9.1 配准

配准算法及相关概念介绍(书11.1)\如何使用icp(书11.2.1，11.2.4)\如何逐步匹配多幅点云（书11.2.2）

1 理论基础

The PCL Registration API官网理论

02-点云配准原理概述

点云配准概念

由于三维扫描仪设备受到测量方式和被测物体形状的条件限制，一次扫描往往只能获取到局部的点云信息，进而需要进行多次扫描，然后每次扫描时得到的点云都有独立的坐标系，不可以直接进行拼接。

在逆向工程、计算机视觉、文物数字化等领域中，由于点云的不完整、旋转错位、平移错位等，使得要得到完整点云就需要对多个局部点云进行配准。

为了得到被测物体的完整数据模型,需要确定一个合适的坐标变换， **将从各个视角得到的点集合并到一个统一的坐标系下形成一个完整的数据点云，然后就可以方便地进行可视化等操作，这就是[点云数据的配准](http://pointclouds.org/documentation/tutorials/registration_api.php#registration-api)。**

点云配准方法

点云配准步骤上可以分为**粗配准（Coarse Registration）和精配准（Fine Registration）**两个阶段。

粗配准

粗配准是指在点云相对位姿完全未知的情况下对点云进行配准，找到一个可以让两块点云相对近似的旋转平移变换矩阵，进而将待配准点云数据转换到统一的坐标系内，可以为精配准提供良好的初始值。常见粗配准算法：

- **基于特征匹配(PFH)的配准算法：**
    - SAC-IA 采样一致性初始配准算法（Sample Consensus Initial Alignment）PCL库已实现，基于FPFH
- **基于穷举搜索的配准算法：**
    - 4PCS 四点一致集配准算法（4-Point Congruent Set）
    - Super4PCS

精配准

精配准是指在粗配准的基础上，让点云之间的空间位置差异最小化，得到一个更加精准的旋转平移变换矩阵。该算法的运行速度以及向全局最优化的收敛性却在很大程度上依赖于给定的**初始变换估计**以及在迭代过程中**对应关系的确立。**所以需要各种粗配准技术为ICP算法提供较好的位置，在迭代过程中确立正确对应点集能避免迭代陷入局部极值，决定了算法的收敛速度和最终的配准精度。最常见的精配准算法是ICP及其变种。

- **ICP 迭代最近点算法（Iterative Cloest Point）**

    - GICP
    - NICP
    - MBICP
- **NDT 正态分布变换算法（Normal Distributions Transform）**

其他配准：

- **依赖平台设备：**将被测物体放在平台上，利用控制器对平台进行控制，使之按照指定角度转动，通过多次测量可以得到不同视角下的点云，由于提前获知了距离及角度信息，则可以直接对所有点云进行配准。
- **辅助标志点：**通过在被测物体表面粘贴标签，**将这些标签作为标志点，对多次测量得到的点云数据进行配**准时，**对这些有显著特征的标签进行识别配准**，代替了对整体点云的配准，提高效率，精确度。

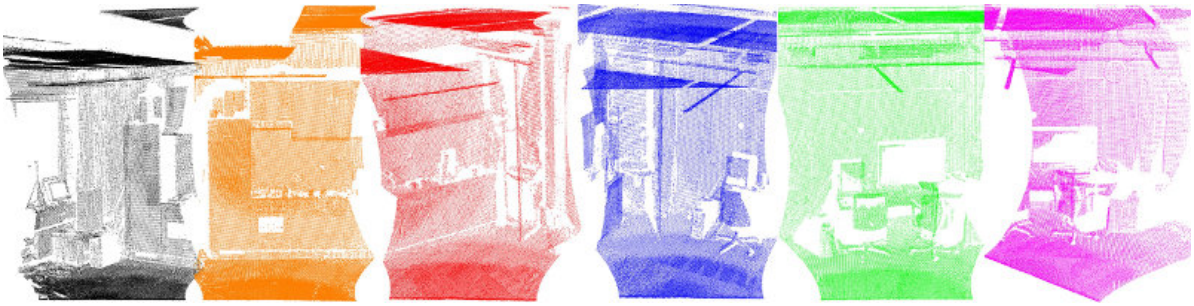
自动配准技术

通常所说的点云配准就是指自动配准，点云自动配准技术是通过一定的算法或者统计学规律，利用计算机计算两块点云之间的错位，从而达到把两片点云自动配准的效果。本质上就是把不同坐标系中测量得到的数据点云进行坐标变换，从而得到整体的数据模型。

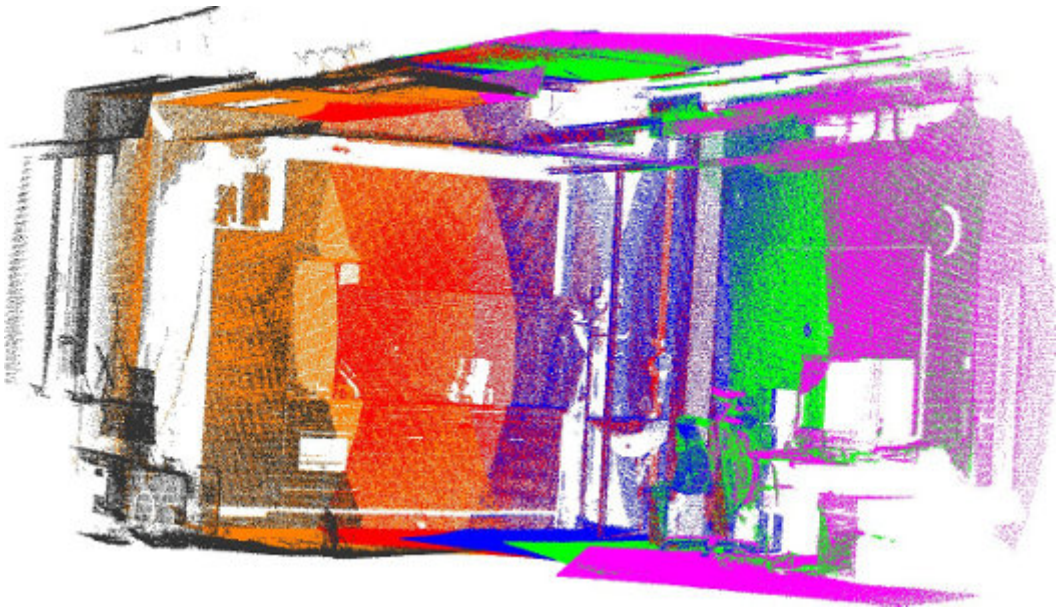
即**求得坐标变换参数 R（旋转矩阵）和 T（平移向量**）,使得两视角下测得的三维数据经坐标变换后**的距离最小。**

配准算法按照实现过程可以分为**整体配准和局部配准。**

PCL实现的配准算法示例



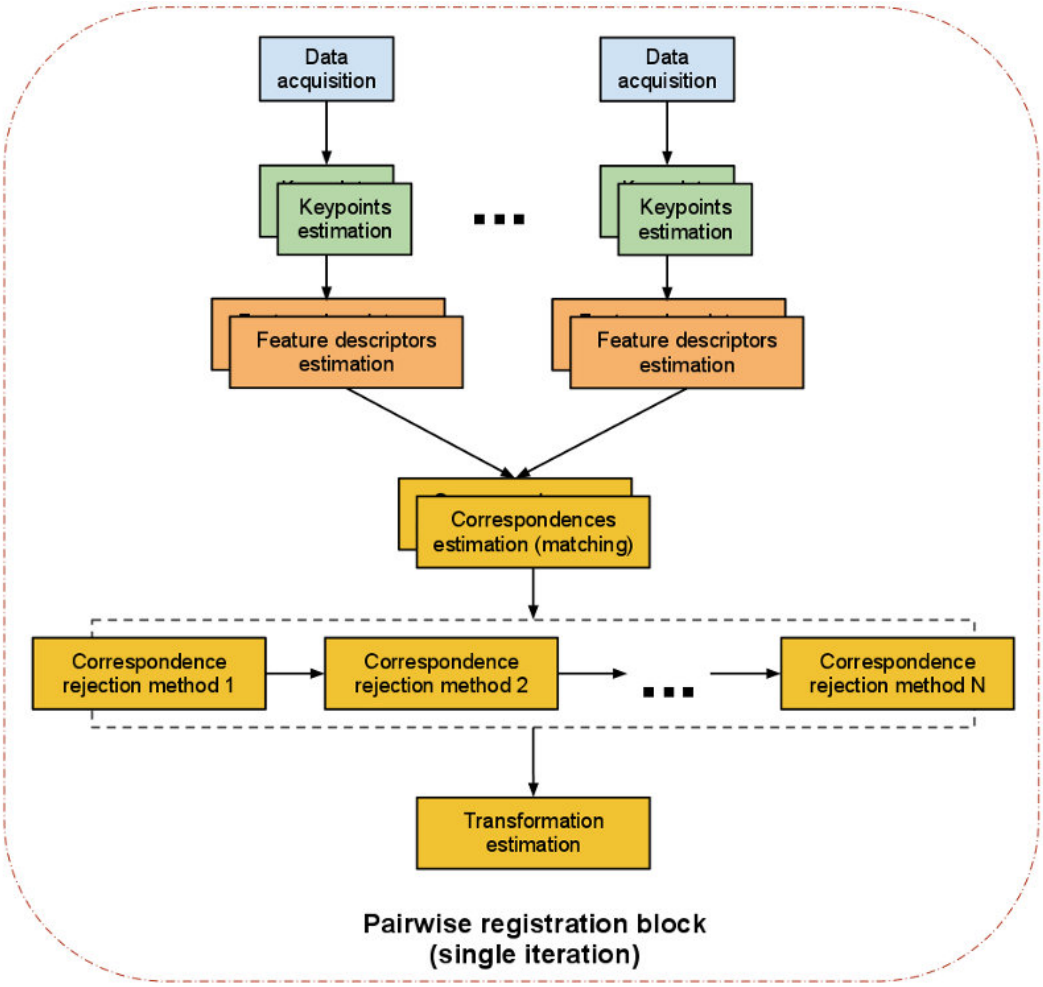
上图中给出了示例，使用倾斜的2D激光设备获取了六个独立数据集。由于每个单独的扫描仅代表周围世界的一小部分，因此必须找到将它们配准在一起的方法，从而创建完整的点云模型，如下图所示。



PCL提供的配准库算法是通过在给定的输入数据集中找到正确的点对应关系，并将每个单独的数据集转换为一致的全局坐标系的刚性变换。理想情况下，如果在输入数据集中完全知道点对应关系，则该配准范式可以轻松解决。这意味着一个数据集中选定的关键点列表必须与另一个数据集中的点列表“重合”。此外，如果估计的对应关系“完美”匹配，则配准问题具有封闭式解决方案。 PCL包含一组功能强大的算法，这些算法可以估算多组对应关系，排除不良对应关系，从而以可靠的方式估算转换关系方法。以下将分别描述它们：

两两配准步骤

两两配准（pairwise registration）：我们称一对点云数据集的配准问题为两两配准（pairwise registration）。通常通过应用一个**估算得到的表示平移和旋转的 4 × 4 刚体变换矩阵**来使一个点云数据集精确地与另一个点云数据集(目标数据集)进行完美配准。



具体实现步骤如下：

1. 首先从两个数据集中按照同样的关键点选取标准，提取关键点。
2. 对选择的所有关键点分别计算其特征描述子。
3. 结合特征描述子在两个数据集中的坐标的位置，以两者之间特征和位置的相似度为基础，估算它们的对应关系，初步估计对应点对。
4. 假定数据是有噪声的，除去对配准有影响的错误的对应点对。
5. 利用剩余的正确对应关系来估算刚体变换，完成配准。

整个配准过程最重要的是关键点的提取以及关键点的特征描述，以确保对应估计的准确性和效率，这样才能保证后续流程中的刚体变换矩阵估计的无误性。接下来我们**对单次迭代的每一步进行解读：**

1.关键点提取

关键点是在场景中具有“特殊属性”的兴趣点，例如书的一角或书上写有“ PCL”的字母“ P”。 **PCL中有许多不同的关键点提取技术，如NARF，SIFT和FAST。**另外，您也可以将每个点或子集作为关键点。如果不进行关键点提取，直接“将两个kinect数据集执行对应估计”会产生的问题是：每帧中有300k点，因此可以有300k^2个对应关系，这个数量太庞大不利于计算。

2.特征描述子

基于找到的关键点，我们必须提取特征，在此我们封装解析点云数据并生成向量以相互比较。同样，有许多特征描述符提取技术可供选择，例如NARF，FPFH，BRIEF或SIFT。

所谓局部特征描述子就是用来刻画图像中的这些局部共性的，而我们也可以将一幅图像映射（变换）为一个局部特征的集合。理想的局部特征应具有平移、缩放、旋转不变性，同时对光照变化、仿射及投影影响也应有很好的鲁棒性。传统的局部特征往往是直接提取角点或边缘，对环境的适应能力较差。1999年British Columbia大学 David G.Lowe 教授总结了现有的基于不变量技术的特征检测方法，并正式提出了一种基于尺度空间的、对图像缩放、旋转甚至仿射变换保持不变性的图像局部特征描述算子 - SIFT（尺度不变特征变换），这种算法在2004年被加以完善。

3. 对应关系估计

对应关系估计(correspondences estimation)

假设我们已经得到由两次扫描的点云数据获得的两组特征向量，在此基础上，我们必须找到相似特征再确定数据的重叠部分才能进行配准。根据特征的类型，PCL 使用不同方法来搜索特征之间的对应关系。

- 进行**点匹配point matching时**(使用点的 xyz 三维坐标作为特征值)，针对有序点云数据和无序点云数据有不同的处理策略：
    - 穷举配准(brute force matching）简称BFMatching，或称野蛮匹配。
    - kd-tree最近邻查询，Fast Library for Approximate Nearest Neighbors.(FLANN)。
    - 在有序点云数据的图像空间中查找。
    - 在无序点云 数据的索引空间中查找。
- 进行**特征匹配feature matching时**(不使用点的坐标，而是某些由查询点邻域确定的特征，如法向量、局部或全局形状直方图等)，有以下几种方法：
    1. 穷举配准(brute force matching）简称BFMatching，或称野蛮匹配。
    2. kd-tree最近邻查询，Fast Library for Approximate Nearest Neighbors.(FLANN)。
- 除了搜索之外，对**应估计也区分了两种类型：**
    1. **直接对应估计(默认)：**为点云 A 中的每一个点搜索点云 B 中的对应点,确认最终对应点对。
    2. **“相互”(Reciprocal)对应估计** : 首先为点云 A 中的点到点云 B 搜索对应点 ， 然后又从点云 B 到点云 A 搜索对应点，最后只取交集作为对应点对。

所有这些在 PCL 类设计和实现中都以**函数的形式**让用户可以自由设定和使用。

4. 对应关系去除

对应关系去除(correspondences rejection)

由于噪声的影响，通常并不是所有估计的对应关系都是正确的 。由于错误的对应关系对于最终的刚体变换矩阵的估算会产生负面的影响，所以必须去除它们，于是我们**使用随机采样一致性(Random Sample Consensus , RANSAC)估算或者其他方法剔除错误对应关系**，最终只保留一定比例的对应关系，这样即能提高变换矩阵的估算精度也可以提高配准速度。 遇到有一对多对应关系的特例情况，即目标模型中的一个点在源中有若干个点与之对应。可以通过只取与其距离最近的对应点或者检查附近的其他匹配的滤波方法过滤掉其他伪对应关系。同样地针对对应关系的去除，PCL 有单独设计类与之对应 。

5. 变换矩阵估算

最后一步就是计算实际的转换关系

1. 根据对应关系评估误差值
2. 估算相机位姿之间的刚性变换，并最小化误差指标
3. 优化点的结构
4. 使用刚性变换将源旋转/变换到目标上
5. 迭代此过程直到满足预定的收敛标准

2 代码实践

类&函数：https://pointclouds.org/documentation/group__registration.html

{ICP原理公式：https://zhuanlan.zhihu.com/p/107218828?utm_source=wechat_session}

1 使用迭代最近点算法(ICP) How to use iterative closest point

迭代最近点算法（Iterative Closest Point，简称ICP算法）

$$E(R,T)=\frac{1}{n}\sum_{i=1}^{n}\|q_i-(Rp_i+T)\|^2$$

class	pcl::IterativeClosestPoint< PointSource, PointTarget, Scalar >
	IterativeClosestPoint provides a base implementation of the Iterative Closest Point algorithm. More...提供了迭代最近点算法的基本实现

Usage example:

```
IterativeClosestPoint<PointXYZ, PointXYZ> icp;
// 设置输入源和目标
icp.setInputCloud (cloud_source);
icp.setInputTarget (cloud_target);

// 将最大相似距离设置为5cm
icp.setMaxCorrespondenceDistance (0.05);  // 距离较大的将被忽略
// 设置最大迭代次数（标准1）
icp.setMaximumIterations (50);
// 设置转换ϵ（标准2）？
icp.setTransformationEpsilon (1e-8);
// 设置欧几里德距离差ϵ（标准3）？
icp.setEuclideanFitnessEpsilon (1);

// 执行校准
icp.align (cloud_source_registered);

// 获取将点云源与配准的点云源对齐的转换
Eigen::Matrix4f transformation = icp.getFinalTransformation ();
```

代码实现

实例分析：如何使用迭代最近点算法：在代码中使用ICP迭代最近点算法，程序随机生成一个点与作为源点云，并将其沿x轴平移后作为目标点云，然后利用ICP估计源到目标的刚体变换橘子，中间对所有信息都打印出来

创建文件：`iterative_closest_point.cpp`

准备资源：`无`

编译执行：`./iterative_closest_point  ../bunny.pcd`

```
/*
使用迭代最近点算法(ICP)
 */
#include <iostream>                //标准输入输出头文件
#include <pcl/io/pcd_io.h>         //I/O操作头文件
#include <pcl/point_types.h>       //点类型定义头文件
#include <pcl/registration/icp.h> //ICP配准类相关头文件

int main(int argc, char **argv)
{
    // 定义输入和输出点云
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_in(new
pcl::PointCloud<pcl::PointXYZ>);
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_out(new
pcl::PointCloud<pcl::PointXYZ>);

    // 随机填充无序点云
    cloud_in->width = 5;  //设置点云宽度
    cloud_in->height = 1; //设置点云为无序点
    cloud_in->is_dense = false;
    cloud_in->points.resize(cloud_in->width * cloud_in->height);
    for (size_t i = 0; i < cloud_in->points.size(); ++i)
    {
        cloud_in->points[i].x = 1024 * rand() / (RAND_MAX + 1.0f);
        cloud_in->points[i].y = 1024 * rand() / (RAND_MAX + 1.0f);
        cloud_in->points[i].z = 1024 * rand() / (RAND_MAX + 1.0f);
    }
    std::cout << "Saved " << cloud_in->points.size() << " data points to input:"
//打印点云总数
              << std::endl;
    for (size_t i = 0; i < cloud_in->points.size(); ++i)            //打印坐标
        std::cout << "    " << cloud_in->points[i].x << " " << cloud_in-
>points[i].y << " " << cloud_in->points[i].z << std::endl;
    *cloud_out = *cloud_in;
    std::cout << "size:" << cloud_out->points.size() << std::endl;

    // //实现一个简单的点云刚体变换，以构造目标点云，将cloud_out中的x平移0.7f米，然后再次输
出数据值。
    for (size_t i = 0; i < cloud_in->points.size(); ++i)
        cloud_out->points[i].x = cloud_in->points[i].x + 0.7f;
    // 打印这些点
    std::cout << "Transformed " << cloud_in->points.size() << " data points:"
              << std::endl;
```

```
    for (size_t i = 0; i < cloud_out->points.size(); ++i)
  //打印构造出来的目标点云
        std::cout << "    " << cloud_out->points[i].x << " " << cloud_out-
>points[i].y << " " << cloud_out->points[i].z << std::endl;

    // 创建IterativeClosestPoint的实例
    // setInputSource将cloud_in作为输入点云
    // setInputTarget将平移后的cloud_out作为目标点云
    pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> icp;
    icp.setInputSource(cloud_in);
    icp.setInputTarget(cloud_out);

    // 创建一个 pcl::PointCloud<pcl::PointXYZ>实例 Final 对象,存储配准变换后的源点云,
    // 应用 ICP 算法后, IterativeClosestPoint 能够保存结果点云集,如果这两个点云匹配正确的
话
    // （即仅对其中一个应用某种刚体变换，就可以得到两个在同一坐标系下相同的点云）,那么 icp.
hasConverged()= 1 (true),
    // 然后会输出最终变换矩阵的匹配分数和变换矩阵等信息。
    pcl::PointCloud<pcl::PointXYZ> Final;
    icp.align(Final);
    std::cout << "has converged:" << icp.hasConverged() << " score: " <<
icp.getFitnessScore() << std::endl;
    const pcl::Registration<pcl::PointXYZ, pcl::PointXYZ, float>::Matrix4
&matrix = icp.getFinalTransformation();
    std::cout << matrix << std::endl;
    return (0);
}
```

输出结果

可以由试验结果看得出变换后的点云只是在x轴的值增加了固定的值0.7,然后由这目标点云与源点云计算出它的旋转与平移，明显可以看出最后一行的x值为0.7

```
est_point.cpp.o
[100%] Linking CXX executable iterative_closest_point
[100%] Built target iterative_closest_point
➜  build git:(master) ✗ ./iterative_closest_point
Saved 5 data points to input:
    0.352222 -0.151883 -0.106395
    -0.397406 -0.473106 0.292602
    -0.731898 0.667105 0.441304
    -0.734766 0.854581 -0.0361733
    -0.4607 -0.277468 -0.916762
size:5
Transformed 5 data points:
    1.05222 -0.151883 -0.106395
    0.302594 -0.473106 0.292602
    -0.0318983 0.667105 0.441304
    -0.0347655 0.854581 -0.0361733
    0.2393 -0.277468 -0.916762
has converged:1 score: 6.8559e-14
           1 -7.93021e-09 -5.43139e-08          0.7
-2.01282e-07            1 -4.93211e-08  -8.8662e-08
-2.45116e-08  1.85975e-07            1 -1.04308e-08
           0            0            0            1
➜  build git:(master) ✗
```

实现效果

2 如何逐步匹配多幅点云 How to incrementally register pairs of clouds

本实例是使用迭代最近点算法，逐步实现地对一系列点云进行两两匹配，**他的思想是对所有的点云进行变换，使得都与第一个点云统一坐标系中**，在每个连贯的有重叠的点云之间找出最佳的变换，并积累这些变换到全部的点云，能够进行ICP算法的点云需要粗略的预匹配(比如在一个机器人的量距内或者在地图的框架内)，并且一个点云与另一个点云需要有重叠的部分。

代码实现

创建文件：`pairwise_incremental_registration.cpp`

准备资源：`capture0001.pcd capture0002.pcd capture0003.pcd capture0004.pcd capture0005.pcd`

.pcd文件下载地址：https://github.com/PointCloudLibrary/data/tree/master/tutorials/pairwise

（ubuntu）编译执行：`./pairwise_incremental_registration ../pairwise/frame_00000.pcd ../pairwise/capture0001.pcd ../pairwise/capture0002.pcd ../pairwise/capture0003.pcd ../pairwise/capture0004.pcd ../pairwise/capture0005.pcd`

（win）编译执行：

重新生成项目
到该项目的Debug目录下，按住Shift，同时点击鼠标右键，在当前窗口打开CMD窗口。
在命令行中输入

```
.\pairwise_incremental_registration.exe capture0001.pcd capture0002.pcd capture0003.pcd capture0004.pcd capture0005.pcd
```

执行程序。

```
/*
 *如何逐步匹配多幅点云
 */
#include <boost/make_shared.hpp>                //boost指针相关头文件
#include <pcl/point_types.h>                    //点类型定义头文件
#include <pcl/point_cloud.h>                    //点云类定义头文件
#include <pcl/point_representation.h>           //点表示相关的头文件
#include <pcl/io/pcd_io.h>                      //PCD文件打开存储类头文件
#include <pcl/filters/voxel_grid.h>              //用于体素网格化的滤波类头文件
#include <pcl/filters/filter.h>                 //滤波相关头文件
#include <pcl/features/normal_3d.h>             //法线特征头文件
#include <pcl/registration/icp.h>               //ICP类相关头文件
```

```
#include <pcl/registration/icp_nl.h>            //非线性ICP 相关头文件
#include <pcl/registration/transforms.h>        //变换矩阵类头文件
#include <pcl/visualization/pcl_visualizer.h>  //可视化类头文件

using pcl::visualization::PointCloudColorHandlerCustom;
using pcl::visualization::PointCloudColorHandlerGenericField;

//定义
typedef pcl::PointXYZ PointT;
typedef pcl::PointCloud<PointT> PointCloud; //申明pcl::PointXYZ数据
typedef pcl::PointNormal PointNormalT;
typedef pcl::PointCloud<PointNormalT> PointCloudWithNormals;

// 申明一个全局可视化对象变量，定义左右视点分别显示配准前和配准后的结果点云
pcl::visualization::PCLVisualizer *p; //创建可视化对象
int vp_1, vp_2;                         //定义存储 左 右视点的ID

//申明一个结构体方便对点云以文件名和点云对象进行成对处理和管理点云，处理过程中可以同时接受多个点云文件的输入
struct PCD
{
    PointCloud::Ptr cloud; //点云共享指针
    std::string f_name;    //文件名称

    PCD() : cloud(new PointCloud){};
};

struct PCDComparator //文件比较处理
{
    bool operator()(const PCD &p1, const PCD &p2)
    {
        return (p1.f_name < p2.f_name);
    }
};

// 以< x, y, z, curvature >形式定义一个新的点表示
class MyPointRepresentation : public pcl::PointRepresentation<PointNormalT>
{
    using pcl::PointRepresentation<PointNormalT>::nr_dimensions_;

public:
    MyPointRepresentation()
    {
        nr_dimensions_ = 4; //定义点的维度
    }

    // 重载copyToFloatArray方法将点转化为四维数组
    virtual void copyToFloatArray(const PointNormalT &p, float *out) const
    {
        // < x, y, z, curvature >
        out[0] = p.x;
        out[1] = p.y;
        out[2] = p.z;
        out[3] = p.curvature; // 曲率
    }
};

/** \左视图用来显示未匹配的源和目标点云*/
```

```
void showCloudsLeft(const PointCloud::Ptr cloud_target, const PointCloud::Ptr
cloud_source)
{
    p->removePointCloud("vp1_target");
    p->removePointCloud("vp1_source");

    PointCloudColorHandlerCustom<PointT> tgt_h(cloud_target, 0, 255, 0);
    PointCloudColorHandlerCustom<PointT> src_h(cloud_source, 255, 0, 0);
    p->addPointCloud(cloud_target, tgt_h, "vp1_target", vp_1);
    p->addPointCloud(cloud_source, src_h, "vp1_source", vp_1);

    PCL_INFO("Press q to begin the registration.\n");
    p->spin();
}

/** \右边显示配准后的源和目标点云*/
void showCloudsRight(const PointCloudWithNormals::Ptr cloud_target, const
PointCloudWithNormals::Ptr cloud_source)
{
    p->removePointCloud("source");
    p->removePointCloud("target");

    PointCloudColorHandlerGenericField<PointNormalT>
tgt_color_handler(cloud_target, "curvature");
    if (!tgt_color_handler.isCapable())
        PCL_WARN("Cannot create curvature color handler!");

    PointCloudColorHandlerGenericField<PointNormalT>
src_color_handler(cloud_source, "curvature");
    if (!src_color_handler.isCapable())
        PCL_WARN("Cannot create curvature color handler!");

    p->addPointCloud(cloud_target, tgt_color_handler, "target", vp_2);
    p->addPointCloud(cloud_source, src_color_handler, "source", vp_2);

    p->spinOnce();
}

////////////////////////////////////////////////////////////////////////////////
/** \brief Load a set of PCD files that we want to register together
  * \param argc the number of arguments (pass from main ())
  * \param argv the actual command line arguments (pass from main ())
  * \param models the resultant vector of point cloud datasets
  */
void loadData(int argc, char **argv, std::vector<PCD,
Eigen::aligned_allocator<PCD>> &models)
{
    std::string extension(".pcd");
    // 第一个参数是命令本身，所以要从第二个参数开始解析
    for (int i = 1; i < argc; i++)
    {
        std::string fname = std::string(argv[i]);
        // PCD文件名至少为5个字符大小字符串（因为后缀名.pcd就已经占了四个字符位置）
        if (fname.size() <= extension.size())
            continue;

        std::transform(fname.begin(), fname.end(), fname.begin(), (int (*)
(int))tolower);
```

```
        //检查参数是否为一个pcd后缀的文件
        if (fname.compare(fname.size() - extension.size(), extension.size(),
extension) == 0)
        {
            //加载点云并保存在总体的点云列表中
            PCD m;
            m.f_name = argv[i];
            pcl::io::loadPCDFile(argv[i], *m.cloud);
            //从点云中移除NAN点也就是无效点
            std::vector<int> indices;
            pcl::removeNaNFromPointCloud(*m.cloud, *m.cloud, indices);

            models.push_back(m);
        }
    }
}

////////////////////////////////////////////////////////////////////////////////
/** \brief Align a pair of PointCloud datasets and return the result
  * \param cloud_src the source PointCloud
  * \param cloud_tgt the target PointCloud
  * \param output the resultant aligned source PointCloud
  * \param final_transform the resultant transform between source and target
  */
///实现匹配，其中参数有输入一组需要配准的点云，以及是否需要进行下采样，其他参数输出配准后的点云
以及变换矩阵
void pairAlign (const PointCloud::Ptr cloud_src, const PointCloud::Ptr cloud_tgt,
PointCloud::Ptr output, Eigen::Matrix4f &final_transform, bool downsample =
false)
{
    // Downsample for consistency and speed
    // \note enable this for large datasets
    PointCloud::Ptr src(new PointCloud); //存储滤波后的源点云
    PointCloud::Ptr tgt(new PointCloud); //存储滤波后的目标点云
    pcl::VoxelGrid<PointT> grid;         //滤波处理对象
    if (downsample)
    {
        grid.setLeafSize(0.05, 0.05, 0.05); //设置滤波时采用的体素大小
        grid.setInputCloud(cloud_src);
        grid.filter(*src);

        grid.setInputCloud(cloud_tgt);
        grid.filter(*tgt);
    }
    else
    {
        src = cloud_src;
        tgt = cloud_tgt;
    }
    // 计算表面的法向量和曲率
    PointCloudWithNormals::Ptr points_with_normals_src(new
PointCloudWithNormals);
    PointCloudWithNormals::Ptr points_with_normals_tgt(new
PointCloudWithNormals);

    pcl::NormalEstimation<PointT, PointNormalT> norm_est; //点云法线估计对象
    pcl::search::KdTree<pcl::PointXYZ>::Ptr tree(new
pcl::search::KdTree<pcl::PointXYZ>());
```

```
    norm_est.setSearchMethod(tree);
    norm_est.setKSearch(30);

    norm_est.setInputCloud(src);
    norm_est.compute(*points_with_normals_src);
    pcl::copyPointCloud(*src, *points_with_normals_src);

    norm_est.setInputCloud(tgt);
    norm_est.compute(*points_with_normals_tgt);
    pcl::copyPointCloud(*tgt, *points_with_normals_tgt);

    //
    // 实例化我们的自定义点表示（定义见上文）
    MyPointRepresentation point_representation;
    //.并对“曲率”尺寸进行加权，使其与x、y和z平衡
    float alpha[4] = {1.0, 1.0, 1.0, 1.0};
    point_representation.setRescaleValues(alpha);

    //
    // 配准
    pcl::IterativeClosestPointNonLinear<PointNormalT, PointNormalT> reg; // 配准对
象
    reg.setTransformationEpsilon(1e-6);
        //设置收敛判断条件，越小精度越大，收敛也越慢
        // Set the maximum distance between two correspondences (src<->tgt) to
10cm大于此值的点对不考虑
        // Note: adjust this based on the size of your datasets
        reg.setMaxCorrespondenceDistance(0.1);
    // 设置点表示
    reg.setPointRepresentation(boost::make_shared<const MyPointRepresentation>
(point_representation));

    reg.setInputSource(points_with_normals_src); // 设置源点云
    reg.setInputTarget(points_with_normals_tgt); // 设置目标点云
    //
    // Run the same optimization in a loop and visualize the results
    Eigen::Matrix4f Ti = Eigen::Matrix4f::Identity(), prev, targetToSource;
    PointCloudWithNormals::Ptr reg_result = points_with_normals_src;
    reg.setMaximumIterations(2); ////设置最大的迭代次数，即每迭代两次就认为收敛，停止内部
迭代
    for (int i = 0; i < 30; ++i) //手动迭代，每手动迭代一次，在配准结果视口对迭代的最新结
果进行刷新显示
    {
        PCL_INFO("Iteration Nr. %d.\n", i);

        // 存储点云以便可视化
        points_with_normals_src = reg_result;

        // Estimate
        reg.setInputSource(points_with_normals_src);
        reg.align(*reg_result);

        //accumulate transformation between each Iteration
        Ti = reg.getFinalTransformation() * Ti;

        //if the difference between this transformation and the previous one
        //is smaller than the threshold, refine the process by reducing
        //the maximal correspondence distance
```

```
        if (fabs((reg.getLastIncrementalTransformation() - prev).sum()) <
reg.getTransformationEpsilon())
            reg.setMaxCorrespondenceDistance(reg.getMaxCorrespondenceDistance() -
0.001);

        prev = reg.getLastIncrementalTransformation();

        // visualize current state
        showCloudsRight(points_with_normals_tgt, points_with_normals_src);
    }

    //
    // Get the transformation from target to source
    targetToSource = Ti.inverse(); //deidao

    //
    // Transform target back in source frame
    pcl::transformPointCloud(*cloud_tgt, *output, targetToSource);

    p->removePointCloud("source");
    p->removePointCloud("target");

    PointCloudColorHandlerCustom<PointT> cloud_tgt_h(output, 0, 255, 0);
    PointCloudColorHandlerCustom<PointT> cloud_src_h(cloud_src, 255, 0, 0);
    p->addPointCloud(output, cloud_tgt_h, "target", vp_2);
    p->addPointCloud(cloud_src, cloud_src_h, "source", vp_2);

    PCL_INFO("Press q to continue the registration.\n");
    p->spin();

    p->removePointCloud("source");
    p->removePointCloud("target");

    //add the source to the transformed target
    *output += *cloud_src;

    final_transform = targetToSource;
}

int main(int argc, char **argv)
{
    // 存储管理所有打开的点云
    std::vector<PCD, Eigen::aligned_allocator<PCD>> data;
    loadData(argc, argv, data); // 加载所有点云到data

    // 检查输入
    if (data.empty())
    {
        PCL_ERROR("Syntax is: %s <source.pcd> <target.pcd> [*]", argv[0]);
        PCL_ERROR("[*] - multiple files can be added. The registration results of
(i, i+1) will be registered against (i+2), etc");
        return (-1);
    }
    PCL_INFO("Loaded %d datasets.", (int)data.size());

    // 创建PCL可视化对象
    p = new pcl::visualization::PCLVisualizer(argc, argv, "Pairwise Incremental
Registration example");
```

```
    p->createViewPort(0.0, 0, 0.5, 1.0, vp_1); //用左半窗口创建视口vp_1
    p->createViewPort(0.5, 0, 1.0, 1.0, vp_2); //用右半窗口创建视口vp_2

    PointCloud::Ptr result(new PointCloud), source, target;
    Eigen::Matrix4f GlobalTransform = Eigen::Matrix4f::Identity(),
pairTransform;

    for (size_t i = 1; i < data.size(); ++i) //循环处理所有点云================重点
=====================================
    {
        source = data[i - 1].cloud; //连续配准
        target = data[i].cloud;     // 相邻两组点云

        showCloudsLeft(source, target); //可视化为配准的源和目标点云
                                        //调用子函数完成一组点云的配准，temp返回配准后两
组点云在第一组点云坐标下的点云
        PointCloud::Ptr temp(new PointCloud);
        PCL_INFO("Aligning %s (%d) with %s (%d).\n", data[i - 1].f_name.c_str(),
source->points.size(), data[i].f_name.c_str(), target->points.size());
             // pairTransform返回从目标点云target到source的变换矩阵
            pairAlign(source, target, temp, pairTransform, true);  //
===================重点========================

        //把当前两两配准后的点云temp转化到全局坐标系下返回result
        pcl::transformPointCloud(*temp, *result, GlobalTransform);

        //用当前的两组点云之间的变换更新全局变换
        GlobalTransform = GlobalTransform * pairTransform;

        //保存转换到第一个点云坐标下的当前配准后的两组点云result到文件i.pcd
        std::stringstream ss;
        ss << i << ".pcd";
        pcl::io::savePCDFile(ss.str(), *result, true);
    }
}
/* ]--- */
```

输出结果

```
→  build git:(master) ✗ ./pairwise_incremental_registration ../pairwise/frame_00000.pcd ../pairwise/capture0001.pcd  .
ture0004.pcd   ../pairwise/capture0005.pcd
Loaded 6 datasets.Press q to begin the registration.
Aligning ../pairwise/frame_00000.pcd (258074) with ../pairwise/capture0001.pcd (249647).
Iteration Nr. 0.
Iteration Nr. 1.
Iteration Nr. 2.
Iteration Nr. 3.
Iteration Nr. 4.
Iteration Nr. 5.
Iteration Nr. 6.
Iteration Nr. 7.
Iteration Nr. 8.
Iteration Nr. 9.
Iteration Nr. 10.
Iteration Nr. 11.
Iteration Nr. 12.
Iteration Nr. 13.
Iteration Nr. 14.
Iteration Nr. 15.
Iteration Nr. 16.
Iteration Nr. 17.
Iteration Nr. 18.
Iteration Nr. 19.
Iteration Nr. 20.
Iteration Nr. 21.
Iteration Nr. 22.
Iteration Nr. 23.
Iteration Nr. 24.
Iteration Nr. 25.
Iteration Nr. 26.
Iteration Nr. 27.
Iteration Nr. 28.
Iteration Nr. 29.
Press q to continue the registration.
Press q to begin the registration.
Aligning ../pairwise/capture0001.pcd (249647) with ../pairwise/capture0002.pcd (249931).
Iteration Nr. 0.
Iteration Nr. 1.
```

实现效果

如果观察不到结果，就按键R来重设摄像头，**调整角度可以观察到有红绿两组点云显示在窗口的左边，红色为源点云**，将看到下面的类似结果，命令行提示需要执行配准按下Q，按下后可以发现左边的窗口不断的调整点云，其实是配准过程中的迭代中间结果的输出，**在迭代次数小于设定的次数之前，右边会不断刷新最新的配准结果，**直到收敛，迭代次数30次完成整个匹配的过程，**再次按下Q后会看到存储的1.pcd文件，此文件为第一个和第二个点云配准后与第一个输入点云在同一个坐标系下的点云。**