OPEN3D点云全局配准

[1. 概述 2](#_Toc32669)

[2. 可视化 2](#_Toc3243)

[3. 输入 2](#_Toc31443)

[4. RANSAC 3](#_Toc2390)

[5. 局部优化 5](#_Toc18441)

[6. 快速全局配准 6](#_Toc11687)

[7. 基准 7](#_Toc13910)

[8. 快速全局配准 7](#_Toc21410)

[9. 多视角点云配准 9](#_Toc27112)

[(1) 输入 9](#_Toc24583)

[(2) 姿态图 9](#_Toc2182)

[10. 参考文献 13](#_Toc3432)

# **概述**

Open3D是一个开源库，支持快速开发和处理3D数据。Open3D在c++和Python中公开了一组精心选择的数据结构和算法。后端是高度优化的，并且是为并行化而设置的。

ICP配准和彩色点云配准都被称为局部点云配准方法，因为他们都依赖一个粗糙的对齐作为初始化。本篇教程将会展现另一种被称为全局配准的配准方法.这种系列的算法不要求一个初始化的对齐,通常会输出一个没那么精准的对齐结果,并且使用该结果作为局部配准的初始化.

# 可视化

该辅助函数可以将配准的源点云和目标点云一起可视化。

def draw\_registration\_result(source, target, transformation):

source\_temp = copy.deepcopy(source)

target\_temp = copy.deepcopy(target)

source\_temp.paint\_uniform\_color([1, 0.706, 0])

target\_temp.paint\_uniform\_color([0, 0.651, 0.929])

source\_temp.transform(transformation)

o3d.visualization.draw\_geometries([source\_temp, target\_temp])

注意:这里原来的教程里可视化函数都加了初始视角之类的,但是很多人反映这个会报错,并且官方函数里也没给出可接受的参数,所以在这里把初始视角的参数都去掉了

提取几何特征

我们降采样点云,估计法线,之后对每个点计算FPFH特征.FPFH特征是一个描述点的局部几何属性的33维的向量.在33维空间中进行最近邻查询可以返回具有近似几何结构的点.详细细节请查看 [Rasu2009].

def preprocess\_point\_cloud(pcd, voxel\_size):  
 print(":: 使用大小为为{}的体素下采样点云.".format(voxel\_size))  
 pcd\_down = pcd.voxel\_down\_sample(voxel\_size)  
  
 radius\_normal = voxel\_size \* 2  
 print(":: 使用搜索半径为{}估计法线".format(radius\_normal))  
 pcd\_down.estimate\_normals(o3d.geometry.KDTreeSearchParamHybrid(radius=radius\_normal, max\_nn=30))  
 radius\_feature = voxel\_size \* 5  
 print(":: 使用搜索半径为{}计算FPFH特征".format(radius\_feature))  
 pcd\_fpfh = o3d.pipelines.registration.compute\_fpfh\_feature(pcd\_down, o3d.geometry.KDTreeSearchParamHybrid(  
 radius=radius\_feature, max\_nn=100))  
 return pcd\_down, pcd\_fpfh

# 输入

以下代码从两个文件中读取源点云和目标点云.这一对点云使用单位矩阵作为初始矩阵之后是不对齐的.

def prepare\_dataset(voxel\_size):

print(":: Load two point clouds and disturb initial pose.")

source = o3d.io.read\_point\_cloud("../../TestData/ICP/cloud\_bin\_0.pcd")

target = o3d.io.read\_point\_cloud("../../TestData/ICP/cloud\_bin\_1.pcd")

trans\_init = np.asarray([[0.0, 0.0, 1.0, 0.0], [1.0, 0.0, 0.0, 0.0],

[0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 0.0, 1.0]])

source.transform(trans\_init)

draw\_registration\_result(source, target, np.identity(4))

source\_down, source\_fpfh = preprocess\_point\_cloud(source, voxel\_size)

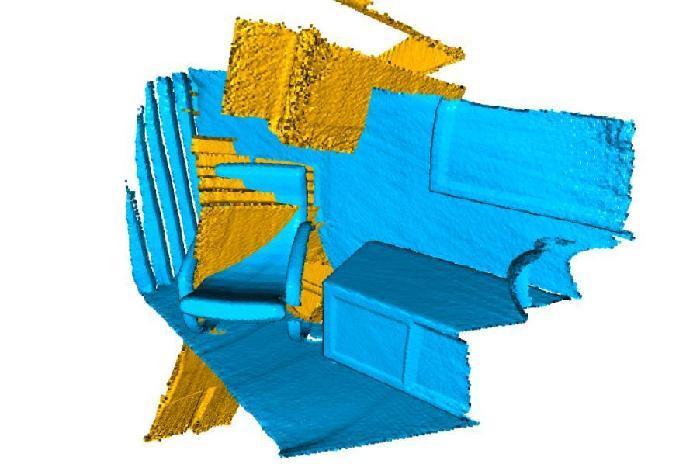
target\_down, target\_fpfh = preprocess\_point\_cloud(target, voxel\_size)

return source, target, source\_down, target\_down, source\_fpfh, target\_fpfh

voxel\_size = 0.05 # means 5cm for this dataset

source, target, source\_down, target\_down, source\_fpfh, target\_fpfh = prepare\_dataset(voxel\_size)

:: Load two point clouds and disturb initial pose.  
:: Downsample with a voxel size 0.050.  
:: Estimate normal with search radius 0.100.  
:: Compute FPFH feature with search radius 0.250.  
:: Downsample with a voxel size 0.050.  
:: Estimate normal with search radius 0.100.  
:: Compute FPFH feature with search radius 0.250.



# RANSAC

我们使用RANSAC进行全局配准.在RANSAC迭代中，我们每次从源点云中选取　ransac\_n 个随机点.通过在33维FPFH特征空间中查询最邻近,可以在目标点云中找到他们的对应点.剪枝步骤需要使用快速修剪算法来提早拒绝错误匹配.  
Open3d提供以下剪枝算法:

CorrespondenceCheckerBasedOnDistance检查对应的点云是否接近(也就是距离是否小于指定阈值)

CorrespondenceCheckerBasedOnEdgeLength检查从源点云和目标点云对应中分别画上两条任意边(两个顶点连成的线)是否近似.

CorrespondenceCheckerBasedOnNormal考虑的所有的对应的顶点法线的密切关系.他计算了两个法线向量的点积.使用弧度作为阈值.

只有通过剪枝步骤的匹配才用于转换,该转换将在整个点云上进行验证.核心函数是 ：

registration\_ransac\_based\_on\_feature\_matching.

RANSACConvergenceCriteria是里面一个十分重要的超参数.他定义了RANSAC迭代的最大次数和验证的最大次数.这两个值越大,那么结果越准确,但同时也要花费更多的时间.  
我们是基于[Choi2015]提供的的经验来设置RANSAC的超参数.

def execute\_global\_registration(source\_down, target\_down, source\_fpfh, target\_fpfh, voxel\_size):

distance\_threshold = voxel\_size \* 1.5

print(":: 对下采样的点云进行RANSAC配准.")

print(" 下采样体素的大小为： %.3f," % voxel\_size)

print(" 使用宽松的距离阈值： %.3f." % distance\_threshold)

result = o3d.pipelines.registration.registration\_ransac\_based\_on\_feature\_matching(

source\_down, target\_down, source\_fpfh, target\_fpfh, True, distance\_threshold,

o3d.pipelines.registration.TransformationEstimationPointToPoint(False), 3,

[o3d.pipelines.registration.CorrespondenceCheckerBasedOnEdgeLength(0.9),

o3d.pipelines.registration.CorrespondenceCheckerBasedOnDistance(distance\_threshold)

], o3d.pipelines.registration.RANSACConvergenceCriteria(100000, 0.999))

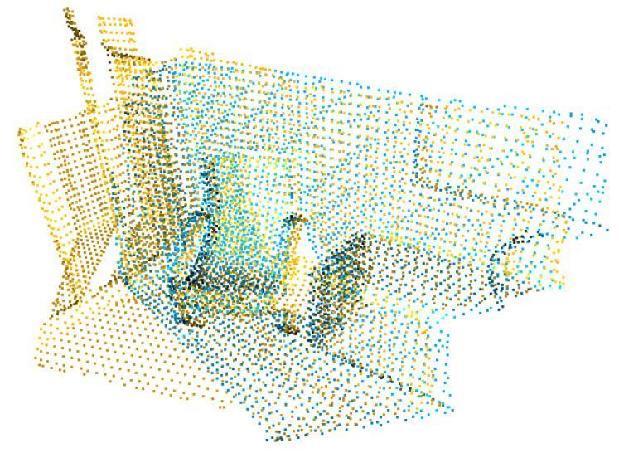
return result

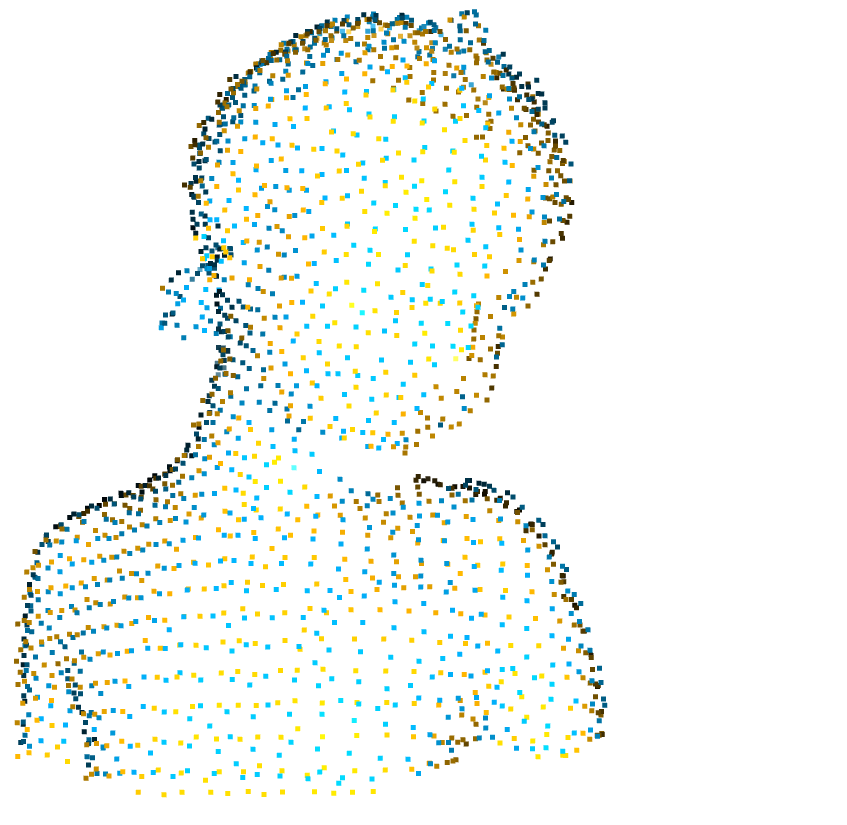
result\_ransac = execute\_global\_registration(source\_down, target\_down, source\_fpfh, target\_fpfh, voxel\_size)

print(result\_ransac)

draw\_registration\_result(source\_down, target\_down, result\_ransac.transformation)

:: RANSAC registration on downsampled point clouds.  
Since the downsampling voxel size is 0.050,  
we use a liberal distance threshold 0.075.  
registration::RegistrationResult with fitness=6.773109e-01, inlier\_rmse=3.332039e-02, and correspondence\_set size of 3224  
Access transformation to get result.





# 局部优化

由于性能原因,全局配准只能在大规模降采样的点云上执行,配准的结果不够精细,我们使用 Point-to-plane ICP 去进一步优化配准结果.

def refine\_registration(source, target, source\_fpfh, target\_fpfh, voxel\_size):

distance\_threshold = voxel\_size \* 0.4

print(":: Point-to-plane ICP registration is applied on original point")

print(" clouds to refine the alignment. This time we use a strict")

print(" distance threshold %.3f." % distance\_threshold)

result = o3d.registration.registration\_icp(

source, target, distance\_threshold, result\_ransac.transformation,

o3d.registration.TransformationEstimationPointToPlane())

return result

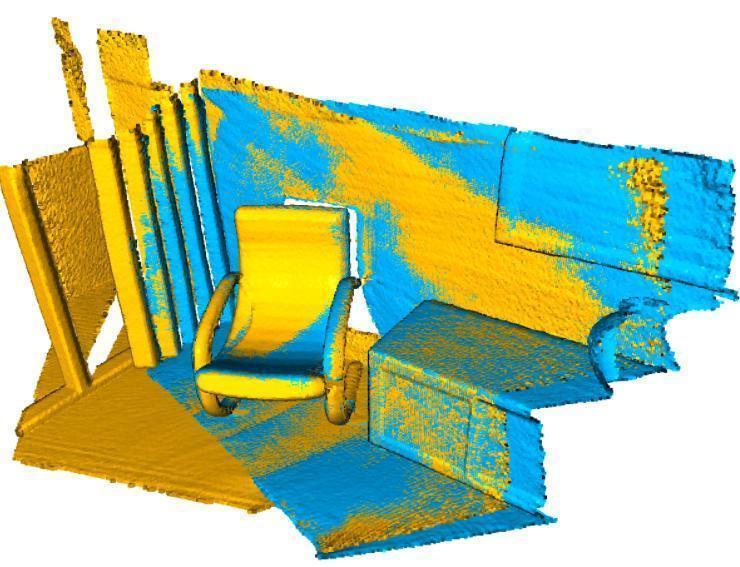
result\_icp = refine\_registration(source, target, source\_fpfh, target\_fpfh,

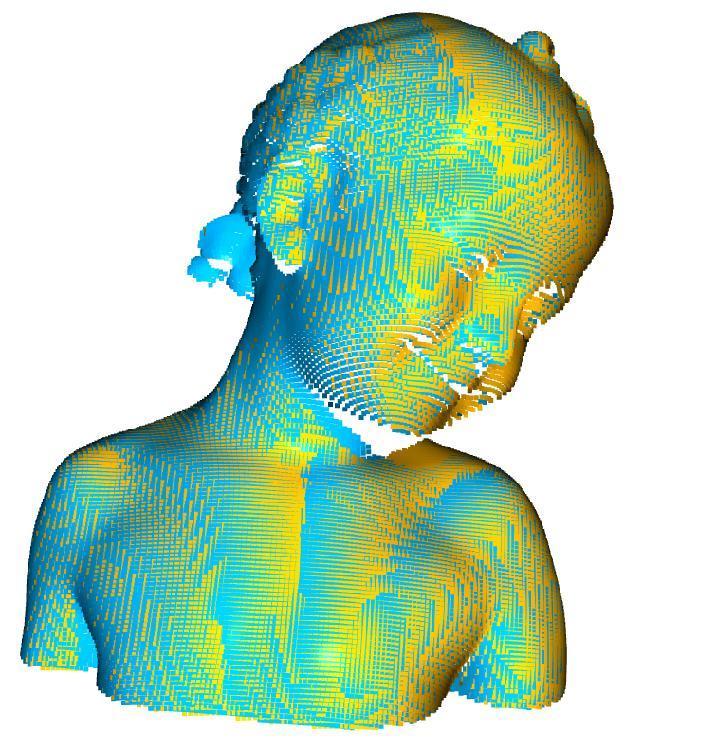
voxel\_size)

print(result\_icp)

draw\_registration\_result(source, target, result\_icp.transformation)

:: Point-to-plane ICP registration is applied on original point  
clouds to refine the alignment. This time we use a strict  
distance threshold 0.020.  
registration::RegistrationResult with fitness=6.210275e-01, inlier\_rmse=6.565175e-03, and correspondence\_set size of 123482  
Access transformation to get result.





# 快速全局配准

由于无数的模型推荐和评估,导致基于RANSAC的全局配准需要很长的时间.  
[Zhou2016] 提出了一种加速的方法,该方法可以快速的优化几乎没有对应关系的线处理权重( [Zhou2016] introduced a faster approach that quickly optimizes line process weights of few correspondences).这样在每次迭代的时候没有模型建议和评估,该方法就在计算的时候节约的大量的时间.(建议看看原论文,这个感觉翻译不好,有更好建议的欢迎留言.)  
这篇教程比较了基于RANSAC的全局配准和[Zhou2016]方法的运行时间.

输入

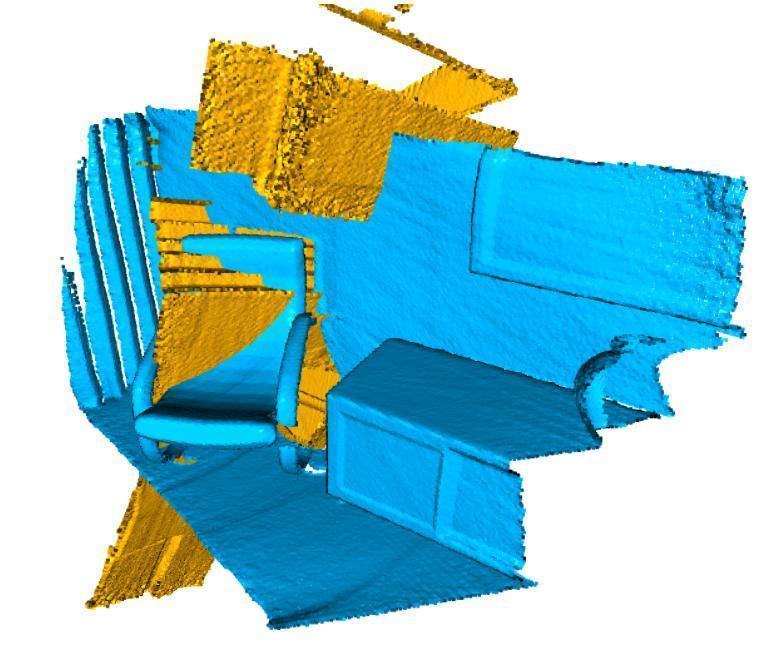
我们使用上面全局配准的输入例子.

voxel\_size = 0.05 # means 5cm for the dataset

source, target, source\_down, target\_down, source\_fpfh, target\_fpfh = \

prepare\_dataset(voxel\_size)

:: Load two point clouds and disturb initial pose.  
:: Downsample with a voxel size 0.050.  
:: Estimate normal with search radius 0.100.  
:: Compute FPFH feature with search radius 0.250.  
:: Downsample with a voxel size 0.050.  
:: Estimate normal with search radius 0.100.  
:: Compute FPFH feature with search radius 0.250.



# **基准**

在下面代码中,我们将计时全局配准算法.

start = time.time()

result\_ransac = execute\_global\_registration(source\_down, target\_down,

source\_fpfh, target\_fpfh,

voxel\_size)

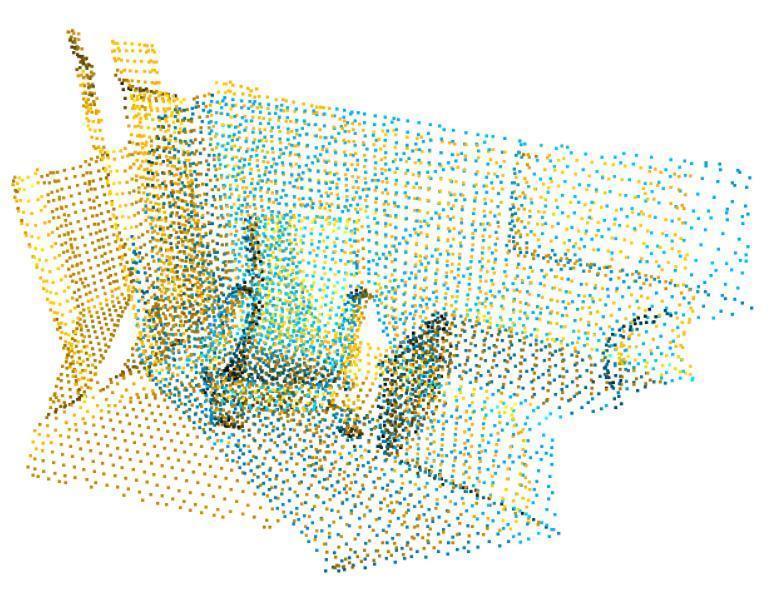
print("Global registration took %.3f sec.\n" % (time.time() - start))

print(result\_ransac)

draw\_registration\_result(source\_down, target\_down,

result\_ransac.transformation)

:: RANSAC registration on downsampled point clouds.  
Since the downsampling voxel size is 0.050,  
we use a liberal distance threshold 0.075.  
Global registration took 0.085 sec.  
registration::RegistrationResult with fitness=6.760504e-01, inlier\_rmse=2.596653e-02, and correspondence\_set size of 3218  
Access transformation to get result.



# 快速全局配准

我们采用和基准相同的输入,下面的代码调用了了[Zhou2016]的实现.

def execute\_fast\_global\_registration(source\_down, target\_down, source\_fpfh,

target\_fpfh, voxel\_size):

distance\_threshold = voxel\_size \* 0.5

print(":: Apply fast global registration with distance threshold %.3f" \

% distance\_threshold)

result = o3d.registration.registration\_fast\_based\_on\_feature\_matching(

source\_down, target\_down, source\_fpfh, target\_fpfh,

o3d.registration.FastGlobalRegistrationOption(

maximum\_correspondence\_distance=distance\_threshold))

    return result

start = time.time()

result\_fast = execute\_fast\_global\_registration(source\_down, target\_down,

source\_fpfh, target\_fpfh,

voxel\_size)

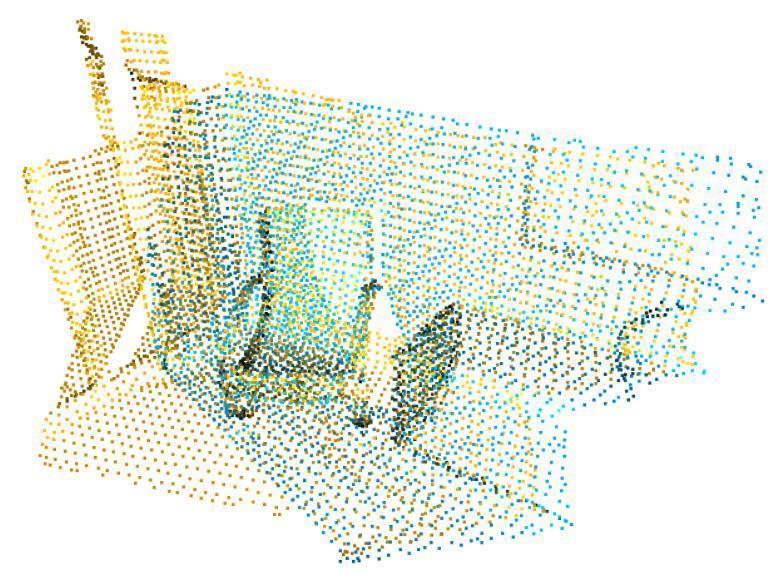
print("Fast global registration took %.3f sec.\n" % (time.time() - start))

print(result\_fast)

draw\_registration\_result(source\_down, target\_down,

result\_fast.transformation)

:: Apply fast global registration with distance threshold 0.025  
Fast global registration took 0.128 sec.  
registration::RegistrationResult with fitness=5.054622e-01, inlier\_rmse=1.743545e-02, and correspondence\_set size of 2406  
Access transformation to get result.



经过适当的配置,快速全局配准的精度甚至可以和ICP相媲美.更多实验结果请参阅[Zhou2016].

# 多视角点云配准

Open3D是一个开源库，支持快速开发和处理3D数据。Open3D在c++和Python中公开了一组精心选择的[数据结构](https://so.csdn.net/so/search?q=%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/u013019296/article/details/_blank)和算法。后端是高度优化的，并且是为并行化而设置的。

多视角配准是在全局空间中对齐多个几何形状的过程。比较有代表性的是，输入是一组几何形状Pi（可以是点云或者RGBD图像）。输出是一组刚性变换Ti，变换后的点云TiPi可以在全局空间中对齐。Open3d通过姿态图估计提供了多视角配准的接口。具体的技术细节请参考[Choi2015].

## 输入

教程代码的第一部分是从三个文件中读取三个点云数据，这三个点云将被降采样和可视化，可以看出他们三个是不对齐的。

def load\_point\_clouds(voxel\_size=0.0):

pcds = []

for i in range(3):

pcd = o3d.io.read\_point\_cloud("../../TestData/ICP/cloud\_bin\_%d.pcd" % i)

pcd\_down = pcd.voxel\_down\_sample(voxel\_size=voxel\_size)

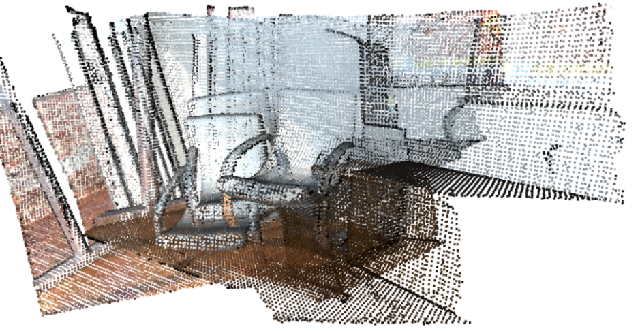
pcds.append(pcd\_down)

return pcds

voxel\_size = 0.02

pcds\_down = load\_point\_clouds(voxel\_size)

o3d.visualization.draw\_geometries(pcds\_down)



## 姿态图

姿态图有两个关键的基础：节点和边。节点是与姿态矩阵i关联的一组几何体Pi，通过该矩阵能够将Pi转换到全局空间。集和Ti是一组待优化的未知的变量。  
 PoseGraph.nodes是PoseGraphNode的列表。我们设P0的空间是全局空间。因此T0是单位矩阵。其他的姿态矩阵通过累加相邻节点之间的变换来初始化。相邻节点通常都有着大规模的重叠并且能够通过Point-to-plane ICP来配准。

姿态图的边连接着两个重叠的节点（几何形状）。每个边都包含着能够将源几何Pi 和目标几何Pj对齐的变换矩阵Ti,j。本教程使用Point-to-plane ICP来估计变换矩阵。在更复杂的情况中，成对的配准问题一般是通过全局配准来解决的。

[Choi2015] 观察到，成对的配准容易出错。甚至错误的匹配会大于正确的匹配，因此，他们将姿态图的边分为两类。Odometry edges连接着邻域节点，使用局部配准的方式比如ICP就可以对齐他们。Loop closure edges连接着非邻域的节点。该对齐是通过不太可靠的全局配准找到的。在Open3d中，这两类边缘通过PoseGraphEdge初始化程序中的uncertain参数来确定。

除了旋转矩阵Ti以外，用户也可以去设置每一条边的信息矩阵Ai。如果是通过  
get\_information\_matrix\_from\_point\_clouds设置的信息矩阵Ai，那么姿态图的边的损失将以 line process weight 近似于两组节点对应点集的RMSE。有关详细细节请参考[Choi2015] 和 the Redwood registration benchmark。

下面的脚本创造了具有三个节点和三个边的姿态图。这些边里，两个是odometry edges（uncertain = False），一个是loop closure edge（uncertain = True）。

def pairwise\_registration(source, target):

print("Apply point-to-plane ICP")

icp\_coarse = o3d.registration.registration\_icp(

source, target, max\_correspondence\_distance\_coarse, np.identity(4),

o3d.registration.TransformationEstimationPointToPlane())

icp\_fine = o3d.registration.registration\_icp(

source, target, max\_correspondence\_distance\_fine,

icp\_coarse.transformation,

o3d.registration.TransformationEstimationPointToPlane())

transformation\_icp = icp\_fine.transformation

information\_icp = o3d.registration.get\_information\_matrix\_from\_point\_clouds(

source, target, max\_correspondence\_distance\_fine,

icp\_fine.transformation)

return transformation\_icp, information\_icp

def full\_registration(pcds, max\_correspondence\_distance\_coarse,

max\_correspondence\_distance\_fine):

pose\_graph = o3d.registration.PoseGraph()

odometry = np.identity(4)

pose\_graph.nodes.append(o3d.registration.PoseGraphNode(odometry))

n\_pcds = len(pcds)

for source\_id in range(n\_pcds):

for target\_id in range(source\_id + 1, n\_pcds):

transformation\_icp, information\_icp = pairwise\_registration(

pcds[source\_id], pcds[target\_id])

print("Build o3d.registration.PoseGraph")

if target\_id == source\_id + 1: # odometry case

odometry = np.dot(transformation\_icp, odometry)

pose\_graph.nodes.append(

o3d.registration.PoseGraphNode(np.linalg.inv(odometry)))

pose\_graph.edges.append(

o3d.registration.PoseGraphEdge(source\_id,

target\_id,

transformation\_icp,

information\_icp,

uncertain=False))

else: # loop closure case

pose\_graph.edges.append(

o3d.registration.PoseGraphEdge(source\_id,

target\_id,

transformation\_icp,

information\_icp,

uncertain=True))

return pose\_graph

print("Full registration ...")

max\_correspondence\_distance\_coarse = voxel\_size \* 15

max\_correspondence\_distance\_fine = voxel\_size \* 1.5

with o3d.utility.VerbosityContextManager(o3d.utility.VerbosityLevel.Debug) as cm:

pose\_graph = full\_registration(pcds\_down,

max\_correspondence\_distance\_coarse,

max\_correspondence\_distance\_fine)

Open3d使用函数global\_optimization进行姿态图估计，可以选择两种类型的优化算法，分别是GlobalOptimizationGaussNewto和GlobalOptimizationLevenbergMarquardt。比较推荐后一种的原因是因为它具有比较好的收敛性。GlobalOptimizationConvergenceCriteria类可以用来设置最大迭代次数和别的优化参数。

GlobalOptimizationOption定于了两个参数。max\_correspondence\_distance定义了对应阈值。edge\_prune\_threshold是修剪异常边缘的阈值。reference\_node是被视为全局空间的节点ID。

print("Optimizing PoseGraph ...")

option = o3d.registration.GlobalOptimizationOption(

max\_correspondence\_distance=max\_correspondence\_distance\_fine,

edge\_prune\_threshold=0.25,

reference\_node=0)

with o3d.utility.VerbosityContextManager(o3d.utility.VerbosityLevel.Debug) as cm:

o3d.registration.global\_optimization(

pose\_graph, o3d.registration.GlobalOptimizationLevenbergMarquardt(),

o3d.registration.GlobalOptimizationConvergenceCriteria(), option)

全局优化在姿态图上执行两次。第一遍将考虑所有边缘的情况优化原始姿态图的姿态，并尽量区分不确定边缘之间的错误对齐。这些错误对齐将会产生小的 line process weights，他们将会在第一遍被剔除。第二遍将会在没有这些边的情况下运行，产生更紧密地全局对齐效果。在这个例子中，所有的边都将被考虑为真实的匹配，所以第二遍将会立即终止。

可视化操作

使用```draw\_geometries``函数可视化变换点云。

print("Transform points and display")

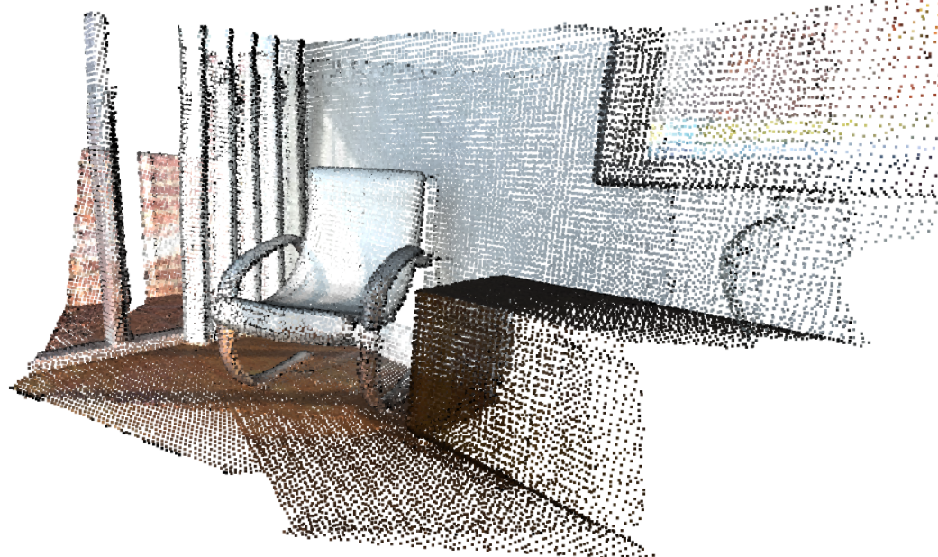
for point\_id in range(len(pcds\_down)):

print(pose\_graph.nodes[point\_id].pose)

pcds\_down[point\_id].transform(pose\_graph.nodes[point\_id].pose)

o3d.visualization.draw\_geometries(pcds\_down)

Transform points and display  
[[ 1.00000000e+00 -2.50509994e-19 0.00000000e+00 0.00000000e+00]  
[-3.35636805e-20 1.00000000e+00 1.08420217e-19 -8.67361738e-19]  
[-1.08420217e-19 -1.08420217e-19 1.00000000e+00 0.00000000e+00]  
[ 0.00000000e+00 0.00000000e+00 0.00000000e+00 1.00000000e+00]]  
[[ 0.8401689 -0.14645453 0.52217554 0.34785474]  
[ 0.00617659 0.96536804 0.2608187 -0.39427149]  
[-0.54228965 -0.2159065 0.81197679 1.7300472 ]  
[ 0. 0. 0. 1. ]]  
[[ 0.96271237 -0.07178412 0.2608293 0.3765243 ]  
[-0.00196124 0.96227508 0.27207136 -0.48956598]  
[-0.27051994 -0.26243801 0.92625334 1.29770817]  
[ 0. 0. 0. 1. ]]



得到合并的点云

PointCloud是可以很方便的使用+来合并两组点云成为一个整体。合并之后，将会使用voxel\_down\_sample进行重新采样。建议在合并之后对点云进行后处理，因为这样可以减少重复的点后者较为密集的点。

pcds = load\_point\_clouds(voxel\_size)

pcd\_combined = o3d.geometry.PointCloud()

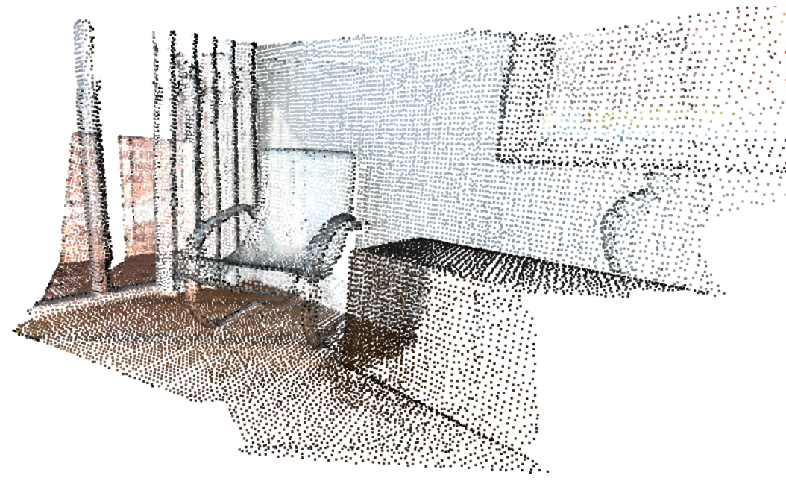
for point\_id in range(len(pcds)):

pcds[point\_id].transform(pose\_graph.nodes[point\_id].pose)

pcd\_combined += pcds[point\_id]

pcd\_combined\_down = pcd\_combined.voxel\_down\_sample(voxel\_size=voxel\_size)

o3d.io.write\_point\_cloud("multiway\_registration.pcd", pcd\_combined\_down)

o3d.visualization.draw\_geometries([pcd\_combined\_down])

尽管这个教程展示的点云的多视角配准，但是相同的处理步骤可以应用于RGBD图像，请参看 Make fragments 示例。

# 参考文献

<https://blog.csdn.net/u014072827/article/details/113788879>

<https://www.freesion.com/article/33541418095/>

<https://www.freesion.com/article/75891338419/>

<http://www.open3d.org/docs/release/tutorial/pipelines/global_registration.html>