Rotation-Translation-Decoupled Initialization Solution for VIO

> 核心贡献：提出一种不需要 3D 点的鲁棒VIO初始化方法。

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 $\mathbf{R}_{\mathbf{1}}^{\mathbf{b_0}}, \mathbf{R}_{\mathbf{n}}^{\mathbf{b_0}}$。
- 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 $\mathbf{v}_{\mathbf{b_0}}^{\mathbf{b_1}} \cdots \mathbf{v}_{\mathbf{b_0}}^{\mathbf{b_n}}$ 和平移 $\mathbf{p}_{\mathbf{b_0}}^{\mathbf{b_1}} \cdots \mathbf{p}_{\mathbf{b_0}}^{\mathbf{b_n}}$ 。
- 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

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

其中预积分项

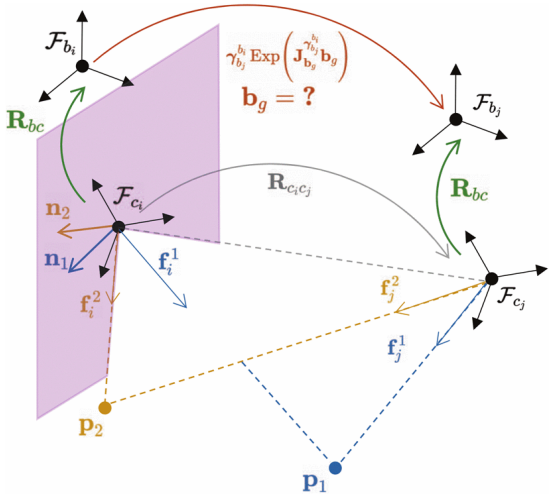
$$
\begin{aligned}
\alpha_{b_j}^{b_i} &= \sum_{k=i}^{j-1} \left(\left(\sum_{f=i}^{k-1} \mathbf{R}_{b_i b_f} \mathbf{a}_f^m \Delta t \right) \Delta t + \frac{1}{2} \mathbf{R}_{b_i b_k} \mathbf{a}_k^m \Delta t^2 \right) \\
\beta_{b_j}^{b_i} &= \sum_{k=i}^{j-1} \mathbf{R}_{b_i b_{k_i}} \mathbf{a}_k^m \Delta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

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:

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



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，$\mathbf{n}^k = \lfloor \mathbf{f}_i^k \rfloor_\times \mathbf{R}_{c_ic_j}\mathbf{f}_j^k$。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 $\mathbf{N} = \left[\mathbf{n}^1 \dots \mathbf{n}^n\right]$，同时共面在代数上等价于最小化 $\mathbf{M} = \mathbf{N}\mathbf{N}^\top$ 的特征值，使其逼近于 0。最终关于 $\mathbf{R}_{c_ic_j}$ 的优化问题可以建模成：

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

$\lambda_{\mathbf{M}_{ij}}$ 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：

$$\begin{aligned}\mathbf{R}_{c_ic_j} &= \mathbf{R}_{bc}^\top \mathbf{R}_{b_ib_j}\mathbf{R}_{bc} \\ \mathbf{p}_{c_ic_j} &= \mathbf{R}_{bc}^\top(\mathbf{p}_{b_ib_j} + \mathbf{R}_{b_ib_j}\mathbf{p}_{bc} - \mathbf{p}_{bc})\end{aligned} \tag{5}$$

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

$$\begin{aligned}\mathbf{M}'_{ij} = \sum_{k=1}^{n} &\left(\lfloor \mathbf{f}_i^k \rfloor_\times \mathbf{R}_{bc}^\top \gamma_{b_j}^{b_i} \mathrm{Exp}\left(\mathbf{J}_{\mathbf{b}_g}^{\gamma_{b_j}^{b_i}}\mathbf{b}_g\right)\mathbf{R}_{bc}\mathbf{f}_j^k \right) \\ &\left(\lfloor \mathbf{f}_i^k \rfloor_\times \mathbf{R}_{bc}^\top \gamma_{b_j}^{b_i} \mathrm{Exp}\left(\mathbf{J}_{\mathbf{b}_g}^{\gamma_{b_j}^{b_i}}\mathbf{b}_g\right)\mathbf{R}_{bc}\mathbf{f}_j^k \right)^\top \end{aligned} \tag{6}$$

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

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

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

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

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

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

$\mathbf{R}$ 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：

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

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 3×3 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。

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

$$m_{22} = \mathbf{r}_1 \left(\sum_{i=1}^n f_{zi}^2 \mathbf{F}'_i\right) \mathbf{r}_1^T - \mathbf{r}_1 \left(2\sum_{i=1}^n f_{xi} f_{zi} \mathbf{F}'_i\right) \mathbf{r}_3^T + \mathbf{r}_3 \left(\sum_{i=1}^n f_{xi}^2 \mathbf{F}'_i\right) \mathbf{r}_3^T$$

$$m_{33} = \mathbf{r}_2 \left(\sum_{i=1}^n f_{xi}^2 \mathbf{F}'_i\right) \mathbf{r}_2^T - \mathbf{r}_1 \left(2\sum_{i=1}^n f_{xi} f_{yi} \mathbf{F}'_i\right) \mathbf{r}_2^T + \mathbf{r}_1 \left(\sum_{i=1}^n f_{yi}^2 \mathbf{F}'_i\right) \mathbf{r}_1^T$$

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

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

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

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

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

$$m_{22} = \mathbf{r}_1(\sum_{i=1}^{n} f_{zi}^2 \mathbf{F}_i')\mathbf{r}_1^T - \mathbf{r}_1(2\sum_{i=1}^{n} f_{xi}f_{zi}\mathbf{F}_i')\mathbf{r}_3^T + \mathbf{r}_3(\sum_{i=1}^{n} f_{xi}^2 \mathbf{F}_i')\mathbf{r}_3^T$$

$$m_{33} = \mathbf{r}_2(\sum_{i=1}^{n} f_{xi}^2 \mathbf{F}_i')\mathbf{r}_2^T - \mathbf{r}_1(2\sum_{i=1}^{n} f_{xi}f_{yi}\mathbf{F}_i')\mathbf{r}_2^T + \mathbf{r}_1(\sum_{i=1}^{n} f_{yi}^2 \mathbf{F}_i')\mathbf{r}_1^T$$

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

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

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

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 =
BiasSolverCostFunctor::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 `BiasSolverCostFunctor` 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<BiasSolverCostFunctor, 1, 3>(
        new BiasSolverCostFunctor(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;    // fj'
        Eigen::Vector3d f2 = bearings2[i].normalized();
        f2 = _qic * f2;            // fk'

        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 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

$$m_{11} = \underbrace{\mathbf{r}_3 \mathbf{F}_{\mathbf{yy}} \mathbf{r}_3^T}_{A} \underbrace{-2\mathbf{r}_3 \mathbf{F}_{\mathbf{yz}} \mathbf{r}_2^T}_{B} \underbrace{+\mathbf{r}_2 \mathbf{F}_{\mathbf{zz}} \mathbf{r}_2^T}_{C}$$

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 $cayley = C$表示

$$m_{11} = F(R)$$
$$R = G(C)$$
$$m_{11} = F(G(C))$$

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

$$\frac{\partial dm_{11}}{\partial dC_x} = \underbrace{2\frac{\partial \mathbf{R}}{\partial \mathbf{C_x}}_{\mathbf{3}} \mathbf{F}_{\mathbf{yy}} \mathbf{r}_3^T}_{J_A} \underbrace{-2\frac{\partial R}{\partial C_x}_{3} \mathbf{F}_{\mathbf{yz}} \mathbf{r}_2^T}_{J_B} \underbrace{-2\mathbf{r}_3 \mathbf{F}_{\mathbf{yz}} \frac{\partial R}{\partial C_x}_{2}}_{J_C} \underbrace{+2\frac{\partial \mathbf{R}}{\partial \mathbf{C_x}}_{\mathbf{2}} \mathbf{F}_{\mathbf{yy}} \mathbf{r}_3^T}_{J_D}$$

其中例如 $\frac{\partial \mathbf{R}}{\partial \mathbf{C_x}}_{\mathbf{3}}$ 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：

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

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;
```

3.0 Velocity and Gravity Estimator

无论是松耦合还是紧耦合初始化都需要计算出旋转和平移，前面已经计算出了取出零偏的旋转，接下来则是平移的求解。

3.1 Analytically Solve for Translation

这部分的实现是参考文献 [3] 中的相关证明，前20个公式对贺博论文中提到的 LIGT 约束进行了详细推导。现假定有三个关键帧，序号分别表示为 r, i, l。则 LiGT约束可以表示为：

$$\mathbf{B}\mathbf{p}_{c_1c_r} + \mathbf{C}\mathbf{p}_{c_1c_i} + \mathbf{D}\mathbf{p}_{c_1c_l} = 0, \quad 1 \leq i \leq n, i \neq l$$

其中：

$$\begin{aligned}
\mathbf{B} &= \left\lfloor \mathbf{f}_i^k \right\rfloor_\times \mathbf{R}_{c_ic_l} \mathbf{f}_l^k \boldsymbol{a}_{lr}^\top \mathbf{R}_{c_rc_1} \\
\mathbf{C} &= \theta_{lr}^2 \left\lfloor \mathbf{f}_i^k \right\rfloor_\times \mathbf{R}_{c_ic_1} \\
\mathbf{D} &= -(\mathbf{B} + \mathbf{C}) \\
\boldsymbol{a}_{lr}^\top &= \left(\left\lfloor \mathbf{R}_{c_rc_l} \mathbf{f}_l^k \right\rfloor_\times \mathbf{f}_r^k \right)^\top \left\lfloor \mathbf{f}_r^k \right\rfloor_\times \\
\theta_{lr} &= \left\| \left\lfloor \mathbf{f}_r^k \right\rfloor_\times \mathbf{R}_{c_rc_l} \mathbf{f}_l^k \right\|
\end{aligned}$$

最终构建的线性方程为

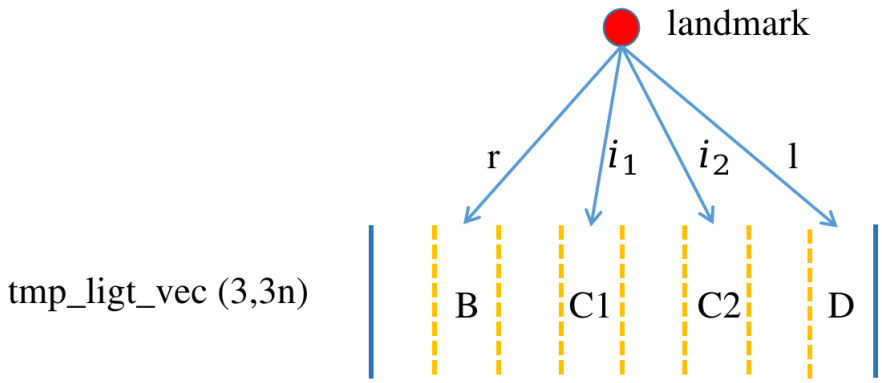
$$L \cdot t = 0$$

在实际中为了保证方程可解，一般会左乘一个 L^T，变成了如下形式，记为 LTL。

$$L^T L \cdot t = 0$$

现在来分析一个 LTL 的大小，假设现在有 n 个关键帧，landmark 中被超过3帧观测到的个数为 m。则除去参考帧 c_1外，构成 L 矩阵的维度应该会是 $[3, 3(n-1)]$。则 LTL 的维度为 $[3n-3, 3n-3]$。

假设现在有一个 landmark 被多个（多于3个）关键帧观测到，我们首先据特定条件调出两个备选关键帧作为 r, l，然后遍历其所有的关键帧上的观测，除了 l 其他的均可以作为 i，来构建 L 矩阵，其中每一个 i 可以提供一个 BCD 约束。具体情境如下所示：



该landmark能够被四个关键帧看到，选择了其中两个作为参考，分别记录为 r, l然后计算出相应的 B, C_i, D放入对应位置。只不过这里是L，程序中是$L^T L$因此还要放入LLT矩阵对应位置。构建完成之后直接求解即可得到每个关键帧相对于第一帧的位移。

程序中求解后直接赋值

```
/// evectors 为求解结果
for (int i = 0; i < local_active_frames.size(); i++) {
    rotation[i] = frame_rot.at(int_frameid2_time_frameid.at(i)) *
Rbc_.transpose();
    position[i] = evectors.middleRows<3>(3 * i);
}
```

3.2 Loosely-Coupled Solution

贺博论文中结论表示松耦合的方法基本上在大部分情况下均是最优，因此首先基于松耦合进行求解分析。在得到旋转和平移之后就可以求解每一帧的速度和重力以及尺度因子。这里的建模方式和 VINS-Mono 类似。

> Define $\mathcal{X}$ as a vector of initial state variables, $\mathbf{v}_{b_n}^{b_n}$ means the velocity of body in $\mathcal{F}_{b_n}$, $\mathbf{g}^{c_0}$ is the gravity in $\mathcal{F}_{c_0}$ and s is the metric scale.

$$\mathcal{X} = \left[\mathbf{v}_{b_0}^{b_0}, \mathbf{v}_{b_1}^{b_1}, \cdots, \mathbf{v}_{b_n}^{b_n}, s, \mathbf{g}^{c_0}\right] \in \mathbb{R}^{3(n+1)+1+3}$$

根据预积分观测有：

$$\begin{aligned}
\hat{\boldsymbol{\alpha}}_{b_k}^{b_i} &= \mathbf{R}_{b_i c_0}\left(s\left(\mathbf{p}_{c_0 b_k} - \mathbf{p}_{c_0 b_i}\right) + \frac{1}{2}\mathbf{g}^{c_0}\Delta t_{ik}^2\right) - \mathbf{v}_{b_i}^{b_i}\Delta t_{ik} \\
\hat{\boldsymbol{\beta}}_{b_k}^{b_i} &= \mathbf{R}_{b_i c_0}\left(\mathbf{R}_{c_0 b_k}\mathbf{v}_{b_k}^{b_k} + \mathbf{g}^{c_0}\Delta t_{ik}\right) - \mathbf{v}_{b_i}^{b_i}
\end{aligned}$$

将 $s\mathbf{p}_{c_0 b_k} = s\mathbf{p}_{c_0 c_k} - \mathbf{R}_{c_0 b_k}\mathbf{p}_{bc}$，带入上面：

$$\begin{aligned}
\hat{\boldsymbol{\alpha}}_{b_k}^{b_i} &= \mathbf{R}_{b_i c_0}\left(\left(s\mathbf{p}_{c_0 c_k} - \mathbf{R}_{c_0 b_k}\mathbf{p}_{bc} - \left(s\mathbf{p}_{c_0 c_i} - \mathbf{R}_{c_0 b_i}\mathbf{p}_{bc}\right)\right) + \frac{1}{2}\mathbf{g}^{c_0}\Delta t_{ik}^2\right) - \mathbf{v}_{b_i}^{b_i}\Delta t_{ik} \\
\hat{\boldsymbol{\beta}}_{b_k}^{b_i} &= \mathbf{R}_{b_i c_0}\left(\mathbf{R}_{c_0 b_k}\mathbf{v}_{b_k}^{b_k} + \mathbf{g}^{c_0}\Delta t_{ik}\right) - \mathbf{v}_{b_i}^{b_i}
\end{aligned}$$

化简得：

$$\begin{aligned}
\hat{\boldsymbol{\alpha}}_{b_k}^{b_i} &= \mathbf{R}_{b_i c_0}\left(s\left(\mathbf{p}_{c_0 c_k} - \mathbf{p}_{c_0 c_i}\right) - \mathbf{R}_{c_0 b_k}\mathbf{p}_{bc} + \mathbf{R}_{c_0 b_i}\mathbf{p}_{bc} + \frac{1}{2}\mathbf{g}^{c_0}\Delta t_{ik}^2\right) - \mathbf{v}_{b_i}^{b_i}\Delta t_{ik} \\
&= s\mathbf{R}_{b_i c_0}\left(\mathbf{p}_{c_0 c_k} - \mathbf{p}_{c_0 c_i}\right) - \mathbf{R}_{b_i c_0}\mathbf{R}_{c_0 b_k}\mathbf{p}_{bc} + \mathbf{p}_{bc} + \frac{1}{2}\mathbf{R}_{b_i c_0}\mathbf{g}^{c_0}\Delta t_{ik}^2 - \mathbf{v}_{b_i}^{b_i}\Delta t_{ik} \\
\hat{\boldsymbol{\beta}}_{b_k}^{b_i} &= \mathbf{R}_{b_i c_0}\left(\mathbf{R}_{c_0 b_k}\mathbf{v}_{b_k}^{b_k} + \mathbf{g}^{c_0}\Delta t_{ik}\right) - \mathbf{v}_{b_i}^{b_i} \\
&= \mathbf{R}_{b_i c_0}\mathbf{R}_{c_0 b_k}\mathbf{v}_{b_k}^{b_k} + \mathbf{R}_{b_i c_0}\mathbf{g}^{c_0}\Delta t_{ik} - \mathbf{v}_{b_i}^{b_i}
\end{aligned}$$

将待估计变量放到方程右边，有：

$$\mathbf{H}_{b_k}^{b_i}\mathcal{X}_i = \hat{\mathbf{z}}_{b_k}^{b_i}$$

$$\mathbf{H}_{b_k}^{b_i} = \begin{bmatrix} -\mathbf{I}\Delta t_{ik} & \mathbf{0} & \mathbf{R}_{b_i c_0}\left(\mathbf{p}_{c_k}^{c_0} - \mathbf{p}_{c_i}^{c_0}\right) & \frac{1}{2}\mathbf{R}_{b_i c_0}\Delta t_{ik}^2 \\ -\mathbf{I} & \mathbf{R}_{b_i c_0}\mathbf{R}_{c_0 b_k} & \mathbf{0} & \mathbf{R}_{b_i c_0}\Delta t_{ik} \end{bmatrix}$$

$$\hat{\mathbf{z}}_{b_k}^{b_i} = \begin{bmatrix} \hat{\boldsymbol{\alpha}}_{b_i b_k} - \mathbf{p}_{bc} + \mathbf{R}_{b_i c_0}\mathbf{R}_{c_0 b_k}\mathbf{p}_{bc} \\ \hat{\boldsymbol{\beta}}_{b_i b_k} \end{bmatrix}$$

H 为 6×10的矩阵，z 为6×1的矩阵，对应程序中

```
MatrixXd tmp_A(6, 10);
tmp_A.setZero();
VectorXd tmp_b(6);
tmp_b.setZero();
double dt = imu_meas[i].sum_dt_;
tmp_A.block<3, 3>(0, 0) = -dt * Matrix3d::Identity();
tmp_A.block<3, 1>(0, 6) = rotation[i].transpose() * (position[j] - position[i]) /
100.0;
tmp_A.block<3, 3>(0, 7) = rotation[i].transpose() * dt * dt / 2 *
Matrix3d::Identity() * G.norm();
tmp_A.block<3, 3>(3, 0) = -Matrix3d::Identity();
tmp_A.block<3, 3>(3, 3) = rotation[i].transpose() * rotation[j];
tmp_A.block<3, 3>(3, 7) = rotation[i].transpose() * dt * Matrix3d::Identity() *
G.norm();
tmp_b.block<3, 1>(0, 0) = imu_meas[i].dP_ + rotation[i].transpose() * rotation[j]
* pbc_ - pbc_;
tmp_b.block<3, 1>(3, 0) = imu_meas[i].dV_;
```

程序中在对平移进行计算时除了100，是为了保持数值稳定性，只要在求解得到的结果中也除以100就行了，还有要注意的是这里在关于重力的部分乘了模长，也就是说求解出来的向量关于重力部分是单位向量，只有方向约束，要将结果也乘以G的模长。这里是其中两帧的约束，遍历所有关键帧构建大矩阵即可。

由于重力的模长为定值，在求解该方程的时候同时加入了重力向量的模长约束，因此求解该问题是一个带约束的优化问题，程序中是利用的拉格朗日乘子法进行的求解。将该约束展开

$$
\begin{aligned}
\mathcal{C}(x) &= \sum_{k\in\mathcal{K}} \|r(x)\| \\
&= (\mathbf{Ax}-\mathbf{b})^\top(\mathbf{Ax}-\mathbf{b}) \\
&= \sum_{k\in\mathcal{K}} \left(\mathbf{x}^\top\mathbf{A}^\top\mathbf{Ax} - 2\mathbf{b}^\top\mathbf{Ax} + \mathbf{b}^\top\mathbf{b}\right) \\
&= \mathbf{x}^\top\mathbf{Mx} + \mathbf{m}^\top\mathbf{x} + Q
\end{aligned}
$$

未知参数中的重力约束可以写作:

$$
\mathbf{x}^\top\mathbf{Wx} = G^2, \mathbf{W} = \begin{bmatrix} \mathbf{0} & \mathbf{0}^\top \\ \mathbf{0} & \mathbf{I}_{3\times3} \end{bmatrix}
$$

最后我们要求解的方程为:

$$
\begin{aligned}
&\mathcal{X}^* = \arg\min_{\mathbf{x}} = \mathbf{x}^\top\mathbf{Mx} + \mathbf{m}^\top\mathbf{x} \\
&\text{subject to } \mathbf{x}^\top\mathbf{Wx} = \mathrm{G}^2
\end{aligned}
$$

利用Lagrange乘子法，能够得到:

$$
\mathcal{L}(\mathbf{x}) = \mathbf{x}^\top\mathbf{Mx} + \mathbf{m}^\top\mathbf{x} + \lambda\left(\mathbf{x}^\top\mathbf{Wx} - G^2\right)
$$

对该公式求导:

$$
\frac{\partial\mathcal{L}(\mathbf{x})}{\partial\mathbf{x}} = 2\mathbf{x}^\top\mathbf{M} + \mathbf{m}^\top + 2\lambda\mathbf{x}^\top\mathbf{W} = \mathbf{0}^\top
$$

将 $\mathbf{x} = -(2\mathbf{M}+2\lambda\mathbf{W})^{-\top}\mathbf{m}$ 反代回 $\mathbf{x}^\top\mathbf{Wx} = G^2$:

$$
\mathbf{m}^\top(2\mathbf{M}+2\lambda\mathbf{W})^{-1}\mathbf{W}(2\mathbf{M}+2\lambda\mathbf{W})^{-\top}\mathbf{m} = G^2
$$

> 这是一个6次多项式（为什么是6次下面进行证明）求根问题，具体求解可参考 [4] 的公式 (34-45)

分块矩阵求逆

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} + A^{-1}B\big(D - CA^{-1}B\big)^{-1}CA^{-1} & -A^{-1}B\big(D - CA^{-1}B\big)^{-1} \\ -\big(D - CA^{-1}B\big)^{-1}CA^{-1} & \big(D - CA^{-1}B\big)^{-1} \end{bmatrix}$$

- **PROOF:**

将矩阵 $(2\mathbf{M} + 2\lambda\mathbf{W})$ 分成四个矩阵块:

$$2\mathbf{M} + 2\lambda\mathbf{W} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B}^\top & \mathbf{D} + 2\lambda\mathbf{I}_{3\times 3} \end{bmatrix}$$

因为 $\mathbf{W}$ 是稀疏的, 我们只需要计算 $(2\mathbf{M} + 2\lambda\mathbf{W})^{-1}$ 剩下的列，使用分块矩阵求逆:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B}^\top & \mathbf{D} + 2\lambda\mathbf{I}_{3\times 3} \end{bmatrix}^{-1} = \begin{bmatrix} \star & -\mathbf{A}^{-1}\mathbf{B}\mathbf{Q}^{-1} \\ \star & \mathbf{Q}^{-1} \end{bmatrix}$$

其中 $\mathbf{Q} = \mathbf{D} - \mathbf{B}^\top\mathbf{A}^{-1}\mathbf{B} + 2\lambda\mathbf{I}_{3\times 3} \doteq \mathbf{S} + 2\lambda\mathbf{I}_{3\times 3}$

约束问题可以化简成下面形式:

$$\boldsymbol{m}^\top\mathbf{F}m = G^2$$
$$\mathbf{F} = \begin{bmatrix} \mathbf{A}^{-1}\mathbf{B}\mathbf{Q}^{-1}\mathbf{Q}^{-\top}\mathbf{B}^\top\mathbf{A}^{-\top} & -\mathbf{A}^{-1}\mathbf{B}\mathbf{Q}^{-1}\mathbf{Q}^{-\top} \\ \text{symm} & \mathbf{Q}^{-1}\mathbf{Q}^{-\top} \end{bmatrix}$$

只有 $\mathbf{Q}^{-1}$ 依赖 λ .利用矩阵求逆伴随性质可以写成:

$$\begin{aligned} \mathbf{Q}^{-1} = (\mathbf{S} + 2\lambda\mathbf{I}_{3\times 3})^{-1} &= \frac{(\mathbf{S} + 2\lambda\mathbf{I}_{3\times 3})^{\mathrm{A}}}{p(\lambda)} \\ &= \frac{\mathbf{S}^{\mathrm{A}} + 4\lambda^2\mathbf{I}_{3\times 3} + 2\lambda\left(\mathrm{tr}(\mathbf{S})\mathbf{I}_{3\times 3} - \mathbf{S}\right)}{p(\lambda)} \end{aligned}$$

其中

$$p(\lambda) \doteq \det\left(\mathbf{S} + 2\lambda\mathbf{I}_{3\times 3}\right), \quad \mathbf{S}^{\mathrm{A}} = \det(\mathbf{S})\mathbf{S}^{-1}$$

因为 $\mathbf{Q}^{-1}$ 是对称的，有:

$$\begin{aligned} \mathbf{Q}^{-1}\mathbf{Q}^{-\top} &= \frac{\left(\mathbf{S}^{\mathrm{A}} + 4\lambda^2\mathbf{I}_{3\times 3} + 2\lambda\mathbf{U}\right)^2}{p(\lambda)^2} \\ &= \frac{16\lambda^4\mathbf{I}_{3\times 3} + 4\lambda^2\left(2\mathbf{S}^{\mathrm{A}} + \mathbf{U}^2\right)}{p(\lambda)^2} \\ &\quad + \frac{16\lambda^3\mathbf{U} + 2\lambda\left(\mathbf{S}^{\mathrm{A}}\mathbf{U} + \mathbf{U}\mathbf{S}^{\mathrm{A}}\right) + \mathbf{S}^{\mathrm{A}}}{p(\lambda)^2} \end{aligned}$$

其中

$$\mathbf{U} \doteq \mathrm{tr}(\mathbf{S})\mathbf{I}_{3\times 3} - \mathbf{S}$$

因此，最终可以写成关于 λ 的多项式:

$$
\begin{aligned}
&16\lambda^4 \boldsymbol{m}^\top \begin{bmatrix} \mathbf{A}^{-1}\mathbf{B}\mathbf{B}^\top\mathbf{A}^{-\top} & -\mathbf{A}^{-1}\mathbf{B} \\ \text{symm} & \mathbf{I}_{3\times 3} \end{bmatrix} \boldsymbol{m} \\
&+16\lambda^3 \boldsymbol{m}^\top \begin{bmatrix} \mathbf{A}^{-1}\mathbf{B}\mathbf{U}\mathbf{B}^\top\mathbf{A}^{-\top} & -\mathbf{A}^{-1}\mathbf{B}\mathbf{U} \\ \text{symm} & \mathbf{U} \end{bmatrix} \boldsymbol{m} \\
&+4\lambda^2 \boldsymbol{m}^\top \begin{bmatrix} \mathbf{A}^{-1}\mathbf{B}\mathbf{X}\mathbf{B}^\top\mathbf{A}^{-\top} & -\mathbf{A}^{-1}\mathbf{B}\mathbf{X} \\ \text{symm}^{-1} & \mathbf{X} \end{bmatrix} \boldsymbol{m} \\
&+2\lambda \boldsymbol{m}^\top \begin{bmatrix} \mathbf{A}^{-1}\mathbf{B}\mathbf{Y}\mathbf{B}^\top\mathbf{A}^{-\top} & -\mathbf{A}^{-1}\mathbf{B}\mathbf{Y} \\ \text{symm} & \mathbf{Y} \end{bmatrix} \boldsymbol{m} \\
&+\boldsymbol{m}^\top \begin{bmatrix} \mathbf{A}^{-1}\mathbf{B}\mathbf{S}^{\mathrm{A}^2}\mathbf{B}^\top\mathbf{A}^{-\top} & -\mathbf{A}^{-1}\mathbf{B}\mathbf{S}^{\mathrm{A}^2} \\ \text{symm} & \mathbf{S}^{\mathrm{A}^2} \end{bmatrix} \boldsymbol{m} \\
&= G^2 p(\lambda)^2
\end{aligned}
$$

对应到程序中gravityRefine()函数得到，程序中字母表示也和推导基本保持一致。

```
int q = M.rows() - 3;
Eigen::MatrixXd A = 2. * M.block(0, 0, q, q);
Eigen::MatrixXd Bt = 2. * M.block(q, 0, 3, q);
Eigen::MatrixXd BtAi = Bt * A.inverse();
Eigen::Matrix3d D = 2. * M.block(q, q, 3, 3);
Eigen::Matrix3d S = D - BtAi * Bt.transpose();
Eigen::Matrix3d Sa = S.determinant() * S.inverse();
Eigen::Matrix3d U = S.trace() * Eigen::Matrix3d::Identity() - S;
```

然后就是首先构建多项式左边部分的系数，这里的 `c0,c1,c2,c3,c4` 对应上面推导中的关于 λ 多项式左边部分中 $\lambda^0, \lambda^1, \lambda^2, \lambda^3, \lambda^4$ 的系数。

```
Eigen::Vector3d v1 = BtAi * m.head(q);
Eigen::Vector3d m2 = m.tail<3>();
Eigen::Matrix3d X;
Eigen::Vector3d Xm2;
const double c4 = 16. * (v1.dot(v1) - 2. * v1.dot(m2) + m2.dot(m2));
X = U;
Xm2 = X * m2;
const double c3 = 16. * (v1.dot(X * v1) - 2. * v1.dot(Xm2) + m2.dot(Xm2));
X = 2. * Sa + U * U;
Xm2 = X * m2;
const double c2 = 4. * (v1.dot(X * v1) - 2. * v1.dot(Xm2) + m2.dot(Xm2));
X = Sa * U + U * Sa;
Xm2 = X * m2;
const double c1 = 2. * (v1.dot(X * v1) - 2. * v1.dot(Xm2) + m2.dot(Xm2));
X = Sa * Sa;
Xm2 = X * m2;
const double c0 = (v1.dot(X * v1) - 2. * v1.dot(Xm2) + m2.dot(Xm2));
```

同时构建关于 λ 多项式右边部分，右边是关于 λ 的6次多项式

$$
p(\lambda)^2 \doteq \det{}^2 \left(\mathbf{S} + 2\lambda \mathbf{I}_{3\times 3}\right)
$$

这部分直接行列式展开应该就能得到如下的结果，个人并没有验证推导。

```
const double s00 = S(0, 0), s01 = S(0, 1), s02 = S(0, 2);
const double s11 = S(1, 1), s12 = S(1, 2), s22 = S(2, 2);
```

```
const double t1 = s00 + s11 + s22;
const double t2 = s00 * s11 + s00 * s22 + s11 * s22
    - std::pow(s01, 2) - std::pow(s02, 2) - std::pow(s12, 2);
const double t3 = s00 * s11 * s22 + 2. * s01 * s02 * s12
    - s00 * std::pow(s12, 2) - s11 * std::pow(s02, 2) - s22 * std::pow(s01, 2);

Eigen::VectorXd coeffs(7);
coeffs << 64.,
        64. * t1,
        16. * (std::pow(t1, 2) + 2. * t2),
        16. * (t1 * t2 + t3),
        4. * (std::pow(t2, 2) + 2. * t1 * t3),
        4. * t3 * t2,
        std::pow(t3, 2);
```

然后将等式左边的部分移动到右边同时除以 G 模长

```
const double G2i = 1. / std::pow(gravity_mag, 2);
coeffs(2) -= c4 * G2i;
coeffs(3) -= c3 * G2i;
coeffs(4) -= c2 * G2i;
coeffs(5) -= c1 * G2i;
coeffs(6) -= c0 * G2i;
```

接下来则是调用多项式求根函数进行求解，求解之后遍历每个解，找出使得代价最小的那个，即为最终解。

$$\boldsymbol{x} = -(2\mathbf{M} + 2\lambda\mathbf{W})^{-\top}\boldsymbol{m}.$$

```
Eigen::VectorXd real, imag;
if (!FindPolynomialRootsCompanionMatrix(coeffs, &real, &imag)) {
    return false;
}
/// 这里我们是要实数解
Eigen::VectorXd lambdas = real_roots(real, imag);
if (lambdas.size() == 0) {
    return false;
}

Eigen::MatrixXd W(M.rows(), M.rows());
W.setZero();
W.block<3, 3>(q, q) = Eigen::Matrix3d::Identity();

Eigen::VectorXd solution;
double min_cost = std::numeric_limits<double>::max();
for (Eigen::VectorXd::Index i = 0; i < lambdas.size(); ++i) {
    const double lambda = lambdas(i);
    /// LU 分解加快求解速度
    Eigen::FullPivLU<Eigen::MatrixXd> lu(2. * M + 2. * lambda * W);
    /// 根据上面的公式，得到 x
    Eigen::VectorXd x_ = -lu.inverse() * m;

    double cost = x_.transpose() * M * x_;
    cost += m.transpose() * x_;
    cost += Q;
```

```
    if (cost < min_cost) {
        solution = x_;
        min_cost = cost;
    }
}
```

求解出结果后进行赋值

```
gravity = x.tail(3) * G.norm(); /// 重力要乘以模长
s = x(n_state - 4) / 100.0; /// 尺度也要除以100
x(n_state - 4) = s;

for (int i = int_frameid2_time_frameid.size() - 1; i >= 0; i--) {
    position[i] = s * position[i] - rotation[i] * pbc_;
    velocity[i] = rotation[i] * x.segment<3>(i * 3);
}

Eigen::Matrix3d rot0 = rotation[0].transpose();
gravity = rot0 * gravity;
for (int i = 0; i < int_frameid2_time_frameid.size(); i++) {
    rotation[i] = rot0 * rotation[i];
    position[i] = rot0 * position[i];
    velocity[i] = rot0 * velocity[i];
}
```

3.3 Tightly-Coupled Solution

根据贺博论文，Loosely-Coupled Solution 效果明显更好。

4.0 Reference

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

[2] Rotation-Only Bundle Adjustment (CVPR)

[3] A Pose-Only Solution to Visual Reconstruction and Navigation (TPAMI)

[4] Estimator initialization in vision-aided inertial navigation with unknown camera-IMU calibration (IROS)