移量，把这个位移量加到A‘上去得到C点，随后再求出C'点，求出CC'的位移量.....
一般迭代几次后，就可以求出A的位置了。 高度非线性很难根据畸变坐标反求去畸变坐标

vins不适用opencv自带的函数，而是自己实现去畸变的原因，是为了计算更快。

2 预积分

预积分本身在VINS中没有依赖其他的第三方库，主要是自己实现的。GTSAM等库中有现成的写好的预积分相关的代码可供调用。
和预积分相关的实现主要在intergration_base 的头文件内。本部分和光流部分类似，也属于预处理的部分。

2.1 基础知识回忆 参考 刘嘉林

首先回忆矩阵的表示方法
R=[3x3]的旋转矩阵，他的好处是可以直接进行矩阵运算。

$$R_{12} x R_{23} = R_{13}$$

但是旋转矩阵有个坏处，他只有3自由度，但是却有9个量表示，所以他有额外的约束，求导非常麻烦。我们没办法对旋转矩阵进行求导，如果考虑约束进行求导就很困难。

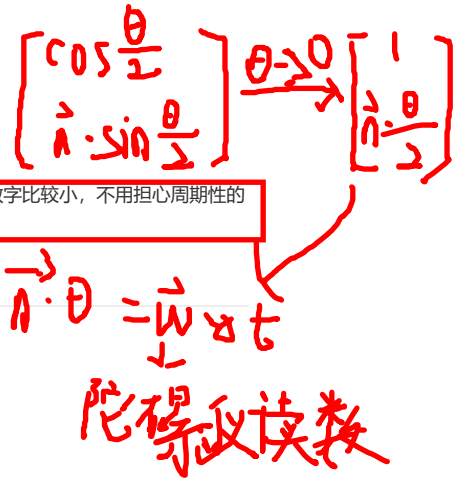
还有一种表示旋转的方式，是采用旋转向量。想象一下，我们绕一个任意轴旋转θ度数，那么就表示一个旋转。所以我们只要给出一个旋转轴和一个度数即可。这样我们只要3个数字就能表示。但是旋转向量有周期性，比如转360度和转0度是一样的。

还有欧拉角，他是把旋转拆解为绕不同方向的旋转。但是有个巨大的缺点，就是存在万向节死锁问题，无法进行球面平滑插值。
欧拉角的优点是交互性能好，容易理解。

四元数也挺好用的，他没有奇异性，只用4个数值表达，但是我们要求四元数是单位四元数。4个量表示3自由度的旋转还是导致了过参数化。描述四元数的方差也非常不方便。四元数的乘法是一套他自定义的乘法规则，不同于矩阵的乘法。

四元数和旋转向量和旋转矩阵可以转换。
四元数的虚部等于$cos \frac{\theta}{2}$,四元数的实部等于$sin\frac{\theta}{2}$乘上单位旋转向量。
当θ足够小的时候，我们会有一些近似的计算。 只有单位四元数才能表示旋转

当θ趋近于0的时候，$cos\frac{\theta}{2}$趋近于1，$sin\frac{\theta}{2}$趋近于$\theta/2$。
我们把IMU的陀螺仪的读数ω乘上一个时间，就得到旋转的变化。



我们预积分主要使用误差卡尔曼模型，维护的主要是误差量。所以不用担心旋转向量超过π的问题，因为数字比较小，不用担心周期性的问题。

2.1.1连续时间的PVQ积分

我们定义几个常用变量：P位置，V速度，Q姿态。

位置和速度的连续时间积分如下

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

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

第一个公式是速度、加速度和位置的关系公式的变种。其中考虑了加速度计中有重力的分量，所以公式会比较复杂。IMU的加速度计总会读到重力数据，所以一般要去掉加速度的信息。
考虑到VINS对加速度计建模的时候，还有零偏等数据，我们都要从数据中去掉这些干扰。
数据本身还有高斯噪声，但是这个就比较难以剔除了。
当然，IMU读到的数据并不是世界坐标系下的，所以还需要用一个R矩阵乘到世界系下。在世界系下，IMU的重力分量就应该是在Z轴了。

2.1.2 四元数的乘法

之前我们没介绍四元数的乘法，这里会详细的介绍。

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

四元数乘四元数还得到的是四元数。
我们希望写为矩阵相乘的方式，虽然eigen中已经重载好了。推导中我们还是要改写一下。

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

其中左右矩阵都可以改写为矩阵形式

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

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

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

$$[\mathbf{a}]_\times \mathbf{b} = \mathbf{a} \times \mathbf{b}, \quad \forall \mathbf{a}, \mathbf{b} \in \mathbb{R}^3.$$

四元数求导推导如下

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

where

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

2.2 IMU积分 卡尔曼滤波中，imu是在预测，相机是在更新

假设给定一个一维世界，我们大概知道自己的腿长，以及可以使用GPS，来推断我们当前的位置。卡尔曼滤波可以让我们得到一个最优估计。
我们从一个时刻开始，往前走（比如走了K步），此时你得到一个GPS信号，你通过你走路对自己腿长的估计，又估了一个位置。把两个位置用卡尔曼滤波融合后，就算出一个新的最优估计。

如果换成视觉和IMU的融合的话，IMU是和腿差不多的东西，知道走多远。GPS信号，就等同于图像帧给的定位。

我们通过IMU积分，得到位置的变化量，或者速度的变化量，以及姿态的变化量。
经过融合，得到新的置信度更高的姿态。
可以看出来，IMU在滤波中的主要功能是做预测。
在传统的滤波中问题中，IMU主要就是直接积分，不需要预积分。

回忆前面的内容，前端光流的计算结果，用不超过10HZ的频率向后端发送。而IMU通常在100Hz或更高，也就是说2帧之间有一些IMU的数据。
但是VINS采用的是优化框架，不是滤波框架。
在优化问题中，会采用多个过去的信息，来计算并更新当前和过去的状态。（滤波只会维护当前的位姿估计，不会对过去的状态进行修改）
在优化问题中，每2个关键帧之间的步长（用IMU积分出来），一般作为帧间约束。
此时，预积分量视为帧间位姿。
预积分的作用是形成预积分量，作为优化的约束。

2.3 离散时间预积分

vins中采用中值积分的方法。这里几个变量表示的含义分别为：α是相邻2帧位置的增量；β 速度增量；γ 是姿态的增量。 IMU的读取时离散的，在两帧IMU帧间是采取的中值积分

注意：对于加速度计，他的测量是包涵重力加速度的，所以我们一般要抵消掉这个重力加速度分量。

我们首先回忆中学物理学过的简单知识：

$$\triangle x = v_0 t + \frac{1}{2} a t^2$$

$$\triangle v = a t$$

我们加速度计读出来的也是a,在2帧IMU数据之间可能有好几个a 的数据，这里用中值积分近似，这样就可以认为上面公式里的a 是恒定不变的了。就不需要用积分的形式了。

当然要注意一点，IMU读数都是IMU坐标系下，要转移到世界坐标系下。

在预积分中，要注意所有的数据都是有坐标系的。vins用的是b_k 时刻的坐标系。

新来数据的时候，先更新姿态，后更新速度和位置。

我们需要提前把imu的数据积分起来，从而避免反复积分。希望这样的数据只和IMU相关，和前后帧无关。

此时我们还少考虑一个东西——置信度，协方差矩阵。
可以这么理解，如果我们闭着眼走10步，肯定对我们的位置特别不确定。如果闭着眼只走1步，那我们对自己的位置还是比较确定的。对于IMU也是如此，IMU数据多了，一般积分起来可信度就低多了。

2.4 误差卡尔曼

https://blog.csdn.net/whatiscode/article/details/126242319

$$\boldsymbol{\alpha}_{b_{k+1}}^{b_k} = \iint_{t\in[t_k,t_{k+1}]} \mathbf{R}_t^{b_k}\left(\hat{\mathbf{a}}_t - \mathbf{b}_{a_t} - \mathbf{n}_a\right)dt^2$$
$$\boldsymbol{\beta}_{b_{k+1}}^{b_k} = \int_{t\in[t_k,t_{k+1}]} \mathbf{R}_t^{b_k}\left(\hat{\mathbf{a}}_t - \mathbf{b}_{a_t} - \mathbf{n}_a\right)dt$$
$$\boldsymbol{\gamma}_{b_{k+1}}^{b_k} = \int_{t\in[t_k,t_{k+1}]} \frac{1}{2}\boldsymbol{\Omega}\left(\hat{\boldsymbol{\omega}}_t - \mathbf{b}_{w_t} - \mathbf{n}_w\right)\boldsymbol{\gamma}_t^{b_k}dt$$

分别对应位移、速度、姿态的预积分量。
但是我们得到的IMU数据是离散的。
所以采用离散的方式进行积分

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

这样两个图像帧之间若干个IMU数据的数值的预积分量就可以被计算出来了。(但是预积分量的置信度的求解还是不知道，我们会慢慢介绍)

VINS是用误差卡尔曼的理论进行相关的计算。
正常的卡尔曼滤波，是把预测值和观测值融合出最优估计值。随后再用这一次的最优估计的数值继续往下计算。
误差卡尔曼略区别于普通卡尔曼，我们扔进去融合的是误差量（观测的误差量，预测的误差量）。预测值的误差则就是"预测的误差量"。

但是这里又一个问题：预测的误差是多少呢？这在预测过程中，他的误差一直是0。想想一下，我们闭着眼睛走路，我们走的远后对自己当前位置的把握是越来越不确定。但是我们脑子里对自己当前位置的认知，都是认为"我就是觉得我在这个位置，不管对不对，但是我觉得在这个位置可能性最大"。 这也就是误差卡尔曼里，预测值的误差都是0。如果不是0的话，我们直接把误差加到预测里，来修正预测就行了。
换句话说，虽然我们预测的误差是0，但是预测误差的协方差是比较大的。

观测的误差是什么呢？是观测到预测的差。这个差也伴有一个协方差来刻画他的不确定性。

我们把这两个误差丢给卡尔曼，得到一个更好的误差，我们把误差补偿到我们的预测中，则可以得到置信度更高的位置。此时我们的误差又会变成0。我们再预测的过程中，我们的方差会越来越大，但是经过卡尔曼滤波后，我们的方差会相比于之前就会有所下降。卡尔曼滤波后的方差，会小于预测，也会小于观测。

回到问题：为什么我们要在IMU预积分中使用误差卡尔曼？
1、旋转的误差状态经常是很小的值，但是参数缺和旋转的自由度相等。
在误差卡尔曼中，我们用旋转向量表示误差。四元数是一个过参数化的表示形式，用4个数字表示3自由度的旋转。四自由度的协方差矩阵表示3自由度的旋转会很奇怪。为了解决这个问题，我们用3x3的协方差矩阵来表示旋转的置信度会比较好。
2、通常误差系统都是接近于0的。（我们的误差一般不会太大，所以接近0非常合理啊，误差太大的话，我们的系统就凉了）因此我们可以避免很多问题，如万向节死锁问题。用误差量可以避免这些问题。
3、误差很小，意味着二阶导基本可以忽略不计，所以求一阶导很快
4、误差变化很慢，我们用卡尔曼修正的话，我们的观测可以少一些。
其中2,3最重要。

预测值误差为0，根据优化观测值与预测值的误差；
在预测值误差基础上加上这个优化的误差，再加预测值，
最终得到最优估计值

2.4.1 误差卡尔曼中的一些表述

我们定义三类数值：真值： 很难有真值；名义值：估计出来的值，我们希望他和真值足够接近；误差值：名义值和真值的误差，显然误差值也是得不到的，如果知道误差值，我们就有真值了。所以通常我们认为我们的误差值是0，我们认为我们求得的名义值就是我们能得到的最优值。
我们用误差的协方差矩阵，来刻画我们对当前状态的不确定度。

加速度a(加速度计的读数)是真实值a_t+零偏b_a+噪声n_a。(忽略重力加速度)

$$a = a_t + b_a + n_a$$

vins中通常直接把读数减去bias就作为对当前加速度的最优估计。而传统误差卡尔曼还需要去掉噪声。
在IMU预积分中用误差卡尔曼可以避免IMU预积分过参数化。

2.4.2 推导

四元数也存在过参数化问题，旋转向量没有；
但是旋转向量有周期化问题，在一般的KF中没法用，
所以，四元数或旋转矩阵仍是KF中最好的选择；
现在是ESKF，误差很小时旋转向量不存在周期性问题，可用

首先推导论文中公式（29）

$$\begin{bmatrix} \delta\dot{\boldsymbol{\alpha}}_t^{b_k} \\ \delta\dot{\boldsymbol{\beta}}_t^{b_k} \\ \delta\dot{\boldsymbol{\theta}}_t^{b_k} \\ \delta\dot{\mathbf{b}}_{a_t} \\ \delta\dot{\mathbf{b}}_{w_t} \end{bmatrix} = \begin{bmatrix} 0 & \mathbf{I} & 0 & 0 & 0 \\ 0 & 0 & -\mathbf{R}_t^{b_k}\left[\hat{\mathbf{a}}_t - \mathbf{b}_{a_t}\right]_\times & -\mathbf{R}_t^{b_k} & 0 \\ 0 & 0 & -\left[\hat{\boldsymbol{\omega}}_t - \mathbf{b}_{w_t}\right]_\times & 0 & -\mathbf{I} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \delta\boldsymbol{\alpha}_t^{b_k} \\ \delta\boldsymbol{\beta}_t^{b_k} \\ \delta\boldsymbol{\theta}_t^{b_k} \\ \delta\mathbf{b}_{a_t} \\ \delta\mathbf{b}_{w_t} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ -\mathbf{R}_t^{b_k} & 0 & 0 & 0 \\ 0 & -\mathbf{I} & 0 & 0 \\ 0 & 0 & \mathbf{I} & 0 \\ 0 & 0 & 0 & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{n}_a \\ \mathbf{n}_w \\ \mathbf{n}_{b_a} \\ \mathbf{n}_{b_w} \end{bmatrix} = \mathbf{F}_t\delta\mathbf{z}_t^{b_k} + \mathbf{G}_t\mathbf{n}_t.$$

α 是位置
β 是速度
θ 姿态
b_a 加速度计的0偏
b_w 陀螺仪的0偏

我们需要知道他们的导数，其结论就是论文公式29的部分。我们这里需要对其进行推导。

2.4.2.1 位置

为什么需要知道它的导数？

我们以速度为例，位置的一阶导显然是速度。速度的真实值的导数，肯定是加速度真实值。
所以

$$P_t' = v_t'$$

名义值也应该是一样满足。

$$P' = v_t$$

那么误差值经过组合上述2个式子，得到

$$\triangle p' = \triangle v$$

所以$\delta \alpha ' = \delta \beta$.
而这就可以很轻松的把矩阵的第一行给写出来。

2.4.2.2 0偏

0偏的导数是一个均值为0的高斯分布。轻松得到矩阵第四，第五行的内容。

2.4.2.3 速度

速度的导数肯定是加速度。
预积分是b_k到b_{k+1}帧时间段之间的预积分，所以我们的速度是相对于k帧的速度。加速度计读的是当前姿态下，加速度的数值，我们需要把他转到b_k帧下的加速度。当然，我们要去掉0偏。
列出真实值

$$v_t' = R_{b_k,t}(a_t - b_a)$$

列出误差值

$$(v + \triangle v)' = R\triangle R(a - n_a - b_a - \triangle b_a)$$

通常我们把$\triangle R$写为$\exp(\phi\ \hat{})$的形式，约等于 $I + \triangle \phi\ \hat{}$
化简得

$$\triangle v' = R(-n_a - \triangle b_a) + \triangle \phi (a - b_a)$$

再化简得

$$\triangle v' = -R(a - b_a)\triangle \theta - Rn_a - R\triangle b_a$$

2.4.2.4 角度

回忆四元数的导数

$$\dot{q} = \frac{1}{2}\Omega(w)q$$

那么(自动把3x1矩阵变为4x1的四元数，补上虚部0)

$$\dot{q_t} = \frac{1}{2}q_t \otimes (w_t - b_{w_t})$$

得到

$$(q \otimes \delta q)' = \frac{1}{2}q \otimes \delta q \otimes (w - n_w - b_w - \delta b_w)$$

记得消除二阶无穷小量，代入名义值和真实值。
我们假设$y=w-b_w$, $x=w-n_w-b_w-\delta b_w$
得到

$$2\delta\dot{q} = ([x]_R \cdot -[y]_L) \cdot \delta q$$
$$= (\begin{pmatrix} 0 & -x^T \\ x & -x^\wedge \end{pmatrix} - \begin{pmatrix} 0 & -y^T \\ y & -y^\wedge \end{pmatrix})\delta q$$

再次化简得到

$$\begin{pmatrix} 0 \\ \delta\theta' \end{pmatrix} = \begin{pmatrix} 0 & -x^T + y^T \\ x - y & -x^\wedge - y^\wedge \end{pmatrix} \begin{pmatrix} 1 \\ \delta\theta/2 \end{pmatrix} =$$

所以

$$\delta\theta' = x - y - (x+y)\cdot\cdot\frac{\delta\theta}{2}$$

把x和y替换回去，就可以和论文中的公式对上了。

3.1 离散时间IMU误差状态推导

以下是完整的IMU推导!!

如何根据连续时间的方程推导为离散时间呢?
我们用$\dot{\delta x}=F_t\delta x+G_t n_t$表示上一章节的矩阵结果。
这里δx 是一个15x1的向量，
但是只有连续问题才可以求导，离散的是不能求导的。

我们需要求出离散时间下，误差的变化。也就是k和k+1时刻他们的变化量是多少。
正常来说，δt是IMU频率的倒数，一般在10ms左右。

我们回忆导数的含义，f(x+dx)约等于f(x)+J(x)dx
那么代入我们的问题就是

$$\begin{aligned}\delta x_{k+1} &= \delta x_k + \dot{\delta x}\triangle t \\ &= \delta x_k + (F_t\delta x + G_t n_t)\triangle t \\ &= (I + F_t\triangle t)\delta x_k + G_t\triangle t\cdot n_t\end{aligned}$$

此时，F_t和n_t都是时间相关的。VINS假设随机游走在2帧相邻之间，零偏不变。
所以我们把所有数值都列进去的矩阵如下

$$\begin{bmatrix}\delta\alpha_{k+1}\\ \delta\theta_{k+1}\\ \delta\beta_{k+1}\\ \delta b_{ak+1}\\ \delta b_{wk+1}\end{bmatrix} = \begin{bmatrix}I & f_{01} & \delta t & f_{03} & f_{04}\\ 0 & f_{11} & 0 & 0 & -\delta t\\ 0 & f_{21} & I & f_{23} & f_{24}\\ 0 & 0 & 0 & I & 0\\ 0 & 0 & 0 & 0 & I\end{bmatrix}\begin{bmatrix}\delta\alpha_k\\ \delta\theta_k\\ \delta\beta_k\\ \delta b_{ak}\\ \delta b_{wk}\end{bmatrix} + \begin{bmatrix}v_{00} & v_{01} & v_{02} & v_{03} & 0 & 0\\ 0 & \frac{-\delta t}{2} & 0 & \frac{-\delta t}{2} & 0 & 0\\ \frac{-R_k\delta t}{2} & v_{21} & \frac{-R_{k+1}\delta t}{2} & v_{23} & 0 & 0\\ 0 & 0 & 0 & 0 & \delta t & 0\\ 0 & 0 & 0 & 0 & 0 & \delta t\end{bmatrix}\begin{bmatrix}n_{ak}\\ n_{wk}\\ n_{ak+1}\\ n_{wk+1}\\ n_{ba}\\ n_{bw}\end{bmatrix}$$

15x1　15x15　15x18　18x1

$$f_{01} = \frac{\delta t}{2}f_{21} = -\frac{1}{4}R_k(\hat{a}_k - b_{a_k})^\wedge\delta t^2 - \frac{1}{4}R_{k+1}(\hat{a}_{k+1} - b_{a_k})^\wedge\left[I - \left(\frac{\widehat{\omega}_k + \widehat{\omega}_{k+1}}{2} - b_{\omega_k}\right)^\wedge\delta t\right]\delta t^2$$

$$f_{03} = -\frac{1}{4}(R_k + R_{k+1})\delta t^2$$

$$f_{04} = \frac{\delta t}{2}f_{24} = \frac{1}{4}R_{k+1}(\hat{a}_{k+1} - b_{a_k})$$

先算δθ → q姿态

$$f_{11} = I - \left(\frac{\widehat{\omega}_k + \widehat{\omega}_{k+1}}{2} - b_{\omega_k}\right)^\wedge\delta t$$

$$f_{21} = -\frac{1}{2}R_k(\hat{a}_k - b_{a_k})^\wedge\delta t - \frac{1}{2}R_{k+1}(\hat{a}_{k+1} - b_{a_k})^\wedge\left[I - \left(\frac{\widehat{\omega}_k + \widehat{\omega}_{k+1}}{2} - b_{\omega_k}\right)^\wedge\delta t\right]\delta t$$

$$f_{23} = -\frac{1}{2}(R_k + R_{k+1})\delta t \quad f_{24} = \frac{1}{2}R_{k+1}(\hat{a}_{k+1} - b_{a_k})^\wedge\delta t^2$$

$$v_{00} = -\frac{1}{4}R_k\delta t^2$$

$$v_{01} = v_{03} = \frac{\delta t}{2}v_{21} = \frac{1}{4}R_{k+1}(\hat{a}_{k+1} - b_{a_k})^\wedge\delta t^2\frac{\delta t}{2}$$

$$v_{02} = -\frac{1}{4}R_{k+1}\delta t^2$$

$$v_{21} = v_{23} = \frac{1}{4}R_{k+1}(\hat{a}_{k+1} - b_{a_k})$$

有了这些数据，我们就可以快了的通过第k项的误差状态，推出第k+1项的误差状态了。

3.2 预积分中IMU 零偏的建模

观测值等于真实值+噪声+零偏。

$$\begin{aligned}\boldsymbol{\alpha}_{b_{k+1}}^{b_k} &= \iint_{t\in[t_k,t_{k+1}]}\mathbf{R}_t^{b_k}\left(\hat{\mathbf{a}}_t - \mathbf{b}_{a_t} - \mathbf{n}_a\right)dt^2\\ \boldsymbol{\beta}_{b_{k+1}}^{b_k} &= \int_{t\in[t_k,t_{k+1}]}\mathbf{R}_t^{b_k}\left(\hat{\mathbf{a}}_t - \mathbf{b}_{a_t} - \mathbf{n}_a\right)dt\\ \boldsymbol{\gamma}_{b_{k+1}}^{b_k} &= \int_{t\in[t_k,t_{k+1}]}\frac{1}{2}\boldsymbol{\Omega}\left(\hat{\boldsymbol{\omega}}_t - \mathbf{b}_{w_t} - \mathbf{n}_w\right)\boldsymbol{\gamma}_t^{b_k}dt\\ \boldsymbol{\alpha}_{b_{k+1}}^{b_k} &\approx \hat{\boldsymbol{\alpha}}_{b_{k+1}}^{b_k} + \mathbf{J}_{b_a}^{\alpha}\delta\mathbf{b}_{a_k} + \mathbf{J}_{b_w}^{\alpha}\delta\mathbf{b}_{w_k}\\ \boldsymbol{\beta}_{b_{k+1}}^{b_k} &\approx \hat{\boldsymbol{\beta}}_{b_{k+1}}^{b_k} + \mathbf{J}_{b_a}^{\beta}\delta\mathbf{b}_{a_k} + \mathbf{J}_{b_w}^{\beta}\delta\mathbf{b}_{w_k}\\ \boldsymbol{\gamma}_{b_{k+1}}^{b_k} &\approx \hat{\boldsymbol{\gamma}}_{b_{k+1}}^{b_k}\otimes\begin{bmatrix}1\\ \frac{1}{2}\mathbf{J}_{b_w}^{\gamma}\delta\mathbf{b}_{w_k}\end{bmatrix}\end{aligned}$$

看，我们再IMU预积分的时候，这是连续时间的表达式。
VINS MONO是紧耦合的VIO，零偏是待估计量。在优化问题中，每次优化后，零偏就可能发生变化。
无论是加速度计还是陀螺仪的零偏，都会被优化所调整。当我们用IMU预积分结果，当我们计算的0偏变化后，我们预积分量可能要重算一遍的话，就会有巨大的工作量。
算积分的工作量还是比较大的（矩阵也不算小，乘起来也不算快），重新计算会有巨大的工作量。

零偏也是优化量，它是一个贯穿全局的值，PVQ在第N个图像帧积分结束了，等到N+1时刻，把零偏优化了一次，因为状态的估计是离散连续的，你不可能说之前的所有IMU预积分按照旧的零偏是错误的，你只能去补偿，还好零偏的优化不大

我们采取了一个方法来减少工作量，如果我们有预积分量对于0偏的雅克比矩阵j，我们就可以利用泰勒一阶展开的方式来近似计算。这样就可以大大降低运算的计算量。

我们之前得到 $\delta x_{k+1}=F \delta x_k + Gn$ 形式的式子，我们要对式子写为

$$
\begin{aligned}
x_{k+1} &= f(x_k) \\
x_{k+1} + \delta x_{k+1} &= f(x_k + \delta x_k) \\
x_{k+1} + \delta x_{k+1} &= f(x_k) + J\delta x_k \\
\delta x_{k+1} &= J\delta x_k
\end{aligned}
$$

上式和$\delta x_{k+1}=F \delta x_k + Gn$合并可以得到（忽略噪声项，他们和雅克比无关）

$$
J_{k+1} = FJ_k
$$

现在我们可以在K时刻的雅克比矩阵，乘上F后就得到K+1时刻的雅克比矩阵了。
最后算出来的雅克比矩阵，是最终的预积分量对自身的雅克比矩阵。

3.3 协方差计算 使用ESKF的原因之一就是方便协方差的计算，因为可以直接略取二阶误差小量

如果a是一个向量，他的方差是A
那么Fa的方差是 FAF^T

如果b是向量，他的方差是B
那么a+b的方差是A+B

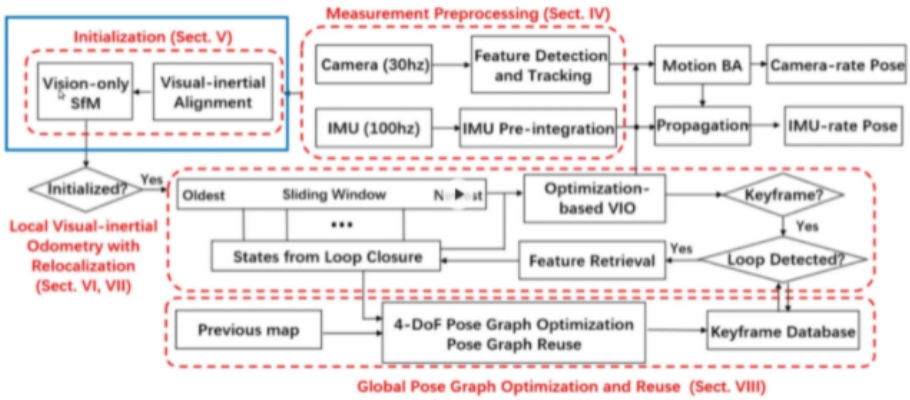
那么对于 $\delta x{k+1}=F \delta x_k + Gn_k$ ，如果 $x{k+1}$的协方差是P_k, n_k的协方差是V_k, 那么新的δx_{k+1}的协方差就是$FP_kF^T+GV_kG^T$.

4 VIO 初始化

预积分的代码就一个头文件，但是涉及的公式特别多。包括离散时间状态量，误差量的的计算。
根据连续时间的导数关系，得到离散时间的dx和dx+1的关系，再根据关系，算出预积分量的协方差矩阵（置信度）。
最后算预积分零偏的雅克比矩阵，这样适合微调。

VIO的初始化的代码量会多一些，他不是简单的对传感器的处理方式，还涉及视觉和惯性对齐，还涉及纯视觉SLAM的部分。相关的理论和公式也比较多。这一章节的公式和代码都比较多。

在paper中，初始化的部分属于蓝色的方框内。



sfm是用纯图片做三维重建。再用预积分的结果进行对齐，实现初始化的过程。
我们已经讲过图像和IMU的预处理，初始化就是利用他们的结果的第一个应用。

4.1为什么要初始化?

VINS是基于优化的VIO，非常依赖初始值。优化问题，经常用迭代下降的方式，初始值不好的话，会陷入局部最优解（或者不是我们想要的部分极小值）。 为了保证系统的极小值是我们要的，那么初始值就需要距离正确值比较近。这样迭代次数也少一些，也节约了计算时间和运算资源，也可以保证实时性。
但是初始化很难给到完全接近的装状态，所以这里采用松耦合的方式。

4.1.1初始化哪些变量?

$$
\begin{aligned}
\mathcal{X} &= [\mathbf{x}_0, \mathbf{x}_1, \dots \mathbf{x}_n, \mathbf{x}_c^b, \lambda_0, \lambda_1, \dots \lambda_m] \\
\mathbf{x}_k &= [\mathbf{p}_{b_k}^w, \mathbf{v}_{b_k}^w, \mathbf{q}_{b_k}^w, \mathbf{b}_a, \mathbf{b}_g], k \in [0, n] \\
\mathbf{x}_c^b &= [\mathbf{p}_c^b, \mathbf{q}_c^b]
\end{aligned}
$$

论文里优化如上这些变量。分别是滑窗，状态，和外参。

外参是需要求解的，零偏也是需要优化的变量。
λ是地图点，是为了更好的优化位姿。地图点也是我们要求解的，我们希望通过初始化来求解。

有些变量不是通过初始化得到的，比如加速度计的零偏，初始化很难得到准确的加速度零偏的结果。
平移外参也很难求解，VINS里是不去估计他们的，直接假设已经提前知道了平移外参。

加速度计的零偏不好求，就算我们忽略掉，对结果的影响也比较小。在后端中，加速度计的零偏再进行优化。初始的时候零偏当做0直接算，对结果影响不大。

作者认为我们不需要特别准确的外参。因为在非线性优化里，我们会持续的优化这个外参。所以一开始不需要特别准确。之后我们再准确的估计出来"

为什么不优化加速度计的零偏？因为他很难从重力中区分出来。就算忽略掉，也对结果影响不大。

4.2 后端优化预处理

简单的讲解一部分代码。
和之前的代码类似

```
readParameters
```

是读参数用的，通过ROS参数服务器获取配置文件。这些参数是yaml格式的。
这里要设置单次优化的最大求解时间，最大迭代次数，以及根据视差确定关键帧。
VINS判断关键帧比较简单，就是判断2帧之间的视差。同一个特征点，在2个图像里像素变化太大了，就视为关键帧。
在这里还要注意一下IMU的一些参数：噪声和随机游走的的参数，都是提前获得的。

如果外参足够精确，是提前给定的，那么就不需要优化外参了。
如果完全不知道外参，那么就认为外参是单位矩阵。

后端优化还有一个功能，估计传感器和相机之间的时间误差问题。后面会介绍这个部分。

对于视觉，也肯定需要重投影误差的准确度的衡量。这里是通过虚拟相机，统一设置成1.5个像素的误差。

4.2.1 predict

算2个时间戳，算IMU之间的时间差。提取加速度，角速度，去掉bias和重力影响。根据上一个时刻，IMU在世界坐标系下的位姿，根据2个时刻IMU数据的平均值，视为这一小段时间的IMU的位姿在世界的变化量。
这时候，就可以算出最新时刻的IMU在世界的位姿，以及当前的速度。

4.2.2 相机和IMU时间对齐 getMeasurement函数

这一步操作的主要原因，是IMU和相机可能时间并不完全同步。 IMU数据通常要密集一些，但是还是不会很巧合的和相机数据的时间完全一致，这样的话，我们就算出一个虚假的IMU（通过各种方法）插入到和图像对齐的时间位置上。

在这里，我们需要保证最后的IMU时间要大于图像帧最早的时间。如果IMU最新的时间比图像还要早，那么就需要等待新的IMU数据。
如果IMU最早的时间戳，也显然需要小于时间的时间戳。这样的话，就扔掉图像，再等一个新的图像。
保证了每个图像的时间被2个IMU夹着，就可以进行对齐操作了。

4.2.3 processIMU函数

主要功能是更新预积分量，执行预积分的操作。
其次功能是是给优化变量提供初始值(利用IMU进行状态推算)，
这里会整个代码是在一个死循环里。首先用时间对齐后的图像和IMU的数值。遍历每一组匹配好的IMU和图像的数据。整理好所有对齐好的数据，调用processImage。这个函数我们之后还会多次涉及。

4.2.4 processImage函数

第一个事情是对特征点信息进行预处理。把特征点加入特征点管理器，并判断是否是关键帧。

VINS是对每个特征点进行编号，用一个list维护。每一个元素包涵id，起始帧，深度，求解状态，和属性。
起始帧就是从第几帧看到这个特征。深度是在起始帧下的深度。求解状态是是否被正确的三角化出来。一个ID可能被多帧看到，所以他还有在其他帧的属性。这里要记录他在其他帧中的像素坐标，归一化相机坐系坐标，以及特征点速度。这里就需要维护上述的信息。

这在addFeatureCheckParallax函数中，是主要是检查上一帧是否时关键帧。
如何判断一个帧是关键帧？如果他是第一次出现的帧，肯定是关键帧。如果当前帧追踪到的点过少，也认为上一帧就是关键帧。判断关键帧主要看倒数第二帧，和倒数第三帧是否视差大，如果视差大，那么也可以视为关键帧。
如果得到了关键帧，那么就需要把滑动窗口里最早的一帧给扔掉。如果视差不够的话，并且窗口满了的话，那么倒数第二帧就需要被弹出了。
判断完当前图片是否是关键帧后，处理好滑动窗口队列，并整理数据。接下来的任务就是初始化。

4.3 旋转外参初始化

本节内容主要是初始化处理的工作，首先是外参初始化。
外参可能有先验，或者完全没有先验的数值，甚至肯能有非常优秀的外参数值（后面计算不需要再额外优化）

本节首先介绍如何标定旋转外参。

首先假设IMU b_k和b_{k+1} 时间IMU之间的变化，我们也知道2个时刻图像帧之间的变化。
标定旋转外参的时候用不着t，只考虑q（四元数）之间的变化。

我们是知道IMU在2个时刻之间的旋转的，也知道图像帧2个时刻之间的变化。
那么我们可以直接利用b_k时刻的外参，表达出b_k时刻相机到$b*{k+1}*$时刻*imu*系的旋转。
这里有两种途径，一种是用*IMU*积分的结果作为*b_k*到*$b*{k+1}$时刻旋转的“桥梁”，另一种方法是用图像帧之间的旋转作为“桥梁”。

$$q_{cb} \otimes q_{b_k b_{k+1}}$$
$$q_{c_k c_{k+1}} \otimes q_{cb}$$

公式中$q{cb}$是外参。
显然两个式子是相等的。我们的目的是求解$q{cb}$，实际上这是一个经典的AX=XB问题。
四元数可以通过变换转为矩阵的乘法，这样四元数就被转化为矩阵乘法问题了。
这里化简可以根据四元数的特点，可以轻松的把四元数乘法右边的部分转化到左边（四元数乘法可以有右矩阵，或者左矩阵的方式）。这样的话最后的形式可以被化简为
$Rq{cb}=Lq{cb}$
其中R和L分别是$q{b_kb{k+1}}$的右矩阵，以及$q{c_kc{k+1}}$的左矩阵。
这样就可以继续写为
$(R-L)q{cb}=0$
其中R和L都是已知的矩阵。
当然，我们不仅仅有k，还有k+1，k+2等帧，这样就列为了

$$\begin{aligned}(R_k - L_k) \cdot q_{cb} &= 0 \\ (R_{k+1} - L_{k+1}) \cdot q_{cb} &= 0 \\ (R_{k+2} - L_{k+2}) \cdot q_{cb} &= 0 \\ &\vdots \\ (R_{k+n} - L_{k+n}) \cdot q_{cb} &= 0\end{aligned}$$

聪明的小伙伴可以一下子知道怎么解这个问题了。
写为矩阵的形式

$$\begin{pmatrix} R_k - L_k \\ R_{k+1} - L_{k+1} \\ R_{k+2} - L_{k+2}) \\ \vdots \\ R_{k+n} - L_{k+n}) \end{pmatrix} q_{cb} = 0$$

左边有n+1行个4x4矩阵，是4(n+1)x4的矩阵。中间是4x1的矩阵。我们用A表示**最**左边这个矩阵，他是已知的。唯一未知 的的是q_{cb}。

下面的任务就是解方程了。

4.3.1 代码

4.3.1.1 getCorresponding

这个函数是得到2帧看到的特征点，并计算他们在各自帧下的坐标是什么样子。
首先遍历所有的featrue，要确保这个feature同时被查询的2个帧的时间都能对的上。
然后把他们在2帧下归一化相机坐标系的坐标捆绑起来，整个打包发出来。这就是上述函数的主要功能和实现。

4.3.1.2 CalibrationExRotation

初始化旋转外参。
首先用之前介绍的返回内的内容，直接计算出2帧的旋转，

4.3.1.3 计算2帧的旋转 solveRelativeR

输入就是2图像之间被绑定好关系的**特征点对** (注意是点对)。
首先需要保证要有9对以上的匹配。（想一想，匹配点多，也会精度高。）
调用opencv的函数findFundamentalMat函数，得到本质矩阵（代码里用E表示）。
随后对矩阵E进行分解（参考《Multiple View Geometry in Computer vision》期待开课讲这本书）就可以得到其中的旋转部分。
本质矩阵分解有4种不同的结果（十四讲里也有说），随后我们check这4个解哪一个可以用。其行列式的值必须为1。这里还用了测试三角化的方式进行检查(函数testTriangulation)。
已知道Rt的变化量，以及知道2帧的点对关系，利用opencv的函数trianglePoints实现三角化（得到三角化后的点云）。
根据三角化点云的坐标，我们可以肯定这些点的深度在2帧里都是大于0的，我们统计深度大于0的点占所有点的比例。比例越大，这样的求解是正确的可能性越大。
这样我们就利用图片得到了2帧之间的旋转了。

4.3.1.4 构建A矩阵

我们可能求解出很多相邻帧之间的旋转，我们把他们都用一个vector保存起来。类似的，我们还有很多相邻帧之间IMU积出的旋转，用另外一个vector保存起来（当然，这里要注意一下IMU坐标系和相机坐标系的转换）。

此时构建上文所述的A矩阵所需的材料已经足够了。
但是我们依旧需要注意一些细节。如果某个相邻帧之间，IMU积分的旋转和图像估算的旋转差距太多了怎么办?
代码里，根据误差的度数来降低旋转误差超过5度权重。

构造L和R矩阵，就利用四元数乘法转化矩阵乘法的方式进行转化。注意代码里是虚部在前，实部在后表达四元数。
注意最后要每一行还要乘上权重。

4.3.1.5 求解外参

此时是一个Ax=0超定方程的问题。
A是Nx4，x是4x1的矩阵，我们只能得到最小二乘解。
我们对A进行SVD分解得到$A=UDV^T$
写下来就是$UDV^T\cdot \mathrm{x}=0$
其中V是奇异向量(4x4矩阵)，D是奇异值(Nx4的矩阵)，U是NxN矩阵。
其中U和V是正交矩阵，我们的目标是让式子左边部分尽可能接近0（一般误差的存在，等于0几乎不可能）。
所以问题可以转化为求解

$$||UDV^T\cdot \mathrm{x}||_{min}$$

的最小值。
其中U是正交矩阵，我们知道正交矩阵乘一个向量，只改变向量的方向，不会改变向量的模长。所以问题再次等价为

$$||DV^T\cdot \mathrm{x}||_{min}$$

的最小值。
我们知道V也是正交矩阵，所以把$V^T \cdot \mathrm{x}$用y 来表示，因为V也是正交矩阵，也只能改变x的方向，不太能改变太多。
问题转化为

$$||D\cdot \mathrm{y}||_{min}$$

的最小值。
此时多了一个约束，x的模为1.所以y的模肯定也是1.
D是一个对角矩阵，只有前4行有值。

$$D=\begin{pmatrix}\sigma_1 & 0 & 0 & 0\\ 0 & \sigma_2 & 0 & 0\\ 0 & 0 & \sigma_3 & 0\\ 0 & 0 & 0 & \sigma_4\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ \vdots & & & \\ 0 & 0 & 0 & 0\end{pmatrix}$$

并且σ还从大到小排序
那么显然最后结果就是

$$\sum_{i=1}^{4}\sigma_i y_i$$

其中又因为y的模长为1，则我们把所有y的权重都分给最小的σ_4,那么y的取值就是$[0\ 0\ 0\ 1]^T$肯定是最优解。
所以当$V^T \cdot x=[0\ 0\ 0\ 1]^T$ 的时候是最优解。根据V矩阵的特点（正交，且每一列模长为1），x取V矩阵的第4列即可。
没看懂？没关系，更加详细的推导，我推荐看这一篇文章 点这里。

回到代码，代码里用的是也是现成的SVD分解进行求解。代码里多了一个判断，
1、要积累足够的旋转匹配对（匹配多，可信度更高）
2、检查倒数第二个奇异值要大于0.25。如果有几个奇异值都约等于0的话，那么可能数值不太稳定。这意味着整个系统的旋转太少。

最后，如果CalibrationExRotation函数返回了true，我们就得到了的旋转外参值。

5.1单目视觉位姿估计

求解完IMU和图像旋转外参标定后，就到了解决基于图像纯视觉单目SLAM问题了。

我们希望有足够多的数据再滑窗里，再进行单目SLAM三维重建。

我们还希望有可信的外参，所以一般初始化失败后，我们希望稍微等那么0.1秒后再进行初始化（不要掉进同一个坑2次，还在这个地方初始化很可能还会失败）。

满足数据帧足够多，同时没有初始化失败的话这两个条件后，就可以进行单目SLAM了。
我们的目的是求解滑窗里的位姿和3D点。整个过程一共分为四步:
首先check一下IMU的能观性，希望IMU得到足够的激励。
第二步是简单的SFM，简单的三维重建。这一步只针对滑窗里的东西
第三步是利用滑窗的位姿，对所有帧进行恢复。恢复完后，就进行视觉，惯性的对齐。
第四步：视觉重建的结果和IMU结果进行对齐。

5.1.1 check一下IMU能观性

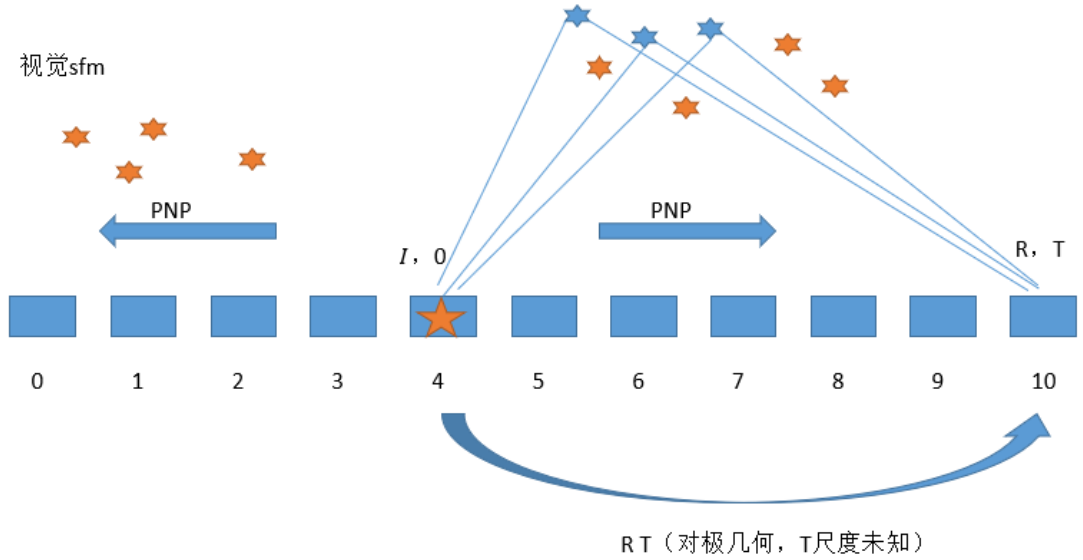
这一步我们求解IMU在一段时间里加速度的方差，如果方差较大，说明IMU得到了充分的激励（动的挺都）。方差较少可能IMU没怎么动。
作者的代码里最终并没有区分两种情况，只是写在了这里。可能作者经过实验发现这个影响较小，所以作者注释掉了这段代码。

5.1.2 全局单目SLAM

首先遍历特征点管理器的特征点。对每一个特征点，我们都默认他**没有**被三角化成功，并且保存特征点在每一帧中的性质。
VINS用一个vector保存所有的特征点。
首先假设一共11帧数据，我们首先要选择一个“枢纽帧”。我们希望枢纽帧尽可能距离最后一帧比较远，如果比较近的话，平移就比较小，这种情况下分解出来的移动接近于0，这样不太好。我们不希望有“纯旋转”。但是不希望太远，因为太远的话，2帧的关联就比较少（能关联的特征点比较少），这样也不好。

在保证匹配点足够多的情况下，距离最后一帧尽可能的远。

假设选择了第四帧，如图所示



假设他的旋转是单位矩阵，平移是0.通过对极几何，求解出4和10之间的相对位姿态（R和t，但是t尺度未知）。
这样我们就得到了第4，和第10帧的位姿。
此时我们可以三角化出很多点的坐标。

有了更多被三角化后的点，我们就可以用pnp的方式求解出第5-第9帧的位姿。此时我们可以三角化出更多的点（比如在第五帧，我们又有了更多的特征点，从而可以三角化出更多的点）

求解完枢纽帧到最后一帧之间的位姿后，前几帧的方式也用类似的方法即可得到他们的位姿。

5.1.3 relativePose 寻找枢纽帧的函数

首先求解枢纽帧和最后一帧的Rt。
我们穷举每一帧到最后一帧的匹配对，如果匹配对超过20个匹配，则可以进行一些检查看看这帧是否符合要求。
如何检查呢? 计算2帧的平均视差，得大于30个像素。
随后求解2帧的Rt。

求解Rt的函数是solveRelativeRT,这里依旧是调用opencv的函数求解本质矩阵E。得到E后，直接用opencv的函数cv::recoverPose 分解出R和t。这里要求判断求解E的内点的数量，必须大于12个才认为是可行的解。

得到Rt后，就有了枢纽帧到最后一帧的关系了。

下一个任务就是通过枢纽帧和最后一帧，求解这之间其他帧的Rt。
这些事情全部都是在sfm.construct函数里完成。

5.1.4 sfm.construct函数

这里注意一个细节，通常视觉slam维护的是Tcw，但是vins维护的是Twc。

这里着重讲一下三角化部分。他求解的是frame0和frame1之间的frame的三角化。
遍历特征点，并且判断特征点是否已经三角化过。如果没三角化过的点才需要三角化。这些特征点需要同时被frame0和frame1看到，才进行三角化操作。只不过这三角化中，额外增加了一个观测的信息。

这里三角化没有调用opencv的接口，是作者自己写了一个过程。大概含义推导如下：

$$[x\ y\ z]^T = [R\ t] \cdot p$$

其中p是世界点坐标是未知的，Rt是2帧之间的位姿关系。[x y z]是在各自相机系下的观测，也是已知道的（一般是归一化相机坐标系，所以z通常是1）。为了考虑z为1的情况，让公式依旧成立，我们添加系数a

$$[x\ y\ 1]^T = a[R\ t] \cdot p$$

我们改写一下矩阵[R t]部分。这个部分是3x4的矩阵。

$$[R\ t] = \underbrace{\begin{pmatrix} p_0 \\ p_1 \\ p_2 \end{pmatrix}}_{3\times4\ \text{Matrix}}$$

其中p0,p1,p2是1x4的矩阵。
合并3x4矩阵和p后得到

$$[x\ y\ 1]^T = a \begin{pmatrix} p_0 \cdot p \\ p_1 \cdot p \\ p_2 \cdot p \end{pmatrix}_{3\times1\ \text{matrix}}$$

此时左右两个向量是平行的，**并且叉乘为0。**
写为向量叉乘的形式后，得到

$$yp_2p - p_1p = 0$$
$$p_0p - xp_2p = 0$$
$$xp_1p - yp_0p = 0$$

整理得到

$$(yp_2 - p_1)p = 0$$
$$(p_0 - xp_2)p = 0$$
$$(xp_1 - yp_0)p = 0$$

随后可以得到

$$x = ap_0p$$
$$y = ap_1p$$

回带得到

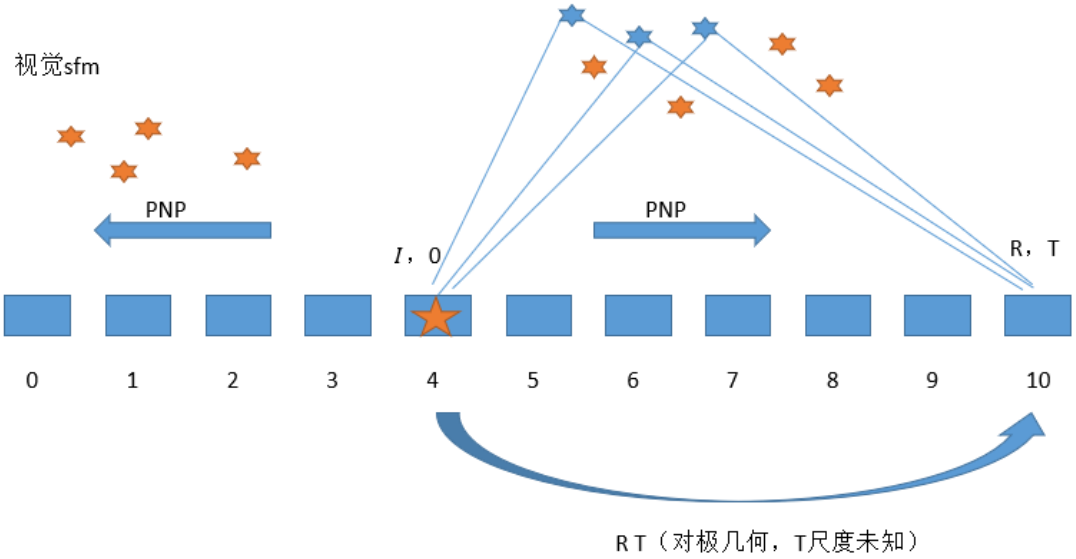
$$ap_0p_1 - ap_1p_0 = 0$$

此时发现整理的第三个式子都没有意义，根本不需要(一定成立)。
我们一个pose可以得到2个式子，如果有2个pose就可以得到4个式子。也就是函数triangulatePoint中有4行的原因。
如果我们一个特征点被多个帧看到的话，还可以继续把这个矩阵写的更大。此处依旧把问题改为AX=0的问题，还是可以用奇异值分解的方式进行求解。

此时我们完成了2帧之间三角化的过程。

5.1.5 solveFrameBvPnp

这是通过pnp求解其他帧的过程。
主要内容依旧是遍历所有特征点，我们只使用三角化成功的特征点。当然我们还要检查这些三角化后的点，看他们是否能被当前要做PNP的这一帧能否看到。
我们还希望能用的匹配点尽量多一些，太少则认为匹配失败。这里的PNP求解是直接调用opencv的接口进行PNP求解。特别的，这里我们用上一帧的位姿作为初始值进行求解。此时函数帧的位姿就可被求解出来了。
现在我们也有了新的帧的位姿，可以去和最后一帧进行三角化，从而三角化出更多的点，来维持整个系统继续工作下去。



从这个图上来看，我们可以把4-10帧的位姿都求解出来，考虑到三角化的策略是用当前帧和最后帧进行三角化，但是部分特征点不能被最后一帧看到的特征点，就不能被三角化出来。我们还需要把这些不能三角化的特征点，去和枢纽帧进行三角化。

此时0-3帧的数据还没有被求解。因为3距离4最接近，我们尽量把能和枢纽帧匹配的特征点给三角化出来。比如我们先求解第三帧的位置，随后再求2，求1最后求第0帧。

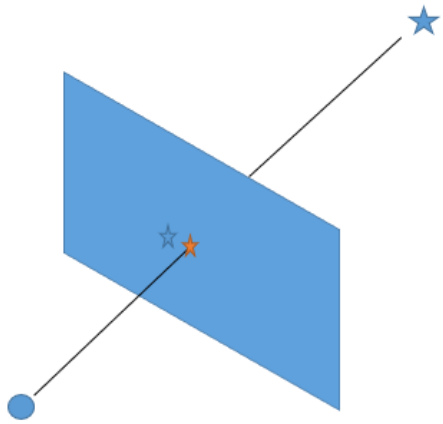
最终，我们求出了整个滑窗中的位姿，和一堆三角化后的点。我们依旧有一些点没有被三角化，我们遍历所有的特征点，用他们最初和最后一帧的位姿进行三角化。总之就是想尽办法把所有点都三角化出来。

此时我们三角化的并不太准，因为我们只是用2帧进行简单的三角化。为了让结果更加精确，我们还是做一个全局BA。我们之前的工作，就是给BA一个初始值而已。

5.2 基于ceres自动求导

平移是没办法得到尺度的，单目slam是没有真实坐标的。这也是我们亟待解决的问题。
vins是利用ceres进行优化求解的。ceres有多种求导方式，数值、解析和自动求导。其中自动求导就不需要我们人工进行雅克比矩阵的求解了。

在初始化问题中，我们要优化的**状态量x**，包括滑窗中11帧的位姿，和一堆三角化的点。
我们希望优化使得重投影误差最小



图中，蓝色的星，是3D点的位置，圆球是相机的位置。理论上，我们可以求出五角星在相机的像素平面上和我们检测的像素有多大的像素差（蓝色平面上，2个五角星之间的距离）。不管是调整3D点，还是调整相机位置，我们希望最后结果重投影误差最小。

那么如何求解这个x呢？交给ceres求解即可。
ceres的优点是我们只需要定义残差，就可以自动计算了。缺点也很明显，是自动求导是需要时间的，肯定不如直接给出雅克比矩阵的解析要快。解析求导需要手动求，大量手动求导的赋值工作很影响代码的美观。我们初始化的部分对实时性要求略低，就不需要解析解了。

正常来说，迭代优化问题需要有“变化量”作为步长，但是四元数是没有“加法”的。但是ceres帮我们写好了关于四元数相关的功能，所以使用ceres来进行优化很省功夫。

这里要注意，枢纽帧的旋转不参与优化的，最后一帧的平移也不参与优化。这样整个系统的尺度就不会发生变化了。

经过优化，如果求解成功，我们就得到了滑窗中所有的信息了。如果求解失败，则丢弃最小帧，准备下一次求解。

我们有了关键帧后，就可以恢复出其他普通非关键帧的位姿了。我们遍历这些普通帧，如普通帧的时间戳和滑窗一样，则说明他是关键帧。
对于关键帧，我们就直接得到了这个帧的位姿态，记录下即可。
如果不是关键帧，那么就用PNP求解位姿即可。

5.3陀螺仪零偏初始化

视觉惯性是要进行对齐的，但是这我们陀螺仪有个0偏问题。
加速度计的0偏和IMU外参，不在我们初始化中估计，我们只估计了旋转的外参和陀螺仪的0偏。

$$\mathbf{q}_{b_k}^{c_0} = \mathbf{q}_{c_k}^{c_0} \otimes \left(\mathbf{q}_c^b\right)^{-1}$$
$$s\overline{\mathbf{p}}_{b_k}^{c_0} = s\overline{\mathbf{p}}_{c_k}^{c_0} - \mathbf{R}_{b_k}^{c_0}\mathbf{p}_c^b$$

公式是图像枢纽帧0到第k帧的IMU坐标系下的旋转和平移的方法。

0是枢纽帧，c是相机系
b_k是第k帧在imu系下。
c_k是第k帧在相机系下。
q^b_c是外参，旋转外参的估计量我们之前是已经求解出了。
第二行是位移。视觉估计的位移是带尺度的，其中最重要的p_c^b也是假设已知的。
第二行的$R_{b_k}^{c_0}$是第一行的等式左边部分。

$$\min_{\delta b_w} \sum_{k\in\mathcal{B}} \left\| {\mathbf{q}_{b_{k+1}}^{c_0}}^{-1} \otimes \mathbf{q}_{b_k}^{c_0} \otimes \boldsymbol{\gamma}_{b_{k+1}}^{b_k} \right\|^2$$
$$\boldsymbol{\gamma}_{b_{k+1}}^{b_k} \approx \hat{\boldsymbol{\gamma}}_{b_{k+1}}^{b_k} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\mathbf{J}_{b_w}^{\gamma}\delta\mathbf{b}_w \end{bmatrix}$$

上面公式中γ是预积分得到的旋转的数值。正常来说，这个公式括号内的内容应该是单位矩阵，因为IMU积分出的旋转应该和相机计算的旋转几乎一致。但是因为有0偏的存在，所以这个竖线括号里的内容不是单位阵。
其中γ的数值包涵了0偏，所以他是理论值$\hat{\gamma}$再乘一个0偏。带来的影响。

理想情况第一行的乘积结果是单位四元数，
也就是如下

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = {q_{b_{k+1}}^{c_0}}^{-1} \otimes q_{b_k}^{c_0} \otimes \gamma \otimes \begin{pmatrix} 1 \\ \frac{1}{2}J_{b_w}^{\gamma}\delta b_w \end{pmatrix}$$

我们把等式右边的前3项移到左边来，

$$\gamma^{-1} \otimes q_{b_k}^{c_0 -1} \otimes q_{b_{k+1}}^{c_0} = \begin{pmatrix} 1 \\ \frac{1}{2} J_{b_w}^{\gamma} \delta b_w \end{pmatrix}$$

显然上面等式，他们的虚部和实部都应该相等。我们用vec表示等式左边部分的虚部。
那么就可以得到(省略各个角标)

$$\frac{1}{2} J \delta b = \text{vec}$$

挪动1/2得到

$$J \delta b = 2\text{vec}$$

我们观察这个式子，其中γ是通过预积分量已经知道的，$q{bk}^{c_0}$等是可以求得的。唯一的未知就是δb这个*bias*了。他就是我们求的*bias*的变化量。这就是我们要补偿多少的*bias*，才能让整个式子的结果更接近实际情况（结果为单位矩阵）。
这是一个*Ax=b*的模型，
我们这只有*b_k*到*$b*{k+1}$的部分，还可以添加到b_{k+2}的部分。我们有很多组方程，如何求解Ax=b模型的最小二乘解呢?
我们最小化$||Ax-b=0||$的结果即可。有一个简单的记忆求解解析式的方法如下（并不是推导！）

$$\begin{aligned} Ax &= b \\ A^T Ax &= A^T b \\ x &= (A^T A)^{-1} A^T b \end{aligned}$$

那么就得到x的解。也就是0偏的估计量。
代码里使用eigen的ldlt函数直接求解。

这些内容对应代码solveGyroscopeBias函数。我们得到bias后，把这个bias补偿到滑动窗口里，同时更新预积分量中的所有bias有关系的数值。这里重新更新预积分量，只在初始化里这么做。在之后的滑窗优化里，就不会这么做了。因为最早的时候我们没有对bias进行估计。

5.4 视觉惯性对齐求解

再LinearAlignment函数里，会求解各帧的速度，以及通过预积分恢复视觉的尺度，然后把重力的方向恢复过来，都是再这个函数里完成的。

本节主要讲解这个函数是做了什么了。
假设我们已经知道了预积分量 α,β,γ, 那么重力方向也是我们要估计的状态量，再这个过程中，我们要求解出尺度、重力方向、以及每一帧的速度。
再vins的优化器中，我们要优化的是位置和姿态，姿态我们有了，但是没对齐到重力坐标系下。
位置我们差一个尺度，还有bias。但是加速度计的bias我们是不估计的，陀螺仪的bias我们刚刚已经估计出来了。外有旋转外参我们也估计了（平移外参我们当做是已知量），还差一个速度需要估计。

根据预积分的定义

$$R_{c_0}^{b_k} P_{b_{k+1}}^{c_0} = R_{c_0}^{b_k} (P_{bk}^{c0} + v_{bk}^{c0} \triangle - \frac{1}{2} g^{c0} \triangle t^2) + \alpha$$

(c0换位w的话，就和课堂上推导的完全一致了)
因为我们还没有重力方向，所以我们还不能把c0写为w。

下面的事情就是根据之前已知的式子

$$s\overline{\mathbf{p}}_{b_k}^{c_0} = s\overline{\mathbf{p}}_{c_k}^{c_0} - \mathbf{R}_{b_k}^{c_0} \mathbf{p}_c^b$$

根据几个重要已知量，最终对应论文里的公式（10）

$$\begin{aligned} \mathcal{X}_I &= \left[\mathbf{v}_{b_0}^{b_0}, \mathbf{v}_{b_1}^{b_1}, \dots \mathbf{v}_{b_n}^{b_n}, \mathbf{g}^{c_0}, s \right] \\ \hat{\mathbf{z}}_{b_{k+1}}^{b_k} &= \begin{bmatrix} \hat{\boldsymbol{\alpha}}_{b_{k+1}}^{b_k} - \mathbf{p}_c^b + \mathbf{R}_{c_0}^{b_k} \mathbf{R}_{b_{k+1}}^{c_0} \mathbf{p}_c^b \\ \hat{\boldsymbol{\beta}}_{b_{k+1}}^{b_k} \end{bmatrix} = \mathbf{H}_{b_{k+1}}^{b_k} \mathcal{X}_I + \mathbf{n}_{b_{k+1}}^{b_k} \end{aligned}$$

这里要额外提一下，如何搞一个Hx=b的问题。其中x矩阵是第k帧的速度、k+1帧的速度以及c0系下重力g以及一个尺度s组成。
这是一个 6x10的矩阵乘一个10x1的矩阵，得到的b矩阵是6x1的矩阵的过程。
但是我们可以把很多个过程拼成一个大的矩阵来一次求解。这里我们重新定义一个x矩阵，他由n个速度向量矩阵，再加上一个g。那么这个大的新x向量的大小就是3n+4（速度和g都是3个数字组成，尺度s是一个数字组成，所以一共3n+4个数字）。这个数字也对应代码里n_state的定义。
我们前面介绍过求解Hx=b的方法（加转置的方法），

这里需要做的就是把下面的矩阵改写为一整个矩阵就行了。

$$\begin{pmatrix} & \\ & \end{pmatrix} \begin{pmatrix} v_k \\ v_{k+1} \\ g \\ s \end{pmatrix} = \begin{pmatrix} \\ \end{pmatrix}$$

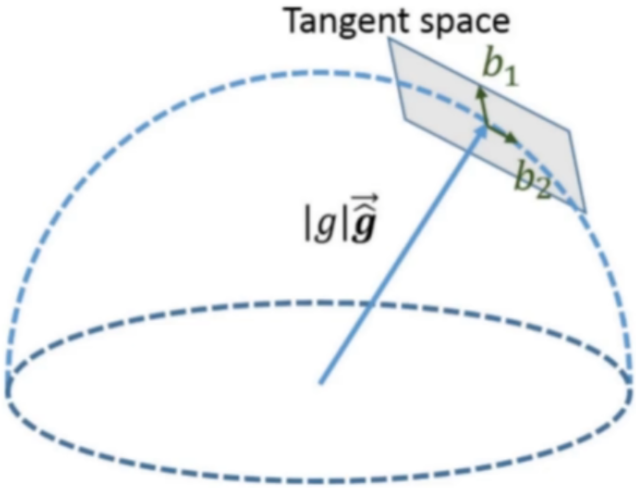
因为我们有N个v_k，就是写为下面的形式

$$\begin{pmatrix} & & \\ & & \\ & & \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ v_n \\ gs \end{pmatrix} = \begin{pmatrix} \\ \\ \end{pmatrix}$$

具体的方式就是“矩阵分块”，最终实现用一个大矩阵一次把所有的速度都求解出来。

我们通过解方程，求出每一帧的速度，和枢纽帧的重力方向和大小，以及尺度。这样我们就实现了IMU和重力的对齐。我们利用IMU加速度计的特性有了重力方向，我们有了每一帧的速度。
此时我们还需要，利用重力的先验知识，对刚才求解的数值进行修正。

重力方向矫正



大部分情况下，重力实际上是已经知道的，我们可以通过查表等方式是可以知道当地的重力加速度的。如果我们求解的实际的重力加速度和重力不同，那么肯定就有一些问题。
我们在VINS的初始化中，还需要进行调整。我们借助已知重力加速度大小9.81进行调整。我们取出计算出的重力的方向，然后对大小进行调整。
为什么我们不能直接设置重力呢？因为解方程的时候，我们把误差是分散出来的，所以我们需要的是“微调”。 我们调整重力的方向，大小是不改的。所以我们是在一个球面进行变化的（如论文中图片所示）。我们用b1和b2两个矢量的矢量和进行调整。
此时，借助已知重力的数值后，我们就把重力的3自由度，变为2个自由度了。因为b1和b2两个向量（都是单位向量而且相互正交，并且可以根据g的初始位置可以提前定义好），所以我们只要给b1和b2一个权重，就可以对重力的方向进行移动了。此时重力的自由度就从3变为2了。 （也可以这么理解，重力本身有x，y，z是3个自由度的，一旦这个重力的数字确定了，那么x、y、z的模长也就确定了。 那么我们只要知道x和y，就直接可以确定z了，所以是2自由度的）。

在代码里，会对重力方向进行check，已知一个大G（9.78），距离大G太远，那么不行。显然，尺度也必须是正数。
在代码中，利用重力大小进行结果修复的函数是RefineGravity。
一旦我们完成了视觉和惯性的对齐后，算法部分的工作就完成了。

6 后端优化

6.1 基于滑动窗口的非线性优化

非线性优化中，初始值非常重要。如果初始值和真值差的远的话，那么有可能陷入局部最优解，而得不到好用的解。VINS初始化成功后，就不会进行初始化了，主要就是靠后端了。
非线性优化中，vins主要依赖的google的cerse来进行求解。利用ceres自带的功能，定义核函数。但是初始化中，我们采用的是自动求导，但是在这里我们为了更快的速度，就没有采用自动求导。这就要求我们每一个残差对每一个变量的雅克比都得求出来，会比较麻烦，但是确实会更快。

ceres手动求导，需要继承一个虚函数，实现一个Evaluate的函数。下图是一个对ceres自动求导和解析求导的讨论。

	ceres自动求导	ceres解析求导
残差、雅克比是否自己计算?	只需要定义残差计算方式	既需要定义残差计算方式又需要定义残差对各个参数块的雅克比矩阵
costfuntion的继承方式	AutoDiffCostFunction	SizedCostFunction
如何提供残差（雅克比）	自定义一个重载了括号运算符的模板函数，用来提供残差的计算方式	重载Evaluate函数用来计算残差和雅克比

lossfunction是用于筛除outlier，残差特别大的个别变量，可能会把整个优化的结果影响特别大，而柯西核函数是一个常用的效果较好的鲁邦核函数，所以vins中也采用了这个核函数来让结果更好。

现在我们回忆论文里的公式25,
如果我们2帧之间的状态是完全通过IMU积分算出来的，那么我们公式25就是成立的。
但是我们的旋转和平移不仅仅是IMU得到的，还有我们重投影误差的一部分因素，所以这里的等于号不能成立。为了最小化重投影的误差，我们肯定会对这些变量进行调整的。
我们把预积分量作为一个约束就可以了。

$$\mathbf{r}_{\mathcal{B}}\left(\hat{\mathbf{z}}_{b_{k+1}}^{b_k}, \mathcal{X}\right) = \begin{bmatrix} \delta\boldsymbol{\alpha}_{b_{k+1}}^{b_k} \\ \delta\boldsymbol{\beta}_{b_{k+1}}^{b_k} \\ \delta\boldsymbol{\theta}_{b_{k+1}}^{b_k} \\ \delta\mathbf{b}_a \\ \delta\mathbf{b}_g \end{bmatrix} = \begin{bmatrix} \mathbf{R}_w^{b_k}\left(\mathbf{p}_{b_{k+1}}^w - \mathbf{p}_{b_k}^w + \frac{1}{2}\mathbf{g}^w\Delta t_k^2 - \mathbf{v}_{b_k}^w\Delta t_k\right) - \hat{\boldsymbol{\alpha}}_{b_{k+1}}^{b_k} \\ \mathbf{R}_w^{b_k}\left(\mathbf{v}_{b_{k+1}}^w + \mathbf{g}^w\Delta t_k - \mathbf{v}_{b_k}^w\right) - \hat{\boldsymbol{\beta}}_{b_{k+1}}^{b_k} \\ 2\left[\mathbf{q}_{b_k}^{w^{-1}} \otimes \mathbf{q}_{b_{k+1}}^w \otimes \left(\hat{\boldsymbol{\gamma}}_{b_{k+1}}^{b_k}\right)^{-1}\right]_{xyz} \\ \mathbf{b}_{ab_{k+1}} - \mathbf{b}_{ab_k} \\ \mathbf{b}_{wb_{k+1}} - \mathbf{b}_{wb_k} \end{bmatrix}$$

这些残差的定义，我们都需要在代码里定义好。ceres不像G2O，可以直接设置信息矩阵，我们最后维护的都是带权重的误差，所以代码里用的是LLT分解的方式来搞到信息矩阵。

6.2 雅克比矩阵求解

雅克比矩阵是什么样的形式呢？一般来说，是m维的y，对n维的x求雅克比矩阵。矩阵的每一个元素，正好对应y的某一个元素对x某一个元素求偏导数。最后的雅克比矩阵的大小一般是（m x n 的）。矩阵的第（i，j）个元素，就是y_i对x_j的偏导数。
行数就是多少个残差，多少个y就多少行。
回到预积分的问题里，预积分的残差一共15维对应y，x对应着参数块一共（7+15）* 2 个。代码里是分为4块雅克比（7+9+7+9的形式）。

我们回忆《十四讲》中，李代数的求导基本都是基于扰动的求导。十四讲是左扰动，但是我们这里是右扰动，要注意这个区别。如果求导的方式不记得，记得回忆十四讲的内容哦。

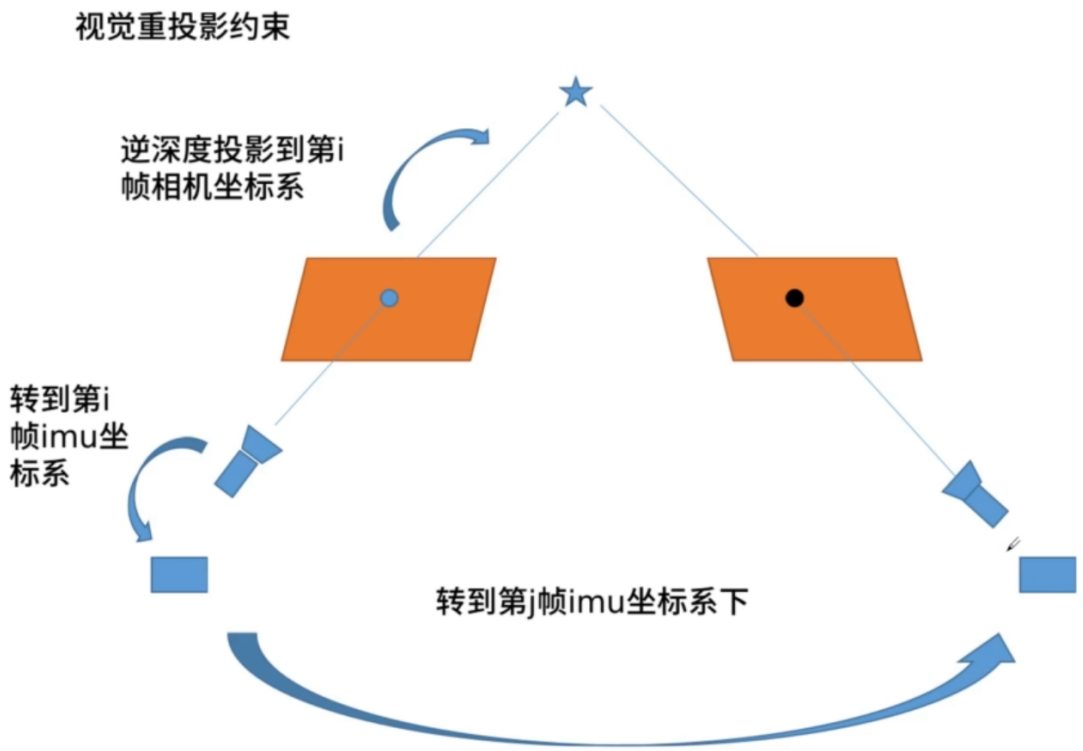
$$\delta\boldsymbol{\alpha}_{b_{k+1}}^{b_k} = \mathbf{R}_w^{b_k}\left(\mathbf{p}_{b_{k+1}}^w - \mathbf{p}_{b_k}^w + \frac{1}{2}\mathbf{g}^w\Delta t_k^2 - \mathbf{v}_{b_k}^w\Delta t_k\right) - \hat{\boldsymbol{\alpha}}_{b_{k+1}}^{b_k}$$

$$\delta\boldsymbol{\beta}_{b_{k+1}}^{b_k} = \mathbf{R}_w^{b_k}\left(\mathbf{v}_{b_{k+1}}^w + \mathbf{g}^w\Delta t_k - \mathbf{v}_{b_k}^w\right) - \hat{\boldsymbol{\beta}}_{b_{k+1}}^{b_k}$$

我们有上面两个式子，随后就是用预积分量对于k、k+1时刻的速度、零偏，位置，旋转的求导即可得到雅克比矩阵。

6.3 视觉重投影

VIO是视觉和惯性组成的系统。惯性是上面用IMU带来的约束，我们这一节主要讲的是视觉重投影的约束。视觉的重投影



地图点是用逆深度的方式和第一帧所绑定的。利用逆深度，就可以求出他在相机坐标系下3D点坐标的位置，随后通过外参的转换弄到世界坐标系，随后就可以求出这一帧在第i帧的3D点，在第j帧的归一化坐标系下投影的位置。当然，我们还知道本身通过光流也是知道相机坐标的，这个坐标的差异就构成了重投影的约束。
首先我们用逆深度，投影到第i帧的坐标系中。第i帧就是滑窗中，第一个观测到这个3D点的这一帧。也就是3D点逆深度所绑定的帧。因为我们保存的不是三维的量，而是一个一维的量，所以他不能独立保存，必须依赖某个帧来保存。如果不用逆深度的话，我们通常用xyz来保存。但是我们用一维来保存，可以加快我们的求解（维度只有原来的三分之一了）。另一方面我们用逆深度而不是深度的原因，有助于提升数值稳定性。
我们通过逆深度，把3D点投影到第i帧的相机坐标系，我们还需要通过外参转到第i帧的世界坐标系，再转到第j帧的IMU坐标系下。当然，我们还需要转到第j帧的相机坐标系下，然后还需要重新投影到第j帧的相机坐标平面上。这就需要一个3D到2D的转换，这样就可以得到一个投影点。

我们的优化的问题就是对投影点的约束建模，随后还需要对这个约束的误差求解雅克比矩阵。当然了，我们优化的东西还有相机到IMU的外参（这也就是所谓的紧耦合的一部分），所以我们把外参也加入到这个优化的过程中俩。别忘了我们还要优化我们的3D点的逆深度。所以我们一共有3种变量，2帧IMU之间的位姿态，IMU和相机的外参，3D点的逆深度。
用公式描述如下：

$$\begin{bmatrix} x_{c_j} \\ y_{c_j} \\ z_{c_j} \\ 1 \end{bmatrix} = \mathbf{T}_{bc}^{-1}\mathbf{T}_{wb_j}^{-1}\mathbf{T}_{wb_i}\mathbf{T}_{bc}\begin{bmatrix} \frac{1}{\lambda}y_{c_i} \\ \frac{1}{\lambda}v_{c_i} \\ \frac{1}{\lambda} \\ 1 \end{bmatrix}$$

$\frac{1}{\lambda}$就是逆深度了。
如果拆开旋转和平移，就得到

$$\begin{bmatrix} x_{c_j} \\ y_{c_j} \\ z_{c_j} \end{bmatrix} = \mathbf{R}_{bc}^{\top}\mathbf{R}_{wb_j}^{\top}\mathbf{R}_{wb_j}\mathbf{R}_{bc}P^{c_i} + \mathbf{R}_{bc}^{\top}\left(\mathbf{R}_{wb_j}^{\top}\left(\left(\mathbf{R}_{wb_i}\mathbf{p}_{bc}+\mathbf{p}_{wb_i}\right)-\mathbf{p}_{wb_j}\right)-\mathbf{p}_{bc}\right)$$

而残差，就是归一化后和光流的结果相减即可（就是视觉的重投影误差）

$$\mathbf{r}_c = \begin{bmatrix} \frac{x_{c_j}}{z_{c_j}} - u_{c_j} \\ \frac{y_{c_j}}{z_{c_j}} - v_{c_j} \end{bmatrix}$$

随后要计算的又是雅克比了。
我们实际上要求解的是残差对各个优化变量的求导。这里的如果直接求导十分复杂，所以采用链式求导的方式进行求解。

这里在代码里需要额外注意一下，作者写死了视觉重投影的视觉的置信度。

剩下的就是对如下这个公式的各个部分进行分别求导，从而可以得到雅克比矩阵。

6.4 边缘化

预积分和重投影的约束借住ceres来求解，优化和构建基本都是靠ceres解决。但是边缘化没有调用接口，操作不太复杂，但是没借住第三方库，都是自己实现的。
所谓边缘化，就是滑动窗口里的元素不能无限多。每当新的关键帧进来，就要有一个老的关键帧出去。暴力的把最旧的数据扔出去并不合理，因为可能最旧的关键帧还特别核心，我们还需要多多利用他的信息。
在SLAM里，是采用舒尔补的方式。后端中，大多采用解如下方程

$$H\sigma x = b$$
$$\begin{bmatrix} \Lambda_a & \Lambda_b \\ \Lambda_b^T & \Lambda_c \end{bmatrix} \begin{bmatrix} \delta x_a \\ \delta x_b \end{bmatrix} = \begin{bmatrix} g_a \\ g_b \end{bmatrix}$$

其中H是J^TJ,b是$-J^Te$,也就是误差.通过解方程求出δx并补偿到方程里，再进行下一轮的求解。我们对这个式子分解，考虑到H的特点一定是方正，且转置后等于本身。
δx_a就是即将被扔出滑窗的第0帧的位姿和状态， δx_b就是剩下的帧的位姿态和状态。舒尔布就是很简单的消元，《十四讲》里有讲。
公式花间可得

$$\begin{bmatrix} \Lambda_a & \Lambda_b \\ 0 & \Lambda_c - \Lambda_b^T \Lambda_a^{-1} \Lambda_b \end{bmatrix} \begin{bmatrix} \delta x_a \\ \delta x_b \end{bmatrix} = \begin{bmatrix} g_a \\ g_b - \Lambda_b^T \Lambda_a^{-1} g_a \end{bmatrix}$$

随后得到

$$\left(\Lambda_c - \Lambda_b^T \Lambda_a^{-1} \Lambda_b\right) \delta x_b = g_b - \Lambda_b^T \Lambda_a^{-1} g_a$$

此时公式已经没有δx_a了。但是实际计算过程中，还是利用了本身有δx_a带来的约束，只不过碰巧乘以0给化简掉了。

我们为了得到舒尔布的形式，我们首先要构造H矩阵。对每一个J^TJ累加，每个部分得到H的一部分，最后合起来就有了。
点此参考博客
总体来说，就是本身要被扔掉的第0帧，他可能和后面的帧有约束关系（一起看到某个地图点，和下一帧之间有IMU约束等），扔掉这个第0帧的数据的话，这些约束信息需要被保留下来。相当于建立起地图点和地图点之间的约束关系。

7 回环检测与地图的保存与载入

回环检测在vins中，采用了工程上常用的办法：词袋。
用词袋建立数据库，看关键帧中有没有在数据库中有比较“接近”的关键帧。一旦认为回环检测成功后，我们就进行一次4自由度的位姿图优化，因为我们加IMU之后，由于重力可观性，他的rolling和pitch角是全局可观的。也就是说，只有yaw和xyz不可观，所以进行4自由度的位姿图优化。

这里还涉及地图的保存和加载，也就是可以让事先建立的地图加载进来，基于之前的视觉地图，进行下一步的定位和回环检测等功能。回环检测的部分，和之前的部分相比，公式较少。这个部分有一个特别的地方，坐标变换比较多一些而已。

回环检测中，主要是2种坐标变换的概念。
一种是VIO的位姿态，一种是最终位姿。
考虑到VIO给回环检测提供当前的位姿，但是回环检测后的结果并不会改变里程计的信息。里程计的特点是平滑的，不容易突变。通过回环更新当前的位姿后，继续用VIO作为当前位姿进行输出。

7.1 代码解读

这部分代码主要是pose_graph_node.cpp开始。
一开始一部分帧是不做回环检测的，考虑到回环检测比较慢，所以期望2帧位移较大再进行回环检测。
这里不同于orb-slam，vins把词袋用二进制保存了，来提升读取词袋的速度。
用长度64-128-256的01来描述一个点，再orb-slam里是把描述子信息放在代码里。vins则是单独用一个文件保存。

核心的回环线程是在process函数实现的。
我们使用的核心数据是kf的位姿、原图、地图点、像素坐标。通过不断的降采样（减少数据数量），来降低进行回环检测的频率（回环检测费时间，而且不需要太高频率进行）。通过遍历像素坐标，对这些坐标的描述子用bdow的接口计算出来。
考虑到本身特征略少，所以作者又多提了一些特征子。考虑到之前的特征信息，这时候提的特征子还没有放到归一化相机坐标系下的坐标，这里新提的特征也需要归一化一下。

具体的“回环检测”的检测部分，是在posegraph::detectLoop实现的。采用dbow的接口进行查询回环。查找4个备选值。查询完后，把我们用于查询的描述信息也加进dbow的数据库。

这里会找到很多回环帧，但是可能有误检，所以作者用一些阈值和策略卡出候选帧，主要是希望索引尽可能小，得分尽可能高。

考虑到还是可能有错检测，我们要进行回环校验。我们通过findConnection来完成。我们用当前帧和候选的帧，计算描述子之间的的汉明距离（描述2个描述子的相似度），找到得分最好的描述子（小于80），来进行匹配。
采用了暴力的搜索方式所以比较慢。

校验的方式用PNP的方式进行几何校验。判断PNP的内点数量是否大于25个来判断是不是有效的回环。对结果的旋转和平移用阈值卡一下，进行校验，都得小于阈值才算有效的回环。

如果检测回环成功的话，我们有2个当前帧的位姿，一个是前端VIO，一个是回环得到的位姿。这两个位姿会有一个位姿差。我们分他们是否在一个sequence里讨论。如果不在一个sequence里，就把他们进行合并。随后把序列当前帧之前的位姿都更新到同一个坐标系下，就实现了合并。

在接受回环信息后，我们需要先判断回环信息还是否在滑窗中。我们需要优化当前坐标系下回环帧的位姿。随后当前帧和回环帧一般比较接近，把当前帧的位姿作为初始位姿。
首先遍历特征点，判断特征点是否有效。对特征点的起始点的id取出来，这个起始点应该**小于等于**当前帧在滑窗的id。构建一个重投影约束，他约束着的地图点起始帧和回环帧之间的关系。把约束加到ceres中，优化的时候依旧要约束逆深度等变量，和之前的优化方法基本一致。

求解完优化问题后，我们回环帧在当前VIO坐标系下的已经被调整到一个比较好的位姿状态了。同样的，我们也会调整固定滑窗第一帧的yaw和xyz，不可观的4自由度也会被固定下来。

我们计算完这个优化后的回环帧和当前帧的相对位姿态，求出准确的回环帧的位姿。此时也就可以知道准确的当前帧在VIO和回环检测情况的位姿的变化量了。

我们要优化的是最早的回环帧到最新的当前帧之间的位姿。在更早的位姿态就不会优化了，降低要优化的数据量。
考虑到位姿比较多，参数这里设置了最大只迭代优化5次。考虑到误匹配的问题，这里依旧用huber核函数。这里还要注意，类似于对四元数的约束，这里需要约束yaw的取值范围是0-2pi范围内。
为了避免帧间飘移，我们这里固定了最早的回环帧以及加载的地图都保持不变。注意，这里每个帧都和周围相邻的若干帧都存在约束。
四自由度全局优化的时候，和初始化一样采用了自动求导，没有用解析求导。

vins视觉的地图的组成，是用关键帧位基本单元，存储关键帧的位姿、特征点、描述信息来保存的。

在vins的pose graph的节点里，还有视觉地图的保存和加载的功能。他通过command的线程，去捕捉你输入s或者n来判断保存地图还是新建一个地图。在vins里sequence的含义就是一个新地图。每当我们新建一个sequence的话，还会检查不得超过5个。
我们要操作keyframelist的时候，要提前加锁。我们遍历这些keyframe作为地图保存起来，保存他们的索引、时间戳、posegraph偏移等信息。其中，旋转用四元数的形式来保存。

vins提供地图加载功能，类似的把保存的信息全部都加载过来。

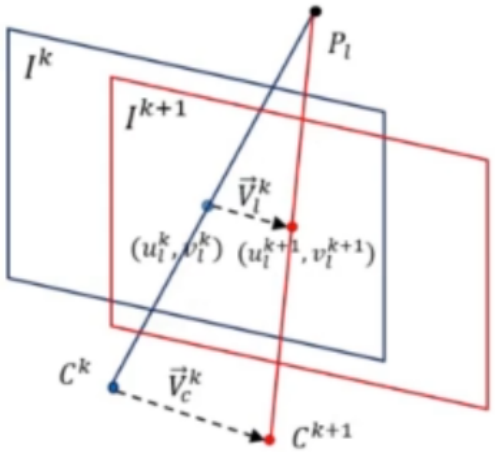
8 传感器时延估计

本章是vins里时间延时的优化相关的内容。在vins中，我们不仅仅优化位姿，3D逆深度，传感器之间外参信息，还会优化传感器之间的时间延时。考虑到kitti数据集中，时间同步都是做好的。但是现实的情况中，多传感器的方案中往往会有外参标定和时间延时的问题。我们的数据的时间戳，很多时候无法得到正确时间戳的保证。比如说，第12秒的数据，我们可能第12.1秒才收到数据，这就会带来一些麻烦。vins中采用了比较好的方法来解决了时间问题。
不同传感器的时间延时对系统的鲁棒和精确性是一个重要的因素。不同传感器时间戳受到触发和传输的延时，导致时间戳无法对齐。通过把时间延时作为优化变量，加入到滑窗中，整体的形成slam系统，从而得出时间延时。
注意这里作者做了一个很强的假设：时间延时是固定的。也就是传感器的时间戳时间差是固定的数值。

IMU和相机很多时候没有进行严格的硬件同步（在工业级的应用中，往往做了严格的同步）。所以很多时候得到数据的时间戳，并不是数据真实的时间戳。所以相机和IMU之间的延时，可以用一个数字表示。也就是说，最后我们用到时间戳的时候，可以简单的对时间戳加一个数字就得到“对齐”的时间戳了。在这里，作者假设了IMU的时间戳是固定的，我们对相机的时间戳进行加减即可。

特征点的速度可以被用来估计时间延时。把相机和IMU的时间序列一起移动的话会比较麻烦，因为没办法对图像进行插值。因为2张图片之间插值会特别麻烦，所以对图片进行时间补偿比较不现实。但是对IMU进行插值的话，也会带来巨大的计算量。（插值会带来预积分数据的重新计算）

所以对图片和预积分重新计算都不好，所以作者采用了移动特征点的方法来解决这些问题。作者做了一个假设，在短时间内，相机的运动是匀速运动。基于假设，计算特征点在相机平面上的运动。利用不同时刻，在相机上的投影，则可以很轻松的算出在中间某个时刻特征点的坐标（假设了特征点的速度也是匀速运动）。



如果我们特征点是用3D坐标表示的话，就直接利用特征点投影的坐标，和特征点速度，以及补偿时间，来算出对应时刻特征点在图像上的坐标。

如果我们的数据是用逆深度保存的，我们需要至少2帧数据才能形成约束。把绑定了深度信息的像素点，投影到其他帧上去，之后再利用特征点速度和补偿时间，求解出补偿时间后的特征点坐标。

有了上述建模后，就是优化求出最优的时间估计了。类似于vins中的约束方程，只不过需要修改一下视觉的约束，添加时间补偿的结果即可。

9 vins-fusion

vins-fusion是vins-mono的升级版本。是一个基于优化框架的多传感器的优化算法。vins mono是单目+IMU的传感器组合，是能够恢复出6自由度位姿的传感器组件。（激光雷达也可以，但是有一些缺陷）

vins-fusion额外支持双目+IMU，只用双目，单目+IMU以及对GPS信息的使用。

编译运行vins-fusion中，依旧是采用catkin_make来编译，基本和vins-mono一致。

但是有可能在运行的时候崩溃！考虑到ROS崩溃大多数情况下是看不到有效信息的。如何知道在哪里崩溃呢？通常的做法是使用gdb，他是一个在linux下调试的工具。把gdb用在ROS中的话，需要一个launch文件，通过增加参数来解决。
我们本身是用rosrun来启动节点的，但是我们用gbd启动就必须用launch。
通过在launch中添加launch-prefix="xterm -e gdb -ex run --args"
随后一般可以在gdb中看到崩溃信息。
一般堆栈信息里可以看到有效的崩溃信息，vins-fusion可能会在边缘化的代码里导致的崩溃。
考虑到Eigen里的报错，都比较麻烦。这通常和本地环境关系比较大。为了避免环境问题，有一个解决的方法是“docker”。 docker是一个虚拟机，他可以把代码运行环境共享给大家，避免环境不同带来的运行问题。

考虑到数据集不同，会有不同的数据入口，vins-fusion有多个主函数入口。
这里有一个特别不一样的地方，叫“multiple-thread”的参数，他的作用是多线程模式的开启。他主要是一个线程一个跑光流，一个跑优化。这样他就和之前不一样，就实现了“流水线”，增加了运行效率。主要这考虑到有的数据是播放bag，是实时的，2帧图片时间间隔是固定的，所以用多线程来增加运行效率。还有数据集比如是kitti，他是固定的顺序图片的，一帧算久一点也没关系。

对于双目相机而言，最重要的是时间同步。我们期望起码双目是同时拍摄的，一般来说都是用硬件对两个相机同步。如果没办法硬件同步，那就只能用软件进行同步了。vins fusion里假设2个相机数据在3ms以内就算同一时间的数据。

9.1 双向光流

对于对于光流的追踪问题，在vins fusion中略有改进。之前的做法是用金字塔的方式来解决初始值的问题，fusion中采用预测的方式来加快搜索。如果预测的好，就只要用2层金字塔就可以了。如果追踪的结果比较少（只追了10个点），那么重新用4层金字塔来做。大家可以测试一下，用2层失败的频率是多少。如果没能产生较好的先验，就直接用4层金字塔来追踪。他采用了双向光流的方式来check光流追踪的是否优秀。主要思想是把追踪后的结果，反过来去追踪原来图像的点。 如果两次追踪都成功了，那么如果像素差小于0.5个像素，就认为是好的追踪结果。其他的相关光流操作基本和mono一致。

考虑得到双目模式，在光流追踪中有一些额外的工作。我们采用左目相机（主相机）去找右相机的匹配关系。这里依旧采用光流的方式来完成匹配。考虑到2个相机是有一个基线，这里还是采用4层金字塔的追踪。这里也一样可以用双向光流来提升光流检测的鲁棒性。

9.2 初始化

不太同于mono里，对于第一帧IMU，我们要根据IMU的重力方向，来估计一个大概的姿态。考虑到roll和pitch可观，那么用重力方向来获得他们的姿态角。代码里，求解加速度的平均值。我们把第一帧的yaw角置0，随后roll和pitch通过加速度计的均值来算。加速度计大多数分量来自于重力9.8，只要运动没有太激烈，一般来说roll和pitch都和真值比较接近。vins-mono里的初始化则更加复杂一些。

双目和IMU以及纯双目的初始化的好处，是可以直接用双目恢复出特征点深度，从而得到真实的尺度。所以不需要像单目那样费力的恢复尺度。为了做PNP，我们首先要有一些特征点的信息。我们肯定没办法直接获得，所以通过三角化的方式获得。
三角化主要是把双目都能看到的特征点，用2个相机外参即可恢复出3D信息。通过左右目相机的位姿，就可以直接算出3d点的位置了。如果在双目中，有些特征点只能被一个相机看到（那也就没法三角化了），那么恢复3D点位姿的话，就只能通过连续帧了。
如果想进行运动估计，肯定需要2帧及以上的数据才行。取出起始帧位姿，和第二帧，把前一阵当左目，把后一帧当右目来恢复出3D点位姿。

通过双目相机恢复3D信息，根据PNP恢复后面帧的位姿，并三角化出更多3D点信息，最后就可以填满滑动窗口。滑窗填满后，就可以进行陀螺仪的初始化了和求解零偏等信息，求解的方式和vins-mono一样。
显然，当时双目模型的话（没有IMU），那么关于IMU零偏等信息都不需要优化计算了。

9.3 vins-fusion 后端优化

大多数部分和mono部分都是一样的。如果没有IMU的情况，也就是双目的情况，反而不需要进行IMU相关的数据的估计，比如零偏的估计。我们需要把滑窗里的第一帧给固定住，考虑到单目VIO不能观的自由度是4，用IMU的加速度计（利用重力加速度的信息）就可以估计出pitch和roll角。如果没有IMU，那么pitch和roll都不能观，这样的话我们就在6自由度里自由发挥。考虑到所有的约束都是帧间约束，也就是他们之间都是相对关系，不是绝对关系。避免优化的时候把所有的位姿都“串了”， 所以要固定第一帧的位姿。
这里有一个和mono不一样的改进，只在有运动激励的情况下，才会更新外参。当滑窗被填满，滑窗第0帧速度的绝对值大于0.2米每秒才会开始估计外参。
是否估计延时的参数，同样是考虑在有足够运动激励的情况下才会进行估计。
考虑到双目就没有预积分约束，但是重投影约束一定是有的。

不完全同于mono里的残差计算，这里有了双目的模型，所以有右相机的投影残差计算。
如果要约束的第i和第j不是同一帧。 第i帧通常是左目的观测，和第j帧的左右目会产生约束。对于同样是第j帧的左目，其约束和vins-mono是一样的。和右目的约束则略有不同，其推导主要是坐标系的转化。首先我们需要把第i帧相机坐标系下的点，转到第j帧右目相机的坐标系下，就可以进行投影了。随后利用投影结果和"提点"的结果进行残差计算。
雅可比的计算，就是残差对每一个变量的推导，依旧采用链式求导的方式即可得出最后的雅可比矩阵。
如何要约束的i和j是同一帧，这种情况是无法对位姿形成约束，但是可以对外参和特征点深度形成约束。两帧之间双目相机，通过三角化可以算出深度；维护的相机之间的点，所以IMU外参也可以优化。计算残差的话，也是先用时间延时进行补偿，随后算出逆深度（就相当于求出了相机坐标系下的点）。随后点在各种坐标系下的坐标都能求出来,这样就可以轻松的求出在右目相机的投影，从而计算出残差。类似的，用链式求导求出相关的雅可比即可。

9.4 外点剔除

光流追踪的点比较多，虽然我们有一些筛选机制，但是还是可能会有一些点追错了。只考虑前端的筛选，还是不太够，所以在后端中依旧做了一些筛选剔除机制。
首先我们只对观测数量**大于等于4**的点进行判断，（观测太少的话就不用管了）。

对于要进行判断的特征点，我们把滑窗中第一个观测到该点的id叫做起始帧，我们用IMU_I表示。如果后面的IMU_J和IMU_I不同，则会进行重投影的计算。
对于单目的情况，利用2帧之间的旋转平移关系，则可以进行投影计算。用重投影的误差，和光流的结果进行对比得到误差，累加所有误差。对于双目的情况，还要计算右目相机的重投影误差。最后根据平均误差恢复到像素上（乘上焦距）超过3个像素，则认为是outlier，剔

除当前点。剔除的方法，就把这个点的id用set保存起来。

删除外点的操作，就是遍历特征点，看看他们是否在我们刚刚保存的set里，在的话就给他删掉。

在跑离线数据（不是rosbag）的情况下，vins-fusion增加了预测功能。我们前文介绍了有先验采用2层金字塔，没有先验采用4层金字塔。预测功能是在esitimator.cpp的 predictPtsInNextFram中实现的。这里的技巧非常巧妙。我们保证滑窗中有2帧的情况下，我们采用匀速模型。利用上一帧到当前帧的位姿变换（假设速度不变），预测下一帧的位置。我们遍历所有特征点，然后判断这些特征点在下一帧是否是有效观测（这个特征点起码在滑窗中最后一帧也被追踪到），随后取出特征点的深度进行投影，这样就得到了预测的像素的结果。

9.5 GPS融合

回环检测部分，在fusion中并没有IMU相关的功能，依旧是基于磁带的功能。全局融合中，采用了和GPS进行融合的方式来提升系统的性能。
相关代码里，主要订阅了VIO和GPS的定位信息，
考虑到VIO是四自由度不可观的。简单的说，我们没有任何信息可以约束我们系统绝对的位置。（相同的场景，我们把整个系统移动到地球另一边，我们会运行出一摸一样的结果。但是我们并不能通过VIO系统知道我们已经到地球了另一边了。我们只能通过重力方向，知道pitch和roll，其他信息都是没办法知道的。）
考虑到我们只有GPS，只能得到xyz信息给出约束。
我们先把VIO坐标转换到GPS提供的全局坐标系的转换。随后，我们要找的就是进行GPS信息对齐（此时没有任何时间补偿算法），利用10ms的阈值进行数据同步。
GPS的数据是经纬度，但是我们VIO的单位通常是米，他们是不一样的度量单位，所以要进行数据转换，把经纬度转化到笛卡尔坐标系。作者这里采用了一个开源坐标系转化的库来进行转化。我们把第一帧经纬度作为坐标系原点，后面就可以知道坐标和第一帧经纬度的偏移结果，就可以推算出后面的经纬度的xyz的坐标了。

这里我们构建优化模型，建立帧间约束和GPS带来的“绝对”的约束。帧间约束是VIO位姿带来的，GPS的绝对约束则是约束一些有GPS信息的位姿的具体的xyz信息。
当优化结束了，除了需要更新VIO的位姿信息外，还需要更新GPS到VIO坐标系的转换关系。我们利用最后一个有GPS信号的VIO位姿，和GPS的的xyz信息的，来计算他们之间的转换关系。

这里看出，GPS主要给slam系统提供了绝对位姿估计的信息。