SLAM Trick 1: Epipolar Constraints

陈烁龙

2022 年 4 月 30 日

目录

插图

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对极几何约束，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：

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

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：

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

其中：

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

即：

$$s_2 X_2 = s_1 R_{21} X_1 + t_{21} \tag{4}$$

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：

$$s_2 t_{21}^{\wedge} X_2 = s_1 t_{21}^{\wedge} R_{21} X_1 \tag{5}$$

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：

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

$$\rightarrow \begin{cases} X_2^T t_{21}^{\wedge} R_{21} X_1 = 0 \\ p_2^T K^{-T} t_{21}^{\wedge} R_{21} K^{-1} p_1 = 0 \end{cases} \tag{6}$$

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

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

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：

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

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：

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

即：

$$\begin{aligned} \begin{pmatrix} x_1x_2 \\ y_1x_2 \\ x_2 \\ x_1y_2 \\ y_1y_2 \\ y_2 \\ x_1 \\ y_1 \end{pmatrix}^T \begin{pmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \\ e_8 \end{pmatrix} = -1 \\ \rightarrow AX = l \end{aligned} \tag{10}$$

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：

$$X = (A^TA)^{-1}A^Tl \tag{11}$$

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：

$$\begin{cases} \begin{pmatrix} a & b & c \end{pmatrix} = p_2^TF \\ e = p_2^TFp_1 = au_1 + bv_1 + c \end{cases}$$

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[2]，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：

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

但是，在提取特征的时候，我们为了保证特征点的尺度不变性，我们使用了图像金字塔，导致不同层提取得到的特征点的误差不同。越靠上层[3]，特征点探测的误差相对于第 1 层应该越大。假设相邻层之间的尺度因子[4] $sf = 1.2$，那么对于第 i 层来说，其特征点探测的误差应该为

$$\sigma_i^2 = 1.2^{2(i-1)}$$

[2]事实上这和具体使用的特帧点检测算法有关。实验中，我们使用了 $FAST$ 提取特征点，其误差为 1 像素。

[3]这里指的是图像被缩小得越厉害的层。

[4]即相邻上层相对于下层被缩小的倍数。

比如对于第 1 层，$i=1$，$\sigma_1^2=1.0$；对于第 2 层，$i=2$，$\sigma_2^2=1.44$。以此类推。

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：

$$\begin{cases} m_i \in goodMatches, Y_i < 1.323\sigma_i^2 \\ m_i \notin goodMatches, Y_i \geq 1.323\sigma_i^2 \end{cases}$$

对于不通过检验的点，我们将其去除[5]，而后重新计算参数，直至所有“好的匹配”都通过的假设检验。

下代码列表展示了我们如何基于假设检验的粗差剔除策略，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

```cpp
namespace ns_st2 {
  /**
   * @brief to get function matrix based on the epipolar constraints
   *
   * @param kps1 the keypoints in the first image
   * @param kps2 the keypoints in the second image
   * @param srcMatches the source matches, It can be matching data without preprocessing
   * @param goodMatches the good matches that this algorithm return
   * @param quantile the quantile to judge whether a match is an outlier
   * @param scaleFactor the scale factor used in orb feature algorithm
   * @param layerNum the layer total quantity in orb feature algorithm
   * @return Eigen::Matrix3f the function matrix
   */
  static Eigen::Matrix3f solveEpipolar(
      const std::vector<cv::KeyPoint> &kps1,
      const std::vector<cv::KeyPoint> &kps2,
      const std::vector<cv::DMatch> &srcMatches,
      const CameraInnerParam &innerParam,
      std::vector<cv::DMatch> *goodMatches = nullptr,
      float quantile = 1.323, float scaleFactor = 1.2f, uint layerNum = 8) {

    CV_Assert(srcMatches.size() >= 8);

    // clean source data
    std::vector<cv::DMatch> matches;
    matches.reserve(0.5 * srcMatches.size());

    auto minDisIter = std::min_element(
        srcMatches.cbegin(), srcMatches.cend(),
        [](const cv::DMatch &m1, const cv::DMatch &m2) {
          return m1.distance < m2.distance;
        });

    for (int i = 0; i != srcMatches.size(); ++i) {
      // filter bad matches
      if (srcMatches.at(i).distance < std::max(30.0f, 2.0f * minDisIter->distance)) {
        matches.push_back(srcMatches.at(i));
      }
    }

    // matrices for least square
    Eigen::MatrixXf matA(matches.size(), 8), vecl = -Eigen::VectorXf::Ones(matches.size());
    Eigen::Vector<float, 8> vecX;

    // record the outliers' index in the matches
    std::set<int> outliers;

    float fx = innerParam.fx, fy = innerParam.fy, fxInv = 1.0f / fx, fyInv = 1.0f / fy;
    float cx = innerParam.cx, cy = innerParam.cy;

    // key point on different layer with different sigma
    std::vector<float> sigma2(layerNum);
    sigma2.front() = 1.0f;
    float layerScale = 1.0;
    for (int i = 1; i != layerNum; ++i) {
      layerScale *= scaleFactor;
      sigma2.at(i) = layerScale * layerScale;
    }

    Eigen::Matrix3f K = innerParam.toEigenMatrix(), KInv = K.inverse();
    Eigen::Matrix3f matF, matE;

    // construct the A matrix
    for (int i = 0; i != matches.size(); ++i) {
      const auto &match = matches.at(i);
      const cv::KeyPoint &kp1 = kps1.at(match.queryIdx);
      const cv::KeyPoint &kp2 = kps2.at(match.trainIdx);

      float u1 = kp1.pt.x, v1 = kp1.pt.y;
      float u2 = kp2.pt.x, v2 = kp2.pt.y;

      float x1 = (u1 - cx) * fxInv, y1 = (v1 - cy) * fyInv;
      float x2 = (u2 - cx) * fxInv, y2 = (v2 - cy) * fyInv;
```

[5] 在代码中，我们是简单的将该匹配对应的系数矩阵行 $A.row(i)$ 置为 0。

```
      matA(i, 0) = x1 * x2, matA(i, 1) = y1 * x2;
      matA(i, 2) = x2, matA(i, 3) = x1 * y2;
      matA(i, 4) = y1 * y2, matA(i, 5) = y2;
      matA(i, 6) = x1, matA(i, 7) = y1;
    }

    // find outliers [the condition is to ensure that the equation has a solution]
    while (matches.size() - outliers.size() > 8) {

      // solve
      vecX = (matA.transpose() * matA).inverse() * matA.transpose() * vecl;

      matE(0, 0) = vecX(0), matE(0, 1) = vecX(1), matE(0, 2) = vecX(2);
      matE(1, 0) = vecX(3), matE(1, 1) = vecX(4), matE(1, 2) = vecX(5);
      matE(2, 0) = vecX(6), matE(2, 1) = vecX(7), matE(2, 2) = 1.0f;

      matE.normalize();

      matF = KInv.transpose() * matE * KInv;

      // find the badest outliers
      float maxVar = 0.0;
      int maxIdx = -1;

      for (int i = 0; i != matches.size(); ++i) {
        // if current match was a invaild match, then continue
        if (outliers.count(i) != 0) {
          continue;
        }

        const auto &match = matches.at(i);
        const cv::KeyPoint &kp1 = kps1.at(match.queryIdx);
        const cv::KeyPoint &kp2 = kps2.at(match.trainIdx);

        float u1 = kp1.pt.x, v1 = kp1.pt.y;
        float u2 = kp2.pt.x, v2 = kp2.pt.y;

        Eigen::Vector3f p2(u2, v2, 1.0f);
        Eigen::Matrix<float, 1, 3> temp = p2.transpose() * matF;

        float a = temp(0, 0), b = temp(0, 1), c = temp(0, 2);

        float num = a * u1 + b * v1 + c;
        float den = a * a + b * b;

        if (den == 0.0) {
          continue;
        }

        float statistics = num * num / den;

        if (statistics < quantile * sigma2.at(kps2.at(match.trainIdx).octave)) {
          // current match is a good match
          continue;
        }

        if (maxVar < statistics) {
          maxVar = statistics;
          maxIdx = i;
        }
      }

      if (maxIdx == -1) {
        // which means no outliers in current left matches
        break;
      } else {
        // remove the outlier's affect
        matA.row(maxIdx).setZero();
        outliers.insert(maxIdx);
      }
    }

    if (goodMatches != nullptr) {
      goodMatches->clear();
      goodMatches->resize(matches.size() - outliers.size());
      int count = 0;
      for (int i = 0; i != matches.size(); ++i) {
        if (outliers.count(i) == 0) {
          // it's not a outliers
          goodMatches->at(count++) = (matches.at(i));
        }
      }
```

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。

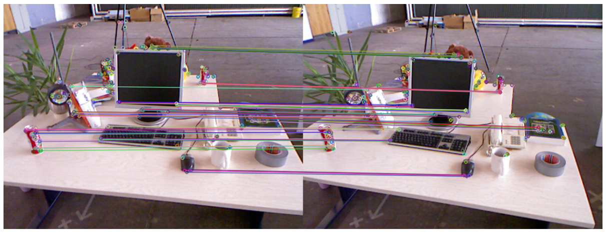

图 2: OpenCV 的匹配关系

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

```
{'cost time': 10.38270(MS)}
matched points: 75
essential matrix:
[0.009476120410375273, 0.2124514409709818,
     0.1143379074350115;
 -0.1982554336303657, 0.03327101904283745,
      -0.669375737179301;
 -0.06928586652949775, 0.6693726556233317,
      0.01910739912399924]
mean: -0.000195134
errorMeanNorm: 0.00124172
sigma: 0.00163348
```

2. 假设检验估计

图 3 为使用假设检验估计后的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。

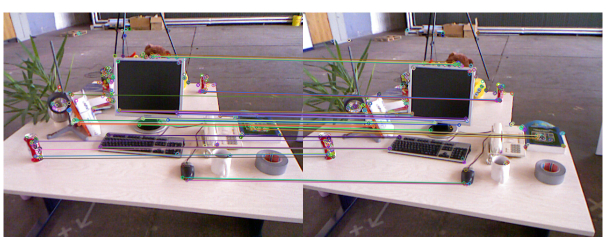

图 3: 假设检验的匹配关系

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

```
{'cost time': 0.17586(MS)}
matched points: 59
essential matrix:
0.0238687 0.565404 0.213721
-0.509386 0.08651 -0.389609
 -0.17464 0.429546 0.0176223
mean: 1.25362e-05
errorMeanNorm: 0.000383489
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[6]。

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：

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

进而有四种解：

$$\rightarrow \begin{cases} t_1^{\wedge} = UR_Z(\frac{\pi}{2})\Sigma U^T \\ R_1 = UR_Z^T(\frac{\pi}{2})V^T \end{cases}$$

$$\rightarrow \begin{cases} t_2^{\wedge} = -UR_Z(\frac{\pi}{2})\Sigma U^T \\ R_2 = UR_Z^T(\frac{\pi}{2})V^T \end{cases}$$

$$\rightarrow \begin{cases} t_3^{\wedge} = UR_Z(-\frac{\pi}{2})\Sigma U^T \\ R_3 = UR_Z^T(-\frac{\pi}{2})V^T \end{cases}$$

$$\rightarrow \begin{cases} t_4^{\wedge} = -UR_Z(-\frac{\pi}{2})\Sigma U^T \\ R_4 = UR_Z^T(-\frac{\pi}{2})V^T \end{cases}$$

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 $\Sigma = diag(\sigma, \sigma, 0)$，但

[6] 我们认为这会对效率产生较大的影响。

是实际情况会有差别，这时我们可以重新构造矩阵 $\Sigma = diag(\frac{\sigma_1+\sigma_2}{2}, \frac{\sigma_1+\sigma_2}{2}, 0)$。

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

```cpp
namespace ns_st2 {
 /**
  * @brief recovery the movement from the essential matrix
  *
  * @param eMatrix the essential matrix
  * @param K the camera's inner paramters
  * @param kp1 the key point in first frame
  * @param kp2 the key point in second frame
  * @param rot21 rotation matrix from first frame to second frame
  * @param t21 translation matrix from first frame to second frame
  * @return true the process is successful
  * @return false the process is failed
  */
 static bool recoveryMove(
     const Eigen::Matrix3f &eMatrix,
     const Eigen::Matrix3f &K,
     const cv::KeyPoint &kp1,
     const cv::KeyPoint &kp2,
     Eigen::Matrix3f &rot21,
     Eigen::Vector3f &t21) {

   // SVD decomposition
   Eigen::JacobiSVD<Eigen::Matrix3f> svd(eMatrix.normalized(), Eigen::ComputeFullU | Eigen::ComputeFullV);
   Eigen::Vector3f singularVal = svd.singularValues();
   Eigen::Matrix3f uMatrix = svd.matrixU();
   Eigen::Matrix3f vMatrix = svd.matrixV();

   if (uMatrix.determinant() < 0.0f) {
     uMatrix *= -1.0f;
   }
   if (vMatrix.determinant() < 0.0f) {
     vMatrix *= -1.0f;
   }

   // normalize sigma matrix
   float sigma = 0.5f * (singularVal(0) + singularVal(1));
   Eigen::Matrix3f sigmaMatrix = Eigen::Matrix3f::Zero();
   sigmaMatrix(0, 0) = sigmaMatrix(1, 1) = sigma;

   // temp matrices
   Eigen::Matrix3f pRotMat = Eigen::Matrix3f::Zero();
   pRotMat(0, 1) = -1.0f, pRotMat(1, 0) = 1.0, pRotMat(2, 2) = 1.0f;

   Eigen::Matrix3f nRotMat = Eigen::Matrix3f::Zero();
   nRotMat(0, 1) = 1.0f, nRotMat(1, 0) = -1.0, nRotMat(2, 2) = 1.0f;

   // to normalize a rotation matrix
   auto normRot = [](Eigen::Matrix3f &rot) -> void {
     // for loop two times
     for (int i = 0; i != 2; ++i) {
       // normalize rows
       Eigen::Vector3f row1 = rot.row(0);
       Eigen::Vector3f row2 = rot.row(1).normalized();
       Eigen::Vector3f row3 = row1.cross(row2).normalized();
       row1 = row2.cross(row3);
       rot.row(0) = row1;
       rot.row(1) = row2;
       rot.row(2) = row3;

       // normalize cols
       Eigen::Vector3f col1 = rot.col(0);
       Eigen::Vector3f col2 = rot.col(1).normalized();
       Eigen::Vector3f col3 = col1.cross(col2).normalized();
       col1 = col2.cross(col3);
       rot.col(0) = col1;
       rot.col(1) = col2;
       rot.col(2) = col3;
     }
   };

   // check a solution is right
   auto checkSolution = [&kp1, &kp2, &K, &rot21, &t21](const Eigen::Matrix3f &rot, const Eigen::Vector3f &t) -> bool {
     // compute the depth
     std::pair<float, float> depth = triangulation(kp1, kp2, K, rot, t);

     // if two values are positive
     if (depth.first > 0.0f && depth.second > 0.0f) {
       rot21 = rot;
       t21 = t;
       return true;
     } else {
       return false;
     }
   };

   // different solutions
   // solution 1
```

```
    Eigen::Matrix3f R1 = uMatrix * pRotMat.transpose() * vMatrix.transpose();
    normRot(R1);
    Eigen::Vector3f t1 = ns_st0::antisymmetric(Eigen::Matrix3f(uMatrix * pRotMat * sigmaMatrix * uMatrix.transpose())).normalized();
    if (checkSolution(R1, t1)) {
      return true;
    }

    // solution 2
    Eigen::Matrix3f R2 = R1;
    Eigen::Vector3f t2 = -t1;
    if (checkSolution(R2, t2)) {
      return true;
    }

    // solution 3
    Eigen::Matrix3f R3 = uMatrix * nRotMat.transpose() * vMatrix.transpose();
    normRot(R3);
    Eigen::Vector3f t3 = ns_st0::antisymmetric(Eigen::Matrix3f(uMatrix * nRotMat * sigmaMatrix * uMatrix.transpose())).normalized();
    if (checkSolution(R3, t3)) {
      return true;
    }

    // solution 4
    Eigen::Matrix3f R4 = R3;
    Eigen::Vector3f t4 = -t3;
    if (checkSolution(R4, t4)) {
      return true;
    }

    return false;
  }
} // 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

```
rotation matrix
  0.994912 -0.0930117 0.0387284
 0.0912133 0.994774 0.045872
-0.0427926 -0.0421061 0.998196
translation vector
-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：

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

求解得到 s_1 后，可同样采用最小二乘法求解 s_2：

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

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

```
namespace ns_st2 {
  /**
   * @brief to trangular a pair points on the normalized coordinate
   *
   * @param X1 the point on the first camera's normalized coordinate
   * @param X2 the point on the second camera's normalized coordinate
   * @param rot21 the rotation from first camera to second camera
```

```
 * @param t21 the translation from first camera to second camera
 * @param P1 the point on the first camera's coordinate
 * @param P2 the point on the second camera's coordinate
 * @return std::pair<float, float> the depth pair
 */
static std::pair<float, float> triangulation(
    const Eigen::Vector3f &X1,
    const Eigen::Vector3f &X2,
    const Eigen::Matrix3f &rot21,
    const Eigen::Vector3f &t21,
    Eigen::Vector3f *P1 = nullptr,
    Eigen::Vector3f *P2 = nullptr) {

  // the depth
  float s1, s2;

  // to solve s1
  Eigen::Vector3f aVec = ns_st0::antisymmetric(X2) * rot21 * X1;
  Eigen::Vector3f lVec = -ns_st0::antisymmetric(X2) * t21;
  s1 = ((aVec.transpose() * aVec).inverse() * aVec.transpose() * lVec)(0, 0);

  // to solve s2
  Eigen::Vector3f bVec = X2;
  Eigen::Vector3f mVec = s1 * rot21 * X1 + t21;
  s2 = ((bVec.transpose() * bVec).inverse() * bVec.transpose() * mVec)(0, 0);

  if (P1 != nullptr) {
    *P1 = s1 * X1;
  }

  if (P2 != nullptr) {
    *P2 = s2 * X2;
  }

  return {s1, s2};
}

/**
 * @brief to trangular a pair points on the pixel coordinate
 *
 * @param P1 the point on the first camera's pixel coordinate
 * @param P2 the point on the second camera's pixel coordinate
 * @param K the camera's inner parameter matrix
 * @param rot21 the rotation from first camera to second camera
 * @param t21 the translation from first camera to second camera
 * @param P1 the point on the first camera's coordinate
 * @param P2 the point on the second camera's coordinate
 * @return std::pair<float, float> the depth pair
 */
static std::pair<float, float> triangulation(
    const cv::KeyPoint &p1,
    const cv::KeyPoint &p2,
    const Eigen::Matrix3f &K,
    const Eigen::Matrix3f &rot21,
    const Eigen::Vector3f &t21,
    Eigen::Vector3f *P1 = nullptr,
    Eigen::Vector3f *P2 = nullptr) {
  Eigen::Vector3f X1 = K.inverse() * Eigen::Vector3f(p1.pt.x, p1.pt.y, 1.0f);
  Eigen::Vector3f X2 = K.inverse() * Eigen::Vector3f(p2.pt.x, p2.pt.y, 1.0f);
  return triangulation(X1, X2, rot21, t21, P1, P2);
}
} // 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[7]：

Listing 7: 解算结果

```
point's depth which used to check soluation
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

[7]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)。