

# A Rotation-Translation-Decoupled Solution for Robust and Efficient Visual-Inertial Initialization

## 1.0 Notations and Preliminaries

估计流程

- 认为旋转外参数已知。
- 利用视觉约束紧耦合估计陀螺仪 bias，并重新预积分角速度得到旋转  $\mathbf{R}_1^{\mathbf{b}_0}, \mathbf{R}_n^{\mathbf{b}_0}$ 。
- 松耦合/紧耦合估计初始化速度  $\mathbf{v}_{\mathbf{b}_0}^{\mathbf{b}_0}$  和重力向量  $\mathbf{g}^{\mathbf{b}_0}$ ，并积分得到多帧速度  $\mathbf{v}_{\mathbf{b}_0}^{\mathbf{b}_1} \dots \mathbf{v}_{\mathbf{b}_0}^{\mathbf{b}_n}$  和平移  $\mathbf{p}_{\mathbf{b}_0}^{\mathbf{b}_1} \dots \mathbf{p}_{\mathbf{b}_0}^{\mathbf{b}_n}$ 。
- 求解世界坐标系  $\mathbf{w}$  和初始 body 系  $\mathbf{b}_0$  之间的旋转矩阵，并将轨迹对齐到世界坐标系。
- 三角化视觉landmarks。

IMU 积分公式

$$\begin{aligned}\mathbf{p}_{b_1 b_j} &= \mathbf{p}_{b_1 b_i} + \mathbf{v}_{b_1}^{b_i} \Delta t_{ij} - \frac{1}{2} \mathbf{g}^{b_1} \Delta t_{ij}^2 + \mathbf{R}_{b_1 b_i} \alpha_{b_j}^{b_i} \\ \mathbf{v}_{b_j}^{b_1} &= \mathbf{v}_{b_i}^{b_1} - \mathbf{g}^{b_1} \Delta t_{ij} + \mathbf{R}_{b_1 b_i} \beta_{b_j}^{b_i} \\ \mathbf{R}_{b_1 b_j} &= \mathbf{R}_{b_1 b_i} \gamma_{b_j}^{b_i}\end{aligned}\tag{1}$$

其中预积分项

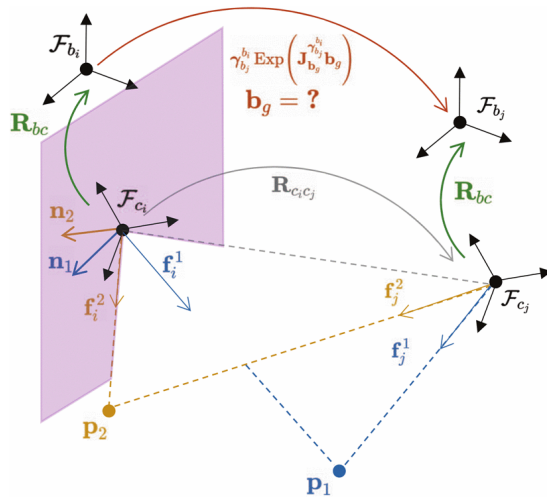
$$\begin{aligned}\alpha_{b_j}^{b_i} &= \sum_{k=i}^{j-1} \left( \left( \sum_{f=i}^{k-1} \mathbf{R}_{b_i b_f} \mathbf{a}_f^m \Delta t \right) \Delta t + \frac{1}{2} \mathbf{R}_{b_i b_k} \mathbf{a}_k^m \Delta t^2 \right) \\ \beta_{b_j}^{b_i} &= \sum_{k=i}^{j-1} \mathbf{R}_{b_i b_k} \mathbf{a}_k^m \Delta t \\ \gamma_{b_j}^{b_i} &= \prod_{k=i}^{j-1} \text{Exp}(\omega_k^m \Delta t)\end{aligned}\tag{2}$$

## 2.0 Gyroscope Bias Optimizer

### 2.1 Build Gyroscope Bias Opt Model

在上面的预积分方程中并没有考虑 bias 对测量的影响，由于重力向量和加速度 bias 是耦合在一起的，在没有运动激励的场景下是比较难解耦的，论文中忽略加速度 bias 在初始化阶段的影响，这样做也不会特别影响到初始化结果。因此在该文章中认为初始化加速度零偏为 0。而陀螺 bias 对旋转预积分量的影响可以用一阶泰勒展开近似表示：

$$\hat{\gamma}_{b_j}^{b_i} = \gamma_{b_j}^{b_i} \text{Exp} \left( \mathbf{J}_{\mathbf{b}_g}^{\gamma_{b_j}^{b_i}} \mathbf{b}_g \right)\tag{3}$$



如图所示，如果一个3D点  $\mathbf{p}_1$  能够被两个相机观测到，两个相机的光心  $\mathcal{F}_{c_i}$  和  $\mathcal{F}_{c_j}$  和 3D 点可以构成一个极平面。定义  $\mathbf{f}_i^1$  和  $\mathbf{f}_j^1$  为从  $\mathcal{F}_{c_i}$   $\mathcal{F}_{c_j}$  到  $\mathbf{p}_1$  的单位向量。则极平面的法向量可以通过叉乘计算， $\mathbf{n}^k = [\mathbf{f}_i^k]_{\times} \mathbf{R}_{c_i c_j} \mathbf{f}_j^k$ 。同时所有极平面的法向量都垂直于平移向量，如果我们在两个图像中有  $n$  个匹配点，那么将他们放在一起可以构成  $\mathbf{N} = [\mathbf{n}^1 \dots \mathbf{n}^n]$ ，同时共面在代数上等价于最小化  $\mathbf{M} = \mathbf{N}\mathbf{N}^T$  的特征值，使其逼近于 0。最终关于  $\mathbf{R}_{c_i c_j}$  的优化问题可以建模成：

$$\begin{aligned} \mathbf{R}_{c_i c_j}^* &= \arg \min_{\mathbf{R}_{c_i c_j}} \lambda_{\mathbf{M}_{ij}, \min} \\ \text{with } \mathbf{M}_{ij} &= \sum_{k=1}^n (\lfloor \mathbf{f}_i^k \rfloor \times \mathbf{R}_{c_i c_j} \mathbf{f}_j^k) (\lfloor \mathbf{f}_i^k \rfloor \times \mathbf{R}_{c_i c_j} \mathbf{f}_j^k)^\top \end{aligned} \quad (4)$$

$\lambda_{\mathbf{M}_{ij}}$  表示的是  $\mathbf{M}_{ij}$  的最小特征值。

在已知外参的情况下可以将相机之间的相对位姿表示在 body 系下面:

$$\begin{aligned}\mathbf{R}_{c_i c_j} &= \mathbf{R}_{bc}^\top \mathbf{R}_{b_i b_j} \mathbf{R}_{bc} \\ \mathbf{p}_{c_i c_j} &= \mathbf{R}_{bc}^\top (\mathbf{p}_{b_i b_j} + \mathbf{R}_{b_i b_j} \mathbf{p}_{bc} - \mathbf{p}_{bc})\end{aligned}\quad (5)$$

则将上式带入  $\mathbf{R}_{c_i c_j}$  的优化方程中可得:

$$\mathbf{M}'_{ij} = \sum_{k=1}^n \left( \lfloor \mathbf{f}_i^k \rfloor \times \mathbf{R}_{bc}^\top \gamma_{b_j}^{b_i} \text{Exp} \left( \mathbf{J}_{\mathbf{b}_g}^{b_i} \mathbf{b}_g \right) \mathbf{R}_{bc} \mathbf{f}_j^k \right) \left( \lfloor \mathbf{f}_i^k \rfloor \times \mathbf{R}_{bc}^\top \gamma_{b_j}^{b_i} \text{Exp} \left( \mathbf{J}_{\mathbf{b}_g}^{b_i} \mathbf{b}_g \right) \mathbf{R}_{bc} \mathbf{f}_j^k \right)^\top \quad (6)$$

注意到这里面只有  $b_g$  是未知的待估计参数。下面对该式进行化简。

定义  $\mathbf{R} = \mathbf{R}_{bc}^\top \gamma_{b_j}^{b_i}$  :

$$\mathbf{M}'_{ij} = \sum_{k=1}^n \left( [\mathbf{f}_i^k]_{\times} \mathbf{R} \text{Exp} \left( \mathbf{J}_{\mathbf{b}_g}^{b_i} \mathbf{b}_g \right) \mathbf{R}_{bc} \mathbf{f}_j^k \right) \left( [\mathbf{f}_i^k]_{\times} \mathbf{R} \text{Exp} \left( \mathbf{J}_{\mathbf{b}_g}^{b_i} \mathbf{b}_g \right) \mathbf{R}_{bc} \mathbf{f}_j^k \right)^{\top}$$

因为  $\mathbf{M}'_{ij}$  包含向量的旋转和叉乘, 可以用  $[\mathbf{f}^k_i]_{\times} \mathbf{R} = \mathbf{R} [\mathbf{R}^{\top} \mathbf{f}^k_i]_{\times}$  去进一步化简。

$$\mathbf{M}'_{ij} = \sum_{k=1}^n \left( \mathbf{R} [\mathbf{R}^\top \mathbf{f}_i^k]_{\times} \text{Exp} \left( \mathbf{J}_{\mathbf{b}_g}^{b_i} \mathbf{b}_g \right) \mathbf{R}_{bc} \mathbf{f}_j^k \right) \left( \mathbf{R} [\mathbf{R}^\top \mathbf{f}_i^k]_{\times} \text{Exp} \left( \mathbf{J}_{\mathbf{b}_g}^{b_i} \mathbf{b}_g \right) \mathbf{R}_{bc} \mathbf{f}_j^k \right)^\top$$

定义  $\mathbf{f}_i^{k'} = \mathbf{R}^\top \mathbf{f}_i^k$ , and  $\mathbf{f}_j^{k'} = \mathbf{R}_{bc} \mathbf{f}_j^k$

$$\mathbf{M}'_{ij} = \sum_{k=1}^n \left( \mathbf{R} \begin{bmatrix} \mathbf{f}_i^{k'} \end{bmatrix}_{\times} \text{Exp} \left( \mathbf{J}_{\mathbf{b}_g}^{\gamma_{b_j}} \mathbf{b}_g \right) \mathbf{f}_j^{k'} \right) \left( \mathbf{R} \begin{bmatrix} \mathbf{f}_i^{k'} \end{bmatrix}_{\times} \text{Exp} \left( \mathbf{J}_{\mathbf{b}_g}^{\gamma_{b_j}} \mathbf{b}_g \right) \mathbf{f}_j^{k'} \right)^{\top}$$

$\mathbf{R}$  是正交矩阵，不影响特征值大小， $\mathbf{R}$  可以忽略。

让  $\mathcal{E}$  表示观察到足够共同特征的关键帧对的集合。因为陀螺 bias 随时间变动比较慢，因此可以认为在初始化过程中是一个常数。最终问题建模如下：

$$\begin{aligned} \mathbf{b}_g^* &= \arg \min_{\mathbf{b}_g} \lambda \\ \text{with } \lambda &= \sum_{(i,j) \in \mathcal{E}} \lambda_{\mathbf{M}'_{ij}, \min} \end{aligned} \quad (8)$$

同时在优化的时候，陀螺 bias 其实是非常小的，因此可以设置初值为 0。

最小特征值的计算和  $\mathbf{M}$  矩阵的构建在论文中是直接省略的，但是这两个部分却反而是在程序中最为核心的部分，因此来进行具体的细节分析，首先根据上面公式可以得到  $\mathbf{M}$  矩阵是一个  $3 \times 3$  的对称矩阵，其中每个元素分别计算如下，参考文献 [Direct Optimization of Frame-to-Frame Rotation](#)。

$$\begin{aligned} m_{11} &= \mathbf{r}_3 \left( \sum_{i=1}^n f_{yi}^2 \mathbf{F}'_i \right) \mathbf{r}_3^T - \mathbf{r}_3 \left( 2 \sum_{i=1}^n f_{yi} f_{zi} \mathbf{F}'_i \right) \mathbf{r}_2^T + \mathbf{r}_2 \left( \sum_{i=1}^n f_{zi}^2 \mathbf{F}'_i \right) \mathbf{r}_2^T \\ m_{22} &= \mathbf{r}_1 \left( \sum_{i=1}^n f_{zi}^2 \mathbf{F}'_i \right) \mathbf{r}_1^T - \mathbf{r}_1 \left( 2 \sum_{i=1}^n f_{xi} f_{zi} \mathbf{F}'_i \right) \mathbf{r}_3^T + \mathbf{r}_3 \left( \sum_{i=1}^n f_{xi}^2 \mathbf{F}'_i \right) \mathbf{r}_3^T \\ m_{33} &= \mathbf{r}_2 \left( \sum_{i=1}^n f_{xi}^2 \mathbf{F}'_i \right) \mathbf{r}_2^T - \mathbf{r}_1 \left( 2 \sum_{i=1}^n f_{xi} f_{yi} \mathbf{F}'_i \right) \mathbf{r}_2^T + \mathbf{r}_1 \left( \sum_{i=1}^n f_{yi}^2 \mathbf{F}'_i \right) \mathbf{r}_1^T \\ m_{12} &= m_{21} = \mathbf{r}_1 \left( \sum_{i=1}^n f_{yi} f_{zi} \mathbf{F}'_i \right) \mathbf{r}_3^T - \mathbf{r}_1 \left( \sum_{i=1}^n f_{zi}^2 \mathbf{F}'_i \right) \mathbf{r}_2^T \\ &\quad - \mathbf{r}_3 \left( \sum_{i=1}^n f_{xi} f_{yi} \mathbf{F}'_i \right) \mathbf{r}_3^T + \mathbf{r}_3 \left( \sum_{i=1}^n f_{xi} f_{zi} \mathbf{F}'_i \right) \mathbf{r}_2^T \\ m_{13} &= m_{31} = \mathbf{r}_2 \left( \sum_{i=1}^n f_{xi} f_{yi} \mathbf{F}'_i \right) \mathbf{r}_3^T - \mathbf{r}_2 \left( \sum_{i=1}^n f_{xi} f_{zi} \mathbf{F}'_i \right) \mathbf{r}_2^T \\ &\quad - \mathbf{r}_1 \left( \sum_{i=1}^n f_{yi}^2 \mathbf{F}'_i \right) \mathbf{r}_3^T + \mathbf{r}_1 \left( \sum_{i=1}^n f_{yi} f_{zi} \mathbf{F}'_i \right) \mathbf{r}_2^T \\ m_{23} &= m_{32} = \mathbf{r}_1 \left( \sum_{i=1}^n f_{xi} f_{zi} \mathbf{F}'_i \right) \mathbf{r}_2^T - \mathbf{r}_1 \left( \sum_{i=1}^n f_{zi} f_{yi} \mathbf{F}'_i \right) \mathbf{r}_1^T \\ &\quad - \mathbf{r}_3 \left( \sum_{i=1}^n f_{xi}^2 \mathbf{F}'_i \right) \mathbf{r}_2^T + \mathbf{r}_3 \left( \sum_{i=1}^n f_{xi} f_{yi} \mathbf{F}'_i \right) \mathbf{r}_1^T \end{aligned}$$

观察可以发现求和符号中一共有6种不同的  $f_i f_j$  组合，我们对其用以颜色表示可得：

$$\begin{aligned}
m_{11} &= \mathbf{r}_3 \left( \sum_{i=1}^n f_{yi}^2 \mathbf{F}'_i \right) \mathbf{r}_3^T - \mathbf{r}_3 \left( 2 \sum_{i=1}^n f_{yi} f_{zi} \mathbf{F}'_i \right) \mathbf{r}_2^T + \mathbf{r}_2 \left( \sum_{i=1}^n f_{zi}^2 \mathbf{F}'_i \right) \mathbf{r}_2^T \\
m_{22} &= \mathbf{r}_1 \left( \sum_{i=1}^n f_{zi}^2 \mathbf{F}'_i \right) \mathbf{r}_1^T - \mathbf{r}_1 \left( 2 \sum_{i=1}^n f_{xi} f_{zi} \mathbf{F}'_i \right) \mathbf{r}_3^T + \mathbf{r}_3 \left( \sum_{i=1}^n f_{xi}^2 \mathbf{F}'_i \right) \mathbf{r}_3^T \\
m_{33} &= \mathbf{r}_2 \left( \sum_{i=1}^n f_{xi}^2 \mathbf{F}'_i \right) \mathbf{r}_2^T - \mathbf{r}_1 \left( 2 \sum_{i=1}^n f_{xi} f_{yi} \mathbf{F}'_i \right) \mathbf{r}_2^T + \mathbf{r}_1 \left( \sum_{i=1}^n f_{yi}^2 \mathbf{F}'_i \right) \mathbf{r}_1^T \\
\\
m_{12} = m_{21} &= \mathbf{r}_1 \left( \sum_{i=1}^n f_{yi} f_{zi} \mathbf{F}'_i \right) \mathbf{r}_3^T - \mathbf{r}_1 \left( \sum_{i=1}^n f_{zi}^2 \mathbf{F}'_i \right) \mathbf{r}_2^T \\
&\quad - \mathbf{r}_3 \left( \sum_{i=1}^n f_{xi} f_{yi} \mathbf{F}'_i \right) \mathbf{r}_3^T + \mathbf{r}_3 \left( \sum_{i=1}^n f_{xi} f_{zi} \mathbf{F}'_i \right) \mathbf{r}_2^T \\
m_{13} = m_{31} &= \mathbf{r}_2 \left( \sum_{i=1}^n f_{xi} f_{yi} \mathbf{F}'_i \right) \mathbf{r}_3^T - \mathbf{r}_2 \left( \sum_{i=1}^n f_{xi} f_{zi} \mathbf{F}'_i \right) \mathbf{r}_2^T \\
&\quad - \mathbf{r}_1 \left( \sum_{i=1}^n f_{yi}^2 \mathbf{F}'_i \right) \mathbf{r}_3^T + \mathbf{r}_1 \left( \sum_{i=1}^n f_{yi} f_{zi} \mathbf{F}'_i \right) \mathbf{r}_2^T \\
m_{23} = m_{32} &= \mathbf{r}_1 \left( \sum_{i=1}^n f_{xi} f_{zi} \mathbf{F}'_i \right) \mathbf{r}_2^T - \mathbf{r}_1 \left( \sum_{i=1}^n f_{zi} f_{yi} \mathbf{F}'_i \right) \mathbf{r}_1^T \\
&\quad - \mathbf{r}_3 \left( \sum_{i=1}^n f_{xi}^2 \mathbf{F}'_i \right) \mathbf{r}_2^T + \mathbf{r}_3 \left( \sum_{i=1}^n f_{xi} f_{yi} \mathbf{F}'_i \right) \mathbf{r}_1^T
\end{aligned}$$

同时对于对称矩阵闭式求解最小特征值目前网上或者教材中并没有相关资料，具体是在 CVPR2021 一篇论文中提到 [Rotation-Only Bundle Adjustment](#) 中的公式 [19-25] 进行了结论总结（注：并没有证明，证明在作者的supplement 材料中，但是网络上并没有开放）。

where  $\lambda_{\mathbf{M}}(\mathbf{R}_{jk})$  is the smallest eigenvalue of  $\mathbf{M}_{jk}$  (which is a function of  $\mathbf{R}_{jk}$ ). This eigenvalue can be obtained in closed form:

$$\begin{aligned}
b_1 &= -m_{11} - m_{22} - m_{33}, \\
b_2 &= -m_{13}^2 - m_{23}^2 - m_{12}^2 \\
&\quad + m_{11}m_{22} + m_{11}m_{33} + m_{22}m_{33}, \\
b_3 &= m_{22}m_{13}^2 + m_{11}m_{23}^2 + m_{33}m_{12}^2 \\
&\quad - m_{11}m_{22}m_{33} - 2m_{12}m_{23}m_{13}, \\
s &= 2b_1^3 - 9b_1b_2 + 27b_3 \\
t &= 4(b_1^2 - 3b_2)^3 \\
k &= (\sqrt{t}/2)^{1/3} \cos \left( \arccos \left( s/\sqrt{t} \right) / 3 \right), \\
\lambda_{\mathbf{M}}(\mathbf{R}_{jk}) &= (-b_1 - 2k)/3.
\end{aligned}$$

## 2.2 Code Analyze

程序的特征点提取和跟踪部分借用的VINS，在求解陀螺 bias 时使用的 ceres 的自动求导。首先该优化问题的输入为所有关键帧的共视观测，和相邻关键帧之间的IMU预积分量，在程序实现中对关键帧的选择条件是时间，即固定时间段选择一帧作为关键帧。程序中设置的最大关键帧的数量是10。从100帧中取出10帧作为关键帧构建优化问题。

- 首先取出两个相邻关键帧之间的跟踪到的点的归一化坐标。

```

/// 遍历当前滑窗中所有的 landmark，找出这两帧图像之间的跟踪到的点
for (const auto &pts: SFMConstruct) {
    /// 如果该 landmark 有在这两个时间戳对应图像上的观测，也就是说同时被这两帧观测到
    if (pts.second.obs.find(target1_tid) != pts.second.obs.end() &&
        pts.second.obs.find(target2_tid) != pts.second.obs.end()) {
        ++num_obs;
        /// 归一化平面坐标
        fis.push_back(pts.second.obs.at(target1_tid).normalpoint);
        fjs.push_back(pts.second.obs.at(target2_tid).normalpoint);
    }
}

```

- 然后取出预积分观测

```

/// 这两帧图像之间对应的预积分量
vio::IMUPreintegrated imu1 = imu_meas[i];

```

- 构建 ceres 残差块

```

///自动求导
ceres::CostFunction *eigensolver_cost_function =
BiasSolverCostFuncutor::Create(fis, fjs,

    Eigen::Quaterniond(Rbc_),

    imu1);
problem.AddResidualBlock(eigensolver_cost_function, loss_function,
biasg.data());

```

这里我们需要着重对 `BiasSolverCostFuncutor` 进行分析。先看其模板参数和接收的参数

```

static ceres::CostFunction *
Create(const std::vector<Eigen::Vector3d> &bearings1,
       const std::vector<Eigen::Vector3d> &bearings2,
       const Eigen::Quaterniond &qic,
       const vio::IMUPreintegrated &integratePtr) {
    return (new ceres::AutoDiffCostFunction<BiasSolverCostFuncutor, 1, 3>(
        new BiasSolverCostFuncutor(bearings1, bearings2, qic, integratePtr)));
}

```

通过模板参数可知残差的维度为1，优化变量的维度为3，残差就是最小特征值，优化变量为 bg。接收的四个参数分别是：

- 前一帧图像上的特征点。
- 后一帧图像上的特征点，和第一个参数一一对应。
- 第三个参数是外参。
- 第四个则是两帧之间的预积分量。

接下来看其成员变量

```

/// 这里的这 6 个 3x3 的矩阵块就是前面构建M矩阵时所标记颜色的那几个矩阵块
Eigen::Matrix3d xxF_ = Eigen::Matrix3d::Zero();
Eigen::Matrix3d yyF_ = Eigen::Matrix3d::Zero();
Eigen::Matrix3d zzF_ = Eigen::Matrix3d::Zero();
Eigen::Matrix3d xyF_ = Eigen::Matrix3d::Zero();
Eigen::Matrix3d yzF_ = Eigen::Matrix3d::Zero();
Eigen::Matrix3d xzF_ = Eigen::Matrix3d::Zero();
/// 旋转预积分量对 bg 的导数
Eigen::Matrix3d jacobina_q_bg;
/// 旋转预积分量
Eigen::Quaterniond qjk_imu;
/// 外参
Eigen::Quaterniond _qic;

```

构造函数分别首先对 jacobina\_q\_bg, qjk\_imu 赋值。然后遍历所有特征点，计算 6 个对应的矩阵用于后面的残差计算时对M矩阵进行构建。注意这里要先将观测根据外参转换到body系下。

```

BiasSolverCostFunctor(const std::vector<Eigen::Vector3d> &bearings1,
                     const std::vector<Eigen::Vector3d> &bearings2,
                     const Eigen::Quaterniond &qic,
                     const vio::IMUPreintegrated &integrate) : _qic(qic) {
    jacobina_q_bg = integrate.JRg_;
    qjk_imu = integrate.dR_.unit_quaternion();
    Eigen::Quaterniond qcjk = _qic.inverse() * qjk_imu;
    for (int i = 0; i < bearings1.size(); i++) {
        Eigen::Vector3d f1 = bearings1[i].normalized();
        f1 = qcjk.inverse() * f1;    // f1'
        Eigen::Vector3d f2 = bearings2[i].normalized();
        f2 = _qic * f2;              // f2'

        Eigen::Matrix3d F = f2 * f2.transpose();

        double weight = 1.0;
        xxF_ = xxF_ + weight * f1[0] * f1[0] * F;
        yyF_ = yyF_ + weight * f1[1] * f1[1] * F;
        zzF_ = zzF_ + weight * f1[2] * f1[2] * F;
        xyF_ = xyF_ + weight * f1[0] * f1[1] * F;
        yzF_ = yzF_ + weight * f1[1] * f1[2] * F;
        xzF_ = xzF_ + weight * f1[0] * f1[2] * F;
    }
}

```

- 然后就是最核心的残差计算部分，这里是ceres的自动求导必须重载()运算符。

```

template<typename T>
bool operator()(const T *const parameter, T *residual) const {
    Eigen::Map<const Eigen::Matrix<T, 3, 1>> deltaBg(parameter);

    // 论文中 equation (3)
    Eigen::Matrix<T, 3, 1> jacobian_bg = jacobina_q_bg.cast<T>() * deltaBg;

    Eigen::Matrix<T, 4, 1> qij_tmp;
    /// 得到旋转增量（扰动）
    ceres::AngleAxisToQuaternion(jacobian_bg.data(), qij_tmp.data());
    Eigen::Quaternion<T> qij(qij_tmp(0), qij_tmp(1), qij_tmp(2), qij_tmp(3));
    /// 对旋转使用 cayley 参数表示，只有三个变量
    Eigen::Matrix<T, 3, 1> cayley = Quaternion2Cayley<T>(qij);
}

```

```

Eigen::Matrix<T, 1, 3> jacobian;
/// 计算最小特征值，在该函数中会构建 M 矩阵同时闭式求解最小特征值，因为使用的自动求
导，所以这里的 jacobian 其实是用不上的？
T EV = opengv::GetSmallestEVwithJacobian(
    xxF_, yyF_, zzF_, xyF_, yzF_, xzF_, cayley, jacobian);
residual[0] = EV;

return true;
}

```

- GetSmallestEVwithJacobian(), ComposeMwithJacobians()。在ComposeMwithJacobians中结合前面计算 M 的公式，比较清晰。至于其中关于jacobian 的计算也可以直接根据式子得到，需要注意下刚开始的三个矩阵表示的意义，假设

$$cayley = [x, y, z]^T$$

cayley到R的转换公式为：

$R$	expersion	R_jac1	R_jac2	R_jac3
$R(0,0)$	$1 + x^2 - y^2 - z^2$	$2x$	$-2y$	$-2z$
$R(0,1)$	$2(xy - z)$	$2y$	$2x$	$-2$
$R(0,2)$	$2(xz + y)$	$2z$	$2$	$2x$
$R(1,0)$	$2(xy + z)$	$2y$	$2x$	$2$
$R(1,1)$	$1 - x^2 + y^2 - z^2$	$-2x$	$2y$	$-2z$
$R(1,2)$	$2(yz - x)$	$-2$	$2z$	$2y$
$R(2,0)$	$2(xz - y)$	$2z$	$-2$	$2x$
$R(2,1)$	$2(yz + x)$	$2$	$2z$	$2y$
$R(2,2)$	$1 - x^2 - y^2 + z^2$	$-2x$	$-2y$	$2z$

对应下面的程序

```

R_jac1(0, 0) = 2 * cayley[0];
R_jac1(0, 1) = 2 * cayley[1];
R_jac1(0, 2) = 2 * cayley[2];
R_jac1(1, 0) = 2 * cayley[1];
R_jac1(1, 1) = -2 * cayley[0];
R_jac1(1, 2) = -2;
R_jac1(2, 0) = 2 * cayley[2];
R_jac1(2, 1) = 2;
R_jac1(2, 2) = -2 * cayley[0];

R_jac2(0, 0) = -2 * cayley[1];
R_jac2(0, 1) = 2 * cayley[0];
R_jac2(0, 2) = 2;
R_jac2(1, 0) = 2 * cayley[0];
R_jac2(1, 1) = 2 * cayley[1];
R_jac2(1, 2) = 2 * cayley[2];
R_jac2(2, 0) = -2;

```

```

R_jac2(2, 1) = 2 * cayley[2];
R_jac2(2, 2) = -2 * cayley[1];

R_jac3(0, 0) = -2 * cayley[2];
R_jac3(0, 1) = -2;
R_jac3(0, 2) = 2 * cayley[0];
R_jac3(1, 0) = 2;
R_jac3(1, 1) = -2 * cayley[2];
R_jac3(1, 2) = 2 * cayley[1];
R_jac3(2, 0) = 2 * cayley[0];
R_jac3(2, 1) = 2 * cayley[1];
R_jac3(2, 2) = 2 * cayley[2];

```

我们以其中一个为例，比如  $m_{11}$  也就是  $R(0, 0)$ 。

$$m_{11} = \mathbf{r}_3 \left( \sum_{i=1}^n f_{yi}^2 \mathbf{F}'_i \right) \mathbf{r}_3^T - \mathbf{r}_3 \left( 2 \sum_{i=1}^n f_{yi} f_{zi} \mathbf{F}'_i \right) \mathbf{r}_2^T + \mathbf{r}_2 \left( \sum_{i=1}^n f_{zi}^2 \mathbf{F}'_i \right) \mathbf{r}_2^T$$

这里的  $\mathbf{r}_i$  代表的是矩阵的第  $i$  行。公式中求和的部分我们在构造函数中已经计算过了，所以化简上式可得

$$m_{11} = \underbrace{\mathbf{r}_3 \mathbf{F}_{yy} \mathbf{r}_3^T}_A - \underbrace{2 \mathbf{r}_3 \mathbf{F}_{yz} \mathbf{r}_2^T}_B + \underbrace{\mathbf{r}_2 \mathbf{F}_{zz} \mathbf{r}_2^T}_C$$

这里的  $\mathbf{F}_{yy}, \mathbf{F}_{yz}, \mathbf{F}_{zz}$  对应程序中的 `yyF`, `yzF`, `zzF`。首先计算  $m_{11}$  位置的元素值

```

double temp;
/// A
temp = R.row(2) * yyF * R.row(2).transpose();
M(0, 0) = temp;
/// B
temp = -2.0 * R.row(2) * yzF * R.row(1).transpose();
M(0, 0) += temp;
/// C
temp = R.row(1) * zzF * R.row(1).transpose();
M(0, 0) += temp;

```

$m_{11}$  的计算中涉及到了如下复合函数，假设  $\text{cayley} = C$  表示

$$\begin{aligned}
m_{11} &= F(R) \\
R &= G(C) \\
m_{11} &= F(G(C))
\end{aligned}$$

$R$  是九维度的，关于  $C$  的导数已经前置计算完毕，分别存储在三个矩阵中，每个矩阵表示对  $C$  一个轴的导数。则  $m_{11}$  对  $C_x$  的导数为

$$\frac{\partial m_{11}}{\partial C_x} = \underbrace{2 \frac{\partial \mathbf{R}}{\partial C_x} \mathbf{r}_3 \mathbf{F}_{yy} \mathbf{r}_3^T}_{J_A} - \underbrace{2 \frac{\partial \mathbf{R}}{\partial C_x} \mathbf{r}_3 \mathbf{F}_{yz} \mathbf{r}_2^T}_{J_B} - \underbrace{2 \mathbf{r}_3 \mathbf{F}_{yz} \frac{\partial \mathbf{R}}{\partial C_x} \mathbf{r}_2^T}_{J_C} + \underbrace{2 \frac{\partial \mathbf{R}}{\partial C_x} \mathbf{r}_2 \mathbf{F}_{zz} \mathbf{r}_2^T}_{J_D}$$

其中例如  $\frac{\partial \mathbf{R}}{\partial C_x} \mathbf{r}_3$  对应到函数中的 `R_jac1.row(2)`。



```

/// J_A
temp = 2.0 * R_jac1.row(2) * yyF * R.row(2).transpose();
M_jac1(0, 0) = temp;
/// J_B
temp = -2.0 * R_jac1.row(2) * yzF * R.row(1).transpose();
M_jac1(0, 0) += temp;
/// J_C
temp = -2.0 * R.row(2) * yzF * R_jac1.row(1).transpose();
M_jac1(0, 0) += temp;
/// J_D
temp = 2.0 * R_jac1.row(1) * zzF * R.row(1).transpose();
M_jac1(0, 0) += temp;

```

接下来应该计算关于  $C_y, C_z$  的导数，放入到矩阵 `M_jac2` 以及 `M_jac3` 对应元素中，这些部分按照这个思路应该也能推到出来，就不放了。

- 接下来就是通过闭式解计算  $\mathbf{M}$  的最小特征值，同时计算 jacobian，这部分还没有做证明（没有旋转之类的应该不是太麻烦），而且使用的是自动求导，所以就先搁置了。再回顾一下闭式解的公式：

$$\begin{aligned}
 b_1 &= -m_{11} - m_{22} - m_{33}, \\
 b_2 &= -m_{13}^2 - m_{23}^2 - m_{12}^2 \\
 &\quad + m_{11}m_{22} + m_{11}m_{33} + m_{22}m_{33}, \\
 b_3 &= m_{22}m_{13}^2 + m_{11}m_{23}^2 + m_{33}m_{12}^2 \\
 &\quad - m_{11}m_{22}m_{33} - 2m_{12}m_{23}m_{13}, \\
 s &= 2b_1^3 - 9b_1b_2 + 27b_3 \\
 t &= 4(b_1^2 - 3b_2)^3 \\
 k &= (\sqrt{t}/2)^{1/3} \cos\left(\arccos\left(s/\sqrt{t}\right)/3\right), \\
 \lambda_{\mathbf{M}}(\mathbf{R}_{jk}) &= (-b_1 - 2k)/3.
 \end{aligned}$$

这部分程序实现如果不看 jacobian 部分的话比较直观

```

/// 对应 b1
double b = -M(0, 0) - M(1, 1) - M(2, 2);
/// 对应 b2
double c = -pow(M(0, 2), 2) - pow(M(1, 2), 2) - pow(M(0, 1), 2) +
    M(0, 0) * M(1, 1) + M(0, 0) * M(2, 2) + M(1, 1) * M(2, 2);
/// 对应 b3
double d = M(1, 1) * pow(M(0, 2), 2) + M(0, 0) * pow(M(1, 2), 2) + M(2, 2) *
    pow(M(0, 1), 2) -
    M(0, 0) * M(1, 1) * M(2, 2) - 2 * M(0, 1) * M(1, 2) * M(0, 2);
/// 对应 s
double s = 2 * pow(b, 3) - 9 * b * c + 27 * d;
/// 对应 t
double t = 4 * pow((pow(b, 2) - 3 * c), 3);
/// 求 k
double alpha = acos(s / sqrt(t));
double beta = alpha / 3;
double y = cos(beta);
double r = 0.5 * sqrt(t);
double k = w * y;
/// 求解
double smallestEV = (-b - 2 * k) / 3;

```

## **2.2 Velocity And Gravity Estimator**

---

### **2.2.1 Tightly-Coupled Solution**

### **2.2.2 Loosely-Coupled Solution**

## **3.0 Reference**

---

[1] Direct Optimization of Frame-to-Frame Rotation (ICRA)

[2] Rotation-Only Bundle Adjustment (CVPR)