SLAM Trick 1: Epipolar Constraints

陈烁龙

2022 年 4 月 29 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

假设我们要利用基础矩阵 F 来恢复运动，我们可以先基于最小二乘法求解得到矩阵 F，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 基础矩阵的求解

基础矩阵 F 的求解是基于上文的对极几何关系式 7 中的第二式来进行的。对于一对已配对的特征点对 $p_1(u_1, v_1)$ 和 $p_2(u_2, v_2)$，我们很容

[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：$v^{\wedge} v = v \times v = 0$

易可以写出如下的关系式：

$$\begin{pmatrix} u_2 & v_2 & 1 \end{pmatrix} \begin{pmatrix} f_1 & f_2 & f_3 \\ f_4 & f_5 & f_6 \\ f_7 & f_8 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ v_1 \\ 1 \end{pmatrix} = 0 \tag{8}$$

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 F 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 8 我们可以得到：

$$\begin{aligned} &u_1u_2f_1 + u_1v_2f_4 + u_1f_7 + v_1u_2f_2 + v_1v_2f_5 \\ &+ v_1f_8 + u_2f_3 + v_2f_6 + 1 = 0 \end{aligned} \tag{9}$$

即：

$$\begin{pmatrix} u_1u_2 \\ v_1u_2 \\ u_2 \\ u_1v_2 \\ v_1v_2 \\ v_2 \\ u_1 \\ v_1 \end{pmatrix}^T \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \end{pmatrix} = -1 \tag{10}$$

$$\rightarrow AX = l$$

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基础矩阵 F，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 F。

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，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，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：

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

如果我们以 0.95 作为显著性水平，则对应的分位点为 3.8415。换句话说，当：

$$\begin{cases} m_i \in goodMatches, Y_i < 3.8415 \\ m_i \notin goodMatches, Y_i \geq 3.8415 \end{cases}$$

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。

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

[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。

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

```
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
  * @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,
     std::vector<cv::DMatch> *goodMatches = nullptr,
     float quantile = 3.8415) {

   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::MatrixXd matA(matches.size(), 8), vecl = -Eigen::VectorXd::Ones(matches.size());
   Eigen::Vector<double, 8> matX;

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

   // construct the A matrix and l vector
   for (int i = 0; i != matches.size(); ++i) {
     const auto &match = matches.at(i);

     float u1 = kps1.at(match.queryIdx).pt.x, v1 = kps1.at(match.queryIdx).pt.y;
     float u2 = kps2.at(match.trainIdx).pt.x, v2 = kps2.at(match.trainIdx).pt.y;

     matA(i, 0) = u1 * u2, matA(i, 1) = v1 * u2;
     matA(i, 2) = u2, matA(i, 3) = u1 * v2;
     matA(i, 4) = v1 * v2, matA(i, 5) = v2;
     matA(i, 6) = u1, matA(i, 7) = v1;
   }

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

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

     // get parameters
     double f1 = matX(0, 0), f2 = matX(1, 0), f3 = matX(2, 0), f4 = matX(3, 0);
     double f5 = matX(4, 0), f6 = matX(5, 0), f7 = matX(6, 0), f8 = matX(7, 0);

     // find the badest outliers
     double 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);
       float u1 = kps1.at(match.queryIdx).pt.x, v1 = kps1.at(match.queryIdx).pt.y;
       float u2 = kps2.at(match.trainIdx).pt.x, v2 = kps2.at(match.trainIdx).pt.y;

       double a = u2 * f1 + v2 * f4 + f7;
       double b = u2 * f2 + v2 * f5 + f8;
       double c = u2 * f3 + v2 * f6 + 1.0;

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

       if (den == 0.0) {
         continue;
       }

       double statistics = num * num / den;

```

```
95        if (statistics < quantile) {
96          // current match is a good match
97          continue;
98        }
99
100       if (maxVar < statistics) {
101         maxVar = statistics;
102         maxIdx = i;
103       }
104     }
105
106     if (maxIdx == -1) {
107       // which means no outliers in current left matches
108       break;
109     } else {
110       // remove the outlier's affect
111       matA.row(maxIdx).setZero();
112       outliers.insert(maxIdx);
113     }
114   }
115
116   // organize F matrix
117   Eigen::Matrix3f matF;
118
119   matF(0, 0) = matX(0, 0), matF(0, 1) = matX(1, 0), matF(0,
          2) = matX(2, 0);
120   matF(1, 0) = matX(3, 0), matF(1, 1) = matX(4, 0), matF(1,
          2) = matX(5, 0);
121   matF(2, 0) = matX(6, 0), matF(2, 1) = matX(7, 0), matF(2,
          2) = 1.0f;
122
123   if (goodMatches != nullptr) {
124     goodMatches->clear();
125     goodMatches->resize(matches.size() - outliers.size());
126     int count = 0;
127     for (int i = 0; i != matches.size(); ++i) {
128       if (outliers.count(i) == 0) {
129         // it's not a outliers
130         goodMatches->at(count++) = (matches.at(i));
131       }
132     }
133   }
134
135   return matF;
136 }
137 } // 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：

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。

   列表 2 为估计的结果统计信息。具体的分析

(a) 原始图片一　　(b) 原始图片二

图 1: 原始图片

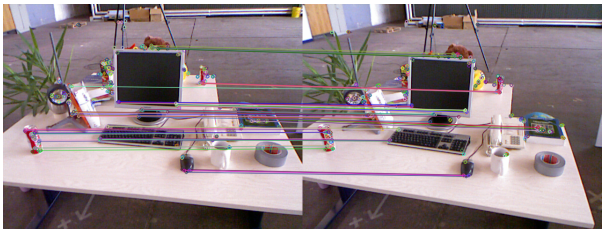

图 2: OpenCV 的匹配关系

留在下文。

Listing 2: OpenCV 结果

```
1 {'cost time': 0.19835(MS)}
2 matched points: 75
3 fun matrix:
4 [4.349427919371183e-06, 0.0001396240865909535,
      -0.01744943987902765;
5  -0.0001330018064455625, 2.353737144767806e-05,
       -0.01416915853179746;
6  0.01774555499943927, 0.003838948850228644, 1]
7 mean: -0.0363587
8 errorMeanNorm: 0.0560156
9 sigma: 0.0626025
```

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。

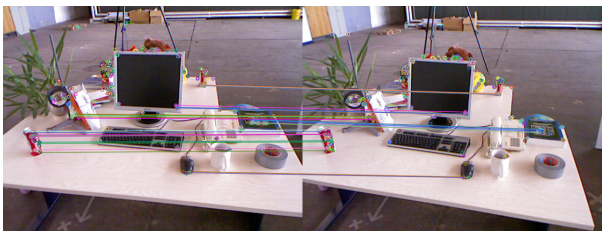

图 3: 假设检验的匹配关系

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': 0.35475(MS)}
matched points: 38
fun matrix:
 2.89776e−06 4.42187e−05 −0.0130419
−4.27177e−05 1.34145e−05 0.00583473
   0.0114956 −0.0128942 1
mean: 4.65536e−05
errorMeanNorm: 0.00583964
sigma: 0.00689752
```

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* 耗时比假设检验少。*OpenCV* 使用了 75 个匹配关系进行估计，统计检验使用了 38 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 较大，二假设检验基本吻合。且 *OpenCV* 的绝对误差均值和标准差都是假设检验的 10 倍左右。可见，在精确度方面，假设检验比较鲁棒，虽然在效率上不及 *OpenCV*[3]。

2.2 恢复运动

在求解得到基础矩阵 F 之后，我们可以先基于内参矩阵，将其转化为本质矩阵 E:

$$\begin{cases} E = t_{21}^{\wedge} R_{21} \\ F = K^{-T} t_{21}^{\wedge} R_{21} K^{-1} \end{cases}$$

$$\rightarrow E = K^T E K$$

而后对本质矩阵 E 进行 SVD 分解:

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

进而有四种解:

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

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

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

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

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 $\Sigma = diag(\frac{\sigma_1+\sigma_2}{2}, \frac{\sigma_1+\sigma_2}{2}, 0)$。

[3]我们认为这不会产生任何大的影响。