Vins-Fusion的阅读笔记（细节篇）

细节篇主要分析vins fusion的紧耦合优化细节。

推导参考博客：https://www.cnblogs.com/buxiaoyi/p/8660854.html

1，前言

细节篇对数学要求很高。但是我不会latex写公式，所以尽量用语言描述吧。细节篇将就基本篇末尾提出的七个问题做深入的研究。先回顾一下紧耦合的优化变量，分成以下三块：

1，各个时刻下imu的位置，速度，姿态，加速度bias，陀螺仪bias

2，imu和camera的位置，姿态

3，滑窗内所有特征点相对于滑窗第一帧的深度值之倒数

另外还会发现，四元数经常用来表达旋转矩阵，而不是用李群李代数。不直接用旋转矩阵表述的原因很明显，对旋转矩阵求导不变，旋转矩阵自包含约束，优化时候自动变为带约束的优化问题。不用李群李代数，原因我不知道，推测是不想引入exp映射。因为四元数自身既可变成一个四乘四的矩阵，又可以变成一个四乘一的向量，变来变去，都是线性代数问题，解决起来方便。如果引入李群，必须要处理指数映射，会不会麻烦些呢？不得而知，仅是猜测。除此之外，旋转矩阵也可以表示成Gibbs，但是这样带一个矩阵求逆，公式也不太好变形。

四元数还有一个好处，就是这个列向量不仅仅可以表示旋转，还能表示“旋转+平移”。

2，IMU误差概述

这里要分析的是imu_factor文件中的代码。在分析之前，找一个博客，把imu误差jacobian推导仔细看一遍，会发现没有想象中的那么难。从误差公式（21）可以看到，误差函数关系关乎两个时刻的优化变量，即第k时刻和第k+1时刻。并且会发现优化变量只包含“各个时刻下imu的位置，速度，姿态，加速度bias，陀螺仪bias”。

jacobian矩阵分成四块：

1，imu误差对第k时刻imu的位置以及姿态求偏导，15乘以7矩阵

2，imu误差对第k时刻imu的速度，加速度bias，陀螺仪bias求偏导，15乘以9矩阵

3，imu误差对第k+1时刻imu的位置以及姿态求偏导，15乘以7矩阵

4，imu误差对第k+1时刻imu的速度，加速度bias，陀螺仪bias求偏导，15乘以9矩阵

（啰嗦一句，imu误差对bias求导，就需要之前预积分时候计算的jacobian。一开始学习预积分，最不明白的是为什么要求jacobian，那是因为通过求jacobian进而更新imu的bias，而不是仅仅求个jacobian本身！往下也会看到，之前求的协方差矩阵也是有用的！）

（再啰嗦一句，求导是扰动求导，对delta X求导，而不是对X求导！）

代码的框架大概是这样的：

```
class IMUFactor : public ceres::SizedCostFunction<15, 7, 9, 7, 9>
{
  public:
    IMUFactor() = delete;
    IMUFactor(IntegrationBase* _pre_integration):pre_integration(_pre_integration)
    {
    }
    virtual bool Evaluate(double const *const *parameters, double *residuals, double
**jacobians) const
    {
        // process ...
    }
```

这一串数字<15, 7, 9, 7, 9>是啥意思？指有四个jacobian矩阵，分别是15乘以7，15乘以9，15乘以7，15乘以9。从代码知道，类IMUFactor是public继承自SizedCostFunction，内部包含引用了类IntegrationBase。

该函数的重点是Evaluate，但注意Evaluate的变量，分别是parameters，residuals，jacobians。为啥是双指针，暂时还不知道。该函数分成以下几个部分执行：

1，变量初始化，名称都能看出来

```
Eigen::Vector3d Pi(parameters[0][0], parameters[0][1], parameters[0][2]);
Eigen::Quaterniond Qi(parameters[0][6], parameters[0][3], parameters[0][4], parameters[0][5]);
Eigen::Vector3d Vi(parameters[1][0], parameters[1][1], parameters[1][2]);
Eigen::Vector3d Bai(parameters[1][3], parameters[1][4], parameters[1][5]);
Eigen::Vector3d Bgi(parameters[1][6], parameters[1][7], parameters[1][8]);

Eigen::Vector3d Pj(parameters[2][0], parameters[2][1], parameters[2][2]);
Eigen::Quaterniond Qj(parameters[2][6], parameters[2][3], parameters[2][4], parameters[2][5]);
Eigen::Vector3d Vj(parameters[3][0], parameters[3][1], parameters[3][2]);
Eigen::Vector3d Baj(parameters[3][3], parameters[3][4], parameters[3][5]);
Eigen::Vector3d Bgj(parameters[3][6], parameters[3][7], parameters[3][8]);
```

2，计算imu误差

```
Eigen::Map<Eigen::Matrix<double, 15, 1>> residual(residuals);
residual = pre_integration->evaluate(Pi, Qi, Vi, Bai, Bgi,
                                     Pj, Qj, Vj, Baj, Bgj);

Eigen::Matrix<double, 15, 15> sqrt_info = Eigen::LLT<Eigen::Matrix<double, 15, 15>>
(pre_integration->covariance.inverse()).matrixL().transpose();
//sqrt_info.setIdentity();
residual = sqrt_info * residual;
```

很有意思的是，这里使用了covariance矩阵。covariance矩阵在预积分的时候也求了哦。顺便看一下类IntegrationBase的evaluate函数。

先看定义式：

```
Eigen::Matrix<double, 15, 1> evaluate(const Eigen::Vector3d &Pi, const Eigen::Quaterniond &Qi,
const Eigen::Vector3d &Vi, const Eigen::Vector3d &Bai, const Eigen::Vector3d &Bgi,
const Eigen::Vector3d &Pj, const Eigen::Quaterniond &Qj, const Eigen::Vector3d &Vj, const
Eigen::Vector3d &Baj, const Eigen::Vector3d &Bgj)
```

可以看到代码的误差函数跟论文是一样的：

```
Eigen::Matrix<double, 15, 1> residuals;
Eigen::Matrix3d dp_dba = jacobian.block<3, 3>(O_P, O_BA);
Eigen::Matrix3d dp_dbg = jacobian.block<3, 3>(O_P, O_BG);
Eigen::Matrix3d dq_dbg = jacobian.block<3, 3>(O_R, O_BG);
Eigen::Matrix3d dv_dba = jacobian.block<3, 3>(O_V, O_BA);
Eigen::Matrix3d dv_dbg = jacobian.block<3, 3>(O_V, O_BG);
Eigen::Vector3d dba = Bai - linearized_ba;
Eigen::Vector3d dbg = Bgi - linearized_bg;

Eigen::Quaterniond corrected_delta_q = delta_q * Utility::deltaQ(dq_dbg * dbg);
Eigen::Vector3d corrected_delta_v = delta_v + dv_dba * dba + dv_dbg * dbg;
Eigen::Vector3d corrected_delta_p = delta_p + dp_dba * dba + dp_dbg * dbg;

residuals.block<3, 1>(O_P, 0) = Qi.inverse() * (0.5 * G * sum_dt * sum_dt + Pj - Pi - Vi *
sum_dt) - corrected_delta_p;
residuals.block<3, 1>(O_R, 0) = 2 * (corrected_delta_q.inverse() * (Qi.inverse() * Qj)).vec();
residuals.block<3, 1>(O_V, 0) = Qi.inverse() * (G * sum_dt + Vj - Vi) - corrected_delta_v;
residuals.block<3, 1>(O_BA, 0) = Baj - Bai;
residuals.block<3, 1>(O_BG, 0) = Bgj - Bgi;
return residuals;
```

之前讲过，求导是扰动求导，扰动求导！上述代码中以“d”开头的变量都是扰动变量delta哦！留意一下correct变量，应该可以理解吧，dp_dba是扰动对扰动求导。

3，计算jacobian

这部分可以细致分为几块内容：

3.a，准备工作，一些扰动量求导，以及报错

```
if (jacobians)
{
   double sum_dt = pre_integration->sum_dt;
   Eigen::Matrix3d dp_dba = pre_integration->jacobian.template block<3, 3>(O_P, O_BA);
   Eigen::Matrix3d dp_dbg = pre_integration->jacobian.template block<3, 3>(O_P, O_BG);
   Eigen::Matrix3d dq_dbg = pre_integration->jacobian.template block<3, 3>(O_R, O_BG);
   Eigen::Matrix3d dv_dba = pre_integration->jacobian.template block<3, 3>(O_V, O_BA);
   Eigen::Matrix3d dv_dbg = pre_integration->jacobian.template block<3, 3>(O_V, O_BG);
   if (pre_integration->jacobian.maxCoeff() > 1e8 || pre_integration->jacobian.minCoeff() <
-1e8)
   {
       ROS_WARN("numerical unstable in preintegration");
   }

   // process ...
```

```
}
```

3.b 求第一个jacobian，imu误差对第k时刻imu的位置以及姿态求偏导

```
if (jacobians[0])
{
   Eigen::Map<Eigen::Matrix<double, 15, 7, Eigen::RowMajor>> jacobian_pose_i(jacobians[0]);
   jacobian_pose_i.setZero();
   jacobian_pose_i.block<3, 3>(O_P, O_P) = -Qi.inverse().toRotationMatrix();
   jacobian_pose_i.block<3, 3>(O_P, O_R) = Utility::skewSymmetric(Qi.inverse() * (0.5 * G *
sum_dt * sum_dt + Pj - Pi - Vi * sum_dt));

    // 因为gamma这个量包含扰动量，而jacobian包含gamma，所以没办法，需要把带扰动的gamma写出来
    // 即jacobian_pose_i.block<3, 3>(O_R, O_R)这一部分
   Eigen::Quaterniond corrected_delta_q = pre_integration->delta_q * Utility::deltaQ(dq_dbg *
(Bgi - pre_integration->linearized_bg));

   jacobian_pose_i.block<3, 3>(O_R, O_R) = -(Utility::Qleft(Qj.inverse() * Qi) *
Utility::Qright(corrected_delta_q)).bottomRightCorner<3, 3>();
   jacobian_pose_i.block<3, 3>(O_V, O_R) = Utility::skewSymmetric(Qi.inverse() * (G * sum_dt +
Vj - Vi));
   jacobian_pose_i = sqrt_info * jacobian_pose_i;

   if (jacobian_pose_i.maxCoeff() > 1e8 || jacobian_pose_i.minCoeff() < -1e8)
   {
      ROS_WARN("numerical unstable in preintegration");
   }
}
```

首先可以看到，jacobian矩阵求出来之后，用sqrt信息矩阵做了一次白化处理。基本上和博客推导结果一样。

3.c 求第二个jacobian，imu误差对第k时刻imu的速度，加速度bias，陀螺仪bias求偏导

```
if (jacobians[1])
{
   // 注意这个赋值语句，jacobian_speedbias_i和jacobians[1]的关系
   Eigen::Map<Eigen::Matrix<double, 15, 9, Eigen::RowMajor>> jacobian_speedbias_i(jacobians[1]);
   jacobian_speedbias_i.setZero();
   jacobian_speedbias_i.block<3, 3>(O_P, O_V - O_V) = -Qi.inverse().toRotationMatrix() * sum_dt;
   jacobian_speedbias_i.block<3, 3>(O_P, O_BA - O_V) = -dp_dba;
   jacobian_speedbias_i.block<3, 3>(O_P, O_BG - O_V) = -dp_dbg;
   jacobian_speedbias_i.block<3, 3>(O_R, O_BG - O_V) = -Utility::Qleft(Qj.inverse() * Qi *
pre_integration->delta_q).bottomRightCorner<3, 3>() * dq_dbg;
   jacobian_speedbias_i.block<3, 3>(O_V, O_V - O_V) = -Qi.inverse().toRotationMatrix();
   jacobian_speedbias_i.block<3, 3>(O_V, O_BA - O_V) = -dv_dba;
   jacobian_speedbias_i.block<3, 3>(O_V, O_BG - O_V) = -dv_dbg;
   jacobian_speedbias_i.block<3, 3>(O_BA, O_BA - O_V) = -Eigen::Matrix3d::Identity();
   jacobian_speedbias_i.block<3, 3>(O_BG, O_BG - O_V) = -Eigen::Matrix3d::Identity();
   jacobian_speedbias_i = sqrt_info * jacobian_speedbias_i;
}
```

和博客的推导结果一样。（当你不会推导jacobian，有没有博客借鉴的时候，可以看代码里的jacobian，试着反向推导）

3.d 求第三个jacobian，imu误差对第k+1时刻imu的位置以及姿态求偏导

```
if (jacobians[2])
{
   Eigen::Map<Eigen::Matrix<double, 15, 7, Eigen::RowMajor>> jacobian_pose_j(jacobians[2]);
   jacobian_pose_j.setZero();
   jacobian_pose_j.block<3, 3>(O_P, O_P) = Qi.inverse().toRotationMatrix();
   // 有correct，原因之前讲了。
   Eigen::Quaterniond corrected_delta_q = pre_integration->delta_q * Utility::deltaQ(dq_dbg *
(Bgi - pre_integration->linearized_bg));
   jacobian_pose_j.block<3, 3>(O_R, O_R) = Utility::Qleft(corrected_delta_q.inverse() *
Qi.inverse() * Qj).bottomRightCorner<3, 3>();
   jacobian_pose_j = sqrt_info * jacobian_pose_j;
}
```

3.e 求第四个jacobian，imu误差对第k+1时刻imu的速度，加速度bias，陀螺仪bias求偏导

```
if (jacobians[3])
{
   Eigen::Map<Eigen::Matrix<double, 15, 9, Eigen::RowMajor>> jacobian_speedbias_j(jacobians[3]);
   jacobian_speedbias_j.setZero();
   jacobian_speedbias_j.block<3, 3>(O_V, O_V - O_V) = Qi.inverse().toRotationMatrix();
   jacobian_speedbias_j.block<3, 3>(O_BA, O_BA - O_V) = Eigen::Matrix3d::Identity();
   jacobian_speedbias_j.block<3, 3>(O_BG, O_BG - O_V) = Eigen::Matrix3d::Identity();
   jacobian_speedbias_j = sqrt_info * jacobian_speedbias_j;
}
```

和博客推导是一致的。

3，构建因子（factor）三要素

至此分析完了imu factor（先不着急分析下一个factor），对内部的机理有了一定的了解。归纳一下，可以发现有意思的事情，那就是一个factor究竟包含了什么，以下是构建因子三要素：

1，误差的定义，error function

2，误差对优化变量求偏导，jacobian

3，误差的协方差矩阵

不得不思考几个问题：

为什么误差会有协方差矩阵呢？那是因为我们建立观测模型的时候，引入了高斯分布的噪声，当然，引入的噪声协方差可能是单位阵，但是推导误差向量的时候，难免会乘以一些矩阵，这样误差的噪声的协方差矩阵不是单位阵了。

如果我们不考虑协方差，这就是一个最小二乘问题，就不是factor了。

难么，自然会问，factor是什么，为什么叫factor？factor是因子的意思，关联因子图和概率图（这两个以及马尔科夫链，可以相互转化），可以看一下《因子图slam》这本书。我们把机器人运动问题，当作一个时间序列问题，这个时间序列问题又归为一个概率图模型，这样就会有一个总分布。为什么会有分布呢，因为我们假定所有的传感器的模型都是高斯模型（如果不是，又该咋办呢？）。通过最大后验概率，可以得到一个可信度高的，关于机器人运动轨迹的结果（也称为概率图推断）。

然后，怎样从概率图，最大后验，求解相机不同时刻位姿呢？因为我们是高斯建模，对总分布求对数，问题就变成一个加权最小二乘问题。优化问题就变成一个形如Ax=b的问题。然而x的维数很高，为了限制x的维数，所以采用了“滑窗优化”，进而限制x的维数。并且求解的时候使用了舒尔补。舒尔补在统计上也是有直观意义的，代数上也是有意义的（即消元）。另外，Ax=b消元方法还有很多，舒尔补并不是最高效的，但毫无疑问是最简单的（可以参见书《因子图slam》）。

求解因子图（factor graph）可以用专门的第三方库gtsam。其实，我们用ceres，或者g2o，只要把上述的factor三要素定义好，就可以直接求解因子图优化问题了。图优化不再神秘！

4，初步了解projection factor

之前讲了factor的三要素。所以首要分析projection factor的误差函数，这里误差函数并不是重投影误差，如下：

a，projection factor的误差（二乘一向量） = 切平面法向量的转置（二乘三向量） * 归一化物理坐标误差（三乘一向量）

b，相对于当前相机坐标系下的归一化物理坐标 = norm（像素的反投影误差）

c，相对于当前相机坐标系下的实际物理坐标 = 深度信息 * 归一化物理坐标

再理解式子（22）就不难了，从第i时刻的相机坐标系到第j时刻的相机坐标系之间的转换比较复杂，但是有规律所询：

假设特征点p对于第i，j帧都是可观测的，那么坐标系转移如下所示：

第i帧相机坐标系 >> 第i帧imu坐标系 >> 世界坐标系 >> 第j帧imu坐标系 >> 第j帧相机坐标系

不过这样的误差项会非常的稠密。假设滑窗大小是W，整个滑窗内有N个特征点可以被所有帧看见，简单计算一下，将会有 N*W 个误差项（实际会稍稍小于这个值），所以限制W的大小也是必要的。

5，ProjectionTwoFrameOneCamFactor

博客在这里推导的jacobian没有考虑tangent vector，所以把他推导的jacobian需要乘以tangent vector才算正确！

现在可以看代码了，先看第一块代码，先分析ProjectionTwoFrameOneCamFactor：

```
if (imu_i != imu_j)
{
    Vector3d pts_j = it_per_frame.point;
    // 很直观，从pts_i到pts_j
    ProjectionTwoFrameOneCamFactor *f_td = new ProjectionTwoFrameOneCamFactor(pts_i, pts_j,
                                                  it_per_id.feature_per_frame[0].velocity,
                                                      it_per_frame.velocity,

 it_per_id.feature_per_frame[0].cur_td,
                                                  it_per_frame.cur_td);
     // AddResidualBlock中待优化的变量应该都了解了
     problem.AddResidualBlock(f_td, loss_function, para_Pose[imu_i], para_Pose[imu_j],
                              para_Ex_Pose[0], para_Feature[feature_index], para_Td[0]);
}
```

第一次看到velocity的时候很费解，不清楚，做进一步分析吧。开始看ProjectionTwoFrameOneCamFactor的头文件：

```
// 经过前面的分析，这里就不难理解了
// <2, 7, 7, 7, 1, 1>意味着有五个jacobian，博客里只有四个，到底发生了什么呢？
// 根据博客的推导，应该是这样的<2, 6, 6, 6, 1>，这个疑问只能往下看
class ProjectionTwoFrameOneCamFactor : public ceres::SizedCostFunction<2, 7, 7, 7, 1, 1>
{
  public:
    ProjectionTwoFrameOneCamFactor(const Eigen::Vector3d &_pts_i, const Eigen::Vector3d &_pts_j,
                       const Eigen::Vector2d &_velocity_i, const Eigen::Vector2d &_velocity_j,
                       const double _td_i, const double _td_j);
    virtual bool Evaluate(double const *const *parameters, double *residuals, double
**jacobians) const;
    void check(double **parameters);

    Eigen::Vector3d pts_i, pts_j;
    // 老天，velocity_i是什么呢，接下来会分析
    Eigen::Vector3d velocity_i, velocity_j;
    double td_i, td_j;
    Eigen::Matrix<double, 2, 3> tangent_base;
    static Eigen::Matrix2d sqrt_info;
    static double sum_t;
};
```

接着看相应的cpp文件吧，

先看一下ProjectionTwoFrameOneCamFactor的构造函数：

```
ProjectionTwoFrameOneCamFactor::ProjectionTwoFrameOneCamFactor
                          (const Eigen::Vector3d &_pts_i, const Eigen::Vector3d &_pts_j,
                           const Eigen::Vector2d &_velocity_i, const Eigen::Vector2d
&_velocity_j,
                           const double _td_i, const double _td_j) :
                                 pts_i(_pts_i), pts_j(_pts_j),
                                 td_i(_td_i), td_j(_td_j)
```

```
                                // 上述代码中td_i和td_j是捕获该特征点的时间戳
                                // _velocity_i是第ith帧的特征点的像素速度（光流速度，由光流函数获得）
    velocity_i.x() = _velocity_i.x();
    velocity_i.y() = _velocity_i.y();
    velocity_i.z() = 0;
    velocity_j.x() = _velocity_j.x();
    velocity_j.y() = _velocity_j.y();
    velocity_j.z() = 0;

    // 这里的b1和b2求解很有意思
    Eigen::Vector3d b1, b2;
    Eigen::Vector3d a = pts_j.normalized();
    Eigen::Vector3d tmp(0, 0, 1);
    if(a == tmp)
        tmp << 1, 0, 0;
    b1 = (tmp - a * (a.transpose() * tmp)).normalized();
    b2 = a.cross(b1);
    tangent_base.block<1, 3>(0, 0) = b1.transpose();
    tangent_base.block<1, 3>(1, 0) = b2.transpose();
};
```

向量b1和b2，是特征点P在归一化坐标系下，也就是一个单位球上的切向量。可以拿草稿本简单推导一下。

然后看Evaluate函数，该函数分成三部分：

1，初始化准备

```
TicToc tic_toc;
Eigen::Vector3d Pi(parameters[0][0], parameters[0][1], parameters[0][2]);
Eigen::Quaterniond Qi(parameters[0][6], parameters[0][3], parameters[0][4], parameters[0][5]);
Eigen::Vector3d Pj(parameters[1][0], parameters[1][1], parameters[1][2]);
Eigen::Quaterniond Qj(parameters[1][6], parameters[1][3], parameters[1][4], parameters[1][5]);
Eigen::Vector3d tic(parameters[2][0], parameters[2][1], parameters[2][2]);
Eigen::Quaterniond qic(parameters[2][6], parameters[2][3], parameters[2][4], parameters[2][5]);
double inv_dep_i = parameters[3][0];
double td = parameters[4][0]; //做优化，这一时刻的时间戳
```

2，误差函数定义

```
Eigen::Vector3d pts_i_td, pts_j_td;
// 因为光流捕获特征点的时间和做优化的时间不一样，可能相差了几个毫秒
// 所以要对特征点的位置做一次预估计
// td也是需要优化的变量，接下来就会看到
pts_i_td = pts_i - (td - td_i) * velocity_i;
pts_j_td = pts_j - (td - td_j) * velocity_j;
Eigen::Vector3d pts_camera_i = pts_i_td / inv_dep_i;
Eigen::Vector3d pts_imu_i = qic * pts_camera_i + tic;
Eigen::Vector3d pts_w = Qi * pts_imu_i + Pi;
Eigen::Vector3d pts_imu_j = Qj.inverse() * (pts_w - Pj);
Eigen::Vector3d pts_camera_j = qic.inverse() * (pts_imu_j - tic);
Eigen::Map<Eigen::Vector2d> residual(residuals);
```

```
residual =  tangent_base * (pts_camera_j.normalized() - pts_j_td.normalized());
residual = sqrt_info * residual;
```

3，相关jacobian

这一部分也要分成几块来阅读：

3.1 准备工作

```
if (jacobians)
{
   Eigen::Matrix3d Ri = Qi.toRotationMatrix();
   Eigen::Matrix3d Rj = Qj.toRotationMatrix();
   Eigen::Matrix3d ric = qic.toRotationMatrix();
   Eigen::Matrix<double, 2, 3> reduce(2, 3);

   double norm = pts_camera_j.norm();
   Eigen::Matrix3d norm_jaco;
   double x1, x2, x3;
   x1 = pts_camera_j(0);
   x2 = pts_camera_j(1);
   x3 = pts_camera_j(2);
   norm_jaco <<
  1.0 / norm - x1 * x1 / pow(norm, 3), - x1 * x2 / pow(norm, 3), - x1 * x3 / pow(norm, 3),
  - x1 * x2 / pow(norm, 3), 1.0 / norm - x2 * x2 / pow(norm, 3), - x2 * x3 / pow(norm, 3),
  - x1 * x3 / pow(norm, 3),  - x2 * x3 / pow(norm, 3),  1.0 / norm - x3 * x3 / pow(norm, 3);
   reduce = tangent_base * norm_jaco;
   reduce = sqrt_info * reduce;

   // process ...
}
```

在这里会有疑惑，norm_jaco究竟是什么呢？tangent_base应该知道吧。

在那那篇博客里，给出了在相机坐标系下特征点的物理坐标对诸多优化变量的求导，即jacobian。但是，实际的误差函数是归一化坐标平面下的误差。从相机坐标系到其相应的归一化坐标，需要一个norm函数（归一化操作），根据链式法则，自然需要一个norm_jaco。然后得到归一化平面的误差之后，还得乘以该特征点在归一化平面的切向量，即误差向量投影在相机成像平面上。

3.2 第一个jacobian，对第i时刻pose求导（imu的pose）

```
if (jacobians[0])
{
   Eigen::Map<Eigen::Matrix<double, 2, 7, Eigen::RowMajor>> jacobian_pose_i(jacobians[0]);
   // 那篇博客分析的是jaco_i
   Eigen::Matrix<double, 3, 6> jaco_i;
   jaco_i.leftCols<3>() = ric.transpose() * Rj.transpose();
   jaco_i.rightCols<3>() = ric.transpose() * Rj.transpose() * Ri * -
                                           Utility::skewSymmetric(pts_imu_i);
   // 得到归一化平面的误差，理由如前所述
   jacobian_pose_i.leftCols<6>() = reduce * jaco_i;
   // 末尾一行置零，从2乘6的矩阵，变成2乘7的矩阵，为啥，原因不太清楚
   jacobian_pose_i.rightCols<1>().setZero();
}
```

jaco_i与博客推导一致。

3.3 第二个jacobian，对第j时刻的pose求导

```
if (jacobians[1])
{
   Eigen::Map<Eigen::Matrix<double, 2, 7, Eigen::RowMajor>> jacobian_pose_j(jacobians[1]);
   Eigen::Matrix<double, 3, 6> jaco_j;
   jaco_j.leftCols<3>() = ric.transpose() * -Rj.transpose();
   jaco_j.rightCols<3>() = ric.transpose() * Utility::skewSymmetric(pts_imu_j);
   jacobian_pose_j.leftCols<6>() = reduce * jaco_j;
   jacobian_pose_j.rightCols<1>().setZero();
}
```

jaco_j与博客推导一致。

3.4 第三个jacobian，对imu和camera的相对位姿求导

```
if (jacobians[2])
{
   Eigen::Map<Eigen::Matrix<double, 2, 7, Eigen::RowMajor>> jacobian_ex_pose(jacobians[2]);
   Eigen::Matrix<double, 3, 6> jaco_ex;
   jaco_ex.leftCols<3>() = ric.transpose() * (Rj.transpose() * Ri -
Eigen::Matrix3d::Identity());
   Eigen::Matrix3d tmp_r = ric.transpose() * Rj.transpose() * Ri * ric;
   jaco_ex.rightCols<3>() = -tmp_r * Utility::skewSymmetric(pts_camera_i) +
                                     Utility::skewSymmetric(tmp_r * pts_camera_i) +
           Utility::skewSymmetric(ric.transpose() * (Rj.transpose() * (Ri * tic + Pi - Pj) -
tic));
   jacobian_ex_pose.leftCols<6>() = reduce * jaco_ex;
   jacobian_ex_pose.rightCols<1>().setZero();
}
```

jaco_ex与博客推导似乎一致。

3.5 第四个jacobian，对特征点的深度求导

```
if (jacobians[3])
{
   Eigen::Map<Eigen::Vector2d> jacobian_feature(jacobians[3]);
   jacobian_feature = reduce * ric.transpose() * Rj.transpose() * Ri * ric * pts_i_td * -1.0 /
(inv_dep_i * inv_dep_i);
}
```

jacobian_feature与博客推导的一样。

3.6 第五个jacobian，对td求导

```
if (jacobians[4])
{
   Eigen::Map<Eigen::Vector2d> jacobian_td(jacobians[4]);
   jacobian_td = reduce * ric.transpose() * Rj.transpose() * Ri * ric * velocity_i / inv_dep_i *
-1.0  + sqrt_info * velocity_j.head(2);
}
```

对于vins fusion，td是一个新参数（待优化的时间），这个参数，之前矫正像素的时候出现过。因为新加了这个有优化变量，所以之前推导的jacobian会额外加一行。

总之，ProjectionTwoFrameOneCamFactor 的分析就到这里。

6，ProjectionTwoFrameTwoCamFactor

先回顾一下它出场的位置：

```
if(STEREO && it_per_frame.is_stereo)
{
   // 关键的一句话
   Vector3d pts_j_right = it_per_frame.pointRight;
   if(imu_i != imu_j)
   {
      // 在这里哦
      ProjectionTwoFrameTwoCamFactor *f = new ProjectionTwoFrameTwoCamFactor(pts_i, pts_j_right,
                                             it_per_id.feature_per_frame[0].velocity,
                                               it_per_frame.velocityRight,
                                          it_per_id.feature_per_frame[0].cur_td,
it_per_frame.cur_td);
       problem.AddResidualBlock(f, loss_function, para_Pose[imu_i], para_Pose[imu_j],
                        para_Ex_Pose[0], para_Ex_Pose[1], para_Feature[feature_index],
para_Td[0]);
    }
    else
    {
       // 这是下一节需要讲的东西
       ProjectionOneFrameTwoCamFactor *f = new ProjectionOneFrameTwoCamFactor(pts_i,
pts_j_right,
```

```
it_per_id.feature_per_frame[0].velocity,
                it_per_frame.velocityRight,
                                    it_per_id.feature_per_frame[0].cur_td,
it_per_frame.cur_td);
       problem.AddResidualBlock(f, loss_function, para_Ex_Pose[0], para_Ex_Pose[1],
                                    para_Feature[feature_index], para_Td[0]);
     }
}
```

注意看这句代码：

```
Vector3d pts_j_right = it_per_frame.pointRight;
```

注意，它们的区别是：

之前的factor是从第i时刻的左帧到第j时刻左帧建立的，

现在的factor是从第i时刻的左帧到第j时刻右帧建立的，（区别就在这里！）这个factor的细节和之前的一样！

7，ProjectionOneFrameTwoCamFactor

注意这个factor的前缀条件：imu_i = imu_j，

所以这个factor是从第j时刻的左帧到第j时刻的右帧建立的！

它的优化变量主要是：

a，imu到左相机的位姿

b，imu到右相机的位姿

c，特征点的深度信息

d，td变量（意义之前讲过哦）

有兴趣的话可以自己打开cpp文件看看，经过之前对factor的分析，再分析它已经小菜一碟了。因为博客主要讲的是vins mono，而不是这里的vins fusion，所以这部分推导还不清楚，以后会自己推推。这里不做分析。

8，factor与多传感融合

题外话。

如果我有GPS，或者轮盘编码器，亦或是其他传感器。只要我把误差函数写出来，优化变量拎出来，高斯模型建号，jacobian求好，我就可以写一个factor的函数，把这个合并到总体的大问题中。因此，factor具有可扩展性。

另外，自己做实验发现，相比vins mono，vins fusion明显慢了很多，那是因为需要优化的变量十分多，导致CPU运算变慢吧。而且这个大型的优化运算似乎没能并行化处理，如果求解Ax=b这个问题经过GPU加速，也许速度会更快吧。

9，了解slideWindow

其实optimization这个函数还没有分析完，目前看了imu和camera的相关factor。在了解marginalization的factor前，先看看slideWindow这个函数，提前做做铺垫。

在那篇博客中，对slidewindow的原因做了很清楚的叙述，这里摘抄如下：

1，利用Sliding Window做优化的过程中，边缘化的目的主要有两个：

- 滑窗内的pose和feature个数是有限的，在系统优化的过程中，势必要不断将一些pose和feature移除滑窗。
- 如果当前帧图像和上一帧添加到滑窗的图像帧视差很小，则测量的协方差(重投影误差)会很大，进而会恶化优化结果。LIFO导致了协方差的增大，而恶化优化结果

2，直接进行边缘化而不加入先验条件的后果：

- 无故地移除这些pose和feature会丢弃帧间约束，会降低了优化器的精度，所以在移除pose和feature的时候需要将相关联的约束转变为一个先验的约束条件作为prior放到优化问题中
- 在边缘化的过程中，不加先验的边缘化会导致系统尺度的缺失(参考[6])，尤其是系统在进行**退化运动**时(如无人机的悬停和恒速运动)。一般来说只有两个轴向的加速度不为0的时候，才能保证尺度可观，而退化运动对于无人机或者机器人来说是不可避免的。所以在系统处于退化运动的时候，要加入先验信息保证尺度的客观性。 以上就可以描述为边缘化的目的以及在边缘化中加入先验约束的原因。

总而言之吧，边缘化就是“扔帧”，但是不“裸扔”，而是把扔掉的帧附带的信息以某种形式保留起来（像是一种记忆作用），从而提高优化器的精度。我们先不去理会边缘化的具体数学推导。

丢帧大致过程分为两种情况：

MARGIN_OLD ：如果要删去最后一帧，则滑窗每一个帧向后挪动一个位置，把最后一帧给“挤掉”。（还让最新帧等于次新帧，因为这样移位有空缺嘛。）

MARGIN_NEW：如果要删去次新帧（不是最新帧，是次新帧），则仅仅让次新帧的信息等于新帧。

10，slidewindow中的MARGIN_OLD

这个过程分解为如下几步：

1，把最老的帧挤掉

```
if (frame_count == WINDOW_SIZE)
{
   for (int i = 0; i < WINDOW_SIZE; i++)
   {
       Headers[i] = Headers[i + 1];
       Rs[i].swap(Rs[i + 1]);
       Ps[i].swap(Ps[i + 1]);
       if(USE_IMU)
       {
          std::swap(pre_integrations[i], pre_integrations[i + 1]);
          dt_buf[i].swap(dt_buf[i + 1]);
          linear_acceleration_buf[i].swap(linear_acceleration_buf[i + 1]);
          angular_velocity_buf[i].swap(angular_velocity_buf[i + 1]);
          Vs[i].swap(Vs[i + 1]);
          Bas[i].swap(Bas[i + 1]);
          Bgs[i].swap(Bgs[i + 1]);
       }
   }
   if (true || solver_flag == INITIAL)
   {
      map<double, ImageFrame>::iterator it_0;
      it_0 = all_image_frame.find(t_0);

      delete it_0->second.pre_integration;
```

```
        all_image_frame.erase(all_image_frame.begin(), it_0);
    }
    // process ...
}
```

2，最新的帧信息等于次新的帧信息

```
Headers[WINDOW_SIZE] = Headers[WINDOW_SIZE - 1];
Ps[WINDOW_SIZE] = Ps[WINDOW_SIZE - 1];
Rs[WINDOW_SIZE] = Rs[WINDOW_SIZE - 1];

if(USE_IMU)
{
    Vs[WINDOW_SIZE] = Vs[WINDOW_SIZE - 1];
    Bas[WINDOW_SIZE] = Bas[WINDOW_SIZE - 1];
    Bgs[WINDOW_SIZE] = Bgs[WINDOW_SIZE - 1];

    delete pre_integrations[WINDOW_SIZE];
    pre_integrations[WINDOW_SIZE] = new IntegrationBase{acc_0, gyr_0, Bas[WINDOW_SIZE],
Bgs[WINDOW_SIZE]};
    dt_buf[WINDOW_SIZE].clear();
    linear_acceleration_buf[WINDOW_SIZE].clear();
    angular_velocity_buf[WINDOW_SIZE].clear();
}
```

3，slideWindowOld()

```
slideWindowOld();
```

重点是slideWindowOld咋工作的，看看函数哈：

```
void Estimator::slideWindowOld()
{
    sum_of_back++;
    bool shift_depth = solver_flag == NON_LINEAR ? true : false;
    if (shift_depth)
    {
        // 0指的是丢弃老帧对应的R，P
        // 1指的是丢弃老帧的下一帧对应的R，P
        // 做一个交接工作，把老帧的信息传给新帧
        Matrix3d R0, R1;
        Vector3d P0, P1;
        R0 = back_R0 * ric[0];
        R1 = Rs[0] * ric[0];
        P0 = back_P0 + back_R0 * tic[0];
        P1 = Ps[0] + Rs[0] * tic[0];
        // 交接工作
        f_manager.removeBackShiftDepth(R0, P0, R1, P1);
    }

    else
```

```
        f_manager.removeBack();
}
```

来看看removeBackShiftDepth吧：

函数定义式如下所示：

```
void FeatureManager::removeBackShiftDepth(Eigen::Matrix3d marg_R, Eigen::Vector3d marg_P,
                                          Eigen::Matrix3d new_R, Eigen::Vector3d new_P)
```

细节代码如下所示：

```
for (auto it = feature.begin(), it_next = feature.begin(); it != feature.end(); it = it_next)
{
   it_next++;
   // 把滑窗中，第2帧到n帧，缩减成，第1帧到第n-1帧
   if (it->start_frame != 0)
      it->start_frame--;
   // 对于原来的第一帧，就要丢弃了，在丢弃之前，把相关信息传给第2帧
   else
   {
      Eigen::Vector3d uv_i = it->feature_per_frame[0].point;
      it->feature_per_frame.erase(it->feature_per_frame.begin());
      // 如果丢弃的帧，没啥特征点，丢了算了
      if (it->feature_per_frame.size() < 2)
      {
         feature.erase(it);
         continue;
      }
      // 正题来了
      else
      {
          // 把这些特征点的深度估计交接上
          Eigen::Vector3d pts_i = uv_i * it->estimated_depth;
          Eigen::Vector3d w_pts_i = marg_R * pts_i + marg_P;
          Eigen::Vector3d pts_j = new_R.transpose() * (w_pts_i - new_P);
          double dep_j = pts_j(2);
          if (dep_j > 0)
             it->estimated_depth = dep_j;
          else
             it->estimated_depth = INIT_DEPTH;
       }
  }
}
```

在margin_old中，似乎“边缘化”并不神秘。

11，slidewindow中的MARGIN_NEW

这一块分成两步：

1，把次新帧换掉，

```
if (frame_count == WINDOW_SIZE)
{
   Headers[frame_count - 1] = Headers[frame_count];
   Ps[frame_count - 1] = Ps[frame_count];
   Rs[frame_count - 1] = Rs[frame_count];
   if(USE_IMU)
   {
       // 似乎是次新帧换成最新帧，imu做一次预积分，具体我不太清楚
       for (unsigned int i = 0; i < dt_buf[frame_count].size(); i++)
       {
           double tmp_dt = dt_buf[frame_count][i];
           Vector3d tmp_linear_acceleration = linear_acceleration_buf[frame_count][i];
           Vector3d tmp_angular_velocity = angular_velocity_buf[frame_count][i];
           pre_integrations[frame_count - 1]->push_back(tmp_dt, tmp_linear_acceleration,
tmp_angular_velocity);
           dt_buf[frame_count - 1].push_back(tmp_dt);
           linear_acceleration_buf[frame_count - 1].push_back(tmp_linear_acceleration);
           angular_velocity_buf[frame_count - 1].push_back(tmp_angular_velocity);
        }
        Vs[frame_count - 1] = Vs[frame_count];
        Bas[frame_count - 1] = Bas[frame_count];
        Bgs[frame_count - 1] = Bgs[frame_count];
        delete pre_integrations[WINDOW_SIZE];
        pre_integrations[WINDOW_SIZE] = new IntegrationBase{acc_0, gyr_0, Bas[WINDOW_SIZE],
Bgs[WINDOW_SIZE]};
        dt_buf[WINDOW_SIZE].clear();
        linear_acceleration_buf[WINDOW_SIZE].clear();
        angular_velocity_buf[WINDOW_SIZE].clear();
    }
    // process ...
}
```

2，slideWindowNew

```
slideWindowNew();
```

看看new吧，

```
void Estimator::slideWindowNew()
{
    sum_of_front++;
    f_manager.removeFront(frame_count);
}
```

没有old那么复杂，因为new不涉及交接过程，看看removeFront，

```
for (auto it = feature.begin(), it_next = feature.begin(); it != feature.end(); it = it_next)
{
```

```
    it_next++;
    // 把最新帧变成次新帧
    if (it->start_frame == frame_count)
    {
       it->start_frame--;
    }
    else
    {
       int j = WINDOW_SIZE - 1 - it->start_frame;
       // 相当于只有it->endFrame() = frame_count - 1，就是次新帧，才执行后面的代码
       if (it->endFrame() < frame_count - 1)
          continue;
       // 就是在剔除次新帧中的特征点，好像把这一帧的特征点都剃没了？？
       it->feature_per_frame.erase(it->feature_per_frame.begin() + j);
       if (it->feature_per_frame.size() == 0)
         feature.erase(it);
    }
}
```

12，初步分析MarginalizationFactor

对slidewindow有一定的了解，好吧，是时候分析这个家伙了。先回顾一下它出现的代码（出现不止一处，现分析一个简单的）：

```
if (last_marginalization_info && last_marginalization_info->valid)
{
 // construct new marginlization_factor
 MarginalizationFactor *marginalization_factor = new
                                          MarginalizationFactor(last_marginalization_info);
 problem.AddResidualBlock(marginalization_factor, NULL,
                                          last_marginalization_parameter_blocks);
}
```

先看看MarginalizationFactor吧，有点麻烦，先来看一下它的头文件：

```
// 需要四个线程
const int NUM_THREADS = 4;

struct ResidualBlockInfo
{
    ResidualBlockInfo(ceres::CostFunction *_cost_function, ceres::LossFunction *_loss_function,
std::vector<double *> _parameter_blocks, std::vector<int> _drop_set)
        : cost_function(_cost_function), loss_function(_loss_function),
parameter_blocks(_parameter_blocks), drop_set(_drop_set) {}

    void Evaluate();

    ceres::CostFunction *cost_function;
    ceres::LossFunction *loss_function;
    std::vector<double *> parameter_blocks;
    std::vector<int> drop_set;
```

```
    double **raw_jacobians;
    std::vector<Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>>
jacobians;
    Eigen::VectorXd residuals;

    int localSize(int size)
    {
        return size == 7 ? 6 : size;
    }
};

struct ThreadsStruct
{
    std::vector<ResidualBlockInfo *> sub_factors;
    Eigen::MatrixXd A;
    Eigen::VectorXd b;
    std::unordered_map<long, int> parameter_block_size; //global size
    std::unordered_map<long, int> parameter_block_idx; //local size
};

class MarginalizationInfo
{
  public:
    MarginalizationInfo(){valid = true;};
    ~MarginalizationInfo();
    int localSize(int size) const;
    int globalSize(int size) const;
    void addResidualBlockInfo(ResidualBlockInfo *residual_block_info);
    void preMarginalize();
    void marginalize();
    std::vector<double *> getParameterBlocks(std::unordered_map<long, double *> &addr_shift);

    std::vector<ResidualBlockInfo *> factors;
    int m, n;
    std::unordered_map<long, int> parameter_block_size; //global size
    int sum_block_size;
    std::unordered_map<long, int> parameter_block_idx; //local size
    std::unordered_map<long, double *> parameter_block_data;

    std::vector<int> keep_block_size; //global size
    std::vector<int> keep_block_idx;  //local size
    std::vector<double *> keep_block_data;

    Eigen::MatrixXd linearized_jacobians;
    Eigen::VectorXd linearized_residuals;
    const double eps = 1e-8;
    bool valid;

};

class MarginalizationFactor : public ceres::CostFunction

{
```

```
  public:
    MarginalizationFactor(MarginalizationInfo* _marginalization_info);
    virtual bool Evaluate(double const *const *parameters, double *residuals, double
**jacobians) const;

    MarginalizationInfo* marginalization_info;
};
```

一开始看这段代码，不知道这些class的具体含义，但是还是能够找到一些蛛丝：

1，MarginalizationInfo 内含类ResidualBlockInfo （该类提供evaluate的接口）

2，MarginalizationFactor 内含类MarginalizationInfo ，提供evaluate的接口

来看一下，新建一个MarginalizationFactor 需要哪些操作，

```
MarginalizationFactor::MarginalizationFactor(MarginalizationInfo*
_marginalization_info):marginalization_info(_marginalization_info)
{
    int cnt = 0;
    for (auto it : marginalization_info->keep_block_size)
    {
        mutable_parameter_block_sizes()->push_back(it);
        cnt += it;
    }
    //printf("residual size: %d, %d\n", cnt, n);
    set_num_residuals(marginalization_info->n);
};
```

MarginalizationFactor 的初始化依赖于MarginalizationInfo ，而MarginalizationInfo是默认初始化。在这里似乎没有看到什么有用的信息。

另外，对边缘化的数学上的概念不理解，可以看看巧克力大神的博客：https://blog.csdn.net/heyijia0327/article/details/53707261，讲得很清楚，很细致。

现在看 MarginalizationFactor 的evaluate函数吗？不，现在还不到看它的时候。

总之，读博客，我们知道了边缘化的基本原理，但是从这段代码里面还看不出什么来。

13，分析在margin old情况下，marginlize的过程

该过程可以分成几步：

1，数据准备

```
MarginalizationInfo *marginalization_info = new MarginalizationInfo();
// 把优化变量重新写成parameter变量中
vector2double();
```

2，继承上一次的marginlize factor

```

if (last_marginalization_info && last_marginalization_info->valid)
```

```
{
   vector<int> drop_set;
   // 遍历，把含有最老变量的放到drop_set中
   // 在margin old中，要把最老的信息丢掉，丢掉前需要把它变成先验信息的一部分。
   for (int i = 0; i < static_cast<int>(last_marginalization_parameter_blocks.size()); i++)
   {
       if (last_marginalization_parameter_blocks[i] == para_Pose[0] ||
           last_marginalization_parameter_blocks[i] == para_SpeedBias[0])
              drop_set.push_back(i);
   }
   // construct new marginlization_factor
   MarginalizationFactor *marginalization_factor = new
                                                MarginalizationFactor(last_marginalization_info);
   ResidualBlockInfo *residual_block_info = new ResidualBlockInfo(marginalization_factor, NULL,
                                             last_marginalization_parameter_blocks,drop_set);
   marginalization_info->addResidualBlockInfo(residual_block_info);
}
```

ResidualBlockInfo的构造没有专门的函数，需要留意一下drop_set的位置。

重点需要分析的是类marginalization_info的addResidualBlockInfo函数：

```
void MarginalizationInfo::addResidualBlockInfo(ResidualBlockInfo *residual_block_info)
{
    // 把这次的factor放入总的factors中
    // factors是MarginalizationInfo的维护变量
    factors.emplace_back(residual_block_info);

    std::vector<double *> &parameter_blocks = residual_block_info->parameter_blocks;
    std::vector<int> parameter_block_sizes = residual_block_info->cost_function-
>parameter_block_sizes();

    // 这些变量的命名都有些迷，不清楚啥含义，所以先把关联找到吧
    // 对于non-marginlize变量，parameter_block_size
    for (int i = 0; i < static_cast<int>(residual_block_info->parameter_blocks.size()); i++)
    {
        double *addr = parameter_blocks[i];
        int size = parameter_block_sizes[i];
        parameter_block_size[reinterpret_cast<long>(addr)] = size;
    }
    // 对于marginlize变量，parameter_block_idx
    for (int i = 0; i < static_cast<int>(residual_block_info->drop_set.size()); i++)
    {
        double *addr = parameter_blocks[residual_block_info->drop_set[i]];
        parameter_block_idx[reinterpret_cast<long>(addr)] = 0;
    }
}
```

3，把imu factor放入marginlize factor中

```
if(USE_IMU)
{
   if (pre_integrations[1]->sum_dt < 10.0)
   {
      // 边缘化最老帧，所以imu factor跟最老帧和次老帧有关系，跟其他帧没有关系。
      IMUFactor* imu_factor = new IMUFactor(pre_integrations[1]);
      ResidualBlockInfo *residual_block_info = new ResidualBlockInfo(imu_factor, NULL,
           vector<double *>{para_Pose[0], para_SpeedBias[0], para_Pose[1], para_SpeedBias[1]},
                                                                          vector<int>{0, 1});
      // 从这里，可以可以理解marginalization_parameter_blocks和drop_set的具体含义
      marginalization_info->addResidualBlockInfo(residual_block_info);
   }
}
```

4，把视觉相关的factor放到marginlize factor中，

```
int feature_index = -1;
for (auto &it_per_id : f_manager.feature)
{
    it_per_id.used_num = it_per_id.feature_per_frame.size();
    if (it_per_id.used_num < 4)
       continue;
    ++feature_index;
    int imu_i = it_per_id.start_frame, imu_j = imu_i - 1;
    if (imu_i != 0)
       continue;
    // 遍历所有的feature point选出imu_i=0的，就是最老帧和次老帧
    Vector3d pts_i = it_per_id.feature_per_frame[0].point;
    for (auto &it_per_frame : it_per_id.feature_per_frame)
    {
       imu_j++;
       if(imu_i != imu_j)
       {
          Vector3d pts_j = it_per_frame.point;
          // 建立最老帧左帧和次老帧左帧之间的factor
          ProjectionTwoFrameOneCamFactor *f_td = new ProjectionTwoFrameOneCamFactor(pts_i,
pts_j,                                        it_per_id.feature_per_frame[0].velocity,
it_per_frame.velocity,
                                  it_per_id.feature_per_frame[0].cur_td, it_per_frame.cur_td);
          ResidualBlockInfo *residual_block_info = new ResidualBlockInfo(f_td, loss_function,
vector<double *>{para_Pose[imu_i], para_Pose[imu_j], para_Ex_Pose[0],
para_Feature[feature_index], para_Td[0]},vector<int>{0, 3});
          marginalization_info->addResidualBlockInfo(residual_block_info);
       }
       if(STEREO && it_per_frame.is_stereo)
       {
           Vector3d pts_j_right = it_per_frame.pointRight;
           if(imu_i != imu_j)
           {
            // 建立最老帧左帧和次老帧右帧之间的factor

            ProjectionTwoFrameTwoCamFactor *f = new ProjectionTwoFrameTwoCamFactor(pts_i,
```

```
                    pts_j_right, it_per_id.feature_per_frame[0].velocity,
it_per_frame.velocityRight,
                                    it_per_id.feature_per_frame[0].cur_td,
it_per_frame.cur_td);
            ResidualBlockInfo *residual_block_info = new ResidualBlockInfo(f, loss_function,
            vector<double *>{para_Pose[imu_i], para_Pose[imu_j], para_Ex_Pose[0],
para_Ex_Pose[1], para_Feature[feature_index], para_Td[0]},vector<int>{0, 4});
            marginalization_info->addResidualBlockInfo(residual_block_info);
           }
           else
           {
             // 建立最老帧左帧和最老帧右帧之间的factor
             ProjectionOneFrameTwoCamFactor *f = new ProjectionOneFrameTwoCamFactor(pts_i,
               pts_j_right, it_per_id.feature_per_frame[0].velocity, it_per_frame.velocityRight,
                            it_per_id.feature_per_frame[0].cur_td, it_per_frame.cur_td);
             ResidualBlockInfo *residual_block_info = new ResidualBlockInfo(f, loss_function,
             vector<double *>{para_Ex_Pose[0], para_Ex_Pose[1], para_Feature[feature_index],
para_Td[0]},vector<int>{2});
             marginalization_info->addResidualBlockInfo(residual_block_info);
            }
        }
   }
}
```

5，进行边缘化操作

把联合分布（老帧的优化变量，次老帧的优化变量）变成一个条件分布（次老帧的优化变量|老帧的优化变量）

```
TicToc t_pre_margin;
marginalization_info->preMarginalize();
ROS_DEBUG("pre marginalization %f ms", t_pre_margin.toc());

TicToc t_margin;
marginalization_info->marginalize();
ROS_DEBUG("marginalization %f ms", t_margin.toc());
```

这里暂不分析preMarginalize和marginalize，等一会再分析

6，把最老帧挤掉

```
std::unordered_map<long, double *> addr_shift;
for (int i = 1; i <= WINDOW_SIZE; i++)
{
    addr_shift[reinterpret_cast<long>(para_Pose[i])] = para_Pose[i - 1];
    if(USE_IMU)
        addr_shift[reinterpret_cast<long>(para_SpeedBias[i])] = para_SpeedBias[i - 1];
}
for (int i = 0; i < NUM_OF_CAM; i++)
    addr_shift[reinterpret_cast<long>(para_Ex_Pose[i])] = para_Ex_Pose[i];
addr_shift[reinterpret_cast<long>(para_Td[0])] = para_Td[0];
```

7，更新last_marginalization_info

```
vector<double *> parameter_blocks = marginalization_info->getParameterBlocks(addr_shift);
if (last_marginalization_info)
   delete last_marginalization_info;
last_marginalization_info = marginalization_info;
last_marginalization_parameter_blocks = parameter_blocks;
```

至此，margin old的步骤到此就结束了。

PS：翻阅早期的vins论文（2017），发现那时候有了初始化操作，但是还没有margin哦。2018年的论文才把它加上去的，科研都是一步一步走的。

14，margin细节的探讨

在这里我们将分析，上一讲没有分析的preMarginalize和marginalize两个函数。

尽管我对margin的数学概念有所了解，对于高斯分布而言，从联合分布到条件分布，在代数上等价于Schur补（题外话，在ba稀疏解释的时候也有Schur补，Schur补可以用在很多地方）。但是对于一个具体的project，它是怎样margin呢？

首先，联合分布是什么，怎么求的？联合分布是（老帧的优化变量，次老帧的优化变量）。在滑窗优化的时候，把整个滑窗所有帧对应的优化变量，相应的协方差矩阵都求了一遍。联合分布由一个向量和一个大的协方差矩阵表示。具体的计算马上看看代码。

对ceres的loss function不理解，可以看看网址：http://www.ceres-solver.org/nnls_modeling.html#lossfunction

其次，联合分布弄出来之后，再去做条件分布就是纯计算问题了，就是Schur补。

做过热身，可以看看代码了，首先看一下preMarginalize函数

```
void MarginalizationInfo::preMarginalize()
{
    for (auto it : factors)
    {
        // 计算每个factor的鲁棒误差以及鲁棒jacobian，说白了就是误差函数前面见了一个Huber核，
        // 加了Huber核之后，误差和jacobian就变了。
        it->Evaluate();
        std::vector<int> block_sizes = it->cost_function->parameter_block_sizes();
        for (int i = 0; i < static_cast<int>(block_sizes.size()); i++)
        {
            long addr = reinterpret_cast<long>(it->parameter_blocks[i]);
            int size = block_sizes[i];
            if (parameter_block_data.find(addr) == parameter_block_data.end())
            {
                double *data = new double[size];
                memcpy(data, it->parameter_blocks[i], sizeof(double) * size);
                parameter_block_data[addr] = data;
            }
        }
    }
}
```

注意这段代码注释的内容，我们再去看函数ResidualBlockInfo::Evaluate：

```
void ResidualBlockInfo::Evaluate()
{
    residuals.resize(cost_function->num_residuals());

    std::vector<int> block_sizes = cost_function->parameter_block_sizes();
    raw_jacobians = new double *[block_sizes.size()];
    jacobians.resize(block_sizes.size());

    for (int i = 0; i < static_cast<int>(block_sizes.size()); i++)
    {
        jacobians[i].resize(cost_function->num_residuals(), block_sizes[i]);
        raw_jacobians[i] = jacobians[i].data();
        //dim += block_sizes[i] == 7 ? 6 : block_sizes[i];
    }
    // 计算原始的误差函数和jacobian
    cost_function->Evaluate(parameter_blocks.data(), residuals.data(), raw_jacobians);
    // 如果有鲁棒核的话
    // 下面代码建议参看网址：http://www.ceres-solver.org/nnls_modeling.html#lossfunction
    // 就会对下面的运算有更深的理解
    if (loss_function)
    {
        double residual_scaling_, alpha_sq_norm_;
        double sq_norm, rho[3];
        // rho[3]是鲁棒误差函数，其一阶导数，以及二阶导数
        sq_norm = residuals.squaredNorm();
        loss_function->Evaluate(sq_norm, rho);
        double sqrt_rho1_ = sqrt(rho[1]);
        if ((sq_norm == 0.0) || (rho[2] <= 0.0))
        {
            residual_scaling_ = sqrt_rho1_;
            alpha_sq_norm_ = 0.0;
        }
        else
        {
            // 解一个一元二次方程，最后解是这个
            const double D = 1.0 + 2.0 * sq_norm * rho[2] / rho[1];
            const double alpha = 1.0 - sqrt(D);
            residual_scaling_ = sqrt_rho1_ / (1 - alpha);
            alpha_sq_norm_ = alpha / sq_norm;
        }
        for (int i = 0; i < static_cast<int>(parameter_blocks.size()); i++)
        {
            jacobians[i] = sqrt_rho1_ * (jacobians[i] - alpha_sq_norm_ * residuals *
(residuals.transpose() * jacobians[i]));
        }
        residuals *= residual_scaling_;
    }
}
```

总而言之，premarginlize就是求各个factor的鲁棒误差函数以及相应的jacobian。

继premarginlize之后，再去分析marginlize函数，而marginlize的核心是schur补，该函数的执行步骤如下所示：

1，初始化，m是被margin的变量维数，n是次老帧的变量维数

```
int pos = 0;
for (auto &it : parameter_block_idx)
{
    it.second = pos;
    pos += localSize(parameter_block_size[it.first]);
}
m = pos;
for (const auto &it : parameter_block_size)
{
    if (parameter_block_idx.find(it.first) == parameter_block_idx.end())
    {
       parameter_block_idx[it.first] = pos;
       pos += localSize(it.second);
    }
}
n = pos - m;
```

2，计算矩阵A，b（多线程求解），其中A是要被Schur补的

```
TicToc t_summing;
Eigen::MatrixXd A(pos, pos);
Eigen::VectorXd b(pos);
A.setZero();
b.setZero();
TicToc t_thread_summing;
pthread_t tids[NUM_THREADS];
ThreadsStruct threadsstruct[NUM_THREADS];
int i = 0;
for (auto it : factors)
{
    threadsstruct[i].sub_factors.push_back(it);
    i++;
    i = i % NUM_THREADS;
}
for (int i = 0; i < NUM_THREADS; i++)
{
    TicToc zero_matrix;
    threadsstruct[i].A = Eigen::MatrixXd::Zero(pos,pos);
    threadsstruct[i].b = Eigen::VectorXd::Zero(pos);
    threadsstruct[i].parameter_block_size = parameter_block_size;
    threadsstruct[i].parameter_block_idx = parameter_block_idx;
    int ret = pthread_create( &tids[i], NULL, ThreadsConstructA ,(void*)&(threadsstruct[i]));
    if (ret != 0)
    {
        ROS_WARN("pthread_create error");
        ROS_BREAK();
    }

}
```

```
for( int i = NUM_THREADS - 1; i >= 0; i--)
{
    // 为啥是连加，这个可以自己简单推导一下哈
    pthread_join( tids[i], NULL );
    A += threadsstruct[i].A;
    b += threadsstruct[i].b;
}
```

3，对矩阵A进行Schur补（如果不熟悉，可以再看看网址：https://blog.csdn.net/heyijia0327/article/details/53707261）

```
// 强制是对称矩阵
Eigen::MatrixXd Amm = 0.5 * (A.block(0, 0, m, m) + A.block(0, 0, m, m).transpose());
Eigen::SelfAdjointEigenSolver<Eigen::MatrixXd> saes(Amm);
// 注意，这种求逆方式的前提是矩阵是可逆的
Eigen::MatrixXd Amm_inv = saes.eigenvectors() * Eigen::VectorXd((saes.eigenvalues().array() >
eps).select(saes.eigenvalues().array().inverse(), 0)).asDiagonal() *
saes.eigenvectors().transpose();
// 很直接，无需什么解释
Eigen::VectorXd bmm = b.segment(0, m);
Eigen::MatrixXd Amr = A.block(0, m, m, n);
Eigen::MatrixXd Arm = A.block(m, 0, n, m);
Eigen::MatrixXd Arr = A.block(m, m, n, n);
Eigen::VectorXd brr = b.segment(m, n);
A = Arr - Arm * Amm_inv * Amr;
b = brr - Arm * Amm_inv * bmm;
```

4，计算linearized_residuals和linearized_jacobians

```
Eigen::SelfAdjointEigenSolver<Eigen::MatrixXd> saes2(A);
Eigen::VectorXd S = Eigen::VectorXd((saes2.eigenvalues().array() >
eps).select(saes2.eigenvalues().array(), 0));
Eigen::VectorXd S_inv = Eigen::VectorXd((saes2.eigenvalues().array() >
eps).select(saes2.eigenvalues().array().inverse(), 0));

Eigen::VectorXd S_sqrt = S.cwiseSqrt();
Eigen::VectorXd S_inv_sqrt = S_inv.cwiseSqrt();

linearized_jacobians = S_sqrt.asDiagonal() * saes2.eigenvectors().transpose();
linearized_residuals = S_inv_sqrt.asDiagonal() * saes2.eigenvectors().transpose() * b;
```

不禁要问，linearized_residuals和linearized_jacobians分别是什么，相应的几何意义又是什么？首先解释一下它们是什么。在vins的marginlize过程中，利用schur补，把最老帧对应的优化变量信息，转化为次老帧对应的优化变量的先验信息，然后估计了（在这个先验信息下）次老帧对应优化变量的jacobian和residuals。

辣么，linearized_jacobians有啥用呢？这个用处主要在marginlize factor会讲到。留在后面在讲。这里提前剧透一下，marginlize factor，作为一个factor，会有相应的jacobian和residual。该factor的jacobian和residual的一部分就是这里求的linearized_residuals和linearized_jacobians。

另外，这里求的linearized_jacobian也是First Estimate Jacobian，因为我们没有用marginlize提供的先验，再次计算jacobian。至于FEJ有什么性质，我不太清楚，只知道FEJ比较“好”。

也有大牛说，这和传统意义的FEJ不一样。

暂且当它是一种操作吧，哈哈。理解就行了，不在乎它是不是fej。

5，把老帧删掉，滑窗元素整体左移，把最老帧“挤掉”

```
std::unordered_map<long, double *> addr_shift;
for (int i = 1; i <= WINDOW_SIZE; i++)
{
    addr_shift[reinterpret_cast<long>(para_Pose[i])] = para_Pose[i - 1];
    if(USE_IMU)
       addr_shift[reinterpret_cast<long>(para_SpeedBias[i])] = para_SpeedBias[i - 1];
 }
 for (int i = 0; i < NUM_OF_CAM; i++)
    addr_shift[reinterpret_cast<long>(para_Ex_Pose[i])] = para_Ex_Pose[i];
 addr_shift[reinterpret_cast<long>(para_Td[0])] = para_Td[0];
 vector<double *> parameter_blocks = marginalization_info->getParameterBlocks(addr_shift);
```

6，把当前的marginlizeinfo当作last_info

```
if (last_marginalization_info)
   delete last_marginalization_info;
last_marginalization_info = marginalization_info;
last_marginalization_parameter_blocks = parameter_blocks;
```

哈，于是我现在才明白，last_marginalization_info是怎样得到！

故事到这里就结束了吗？从代码上看，已经结束了。然而从逻辑上，并没有哦，还记得这段代码吧：

7，计算marginlize factor（紧耦合优化的一部分，兴许你还记得）

```
if (last_marginalization_info && last_marginalization_info->valid)
{
   MarginalizationFactor *marginalization_factor = new

MarginalizationFactor(last_marginalization_info);
   problem.AddResidualBlock(marginalization_factor, NULL,
                                last_marginalization_parameter_blocks);
}
```

现在来看看MarginalizationFactor的构造函数。我们之前想分析这个构造函数，但发现那个时候分析还不到时候，现在是时候分析他了。主要看这个factor的evaluate函数：

```
bool MarginalizationFactor::Evaluate(double const *const *parameters, double *residuals, double
**jacobians) const
{
    int n = marginalization_info->n;
    int m = marginalization_info->m;
    Eigen::VectorXd dx(n);
    for (int i = 0; i < static_cast<int>(marginalization_info->keep_block_size.size()); i++)
    {
```

```
        int size = marginalization_info->keep_block_size[i];
        int idx = marginalization_info->keep_block_idx[i] - m;
        Eigen::VectorXd x = Eigen::Map<const Eigen::VectorXd>(parameters[i], size);
        Eigen::VectorXd x0 = Eigen::Map<const Eigen::VectorXd>(marginalization_info-
>keep_block_data[i], size);
        if (size != 7)
            // 如果size = 7，dx就是x和x0的差值，x0是初始值，x是优化迭代值。
            // 如果size != 7，虽然不太懂后面代码的意思，但是也是求x和x0的差值
            dx.segment(idx, size) = x - x0;
        else
        {
            dx.segment<3>(idx + 0) = x.head<3>() - x0.head<3>();
            dx.segment<3>(idx + 3) = 2.0 * Utility::positify(Eigen::Quaterniond(x0(6), x0(3),
x0(4), x0(5)).inverse() * Eigen::Quaterniond(x(6), x(3), x(4), x(5))).vec();
            if (!((Eigen::Quaterniond(x0(6), x0(3), x0(4), x0(5)).inverse() *
Eigen::Quaterniond(x(6), x(3), x(4), x(5))).w() >= 0))
            {
                dx.segment<3>(idx + 3) = 2.0 * -Utility::positify(Eigen::Quaterniond(x0(6),
x0(3), x0(4), x0(5)).inverse() * Eigen::Quaterniond(x(6), x(3), x(4), x(5))).vec();
            }
        }
    }
    // 自然根据差值dx求factor的residual，这里就用到了linearized_residuals和linearized_jacobians
    Eigen::Map<Eigen::VectorXd>(residuals, n) = marginalization_info->linearized_residuals +
marginalization_info->linearized_jacobians * dx;
    if (jacobians)
    {
        for (int i = 0; i < static_cast<int>(marginalization_info->keep_block_size.size()); i++)
        {
            // 求factor的jacobian，这里用到了linearized_jacobians。
            if (jacobians[i])
            {
                int size = marginalization_info->keep_block_size[i], local_size =
marginalization_info->localSize(size);
                int idx = marginalization_info->keep_block_idx[i] - m;
                Eigen::Map<Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic,
Eigen::RowMajor>> jacobian(jacobians[i], n, size);
                jacobian.setZero();
                // 在这一行
                jacobian.leftCols(local_size) = marginalization_info-
>linearized_jacobians.middleCols(idx, local_size);
            }
        }
    }
    return true;
}
```

总之，大概把marginlize old的流程和框架过了一遍。网上这一块的资源比较少。相关的公式我也不太会推。

不管怎样吧，vins的未解之谜——“marginlize”，总算解开一部分吧，哈哈，开心。

15，分析在marginold new情况下，marginlize的过程

之前我们分析了marginlize old，现在分析marginlize new，即次新帧替换为最新帧的过程。在new过程，最老帧没有动，所以marginlize的先验信息（仅跟最老帧挂钩）也没有改变。因此整个new过程要简单一点，如下所示：

```
// 到了marginlize new
else
{
  if (last_marginalization_info &&
            std::count(std::begin(last_marginalization_parameter_blocks),
std::end(last_marginalization_parameter_blocks), para_Pose[WINDOW_SIZE - 1]))
  {
     MarginalizationInfo *marginalization_info = new MarginalizationInfo();
     vector2double();
     if (last_marginalization_info && last_marginalization_info->valid)
     {
         vector<int> drop_set;
         for (int i = 0; i < static_cast<int>(last_marginalization_parameter_blocks.size());
i++)
         {
            ROS_ASSERT(last_marginalization_parameter_blocks[i] != para_SpeedBias[WINDOW_SIZE -
1]);
            if (last_marginalization_parameter_blocks[i] == para_Pose[WINDOW_SIZE - 1])
               drop_set.push_back(i);
          }
          // construct new marginlization_factor
          // 因为最老帧不会被扔掉，所以marginlize info沿用之前的last_marginalization_info
          MarginalizationFactor *marginalization_factor = new
                                              MarginalizationFactor(last_marginalization_info);
          ResidualBlockInfo *residual_block_info = new ResidualBlockInfo(marginalization_factor,
                                                   NULL,last_marginalization_parameter_blocks,
                                                                          drop_set);
           marginalization_info->addResidualBlockInfo(residual_block_info);
     }
     TicToc t_pre_margin;
     ROS_DEBUG("begin marginalization");
     marginalization_info->preMarginalize();
     ROS_DEBUG("end pre marginalization, %f ms", t_pre_margin.toc());
     TicToc t_margin;
     ROS_DEBUG("begin marginalization");
     marginalization_info->marginalize();
     ROS_DEBUG("end marginalization, %f ms", t_margin.toc());
     // 把次老帧换掉，滑窗内其他元素移位
     std::unordered_map<long, double *> addr_shift;
     for (int i = 0; i <= WINDOW_SIZE; i++)
     {
          if (i == WINDOW_SIZE - 1)
             continue;
          else if (i == WINDOW_SIZE)
          {
             addr_shift[reinterpret_cast<long>(para_Pose[i])] = para_Pose[i - 1];
             if(USE_IMU)
                addr_shift[reinterpret_cast<long>(para_SpeedBias[i])] = para_SpeedBias[i - 1];
           }
           else
```

```
            {
                // 嗯？这代码没必要写吧。。哈。其实这个for语句没必要用。
                addr_shift[reinterpret_cast<long>(para_Pose[i])] = para_Pose[i];
                if(USE_IMU)
                  addr_shift[reinterpret_cast<long>(para_SpeedBias[i])] = para_SpeedBias[i];
            }
        }
        for (int i = 0; i < NUM_OF_CAM; i++)
            addr_shift[reinterpret_cast<long>(para_Ex_Pose[i])] = para_Ex_Pose[i];
        addr_shift[reinterpret_cast<long>(para_Td[0])] = para_Td[0];
        vector<double *> parameter_blocks = marginalization_info->getParameterBlocks(addr_shift);
        if (last_marginalization_info)
           delete last_marginalization_info;
        last_marginalization_info = marginalization_info;
        last_marginalization_parameter_blocks = parameter_blocks;
       }
    }
}
```

16，小结一下

至此，整个紧耦合优化就分析完了。我们分析了IMU factor，projection factor，slide window，marginlize factor，对上述的这些factor有了初步的认识。接下来，我们把目光转向回环优化这一块。

17，初探pose_graph_node.cpp

开始看一下pose_graph_node.cpp（这里才有关键帧的概念）的main函数，大致分为几步：

1，载入闭环检测词典

```
string vocabulary_file = pkg_path + "/../support_files/brief_k10L6.bin";
cout << "vocabulary_file" << vocabulary_file << endl;
posegraph.loadVocabulary(vocabulary_file);
```

2，载入brief描述pattern

```
BRIEF_PATTERN_FILE = pkg_path + "/../support_files/brief_pattern.yml";
cout << "BRIEF_PATTERN_FILE" << BRIEF_PATTERN_FILE << endl;
```

3，载入相机模型

```
int pn = config_file.find_last_of('/');
std::string configPath = config_file.substr(0, pn);
std::string cam0Calib;
fsSettings["cam0_calib"] >> cam0Calib;
std::string cam0Path = configPath + "/" + cam0Calib;
printf("cam calib path: %s\n", cam0Path.c_str());
// 工厂函数，可以看看effective c++
m_camera = camodocal::CameraFactory::instance()->generateCameraFromYamlFile(cam0Path.c_str());
```

4，载入历史的pose graph

```
LOAD_PREVIOUS_POSE_GRAPH = fsSettings["load_previous_pose_graph"];
VINS_RESULT_PATH = VINS_RESULT_PATH + "/vio_loop.csv";
std::ofstream fout(VINS_RESULT_PATH, std::ios::out);
fout.close();
fsSettings.release();
if (LOAD_PREVIOUS_POSE_GRAPH)
{
   printf("load pose graph\n");
   m_process.lock();
   posegraph.loadPoseGraph();
   m_process.unlock();
   printf("load pose graph finish\n");
   load_flag = 1;
}
else
{
   printf("no previous pose graph\n");
   load_flag = 1;
}
```

5，接受和发布一些消息（相比mono，fusion增加了点云）

```
ros::Subscriber sub_vio = n.subscribe("/vins_estimator/odometry", 2000, vio_callback);
ros::Subscriber sub_image = n.subscribe(IMAGE_TOPIC, 2000, image_callback);
ros::Subscriber sub_pose = n.subscribe("/vins_estimator/keyframe_pose", 2000, pose_callback);
ros::Subscriber sub_extrinsic = n.subscribe("/vins_estimator/extrinsic", 2000,
extrinsic_callback);
ros::Subscriber sub_point = n.subscribe("/vins_estimator/keyframe_point", 2000, point_callback);
ros::Subscriber sub_margin_point = n.subscribe("/vins_estimator/margin_cloud", 2000,
margin_point_callback);

pub_match_img = n.advertise<sensor_msgs::Image>("match_image", 1000);
pub_camera_pose_visual = n.advertise<visualization_msgs::MarkerArray>("camera_pose_visual",
1000);
pub_point_cloud = n.advertise<sensor_msgs::PointCloud>("point_cloud_loop_rect", 1000);
pub_margin_cloud = n.advertise<sensor_msgs::PointCloud>("margin_cloud_loop_rect", 1000);
pub_odometry_rect = n.advertise<nav_msgs::Odometry>("odometry_rect", 1000);
```

6，进入pose的主循环

```
std::thread measurement_process;
std::thread keyboard_command_process;
measurement_process = std::thread(process);
keyboard_command_process = std::thread(command);
ros::spin();
```

分别有两个函数，分别是process和command（这两个函数应该是并行执行的）。首先分析command这个函数。

18，进入pose的process函数

该函数分成如下的几步：

1，初始化

```
sensor_msgs::ImageConstPtr image_msg = NULL;
sensor_msgs::PointCloudConstPtr point_msg = NULL;
nav_msgs::Odometry::ConstPtr pose_msg = NULL;
```

2，确保输入数据，并确保输入数据时间戳同步

```
// find out the messages with same time stamp
m_buf.lock();
if(!image_buf.empty() && !point_buf.empty() && !pose_buf.empty())
{
   // 这些buf都是先进先出的队列，front元素是最先进来的，back元素是刚刚进来的
   // 你可以想象，这些buf都是一群人在排队，或者自己在草=草稿纸上画画看
   // pose_buf发过来了先到了，丢掉
   if (image_buf.front()->header.stamp.toSec() > pose_buf.front()->header.stamp.toSec())
   {
       pose_buf.pop();
       printf("throw pose at beginning\n");
    }
    // point_buf提前发过来了先到了，丢掉
    else if (image_buf.front()->header.stamp.toSec() > point_buf.front()->header.stamp.toSec())
    {
       point_buf.pop();
       printf("throw point at beginning\n");
    }
    // 如果pose_buf比img_buf和point_buf都慢的话，就把多余的img和point丢掉
    else if (image_buf.back()->header.stamp.toSec() >= pose_buf.front()->header.stamp.toSec()
      && point_buf.back()->header.stamp.toSec() >= pose_buf.front()->header.stamp.toSec())
    {
        pose_msg = pose_buf.front();
        pose_buf.pop();
        while (!pose_buf.empty())
          pose_buf.pop();
        while (image_buf.front()->header.stamp.toSec() < pose_msg->header.stamp.toSec())
          image_buf.pop();
        image_msg = image_buf.front();
        image_buf.pop();
        while (point_buf.front()->header.stamp.toSec() < pose_msg->header.stamp.toSec())
           point_buf.pop();
        point_msg = point_buf.front();
        point_buf.pop();
     }
  }
m_buf.unlock();
```

3，跳过前面的几帧

```
if (skip_first_cnt < SKIP_FIRST_CNT)
{
   skip_first_cnt++;
   continue;
}
if (skip_cnt < SKIP_CNT)
{
   skip_cnt++;
   continue;
}
else
{
   skip_cnt = 0;
}
```

4，载入当前图像

```
cv_bridge::CvImageConstPtr ptr;
if (image_msg->encoding == "8UC1")
{
   sensor_msgs::Image img;
   img.header = image_msg->header;
   img.height = image_msg->height;
   img.width = image_msg->width;
   img.is_bigendian = image_msg->is_bigendian;
   img.step = image_msg->step;
   img.data = image_msg->data;
   img.encoding = "mono8";
   ptr = cv_bridge::toCvCopy(img, sensor_msgs::image_encodings::MONO8);
}
else
   ptr = cv_bridge::toCvCopy(image_msg, sensor_msgs::image_encodings::MONO8);
cv::Mat image = ptr->image;
```

5，载入当前时刻的相机位姿

```
Vector3d T = Vector3d(pose_msg->pose.pose.position.x,
                      pose_msg->pose.pose.position.y,
                      pose_msg->pose.pose.position.z);
Matrix3d R = Quaterniond(pose_msg->pose.pose.orientation.w,
                         pose_msg->pose.pose.orientation.x,
                         pose_msg->pose.pose.orientation.y,
                         pose_msg->pose.pose.orientation.z).toRotationMatrix();
```

6，加入关键帧

```
// 引入关键帧的条件很简单，位移距离超过一定的幅度，就需要引入关键帧
if((T - last_t).norm() > SKIP_DIS)
{
    vector<cv::Point3f> point_3d;
```

```
    vector<cv::Point2f> point_2d_uv;
    vector<cv::Point2f> point_2d_normal;
    vector<double> point_id;
    for (unsigned int i = 0; i < point_msg->points.size(); i++)
    {
        cv::Point3f p_3d;
        p_3d.x = point_msg->points[i].x;
        p_3d.y = point_msg->points[i].y;
        p_3d.z = point_msg->points[i].z;
        point_3d.push_back(p_3d);
        cv::Point2f p_2d_uv, p_2d_normal;
        double p_id;
        p_2d_normal.x = point_msg->channels[i].values[0];
        p_2d_normal.y = point_msg->channels[i].values[1];
        p_2d_uv.x = point_msg->channels[i].values[2];
        p_2d_uv.y = point_msg->channels[i].values[3];
        p_id = point_msg->channels[i].values[4];
        point_2d_normal.push_back(p_2d_normal);
        point_2d_uv.push_back(p_2d_uv);
        point_id.push_back(p_id);
    }
    KeyFrame* keyframe = new KeyFrame(pose_msg->header.stamp.toSec(), frame_index, T, R, image,
                                      point_3d, point_2d_uv, point_2d_normal, point_id, sequence);

    m_process.lock();
    start_flag = 1;
    posegraph.addKeyFrame(keyframe, 1);
    m_process.unlock();
    frame_index++;
    last_t = T;
}
```

这一段代码中有两个语句非常重要，分别是Keyframe的添加构造，以及posegraph的添加构造。

再接下来的章节就要详细叙述它们。