

SLAM Trick 1: Epipolar Constraints

陈烁龙

2022 年 4 月 29 日

目录

1	对极几何	3
2	恢复运动	3
2.1	本质矩阵的求解	3
2.2	恢复运动	7
3	三角化	9
4	GitHub	10

插图

1	原始图片	6
2	OpenCV 的匹配关系	6
3	假设检验的匹配关系	7

表格

摘要

在使用单目相机进行 *SLAM* 的时候，必不可少的一步就是初始化。由于单目相机的尺度是未知的，所以一般初始化的方式是通过极几何约束，解算出两帧图像之间的位姿变换，来进行初始化。

关键词：卡方检验，单目相机，对极几何

1 对极几何

对极几何描述了两张影像之间的位姿变换关系，通过解算得到的基础矩阵或者本质矩阵，我们可以从中恢复出位姿。

假设有两张影像 $Frame_1$ 和 $Frame_2$ ，其上有一已配对的特征点对 $p_1(u_1, v_1)$ 和 $p_2(u_2, v_2)$ ，且 p_1 对应的空间点在 $Frame_1$ 的相机坐标系下为 P 。现在我们要求解的是从 $Frame_1$ 到 $Frame_2$ 的位姿变化 R_{21} 和 t_{21} 。我们基于相机的帧孔模型，很容易写出如下的方程组：

$$\begin{cases} s_1 p_1 = K P \\ s_2 p_2 = K(R_{21} P + t_{21}) \end{cases} \quad (1)$$

其中 s_1 、 s_2 代表空间点 P 在各相机坐标系下对应的深度， K 表示相机的内参矩阵（其描述了像平面和归一化像素坐标平面的对应关系）。如果我们将像素点转到对应的归一化像素坐标平面上，也可以得到：

$$\begin{cases} s_1 X_1 = P \\ s_2 X_2 = R_{21} P + t_{21} \end{cases} \quad (2)$$

其中：

$$\begin{cases} X_1 = K^{-1} p_1 \\ X_2 = K^{-1} p_2 \end{cases} \quad (3)$$

即：

$$s_2 X_2 = s_1 R_{21} X_1 + t_{21} \quad (4)$$

对于上式，我们在其左右两边同时左乘 t_{21} 对应的反对称矩阵¹，则有：

$$s_2 t_{21}^\wedge X_2 = s_1 t_{21}^\wedge R_{21} X_1 \quad (5)$$

由于我们并不知道点对应的深度，所以我们需要将深度因子 s_1 、 s_2 消除。为此我们在上式左右两边同时左乘 X_2^T ，由于 $X_2^T t_{21}^\wedge X_2 = 0$ ，所以有：

$$\begin{aligned} 0 &= s_2 X_2^T t_{21}^\wedge X_2 = s_1 X_2^T t_{21}^\wedge R_{21} X_1 \\ &\rightarrow \begin{cases} X_2^T t_{21}^\wedge R_{21} X_1 = 0 \\ p_2^T K^{-T} t_{21}^\wedge R_{21} K^{-1} p_1 = 0 \end{cases} \end{aligned} \quad (6)$$

如果我们记： $E = t_{21}^\wedge R_{21}$ 、 $F = K^{-T} t_{21}^\wedge R_{21} K^{-1}$ ，则有：

$$\begin{cases} X_2^T E X_1 = 0 \\ p_2^T F p_1 = 0 \end{cases} \quad (7)$$

其中矩阵 E 被称为本质矩阵，矩阵 F 被成为基础矩阵，它们只相差了一个相机内参。

2 恢复运动

假设我们要利用本质矩阵 E 来恢复运动，我们可以先基于最小二乘法求解得到矩阵 E ，而后再利用 *SVD* 方法得到旋转矩阵 R_{21} 和平移向量 t_{21} 。

2.1 本质矩阵的求解

本质矩阵 E 的求解是基于上文的对极几何关系式 7 中的第一式来进行的。对于一对已配对的特征点对 $p_1(u_1, v_1)$ 和 $p_2(u_2, v_2)$ ，其对应归一

¹假设有向量 $v(x, y, z)$ ，则其对应的反对称矩阵为：

$$v^\wedge = \begin{pmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{pmatrix}$$

且有： $v^\wedge v = v \times v = 0$

化像素坐标平面上的点 $X_1(x_1, y_1)$ 和 $X_2(x_2, y_2)$, 我们很容易可以写出如下的关系式:

$$\begin{pmatrix} x_2 & y_2 & 1 \end{pmatrix} \begin{pmatrix} e_1 & e_2 & e_3 \\ e_4 & e_5 & e_6 \\ e_7 & e_8 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix} = 0 \quad (8)$$

由于对于式 7 左右乘以任意标量都不会破坏关系, 因此我们直接将矩阵 E 的最后一个元素设为 1, 这不会产生任何影响, 只会方便我们求解。基于式 8 我们可以得到:

$$\begin{aligned} x_1x_2e_1 + x_1y_2e_4 + x_1e_7 + y_1x_2e_2 + y_1y_2e_5 \\ + y_1e_8 + x_2e_3 + y_2e_6 + 1 = 0 \end{aligned} \quad (9)$$

即:

$$\begin{pmatrix} x_1x_2 \\ y_1x_2 \\ x_2 \\ x_1y_2 \\ y_1y_2 \\ y_2 \\ x_1 \\ y_1 \end{pmatrix}^T \begin{pmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \\ e_8 \end{pmatrix} = -1 \quad (10)$$

$$\rightarrow AX = l$$

若有 8 对已配对的点对, 即可求解常规的线性方程组即可解得矩阵 E , 若有多于 8 对已配对的点对, 可使用最小二乘法求解:

$$X = (A^T A)^{-1} A^T l \quad (11)$$

求解出元素后, 最后再拼得矩阵 E 。

当然, 该系列作为 *Slam-Trick*, 一定有一些优化求解的过程。首先, 我们不得不承认, 我们直接通过匹配算法得到的匹配关系有很大部分是错误的。当直接使用这些误匹配进行求解时, 难免会对解算结果有影响。

一方面, 我们首先基于先验的知识, 对匹配进行过滤。我们首先找到匹配距离的最小值, 然后基于该值来判断其他匹配距离是否合理。具体

来说, 我们可以使用两倍的最小距离作为阈值进行过滤。当然, 为了避免过滤得太多, 我们用经验值 30 来作为阈值的下限。即:

$$\begin{cases} m_i \in newMatches, d_i < \max(30, 2d_{min}) \\ m_i \notin newMatches, d_i \geq \max(30, 2d_{min}) \end{cases}$$

这样, 我们就基于原始的匹配关系, 获得了更好的、用于估计基础矩阵的新匹配关系。当然, 具体问题具体分析, 阈值的设定可根据实际的应用场景进行修改。

另一方面, 我们会基于已估计的基础矩阵 (可以通过本质矩阵获得), 对各个匹配关系进行假设检验。我们设:

$$\begin{cases} (a \ b \ c) = p_2^T F \\ e = p_2^T F p_1 = au_1 + bv_1 + c \end{cases}$$

如果我们假设特征点探测的误差满足方差为 1 的正态分布, 即:

$$\begin{cases} u_1 \sim N(\tilde{u}_1, 1^2) \\ v_1 \sim N(\tilde{v}_1, 1^2) \end{cases}$$

那么不难得到:

$$\begin{aligned} e &\sim N(0, a^2 + b^2) \\ \rightarrow Y = \frac{e^2}{a^2 + b^2} &\sim \mathcal{X}^2(1) \end{aligned} \quad (12)$$

如果我们以 0.75 作为显著性水平, 则对应的分位点为 1.323。换句话说, 当:

$$\begin{cases} m_i \in goodMatches, Y_i < 1.323 \\ m_i \notin goodMatches, Y_i \geq 1.323 \end{cases}$$

对于不通过检验的点, 我们将其去除², 而后重新计算参数, 直至所有“好的匹配”都通过的假设检验。

²在下文代码中, 我们是简单的将该匹配对应的系数矩阵行 $A.row(i)$ 置为 0。

下代码列表展示了我们如何基于假设检验的粗差别除策略，估计基础矩阵 F 。首先我们基于经验对初始的匹配关系进行过滤。而后基于过滤后的匹配关系构建线性方程的系数矩阵。之后基于求解得到的参数，不断进行假设检验，直至所有“好的匹配”都通过的假设检验。

Listing 1: 基于对极几何求解基础矩阵

```

1 namespace ns_st2 {
2     /**
3      * @brief to get function matrix based on the epipolar
4      * constraints
5      *
6      * @param kps1 the keypoints in the first image
7      * @param kps2 the keypoints in the second image
8      * @param srcMatches the source matches, It can be matching
9      * data without preprocessing
10     * @param goodMatches the good matches that this algorithm
11     return
12     * @param quantile the quantile to judge whether a match is
13     an outlier
14     * @return Eigen::Matrix3f the function matrix
15     */
16 static Eigen::Matrix3f solveEpipolar(
17     const std::vector<cv::KeyPoint> &kps1,
18     const std::vector<cv::KeyPoint> &kps2,
19     const std::vector<cv::DMatch> &srcMatches,
20     const CameraInnerParam &innerParam,
21     std::vector<cv::DMatch> *goodMatches = nullptr,
22     float quantile = 1.323) {
23
24     CV_Assert(srcMatches.size() >= 8);
25
26     // clean source data
27     std::vector<cv::DMatch> matches;
28     matches.reserve(0.5 * srcMatches.size());
29
30     auto minDisIter = std::min_element(
31         srcMatches.cbegin(), srcMatches.cend(),
32         [](const cv::DMatch &m1, const cv::DMatch &m2) {
33             return m1.distance < m2.distance;
34         });
35
36     for (int i = 0; i != srcMatches.size(); ++i) {
37         // filter bad matches
38         if (srcMatches.at(i).distance < std::max(30.0f, 2.0f *
39             minDisIter->distance)) {
40             matches.push_back(srcMatches.at(i));
41         }
42     }
43
44     // matrices for least square
45     Eigen::MatrixXf matA(matches.size(), 8), vec1 = -Eigen::
46     VectorXf::Ones(matches.size());

```

```

47 Eigen::Vector<float, 8> vecX;
48
49 // record the outliers' index in the matches
50 std::set<int> outliers;
51
52 float fx = innerParam.fx, fy = innerParam.fy, fxInv = 1.0f
53 / fx, fyInv = 1.0f / fy;
54 float cx = innerParam.cx, cy = innerParam.cy;
55
56 Eigen::Matrix3f K = innerParam.toEigenMatrix(), KInv = K.
57 inverse();
58 Eigen::Matrix3f matF, matE;
59
60 // construct the A matrix
61 for (int i = 0; i != matches.size(); ++i) {
62     const auto &match = matches.at(i);
63
64     float u1 = kps1.at(match.queryIdx).pt.x, v1 = kps1.at(
65         match.queryIdx).pt.y;
66     float u2 = kps2.at(match.trainIdx).pt.x, v2 = kps2.at(
67         match.trainIdx).pt.y;
68
69     float x1 = (u1 - cx) * fxInv, y1 = (v1 - cy) * fyInv;
70     float x2 = (u2 - cx) * fxInv, y2 = (v2 - cy) * fyInv;
71
72     matA(i, 0) = x1 * x2, matA(i, 1) = y1 * x2;
73     matA(i, 2) = x2, matA(i, 3) = x1 * y2;
74     matA(i, 4) = y1 * y2, matA(i, 5) = y2;
75     matA(i, 6) = x1, matA(i, 7) = y1;
76 }
77
78 // find outliers [the condition is to ensure that the
79 equation has a solution]
80 while (matches.size() - outliers.size() > 8) {
81
82     // solve
83     vecX = (matA.transpose() * matA).inverse() * matA.
84     transpose() * vec1;
85
86     matE(0, 0) = vecX(0), matE(0, 1) = vecX(1), matE(0, 2) =
87     vecX(2);
88     matE(1, 0) = vecX(3), matE(1, 1) = vecX(4), matE(1, 2) =
89     vecX(5);
90     matE(2, 0) = vecX(6), matE(2, 1) = vecX(7), matE(2, 2) =
91     1.0f;
92
93     matE.normalize();
94
95     matF = KInv.transpose() * matE * KInv;
96
97     // find the badest outliers
98     float maxVar = 0.0;
99     int maxIdx = -1;
100
101     for (int i = 0; i != matches.size(); ++i) {
102         // if current match was a invaild match, then continue
103         if (outliers.count(i) != 0) {

```

```

89     continue;
90 }
91
92 const auto &match = matches.at(i);
93 float u1 = kps1.at(match.queryIdx).pt.x, v1 = kps1.at(
    match.queryIdx).pt.y;
94 float u2 = kps2.at(match.trainIdx).pt.x, v2 = kps2.at(
    match.trainIdx).pt.y;
95
96 Eigen::Vector3f p2(u2, v2, 1.0f);
97 Eigen::Matrix<float, 1, 3> temp = p2.transpose() * matF
    ;
98
99 float a = temp(0, 0), b = temp(0, 1), c = temp(0, 2);
100
101 float num = a * u1 + b * v1 + c;
102 float den = a * a + b * b;
103
104 if (den == 0.0) {
105     continue;
106 }
107
108 float statistics = num * num / den;
109
110 if (statistics < quantile) {
111     // current match is a good match
112     continue;
113 }
114
115 if (maxVar < statistics) {
116     maxVar = statistics;
117     maxIdx = i;
118 }
119 }
120
121 if (maxIdx == -1) {
122     // which means no outliers in current left matches
123     break;
124 } else {
125     // remove the outlier's affect
126     matA.row(maxIdx).setZero();
127     outliers.insert(maxIdx);
128 }
129 }
130
131 if (goodMatches != nullptr) {
132     goodMatches->clear();
133     goodMatches->resize(matches.size() - outliers.size());
134     int count = 0;
135     for (int i = 0; i != matches.size(); ++i) {
136         if (outliers.count(i) == 0) {
137             // it's not a outliers
138             goodMatches->at(count++) = (matches.at(i));
139         }
140     }
141 }
142

```

```

143     return matE;
144 }
145 } // namespace ns_st2

```

图 1 是本次实验使用的原始图片。以下是两种估计结果的对比：



(a) 原始图片一

(b) 原始图片二

图 1: 原始图片

1. OpenCV 估计

图 2 为使用 *OpenCV* 提供的函数，基于 8 点法进行的估计的参考匹配关系。

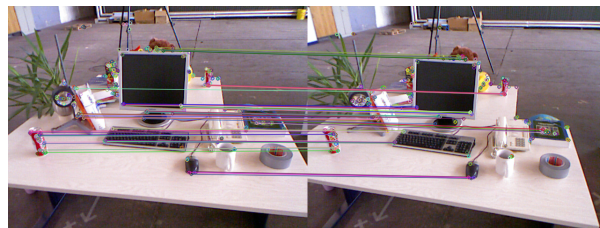


图 2: OpenCV 的匹配关系

列表 2 为估计的结果统计信息。具体的分析留在下文。

Listing 2: OpenCV 结果

```

1  {'cost time': 10.38270(MS)}
2  matched points: 75
3  essential matrix:
4  [0.009476120410375273, 0.2124514409709818,
    0.1143379074350115;
5  -0.1982554336303657, 0.03327101904283745,
    -0.669375737179301;
6  -0.06928586652949775, 0.6693726556233317,
    0.01910739912399924]
7  mean: -0.000195134
8  errorMeanNorm: 0.00124172
9  sigma: 0.00163348

```

2. 假设检验估计

图 3 为使用假设检验估计后得到的最佳匹配关系。其相较于 *OpenCV* 的匹配关系更少，但是足以估计基础矩阵 F 。

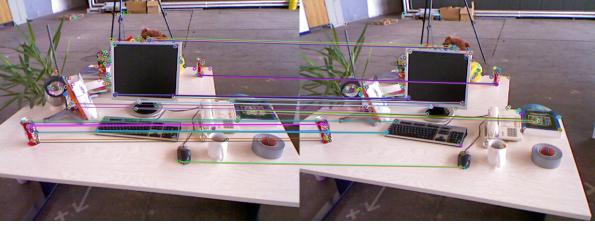


图 3: 假设检验的匹配关系

列表 3 为估计的结果统计信息。具体的分析留在下文。

Listing 3: 假设检验结果

```
1 {'cost time': 0.17586(MS)}
2 matched points: 59
3 essential matrix:
4 0.0238687 0.565404 0.213721
5 -0.509386 0.08651 -0.389609
6 -0.17464 0.429546 0.0176223
7 mean: 1.25362e-05
8 errorMeanNorm: 0.000383489
9 sigma: 0.000474903
```

在列表 2 和列表 3 中，罗列了一些事后的统计量。可以看到，*OpenCV* 耗时比假设检验多了将近 60 倍。*OpenCV* 使用了 75 个匹配关系进行估计，统计检验使用了 59 个匹配关系进行估计。*OpenCV* 的误差均值（即公式 12 中的 e ）偏离真值 0 比假设检验多。*OpenCV* 的绝对误差均值和标准差比假设检验的要大。可见，在精确度方面，*OpenCV* 略占劣势，在效率上远不及假设检验³。

2.2 恢复运动

在求解得到本质矩阵 E 之后，我们可以对本质矩阵 E 进行 *SVD* 分解：

$$E = U\Sigma V^T$$

³我们认为这会对效率产生较大的影响。

进而有四种解：

$$\begin{aligned} &\rightarrow \begin{cases} t_1^\wedge = UR_Z(\frac{\pi}{2})\Sigma U^T \\ R_1 = UR_Z^T(\frac{\pi}{2})V^T \end{cases} \\ &\rightarrow \begin{cases} t_2^\wedge = -UR_Z(\frac{\pi}{2})\Sigma U^T \\ R_2 = UR_Z^T(\frac{\pi}{2})V^T \end{cases} \\ &\rightarrow \begin{cases} t_3^\wedge = UR_Z(-\frac{\pi}{2})\Sigma U^T \\ R_3 = UR_Z^T(-\frac{\pi}{2})V^T \end{cases} \\ &\rightarrow \begin{cases} t_4^\wedge = -UR_Z(-\frac{\pi}{2})\Sigma U^T \\ R_4 = UR_Z^T(-\frac{\pi}{2})V^T \end{cases} \end{aligned}$$

其中 $R_Z(\frac{\pi}{2})$ 表示沿着 Z 轴旋转 90 度得到的旋转矩阵。当然，一般我们进行对本质矩阵 E 的 *SVD* 时，理论上矩阵 $\Sigma = \text{diag}(\sigma, \sigma, 0)$ ，但是实际情况会有差别，这时我们可以重新构造矩阵 $\Sigma = \text{diag}(\frac{\sigma_1+\sigma_2}{2}, \frac{\sigma_1+\sigma_2}{2}, 0)$ 。

但是运动是唯一的，上述四种解中只有一个是正确的。所以我们还需要对某个匹配点对进行三角化，求解其深度，进而排除三个误解，得到最终的解。

列表 4 为使用本质矩阵 E 来实现的运动恢复。在代码中，我们首先对本质矩阵 E 进行 *SVD* 分解，然后构造四种解的情况，并逐一通过三角化的方式进行检查。一旦发现符合满足条件的解，则返回结果。

Listing 4: 恢复运动

```
1 namespace ns_st2 {
2     /**
3      * @brief recovery the movement from the essential matrix
4      *
5      * @param eMatrix the essential matrix
6      * @param K the camera's inner parameters
7      * @param kp1 the key point in first frame
8      * @param kp2 the key point in second frame
9      * @param rot21 rotation matrix from first frame to second
          frame
10     * @param t21 translation matrix from first frame to second
          frame
11     * @return true the process is successful
12     * @return false the process is failed
```

```

13  */
14  static bool recoveryMove(
15      const Eigen::Matrix3f &eMatrix,
16      const Eigen::Matrix3f &K,
17      const cv::KeyPoint &kp1,
18      const cv::KeyPoint &kp2,
19      Eigen::Matrix3f &rot21,
20      Eigen::Vector3f &t21) {
21
22      // SVD decomposition
23      Eigen::JacobiSVD<Eigen::Matrix3f> svd(eMatrix.normalized()
24          , Eigen::ComputeFullU | Eigen::ComputeFullV);
25      Eigen::Vector3f singularVal = svd.singularValues();
26      Eigen::Matrix3f uMatrix = svd.matrixU();
27      Eigen::Matrix3f vMatrix = svd.matrixV();
28
29      // normalize sigma matrix
30      float sigma = 0.5f * (singularVal(0) + singularVal(1));
31      Eigen::Matrix3f sigmaMatrix = Eigen::Matrix3f::Zero();
32      sigmaMatrix(0, 0) = sigmaMatrix(1, 1) = sigma;
33
34      // temp matrices
35      Eigen::Matrix3f pRotMat = Eigen::Matrix3f::Zero();
36      pRotMat(0, 1) = -1.0f, pRotMat(1, 0) = 1.0, pRotMat(2, 2)
37          = 1.0f;
38
39      Eigen::Matrix3f nRotMat = Eigen::Matrix3f::Zero();
40      nRotMat(0, 1) = 1.0f, nRotMat(1, 0) = -1.0, nRotMat(2, 2)
41          = 1.0f;
42
43      // to normalize a rotation matrix
44      auto normRot = [](Eigen::Matrix3f &rot) -> void {
45          // for loop two times
46          for (int i = 0; i != 2; ++i) {
47              // normalize rows
48              Eigen::Vector3f row1 = rot.row(0);
49              Eigen::Vector3f row2 = rot.row(1).normalized();
50              Eigen::Vector3f row3 = row1.cross(row2).normalized();
51              row1 = row2.cross(row3);
52              rot.row(0) = row1;
53              rot.row(1) = row2;
54              rot.row(2) = row3;
55
56              // normalize cols
57              Eigen::Vector3f col1 = rot.col(0);
58              Eigen::Vector3f col2 = rot.col(1).normalized();
59              Eigen::Vector3f col3 = col1.cross(col2).normalized();
60              col1 = col2.cross(col3);
61              rot.col(0) = col1;
62              rot.col(1) = col2;
63              rot.col(2) = col3;
64          }
65      };
66
67      // check a solution is right
68      auto checkSolution = [&kp1, &kp2, &K, &rot21, &t21](const
69          Eigen::Matrix3f &rot, const Eigen::Vector3f &t) ->

```

```

        bool {
66          // compute the depth
67          std::pair<float, float> depth = triangulation(kp1, kp2, K
68              , rot, t);
69
70          // if two values are positive
71          if (depth.first > 0.0f && depth.second > 0.0f) {
72              rot21 = rot;
73              t21 = t;
74              return true;
75          } else {
76              return false;
77          }
78      };
79
80      // different solutions
81      // solution 1
82      Eigen::Matrix3f R1 = uMatrix * pRotMat.transpose() *
83          vMatrix.transpose();
84      normRot(R1);
85      Eigen::Vector3f t1 = ns_st0::antisymmetric(Eigen::Matrix3f
86          (uMatrix * pRotMat * sigmaMatrix * uMatrix.transpose
87          ())).normalized();
88      if (checkSolution(R1, t1)) {
89          return true;
90      }
91
92      // solution 2
93      Eigen::Matrix3f R2 = R1;
94      Eigen::Vector3f t2 = -t1;
95      if (checkSolution(R2, t2)) {
96          return true;
97      }
98
99      // solution 3
100      Eigen::Matrix3f R3 = uMatrix * nRotMat.transpose() *
101          vMatrix.transpose();
102      normRot(R3);
103      Eigen::Vector3f t3 = ns_st0::antisymmetric(Eigen::Matrix3f
104          (uMatrix * nRotMat * sigmaMatrix * uMatrix.transpose
105          ())).normalized();
106      if (checkSolution(R3, t3)) {
107          return true;
108      }
109
110      // solution 4
111      Eigen::Matrix3f R4 = R3;
112      Eigen::Vector3f t4 = -t3;
113      if (checkSolution(R4, t4)) {
114          return true;
115      }
116
117      return false;
118    }
119 } // namespace ns_st2

```

列表 7 为基于上文获得的本质矩阵解算得

到的位姿变换。

Listing 5: 解算结果

```
1 rotation matrix
2   0.994912 -0.0930117 0.0387284
3   0.0912133 0.994774 0.045872
4  -0.0427926 -0.0421061 0.998196
5 translation vector
6  -0.554473 -0.284364 0.78211
```

3 三角化

我们已知 4 中的旋转矩阵和平移向量，所以可在其基础上左乘 X_2 对应的反对称矩阵 X_2^\wedge 。即：

$$0 = s_2 X_2^\wedge X_2 = s_1 X_2^\wedge R_{21} X_1 + X_2^\wedge t_{21}$$

如果令：

$$\begin{cases} A = X_2^\wedge R_{21} X_1 \\ l = -X_2^\wedge t_{21} \end{cases}$$

那么 s_1 可由最小二乘法求解得到：

$$s_1 = (A^T A)^{-1} A^T l \quad (13)$$

求解得到 s_1 后，可同样采用最小二乘法求解 s_2 ：

$$\begin{cases} B = X_2 \\ m = s_1 R_{21} X_1 + t_{21} \\ s_2 = (B^T B)^{-1} B^T m \end{cases} \quad (14)$$

由此，该匹配对所对应的空间点在相机 1 的坐标系下的坐标为 $P_1 = s_1 X_1$ ，在相机 2 的坐标系下的坐标为 $P_2 = s_2 X_2$ 。

列表 6 为三角化的代码。我们实现了两个版本、不同参数的接口，其中第二个版本是依托于第一个版本实现的。在第一个版本中，我们先使用最小二乘法计算点在第一个相机中的深度 s_1 ，而后再计算点在第二个相机中的深度 s_2 。很明显，第一个版本效率更高，但是第二个版本使用更简单。

Listing 6: 三角化

```
1 namespace ns_st2 {
2   /**
3    * @brief to trangular a pair points on the normalized
4    *        coordinate
5    *
6    * @param X1 the point on the first camera's normalized
7    *        coordinate
8    * @param X2 the point on the second camera's normalized
9    *        coordinate
10    * @param rot21 the rotation from first camera to second
11    *        camera
12    * @param t21 the translation from first camera to second
13    *        camera
14    * @param P1 the point on the first camera's coordinate
15    * @param P2 the point on the second camera's coordinate
16    * @return std::pair<float, float> the depth pair
17    */
18 static std::pair<float, float> triangulation(
19     const Eigen::Vector3f &X1,
20     const Eigen::Vector3f &X2,
21     const Eigen::Matrix3f &rot21,
22     const Eigen::Vector3f &t21,
23     Eigen::Vector3f *P1 = nullptr,
24     Eigen::Vector3f *P2 = nullptr) {
25
26     // the depth
27     float s1, s2;
28
29     // to solve s1
30     Eigen::Vector3f aVec = ns_st0::antisymmetric(X2) * rot21 *
31         X1;
32     Eigen::Vector3f lVec = -ns_st0::antisymmetric(X2) * t21;
33     s1 = ((aVec.transpose() * aVec).inverse() * aVec.transpose()
34         * lVec)(0, 0);
35
36     // to solve s2
37     Eigen::Vector3f bVec = X2;
38     Eigen::Vector3f mVec = s1 * rot21 * X1 + t21;
39     s2 = ((bVec.transpose() * bVec).inverse() * bVec.transpose()
40         * mVec)(0, 0);
41
42     if (P1 != nullptr) {
43         *P1 = s1 * X1;
44     }
45
46     if (P2 != nullptr) {
47         *P2 = s2 * X2;
48     }
49
50     return {s1, s2};
51 }
52
53 /**
54 * @brief to trangular a pair points on the pixel coordinate
55 *
56 * @param P1 the point on the first camera's pixel
```

```

coordinate
49 * @param P2 the point on the second camera's pixel
coordinate
50 * @param K the camera's inner parameter matrix
51 * @param rot21 the rotation from first camera to second
camera
52 * @param t21 the translation from first camera to second
camera
53 * @param P1 the point on the first camera's coordinate
54 * @param P2 the point on the second camera's coordinate
55 * @return std::pair<float, float> the depth pair
56 */
57 static std::pair<float, float> triangulation(
58     const cv::KeyPoint &p1,
59     const cv::KeyPoint &p2,
60     const Eigen::Matrix3f &K,
61     const Eigen::Matrix3f &rot21,
62     const Eigen::Vector3f &t21,
63     Eigen::Vector3f *P1 = nullptr,
64     Eigen::Vector3f *P2 = nullptr) {
65     Eigen::Vector3f X1 = K.inverse() * Eigen::Vector3f(p1.pt.x
66         , p1.pt.y, 1.0f);
67     Eigen::Vector3f X2 = K.inverse() * Eigen::Vector3f(p2.pt.x
68         , p2.pt.y, 1.0f);
69     return triangulation(X1, X2, rot21, t21, P1, P2);
70 }
71 } // namespace ns_st2

```

例如，我们上文在解算运动的时候使用到了三角化，当时使用的点的深度如下所示⁴：

Listing 7: 解算结果

```

1 point's depth which used to check soluation
2 6.04497, 6.87744

```

4 GitHub

以下链接为该项目在 *GitHub* 上的地址，点击他，克隆它，使用它：

<https://github.com/Unsigned-Long/slam-tricks/tree/master/st2-epipolar>

⁴注意：由于单目相机尺度不确定，所以我们将解算得到的平移向量规范化。换句话说，表中的深度的单位是 1(个初始解算平移向量)。