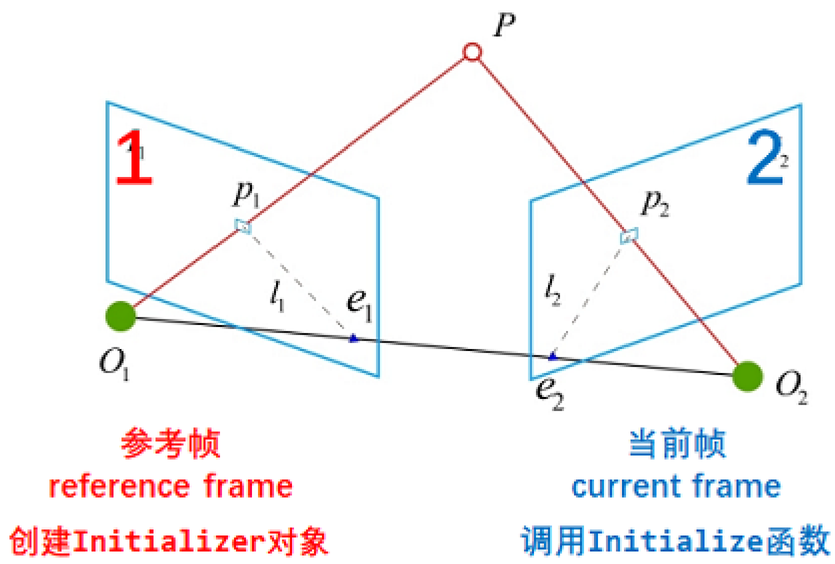
`Initializer` 类仅用于**单目相机**初始化,双目/RGBD相机初始化不用这个类.

# 各成员变量/函数

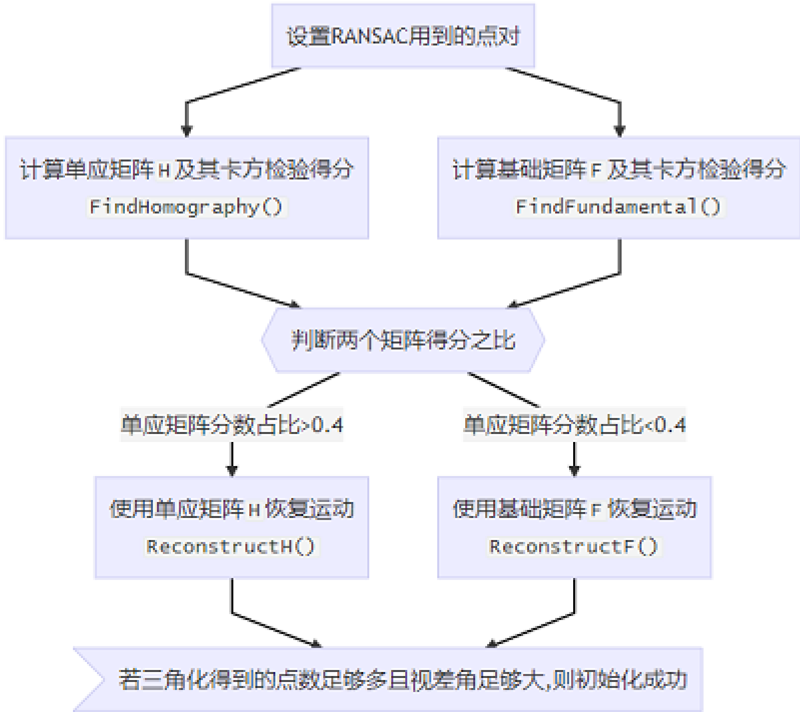
成员变量名中: `1` 代表参考帧(reference frame)中特征点编号, `2` 代表当前帧(current frame)中特征点编号.




| 各成员函数/变量 | 访问控制 | 意义 |
| --- | --- | --- |
| `vector<cv::KeyPoint> mvKeys1` | `private` | 参考帧(reference frame)中的特征点 |
| `vector<cv::KeyPoint> mvKeys2` | `private` | 当前帧(current frame)中的特征点 |
| `vector<pair<int,int>> mvMatches12` | `private` | 从参考帧到当前帧的匹配特征点对 |
| `vector<bool> mvbMatched1` | `private` | 参考帧特征点是否在当前帧存在匹配特征点 |
| `cv::Mat mK` | `private` | 相机内参 |
| `float mSigma, mSigma2` | `private` | 重投影误差阈值及其平方 |
| `int mMaxIterations` | `private` | RANSAC迭代次数 |
| `vector<vector<size_t>> mvSets` | `private` | 二维容器 `N×8`<br>每一层保存RANSAC计算 `H` 和 `F` 矩阵所需的八对点 |

## 初始化函数: `Initialize()`

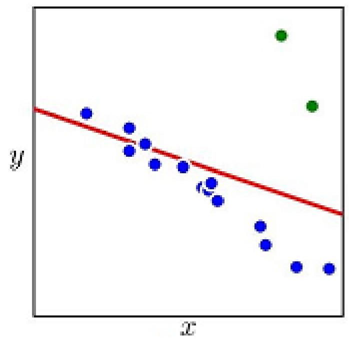
主函数 `Initialize()` 根据两帧间的匹配关系**恢复帧间运动并计算地图点位姿**.

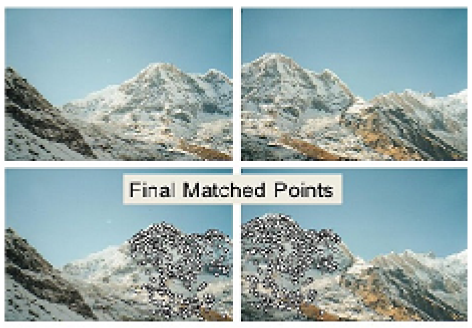



```cpp
bool Initializer::Initialize(const Frame &CurrentFrame,
                             const vector<int> &vMatches12,
                             cv::Mat &R21, cv::Mat &t21,
                             vector<cv::Point3f> &vP3D,
                             vector<bool> &vbTriangulated) {

    // 初始化器Initializer对象创建时就已指定mvKeys1,调用本函数只需指定mvKeys2即可
    mvKeys2 = CurrentFrame.mvKeysUn;        // current frame中的特征点
    mvMatches12.reserve(mvKeys2.size());
    mvbMatched1.resize(mvKeys1.size());

    // step1. 将vMatches12拷贝到mvMatches12,mvMatches12只保存匹配上的特征点对
    for (size_t i = 0, iend = vMatches12.size(); i < iend; i++) {
        if (vMatches12[i] >= 0) {
            mvMatches12.push_back(make_pair(i, vMatches12[i]));
            mvbMatched1[i] = true;
        } else
            mvbMatched1[i] = false;
    }

    // step2. 准备RANSAC运算中需要的特征点对
    const int N = mvMatches12.size();
    vector<size_t> vAllIndices;
    for (int i = 0; i < N; i++) {
        vAllIndices.push_back(i);
    }
    mvSets = vector<vector<size_t> >(mMaxIterations, vector<size_t>(8, 0));
    for (int it = 0; it < mMaxIterations; it++) {
        vector<size_t> vAvailableIndices = vAllIndices;
        for (size_t j = 0; j < 8; j++) {
            int randi = DUtils::Random::RandomInt(0, vAvailableIndices.size() - 1);
            int idx = vAvailableIndices[randi];
            mvSets[it][j] = idx;
            vAvailableIndices[randi] = vAvailableIndices.back();
            vAvailableIndices.pop_back();
        }
    }

    // step3. 计算F矩阵和H矩阵及其置信程度
    vector<bool> vbMatchesInliersH, vbMatchesInliersF;
    float SH, SF;
    cv::Mat H, F;

    thread threadH(&Initializer::FindHomography, this, ref(vbMatchesInliersH), ref(SH), ref(H));
    thread threadF(&Initializer::FindFundamental, this, ref(vbMatchesInliersF), ref(SF), ref(F));
    threadH.join();
    threadF.join();

    // step4. 根据比分计算使用哪个矩阵恢复运动
    float RH = SH / (SH + SF);
    if (RH > 0.40)
        return ReconstructH(vbMatchesInliersH, H, mK, R21, t21, vP3D, vbTriangulated, 1.0, 50);
    else
        return ReconstructF(vbMatchesInliersF, F, mK, R21, t21, vP3D, vbTriangulated, 1.0, 50);
}
```

## 计算基础矩阵 F 和单应矩阵 H

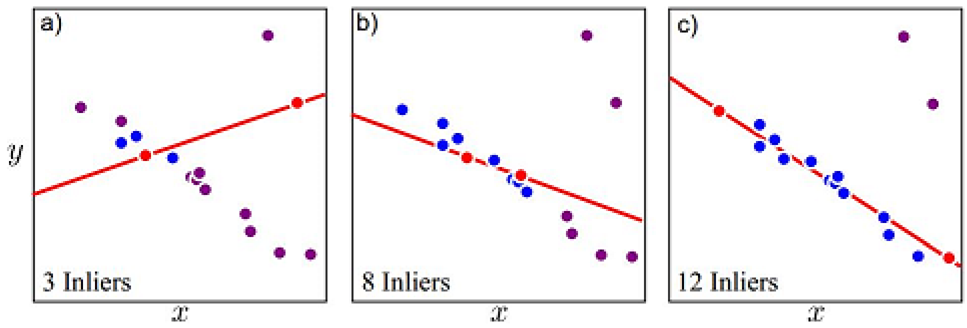
## RANSAC算法




少数外点会极大影响计算结果的准确度.随着采样数量的增加,外点数量也会同时增加,这是一种**系统误差**,无法通过增加采样点来解决.

RANSAC(Random sample consensus,随机采样一致性)算法的思路是少量多次重复实验,每次实验仅使用尽可能少的点来计算,并统计本次计算中的内点数.只要尝试次数足够多的话,总会找到一个包含所有内点的解.

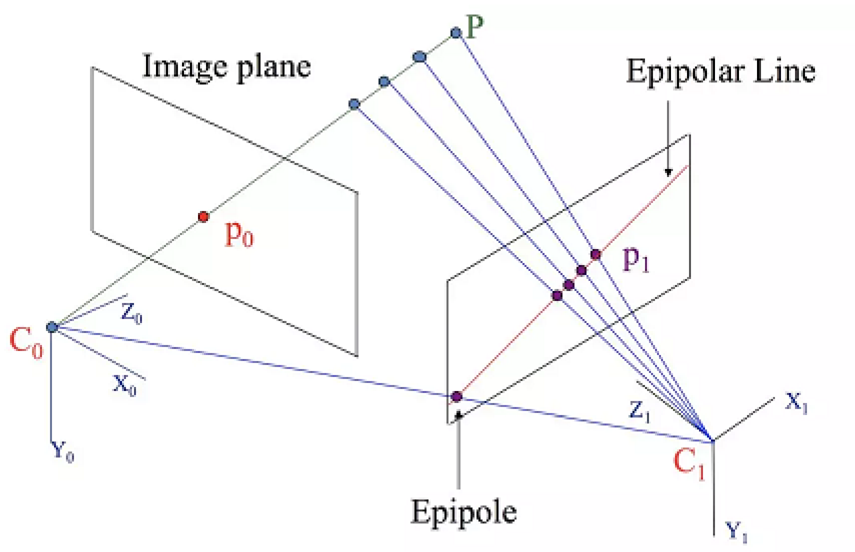



RANSAC算法的核心是减少每次迭代所需的采样点数.从原理上来说,计算 `F` 矩阵最少只需要 `7` 对匹配点,计算 `H` 矩阵最少只需要 `4` 对匹配点;ORB-SLAM2中为了编程方便,每次迭代使用 `8` 对匹配点计算 `F` 和 `H` .

RANSAC: Computed k (p=0.99)

| Sample size n | Proportion of outliers 5% | 10% | 20% | 25% | 30% | 40% | 50% |
|---|---|---|---|---|---|---|---|
| 2 | 2 | 3 | 5 | 6 | 7 | 11 | 17 |
| 3 | 3 | 4 | 7 | 9 | 11 | 19 | 35 |
| 4 | 3 | 5 | 9 | 13 | 17 | 34 | 72 |
| 5 | 4 | 6 | 12 | 17 | 26 | 57 | 146 |
| 6 | 4 | 7 | 16 | 24 | 37 | 97 | 293 |
| 7 | 4 | 8 | 20 | 33 | 54 | 163 | 588 |
| 8 | 5 | 9 | 26 | 44 | 78 | 272 | 1177 |

## 计算基础矩阵`F`: `FindFundamental()`




设点 `P` 在相机1、2坐标系下的坐标分别为$X_1$、$X_2$,在相机1、2成像平面下的像素坐标分别为$x_1$、$x_2$,有:

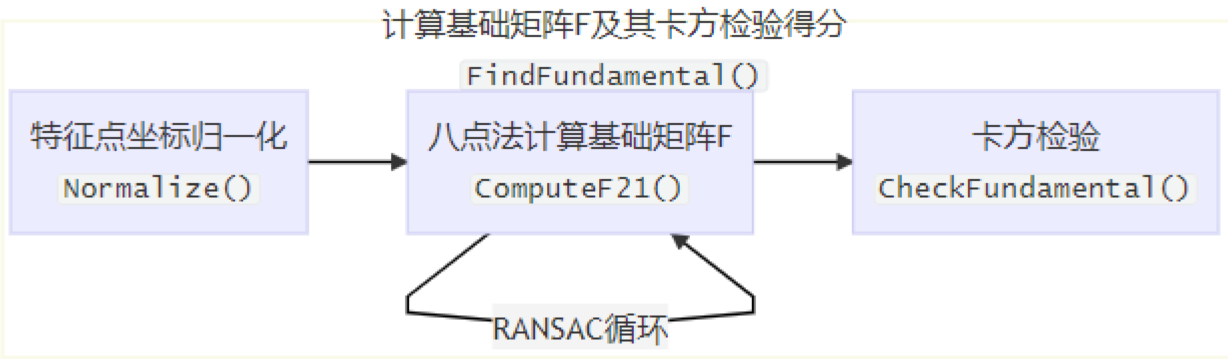
$$X_2^T E X_1 = 0$$
$$x_1 = K_1 X_1$$
$$x_2 = K_2 X_2$$

其中本质矩阵$E = t^\wedge R$.

$$x_2^T k_2^{-T} E K_1^{-1} x_1 = 0$$

令$F = k_2^{-T} E k_1^{-1}$,得到:

$$x_2^T F x_1 = 0$$




```cpp
void Initializer::FindFundamental(vector<bool> &vbMatchesInliers, float &score, cv::Mat &F21) {

    const int N = vbMatchesInliers.size();

    // step1. 特征点归一化
    vector<cv::Point2f> vPn1, vPn2;
    cv::Mat T1, T2;
    Normalize(mvKeys1, vPn1, T1);
    Normalize(mvKeys2, vPn2, T2);
    cv::Mat T2t = T2.t();       // 用于恢复原始尺度

    // step2. RANSAC循环
    score = 0.0;                                        // 最优解得分
    vbMatchesInliers = vector<bool>(N, false);         // 最优解对应的内点
    for (int it = 0; it < mMaxIterations; it++) {
        vector<cv::Point2f> vPn1i(8);
        vector<cv::Point2f> vPn2i(8);
        cv::Mat F21i;
        vector<bool> vbCurrentInliers(N, false);
        float currentScore;

        for (int j = 0; j < 8; j++) {
            int idx = mvSets[it][j];
            vPn1i[j] = vPn1[mvMatches12[idx].first];        // first存储在参考帧1中的特征点索引
            vPn2i[j] = vPn2[mvMatches12[idx].second];       // second存储在当前帧2中的特征点索引
        }

        // step3. 八点法计算单应矩阵H
        cv::Mat Fn = ComputeF21(vPn1i, vPn2i);

        // step4. 恢复原始尺度
        F21i = T2t * Fn * T1;

        // step5. 根据重投影误差进行卡方检验
        currentScore = CheckFundamental(F21i, vbCurrentInliers, mSigma);

        // step6. 记录最优解
        if (currentScore > score) {
            F21 = F21i.clone();
            vbMatchesInliers = vbCurrentInliers;
            score = currentScore;
        }
    }
}

```

## 八点法计算F矩阵: `ComputeF21()`

`F`矩阵的约束:

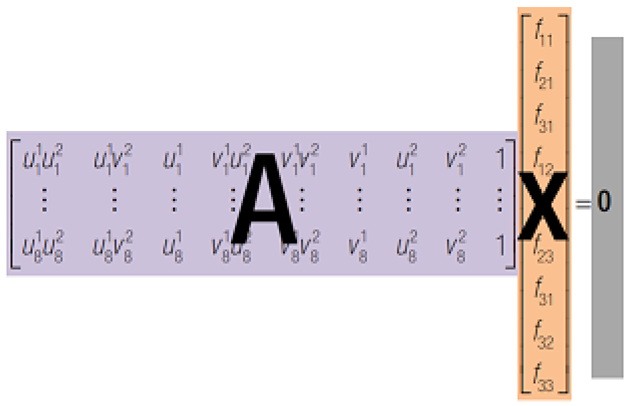
$$
(u_2, v_2, 1)\begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix}\begin{pmatrix} u_1 \\ v_1 \\ 1 \end{pmatrix} = 0
$$

展开成:

$$
u_1 u_2 f_{11} + u_1 v_2 f_{21} + u_1 f_{31} + v_1 u_2 f_{12} + v_1 v_2 f_{22} + v_1 f_{32} + u_2 f_{13} + v_2 f_{23} + f_{33} = 0
$$

由于`F`矩阵的尺度不变性,只需8对特征点即可提供足够的约束.

$$
\begin{pmatrix}
u_1^1 u_2^1 & u_1^1 v_2^1 & u_1^1 & v_1^1 u_2^1 & v_1^1 v_2^1 & v_1^1 & u_2^1 & v_2^1 & 1 \\
u_1^2 u_2^2 & u_1^2 v_2^2 & u_1^2 & v_1^2 u_2^2 & v_1^2 v_2^2 & v_1^2 & u_2^2 & v_2^2 & 1 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
u_1^8 u_2^8 & u_1^8 v_2^8 & u_1^8 & v_1^8 u_2^8 & v_1^8 v_2^8 & v_1^8 & u_2^8 & v_2^8 & 1
\end{pmatrix}
\begin{pmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{pmatrix} = 0
$$



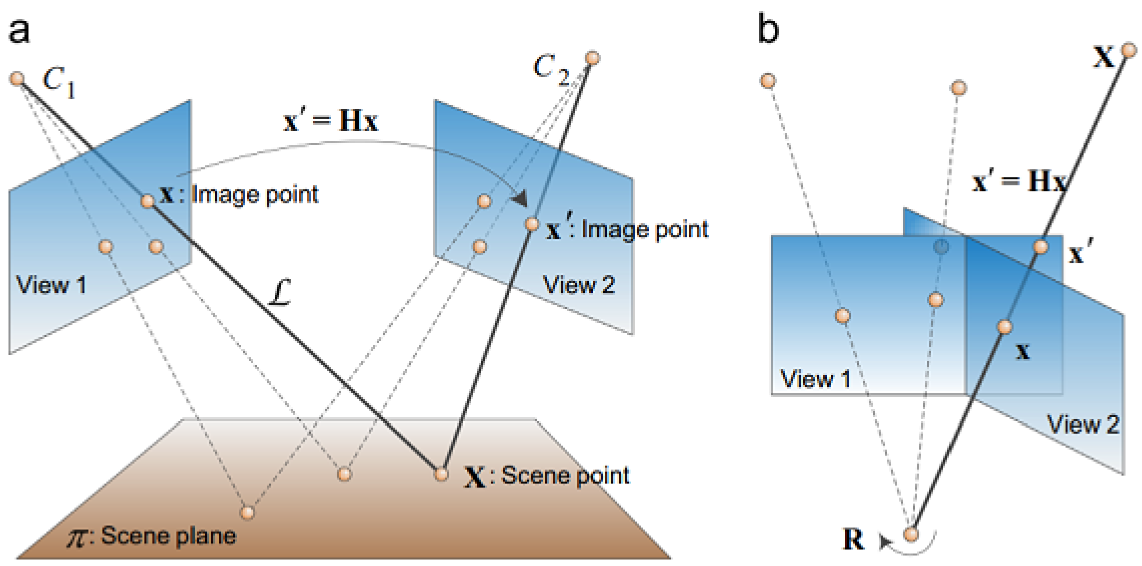
上图中$A$矩阵是一个$8 \times 9$的矩阵,$x$是一个$9 \times 1$的向量;上述方程是一个超定方程,使用SVD分解求最小二乘解.

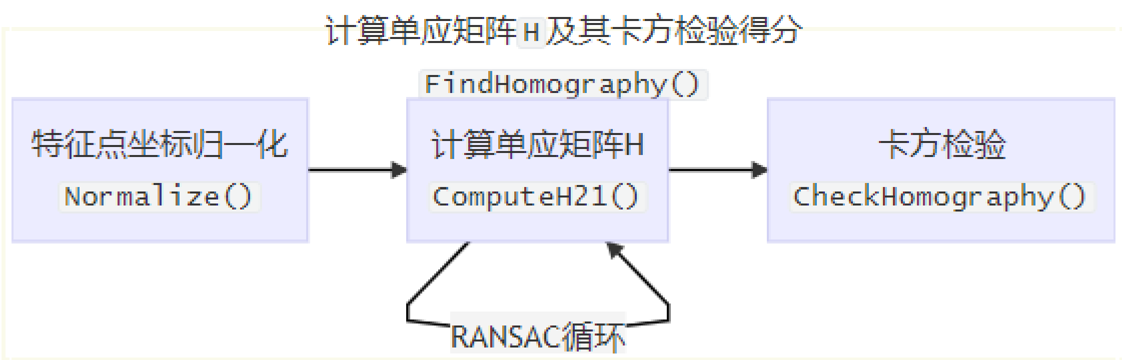
```cpp
cv::Mat Initializer::ComputeF21(const vector<cv::Point2f> &vP1, const vector<cv::Point2f> &vP2) {

    const int N = vP1.size();

    // step1. 构造A矩阵
    cv::Mat A(N, 9, CV_32F);
    for (int i = 0; i < N; i++) {
        const float u1 = vP1[i].x;
        const float v1 = vP1[i].y;
        const float u2 = vP2[i].x;
        const float v2 = vP2[i].y;
        A.at<float>(i, 0) = u2 * u1;
        A.at<float>(i, 1) = u2 * v1;
        A.at<float>(i, 2) = u2;
        A.at<float>(i, 3) = v2 * u1;
        A.at<float>(i, 4) = v2 * v1;
        A.at<float>(i, 5) = v2;
        A.at<float>(i, 6) = u1;
        A.at<float>(i, 7) = v1;
        A.at<float>(i, 8) = 1;
    }

    // step2. 奇异值分解,取vt最后一行
    cv::Mat u, w, vt;
    cv::SVDecomp(A, w, u, vt, cv::SVD::MODIFY_A | cv::SVD::FULL_UV);
    cv::Mat Fpre = vt.row(8).reshape(0, 3); // v的最后一列

    // step3. 将F矩阵的秩强制置为2
    cv::SVDecomp(Fpre, w, u, vt, cv::SVD::MODIFY_A | cv::SVD::FULL_UV);
    w.at<float>(2) = 0;
    return u * cv::Mat::diag(w) * vt;
}
```

## 计算单应矩阵`H`: `FindHomography()`

以下两种情况更适合使用单应矩阵进行初始化:

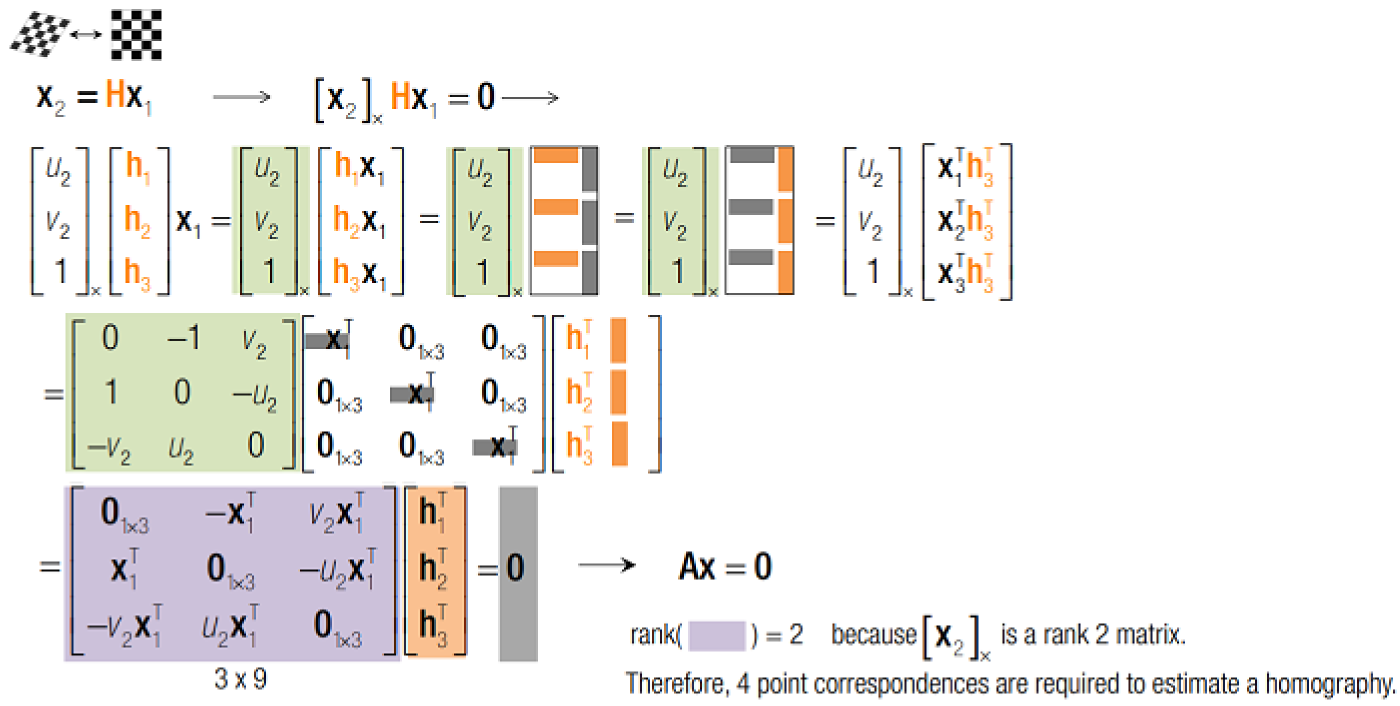
1. 相机看到的场景是一个平面.
2. 连续两帧间没发生平移,只发生旋转.




使用八点法求解单应矩阵`H`的原理类似:

$$x_2 = Hx_1$$




正常来说只用 4 对匹配点就可以计算单应矩阵 H ,但ORB-SLAM2每次RANSAC迭代取 8 对匹配点来计算 H .个人理解这是为了和八点法计算基础矩阵 H 相对应,都使用 8 对匹配点来计算,便于后面比较分数优劣.

```
void Initializer::FindHomography(vector<bool> &vbMatchesInliers, float &score, cv::Mat &H21) {

    const int N = mvMatches12.size();

    // step1. 特征点归一化
    vector<cv::Point2f> vPn1, vPn2;
    cv::Mat T1, T2;
    Normalize(mvKeys1, vPn1, T1);
    Normalize(mvKeys2, vPn2, T2);
    cv::Mat T2inv = T2.inv();       // 用于恢复原始尺度

    // step2. RANSAC循环
    score = 0.0;                                    // 最优解得分
    vbMatchesInliers = vector<bool>(N, false);      // 最优解对应的内点
    for (int it = 0; it < mMaxIterations; it++) {
        vector<cv::Point2f> vPn1i(8);
        vector<cv::Point2f> vPn2i(8);
        cv::Mat H21i, H12i;
        vector<bool> vbCurrentInliers(N, false);
        float currentScore;

        for (size_t j = 0; j < 8; j++) {
            int idx = mvSets[it][j];
            vPn1i[j] = vPn1[mvMatches12[idx].first];    // first存储在参考帧1中的特征点索引
            vPn2i[j] = vPn2[mvMatches12[idx].second];   // second存储在当前帧2中的特征点索引
        }

        // step3. 八点法计算单应矩阵H
        cv::Mat Hn = ComputeH21(vPn1i, vPn2i);

        // step4. 恢复原始尺度
        H21i = T2inv * Hn * T1;
        H12i = H21i.inv();

        // step5. 根据重投影误差进行卡方检验
        currentScore = CheckHomography(H21i, H12i, vbCurrentInliers, mSigma);

        // step6. 记录最优解
        if (currentScore > score) {
            H21 = H21i.clone();
            vbMatchesInliers = vbCurrentInliers;
            score = currentScore;
        }
    }
}
```

```
cv::Mat Initializer::ComputeH21(const vector<cv::Point2f> &vP1, const vector<cv::Point2f> &vP2) {

    const int N = vP1.size();

    // step1. 构造A矩阵
    cv::Mat A(2 * N, 9, CV_32F);
    for (int i = 0; i < N; i++) {
        const float u1 = vP1[i].x;
        const float v1 = vP1[i].y;
        const float u2 = vP2[i].x;
        const float v2 = vP2[i].y;
        A.at<float>(2 * i, 0) = 0.0;
        A.at<float>(2 * i, 1) = 0.0;
        A.at<float>(2 * i, 2) = 0.0;
        A.at<float>(2 * i, 3) = -u1;
```

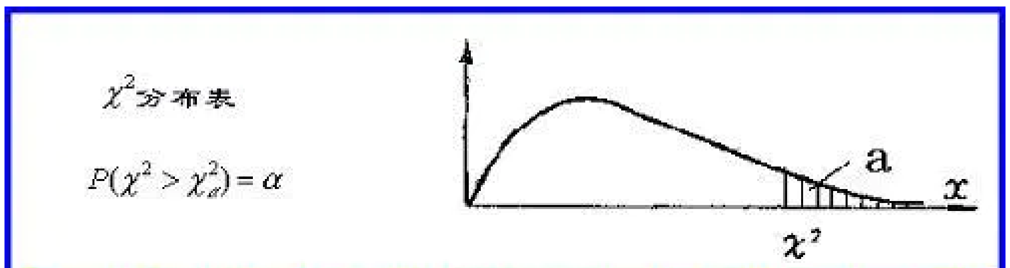
```
        A.at<float>(2 * i, 4) = -v1;
        A.at<float>(2 * i, 5) = -1;
        A.at<float>(2 * i, 6) = v2 * u1;
        A.at<float>(2 * i, 7) = v2 * v1;
        A.at<float>(2 * i, 8) = v2;
        A.at<float>(2 * i + 1, 0) = u1;
        A.at<float>(2 * i + 1, 1) = v1;
        A.at<float>(2 * i + 1, 2) = 1;
        A.at<float>(2 * i + 1, 3) = 0.0;
        A.at<float>(2 * i + 1, 4) = 0.0;
        A.at<float>(2 * i + 1, 5) = 0.0;
        A.at<float>(2 * i + 1, 6) = -u2 * u1;
        A.at<float>(2 * i + 1, 7) = -u2 * v1;
        A.at<float>(2 * i + 1, 8) = -u2;
    }

    // step2. 奇异值分解,取vt最后一行
    cv::Mat u, w, vt;
    cv::SVDecomp(A, w, u, vt, cv::SVD::MODIFY_A |cv::SVD::FULL_UV);
    return vt.row(8).reshape(0, 3);
}
```

## 卡方检验计算置信度得分: `CheckFundamental()`、`CheckHomography()`

卡方检验通过构造检验统计量$\chi^2$来比较**期望结果**和**实际结果**之间的差别,从而得出观察频数极值的发生概率.

$$\chi^2 = \Sigma \frac{(O-E)^2}{E}$$

根据重投影误差构造统计量$\chi^2$,其值越大,观察结果和期望结果之间的差别越显著,某次计算越可能用到了外点.




| F | 0.99 | 0.98 | 0.95 | 0.90 | 0.50 | 0.10 | 0.05 | 0.02 | 0.01 | 0.001 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.000 | 0.001 | 0.004 | 0.016 | 0.045 | 2.71 | 3.84 | 5.41 | 6.64 | 10.83 |
| 2 | 0.020 | 0.040 | 0.103 | 0.211 | 1.36 | 4.61 | 5.99 | 7.82 | 9.21 | 13.82 |
| 3 | 0.115 | 0.185 | 0.352 | 0.584 | 2.366 | 6.25 | 7.82 | 9.84 | 11.34 | 16.27 |
| 4 | 0.297 | 0.429 | 0.711 | 1.064 | 3.357 | 7.78 | 9.49 | 11.67 | 13.28 | 18.47 |
| 5 | 0.554 | 0.752 | 1.145 | 1.610 | 4.351 | 9.24 | 11.07 | 13.39 | 15.09 | 20.52 |
| 6 | 0.874 | 1.134 | 1.635 | 2.204 | 5.35 | 10.65 | 12.69 | 15.03 | 16.81 | 22.46 |
| 7 | 1.239 | 1.564 | 2.167 | 2.833 | 6.35 | 12.02 | 14.07 | 16.62 | 18.48 | 24.37 |
| 8 | 1.646 | 2.032 | 2.731 | 3.490 | 7.34 | 13.36 | 15.51 | 18.17 | 20.09 | 26.13 |
| 9 | 2.088 | 2.532 | 3.325 | 4.168 | 8.34 | 14.68 | 16.92 | 19.68 | 21.67 | 27.88 |
| 10 | 2.558 | 3.059 | 3.940 | 4.865 | 9.34 | 15.99 | 18.31 | 21.16 | 23.21 | 29.59 |

统计量置信度阈值与被检验变量自由度有关: 单目特征点重投影误差的自由度为`2`(`u`,`v`),双目特征点重投影误差自由度为`3`(`u`,`v`,`ur`).

取95%置信度下的卡方检验统计量阈值

- 若统计量大于该阈值,则认为计算矩阵使用到了外点,将其分数设为`0`.
- 若统计量小于该阈值,则将**统计量裕量**设为该解的置信度分数.

```
float Initializer::CheckHomography(const cv::Mat &H21, const cv::Mat &H12,vector<bool> &vbMatchesInliers,
float sigma) {
    const int N = mvMatches12.size();

    // 取出单应矩阵H各位上的值
    const float h11 = H21.at<float>(0, 0);
    const float h12 = H21.at<float>(0, 1);
    const float h13 = H21.at<float>(0, 2);
    const float h21 = H21.at<float>(1, 0);
    const float h22 = H21.at<float>(1, 1);
    const float h23 = H21.at<float>(1, 2);
    const float h31 = H21.at<float>(2, 0);
    const float h32 = H21.at<float>(2, 1);
    const float h33 = H21.at<float>(2, 2);

    const float h11inv = H12.at<float>(0, 0);
    const float h12inv = H12.at<float>(0, 1);
    const float h13inv = H12.at<float>(0, 2);
    const float h21inv = H12.at<float>(1, 0);
    const float h22inv = H12.at<float>(1, 1);
    const float h23inv = H12.at<float>(1, 2);
    const float h31inv = H12.at<float>(2, 0);
    const float h32inv = H12.at<float>(2, 1);
    const float h33inv = H12.at<float>(2, 2);

    vbMatchesInliers.resize(N);      // 标记是否是内点
    float score = 0;                 // 置信度得分
    const float th = 5.991;          // 自由度为2,显著性水平为0.05的卡方分布对应的临界阈值
```

```
    const float invSigmaSquare = 1.0 / (sigma * sigma);     // 信息矩阵,方差平方的倒数


    // 双向投影,计算加权投影误差
    for (int i = 0; i < N; i++) {
        bool bIn = true;

        // step1. 提取特征点对
        const cv::KeyPoint &kp1 = mvKeys1[mvMatches12[i].first];
        const cv::KeyPoint &kp2 = mvKeys2[mvMatches12[i].second];
        const float u1 = kp1.pt.x;
        const float v1 = kp1.pt.y;
        const float u2 = kp2.pt.x;
        const float v2 = kp2.pt.y;

        // step2. 计算img2到img1的重投影误差
        const float w2in1inv = 1.0 / (h31inv * u2 + h32inv * v2 + h33inv);
        const float u2in1 = (h11inv * u2 + h12inv * v2 + h13inv) * w2in1inv;
        const float v2in1 = (h21inv * u2 + h22inv * v2 + h23inv) * w2in1inv;
        const float squareDist1 = (u1 - u2in1) * (u1 - u2in1) + (v1 - v2in1) * (v1 - v2in1);
        const float chiSquare1 = squareDist1 * invSigmaSquare;

        // step3. 离群点标记上,非离群点累加计算得分
        if (chiSquare1 > th)
            bIn = false;
        else
            score += th - chiSquare1;

        // step4. 计算img1到img2的重投影误差
        const float w1in2inv = 1.0 / (h31 * u1 + h32 * v1 + h33);
        const float u1in2 = (h11 * u1 + h12 * v1 + h13) * w1in2inv;
        const float v1in2 = (h21 * u1 + h22 * v1 + h23) * w1in2inv;
        const float squareDist2 = (u2 - u1in2) * (u2 - u1in2) + (v2 - v1in2) * (v2 - v1in2);
        const float chiSquare2 = squareDist2 * invSigmaSquare;

        // step5. 离群点标记上,非离群点累加计算得分
        if (chiSquare2 > th)
            bIn = false;
        else
            score += th - chiSquare2;


        if (bIn)
            vbMatchesInliers[i] = true;
        else
            vbMatchesInliers[i] = false;
    }
    return score;
}
```

## 归一化: `Normalize()`

使用均值和一阶中心矩归一化,归一化可以增强计算稳定性.

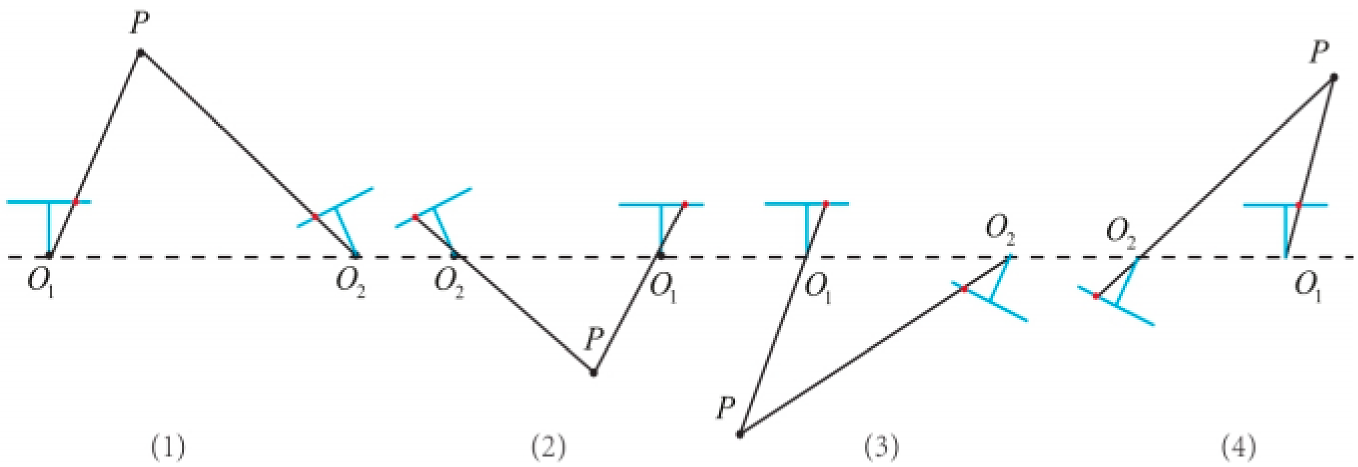
```
void Initializer::Normalize(const vector <cv::KeyPoint> &vKeys, vector <cv::Point2f> &vNormalizedPoints, cv::Mat &T) {
    // step1. 计算均值
    float meanX = 0;
    float meanY = 0;
    for (int i = 0; i < N; i++) {
        meanX += vKeys[i].pt.x;
        meanY += vKeys[i].pt.y;
    }
    meanX = meanX / N;
    meanY = meanY / N;

    // step2. 计算一阶中心矩
    float meanDevX = 0;
    float meanDevY = 0;
    for (int i = 0; i < N; i++) {
        vNormalizedPoints[i].x = vKeys[i].pt.x - meanX;
        vNormalizedPoints[i].y = vKeys[i].pt.y - meanY;
        meanDevX += fabs(vNormalizedPoints[i].x);
        meanDevY += fabs(vNormalizedPoints[i].y);
    }
    meanDevX = meanDevX / N;
    meanDevY = meanDevY / N;
    float sX = 1.0 / meanDevX;
    float sY = 1.0 / meanDevY;

    // step3. 进行归一化
    for (int i = 0; i < N; i++) {
        vNormalizedPoints[i].x = vNormalizedPoints[i].x * sX;
        vNormalizedPoints[i].y = vNormalizedPoints[i].y * sY;
```

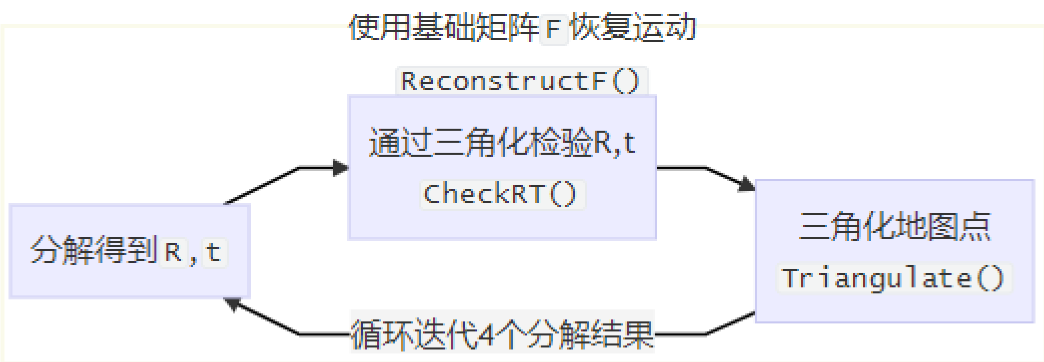
```
    }

    // 记录归一化参数,以便后续恢复尺度
    T = cv::Mat::eye(3, 3, CV_32F);
    T.at<float>(0, 0) = sX;
    T.at<float>(1, 1) = sY;
    T.at<float>(0, 2) = -meanX * sX;
    T.at<float>(1, 2) = -meanY * sY;
}
```

## 使用基础矩阵`F`和单应矩阵`H`恢复运动

### 使用基础矩阵`F`恢复运动: `ReconstructF()`

使用基础矩阵`F`分解`R`、`t`,数学上会得到四个可能的解,因此分解后调用函数`Initializer::CheckRT()`检验分解结果,取相机前方成功三角化数目最多的一组解.

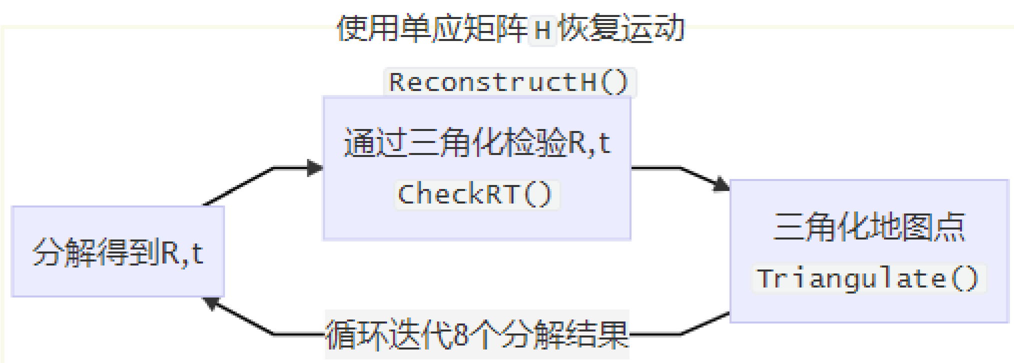



```
bool Initializer::ReconstructF(vector<bool> &vbMatchesInliers, cv::Mat &F21, cv::Mat &K, cv::Mat &R21, cv::Mat &t21, vector<cv::Point3f> &vP3D, vector<bool> &vbTriangulated, float minParallax, int minTriangulated) {

    int N = 0;
    for (size_t i = 0, iend = vbMatchesInliers.size(); i < iend; i++)
        if (vbMatchesInliers[i]) N++;

    // step1. 根据基础矩阵F推算本质矩阵E
    cv::Mat E21 = K.t() * F21 * K;

    // step2. 分解本质矩阵E,得到R,t
    cv::Mat R1, R2, t;
    DecomposeE(E21, R1, R2, t);
    cv::Mat t1 = t;
    cv::Mat t2 = -t;

    // step3. 检验分解出的4对R,t
    vector<cv::Point3f> vP3D1, vP3D2, vP3D3, vP3D4;
    vector<bool> vbTriangulated1, vbTriangulated2, vbTriangulated3, vbTriangulated4;
    float parallax1, parallax2, parallax3, parallax4;
    int nGood1 = CheckRT(R1, t1, mvKeys1, mvKeys2, mvMatches12, vbMatchesInliers, K, vP3D1, 4.0 * mSigma2, vbTriangulated1, parallax1);
    int nGood2 = CheckRT(R2, t1, mvKeys1, mvKeys2, mvMatches12, vbMatchesInliers, K, vP3D2, 4.0 * mSigma2, vbTriangulated2, parallax2);
    int nGood3 = CheckRT(R1, t2, mvKeys1, mvKeys2, mvMatches12, vbMatchesInliers, K, vP3D3, 4.0 * mSigma2, vbTriangulated3, parallax3);
    int nGood4 = CheckRT(R2, t2, mvKeys1, mvKeys2, mvMatches12, vbMatchesInliers, K, vP3D4, 4.0 * mSigma2, vbTriangulated4, parallax4);
    int maxGood = max(nGood1, max(nGood2, max(nGood3, nGood4)));
    R21 = cv::Mat();
    t21 = cv::Mat();
    int nMinGood = max(static_cast<int>(0.9 * N), minTriangulated);

    // step4. ratio test,最优分解应有区分度
    int nsimilar = 0;
    if (nGood1 > 0.7 * maxGood)
        nsimilar++;
    if (nGood2 > 0.7 * maxGood)
        nsimilar++;
    if (nGood3 > 0.7 * maxGood)
        nsimilar++;
    if (nGood4 > 0.7 * maxGood)
```

```
        nsimilar++;
    if (maxGood < nMinGood || nsimilar > 1) {
        return false;
    }

    // step5. 选择记录最佳结果,检验三角化出的特征点数和视差角
    if (maxGood == nGood1) {
        if (parallax1 > minParallax) {
            vP3D = vP3D1;
            vbTriangulated = vbTriangulated1;
            R1.copyTo(R21);
            t1.copyTo(t21);
            return true;
        }
    } else if (maxGood == nGood2) {
        if (parallax2 > minParallax) {
            vP3D = vP3D2;
            vbTriangulated = vbTriangulated2;

            R2.copyTo(R21);
            t1.copyTo(t21);
            return true;
        }
    } else if (maxGood == nGood3) {
        if (parallax3 > minParallax) {
            vP3D = vP3D3;
            vbTriangulated = vbTriangulated3;

            R1.copyTo(R21);
            t2.copyTo(t21);
            return true;
        }
    } else if (maxGood == nGood4) {
        if (parallax4 > minParallax) {
            vP3D = vP3D4;
            vbTriangulated = vbTriangulated4;

            R2.copyTo(R21);
            t2.copyTo(t21);
            return true;
        }
    }

    return false;
}
```

### 使用单应矩阵`H`恢复运动: `ReconstructH()`




### 检验分解结果`R`,`t`

通过成功三角化的特征点个数判断分解结果的好坏: 若某特征点的重投影误差小于`4`且视差角大于`0.36°`,则认为该特征点三角化成功

```
int Initializer::CheckRT(const cv::Mat &R, const cv::Mat &t, const vector<cv::KeyPoint> &vKeys1, const vector<cv::KeyPoint> &vKeys2, const vector<Match> &vMatches12, vector<bool> &vbMatchesInliers, const cv::Mat &K, vector<cv::Point3f> &vP3D, float th2, vector<bool> &vbGood, float &parallax) {

    const float fx = K.at<float>(0, 0);
    const float fy = K.at<float>(1, 1);
    const float cx = K.at<float>(0, 2);
    const float cy = K.at<float>(1, 2);

    vbGood = vector<bool>(vKeys1.size(), false);
    vP3D.resize(vKeys1.size());

    vector<float> vCosParallax;
    vCosParallax.reserve(vKeys1.size());

    // step1. 以相机1光心为世界坐标系,计算相机的投影矩阵和光心位置
    cv::Mat P1(3, 4, CV_32F, cv::Scalar(0));          // P1表示相机1投影矩阵, K[I|0]
    K.copyTo(P1.rowRange(0, 3).colRange(0, 3));
    cv::Mat O1 = cv::Mat::zeros(3, 1, CV_32F);        // O1表示世界坐标下相机1光心位置, O1=0
    cv::Mat P2(3, 4, CV_32F);                         // P2表示相机2投影矩阵, K[R|t]
    R.copyTo(P2.rowRange(0, 3).colRange(0, 3));
```

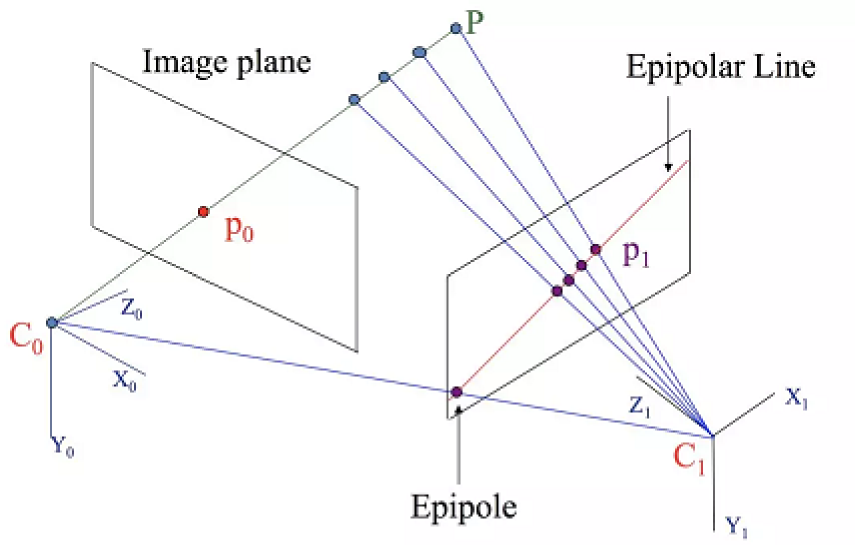
```
    t.copyTo(P2.rowRange(0, 3).col(3));             // O1表示世界坐标下相机2光心位置, O2=-R'*t
    P2 = K * P2;
    cv::Mat O2 = -R.t() * t;

    // 遍历所有特征点对
    int nGood = 0;
    for (size_t i = 0, iend = vMatches12.size(); i < iend; i++) {
        // step2. 三角化地图点
        const cv::KeyPoint &kp1 = vKeys1[vMatches12[i].first];
        const cv::KeyPoint &kp2 = vKeys2[vMatches12[i].second];
        cv::Mat p3dC1;
        Triangulate(kp1, kp2, P1, P2, p3dC1);

        // step3. 检查三角化坐标点合法性:
        // step3.1. 正确三角化的地图点深度值应为正数且视差角足够大
        cv::Mat normal1 = p3dC1 - O1;
        float dist1 = cv::norm(normal1);
        cv::Mat normal2 = p3dC1 - O2;
        float dist2 = cv::norm(normal2);
        float cosParallax = normal1.dot(normal2) / (dist1 * dist2);
        if (p3dC1.at<float>(2) <= 0 && cosParallax < 0.99998)
            continue;
        if (p3dC2.at<float>(2) <= 0 && cosParallax < 0.99998)
            continue;

        // step3.2. 正确三角化的地图点重投影误差应足够小
        float im1x, im1y;
        float invZ1 = 1.0 / p3dC1.at<float>(2);
        im1x = fx * p3dC1.at<float>(0) * invZ1 + cx;
        im1y = fy * p3dC1.at<float>(1) * invZ1 + cy;
        float squareError1 = (im1x - kp1.pt.x) * (im1x - kp1.pt.x) + (im1y - kp1.pt.y) * (im1y - kp1.pt.y);
        if (squareError1 > th2)
            continue;

        float im2x, im2y;
        float invZ2 = 1.0 / p3dC2.at<float>(2);
        im2x = fx * p3dC2.at<float>(0) * invZ2 + cx;
        im2y = fy * p3dC2.at<float>(1) * invZ2 + cy;
        float squareError2 = (im2x - kp2.pt.x) * (im2x - kp2.pt.x) + (im2y - kp2.pt.y) * (im2y - kp2.pt.y);
        if (squareError2 > th2)
            continue;

        // step4. 记录通过检验的地图点
        vCosParallax.push_back(cosParallax);
        vP3D[vMatches12[i].first] = cv::Point3f(p3dC1.at<float>(0), p3dC1.at<float>(1), p3dC1.at<float>(2));
        nGood++;
    }

    // step5. 记录三角化过程中的较小(第50个视差角)
    if (nGood > 0) {
        sort(vCosParallax.begin(), vCosParallax.end());
        size_t idx = min(50, int(vCosParallax.size() - 1));
        parallax = acos(vCosParallax[idx]) * 180 / CV_PI;
    } else
        parallax = 0;

    return nGood;
}
```

SVD求解超定方程

```
void Initializer::Triangulate(const cv::KeyPoint &kp1, const cv::KeyPoint &kp2, const cv::Mat &P1, const cv::Mat &P2, cv::Mat &x3D) {
    cv::Mat A(4, 4, CV_32F);
    A.row(0) = kp1.pt.x * P1.row(2) - P1.row(0);
    A.row(1) = kp1.pt.y * P1.row(2) - P1.row(1);
    A.row(2) = kp2.pt.x * P2.row(2) - P2.row(0);
    A.row(3) = kp2.pt.y * P2.row(2) - P2.row(1);
    cv::Mat u, w, vt;
    cv::SVD::compute(A, w, u, vt, cv::SVD::MODIFY_A | cv::SVD::FULL_UV);
    x3D = vt.row(3).t();
    x3D = x3D.rowRange(0, 3) / x3D.at<float>(3);
}
```

# 对极几何

## 本质矩阵$E$、基础矩阵$F$和单应矩阵$H$




设点$P$在相机1、2坐标系下的坐标分别为$X_1$、$X_2$,在相机1、2成像平面下的像素坐标分别为$x_1$、$x_2$,有:

$$E = t\hat{} R$$
$$F = K_2^{-T} E K_1^{-1}$$
$$x_2^T F x_1 = X_2^T F X_1 = 0$$

- $H$矩阵的自由为8:

  $H$矩阵为$3 \times 3$大小,自由度最大为9;考虑到尺度等价性约束,实际自由度为$9 - 1 = 8$.

- $F$矩阵自由度为7:

  $K1$、$K_2$待定参数各为4,$t$和$R$的待定参数各为3,共14个待定参数.

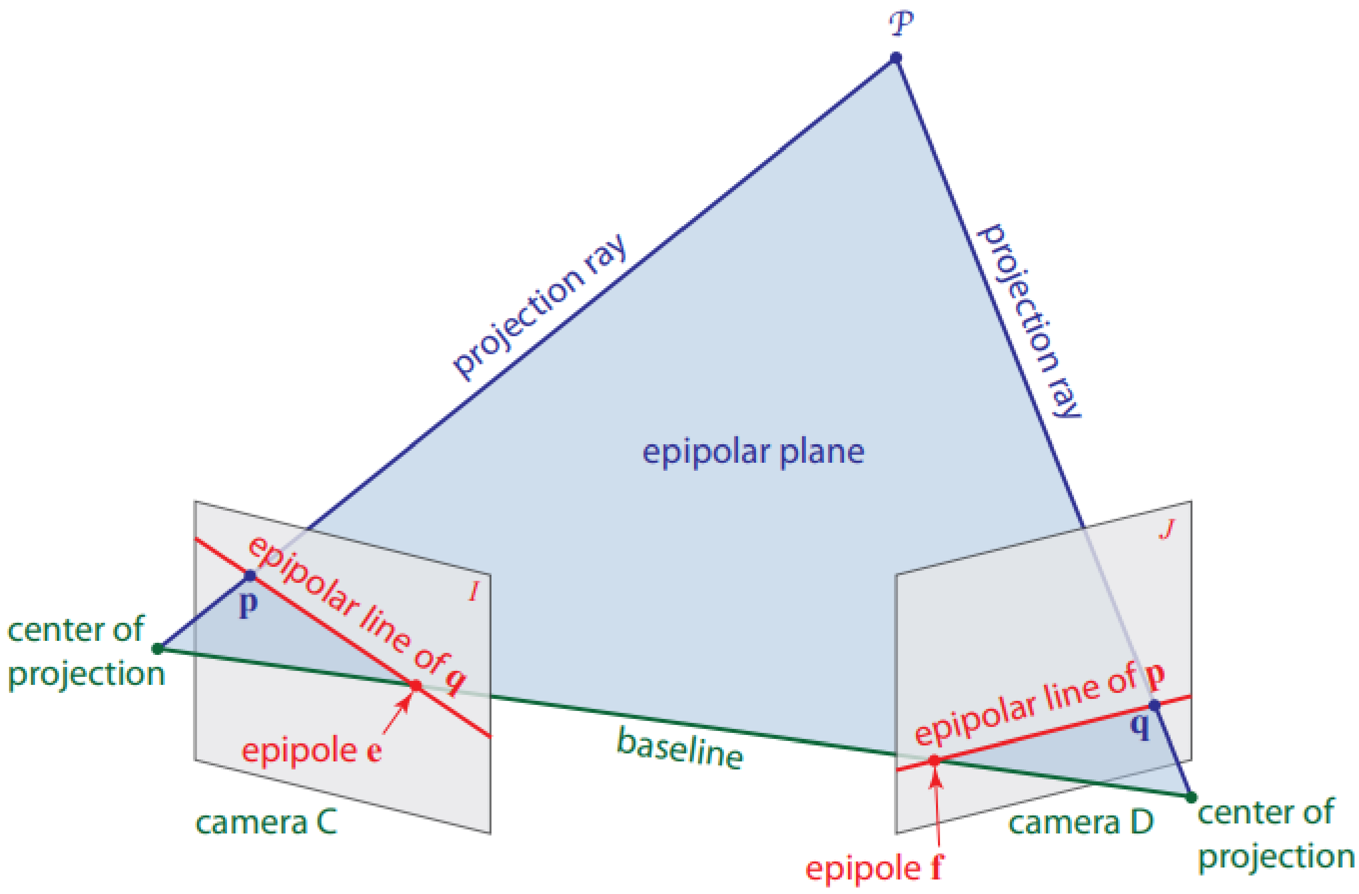
  但$F$矩阵为$3 \times 3$大小,自由度最大为9;考虑到**尺度等价性**和**行列式**$\det(F) = 0$两个约束,实际自由度为$9 - 2 = 7$.

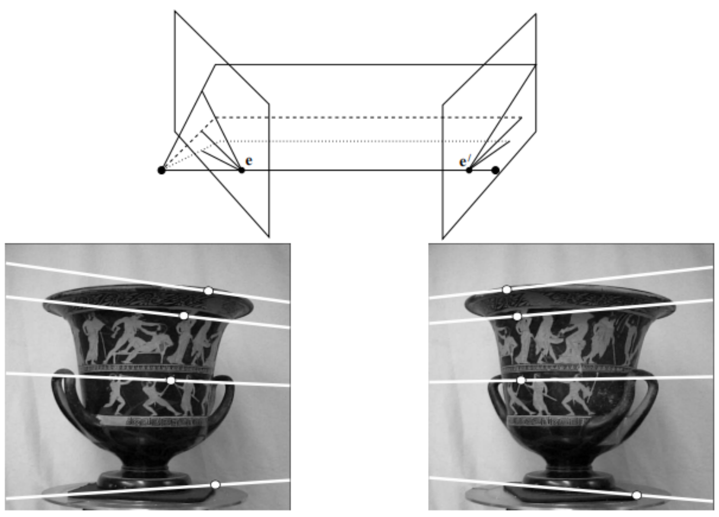
- $E$矩阵的自由度为5:

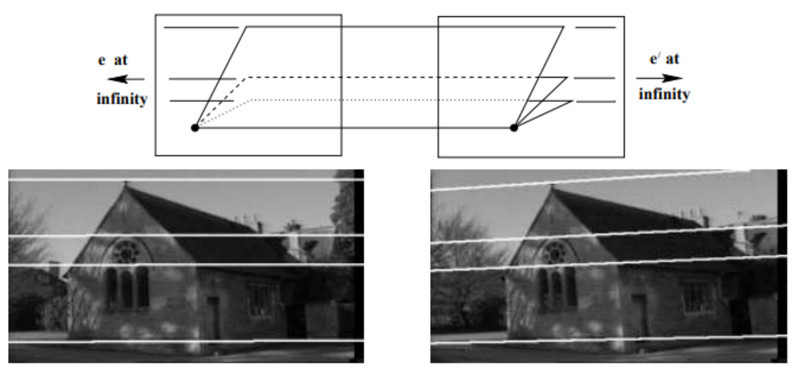
  $t$和$R$的自由度各为3,自由度最大为6,考虑到尺度等价性约束,实际自由度为$6 - 1 = 5$.

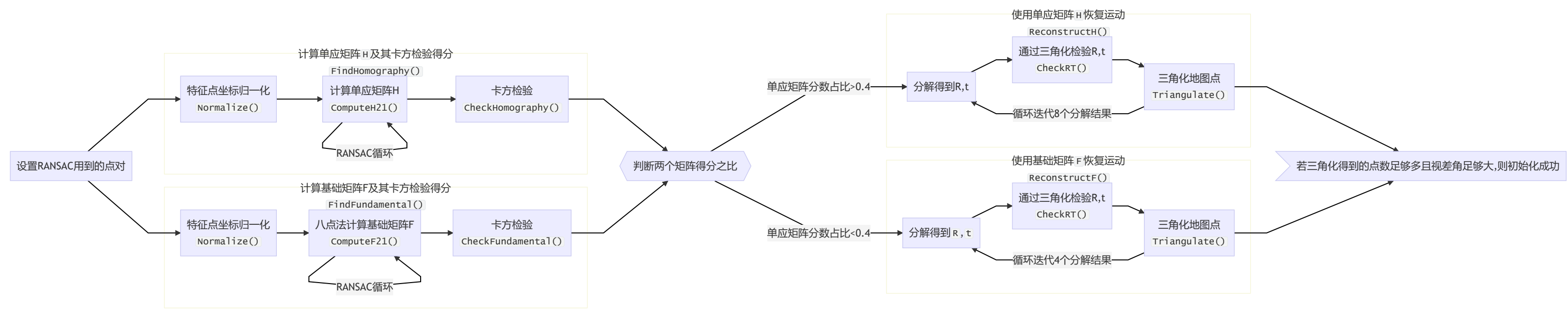
- $E$矩阵的秩为2,从两个方面来解释:
  - $rank(r) = 3$,$rank(t\hat{}) = 2$,因此$rank(E) = rank(t\hat{} R) = min(rank(r), rank(t\hat{})) = 2$
  - 对于某对非0坐标$x_1$、$x_2$,有$(x_2^T E)x_1 = 0$成立,说明方程$(x_2^T E)x_1 = 0$存在非零解,即矩阵$x_2^T E$不满秩.

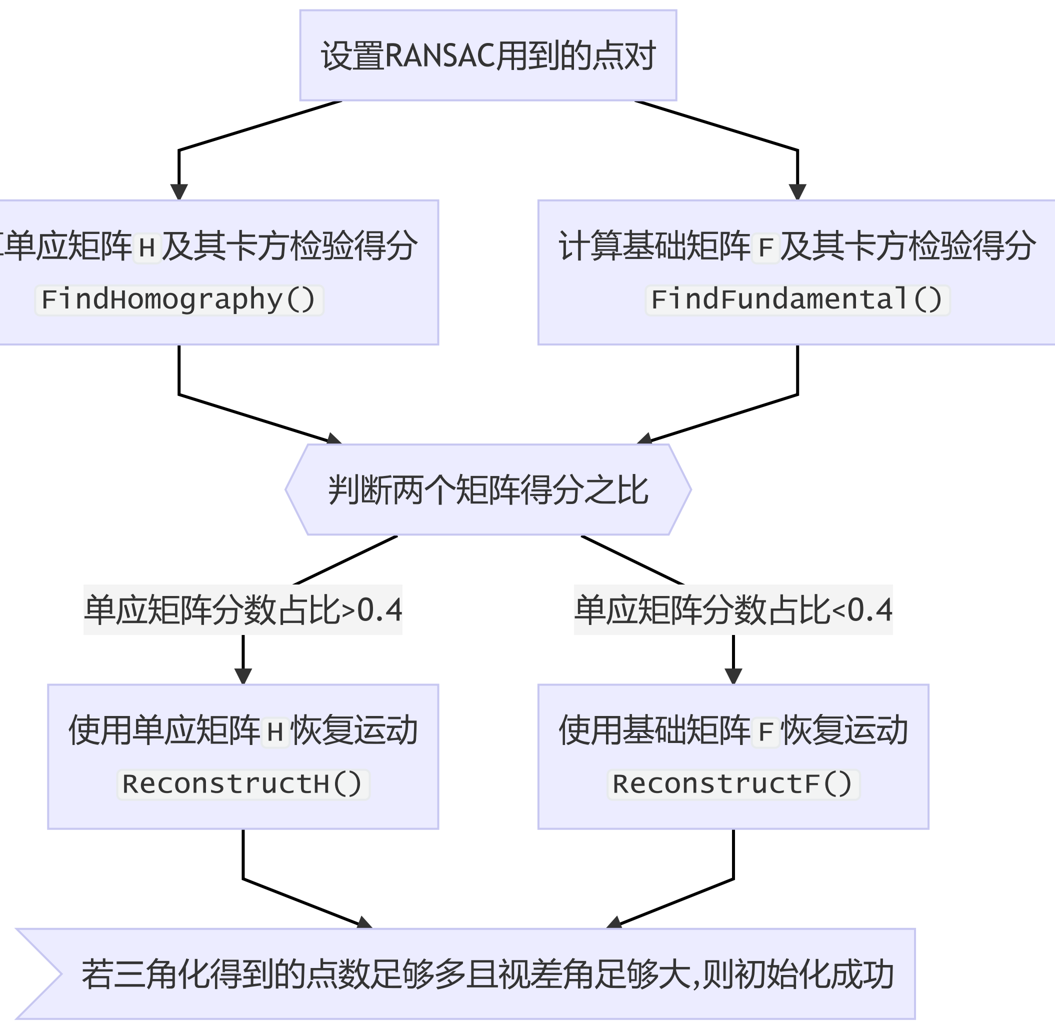
## 极线与极点

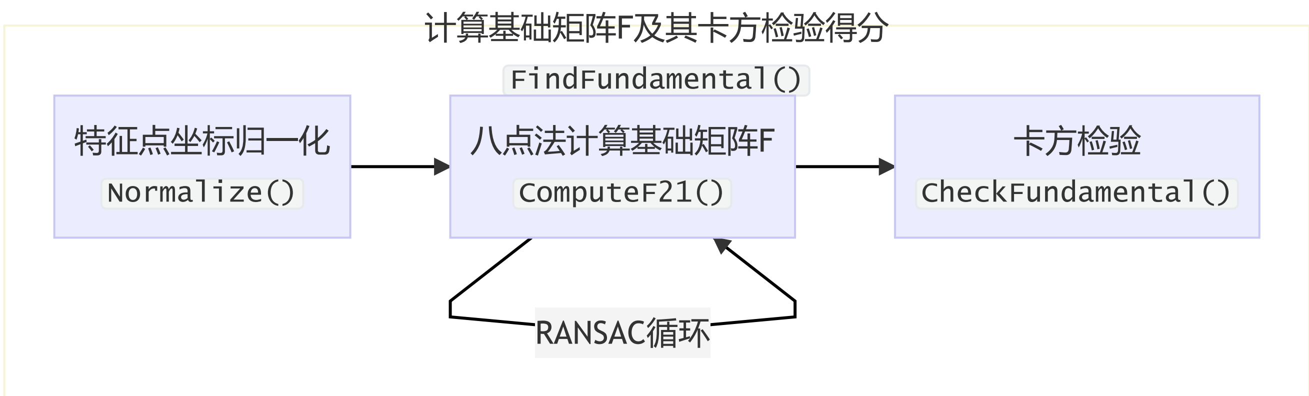


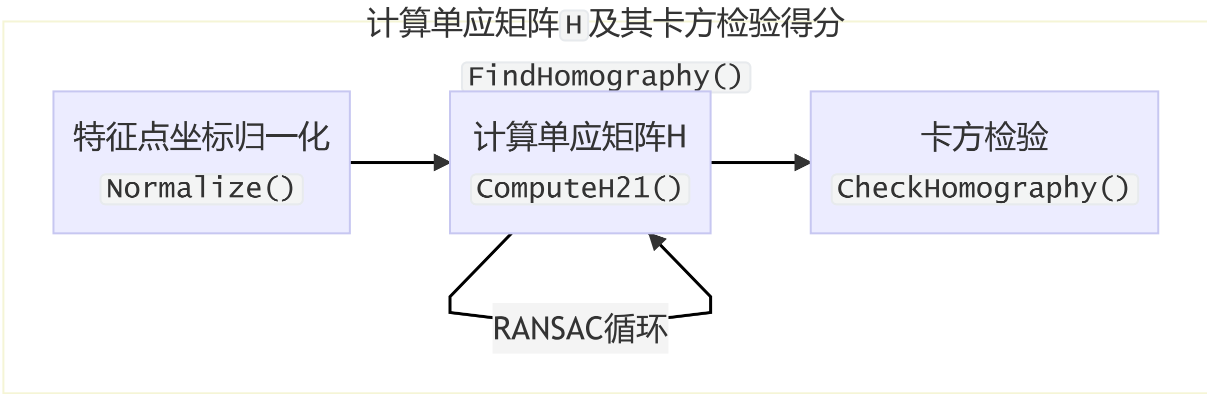



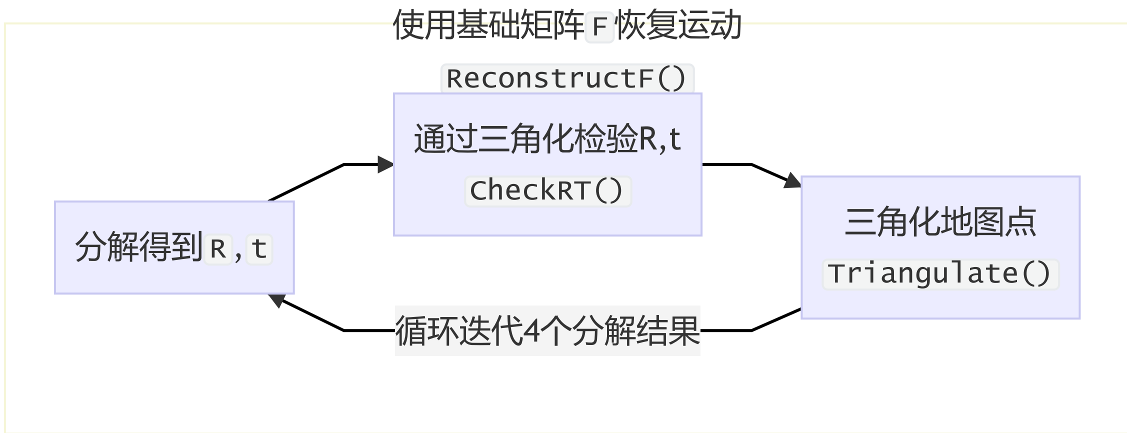


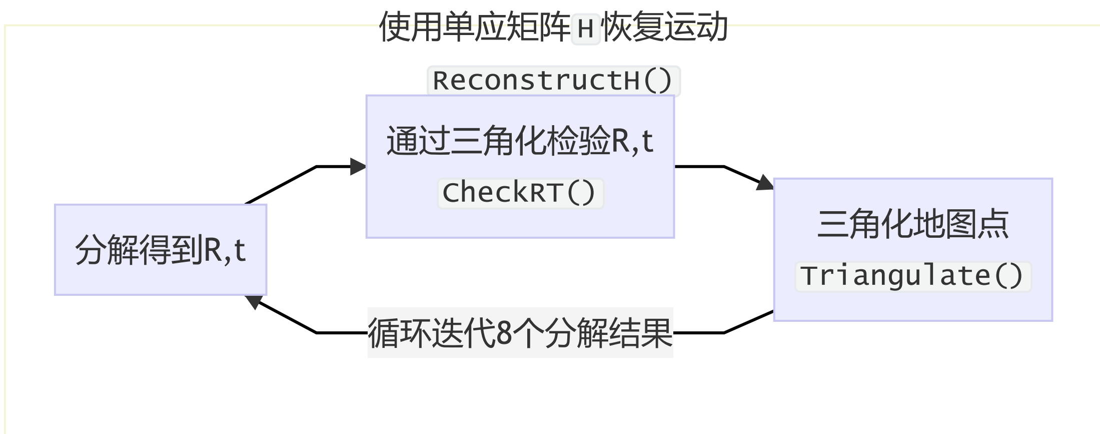

设置RANSAC用到的点对
计算单应矩阵H及其卡方检验得分
FindHomography()
特征点坐标归一化
Normalize()
计算单应矩阵H
ComputeH21()
卡方检验
CheckHomography()
RANSAC循环
计算基础矩阵F及其卡方检验得分
FindFundamental()
特征点坐标归一化
Normalize()
八点法计算基础矩阵F
ComputeF21()
卡方检验
CheckFundamental()
RANSAC循环
判断两个矩阵得分之比
单应矩阵分数占比>0.4
使用单应矩阵H恢复运动
ReconstructH()
分解得到R,t
通过三角化检验R,t
CheckRT()
三角化地图点
Triangulate()
循环迭代8个分解结果
单应矩阵分数占比<0.4
使用基础矩阵F恢复运动
ReconstructF()
分解得到R,t
通过三角化检验R,t
CheckRT()
三角化地图点
Triangulate()
循环迭代4个分解结果
若三角化得到的点数足够多且视差角足够大,则初始化成功


设置RANSAC用到的点对
计算单应矩阵H及其卡方检验得分
FindHomography()
计算基础矩阵F及其卡方检验得分
FindFundamental()
判断两个矩阵得分之比
单应矩阵分数占比>0.4
单应矩阵分数占比<0.4
使用单应矩阵H恢复运动
ReconstructH()
使用基础矩阵F恢复运动
ReconstructF()
若三角化得到的点数足够多且视差角足够大,则初始化成功


计算基础矩阵F及其卡方检验得分
FindFundamental()
特征点坐标归一化
Normalize()
八点法计算基础矩阵F
ComputeF21()
卡方检验
CheckFundamental()
RANSAC循环


计算单应矩阵H及其卡方检验得分
FindHomography()
特征点坐标归一化
Normalize()
计算单应矩阵H
ComputeH21()
卡方检验
CheckHomography()
RANSAC循环


使用基础矩阵F恢复运动
ReconstructF()
通过三角化检验R,t
CheckRT()
分解得到R,t
三角化地图点
Triangulate()
循环迭代4个分解结果


使用单应矩阵H恢复运动
ReconstructH()
通过三角化检验R,t
CheckRT()
分解得到R,t
三角化地图点
Triangulate()
循环迭代8个分解结果