

ICP

陈烁龙

2022 年 6 月 8 日

# 目录

<b>1</b>	<b>ICP 算法原理</b>	<b>1</b>
1.1	二维下的雅可比矩阵 . . . . .	1
1.2	三维下的雅可比矩阵 . . . . .	2
1.3	高斯牛顿法 . . . . .	2
<b>2</b>	<b>实验</b>	<b>2</b>
2.1	基于已知匹配点云的 ICP . . . . .	2
2.2	基于未知匹配点云的 ICP . . . . .	3

# 插图

1	初始点云 . . . . .	2
2	迭代求解过程 . . . . .	3
3	初始点云 . . . . .	3
4	迭代求解过程 . . . . .	4

# 表格

# 摘要

ICP 算法是求解两帧点之间的位姿变换关系的一种经典方法，其对点间有匹配和点间无匹配的点云帧都有着比较鲁棒的估计结果。

**关键词：** ICP，点云，位姿变换，李代数

## 1 ICP 算法原理

如摘要所提，ICP 是一种经典的点云帧间位姿求解算法，中文名为“迭代最近点”。其通过迭代的方式，不断调整位姿，使得转换后的点云位姿间的距离最小。其对于有匹配关系的点云帧或者未知匹配关系的点云帧，都有比较鲁棒的估计。

由于已有匹配关系的点云帧间位姿变换求解比较基础，故由该种场景出发，阐述算法原理。假设现有两帧点云  $PC_1 = \{p_1^1, p_1^2, \dots, p_1^n\}$  和  $PC_2 = \{p_2^1, p_2^2, \dots, p_2^n\}$ ，其中  $PC_1$  中的  $p_1^i$  与  $PC_2$  中的  $p_2^i$  存在对应关系。两帧之间的位姿变换用  $T_{21}[R_{21}|t_{21}]$  表示，表示的是点从第 1 帧到第 2 帧的变换关系。我们的目标是最下化下面的损失函数：

$$\min e = \sum_{i=1}^n e^i = \sum_{i=1}^n (T_{21}p_1^i - p_2^i) \quad (1)$$

为使用高斯-牛顿方法优化误差函数，我们需要对待优化变量进行雅可比矩阵的求解。具体来说，我们需要求解得到：

$$J^i = \frac{\partial e^i}{\partial \xi_{21}} = \frac{\partial (T_{21}p_1^i)}{\partial \xi_{21}}$$

其中， $\xi_{21}$  为  $T_{21}$  的李代数表达形式，二者本质上是是一致的。

### 1.1 二维下的雅可比矩阵

对于二维情况， $T_{21}$  可以显示的表达成下矩阵形式（齐次坐标下的变换矩阵）：

$$T_{21} = \begin{pmatrix} R_{21} & t_{21} \\ 0^T & 1 \end{pmatrix} = \begin{pmatrix} \cos \phi & -\sin \phi & t_x \\ \sin \phi & \cos \phi & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

则基于变换  $T_{21}$ ，将点  $p_1^i = (x_1^i, y_1^i, 1)^T$  变换到点  $p_1^{i'} = (x_1^{i'}, y_1^{i'}, 1)^T$ ，可表示为：

$$p_1^{i'} = T_{21}p_1^i = \begin{pmatrix} x_1^{i'} \\ y_1^{i'} \\ 1 \end{pmatrix} = \begin{pmatrix} x_1^i \cos \phi - y_1^i \sin \phi + t_x \\ x_1^i \sin \phi + y_1^i \cos \phi + t_y \\ 1 \end{pmatrix}$$

至此，我们基于李代数左扰动模型进行求导：

$$\begin{aligned} \frac{\partial (T_{21}p_1^i)}{\partial \delta \xi_{21}} &= \frac{\begin{pmatrix} \cos \delta \phi & -\sin \delta \phi & \delta t_x \\ \sin \delta \phi & \cos \delta \phi & \delta t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1^{i'} \\ y_1^{i'} \\ 1 \end{pmatrix}}{\begin{pmatrix} \delta t_x & \delta t_y & \delta \phi \end{pmatrix}^T} \\ &= \frac{\begin{pmatrix} x_1^{i'} \cos \delta \phi - y_1^{i'} \sin \delta \phi + \delta t_x \\ x_1^{i'} \sin \delta \phi + y_1^{i'} \cos \delta \phi + \delta t_y \\ 1 \end{pmatrix}}{\begin{pmatrix} \delta t_x & \delta t_y & \delta \phi \end{pmatrix}^T} \\ &= \begin{pmatrix} 1 & 0 & -x_1^{i'} \sin \delta \phi - y_1^{i'} \cos \delta \phi \\ 0 & 1 & x_1^{i'} \cos \delta \phi - y_1^{i'} \sin \delta \phi \\ 0 & 0 & 0 \end{pmatrix} \\ &\approx \begin{pmatrix} 1 & 0 & -y_1^{i'} \\ 0 & 1 & x_1^{i'} \\ 0 & 0 & 0 \end{pmatrix} \end{aligned}$$

最后一步的约等于是考虑到扰动量是小量。所以，我们有：

$$J_{2d}^i = \frac{\partial e^i}{\partial \delta \xi_{21}} = \begin{pmatrix} 1 & 0 & -y_1^{i'} \\ 0 & 1 & x_1^{i'} \\ 0 & 0 & 0 \end{pmatrix} \quad (2)$$

## 1.2 三维下的雅可比矩阵

易得：

$$J_{3d}^i = \frac{\partial e^i}{\partial \delta \xi_{21}} = \begin{pmatrix} I_{3 \times 3} & -(T_{21} p_1^i)^\wedge \\ 0_{1 \times 3}^T & 0_{1 \times 3}^T \end{pmatrix}_{4 \times 6}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 & z_1^{i'} & -y_1^{i'} \\ 0 & 1 & 0 & -z_1^{i'} & 0 & x_1^{i'} \\ 0 & 0 & 1 & y_1^{i'} & -x_1^{i'} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (3)$$

该求导过程使用了李代数的左扰动模型。

观察二维和三维下的雅可比矩阵，除了维度不同之外，我们可以看到一些相似之处。如果我们把二维位姿变换移至三维空间进行观察，我们可以发现，二维位姿变换是一种特殊的三维位姿变化，其只绕  $Z$  轴进行旋转，且不沿  $Z$  轴方向平移。

所以，如果我们把  $J_{3d}^i$  的第 3 行和第 3、4、5 列移除<sup>1</sup>，那么剩下的矩阵便是二维位姿变换雅可比矩阵。

## 1.3 高斯牛顿法

对每一对匹配点对计算该雅可比矩阵和误差函数值，而后带入高斯-牛顿迭代方程进行求解。

$$\begin{cases} H^i = J^{iT} J^i \\ H_{6 \times 1} = \sum_{i=1}^n H^i \\ g^i = -J^{iT} e^i \\ g_{6 \times 1} = \sum_{i=1}^n g^i \end{cases} \quad (4)$$

$$H \Delta X = g \quad (5)$$

求解上述方程以更新带求参数。

<sup>1</sup>移除第 3 行第三列表示只绕  $Z$  轴进行旋转，移除第 4、5 列表示不沿  $Z$  轴方向平移，因为这两列包含  $Z$  坐标。

## 2 实验

### 2.1 基于已知匹配点云的 ICP

通过随机生成点的方式，生成了如下图所示的初始点云。图中灰色虚线的连接关系表示了点云之间已知的变换关系。具体来说，在随即生成的点云  $PC_1$  的基础上，设定位姿变换为  $T_{21}[R_{21}|t] = (\pi/4, 2, 2)$ 。同时对转换的结果点云增加小误差。

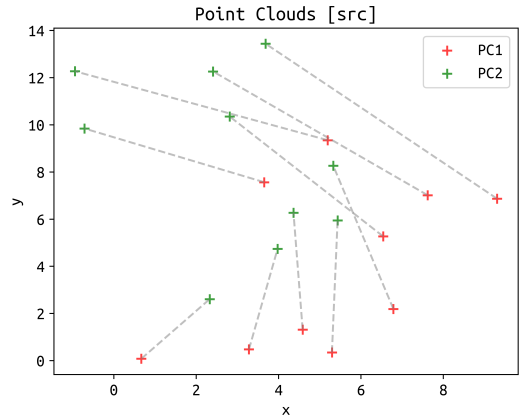


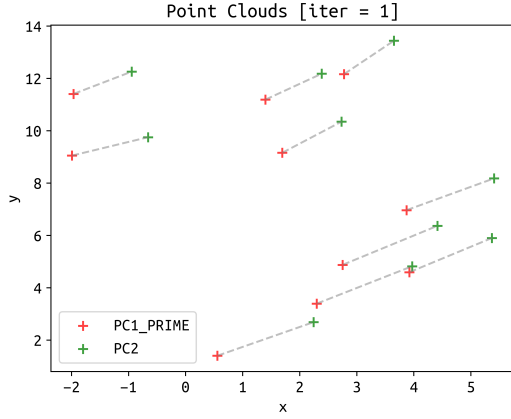
图 1: 初始点云

在代码中，我们进行一定次数的迭代。在每一次迭代中，我们基于当前估计的位姿变换，对所有点对计算误差和雅可比矩阵，而后解线性方程求解增量，并对位姿进行更新。

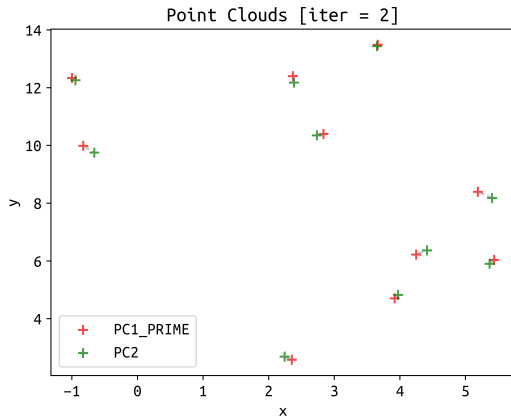
Listing 1: 核心代码

```
1 Sophus::SE2f T21;
2 for (int i = 0; i != iter; ++i) {
3     Eigen::Matrix3f H = Eigen::Matrix3f::Zero();
4     Eigen::Vector3f g(0.0f, 0.0f, 0.0f);
5     for (int i = 0; i != pc1.size(); ++i) {
6         auto &p1 = pc1.at(i);
7         auto &p2 = pc2.at(i);
8         auto p1_prime = T21 * p1;
9         auto error = p1_prime - p2;
10        Eigen::Matrix<float, 2, 3> J;
11        J(0, 0) = 1, J(0, 1) = 0, J(0, 2) = -p1_prime(1);
12        J(1, 0) = 0, J(1, 1) = 1, J(1, 2) = p1_prime(0);
13        H += J.transpose() * J;
14        g -= J.transpose() * error;
15    }
16    auto delta = H.ldlt().solve(g);
17    T21 = Sophus::SE2f::exp(delta) * T21;
```

下图展示了每一次迭代的结果。可以看到，当迭代两次之后，位姿估计已经比较精确。



(a) 迭代 1 次



(b) 迭代 2 次

图 2: 迭代求解过程

## 2.2 基于未知匹配点云的 ICP

对于未知配对关系的点云，为计算误差，我们的策略是将最近的点对作为临时的配对点对。具体来说，对于每一次计算误差，我们先基于当前的位姿估计，将第 1 帧点云中的点进行位姿变换。而后，基于每一个变换后的点，在第 2 帧点云里搜寻最近的一个点作为其匹配点。

当然，直接进行迭代是行不通的，因为极有可能落入局部极小值。为此，我们考虑到两帧点

云的结构是相似的，意味着两帧点云在进行点的归一化之后，之间只差了一个旋转，由此我们基于最近点对构建匹配的假设才能行得通<sup>2</sup>。这样，当我们求解得到旋转量之后，再计算位移量。

我们首先对两帧点云求解重心，而后对点云里的每一个点进行归一化：

$$\begin{cases} Center(PC_j) = c_j = \frac{1}{n} \sum_{i=1}^n p_j^i \\ q_j^i = p_j^i - c_j \end{cases}$$

而后基于新的去重心后的点云帧，求解旋转量  $R_{21}$ 。求解得到解旋转量  $R_{21}$  之后，再进行位移量  $t_{21}$  的求解：

$$t_{21} = p_2^i - R_{21}p_1^i$$

一般使用最小二乘进行求解。

下图为初始点云，其和“实验一”的点云是一致的，不过其点之间的配对关系是未知的。

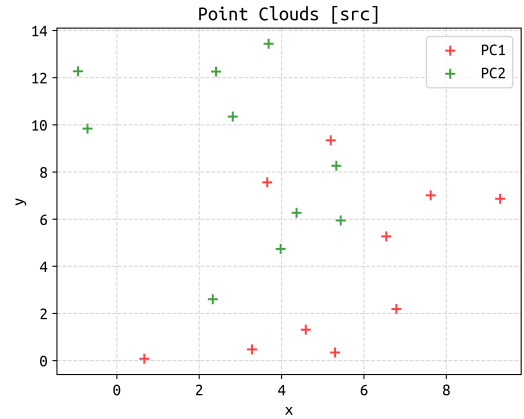


图 3: 初始点云

Listing 2: 核心代码

```
1 auto count = pc1.size();
2 auto pc1_new = normalize(pc1), pc2_new = normalize(pc2);
3 Sophus::SO2f R21;
4 for (int i = 0; i != iter; ++i) {
5     float H = 0.0f;
6     float g = 0.0f;
```

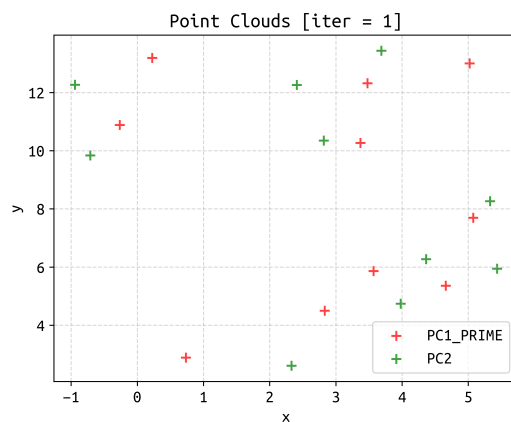
<sup>2</sup>事实上，基于解析方式推导的 ICP 算法可以证明该种措施的合理性。

```

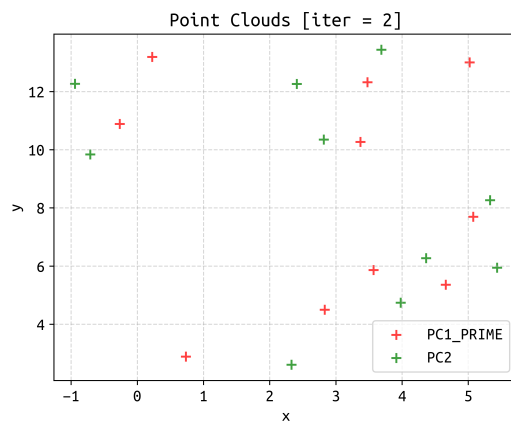
7   for (int i = 0; i != count; ++i) {
8       auto &p1 = pc1_new.at(i);
9       auto p1_prime = R21 * p1;
10      // find min dis point
11      auto min_iter = std::min_element(pc2_new.cbegin(), pc2_new
        .cend(), [p1_prime](const Eigen::Vector2f &p2_i,
        const Eigen::Vector2f &p2_j) {
12          auto p1_prime_t = ns_geo::Point2f(p1_prime(0), p1_prime(1)
            );
13          auto p2_i_t = ns_geo::Point2f(p2_i(0), p2_i(1));
14          auto p2_j_t = ns_geo::Point2f(p2_j(0), p2_j(1));
15          return ns_geo::distance(p1_prime_t, p2_i_t) < ns_geo::
            distance(p1_prime_t, p2_j_t);
16      });
17      auto &p2 = *min_iter;
18      auto error = p1_prime - p2;
19      Eigen::Vector2f J;
20      J(0) = -p1_prime(1);
21      J(1) = p1_prime(0);
22      H += J.transpose() * J;
23      g -= J.transpose() * error;
24  }
25  auto delta = g / H;
26  R21 = Sophus::SO2f::exp(delta) * R21;
27  }
28  Eigen::MatrixXf A(2 * count, 2);
29  Eigen::VectorXf l(2 * count);
30  for (int i = 0; i != count; ++i) {
31      auto trans = pc2.at(i) - R21 * pc1.at(i);
32      A(2 * i + 0, 0) = 1, A(2 * i + 0, 1) = 0;
33      A(2 * i + 1, 0) = 0, A(2 * i + 1, 1) = 1;
34      l(2 * i + 0) = trans(0);
35      l(2 * i + 1) = trans(1);
36  }
37  auto t21 = (A.transpose() * A).inverse() * A.transpose() * l;
38  Sophus::SE2f T21(R21.matrix(), Eigen::Vector2f(t21(0), t21(1))
    );

```

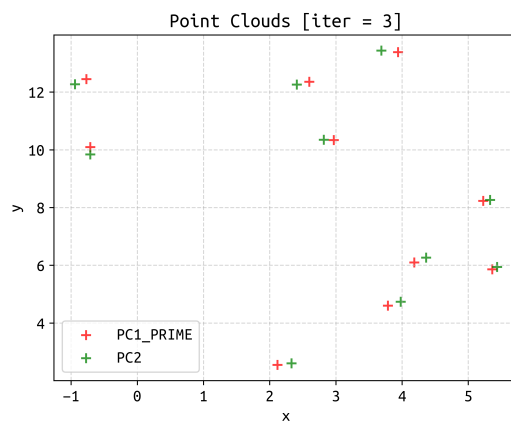
由下图可以看到，基于未知匹配点云的 ICP 在迭代了 4 次之后，才能接近最优解。其相较于基于已知匹配点云的 ICP，计算效率更慢，迭代次数也更多。就是因为一个原因：点之间的匹配关系是未知的。



(a) 迭代 1 次



(b) 迭代 2 次



(c) 迭代 3 次

