S-MSCKF代码笔记

WUT 王泽威

s-msckf主要由image_processor跟msckf_vio两个节点组成。
Image_processor中主要的算法位于ImageProcessor::stereoCallback()。

一 前端图像处理

image_processor::stereoCallback()接收双目图像数据，首先构建图像金字塔，用于光流跟踪、双目匹配，保存在curr_cam1_pyramid_，curr_cam1_pyramid_中，然后判断是否为第一帧。

1 视觉初始化

接收到的图像是第一帧，进行初始化initializeFirstFrame()。
initializeFirstFrame()：
(1)把左图像均匀划分成若干个格子，默认参数是把左图像划分成4x4个格子。
(2)检测左图像中的fast特征。
(3)立体匹配stereoMatch()，保留匹配成功的点。（待详细）
(4)把左右相机匹配上的点cam0_inliers和cam1_inliers分配到预先划分的格子grid_new_features里，grid_new_features涉及的数据类型是：

```
struct FeatureMetaData {
    FeatureIDType id;
    float response;
    int lifetime;//连续跟踪次数
    cv::Point2f cam0_point;//左图像坐标
    cv::Point2f cam1_point;//右图像坐标
  };

typedef std::map<int, std::vector<FeatureMetaData> > GridFeatures;
```

图像中每个格子都有一个索引，在默认4x4个格子的情况下，从左上角到右下角的索引是0~15。对每个格子中的特征点，根据其相应值response排序，取前n个，最终的结果保存在curr_features_ptr变量中。

2 图像跟踪

接收到的图像不是第一帧，进行跟踪。
这部分的执行流程为：trackFeatures() ---> addNewFeatures() ---> pruneGridFeatures()。

2.1trackFeatures()

（1）在imu数据队列中找到上一帧图像时间对应的imu数据，并用一个迭代器指向它，再在imu数据队列里找到当前图像帧时刻对应的imu数据，再设置一个迭代器指向它。计算这段时间内的imu的平均角速度${}^{I}\bar{\boldsymbol{\omega}}$，然后计算这段时间内在相机坐标系下的平均角速度：

$$
\begin{aligned}
{}^{c0}\bar{\boldsymbol{\omega}} &= {}^{I}_{c0}\mathbf{R}\,{}^{I}\bar{\boldsymbol{\omega}} \\
{}^{c1}\bar{\boldsymbol{\omega}} &= {}^{I}_{c1}\mathbf{R}\,{}^{I}\bar{\boldsymbol{\omega}}
\end{aligned}
$$

对应代码：

```
// Transform the mean angular velocity from the IMU
 // frame to the cam0 and cam1 frames.
 Vec3f cam0_mean_ang_vel = R_cam0_imu.t() * mean_ang_vel;
 Vec3f cam1_mean_ang_vel = R_cam1_imu.t() * mean_ang_vel;
```

计算旋转角度，得到当前帧相机位姿关于上一帧的旋转向量，再根据罗德里格斯公式得到旋转矩阵并转置，得到cam0_R_p_c，cam1_R_p_c。最后删除已经经过积分的这一段imu数据，释放内存。

（2）预测经过旋转后的特征点位置。根据上一步计算出来的cam0_R_p_c，cam1_R_p_c，记它们分别为${}^{p}_{c0}\mathbf{R}$,${}^{p}_{c1}\mathbf{R}$。假设camera0中某特征点i在上一帧的像素坐标的齐次向量为

$$
{}^{p}_{i}\mathbf{P} = \begin{pmatrix} u_p \\ v_p \\ 1 \end{pmatrix}
$$

则旋转后的像素坐标的齐次向量为：

$$
{}^{c}_{i}\mathbf{P} = \mathbf{K}{}^{p}_{c0}\mathbf{R}\mathbf{K}^{-1} = \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}
$$

${}^{c}_{i}\bar{\mathbf{P}}$除以$z_i$就可以得到旋转后的像素坐标:

$$
{}^{c}_{i}\bar{\mathbf{P}} = \begin{pmatrix} x_i/z_i \\ y_i/z_i \\ 1 \end{pmatrix}
$$

但是上述过程中并没有考虑图像的畸变，因此只是一种粗略的预测。

然后根据以预测的像素坐标作为初始值对做图像进行光流跟踪，去除跟踪失败的点和跟踪到图像区域外的点。

（3）去除外点（Outlier removal)。外点的去除分成三步：
a.将当前帧的左右图像的特征点进行立体匹配，去除匹配失败的点。
b.利用2-point RANSAC算法去除cam0的误跟踪点。
c.利用2-point RANSAC算法去除cam1的误跟踪点。

2-Point RANSAC:
a.首先计算2-Point RANSAC的迭代次数，根据公式：

$$
N = \frac{\log(1-p)}{\log(1-(1-\varepsilon)^s)}
$$

其中s是数据点的个数，ε是外点的比例，p是成功率。S-MSCKF中对应的代码为：

```
int iter_num = static_cast<int>(
      ceil(log(1-success_probability) / log(1-0.7*0.7)));
```

b.对前后帧对应的特征点进行去畸变undistortPoints，得到归一化坐标。再将旋转矩阵乘以上一帧特征点的归一化坐标进行旋转补偿。

c.将上一帧经过旋转补偿后的归一化点（取前2维）$\{\tilde{x}_{p_i}, \tilde{y}_{p_i}\}_n$跟当前帧的归一化点（前2维）$\{\bar{x}_{c_i}, \bar{y}_{c_i}\}_n$进行归一化rescalePoints。归一化系数的计算公式为：

$$s = \frac{2n}{\sqrt{2}(\sum_i \sqrt{\tilde{x}_{p_i}^2 + \tilde{y}_{p_i}^2} + \sum_i \sqrt{\bar{x}_{c_i}^2 + \bar{y}_{c_i}^2})}$$

然后将$\{\tilde{x}_{p_i}, \tilde{y}_{p_i}\}_n$和$\{\bar{x}_{c_i}, \bar{y}_{c_i}\}_n$乘以该系数，得到$\mathbf{P}_{s_1}, \mathbf{P}_{s_2}$。

c.$\mathbf{P}_{s_1}, \mathbf{P}_{s_2}$作差，得到$\mathbf{\Delta P}_s$。根据阈值distance和$\mathbf{\Delta P}_s$，标记内点和外点，并且计算内点的平均距离。如果内点数小于3，则把所有点标记为外点并且返回函数。这种情况在特征点较少且快速旋转的时候可能发生。

d.判断运动退化，即几乎没有平移的情况。此时RANSAC算法不能很好的工作，匹配点之间的距离几乎为0。这里用上一步计算的内点的平均距离是否大于一定值来判断有没有出现运动退化。

e.2-Point Ransac。前面的都是为这一步做准备。令$\mathbf{P}_{s_1}, \mathbf{P}_{s_2}$的齐次式为$\bar{\mathbf{P}}_{s_1}, \bar{\mathbf{P}}_{s_2}$,则存在对极约束：

$$\bar{\mathbf{P}}_{s_2}^T \mathbf{E} \bar{\mathbf{P}}_{s_1} = 0$$

其中$\mathbf{E} = \mathbf{t}_\times \mathbf{R}$。由于$\mathbf{P}_{s_1}$已经经过了旋转补偿，因此$\mathbf{R}$为单位阵，对极约束可以看成：

$$\bar{\mathbf{P}}_{s_2}^T \mathbf{t}_\times \bar{\mathbf{P}}_{s_1} = 0$$

展开后得到：

$$\begin{pmatrix} x_2 & y_2 & 1 \end{pmatrix} \begin{pmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix} = 0$$

$$\begin{pmatrix} y_1 - y_2 & -(x_1 - x_2) & x_1 y_2 - x_2 y_2 \end{pmatrix} \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} = 0$$

这跟以下代码是符合的：

```
vector<Point2d> pts_diff(pts1_undistorted.size());
  for (int i = 0; i < pts1_undistorted.size(); ++i)
    pts_diff[i] = pts1_undistorted[i] - pts2_undistorted[i];
    ...
    ...
    MatrixXd coeff_t(pts_diff.size(), 3);
  for (int i = 0; i < pts_diff.size(); ++i) {
    coeff_t(i, 0) = pts_diff[i].y;
    coeff_t(i, 1) = -pts_diff[i].x;
    coeff_t(i, 2) = pts1_undistorted[i].x*pts2_undistorted[i].y -
      pts1_undistorted[i].y*pts2_undistorted[i].x;
  }
```

2-Point RANSAC就是每次随机提取2对特征点，根据这2对特征点来求解t_x, t_y, t_z。在实际求解的过程中，把t_x, t_y, t_z其中一个置为1，然后求解剩余2个变量。至于把哪个置为1，则根据对应系数向量的一范数来选择。公式10可以看成：

$$\begin{pmatrix} a & b & c \end{pmatrix} \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} = 0$$

在2对特征点的情况下：

$$\begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{pmatrix} \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} = 0$$

比较左边矩阵各列向量的一范数大小，假设$(c_1, c_2)^T$的一范数最小，则令$t_z = 1$，只需求解t_x, t_y。如果是$(b_1, b_2)^T$的一范数最小，则令$t_y = 1$。以下推导$t_z = 1$的情况，此时公式12可以化成：

$$\begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix} \begin{pmatrix} t_x \\ t_y \end{pmatrix} = - \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$$

令左边系数矩阵为$\mathbf{A}$,右边向量为$-\mathbf{c}$,则待求结果为：

$$\mathbf{t} = \begin{pmatrix} t_x \\ t_y \end{pmatrix} = -\mathbf{A}^{-1}\mathbf{c}$$

这跟代码中的计算过程是一致的。

```
Vector2d coeff_tx(coeff_t(pair_idx1, 0), coeff_t(pair_idx2, 0));
Vector2d coeff_ty(coeff_t(pair_idx1, 1), coeff_t(pair_idx2, 1));
Vector2d coeff_tz(coeff_t(pair_idx1, 2), coeff_t(pair_idx2, 2));
...
...
A << coeff_tx, coeff_ty;
Vector2d solution = A.inverse() * (-coeff_tz);
model(0) = solution(0);
model(1) = solution(1);
model(2) = 1.0;
```

得到$\mathbf{t}$之后计算各对特征点的误差：

$$e_i = \begin{pmatrix} a_i & b_i & c_i \end{pmatrix} \begin{pmatrix} \hat{t}_x \\ \hat{t}_y \\ \hat{t}_z \end{pmatrix}$$

理想状态下上式应该为0，然而总会有误差，尤其是参数计算的不是特别准的时候。根据误差阈值划分内点（inliers）和外点（outliers）。根据内点再次求解$\mathbf{t}$，这次求的是$\mathbf{t}$的最小二乘解，同样也是将某一维置一。

循环迭代，保留内点数最多的一次结果并标记内点。

我的想法：这一步根据内点去求$\mathbf{t}$的最小二乘解我认为有点多余，因为这一步只是对函数中的临时变量this_error和best_error进行更新，而整个twoPointRansac函数要赋值的是inlier_markers，而inlier_markers又由best_inlier_set得到，这一路的变量关联为：

inlier_set ---> best_inlier_set ---> inlier_markers

而inlier_set在第一次求解$\mathbf{t}$的时候（进行求解最下二乘解之前）就已经确定，并且在本次迭代中不会受到之后的代码改变。

d.把经过上述步骤后的跟踪后的特征点按一个个格子保存到curr_features_ptr中，然后根据prev_features_ptr中的特征点数目和curr_features_ptr中的特征点数目计算跟踪成功率。

2.2 addNewFeatures

完成上一步的跟踪后，检测新的特征。

(1)创建mask扣掉curr_features_ptr中已经存在的特征，避免重复检测。

(2)在mask的基础上检测左图像中新的特征，根据特征点分配的格子存储到一个二维vector中，然后每个格子的特征点根据响应值（response）排序，去除响应值较小的特征点，代码里是去掉排在第n个之后的特征点，如果这个格子内的特征点数目不足n，就全数保留。

(3)立体匹配stereoMatch。把匹配成功的点分配到grid_new_features里。然后计算curr_features_ptr中各个格子的vacancy_num，其中vacancy_num为当前格子内的最小特征点数目grid_min_feature_num

减去当前格子内的特征点数。然后把grid_new_features中各个格子内的前vacancy_num个特征点分配到curr_features_ptr中并且赋上id。

2.3 pruneGridFeatures

pruneGridFeatures主要是为了去除连续跟踪次数较多的变量。首先将curr_features_ptr中每个格子内的特征根据连续跟踪次数(liftime)排序，如果这个格子内的特征点数目m超过grid_max_feature_num，则去除前(grid_max_feature_num - m)个跟踪次数最多的点。

2.4 发布特征点

将特征点进行去畸变，转化成归一化坐标发布出去；同时发布各阶段的跟踪数目。

二、VIO

VIO中IMU的状态向量为：

$$\mathbf{x}_I = \begin{pmatrix} {}^I_G\mathbf{q}^T & \mathbf{b}_g^T & {}^G\mathbf{v}_I^T & \mathbf{b}_a^T & {}^G\mathbf{p}_I^T & {}^I_C\mathbf{q}^T & {}^I\mathbf{p}_C^T \end{pmatrix}^T$$

误差状态为：

$$\tilde{\mathbf{x}}_I = \begin{pmatrix} {}^I_G\tilde{\boldsymbol{\theta}}^T & \tilde{\boldsymbol{b}}_g^T & {}^G\tilde{\mathbf{v}}^T & \tilde{\mathbf{b}}_a^T & {}^G\tilde{\mathbf{p}}_I^T & {}^I_C\tilde{\boldsymbol{\theta}}^T & {}^I\tilde{\mathbf{p}}_C^T \end{pmatrix}^T$$

对于普通的变量，如$\mathbf{b}_g, \mathbf{v}_I$等，满足通用的加减关系，比如${}^G\tilde{\mathbf{p}}_I = {}^G\mathbf{p}_I - {}^G\hat{\mathbf{p}}_I$。对于四元数，误差四元数为：

$$\begin{aligned} \delta\mathbf{q} &= \mathbf{q} \otimes \hat{\mathbf{q}}^{-1} \\ &\approx \begin{pmatrix} \frac{1}{2}{}^G_I\tilde{\boldsymbol{\theta}}^T & 1 \end{pmatrix}^T \end{aligned}$$

其中${}^G_I\tilde{\boldsymbol{\theta}}^T \in \mathbb{R}^3$表示一个微小的旋转，通过这种方式将旋转误差降到了三维。考虑最终的误差状态向量，设有N个相机状态，则最终的误差状态向量为：

$$\tilde{\mathbf{x}} = \begin{pmatrix} \tilde{\mathbf{x}}_I^T & \tilde{\mathbf{x}}_{C_1} & \dots & \tilde{\mathbf{x}}_{C_N} \end{pmatrix}^T$$

$$\tilde{\mathbf{x}}_{C_i} = \begin{pmatrix} {}^{C_i}_G\tilde{\boldsymbol{\theta}}^T & {}^G\tilde{\mathbf{p}}_{C_i}^T \end{pmatrix}^T$$

3 VIO初始化

(1) 初始化imu各噪声项，存在state_server.continuous_noise_cov中；

```
double IMUState::gyro_noise = 0.001;
double IMUState::acc_noise = 0.01;
double IMUState::gyro_bias_noise = 0.001;
double IMUState::acc_bias_noise = 0.01;
```

continuous_noise_cov为12x12的对角矩阵

(2) 初始化卡方检验表，置信度为0.95(不知道做什么用)，存在chi_squared_test_table中。
(3) 创建ROS IO接口，订阅imu、features、mocap_odom，发布reset、gt_odom。

imuCallback
接收到的imu数据放在缓存imu_msg_buffer中，如果数据个数大于200，初始化重力和偏置。由于s-msckf默认从静止状态初始化，所以用加速度平均值初始化重力，角速度平均值初始化角速度偏置。把is_gravity_set设置为true。
得到重力向量后构建世界坐标系，令重力方向为z轴负方向。然后定义惯性系相对于世界坐标系的朝向，这里我看的有点懵，他先求解了一个旋转$\mathbf{R}$,使得:

$$\mathbf{R}\mathbf{g}^I = -\mathbf{g}^w$$

然后把初始时刻的旋转$\mathbf{q}_{iw}$定义为:

$$\mathbf{q}_{iw} = quaternion(R^T)$$

这一步目前还不是很懂

```
double gravity_norm = gravity_imu.norm();
IMUState::gravity = Vector3d(0.0, 0.0, -gravity_norm);
Quaterniond q0_i_w = Quaterniond::FromTwoVectors(
    gravity_imu, -IMUState::gravity);
state_server.imu_state.orientation =
    rotationToQuaternion(q0_i_w.toRotationMatrix().transpose());
```

4 featureCallback

4.1 一些预设置

首先判断重力是否已经设置，即判断is_gravity_set是否为true;再判断是否为第一张图像，如果是，把is_first_img置为false，把state_server.imu_state.time置为该图像时间。

4.2 batchImuProcessing

该函数主要用于积分上一次积分时间到当前图像时间的imu数据，也就是积分相邻两个图像时间内的imu数据，并且构建协方差矩阵。

4.2.1 processModel

该函数用于构建$\mathbf{F}$矩阵、$\mathbf{G}$矩阵和$\boldsymbol{\Phi}$矩阵，更新状态协方差$\mathbf{P}$和状态变量$\mathbf{X}$，其中$\mathbf{P}$存放在state_server.state_cov里。

4.2.1.1 构建$\mathbf{F}$、$\mathbf{G}$和$\boldsymbol{\Phi}$

$$\mathbf{F}_{21\times 21} = \begin{pmatrix} -\hat{\boldsymbol{\omega}}_\times & -\mathbf{I}_3 & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ -C(\hat{\mathbf{q}}_I^G)^T\hat{\mathbf{a}}_\times & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & -C(\hat{\mathbf{q}}_I^G)^T & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{I}_3 & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \end{pmatrix}$$

$$\mathbf{G}_{21\times 12} = \begin{pmatrix} -\mathbf{I}_3 & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & -\mathbf{I}_3 & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & -C(\hat{\mathbf{q}}_I^G)^T & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{I}_3 \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \end{pmatrix}$$

其中

$$\boldsymbol{\Phi} = \exp(\int_{t_k}^{t_{k+1}} \mathbf{F}(\tau)d\tau)$$

这里采用3阶泰勒展开来取近似

$$\boldsymbol{\Phi} \approx \mathbf{I} + \mathbf{F}d_t + \frac{1}{2}(\mathbf{F}d_t)^2 + \frac{1}{6}(\mathbf{F}d_t)^3$$

4.2.1.2 predictNewState: 利用RK4积分预测状态

RK4积分一般形式

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$\begin{aligned} k_1 &= f(t_n, \mathbf{x}_n) \\ k_2 &= f(t_n + \frac{1}{2}\Delta t, \mathbf{x}_n + \frac{\Delta t}{2}k_1) \\ k_3 &= f(t_n + \frac{1}{2}\Delta t, \mathbf{x}_n + \frac{\Delta t}{2}k_2) \\ k_4 &= f(t_n + \Delta t, \mathbf{x}_n + \Delta t \cdot k_3) \end{aligned}$$

这一步要对PVQ的状态进行更新，其中对PV的更新用的是RK4积分，对于Q的更新是角速度乘以间隔时间。PVQ保存在state_server.imu_state里面。

```
Vector4d& q = state_server.imu_state.orientation;
Vector3d& v = state_server.imu_state.velocity;
Vector3d& p = state_server.imu_state.position;
```

a.更新Q

首先计算角速度的二范数$\|\boldsymbol{\omega}\|_2$，然后乘以时间间隔得到旋转角度$\phi$。

$$\phi = \|\boldsymbol{\omega}\|_2 \cdot \Delta t$$

于是ϕ对应的四元数为

$$\mathbf{q} = \begin{pmatrix} \cos(\phi/2) \\ \mathbf{u}\sin(\phi/2) \end{pmatrix} = \begin{pmatrix} q_w \\ \mathbf{q}_v \end{pmatrix}$$

其中$\mathbf{u}$为旋转轴的单位向量。于是更新后的四元数为：

$$\begin{aligned} \mathbf{q}_{k+1} &= \mathbf{q} \otimes \mathbf{q}_k \\ &= (q_w \mathbf{I} + \begin{pmatrix} -\lfloor \boldsymbol{\omega} \times \rfloor & \boldsymbol{\omega} \\ -\boldsymbol{\omega}^T & 0 \end{pmatrix}) \mathbf{q}_k \\ &= (q_w \mathbf{I} + \boldsymbol{\Omega}(\boldsymbol{\omega})) \mathbf{q}_k \end{aligned}$$

b.PV的更新(这部分d1和d2在代码里没完全弄明白，待深究)

V:

$$\begin{aligned} k_1^v &= \mathbf{q}_k^{-1} \otimes \mathbf{a}_k + \mathbf{g}^w \\ k_2^v &= d2 * \mathbf{a}_k + \mathbf{g}^w \\ k_3^v &= d2 * \mathbf{a}_k + \mathbf{g}^w \\ k_4^v &= d1 * \mathbf{a}_k + \mathbf{g}^w \end{aligned}$$

P:

$$\begin{aligned} k_1^p &= \mathbf{v}_k \\ k_2^p &= \mathbf{v}_k + \frac{1}{2} k_1^v \Delta t \\ k_3^p &= \mathbf{v}_k + \frac{1}{2} k_2^v \Delta t \\ k_4^p &= \mathbf{v}_k + k_3^v \Delta t \end{aligned}$$

然后把上面的系数代入RK4积分公式就得到了$\mathbf{v}_{k+1}, \mathbf{p}_{k+1}$，然后更新state_server.imu_state。

4.2.1.3 更新转移矩阵$\boldsymbol{\Phi}$和协方差矩阵$\mathbf{P}$

a.$\boldsymbol{\Phi}$的更新（ 感觉像是某种trick，论文里面没有详述，待深究)

b.$\mathbf{P}$的更新

$$\mathbf{P}_{II_{k+1|k}} = \boldsymbol{\Phi}_k \mathbf{P}_{II_{k|k}} \boldsymbol{\Phi}_k^T + \mathbf{Q}_k$$

其中

$$\mathbf{Q}_k = \int_{t_k}^{t_{k+1}} \boldsymbol{\Phi}(t_{k+1}, \tau) \mathbf{G} \mathbf{Q} \mathbf{G} \boldsymbol{\Phi}(t_{k+1}, \tau)^T d\tau$$

代码中对$\mathbf{Q}_k$取近似：

$$\mathbf{Q}_k \approx \boldsymbol{\Phi}_k \mathbf{G} \mathbf{Q} \mathbf{G} \boldsymbol{\Phi}_k^T \Delta t$$

```
Matrix<double, 21, 21> Q = Phi*G*state_server.continuous_noise_cov*
        G.transpose()*Phi.transpose()*dtime;
state_server.state_cov.block<21, 21>(0, 0) =
    Phi*state_server.state_cov.block<21, 21>(0, 0)*Phi.transpose() + Q;
```

c.对相机状态协方差的更新（之后补上)

4.2.2 更新状态ID，删除已经积分的imu数据

4.3 stateAugmentation：添加新的相机状态，更新状态协方差

a.根据imu数据的积分值和imu和相机之间的外参给相机状态（相对于世界坐标系的平移、旋转）赋值。

$$^{C}_{G}\hat{\mathbf{q}} = {}^{C}_{I}\bar{\mathbf{q}} \otimes {}^{G}_{I}\hat{\mathbf{q}} \tag{32}$$

$$^{G}\hat{\mathbf{p}}_C = {}^{G}\hat{\mathbf{p}}_I + C^{T}_{\hat{\mathbf{q}}} {}^{I}\mathbf{p}_C \tag{33}$$

b.更新状态协方差矩阵(待详细推导)

a中得到了一个新的相机位姿，该相机位姿添加到状态向量中，对应的状态协方差矩阵也要进行相应的更新（augmented)，设该相机位姿为第N+1个相机位姿，其对应的误差状态为：

$$\delta T_{C_{N+1} \leftarrow G} = \begin{pmatrix} \delta\boldsymbol{\theta}_{C_{N+1} \leftarrow G} \\ {}^{G}\tilde{\mathbf{p}}_{C_{N+1}} \end{pmatrix} \tag{34}$$

其中，

$$\exp(\delta\boldsymbol{\theta}_{\boldsymbol{C_{N+1}}}) \tag{35}$$

$$\mathbf{J} = \frac{\delta T_{C_{N+1} \leftarrow G}}{\delta \tilde{\mathbf{X}}} \tag{36}$$

更新后的状态协方差为：

$$\begin{aligned}\mathbf{P}_{k|k} &= \begin{pmatrix}\mathbf{I}_{6N+21}\\ \mathbf{J}\end{pmatrix}\mathbf{P}_{k|k}\begin{pmatrix}\mathbf{I}_{6N+21}\\ \mathbf{J}\end{pmatrix}^T\\ &= \begin{pmatrix}\mathbf{P}_{k|k} & \mathbf{P}\mathbf{J}^T\\ \mathbf{P}\mathbf{J} & \mathbf{J}\mathbf{P}\mathbf{J}^T\end{pmatrix}\end{aligned}$$

相机的误差状态与IMU的误差状态存在以下关系：

$$\delta\boldsymbol{\theta}_{C_{N+1}} = {}^{C}_{I}\bar{\mathbf{q}}\delta\boldsymbol{\theta}_{I_{N+1}} \tag{37}$$

$${}^{G}\tilde{\mathbf{p}}_{C_{N+1}} = {}^{G}\tilde{\mathbf{p}}_{I_{N+1}} + {}^{I_{N+1}}_{G}\hat{\bar{\mathbf{q}}}^T({}^{I}\mathbf{p}_C)^{\wedge}\delta\boldsymbol{\theta}_{I_{N+1}} \tag{38}$$

$$\Longleftrightarrow \mathbf{J}_{6\times(21+6N)} = \begin{pmatrix}\mathbf{J}_{I6\times21} & \mathbf{0}_{6\times6N}\end{pmatrix}$$

$$\Longleftrightarrow \mathbf{J}_I = \begin{pmatrix}C({}^{C}_{I}\bar{\mathbf{q}}) & \mathbf{0}_{3\times9} & \mathbf{0}_{3\times3} & \mathbf{I}_3 & \mathbf{0}_{3\times3}\\ C^T_{\hat{\bar{\mathbf{q}}}}({}^{I}\mathbf{p}_C)^{\wedge} & \mathbf{0}_{3\times9} & \mathbf{I}_3 & \mathbf{0}_{3\times3} & \mathbf{I}_3\end{pmatrix}$$

此处$\mathbf{J}_I$跟论文里的推导有点区别，之后深究。
更新状态协方差矩阵后为了确保其对称，与他的转置求和，然后除以二:

$$\mathbf{P} \leftarrow (\mathbf{P} + \mathbf{P}^T)/2$$

4.4 addFeatureObservations

添加新特征、跟踪特征。

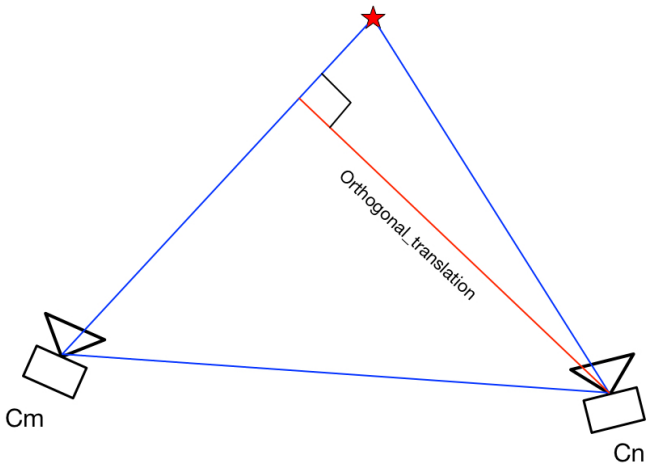
4.5 removeLostFeatures

移除已经不再跟踪的特征点、进行measurement update
如果特征点仍然在跟踪，但是尚未初始化，就根据checkMotion和initializePosition来判断是否为无效点。

4.5.1 检测相机运动

```
bool Feature::checkMotion(const CamStateServer& cam_states);
```

checkMotion主要是check第一次观测到某特征点时的相机位置和最后一次观测到该特征点时的相机位置（当前相机位置）是否存在足够的平移，主要判断下图红色线段长度是否超过阈值。



如图所示，Cm是第一次观测到该特征点时的相机位姿，Cn是最后一次观测到该特征点的相机位姿，如果orthogonal_translation超过阈值，则返回真，否则返回假。

4.5.2 初始化特征点3D位置

```
bool Feature::initializePosition(const CamStateServer& cam_states);
```

如果checkMotion返回真，则对特征点位置进行初始化。
算法中把计算特征点的3D坐标构建成了一个最小二乘问题，利用Levenberg-Marquart算法进行求解。再进行优化之前，首先要得到一个初始估计(initial guess)。

```
void Feature::generateInitialGuess(
    const Eigen::Isometry3d& T_c1_c2, const Eigen::Vector2d& z1,
    const Eigen::Vector2d& z2, Eigen::Vector3d& p)
```

```
// Generate initial guess
  Eigen::Vector3d initial_position(0.0, 0.0, 0.0);
  generateInitialGuess(cam_poses[cam_poses.size()-1], measurements[0],
      measurements[measurements.size()-1], initial_position);
  Eigen::Vector3d solution(
      initial_position(0)/initial_position(2),
      initial_position(1)/initial_position(2),
      1.0/initial_position(2));
```

在计算初始估计时，取观测到该特征点的第一帧的左相机归一化坐标和观测到该特征点的最后一帧的右相机归一化坐标进行三角化来估计深度，通过足够大的平移来保证一个较高的精度。
接下来就是基于L-M算法的非线性优化，优化变量为:

$$\mathbf{x} = \begin{pmatrix} \bar{x} & \bar{y} & \rho \end{pmatrix}^T$$

其中$\bar{x}, \bar{y}$是特征点的归一化坐标，ρ是逆深度。假设该特征点在第j帧图像的观测值的归一化坐标为$\mathbf{m}_j$，第j帧相机位姿相对于第一次观测到该特征点的相机的相对位姿为$\mathbf{R}, \mathbf{t}$，则重投影误差为：

$$\mathbf{r} = \mathbf{R}\mathbf{p} + \rho\mathbf{t} - \mathbf{m}_j$$

$$= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} \bar{x} \\ \bar{y} \\ 1 \end{pmatrix} + \rho \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} - \begin{pmatrix} \bar{x}_{m_j} \\ \bar{y}_{m_j} \\ 1 \end{pmatrix}$$

这里只取$\mathbf{r}$的前两维$\mathbf{r}_{2\times 1} = \begin{pmatrix} r_x & r_y \end{pmatrix}^T$。于是雅克比矩阵为：

$$\mathbf{J} = \begin{pmatrix} \frac{dr_x}{d\bar{x}} & \frac{dr_x}{d\bar{y}} & \frac{dr_x}{d\rho} \\ \frac{dr_y}{d\bar{x}} & \frac{dr_y}{d\bar{y}} & \frac{dr_y}{d\rho} \end{pmatrix}$$

其中，令$\mathbf{p}' = \mathbf{R}\mathbf{p} + \rho\mathbf{t} = \begin{pmatrix} \bar{x}' & \bar{y}' & \bar{z}' \end{pmatrix}^T$。则：

$$\begin{aligned} \frac{dr_x}{d\bar{x}} &= \frac{a_{11}\bar{z}' - a_{31}\bar{z}'}{\bar{z}'^2}, & \frac{dr_x}{d\bar{y}} &= \frac{a_{12}\bar{z}' - a_{32}\bar{x}'}{\bar{z}'^2}, & \frac{dr_x}{d\rho} &= \frac{t_x\bar{z}'^2 - t_z\bar{x}'}{\bar{z}'^2} \\ \frac{dr_y}{d\bar{x}} &= \frac{a_{21}\bar{z}' - \bar{y}'a_{31}}{\bar{z}'^2}, & \frac{dr_y}{d\bar{y}} &= \frac{a_{22}\bar{z}' - a_{32}\bar{y}'}{\bar{z}'^2}, & \frac{dr_y}{d\rho} &= \frac{t_y\bar{z}' - \bar{y}'t_z}{\bar{z}'^2} \end{aligned}$$

代码中对于雅可比的计算位于函数

```
void Feature::jacobian(const Eigen::Isometry3d& T_c0_ci,
    const Eigen::Vector3d& x, const Eigen::Vector2d& z,
    Eigen::Matrix<double, 2, 3>& J, Eigen::Vector2d& r,
    double& w) const
```

其中计算雅克比部分的代码为

```
// Compute the Jacobian.
Eigen::Matrix3d W;
W.leftCols<2>() = T_c0_ci.linear().leftCols<2>();
W.rightCols<1>() = T_c0_ci.translation();

J.row(0) = 1/h3*W.row(0) - h1/(h3*h3)*W.row(2);
J.row(1) = 1/h3*W.row(1) - h2/(h3*h3)*W.row(2);
```

把上式稍作变换，我们就能得到与代码一致的形式：

$$\begin{aligned} \begin{pmatrix} \frac{dr_x}{d\bar{x}} & \frac{dr_x}{d\bar{y}} & \frac{dr_x}{d\rho} \end{pmatrix} &= \frac{1}{\bar{z}'} \begin{pmatrix} a_{11} & a_{12} & t_x \end{pmatrix} - \frac{\bar{x}'}{\bar{z}'^2} \begin{pmatrix} a_{31} & a_{32} & t_z \end{pmatrix} \\ \begin{pmatrix} \frac{dr_y}{d\bar{x}} & \frac{dr_y}{d\bar{y}} & \frac{dr_y}{d\rho} \end{pmatrix} &= \frac{1}{\bar{z}'} \begin{pmatrix} a_{21} & a_{22} & t_y \end{pmatrix} - \frac{\bar{y}'}{\bar{z}'^2} \begin{pmatrix} a_{31} & a_{32} & t_z \end{pmatrix} \end{aligned}$$

然后根据误差来计算权值，如果误差小于一定值，则权值为1.0，若大于该值，根据huber核函数计算权值：

```
r = z_hat - z;
// Compute the weight based on the residual.
double e = r.norm();
if (e <= optimization_config.huber_epsilon)
  w = 1.0;
else
  w = optimization_config.huber_epsilon / (2*e);
```

然后基于L-M算法构建增量方程：

$$(\mathbf{A} + \lambda\mathbf{I})\delta\mathbf{x} = \mathbf{b}$$

λ的选择逻辑

代码中对于λ取值十分清晰，直接把代码搬上来：

```
if (new_cost < total_cost)
{
  is_cost_reduced = true;
  solution = new_solution;
  total_cost = new_cost;
  lambda = lambda/10 > 1e-10 ? lambda/10 : 1e-10;
}
else
{
  is_cost_reduced = false;
  lambda = lambda*10 < 1e12 ? lambda*10 : 1e12;
}
```

其中λ的初始值是1e-3。

4.5.3 计算雅可比矩阵和残差向量

该过程在一个循环中计算与每个特征点相关的雅可比矩阵和残差向量，然后拼成一个大的雅可比矩阵和残差向量，用于下一步的观测更新(measurement update)。

```
// This function computes the Jacobian of all measurements viewed
// in the given camera states of this feature.
void featureJacobian(const FeatureIDType& feature_id,
                     const std::vector<StateIDType>& cam_state_ids,
                     Eigen::MatrixXd& H_x, Eigen::VectorXd& r);
```

该函数放在特征点的循环中执行，每次计算两个雅可比矩阵$\mathbf{H}_{x_j}$，$\mathbf{H}_{f_i}$和残差向量$\mathbf{r}_j$，然后把雅可比矩阵和残差向量投影到$\mathbf{H}_{f_i}$的零空间。

A. 计算$\mathbf{H}_{x_j}$、$\mathbf{H}_{f_j}$和$\mathbf{r}_j$

这个过程也是放在一个循环中进行，把单个小的雅克比矩阵拼成一个稍大的雅克比矩阵。计算单个小的

雅克比矩阵的函数为MsckfVio::measurementJacobian。它计算的是观测误差关于特征点3D坐标和一对相机状态向量的雅克比矩阵。相机i对特征点j的观测残差为：

$$\mathbf{r}_i^j = \mathbf{z}_i^j - \hat{\mathbf{z}}_i^j = \mathbf{H}_{C_i}^j \tilde{\mathbf{x}}_{C_i} + \mathbf{H}_{f_i}^j {}^G\tilde{\mathbf{p}}_j + \mathbf{n}_i^j$$

对应于论文的附录C中。即$\mathbf{H}_{C_i}^j$和$\mathbf{H}_{f_i}^j$。
根据链式法则：

$$\mathbf{H}_{C_i}^j = \frac{\partial \mathbf{z}_i^j}{\partial {}^{C_{i,1}}\mathbf{p}_j} \cdot \frac{\partial {}^{C_{i,1}}\mathbf{p}_j}{\partial \mathbf{x}_{C_{i,1}}} + \frac{\partial \mathbf{z}_i^j}{\partial {}^{C_{i,2}}\mathbf{p}_j} \cdot \frac{\partial {}^{C_{i,2}}\mathbf{p}_j}{\partial \mathbf{x}_{C_{i,1}}}$$
$$\mathbf{H}_{f_i}^j = \frac{\partial \mathbf{z}_i^j}{\partial {}^{C_{i,1}}\mathbf{p}_j} \cdot \frac{\partial {}^{C_{i,1}}\mathbf{p}_j}{\partial {}^{G}\mathbf{p}_j} + \frac{\partial \mathbf{z}_i^j}{\partial {}^{C_{i,2}}\mathbf{p}_j} \cdot \frac{\partial {}^{C_{i,2}}\mathbf{p}_j}{\partial {}^{G}\mathbf{p}_j}$$

其中，

$$\frac{\partial \mathbf{z}_i^j}{\partial {}^{C_{i,1}}\mathbf{p}_j} = \frac{1}{{}^{C_{i,1}}\hat{Z}_j} \begin{pmatrix} 1 & 0 & -\frac{{}^{C_{i,1}}\hat{X}_j}{{}^{C_{i,1}}\hat{Z}_j} \\ 0 & 1 & -\frac{{}^{C_{i,1}}\hat{Y}_j}{{}^{C_{i,1}}\hat{Z}_j} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$
$$\frac{\partial \mathbf{z}_i^j}{\partial {}^{C_{i,2}}\mathbf{p}_j} = \frac{1}{{}^{C_{i,2}}\hat{Z}_j} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & -\frac{{}^{C_{i,2}}\hat{X}_j}{{}^{C_{i,1}}\hat{Z}_j} \\ 0 & 1 & -\frac{{}^{C_{i,2}}\hat{Y}_j}{{}^{C_{i,1}}\hat{Z}_j} \end{pmatrix}$$
$$\frac{\partial {}^{C_{i,1}}\mathbf{p}_j}{\partial \mathbf{x}_{C_{i,1}}} = \begin{pmatrix} \lfloor {}^{C_{i,1}}\hat{\mathbf{p}}_{j\times} \rfloor & -C({}^{C_{i,1}}_{G}\hat{\mathbf{q}}) \end{pmatrix}$$
$$\frac{\partial {}^{C_{i,1}}\mathbf{p}_j}{\partial {}^{G}\mathbf{p}_j} = C({}^{C_{i,1}}_{G}\hat{\mathbf{q}})$$
$$\frac{\partial {}^{C_{i,2}}\mathbf{p}_j}{\partial \mathbf{x}_{C_{i,1}}} = C({}^{C_{i,1}}_{C_{i,2}}\mathbf{q})^\top \begin{pmatrix} \lfloor {}^{C_{i,1}}\hat{\mathbf{p}}_{j\times} \rfloor & -C({}^{C_{i,1}}_{G}\hat{\mathbf{q}}) \end{pmatrix}$$
$$\frac{\partial {}^{C_{i,2}}\mathbf{p}_j}{\partial {}^{G}\mathbf{p}_j} = C({}^{C_{i,1}}_{C_{i,2}}\mathbf{q})^\top C({}^{C_{i,1}}_{G}\mathbf{q})$$

在代码里，为了保证观测约束，对雅可比进行了一定的调整：
这步目前还没整明白

```
// Modifty the measurement Jacobian to ensure
// observability constrain.
Matrix<double, 4, 6> A = H_x;
Matrix<double, 6, 1> u = Matrix<double, 6, 1>::Zero();
u.block<3, 1>(0, 0) = quaternionToRotation(
    cam_state.orientation_null) * IMUState::gravity;
u.block<3, 1>(3, 0) = skewSymmetric(
    p_w-cam_state.position_null) * IMUState::gravity;
H_x = A - A*u*(u.transpose()*u).inverse()*u.transpose();
H_f = -H_x.block<4, 3>(0, 3);

// Compute the residual.
r = z - Vector4d(p_c0(0)/p_c0(2), p_c0(1)/p_c0(2),
    p_c1(0)/p_c1(2), p_c1(1)/p_c1(2));
```

通过把对同一个特征点的观测叠加起来，可以得到特征点j的雅可比矩阵：

$$\mathbf{r}^j = \mathbf{H}_{\mathbf{x}}^j \tilde{\mathbf{x}} + \mathbf{H}_f^j {}^G\tilde{\mathbf{p}}_j + \mathbf{n}^j$$

B. 把雅可比和残差向量投影到$\mathbf{H}_f^i$的零空间

由于当前形式的$\mathbf{r}^j$无法进行标准EKF更新，需要把它投影到$\mathbf{H}_f^i$的零空间,得到

$$\mathbf{r}_0^j = \mathbf{V}^\top \mathbf{r}^j = \mathbf{V}^\top \mathbf{H}_{\mathbf{x}}^j \tilde{\mathbf{x}} + \mathbf{V}^\top \mathbf{n}^j = \mathbf{H}_{\mathbf{x},0}^j \tilde{\mathbf{x}} + \mathbf{n}_0^j$$

$\mathbf{V}^\top$主要通过对$\mathbf{H}_f^i$进行SVD分解得到：

```
// Project the residual and Jacobians onto the nullspace
// of H_fj.
JacobiSVD<MatrixXd> svd_helper(H_fj, ComputeFullU | ComputeThinV);
MatrixXd A = svd_helper.matrixU().rightCols(
    jacobian_row_size - 3);

H_x = A.transpose() * H_xj;
r = A.transpose() * r_j;
```

那么这里有个问题，如果对单个双目观测的误差关于状态向量的雅可比进行投影，计算量更小，为什么不这样做呢?

论文的附录D向我们解释了原因，为了方便法表示，记以下符号为：

$$\frac{\partial \mathbf{z}_i^j}{\partial {}^{C_{i,1}}\mathbf{p}_j} = \begin{pmatrix} \mathbf{J}_1 \\ \mathbf{0} \end{pmatrix}, \frac{\partial \mathbf{z}_i^j}{\partial {}^{C_{i,1}}\mathbf{p}_j} = \begin{pmatrix} \mathbf{0} \\ \mathbf{J}_2 \end{pmatrix}$$
$$\frac{\partial {}^{C_{i,1}}\mathbf{p}_j}{\partial \mathbf{x}_{C_{i,1}}} = \begin{pmatrix} \mathbf{H}_1 \end{pmatrix}, \frac{\partial {}^{C_{i,1}}\mathbf{p}_j}{\partial {}^G\mathbf{p}_j} = \mathbf{H}_2, C({}^{C_{i,1}}_{C_{i,2}}\mathbf{q}) = \mathbf{R}$$

于是有

$$\mathbf{H}_{C_i}^j = \begin{pmatrix} \mathbf{J}_1\mathbf{H}_1 \\ \mathbf{J}_2\mathbf{R}^\top\mathbf{H}_1 \end{pmatrix}, \mathbf{H}_{f_i}^j = \begin{pmatrix} \mathbf{j}_1\mathbf{H}_2 \\ \mathbf{J}_2\mathbf{R}^\top\mathbf{H}_2 \end{pmatrix}$$

设$\mathbf{v} = (\mathbf{v}_1^\top, \mathbf{v}_2^\top)^\top \in \mathbb{R}^4$是$\mathbf{H}_{f_i}^j$的左零空间，于是有

$$\mathbf{v}^\top\mathbf{H}_{f_i}^j = (\mathbf{v}_1^\top\mathbf{J}_1 + \mathbf{v}_2^\top\mathbf{J}_2\mathbf{R}^\top)\mathbf{H}_2 = \mathbf{0}$$

由于$\mathbf{H}_2 = C({}_G^{C_{i,1}}\hat{\mathbf{q}})$是一个旋转矩阵，它的秩等于3，因此$\mathbf{v}_1^\top\mathbf{J}_1 + \mathbf{v}_2^\top\mathbf{J}_2\mathbf{R}^\top = 0$。有这个性质，就可以直接推出$\mathbf{v}$也是$\mathbf{H}_{C_i}^j$的左零空间：

$$\mathbf{v}^\top\mathbf{H}_{C_i}^j = (\mathbf{v}_1^\top\mathbf{J}_1 + \mathbf{v}_2^\top\mathbf{J}_2\mathbf{R}^\top)\mathbf{H}_1 = 0$$

所以单独的一个双目观测不能用于EKF更新。

4.5.4 EKF更新

在把上一步得到的雅可比矩阵$\mathbf{H}_{\mathbf{x},0}^j$和残差向量叠$\mathbf{r}_0^j$叠加起来后，就得到了一个大的雅可比矩阵$\mathbf{H}_{x,0}$和残差向量$\mathbf{r}_0$，为了与代码中的变量保持一致，记为$\mathbf{H}_x$和$\mathbf{r}$。
对应算法位于

```
void MsckfVio::measurementUpdate(const MatrixXd& H, const VectorXd& r)
```

由于$\mathbf{H}_x$的维数较高，为了降低EKF更新的计算量，对$\mathbf{H}_x$进行QR分解：

$$\mathbf{H}_x = \begin{pmatrix} \mathbf{Q}_1 & \mathbf{Q}_2 \end{pmatrix} \begin{pmatrix} \mathbf{T}_H \\ \mathbf{0} \end{pmatrix}$$

其中$\mathbf{Q}_1, \mathbf{Q}_2$是单位正交列向量组，$\mathbf{T}_H$是上三角矩阵，于是

$$\begin{aligned} \mathbf{r} &= \begin{pmatrix} \mathbf{Q}_1 & \mathbf{Q}_2 \end{pmatrix} \begin{pmatrix} \mathbf{T}_H \\ \mathbf{0} \end{pmatrix} \tilde{\mathbf{x}} + \mathbf{n} \\ \Rightarrow \begin{pmatrix} \mathbf{Q}_1^\top\mathbf{r} \\ \mathbf{Q}_2^\top\mathbf{r} \end{pmatrix} &= \begin{pmatrix} \mathbf{T}_H \\ \mathbf{0} \end{pmatrix} \tilde{\mathbf{x}} + \begin{pmatrix} \mathbf{Q}_1^\top\mathbf{n} \\ \mathbf{Q}_2^\top\mathbf{n} \end{pmatrix} \end{aligned}$$

从上式可以很清楚的看出，通过把残差投影到$\mathbf{T}_H$的基向量组$\mathbf{Q}_1$上，可以保留观测的全部有效信息，残差项$\mathbf{Q}_2^\top\mathbf{r}$仅仅是噪声，完全可以忽略，于是可以通过以下残差来进行EKF更新：

$$\mathbf{r}' = \mathbf{Q}_1^\top\mathbf{r} = \mathbf{T}_H\tilde{\mathbf{x}} + \mathbf{n}'$$

计算卡尔曼增益：

$$\mathbf{K} = \mathbf{P}\mathbf{T}_H^\top (\mathbf{T}_H \mathbf{P}\mathbf{T}_H^\top + \mathbf{R})^{-1}$$

于是状态向量的修正增量为：

$$\Delta \mathbf{x} = \mathbf{K}\mathbf{r}$$

然后更新协方差：

$$\mathbf{P}_{k+1|k+1} = (\mathbf{I}_\xi - \mathbf{K}\mathbf{T}_H)\mathbf{P}_{k+1|k}(\mathbf{I}_\xi - \mathbf{K}\mathbf{T}_H)^\top + \mathbf{K}\mathbf{R}\mathbf{K}^\top$$

trick: 代码并没有严格按照公式来，而是做了一些调整

```
// Update state covariance.
MatrixXd I_KH = MatrixXd::Identity(K.rows(), H_thin.cols()) - K*H_thin;
//state_server.state_cov = I_KH*state_server.state_cov*I_KH.transpose() +
//  K*K.transpose()*Feature::observation_noise;
state_server.state_cov = I_KH*state_server.state_cov;

// Fix the covariance to be symmetric
MatrixXd state_cov_fixed = (state_server.state_cov +
    state_server.state_cov.transpose()) / 2.0;
state_server.state_cov = state_cov_fixed;
```

4.6 删除冗余相机位姿

对应算法位于MsckfVio::pruneCamStateBuffer()。

4.6.1 查找冗余相机位姿

对应代码位于MsckfVio::findRedundantCamStates。冗余相机位姿为与关键相机位姿相差小的位姿或者相机状态向量中靠前的位姿。把冗余相机位姿的id放在一个vector中并进行排序，输出。

4.6.2 关于冗余相机位姿的测量更新

如果查找到2个冗余相机位姿，则遍历这2个位姿产生共视的所有特征点构建特征雅可比矩阵，删除冗余相机位姿的观测，关于这些共视点和2个相机位姿进行EKF更新。
最后从相机状态向量里删除冗余相机位姿，删除状态协方差里关于冗余相机位姿的矩阵块。

4.7 发布里程计

位姿发布给ros进行显示。