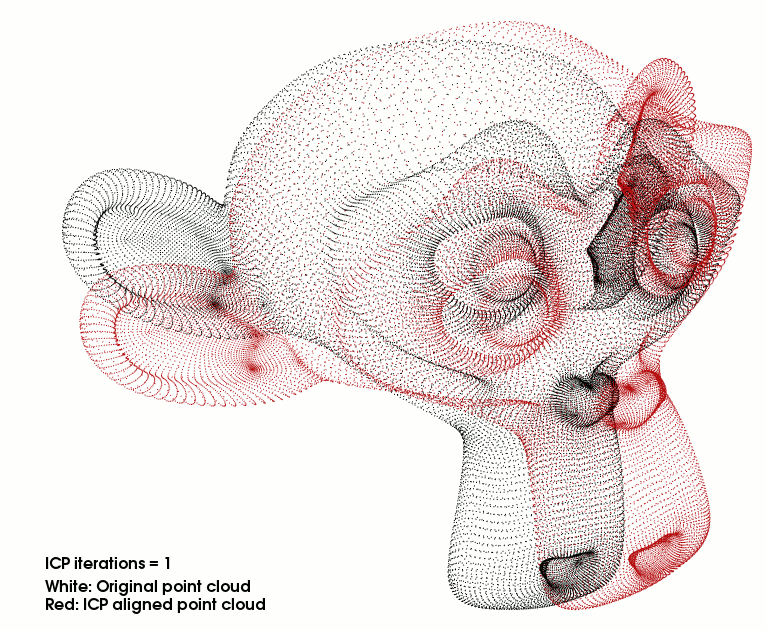
## 点云配准技术及报告

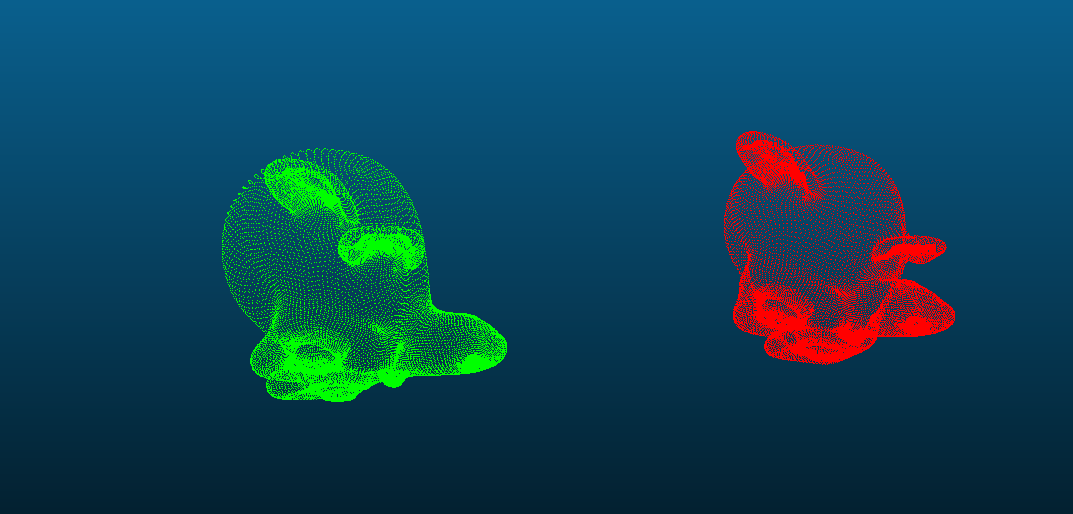
齐勇

2019年8月5日星期一 更新



## 一 PCL中点云配准技术的简单实现

在同一文件夹下，有测试数据文件monkey.ply，该文件是利用Blender创建的默认Monkey模型。



利用如下代码，将**初始点云**（图中绿色点云）进行旋转平移，得到目标点云（图中**红色点云**）。

// 旋转矩阵的具体定义 (请参考 https://en.wikipedia.org/wiki/Rotation\_matrix)

double theta = M\_PI / 20; // 设置旋转弧度的角度

transformation\_matrix (0, 0) = cos (theta);

transformation\_matrix (0, 1) = -sin (theta);

transformation\_matrix (1, 0) = sin (theta);

transformation\_matrix (1, 1) = cos (theta);

// 设置平移矩阵

transformation\_matrix (0, 3) = 0.0;

transformation\_matrix (1, 3) = 0.0;

transformation\_matrix (2, 3) = 4.0;

// 执行初始变换

pcl::transformPointCloud (\*cloud\_in, \*cloud\_icp, transformation\_matrix);

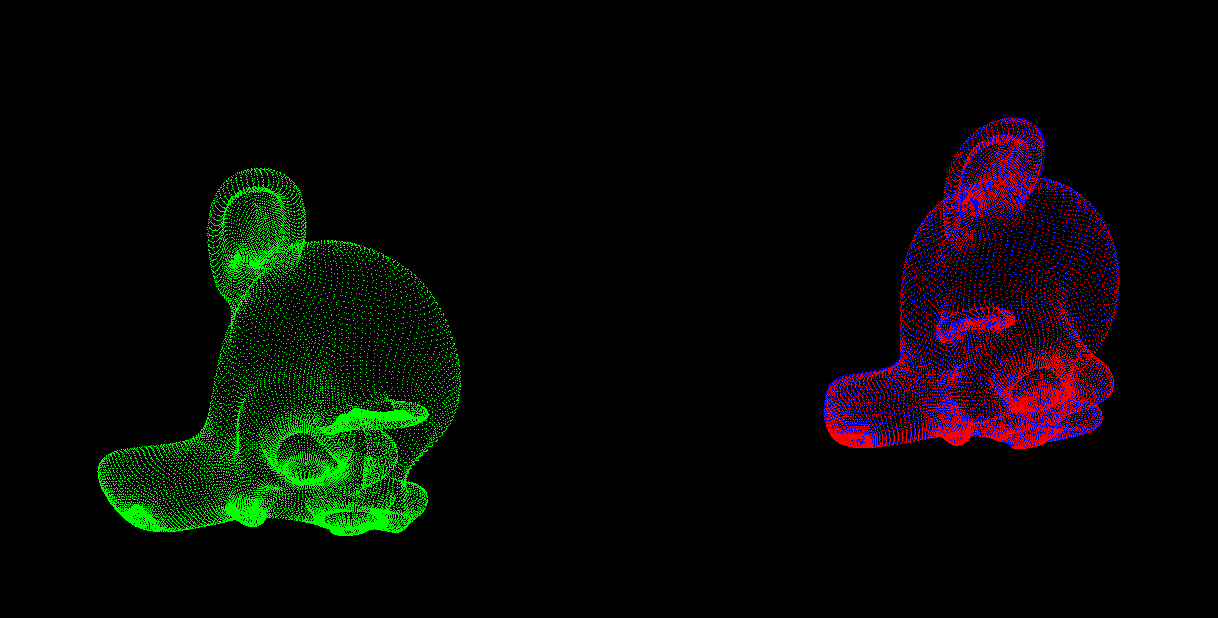
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud\_xyz(new pcl::PointCloud<pcl::PointXYZ>());

pcl::io::savePCDFileASCII("./monkey\_rotated\_trans.pcd",\*cloud\_icp);

完整的PCL点云配准分为粗配准与精配准两个阶段，（此处参考博文：<https://blog.csdn.net/peach_blossom/article/details/78506184，在此，特别感谢作者的无私分享，才能让我们这些后来人少走弯路），示例demo如下（稍作修改）。>

#include **<pcl/registration/ia\_ransac.h>**#include **<pcl/point\_types.h>**#include **<pcl/point\_cloud.h>**#include **<pcl/features/normal\_3d.h>**#include **<pcl/features/fpfh.h>**#include **<pcl/search/kdtree.h>**#include **<pcl/io/pcd\_io.h>**#include **<pcl/io/ply\_io.h>**#include **<pcl/filters/voxel\_grid.h>**#include **<pcl/filters/filter.h>**#include **<pcl/registration/icp.h>**#include **<pcl/visualization/pcl\_visualizer.h>**#include **<time.h>  
  
typedef** pcl::PointXYZ PointT;  
**typedef** pcl::PointCloud<PointT> PointCloud;  
  
*//点云可视化***void** visualize\_pcd(PointCloud::Ptr pcd\_src,  
 PointCloud::Ptr pcd\_tgt,  
 PointCloud::Ptr pcd\_final)  
{  
 *// Create a PCLVisualizer object* pcl::visualization::PCLVisualizer viewer(**"registration Viewer"**);  
 *//viewer.createViewPort (0.0, 0, 0.5, 1.0, vp\_1);  
 // viewer.createViewPort (0.5, 0, 1.0, 1.0, vp\_2);* pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> src\_h (pcd\_src, 0, 255, 0);  
 pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> tgt\_h (pcd\_tgt, 255, 0, 0);  
 pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> final\_h (pcd\_final, 0, 0, 255);  
 viewer.addPointCloud (pcd\_src, src\_h, **"source cloud"**);  
 viewer.addPointCloud (pcd\_tgt, tgt\_h, **"tgt cloud"**);  
 viewer.addPointCloud (pcd\_final, final\_h, **"final cloud"**);  
 *//viewer.addCoordinateSystem(1.0);* **while** (!viewer.wasStopped())  
 {  
 viewer.spinOnce(100);  
 boost::this\_thread::sleep(boost::posix\_time::microseconds(10000));  
 }  
}  
  
*//由旋转平移矩阵计算旋转角度***void** Matrix2Angle (Eigen::Matrix4f &result\_trans,Eigen::Vector3f &result\_angle)  
{  
 **double** ax,ay,az;  
 **if** (result\_trans(2,0)==1 || result\_trans(2,0)==-1)  
 {  
 az=0;  
 **double** dlta;  
 dlta=atan2(result\_trans(0,1),result\_trans(0,2));  
 **if** (result\_trans(2,0)==-1)  
 {  
 ay=**M\_PI**/2;  
 ax=az+dlta;  
 }  
 **else** {  
 ay=-**M\_PI**/2;  
 ax=-az+dlta;  
 }  
 }  
 **else** {  
 ay=-asin(result\_trans(2,0));  
 ax=atan2(result\_trans(2,1)/cos(ay),result\_trans(2,2)/cos(ay));  
 az=atan2(result\_trans(1,0)/cos(ay),result\_trans(0,0)/cos(ay));  
 }  
 result\_angle<<ax,ay,az;  
}  
  
  
**void** doRegistration(std::string src\_cloud\_path,std::string tgt\_cloud\_path, Eigen::Matrix4f &icp\_trans)  
{  
 *//加载点云文件* PointCloud::Ptr cloud\_src\_origin (**new** PointCloud);*//原点云，待配准* pcl::io::loadPLYFile (src\_cloud\_path,\*cloud\_src\_origin);  
 PointCloud::Ptr cloud\_tgt\_origin (**new** PointCloud);*//目标点云* pcl::io::loadPCDFile (tgt\_cloud\_path,\*cloud\_tgt\_origin);  
  
 clock\_t start=clock();  
 *//去除NAN点* std::vector<**int**> indices\_src; *//保存去除的点的索引* pcl::removeNaNFromPointCloud(\*cloud\_src\_origin,\*cloud\_src\_origin, indices\_src);  
 std::cout<<**"remove \*cloud\_src\_origin nan"**<<endl;  
   
 *//下采样滤波* pcl::VoxelGrid<pcl::PointXYZ> voxel\_grid;  
 voxel\_grid.setLeafSize(0.014,0.014,0.014);  
 voxel\_grid.setInputCloud(cloud\_src\_origin);  
 PointCloud::Ptr cloud\_src (**new** PointCloud);  
 voxel\_grid.filter(\*cloud\_src);  
 std::cout<<**"down size \*cloud\_src\_origin from "**<<cloud\_src\_origin->size()<<**"to"**<<cloud\_src->size()<<endl;  
 *//pcl::io::savePCDFileASCII("monkey\_src\_down.pcd",\*cloud\_src);  
  
 //计算表面法线* pcl::NormalEstimation<pcl::PointXYZ,pcl::Normal> ne\_src;  
 ne\_src.setInputCloud(cloud\_src);  
 pcl::search::KdTree< pcl::PointXYZ>::Ptr tree\_src(**new** pcl::search::KdTree< pcl::PointXYZ>());  
 ne\_src.setSearchMethod(tree\_src);  
 pcl::PointCloud<pcl::Normal>::Ptr cloud\_src\_normals(**new** pcl::PointCloud< pcl::Normal>);  
 ne\_src.setRadiusSearch(0.02);  
 ne\_src.compute(\*cloud\_src\_normals);  
  
 std::vector<**int**> indices\_tgt;  
 pcl::removeNaNFromPointCloud(\*cloud\_tgt\_origin,\*cloud\_tgt\_origin, indices\_tgt);  
 std::cout<<**"remove \*cloud\_tgt\_origin nan"**<<endl;  
  
 pcl::VoxelGrid<pcl::PointXYZ> voxel\_grid\_2;  
 voxel\_grid\_2.setLeafSize(0.01,0.01,0.01);  
 voxel\_grid\_2.setInputCloud(cloud\_tgt\_origin);  
 PointCloud::Ptr cloud\_tgt (**new** PointCloud);  
 voxel\_grid\_2.filter(\*cloud\_tgt);  
 std::cout<<**"down size \*cloud\_tgt\_origin.pcd from "**<<cloud\_tgt\_origin->size()<<**"to"**<<cloud\_tgt->size()<<endl;  
 *//pcl::io::savePCDFileASCII("monkey\_tgt\_down.pcd",\*cloud\_tgt);* pcl::NormalEstimation<pcl::PointXYZ,pcl::Normal> ne\_tgt;  
 ne\_tgt.setInputCloud(cloud\_tgt);  
 pcl::search::KdTree< pcl::PointXYZ>::Ptr tree\_tgt(**new** pcl::search::KdTree< pcl::PointXYZ>());  
 ne\_tgt.setSearchMethod(tree\_tgt);  
 pcl::PointCloud<pcl::Normal>::Ptr cloud\_tgt\_normals(**new** pcl::PointCloud< pcl::Normal>);  
 *//ne\_tgt.setKSearch(20);* ne\_tgt.setRadiusSearch(0.02);  
 ne\_tgt.compute(\*cloud\_tgt\_normals);  
  
 *//计算FPFH* pcl::FPFHEstimation<pcl::PointXYZ,pcl::Normal,pcl::FPFHSignature33> fpfh\_src;  
 fpfh\_src.setInputCloud(cloud\_src);  
 fpfh\_src.setInputNormals(cloud\_src\_normals);  
 pcl::search::KdTree<PointT>::Ptr tree\_src\_fpfh (**new** pcl::search::KdTree<PointT>);  
 fpfh\_src.setSearchMethod(tree\_src\_fpfh);  
 pcl::PointCloud<pcl::FPFHSignature33>::Ptr fpfhs\_src(**new** pcl::PointCloud<pcl::FPFHSignature33>());  
 fpfh\_src.setRadiusSearch(0.05);  
 fpfh\_src.compute(\*fpfhs\_src);  
 std::cout<<**"compute \*cloud\_src fpfh"**<<endl;  
  
 pcl::FPFHEstimation<pcl::PointXYZ,pcl::Normal,pcl::FPFHSignature33> fpfh\_tgt;  
 fpfh\_tgt.setInputCloud(cloud\_tgt);  
 fpfh\_tgt.setInputNormals(cloud\_tgt\_normals);  
 pcl::search::KdTree<PointT>::Ptr tree\_tgt\_fpfh (**new** pcl::search::KdTree<PointT>);  
 fpfh\_tgt.setSearchMethod(tree\_tgt\_fpfh);  
 pcl::PointCloud<pcl::FPFHSignature33>::Ptr fpfhs\_tgt(**new** pcl::PointCloud<pcl::FPFHSignature33>());  
 fpfh\_tgt.setRadiusSearch(0.05);  
 fpfh\_tgt.compute(\*fpfhs\_tgt);  
 std::cout<<**"compute \*cloud\_tgt fpfh"**<<endl;  
  
 *//SAC配准* pcl::SampleConsensusInitialAlignment<pcl::PointXYZ, pcl::PointXYZ, pcl::FPFHSignature33> scia;  
 scia.setInputSource(cloud\_src);  
 scia.setInputTarget(cloud\_tgt);  
 scia.setSourceFeatures(fpfhs\_src);  
 scia.setTargetFeatures(fpfhs\_tgt);  
 *//scia.setMinSampleDistance(1);  
 //scia.setNumberOfSamples(2);  
 //scia.setCorrespondenceRandomness(20);* PointCloud::Ptr sac\_result (**new** PointCloud);  
 scia.align(\*sac\_result);  
 std::cout <<**"sac has converged:"**<<scia.hasConverged()<<**" score: "**<<scia.getFitnessScore()<<endl;  
 Eigen::Matrix4f sac\_trans;  
 sac\_trans=scia.getFinalTransformation();  
 std::cout<<sac\_trans<<endl;  
 *//pcl::io::savePCDFileASCII("monkey\_transformed\_sac.pcd",\*sac\_result);* clock\_t sac\_time=clock();  
  
 *//icp配准* PointCloud::Ptr icp\_result (**new** PointCloud);  
 pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> icp;  
 icp.setInputSource(cloud\_src);  
 icp.setInputTarget(cloud\_tgt\_origin);  
 icp.setMaxCorrespondenceDistance (0.04);  
 *// 最大迭代次数* icp.setMaximumIterations (2);  
 *// 两次变化矩阵之间的差值* icp.setTransformationEpsilon (1e-10);  
 *// 均方误差* icp.setEuclideanFitnessEpsilon (0.2);  
 icp.align(\*icp\_result,sac\_trans);  
  
 clock\_t end=clock();  
 cout<<**"total time: "**<<(**double**)(end-start)/(**double**)**CLOCKS\_PER\_SEC**<<**" s"**<<endl;  
  
 cout<<**"sac time: "**<<(**double**)(sac\_time-start)/(**double**)**CLOCKS\_PER\_SEC**<<**" s"**<<endl;  
 cout<<**"icp time: "**<<(**double**)(end-sac\_time)/(**double**)**CLOCKS\_PER\_SEC**<<**" s"**<<endl;  
  
 std::cout << **"ICP has converged:"** << icp.hasConverged()  
 << **" score: "** << icp.getFitnessScore() << std::endl;  
  
 icp\_trans=icp.getFinalTransformation();  
 *//cout<<"ransformationProbability"<<icp.getTransformationProbability()<<endl;* std::cout<<icp\_trans<<endl;  
 *//使用创建的变换对未过滤的输入点云进行变换* pcl::transformPointCloud(\*cloud\_src\_origin, \*icp\_result, icp\_trans);  
 *//保存转换的输入点云  
 //pcl::io::savePCDFileASCII("monkey\_transformed\_sac\_ndt.pcd", \*icp\_result);  
  
 //计算误差* Eigen::Vector3f ANGLE\_origin;  
 ANGLE\_origin<<0,0,**M\_PI**/5;  
 **double** error\_x,error\_y,error\_z;  
 Eigen::Vector3f ANGLE\_result;  
 Matrix2Angle(icp\_trans,ANGLE\_result);  
 error\_x=fabs(ANGLE\_result(0))-fabs(ANGLE\_origin(0));  
 error\_y=fabs(ANGLE\_result(1))-fabs(ANGLE\_origin(1));  
 error\_z=fabs(ANGLE\_result(2))-fabs(ANGLE\_origin(2));  
 cout<<**"original angle in x y z:\n"**<<ANGLE\_origin<<endl;  
 cout<<**"error in aixs\_x: "**<<error\_x<<**" error in aixs\_y: "**<<error\_y<<**" error in aixs\_z: "**<<error\_z<<endl;  
 *//可视化* visualize\_pcd(cloud\_src\_origin,cloud\_tgt\_origin,icp\_result);  
}  
  
**int**main (**int** argc, **char**\*\* argv)  
{  
 std::string src\_cloud\_path=**"../monkey.ply"**;  
 std::string tgt\_cloud\_path=**"../monkey\_rotated\_trans.pcd"**;  
 Eigen::Matrix4f icp\_trans;  
 doRegistration(src\_cloud\_path,tgt\_cloud\_path,icp\_trans);  
 **return** (0);  
}

运行后的效果图如下。



可以发现，程序中即使设置了最大迭代次数为2，但配准后的效果已经很好啦（红色点云与蓝色点云几乎已经重合）。

## 二 深剖PCL中配准技术细节

对于以上配准过程，大致可以总结如下：

**第一步：**加载点云；

**第二步：**下采样滤波；

**第三步：**计算表面法线；

**第四步：**计算FPFH；

**第五步：**SAC\_IA粗配准；

**第六步：**ICP精配准。

接下来，我们简单介绍下各过程，如有不当之处，还请留言批评指正。

**对于第一步：**加载点云，demo中给出了两种点云的格式的读取（ply和pcd格式），当然在PCL中，还有其它数据格式的读取函数封装，比如txt，以及二进制数据格式的读取。

**对于第二步：**关于下采样滤波，PCL实现的Voxel类通过输入的点云数据创建一个三维体素栅格（可把体素栅格想象为微小的空间三维立方体的集合），然后在每个体素（即三维立方体）内，用体素中所有点的重心来近似显示体素中其他点，这样该体素内所有点就用一个重心点最终表示，对于所有体素处理后得到滤波后的点云。

**对于第三步：**计算表面法线。

表面法线是几何表面的重要属性，在很多领域都有大量应用，比如：在进行光照渲染时产生符合可视习惯时需要表面法线信息才能正常表达，对于一个已知的几何体表面，根据垂直于点表面的矢量，因此推断表面某一点的法线方向通常比较简单。然而，由于我们获取的点云数据集在真实物体的表面表现为一组定点样本，这样通常就会有两种解决办法：

1. 使用曲面重建技术，从获取的点云数据集中得到采样点对应的曲面，然后从曲面模型中计算表面法线。
2. 直接从点云数据集中近似推断表面法线。

此处我们简单介绍下对于已知一个点云数据集，如何在每个点处直接近似计算表面法线。

方法简单表述如下：确定表面一点法线的问题近似于估计表面的一个相切面法线的问题，因此转换过来以后就变成一个最小二乘法平面拟合估计问题。因此，估计表面法线的解决方案就变成了分析一个协方差矩阵的特征矢量和特征值（或者PCA——主成分分析），这个协方差矩阵从查询点的近邻元素中创建。

在PCL内估计一个点集对应的协方差矩阵，可以使用以下函数调用实现：

//定义每个表面小块的3x3协方差矩阵的存储对象

Eigen::Matrix3f covariance\_matrix;

//定义一个表面小块的质心坐标16字节对齐存储对象

Eigen::Vector4f xyz\_centroid;

//估计质心坐标

Compute3DCentroid(cloud,xyz\_centroid);

//计算3x3协方差矩阵

computeCovarianceMatrix(cloud,xyz\_centroid,covariance\_matrix);

对于法线的正负向如何确定问题，此处留给读者思考。

**对于第四步：计算FPFH。**

已知点云P中有n个点，那么它的点特征直方图（PFH）的理论计算复杂度是O(nk2),其中k是点云P中每个点p计算特征向量时考虑的邻域数量。对于实时应用或接近实时应用中，密集点云的点特征直方图（PFH）的计算，是一个主要的性能瓶颈。

此处简单介绍下FPFH（快速点特征信息）与PFH(点特征直方图)的主要区别。

1. FPFH没有对全互联Pq点的所有邻近点的计算参数进行统计，因此可能会漏掉了一些重要的点对，而这些漏掉的点对可能对捕获查询点周围的几何特征有贡献。
2. PFH特征模型是在查询点周围的一个精确的邻域半径内，而FPFH还包括半径r范围以外的额外点对（不过在2r内）。

**第五步：SAC\_IA粗配准**

对于初始的变换矩阵粗略估计，贪婪的初始配准方法工作量很大，它使用了点云数据旋转不变的特性。但计算复杂度较高，因此在合并的步骤需要查看所有可能的对应关系。此外，因为这是一个贪婪算法，所以有可能只能得到局部最优解。

因此我们采用采样一致性方法，试图保持相同的对应关系而不必尝试了解有限个对应关系的所有组合。相反，我们从候选对应关系中进行大量的采样并通过以下的步骤对它们中的每一个进行排名。

1. 从点云P中选择n个样本点，为了尽量保证所采样的点具有不同的FPFH特征，确定他们的配对距离大于用户设定的最小值dmin；
2. 对于每个样本点，在点云Q中找到满足FPFH相似的点存入一个列表中。从这些点中随机选择一些代表采样点的对应关系；
3. 计算通过采样点定义的刚体变换和其对应变换，计算点云的度量错误来评价转换的质量。

重复上述的三个步骤，直至取得储存了最佳度量错误，并使用暴力配准部分数据。最后使用Levenberg-Marquardt算法进行非线性局部优化。

**第六步：迭代最近点算法（ICP精配准）**

我们先来观察一下PCL1.8中封装了ICP的函数实现，此处附上ICP的核心函数实现源码：

///////////////////////////////////////////////////////////////////////////////////////////

template <typename PointSource, typename PointTarget, typename Scalar> void

pcl::IterativeClosestPoint<PointSource, PointTarget, Scalar>::computeTransformation (

PointCloudSource &output, const Matrix4 &guess)

{

// Point cloud containing the correspondences of each point in <input, indices>

PointCloudSourcePtr input\_transformed (new PointCloudSource);

nr\_iterations\_ = 0;

converged\_ = false;

// Initialise final transformation to the guessed one

final\_transformation\_ = guess;

// If the guessed transformation is non identity

if (guess != Matrix4::Identity ())

{

input\_transformed->resize (input\_->size ());

// Apply guessed transformation prior to search for neighbours

transformCloud (\*input\_, \*input\_transformed, guess);

}

else

\*input\_transformed = \*input\_;

transformation\_ = Matrix4::Identity ();

// Make blobs if necessary

determineRequiredBlobData ();

PCLPointCloud2::Ptr target\_blob (new PCLPointCloud2);

if (need\_target\_blob\_)

pcl::toPCLPointCloud2 (\*target\_, \*target\_blob);

// Pass in the default target for the Correspondence Estimation/Rejection code

correspondence\_estimation\_->setInputTarget (target\_);

if (correspondence\_estimation\_->requiresTargetNormals ())

correspondence\_estimation\_->setTargetNormals (target\_blob);

// Correspondence Rejectors need a binary blob

for (size\_t i = 0; i < correspondence\_rejectors\_.size (); ++i)

{

registration::CorrespondenceRejector::Ptr& rej = correspondence\_rejectors\_[i];

if (rej->requiresTargetPoints ())

rej->setTargetPoints (target\_blob);

if (rej->requiresTargetNormals () && target\_has\_normals\_)

rej->setTargetNormals (target\_blob);

}

convergence\_criteria\_->setMaximumIterations (max\_iterations\_);

convergence\_criteria\_->setRelativeMSE (euclidean\_fitness\_epsilon\_);

convergence\_criteria\_->setTranslationThreshold (transformation\_epsilon\_);

if (transformation\_rotation\_epsilon\_ > 0)

convergence\_criteria\_->setRotationThreshold (transformation\_rotation\_epsilon\_);

else

convergence\_criteria\_->setRotationThreshold (1.0 - transformation\_epsilon\_);

// Repeat until convergence

do

{

// Get blob data if needed

PCLPointCloud2::Ptr input\_transformed\_blob;

if (need\_source\_blob\_)

{

input\_transformed\_blob.reset (new PCLPointCloud2);

toPCLPointCloud2 (\*input\_transformed, \*input\_transformed\_blob);

}

// Save the previously estimated transformation

previous\_transformation\_ = transformation\_;

// Set the source each iteration, to ensure the dirty flag is updated

correspondence\_estimation\_->setInputSource (input\_transformed);

if (correspondence\_estimation\_->requiresSourceNormals ())

correspondence\_estimation\_->setSourceNormals (input\_transformed\_blob);

// Estimate correspondences

if (use\_reciprocal\_correspondence\_)

correspondence\_estimation\_->determineReciprocalCorrespondences (\*correspondences\_, corr\_dist\_threshold\_);

else

correspondence\_estimation\_->determineCorrespondences (\*correspondences\_, corr\_dist\_threshold\_);

//if (correspondence\_rejectors\_.empty ())

CorrespondencesPtr temp\_correspondences (new Correspondences (\*correspondences\_));

for (size\_t i = 0; i < correspondence\_rejectors\_.size (); ++i)

{

registration::CorrespondenceRejector::Ptr& rej = correspondence\_rejectors\_[i];

PCL\_DEBUG ("Applying a correspondence rejector method: %s.\n", rej->getClassName ().c\_str ());

if (rej->requiresSourcePoints ())

rej->setSourcePoints (input\_transformed\_blob);

if (rej->requiresSourceNormals () && source\_has\_normals\_)

rej->setSourceNormals (input\_transformed\_blob);

rej->setInputCorrespondences (temp\_correspondences);

rej->getCorrespondences (\*correspondences\_);

// Modify input for the next iteration

if (i < correspondence\_rejectors\_.size () - 1)

\*temp\_correspondences = \*correspondences\_;

}

size\_t cnt = correspondences\_->size ();

// Check whether we have enough correspondences

if (static\_cast<int> (cnt) < min\_number\_correspondences\_)

{

PCL\_ERROR ("[pcl::%s::computeTransformation] Not enough correspondences found. Relax your threshold parameters.\n", getClassName ().c\_str ());

convergence\_criteria\_->setConvergenceState(pcl::registration::DefaultConvergenceCriteria<Scalar>::CONVERGENCE\_CRITERIA\_NO\_CORRESPONDENCES);

converged\_ = false;

break;

}

// Estimate the transform

transformation\_estimation\_->estimateRigidTransformation (\*input\_transformed, \*target\_, \*correspondences\_, transformation\_);

// Transform the data

transformCloud (\*input\_transformed, \*input\_transformed, transformation\_);

// Obtain the final transformation

final\_transformation\_ = transformation\_ \* final\_transformation\_;

++nr\_iterations\_;

// Update the vizualization of icp convergence

//if (update\_visualizer\_ != 0)

// update\_visualizer\_(output, source\_indices\_good, \*target\_, target\_indices\_good );

converged\_ = static\_cast<bool> ((\*convergence\_criteria\_));

}

while (convergence\_criteria\_->getConvergenceState() == pcl::registration::DefaultConvergenceCriteria<Scalar>::CONVERGENCE\_CRITERIA\_NOT\_CONVERGED);

// Transform the input cloud using the final transformation

PCL\_DEBUG ("Transformation is:\n\t%5f\t%5f\t%5f\t%5f\n\t%5f\t%5f\t%5f\t%5f\n\t%5f\t%5f\t%5f\t%5f\n\t%5f\t%5f\t%5f\t%5f\n",

final\_transformation\_ (0, 0), final\_transformation\_ (0, 1), final\_transformation\_ (0, 2), final\_transformation\_ (0, 3),

final\_transformation\_ (1, 0), final\_transformation\_ (1, 1), final\_transformation\_ (1, 2), final\_transformation\_ (1, 3),

final\_transformation\_ (2, 0), final\_transformation\_ (2, 1), final\_transformation\_ (2, 2), final\_transformation\_ (2, 3),

final\_transformation\_ (3, 0), final\_transformation\_ (3, 1), final\_transformation\_ (3, 2), final\_transformation\_ (3, 3));

// Copy all the values

output = \*input\_;

// Transform the XYZ + normals

transformCloud (\*input\_, output, final\_transformation\_);

}

在PCL中，IterativeClosestPoint类提供了标准ICP算法的实现（The transformation is estimation based on SVD），其大致思路如下：

1. 将初始配准后的两片点云P’（经过坐标变换后的源点云）和点云Q,作为精配准的初始点集；
2. 对源点云P’中的每一点Pi，在目标点云Q中寻找距离最近的对应点Qi，作为该点在目标点云中的对应点，组成初始对应点对；
3. 初始对应点集中的对应关系并不一定都是正确的，错误的对应关系会影响最终的配准结果，采用方向向量阈值提剔除错误的对应点对；
4. 计算旋转矩阵R和平移向量T，使对应点集之间的均方误差最小；
5. 设定某一阈值E和最大迭代次数N,将上一步得到的刚体变换作用于源点云P’，得到新点云P’’，计算P’’和Q的距离误差，如果两次迭代的误差小于阈值E或者当前的迭代次数大于N，则迭代结束，否则将初始配准的点集更新为P’’和Q,继续重复上述步骤，直至满足收敛条件。

ICP算法对于参数较为敏感，此处简单介绍一下每个参数的含义：

1. setMaximumIterations 设置最大迭代次数，ICP算法最多迭代的次数。
2. setEuclideanFitnessEpsilon 设置收敛条件是均方误差和小于阈值，停止迭代。
3. setMaxCorrespondenceDistance 设置对应点对之间的最大距离（影响因素较大）。
4. setTransformationEpsilon 设置两次变换矩阵之间的差值（一般设置为1e-10）。

## 三 对于PCL中点云配准的耗时与精度测试实验

### 3.1 第一组测试：

最大迭代次数：2

源点云与目标点云数量：125952

体素滤波采样后：22195/26404

sac has converged:1 score: 0.000241576

0.988262 -0.152289 0.0121022 0.0198728

0.15247 0.988182 -0.01579 0.0170109

-0.0095546 0.0174499 0.999802 4.06545

0 0 0 1

total time: 116.16 s

sac time: 115.604 s

icp time: 0.555811 s

ICP has converged:1 score: 4.49345e-05

0.987662 -0.156512 0.00542931 0.000578808

0.156549 0.987646 -0.00710325 0.00409905

-0.00425052 0.00786565 0.999961 4.02878

0 0 0 1

original angle in x y z:

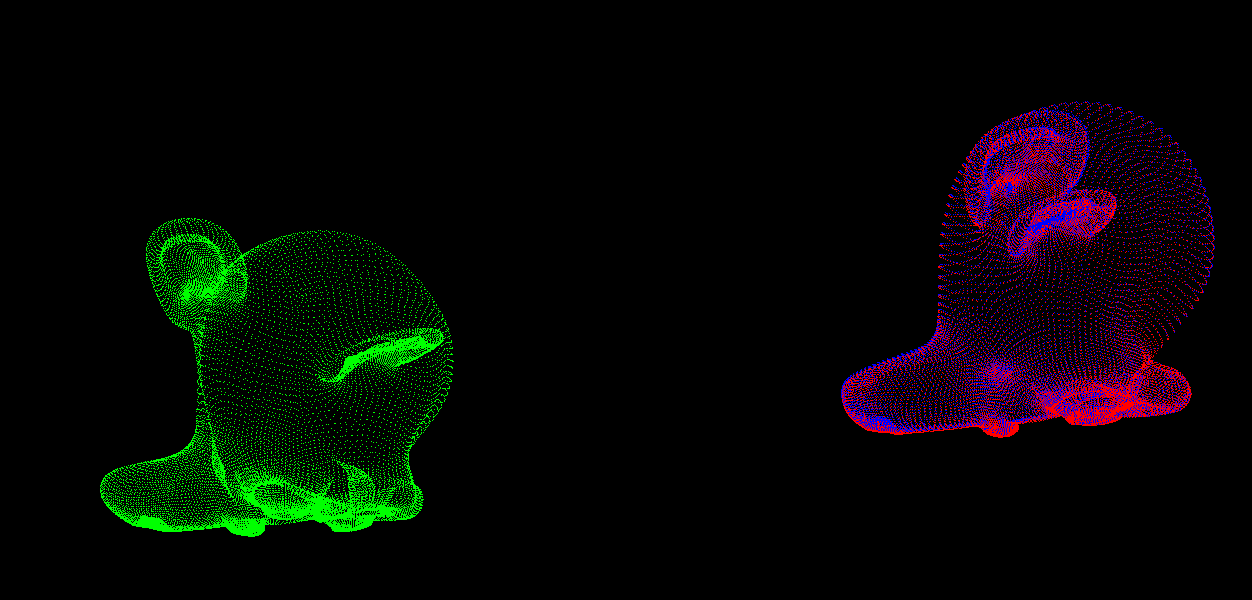
0

0

0.628319

error in aixs\_x: 0.0078658 error in aixs\_y: 0.00425053 error in aixs\_z: -0.471122

配准后效果图：



### 3.2 第二组测试：

最大迭代次数：10

源点云与目标点云数量：125952

体素滤波采样后：22195/26404

sac has converged:1 score: 0.000241576

0.988262 -0.152289 0.0121022 0.0198728

0.15247 0.988182 -0.01579 0.0170109

-0.0095546 0.0174499 0.999802 4.06545

0 0 0 1

total time: 121.906 s

sac time: 120.712 s

icp time: 1.19339 s

ICP has converged:1 score: 4.3379e-06

0.987661 -0.15662 0.0002798 -0.00053462

0.15662 0.98766 -0.000391963 1.63327e-05

-0.000214985 0.000431097 1 4.00136

0 0 0 1

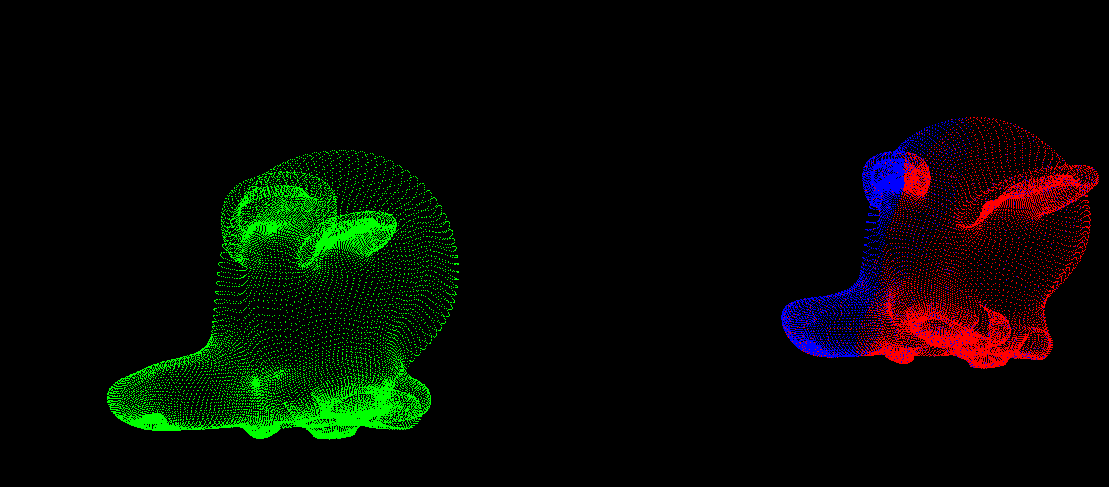
original angle in x y z:

0

0

0.628319

error in aixs\_x: 0.000431096 error in aixs\_y: 0.000214985 error in aixs\_z: -0.471051



### 3.3 第三组实验

最大迭代次数：20

源点云与目标点云数量：125952

体素滤波采样后：22195/26404

sac has converged:1 score: 0.000241576

0.988262 -0.152289 0.0121022 0.0198728

0.15247 0.988182 -0.01579 0.0170109

-0.0095546 0.0174499 0.999802 4.06545

0 0 0 1

total time: 123.31 s

sac time: 122.05 s

icp time: 1.26003 s

ICP has converged:1 score: 4.3379e-06

0.987661 -0.15662 0.0002798 -0.00053462

0.15662 0.98766 -0.000391963 1.63327e-05

-0.000214985 0.000431097 1 4.00136

0 0 0 1

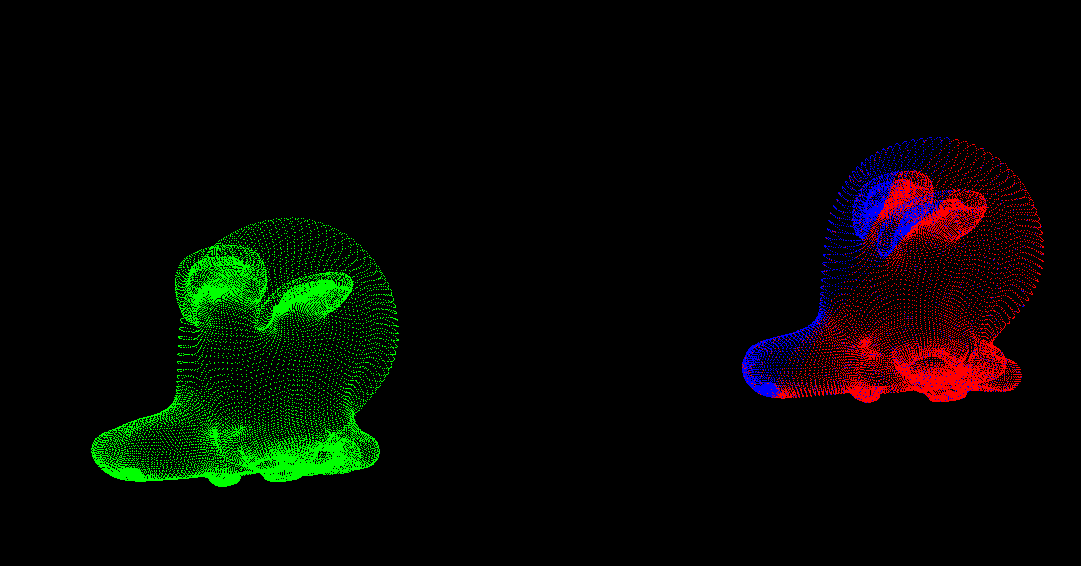
original angle in x y z:

0

0

0.628319

error in aixs\_x: 0.000431096 error in aixs\_y: 0.000214985 error in aixs\_z: -0.471051



由以上实验可以得出，对于两幅较为理想的点云，对于配准：

1. 迭代次数越多，耗时越多；
2. 粗配准的SAC耗时，远大于后期的精配准ICP耗时；
3. 对于Z轴的配准精度，相较于X、Y轴，要差一些。

## 四 对数据集进行点云拼接测试

### 4.1对《视觉SLAM十四讲》中的点云拼接demo测试

测试图片：640×680；

样图demo展示：



共拼接5张图，（1081843个点）

共用时：0.054427s



### 4.2 对TUM数据集进行点云拼接

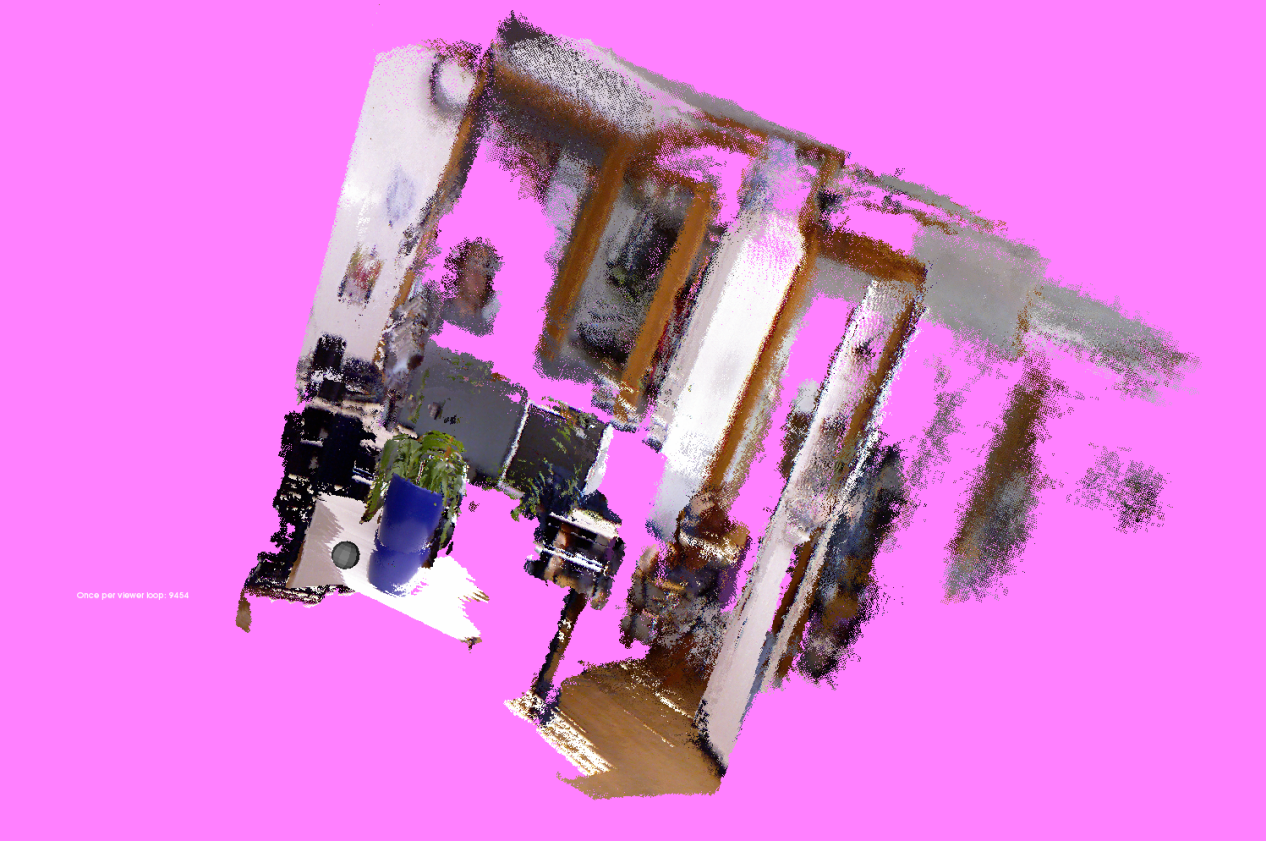
测试图片：640×680；

样图demo展示：



共拼接20张图，（有4445809个点）

共用时：0.236349s



从以上两组实验的数据可以得出如下结论：

对于每个位姿下的相机姿态估计精度较高时，直接将点云进行拼接即可得到全局点云图，无需配准操作，处理速度较高，效果较好。

## 五 展望与思考

当然，对于四中的相机位姿，我们是通过高精度的IMU可以得到，但是在实际使用中，需要我们进行估计相机的位姿，以便为后续进行ICP精配准提供初值（这也是粗配的目的）。因而，如何高精度地获得相机的每个位姿矩阵，成为我们解决点云配准问题的关键，也是在CPU上进行实时重建的关键。