# 相机标定(三)——手眼标定

white_Learner


[相机标定(一)——内参标定与程序实现](#)

[相机标定(二)——图像坐标与世界坐标转换](#)

相机标定(三)——手眼标定

## 一、简述

手眼标定目的在于实现物体在世界坐标系和机器人坐标系中的变换。

在标定时，一般在工作平面设置一个世界坐标系，该坐标系与机器人坐标系不重合，在完成相机的内外参标定后，可计算获得物体在世界坐标系中的位置。若需要机器人与视觉联动，需要获得物体在在机器人坐标系中的坐标。

## 二、实现步骤

通过张正友法标定相机的内参矩阵和畸变参数；(相机标定(一)——内参标定与程序实现)
标定相机外参矩阵，用于图像坐标与世界坐标的转换；(相机标定(二)——图像坐标与世界坐标转换)
设置N个特征点(N>3)，计算其世界坐标，移动机械臂工作末端到特征点，记录末端坐标，获得N组数据；
计算两组数据的R和t，其中特征点世界坐标为A组数据，末端坐标为B组数据；
备注：

计算两组数据的变换矩阵实际上为3D-3D的位姿估计问题，可用迭代最近点（Iterative Closest Point, ICP）求解，实现方法有两种：

利用线性代数的求解（主要是 SVD）（建议采用该方法）
利用非线性优化方式的求解（类似于 Bundle Adjustment）
更多ICP相关算法可参考：[Calibration and Registration Techniques for Robotics](#)的Registering Two Sets of 3DoF Data

## 三、SVD求解

### 3.1 原理

参考：[计算两个对应点集之间的旋转矩阵R和转移矩阵T]

假设有两个点集AA AA和BB BB，且这两个点集合的元素数目相同且一一对应。为了寻找这两个点集之间的旋转矩阵RR RR和平移矩阵tt tt。可以将这个问题建模成如下的公式：

B=R∗A+t

**求解步骤**

计算点集合的中心点
将点集合移动到原点，计算最优旋转矩阵RR RR
计算转移矩阵tt tt

**求解**

**1.旋转矩阵R**

计算中心点

$$P_A = \begin{bmatrix} x_A \\ y_A \\ z_A \end{bmatrix}, P_B = \begin{bmatrix} x_B \\ y_B \\ z_B \end{bmatrix}$$
$$\mu_A = \frac{1}{N}\sum_{i=1}^{N} P_A^i, \mu_B = \frac{1}{N}\sum_{i=1}^{N} P_B^i$$

注意：$P_A^i$，$P_B^i$，$\mu_A$和$\mu_B$为向量

点集重新中心化

$$A_i' = \{P_A^i - \mu_A\}$$
$$B_i' = \{P_B^i - \mu_B\}$$

计算点集之间的协方差矩阵H

$$H=\sum_{i=1}^{N}A_i'B_i'^{T}=\sum_{i=1}^{N}(P_A^i-\mu_A)(P_B^i-\mu_B)^T$$

通过奇异值分解计算最优旋转矩阵

$$[U,S,V]=SVD(H)$$
$$R=VU^T$$

**2.平移矩阵t**

t=−R×μA+μB

**3.2 补充知识**

**1.协方差**

协方差（Covariance）是一种用来度量两个随机变量关系的统计量，定义为：

$$cov(A,B)=\frac{1}{N-1}\sum_{i=1}^{N}(A_i-\mu_A)*(B_i-\mu_B)$$

其中μA，μB分别为A，B的均值

**2.奇异值分解(SVD,Singular Value Decomposition)**

奇异值分解是一个能适用于任意的矩阵的一种分解的方法，公式为：

A=UΣVT

几何含义：对于任意一个矩阵，找到一组两两正交单位向量序列，使得矩阵作用在此向量序列上后得到新的向量序列保持两两正交。

### 3.3 程序实现

```
bool RtbySVDSrv( vector<Eigen::Vector3d> worldPoints, vector<Eigen::Vector3d> robotPoints,Eigen::Vector3d &t,Eigen::Matrix3d &R, Eigen::Quaterniond &q) {

 // check data
 if (worldPoints.size() != robotPoints.size() || worldPoints.size() < 3)
         return false;

 // save data
 int size = worldPoints.size();

 // count centre points
 Eigen::Vector3d worldCentre, robotCentre;
 for (int i = 0; i < size; i++) {
   worldCentre += worldPoints[i];
   robotCentre += robotPoints[i];
 }
 worldCentre /= size;
 robotCentre /= size;

 // count the vector
 vector<Eigen::Vector3d> worldVectors(size), robotVectors(size);
 for (int i = 0; i < size; i++) {
   worldVectors[i] = worldPoints[i] - worldCentre;
   robotVectors[i] = robotPoints[i] - robotCentre;
 }

 // count H
 Eigen::Matrix3d H;
 for (int i = 0; i < size; i++) {
   H += worldVectors[i] * robotVectors[i].transpose();
 }

 // svd count R and Q
 Eigen::JacobiSVD<Eigen::MatrixXd> svd(H, Eigen::ComputeThinU |
                                             Eigen::ComputeThinV);
 Eigen::Matrix3d V = svd.matrixV(), U = svd.matrixU();
 R = V * U.transpose();

 if (R.determinant() < 0)
   R *= -1;
 q = Eigen::Quaterniond(R);
 q.normalize();

 // count t
 t = robotCentre - R * worldCentre;

 return true;
}
```

## 四、非线性优化求解

非线性优化是以迭代的方式去找最优值（选取一组数据变换后与另外一组数据的差值为误差值），以李代数表达位姿时，目标函数可以写成：

$$\min_{\xi} = \frac{1}{2}\sum_{i=1}^{n}\left\|p_i - exp(\xi^{\Lambda})p_i{}'\right\|_2^2$$

ICP问题存在唯一解或无穷多解的情况。在唯一解的情况下，只要我们能找到极小值解，那么这个极小值就是全局最优值（因此不会遇到局部极小而非全局最小的情况）。这意味着ICP求解可以任意选定初始值。

**注意：**

ICP更常用于匹配未知的情况下，此处计算已知匹配点对存在解析解，可分别采用SVD或非线性优化计算，但没必要对SVD计算结果进行优化。

### 4.1 通过Ceres构建优化程序

Ceres可参考SLAM学习——Ceres

损失函数
使用四元数描述旋转并用于优化，优化维度为4个旋转，3个平移，误差维度为3。

**注意：**

欧拉角在描述旋转时存在万向锁的问题，若使用欧拉角描述旋转和用于优化，会在转换到变换矩阵进行旋转变换时产生错误。

```
struct ICP_COST {
  ICP_COST(Point3f p1, Point3f p2) : _p1(p1), _p2(p2) {}
  template <typename T>
  bool operator()(const T *const q, const T *const t, T *residual) const {
    // P2->P1
    T p_21[3];
    p_21[0] = T(_p2.x);
    p_21[1] = T(_p2.y);
    p_21[2] = T(_p2.z);
    ceres::QuaternionRotatePoint(q, p_21, p_21);
    p_21[0] += t[0];
    p_21[1] += t[1];
    p_21[2] += t[2];

    // 误差
    residual[0] = T(_p1.x) - p_21[0];
    residual[1] = T(_p1.y) - p_21[1];
    residual[2] = T(_p1.z) - p_21[2];
    return true;
  }

  const Point3f _p1, _p2;
};

```

## BA优化

```cpp
void bundleAdjustment(const vector<Point3f> &pts1, const vector<Point3f> &pts2,
                      Eigen::Quaterniond ceres_q, Eigen::Vector3d ceres_t) {
  // 优化初始值，可以使用SVD获得值进行优化，或者直接使用0
  double q[4] = {ceres_q.w(), ceres_q.x(), ceres_q.y(), ceres_q.z()};
  double t[3] = {ceres_t[0], ceres_t[1], ceres_t[2]};

  // 构造问题
  ceres::Problem problem;
  int len = pts1.size();
  for (int i = 0; i < len; i++) {
    problem.AddResidualBlock(new ceres::AutoDiffCostFunction<ICP_COST, 3, 4, 3>(
                                 new ICP_COST(pts1[i], pts2[i])),
                             nullptr, q, t);
  }

  // 配置问题并求解
  ceres::Solver::Options options;
  options.linear_solver_type = ceres::DENSE_QR; // 配置增量方程的解法
  options.minimizer_progress_to_stdout = true;  // 输出到cout
  ceres::Solver::Summary summary;               // 优化信息
  ceres::Solve(options, &problem, &summary);    // 开始优化

  // 输出结果
  cout << summary.BriefReport() << endl;
  Eigen::Matrix3d R_Martix = Quaternion2RotationMatrix(q[1], q[2], q[3], q[0]);
  Eigen::Vector3d t_Martix = {t[0], t[1], t[2]};
  cout << "R" << R_Martix << endl;
  cout << "t" << t_Martix << endl;

  // verify
  for (int i = 0; i < 5; i++) {
    cout << "p1 = " << pts1[i] << endl;
    cout << "p2 = " << pts2[i] << endl;
    Eigen::Vector3d point = {pts2[i].x, pts2[i].y, pts2[i].z};
    cout << "p1_count = " << R_Martix * point + t_Martix  << endl;
  }
}
```

## 参考

[计算两个对应点集之间的旋转矩阵R和转移矩阵T]

[Finding optimal rotation and translation between corresponding 3D points]

[奇异值的物理意义是什么？]

《视觉SLAM十四讲》——第七讲 视觉里程计

机器视觉 标定

上一篇：相机标定(二)——图像坐标与世界坐标转换

下一篇：ROS进阶——kinect v1的使用