OKVIS 理解

Extr15

2017.12.14

1 ijrr2015 论文 [1]

机器人的状态 $x_R = [{}_W r_S, q_{WS}, {}_S v, b_g, b_a]$ 分为位姿部分 $x_T = [{}_W r_S, q_{WS}]$ 和速度偏置部分 $x_{sb} = [{}_S v, b_g, b_a]$

这篇论文中速度是表示在 sensor 坐标系 (也就是 body 坐标系)，所以下文中的有些公式跟 [2] 中不一样。

相机和 sensor 之间的外参是 $x_{C_i} = [{}_S r_{C_i}, q_{SC_i}]$

四元数的更新是

$$q_{WS} = \delta q \otimes \bar{q_{WS}} \tag{1}$$

切空间就是李代数，转换成四元数是

$$\delta q = exp\left(\begin{bmatrix} \frac{1}{2}\delta\alpha \\ 1 \end{bmatrix}\right) = \begin{bmatrix} sinc\|\frac{\delta\alpha}{2}\|\frac{\delta\alpha}{2} \\ cos\|\frac{\delta\alpha}{2}\| \end{bmatrix} \tag{2}$$

对指数映射在 $\delta\alpha = 0$ 处线性化，有

$$\delta q \approx \begin{bmatrix} \frac{1}{2}\delta\alpha \\ 1 \end{bmatrix} = q_I + \frac{1}{2}\begin{bmatrix} I_3 \\ 0_{1\times3} \end{bmatrix}\delta\alpha \tag{3}$$

协方差传递过程中的误差状态向量定义为

$$\delta x_R = [\delta p, \delta\alpha, \delta v, \delta b_g, \delta b_a]$$

1.1 重投影误差

第 j 个地图点在第 k 个图像帧的第 i 个相机 (系统可能是双目) 中的重投影误差为

$$e_r^{i,j,k} = z^{i,j,k} - h_i(T_{C_iS}^k T_{SW}^k {}_W l^j)$$

其中 h_i 表示第 i 个相机的投影方程，用 $J_{r,i}$ 表示投影方程的导数

那么 (为了简写，忽略了一些角标)(路标使用了齐次坐标)

$$T^k_{C_iS} T^k_{SW\ W} l^j = T^k_{C_iS}(R_{SW}(l^j_{1:3} - t_{ws} * l^j_4)) \tag{4}$$

所以

$$\frac{\partial e_r}{\partial \delta p} = \frac{\partial e_r}{\partial\, _W r_S} = -J_{r,i} T^k_{C_iS} R_{SW}(-l^j_4) = J_{r,i} T^k_{C_iS} C_{SW} l^j_4 \tag{5}$$

而 $R_{WS} = (1 + \lfloor\delta\alpha\rfloor_\times \bar{R}_{WS})$，$R_{SW} = \bar{R}_{SW}(1 - \lfloor\delta\alpha\rfloor_\times)$

所以 (这里应该是论文中少了一个负号，代码中是有的)

$$\frac{\partial e_r}{\partial \delta \alpha} = -J_{r,i} T^k_{C_iS} R_{SW} \lfloor _W l^j_{1:3} - t_{ws} * {_W l^j_4} \rfloor_\times \tag{6}$$

上面就是论文 [1] 中的 9 式

对应代码

Listing 1: okvis/okvis_ceres/include/okvis/ceres/implementation/ReprojectionError.hpp

```
    Eigen::Vector3d p = hp_W.head<3>() - t_WS_W * hp_W
        [3];
    Eigen::Matrix<double, 4, 6> J;
    J.setZero();
    J.topLeftCorner<3, 3>() = C_SW * hp_W[3];
    J.topRightCorner<3, 3>() = -C_SW * okvis::kinematics
        ::crossMx(p);
```

同样易得论文 [1] 中 10、11 式

$$\frac{\partial e_r}{\partial \delta X^j_L} = -J_{r,i} T^k_{C_iS} \begin{bmatrix} C_{SW} \\ 0_{1\times3} \end{bmatrix} \tag{7}$$

$$\frac{\partial e_r}{\partial \delta X^k_{C_i}} = J_{r,i} \begin{bmatrix} C^k_{C_iS} l^j_4 & -C^k_{C_iS} \lfloor _S l^j_{1:3} - {_S r_{C_i}} * {_S l^j_4} \rfloor_\times \\ 0_{1\times3} & 0_{1\times3} \end{bmatrix} \tag{8}$$

其中 $_S l^j$ 是表示在 sensor 坐标系下的路标

1.2 IMU Kinematics

见 [1] 12 式，其中速度是表示在 sensor 坐标系中，所以

$$
\begin{aligned}
{}_S\dot{v} &= (C_{SW}\dot{\ }{}_W v) \\
&= C_{SW}\,{}_W\dot{v} + \dot{C}_{SW}\,{}_W v \\
&= C_{SW}\,{}_W\dot{v} - \lfloor\omega\rfloor_\times C_{SW}\,{}_W v \\
&= C_{SW}\,{}_W\dot{v} - \lfloor\omega\rfloor_\times\,{}_S v
\end{aligned}
\tag{9}
$$

同理，[1] 15 式根据 [3] 易得。

比如 ${}_W\dot{r}_S = C_{WS}\,{}_S v$，那么

$$
\begin{aligned}
\delta\dot{r} = \dot{r} - \dot{\bar{r}} &= C_{WS}\,{}_S v - \bar{C}_{WS}\,{}_S\bar{v} \\
&= (1 + \lfloor\delta\alpha\rfloor_\times)\bar{C}_{WS}\,{}_S v - \bar{C}_{WS}\,{}_S\bar{v} \\
&= \bar{C}_{WS}({}_S v - {}_S\bar{v}) + \lfloor\delta\alpha\rfloor_\times\bar{C}_{WS}\,{}_S v \\
&\approx \bar{C}_{WS}\delta v - \lfloor\bar{C}_{WS}\,{}_S\bar{v}\rfloor_\times\delta\alpha
\end{aligned}
\tag{10}
$$

1.3 Marginalization 和 FEJ

marg 就是去掉部分变量，把对应的信息转换成保留的变量的先验，用到下一次优化之中。那么由于 ceres 的优化是不断被调用的，那么 marg 过程也就在不断的进行，marg 的过程用到的 H 矩阵就是 J^TJ，而 J 的计算是在某个线性点附近计算的。比如第一次 marg 的时候，是在 $x_0 = [x_{\mu_0}, x_{\lambda_0}]$ 处计算的 jacobian，那么第一次 marg，把 x_{μ_0} 去掉，第二次 marg，可能是把 x_{λ_0} 中的一部分 x_{μ_1} 给去掉; 那么这时对应的 jacobian 是在 x_{λ_0} 处计算，这就是 FEJ，如果是状态变量第一次 marg 后又优化过的估计 x_{λ_1} 处计算，那就是之前通常的做法。

论文中使用 FEJ,用 x_0 表示这个变量在第一次线性化时的点 (代码中是 linearizationPoint .get())，在后面很多次 ceres 的迭代优化中，有了更新，当前估计是 $\bar{x}$，其中的差是 Δx，那么每次 marg 的时候，残差 b 除了每次迭代优化结束后的之外，还应该加上由于 FEJ，即 Δx 的存在所引起的部分，即

$$
b_{new} = b_{old} + \frac{\partial b}{\partial \Delta x} * \Delta x = b_{old} + H_{old}\Delta x
\tag{11}
$$

这次又要 marg 掉一部分，即把上式拆成要 marg 和要保留的，即

$$
\begin{bmatrix} b_{\mu_1} \\ b_{\lambda_1} \end{bmatrix} = \begin{bmatrix} b_{\mu_0} \\ b_{\lambda_0} \end{bmatrix} + \begin{bmatrix} H_{\mu\mu} & H_{\mu\lambda} \\ H_{\lambda\mu} & H_{\lambda\lambda} \end{bmatrix} \begin{bmatrix} \Delta x_\mu \\ \Delta x_\lambda \end{bmatrix}
\tag{12}
$$

把上式代入到论文的 (22b) 式中有

$$
\begin{aligned}
b_\lambda^* &= b_\lambda - H_{\lambda\mu} H_{\mu\mu}^{-1} b_\mu \\
&= \begin{bmatrix} -H_{\lambda\mu} H_{\mu\mu}^{-1} & I \end{bmatrix} \begin{bmatrix} b_\mu \\ b_\lambda \end{bmatrix} \\
&= \begin{bmatrix} -H_{\lambda\mu} H_{\mu\mu}^{-1} & I \end{bmatrix} \left(\begin{bmatrix} b_{\mu_0} \\ b_{\lambda_0} \end{bmatrix} + \begin{bmatrix} H_{\mu\mu} & H_{\mu\lambda} \\ H_{\lambda\mu} & H_{\lambda\lambda} \end{bmatrix} \begin{bmatrix} \Delta x_\mu \\ \Delta x_\lambda \end{bmatrix} \right) \\
&= b_{\lambda_0} - H_{\lambda\mu} H_{\mu\mu}^{-1} b_{\mu_0} + \begin{bmatrix} 0 & H_{\lambda\lambda} - H_{\lambda\mu} H_{\mu\mu}^{-1} H_{\mu\lambda} \end{bmatrix} \begin{bmatrix} \Delta x_\mu \\ \Delta x_\lambda \end{bmatrix} \\
&= b_{\lambda,0}^* + H_{\lambda\lambda}^* \Delta x_\lambda
\end{aligned}
\tag{13}
$$

这就是残差的更新公式，其中 $H_{\lambda\lambda}^*$ 是在 FEJ 处计算的，对应到代码的MarginalizationError ::EvaluateWithMinimalJacobians中的 e = e0_ + J_ * Delta_Chi;

2 代码

很多误差实现了 EvaluateWithMinimalJacobians 函数，其实现中是先求 jacobians-Minimal, 就是误差对切空间 (最小参数化空间) 求导，然后调用 liftJacobian 得到在当前参数下的切空间对原来的参数空间的导数 J_lift，然后相乘得到 jacobians.

比如

```
Eigen::Matrix<double, 6, 7, Eigen::RowMajor> J_lift;
PoseLocalParameterization::liftJacobian(parameters
    [0], J_lift.data());

Eigen::Map<Eigen::Matrix<double, 2, 7, Eigen::
    RowMajor> > J0(jacobians[0]);
J0 = J0_minimal * J_lift;
```

而 liftJacobian 主要是在 PoseLocalParameterization.cpp 中，其实现和 okvis_kinematics /include/okvis/kinematics/implementation/Transformation.hpp 大同小异

2.1 Transformation.hpp

oplus 函数，即加上切空间的一个小量，这里输入参数 delta 是 6*1 的向量，平移量在前，旋转量在后。平移量直接相加，旋转量按 $q_1 = \delta q \otimes q_0$ 计算，即

$$\begin{aligned} q_1 = \delta q \otimes q_0 &= \begin{bmatrix} sinc\|\frac{\delta\alpha}{2}\|\frac{\delta\alpha}{2} \\ cos\|\frac{\delta\alpha}{2}\| \end{bmatrix} \otimes q_0 \\ &\approx \begin{bmatrix} \frac{1}{2}\delta\alpha \\ 1 \end{bmatrix} \otimes q_0 \\ &= [q_0]_R * \begin{bmatrix} \frac{1}{2}\delta\alpha \\ 1 \end{bmatrix} \end{aligned} \tag{14}$$

那么 oplusJacobian 就是 q_1 对 δq 求导， liftJacobian 就是 δq 对 q_1 求导。
所以 oplusJacobian 是

$$\frac{\partial q_1}{\partial \delta\alpha} = [q_0]_R * \begin{bmatrix} \frac{1}{2}I_3 \\ 0_{1\times 3} \end{bmatrix} \tag{15}$$

liftJacobian 是

$$\frac{\partial \delta\alpha}{\partial q_1} = 2 * ([q_0^{-1}]_R)_{topleft(3,4)} \tag{16}$$

对应到代码中 okvis::kinematics::oplus(q_) 就是在求四元数的右乘矩阵（okvis_kinematics/include/okvis/kinematics/operators.hpp 中，plus 函数是四元数的左乘矩阵, oplus 函数是四元数的右乘矩阵)

其他的一些代码:

把切空间增数转换成四元数增量

```
__inline__ Eigen::Quaterniond deltaQ(const Eigen::Vector3d& dAlpha)
{
  Eigen::Vector4d dq;
  double halfnorm = 0.5 * dAlpha.template tail<3>().norm();
  dq.template head<3>() = sinc(halfnorm) * 0.5 * dAlpha.template tail<3>();
  dq[3] = cos(halfnorm);
  return Eigen::Quaterniond(dq);
}
```

求 sinc, 处理当 x 很小时的情况

```
__inline__ double sinc(double x) {
  if (fabs(x) > 1e-6) {
    return sin(x) / x;
  } else {
    static const double c_2 = 1.0 / 6.0;
    static const double c_4 = 1.0 / 120.0;
    static const double c_6 = 1.0 / 5040.0;
    const double x_2 = x * x;
    const double x_4 = x_2 * x_2;
    const double x_6 = x_2 * x_2 * x_2;
    return 1.0 - c_2 * x_2 + c_4 * x_4 - c_6 * x_6;
  }
}
```

李代数的右 jacobian

```
__inline__ Eigen::Matrix3d rightJacobian(const Eigen::Vector3d & PhiVec) {
  const double Phi = PhiVec.norm();
  Eigen::Matrix3d retMat = Eigen::Matrix3d::Identity();
  const Eigen::Matrix3d Phi_x = okvis::kinematics::crossMx(PhiVec);
  const Eigen::Matrix3d Phi_x2 = Phi_x*Phi_x;
  if(Phi < 1.0e-4) {
    retMat += -0.5*Phi_x + 1.0/6.0*Phi_x2;
  } else {
    const double Phi2 = Phi*Phi;
    const double Phi3 = Phi2*Phi;
    retMat += -(1.0-cos(Phi))/(Phi2)*Phi_x + (Phi-sin(Phi))/Phi3*Phi_x2;
  }
  return retMat;
}
```

2.2 ImuError.cpp

先看 redoPreintegration，需要参考 [4]。

预积分的部分

$$\begin{aligned}\Delta\bar{v}_{ij} &= \Sigma_{k=i}^{j-1}\Delta\bar{R}_{i,k}(\tilde{a}_k - \bar{b}_i^a)\Delta t \\ &= \Delta\bar{v}_{i,j-1} + \Delta\bar{R}_{i,j-1}(\tilde{a}_{j-1} - \bar{b}_i^a)\Delta t\end{aligned} \tag{17}$$

同理

$$\begin{aligned}\Delta\bar{p}_{ij} &= \Sigma_{k=i}^{j-1}(\Delta\bar{v}_{ik}\Delta t + \frac{1}{2}\Delta\bar{R}_{i,k}(\tilde{a}_k - \bar{b}_i^a)\Delta t^2 \\ &= \Delta\bar{p}_{i,j-1} + \Delta\bar{v}_{i,j-1}\Delta t + \frac{1}{2}\Delta\bar{R}_{i,j-1}(\tilde{a}_{j-1} - \bar{b}_i^a)\Delta t^2\end{aligned} \tag{18}$$

而 $\Delta\bar{v}_{ij}$ 对应到代码中的 acc_integral_，$\Delta\bar{p}_{ij}$ 对应到代码中的 acc_doubleintegral_

即

```
Delta_q_1 = Delta_q_ * dq;
acc_S_true = (0.5 * (acc_S_0 + acc_S_1) -
    speedAndBiases.segment < 3 > (6));
acc_integral_1 = acc_integral_ + 0.5 * (C + C_1) *
    acc_S_true * dt;
acc_doubleintegral_ += acc_integral_ * dt + 0.25 * (C
    + C_1) * acc_S_true * dt * dt;
```

根据 [4] Appendix B 有 (省去了负号)

$$\begin{aligned}\frac{\partial\Delta\bar{R}_{ij}}{\partial b^g} &= \Sigma_{k=i}^{j-1}[\Delta\tilde{R}_{k+1,j}(\bar{b}_i)^T J_r^k\Delta t] \\ &= \Sigma_{k=i}^{j-1}[\Delta\bar{R}_{k+1,j}^T J_r^k\Delta t] \\ &= \Delta\bar{R}_{j-1,j}^T\frac{\partial\Delta\bar{R}_{i,j-1}}{\partial b^g} + J_r^k\Delta t\end{aligned} \tag{19}$$

其中 $J_r^k = J_r((\omega_k - \bar{b}_i^g)\Delta t)$

$\frac{\partial\Delta\bar{R}_{ij}}{\partial b^g}$ 实际上应该是 $\frac{\partial\Delta\alpha}{\partial b^g}$，对应到代码中的cross_

上式对应

```
const Eigen::Matrix3d cross_1 = dq.inverse().
    toRotationMatrix() * cross_ + okvis::kinematics::
    rightJacobian(omega_S_true * dt) * dt;
```

同理

$$
\begin{aligned}
\frac{\partial\Delta\bar{v}_{ij}}{\partial b^g} &= \Sigma_{k=i}^{j-1}\Delta\bar{R}_{i,k}\lfloor\tilde{a}_k - \bar{b}_i^a\rfloor_\times\frac{\partial\Delta\bar{R}_{ik}}{\partial b^g}\Delta t \\
&= \frac{\partial\Delta\bar{v}_{i,j-1}}{\partial b^g} + \Delta\bar{R}_{i,j-1}\lfloor\tilde{a}_{j-1} - \bar{b}_i^a\rfloor_\times\frac{\partial\Delta\bar{R}_{i,j-1}}{\partial b^g}\Delta t
\end{aligned}
\tag{20}
$$

对应

```
Eigen::Matrix3d dv_db_g_1 = dv_db_g_ + 0.5 * dt * (C
    * acc_S_x * cross_ + C_1 * acc_S_x * cross_1);
```

$$
\begin{aligned}
\frac{\partial\Delta\bar{p}_{ij}}{\partial b^g} &= \Sigma_{k=i}^{j-1}\left(\frac{\partial\Delta\bar{v}_{ik}}{\partial b^g}\Delta t + \frac{1}{2}\Delta\bar{R}_{ik}\lfloor\tilde{a}_k - \bar{b}_i^a\rfloor_\times\frac{\partial\Delta\bar{R}_{ik}}{\partial b^g}\Delta t^2\right) \\
&= \frac{\partial\Delta\bar{p}_{i,j-1}}{\partial b^g} + \frac{\partial\Delta\bar{v}_{i,j-1}}{\partial b^g}\Delta t + \frac{1}{2}\Delta\bar{R}_{i,j-1}\lfloor\tilde{a}_{j-1} - \bar{b}_i^a\rfloor_\times\frac{\partial\Delta\bar{R}_{i,j-1}}{\partial b^g}\Delta t^2
\end{aligned}
\tag{21}
$$

对应

```
dp_db_g_ += dt * dv_db_g_ + 0.25 * dt * dt * (C *
    acc_S_x * cross_ + C_1 * acc_S_x * cross_1);
```

而 $\frac{\partial\Delta\bar{v}_{ij}}{\partial b^a}$ 对应代码中 C_integral_，$\frac{\partial\Delta\bar{p}_{ij}}{\partial b^a}$ 对应代码中 C_doubleintegral_，

$$
\begin{aligned}
\frac{\partial\Delta\bar{v}_{ij}}{\partial b^a} &= \Sigma_{k=i}^{j-1}\Delta\bar{R}_{i,k}\Delta t \\
&= \frac{\partial\Delta\bar{v}_{i,j-1}}{\partial b^a} + \Delta\bar{R}_{i,j-1}\Delta t
\end{aligned}
\tag{22}
$$

对应

```
C_integral_1 = C_integral_ + 0.5 * (C + C_1) * dt;
```

$$
\begin{aligned}
\frac{\partial \Delta \bar{p}_{ij}}{\partial b^a} &= \Sigma_{k=i}^{j-1}\left(\frac{\partial \Delta \bar{v}_{ik}}{\partial b^a}\Delta t + \frac{1}{2}\Delta \bar{R}_{ik}\Delta t^2\right) \\
&= \frac{\partial \Delta \bar{p}_{i,j-1}}{\partial b^a} + \frac{\partial \Delta \bar{v}_{i,j-1}}{\partial b^a}\Delta t + \frac{1}{2}\Delta \bar{R}_{i,j-1}\Delta t^2
\end{aligned} \tag{23}
$$

对应

```
C_doubleintegral_ += C_integral_ * dt + 0.25 * (C +
    C_1) * dt * dt;
```

求 $\Delta \bar{R}_{ij}, \Delta \bar{v}_{ij}, \Delta \bar{p}_{ij}$ 对 b^a, b^g 导数的目的是为了补偿预积分项，即 [4] 44 式。而 F_delta是误差状态的协方差在 propagation 过程中的传递。

2.2.1 EvaluateWithMinimalJacobians 中残差对参数块的导数

残差为 (有些地方混用了下标 01 和 ij 的表示)

$$
\begin{aligned}
\theta_{err} &= 2(Dq \otimes q_{WS_1}^{-1} \otimes q_{WS_0})_{vec} \\
p_{err} &= C_{S_0W}\Delta P + (\Delta p_{ij} + \frac{\partial \Delta \bar{p}_{ij}}{\partial b^g}\delta b^g + \frac{\partial \Delta \bar{p}_{ij}}{\partial b^a}\delta b^a) \\
v_{err} &= C_{S_0W}\Delta V + (\Delta v_{ij} + \frac{\partial \Delta \bar{v}_{ij}}{\partial b^g}\delta b^g + \frac{\partial \Delta \bar{v}_{ij}}{\partial b^a}\delta b^a) \\
b^g_{err} &= -(b^g_j - b^g_i) \\
b^a_{err} &= -(b^a_j - b^a_i)
\end{aligned} \tag{24}
$$

其中

$$
Dq = q(-\frac{\partial \Delta \alpha}{\partial b^g}\delta b^g) \otimes \delta q_{01} \tag{25}
$$

$$
\begin{aligned}
\Delta P &= t_{WS_0} - t_{WS_1} + v_0\Delta T - \frac{1}{2}g\Delta T^2 \\
\Delta V &= v_0 - v_1 - g\Delta T \\
\delta b^g &= b^g_i - \bar{b}^g_i \\
\delta b^a &= b^a_i - \bar{b}^a_i
\end{aligned} \tag{26}
$$

这里残差是换算到了 S_0 的坐标系下进行计算的，而且由于负号已经在 $\Delta P, \Delta V$ 中取了，所以残差的计算中都是加号。

对应代码

```
error.segment<3>(0) = C_S0_W * delta_p_est_W + acc_doubleintegral_ + F0.block<3,6>(0,9)*Delta_b;
error.segment<3>(3) = 2*(Dq*(T_WS_1.q().inverse()*T_WS_0.q())).vec();
error.segment<3>(6) = C_S0_W * delta_v_est_W + acc_integral_ + F0.block<3,6>(6,9)*Delta_b;
error.tail<6>() = speedAndBiases_0.tail<6>() - speedAndBiases_1.tail<6>();
```

代码中delta_p_est_W 对应 ΔP，代码中delta_v_est_W 对应 ΔV

上面的残差分别对参数块进行求导，并利用 $C_{S_0W} = \bar{C}_{S_0W}(1 - \lfloor\theta\rfloor_\times)$，就可得到对应的代码。

举例来说，代码中 F0 表示残差对 T_WS_0, speedAndBiases_0 的最小参数化进行求导。

$$\begin{aligned}\tilde{\theta}_{err} &= 2(Dq \otimes q_{WS_1}^{-1} \otimes \tilde{q}_{WS_0})_{vec} \\ &= 2(Dq \otimes q_{WS_1}^{-1} \otimes \begin{bmatrix} \frac{1}{2}\delta\theta_{q_0} \\ 0 \end{bmatrix} \otimes q_{WS_0})_{vec} \\ &= [(Dq \otimes q_{WS_1}^{-1})_L (q_{WS_0})_R]_{3\times3} \delta\theta_{q_0}\end{aligned} \tag{27}$$

所以

Table 1: F0

$F0_{15*15}$	$0,\ p_0$	$3,\ \delta\theta_0$	$6,\ v_0$	$9,\ b_i^g$	$12,\ b_i^a$
p_{err}	C_{S0W}	$C_{S0W}\lfloor\Delta P\rfloor_\times$	$C_{S0W}\Delta T$	$\frac{\partial\Delta\bar{p}_{ij}}{\partial b^g}$	$\frac{\partial\Delta\bar{p}_{ij}}{\partial b^a}$
θ_{err}	0	$[(Dq \otimes q_{WS_1}^{-1})_L (q_{WS_0})_R]_{3\times3}$		$-\frac{\partial\Delta\alpha}{\partial b^g}[(q_{WS_1}^{-1} \otimes q_{WS_0})_R (Dq)_R]_{3\times3}$	
v_{err}		$C_{S0W}\lfloor\Delta V\rfloor_\times$	C_{S0W}	$\frac{\partial\Delta\bar{v}_{ij}}{\partial b^g}$	$\frac{\partial\Delta\bar{v}_{ij}}{\partial b^a}$
b_{err}^g				I_3	
b_{err}^a					I_3

对应代码

```
Eigen::Matrix<double,15,15> F0 = Eigen::Matrix<double,15,15>::Identity(); // holds for d/db_g, d/db_a
const Eigen::Vector3d delta_p_est_W = T_WS_0.r() - T_WS_1.r() + speedAndBiases_0.head<3>()*Delta_t - 0.5*g_W*Delta_t*Delta_t;
```

```
// _est 表示估计， _W 表示在世界坐标系下，下面算残差时会左乘C_S0_W，转换到S0坐标系下。
const Eigen::Vector3d delta_v_est_W = speedAndBiases_0.head<3>() - speedAndBiases_1.head<3>() - g_W*Delta_t;
const Eigen::Quaterniond Dq = okvis::kinematics::deltaQ(-dalpha_db_g_*Delta_b.head<3>())*Delta_q_;
 // Delta_q_是预积分，所以Dq就是校正过的预积分项
F0.block<3,3>(0,0) = C_S0_W;
F0.block<3,3>(0,3) = C_S0_W * okvis::kinematics::crossMx(delta_p_est_W);
F0.block<3,3>(0,6) = C_S0_W * Eigen::Matrix3d::Identity()*Delta_t;
F0.block<3,3>(0,9) = dp_db_g_;
F0.block<3,3>(0,12) = -C_doubleintegral_;
F0.block<3,3>(3,3) = (okvis::kinematics::plus(Dq*T_WS_1.q().inverse()) *
okvis::kinematics::oplus(T_WS_0.q())).topLeftCorner<3,3>();
F0.block<3,3>(3,9) = (okvis::kinematics::oplus(T_WS_1.q().inverse()*T_WS_0.q())*
okvis::kinematics::oplus(Dq)).topLeftCorner<3,3>()*(-dalpha_db_g_);
F0.block<3,3>(6,3) = C_S0_W * okvis::kinematics::crossMx(delta_v_est_W);
F0.block<3,3>(6,6) = C_S0_W;
F0.block<3,3>(6,9) = dv_db_g_;
F0.block<3,3>(6,12) = -C_integral_;
```

同样可得 F1

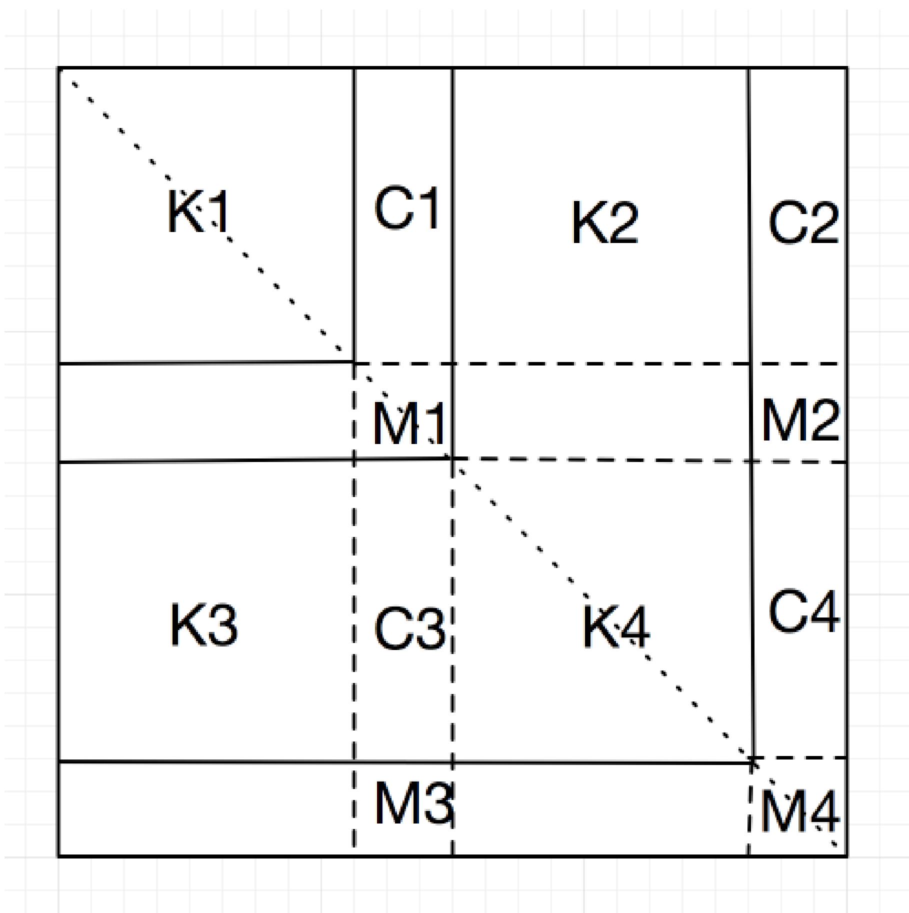
2.3 okvis/ceres/implementation/MarginalizationError.hpp

splitSymmetricMatrix 函数，把矩阵划分成要 marg 的，要保留的，和混合交叉的部分。代码中两个 for 循环都是对marginalizationStartIdxAndLengthPairs2 进行访问，如图 2.3，这里用长度为 2 示例，2*2 个循环下来，要保留的部分依次是图中的 K1,K2,K3,K4, 要 marg 的部分依次是图中的 M1, M2, M3, M4, 混合的部分依次是 C1, C2, C3, C4，那么由于对称性，有些混合的部分是去掉了的，就是图中方格

Table 2: F1

$F1_{15*15}$	0, p_1	3, $\delta\theta_1$	6, v_1	9, b_j^g	12, b_j^a
p_{err}	- C_{S0W}				
θ_{err}	0	$-[(Dq)_L(q_{WS_0})_R(q_{WS_1}^{-1})_L]_{3\times3}$			
v_{err}			$-C_{S0W}$		
b_{err}^g				$-I_3$	
b_{err}^a					$-I_3$

中没有标字母的地方。



2.4

(28)

References

[1] [ijrr2015][leutenegger2014] Keyframe-based visual–inertial odometry using nonlinear optimization.pdf

[2] [RSS2013] Keyframe-Based Visual-Inertial SLAM using Nonlinear Optimization.pdf

[3] [kinematics] Quaternion kinematics for the error-state KF.pdf

[4] [forster TRO2016] on manifold preintegration for real-time visual-inertial odometry.pdf