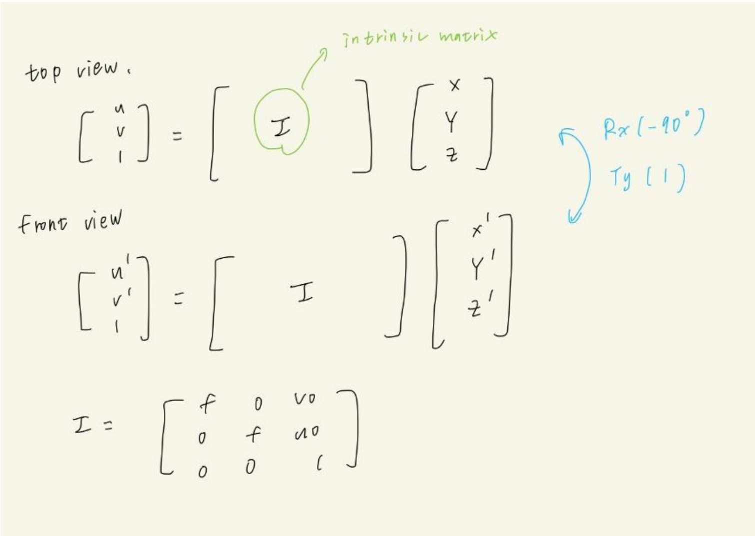
Task1

i. Code

```python
22      def top_to_front(self, theta=0, fov=math.pi/2):
23          """
24              Project the top view pixels to the front view pixels.
25              :return: New pixels on perspective(front) view image
26          """
27
28          ### TODO ###
29          resolution = 512
30          translate = 1
31          theta = theta*math.pi/180   # degree to radian
32          focal_length = float(resolution/2*mpmath.cot(fov/2))
33
34          intrinsic_mat = np.array([[focal_length, 0, resolution/2], [0, focal_length, resolution/2], [0, 0, 1]])
35          trasformation_mat = np.array([[1, 0, 0, 0],
36                                        [0, math.cos(theta), -math.sin(theta), translate],
37                                        [0, math.sin(theta), math.cos(theta), 0],
38                                        [0, 0, 0, 1]])
39
40
41          # print(intrinsic_mat)
42          # print(trasformation_mat)
43          new_pixels = []
44          for i in range(4):
45
46              new_point = np.append(np.array(points[i]).transpose(), 1)                      # homoginious
47              new_point = np.dot(scipy.linalg.inv(intrinsic_mat), new_point)                  # image plane to 3D coordinate
48              new_point = new_point*(-2.5)                                                    # *Z (Z = -2.5)
49
50              new_point = np.append(new_point, 1)                                             # homoginious
51              new_point = np.dot(trasformation_mat, new_point)
52              new_point = np.dot(np.array([[1,0,0,0], [0,1,0,0],[0,0,1,0]]), new_point)
53              new_point = np.dot(intrinsic_mat, new_point)                                    # 3D coordinate to image plane
54              new_pixels.append(np.int32(new_point[0:2]/new_point[2]))
55
56          return new_pixels
```

We can know the relationship between the coordinate of the point in top-view image and its coordinate in front-view image from the following picture.



Thus, there are five major steps to change the coordinate of a point in top-view image into front-view image.

(1) Multiply the point coordinate in top-view image by the inverse matrix of the camera intrinsic matrix -> acquire the 3D coordinate of the object (the camera observed it from the top) (code line 47)

(2) The z value of the result (3D coordinate) we got from (1) is always “1”, which is not the “real” scale of the object. That is because the pin hole camera model rescales the 3D coordinate (divided by its z value).

Fortunately, we have known that we took the top-view image from height of 2.5 unit. We can simply multiply the result of (1) by -2.5 (the camera took pictures toward -z axis, so the object was located at z=-2.5) to rescale the object to its original scale. (code line 48)

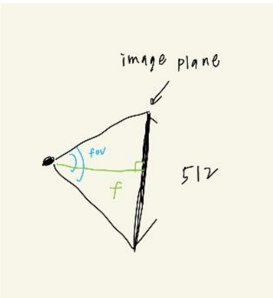
(3) Change the coordinate from “top-view camera coordinate system (TCCS)” to “front-view camera coordinate system (FCCS)”. Both of them are local camera coordinate systems. If we rotate the x-axis of “front-view coordinate system” by -90 degrees, and then translate it up by 1 unit along the y-axis of the world coordinate system, we will find that FCCS now matches TCCS. Thus, the transformation matrix which changes a point from TCCS into FCCS is as shown at code line 35-38. Then I multiply the result of (2) by this transformation matrix to make it be at FCCS (also make it homogeneous). (code line 50-51)

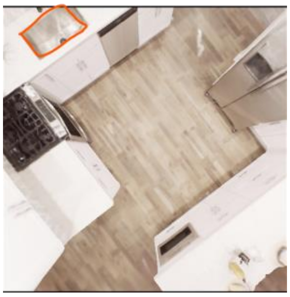
(4) Then, I multiply the result of (3) by the camera intrinsic matrix -> project it onto “front-view image plane” and divide the result by its homogeneous value (the homogeneous value should be “1”). After doing so, we got the coordinate on front-view image plane. (code line 53-54)

As for the focal length of the camera, we have the following relationship:

resolution/2*cot(fov/2)

the difference between the origin of the camera coordinate system and that of the image plane is “resolution/2” for both u and v direction. Thus the camera intrinsic matrix is what shown at code line 34.





ii. Result and Discussion

At the beginning, I tried to select points around the sink (as the above picture) because I thought that it’s a place where I could easily know whether the result is correct or not. However, my result was not correct at all even though I checked my code and thought that it was correct. Then I found that the problem is the “real” height. As what I wrote in (2), we need to multiply the result of (1) by its “real” height. For the point which is located on the ground, its height is -2.5; but for the point which is located around the sink, its height isn’t -2.5 and we have no idea about its “real” height if we didn’t have the depth camera. Thus, I tried to select points on the ground and found that the result got correct.

Task2

i. Code

(1) Unproject depth images:

Because the depth information is scaled into [0,255] in load.py before we stored the depth image, we need to reverse the same scaling process to get the original depth information.

↓ What was done in load.py

```
def transform_depth(image):
    depth_img = (image / 10 * 255).astype(np.uint8)
    return depth_img
```

↓ Reverse the scaling process to unproject the depth image in my reconstruct.py (code line 298)

```python
294             rgb_img = cv2.imread(color_name)     # bgr
295             rgb_img = cv2.cvtColor(rgb_img, cv2.COLOR_BGR2RGB)
296             depth_img = o3d.io.read_image(depth_name)
297             depth_img = np.asarray(depth_img, dtype=np.float32)
298             depth_img = depth_img/255*10
299
```

↓ As what was done in Task 1, multiply the coordinate of each point on the image by the inverse matrix of the camera intrinsic matrix (code line 28-33). Use depth information stored in depth image to scale the 3D coordinates (code line 28-33). Assign the coordinate of points and color of points to the point cloud (code line 45-47).

```python
14  def depth_image_to_point_cloud_by_me(rgb_img, depth_img, intrinsic):
15
16      # rgb_img.shape = (512, 512, 3)
17      # depth_img.shape = (512, 512)
18
19
20      point_num = rgb_img.shape[0] * rgb_img.shape[0]
21      points_xyz = np.zeros((point_num, 3))
22      points_color = np.zeros((point_num, 3))
23      rgbd_img = np.zeros((rgb_img.shape[0], rgb_img.shape[0], 4))
24      rgbd_img[:, :, 0:3] = rgb_img
25      rgbd_img[:, :, 3] = depth_img
26      id = 0
27
28      for u in range(rgb_img.shape[0]):
29          for v in range(rgb_img.shape[1]):
30
31
32              uv = np.array([v, u, 1]).transpose()                    # homogeneous
33              xyz = np.dot(scipy.linalg.inv(intrinsic), uv)           # image plane to 3D coordinate
34              xyz = xyz * depth_img[u, v]
35              points_xyz[id, :] = xyz
36              points_color[id, :] = rgb_img[u, v, :]/255
37
38
39              id = id +1
40
41      # print(np.max(points_xyz[:, 1]))
42      print(points_xyz)
43
44      # Pass xyz to Open3D.o3d.geometry.PointCloud and visualize
45      pcd = o3d.geometry.PointCloud()
46      pcd.points = o3d.utility.Vector3dVector(points_xyz)
47      pcd.colors = o3d.utility.Vector3dVector(points_color)
48      pcd.transform([[1, 0, 0, 0], [0, -1, 0, 0], [0, 0, -1, 0], [0, 0, 0, 1]])
49
50      return rgbd_img, pcd
```

(2) Voxelize the point cloud (code line 87) and apply global registration (code line 102-118) to get an initial transformation matrix which would be refined in the following ICP (local) registration

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

```python
def preprocess_point_cloud(pcd, voxel_size):
    # print(":: Downsample with a voxel size %.3f." % voxel_size)
    pcd_down = pcd.voxel_down_sample(voxel_size)

    radius_normal = voxel_size * 2
    # print(":: Estimate normal with search radius %.3f." % radius_normal)
    pcd_down.estimate_normals(
        o3d.geometry.KDTreeSearchParamHybrid(radius=radius_normal, max_nn=30))

    radius_feature = voxel_size * 5
    # print(":: Compute FPFH feature with search radius %.3f." % 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
```

```python
def execute_global_registration(source_down, target_down, source_fpfh,
                                target_fpfh, voxel_size):
    distance_threshold = voxel_size * 1.5
    # print(":: RANSAC registration on downsampled point clouds.")
    # print("   Since the downsampling voxel size is %.3f," % voxel_size)
    # print("   we use a liberal distance threshold %.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
```

(3) ICP

↓ Open3d ICP (point-to-plane ICP)

Reference:

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

```python
# open3d icp
def refine_registration(source, target, source_fpfh, target_fpfh, voxel_size, initial_transform_mat):
    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.pipelines.registration.registration_icp(
        source, target, distance_threshold, initial_transform_mat,
        o3d.pipelines.registration.TransformationEstimationPointToPlane())
    return result
```

↓ ICP by me (point-to-point ICP)

ICP algorithm implementation:

https://www.796t.com/content/1548524898.html

↓ First, we need to find the corresponding point in the target point cloud (TPC) for each point in the source point cloud (SPC) because the number of target points may not be equal to the number of source points. Also, we have no idea which point in SPC should correspond to which point in

TP C. Thus, I compute the L2 distance between each point in TPC and SPC, and choose the one in TPC which has the minimum L2 distance with the point in SPC as its corresponding point to match in the following process. This function returns the corresponding points of the source points.

```python
def find_nearest_neighbor(source, target):
    # source.shape = (n,3), target.shape = (m,3)
    # print(source.shape, target.shape)
    id = np.zeros((source.shape[0]))                    # shape = (n, ), to store the nearest neighbor id in target for points in source
    dis = np.ones((source.shape[0]))*np.inf      # shape = (n, ), to store the nearest neighbor distance in target for points in source (initialize to infinite)
    correspond_target = np.zeros(source.shape)   # shape = (n, 3)

    for i in range(source.shape[0]):
        for j in range(target.shape[0]):

            if(np.linalg.norm(source[i]-target[j]) < dis[i]):
                id[i] = int(j)
                dis[i] = np.linalg.norm(source[i]-target[j])

        # print(id[i])
        correspond_target[i, :] = target[int(id[i]), :]


    return correspond_target.transpose()
```

↓ After finding points to match, we can compute the rotation and transition matrix with which the source points can transform to points which are closest to the corresponding target points. We first compute the centroid of SPC and TPC. Second, we subtract the centroid from SPC and TPC. Then we do singular value decomposition. What we get from singular value decomposition can be used to compute the best rotation matrix and the best rotation matrix can be used to compute the best transition matrix. At the end, I put them together to form a transformation matrix.

Reference:

The proof of ICP algorithm:

https://zhuanlan.zhihu.com/p/107218828?utm_id=0

Reference code:

https://www.796t.com/content/1548524898.html

```python
def find_best_transform(source, target):
    # source.shape = target.shape = (3, n)
    source_mean = np.mean(source, axis=1)   # shape = (3, )
    target_mean = np.mean(target, axis=1)   # shape = (3, )
    source_prime = source - np.tile(source_mean,(source.shape[1],1)).transpose()
    target_prime = target - np.tile(target_mean,(target.shape[1],1)).transpose()

    u, sigma, vt = np.linalg.svd(np.dot(source_prime, target_prime.transpose()))

    rotation = np.dot(vt.transpose(), u.transpose())
    translate = target_mean - np.dot(rotation, source_mean)
    transform_mat = np.zeros((4, 4))        # homogeneous
    transform_mat[3, 3] = 1
    transform_mat[0:3, 0:3] = rotation
    transform_mat[0:3, 3] = translate

    return transform_mat
```

↓ Thus, the whole ICP process is:

1. Find the corresponding target points for source points (code line 197).
2. Compute the best transformation matrix (code line 198).

3. However, the best transformation matrix may not perfectly make source points match the target points. Thus, I compute the L2 distance between the source points and the target points (code line 204-206). If they aren't close enough, we need to use this new source points (after transformed with the best transformation matrix) to find another transformation matrix which can make it close to the target points. Hence, go back to step 1 or exit from the loop when the number of maximum iterative times has reached or the two point-sets are close enough (code line 209, 210).
4. Remember to multiply every computed transformation matrix (including the initial one) together! (code line 208)

```
191     # error = 0
192     max_iterate = 50
193     result_transform_mat = initial_transform_mat
194 
195     for i in range(max_iterate):
196 
197         correspond_target = find_nearest_neighbor(source.transpose()[:, 0:3], target_points)    # shape = (3, n)
198         transform_mat = find_best_transform(source[0:3, :], correspond_target)                  # homogeneous
199         # print(transform_mat)
200 
201         # update current source
202         source = np.dot(transform_mat, source)
203 
204         error = 0
205         for j in range(source.shape[1]):
206             error = error + np.linalg.norm(source[0:3, j]-correspond_target[:, j])
207 
208         result_transform_mat = np.dot(transform_mat, result_transform_mat)
209         if(error < distance_threshold):
210             break
211 
212     return result_transform_mat
213 
```

(4) We let the point cloud at Ti to be the source point cloud of ICP and let point cloud at Ti-1 to be the target point cloud of ICP. After ICP, we can acquire the transformation matrix which can transform the point cloud from Ti's coordinate system into Ti-1's coordinate system. If we do so for every "i" (i < the number of image data), then we would have transformation matrix which can transform point cloud from T1 to T0, T2 to T1, T3 to T2 and so on. We just need to multiply them together (code line 351, 361), and then we can transform point cloud from every Ti's coordinate system into world coordinate system (T0's coordinate system) (code line 366-371, also combine the point clouds).

```
        # icp by o3d
        result_ransac = execute_global_registration(source_down, target_down,
                                                    source_fpfh, target_fpfh,
                                                    voxel_size)

        result_icp_by_o3d = refine_registration(source_down, target_down, source_fpfh, target_fpfh,
                                                voxel_size, result_ransac.transformation)
        transform_mats_by_o3d.append(np.dot(transform_mats_by_o3d[i-1], result_icp_by_o3d.transformation))


        # icp by me
        result_ransac2 = execute_global_registration(source_down2, target_down2,
                                                     source_fpfh2, target_fpfh2,
                                                     voxel_size2)

        result_icp_by_me = refine_registration_by_me(source_down2, target_down2, result_ransac2.transformation, voxel_size2*0.4)

        transform_mats_by_me.append(np.dot(transform_mats_by_me[i-1], result_icp_by_me))



    # icp by o3d
    pcd_combined1 = pcds[0]
    pcd_combined2 = pcds[0]
    for i in range(1, len(pcds)):
        print('combine: '+str(i))
        pcd_combined1 = pcd_combined1 + copy.deepcopy(pcds[i]).transform(transform_mats_by_o3d[i])
        pcd_combined2 = pcd_combined2 + copy.deepcopy(pcds2[i]).transform(transform_mats_by_me[i])
```

(5) Trajectory visualization:

I stored the camera position and quaternion data in a ".csv" file (code line 227-241, read file).

↓ The first row is the camera position (the first three numbers) and quaternion (the others) data in T0; The second row is the camera position (the first three numbers) and quaternion (the others) data in T2…

```
0.0,0.12523484,-0.25,1.0,0.0,0.0,0.0
0.0,0.12523484,-0.25,0.9961947202682495,0.0,-0.08715573698282242,0.0
0.0,0.12523484,-0.25,0.9848077893257141,0.0,-0.1736481636762619,0.0
0.08550503,0.12523484,-0.48492315,0.9848077893257141,0.0,-0.1736481636762619,0.0
0.08550503,0.12523484,-0.48492315,0.9659258723258972,0.0,-0.258819043636322,0.0
0.08550503,0.12523484,-0.48492315,0.9396926760673523,0.0,-0.3420201539993286,0.0
0.24620196,0.12523484,-0.6764343,0.9396926760673523,0.0,-0.3420201539993286,0.0
0.4068989,0.12523484,-0.8679454,0.9396926760673523,0.0,-0.3420201539993286,0.0
0.4068989,0.12523484,-0.8679454,0.9063078761100769,0.0,-0.4226182699203491,0.0
```

I define T0's coordinate system as the world coordinate system. We can simply subtract T0's camera position from Ti's camera position to know the ground truth camera position in world coordinate system (code line 240).

```
222     # read file to get ground truth
223     data_num = 159
224     ground_truth_points = np.zeros((data_num, 3))
225     ground_truth_lines = create_trajectory_line(data_num)
226     row_id = 0
227     with open('./task2_data/camera_pose.csv', newline='') as csvfile:
228         rows = csv.reader(csvfile)
229 
230         for row in rows:
231             if(row_id >= data_num):
232                 break
233             x = float(row[0])
234             y = float(row[1])
235             z = float(row[2])
236 
237             if(row_id==0):
238                 origin = np.array([x, y, z])
239 
240             ground_truth_points[row_id, :] = np.array([x, y, z]) - origin
241             row_id = row_id + 1
242 
243     ground_truth_line_set = o3d.geometry.LineSet(
244         points=o3d.utility.Vector3dVector(ground_truth_points),
245         lines=o3d.utility.Vector2iVector(ground_truth_lines),
246     )
```

↓ Create the line set. The first point connects to the second point; The second point connects to the third point…

Then I use a built-in function in open3d to make points and lines as a LineSet so that I can draw them with the point cloud later (code line 214-218).

```
214 def create_trajectory_line(data_num):
215     lines = np.zeros((data_num-1, 2), dtype=int)
216     for i in range(data_num-1):
217         lines[i, :] = [i, i+1]
218     return lines
219
```

As for the estimated camera pose (estimated by ICP), the transition part in the transformation matrix from Ti into T0 coordinate system is the estimated camera pose of Ti (code line 382, 383).

```
377     colors = [[1, 0, 0] for i in range(len(ground_truth_lines))]
378     icp_by_o3d_points = np.zeros((data_num, 3))
379     icp_by_me_points = np.zeros((data_num, 3))
380
381     for i in range(data_num):
382         icp_by_o3d_points[i, :] = transform_mats_by_o3d[i][0:3,3]
383         icp_by_me_points[i, :] = transform_mats_by_me[i][0:3,3]
384
385
386
387     icp_by_o3d_line_set = o3d.geometry.LineSet(
388         points=o3d.utility.Vector3dVector(icp_by_o3d_points),
389         lines=o3d.utility.Vector2iVector(ground_truth_lines),
390     )
391     icp_by_o3d_line_set.colors = o3d.utility.Vector3dVector(colors)
392
393     icp_by_me_line_set = o3d.geometry.LineSet(
394         points=o3d.utility.Vector3dVector(icp_by_me_points),
395         lines=o3d.utility.Vector2iVector(ground_truth_lines),
396     )
397     icp_by_me_line_set.colors = o3d.utility.Vector3dVector(colors)
398
```

(6) Compute the average L2 distance between the estimated and ground truth camera pose.

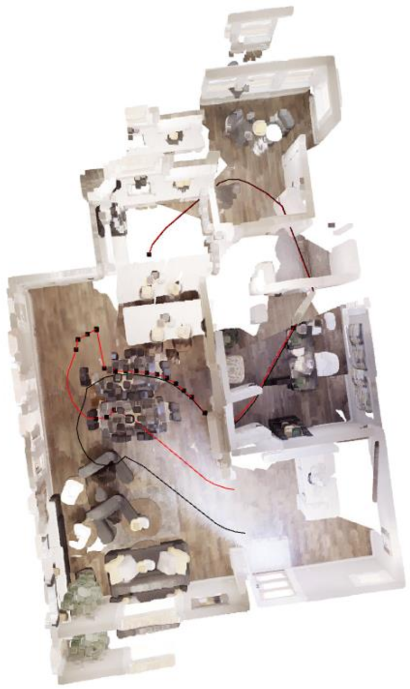
```
    icp_by_o3d_L2_dis = np.mean(np.linalg.norm(icp_by_o3d_points - ground_truth_points, axis=1))
    icp_by_me_L2_dis = np.mean(np.linalg.norm(icp_by_me_points - ground_truth_points, axis=1))

    print('L2 distance: ')
    print('ICP by o3d: ', icp_by_o3d_L2_dis)
    print('ICP by me: ', icp_by_me_L2_dis)
```

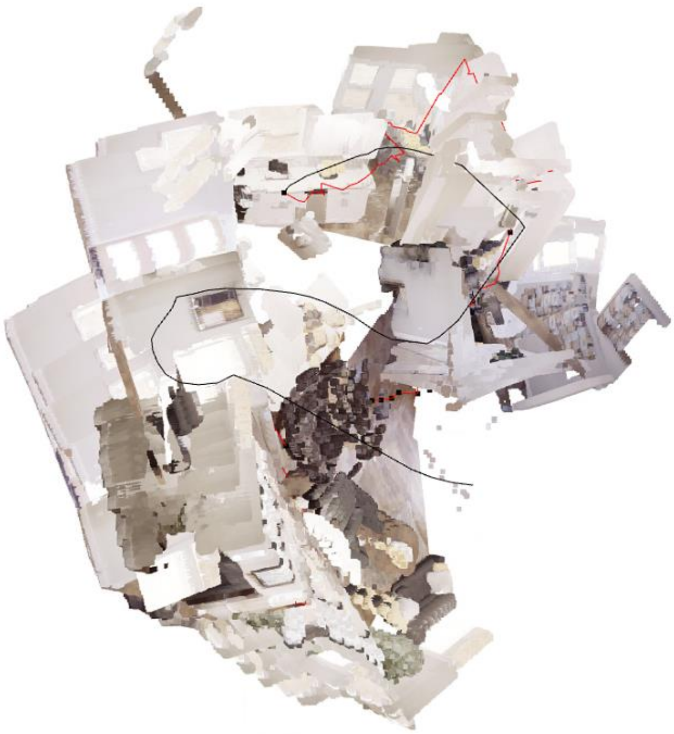
ii. Result and Discussion

Floor 1:

↓ Open3d result



↓ My result

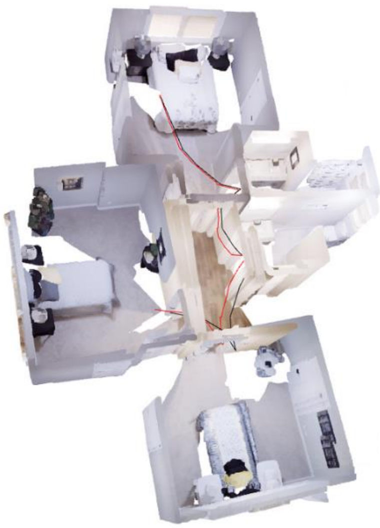


L2 distance:

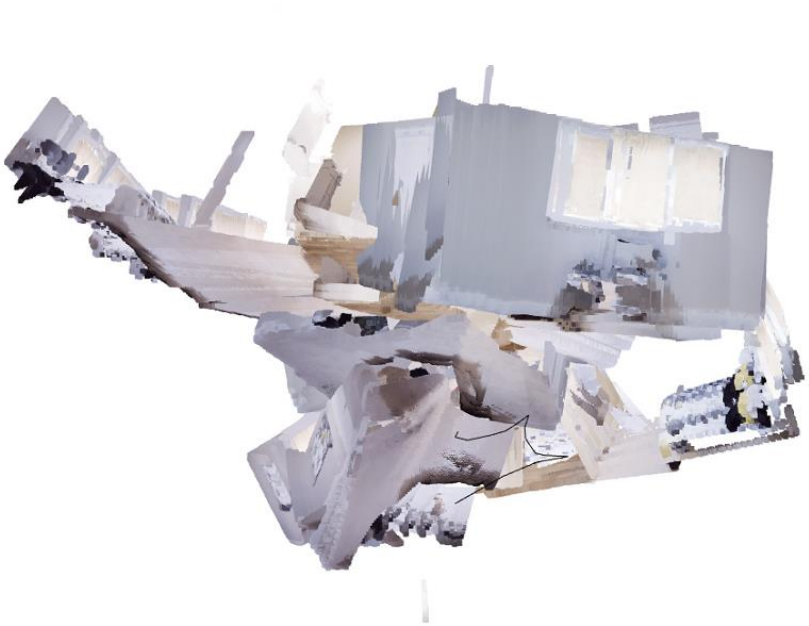
```
ICP by o3d:  0.4899746949113337
ICP by me:  4.1642946653754676
```

Floor 2:

↓ Open3d result



↓ My result



L2 distance:

```
L2 distance:
ICP by o3d:  0.10945650643382078
ICP by me:  3.096445763273858
```

The open3d result is much better than mine. In my ICP implementation, I calculate the L2 distance for every source and every target and let the pair with minimum L2 distance as the corresponding source-target pair. However, some source points may be outliers and cannot find a corresponding target point which is super close to them (as long as the source-target pair has minimum L2 distance compared to other pairs, it's also possible that two different source points correspond to the same target point). In my implementation, although I removed some statistical source outliers before I down-sampled the point cloud, the number of removed outliers is limited. Those points which weren't removed from the previous step may still have a larger L2 distance and I forced them to make a source-target pair in the ICP process. Those pairs having larger L2 distance may influence the best transformation matrix.

```
        pcd2, ind = pcd.remove_statistical_outlier(nb_neighbors=20,
                                                    std_ratio=2)

```

I think another major reason of my worse result is the voxel size used in downsampling the point cloud. We know that ICP algorithm iterate lots of times to find the best transformation matrix, so it is needed to keep the number of the points not so large in order that ICP could be run in a reasonable time. Thus, I set the voxel size to be 0.4, which leads to about 150~250 points to be processed in ICP. However, the voxel size I set for Open3d version of ICP is 0.1, which would lead to about 2000 points in the point cloud (this number is impossible for my ICP to run), so maybe a better transformation matrix could be acquired because an accurate object surface could be represented by more points.

Also, I noticed that what Open3d implements is point-to-plane ICP, but my implementation is point-to-point ICP. I'm not sure how much the impact is, but this may be also a potential reason of my worse result.

In addition, data collection is an important factor that influences the reconstruction result. At the beginning, I thought that the data should be as detailed as possible, so I tried to record the scene from different perspective as much as possible. However, I found that the reconstruction result which reconstructed by using only one photo had been good enough, so maybe using a lot of data just increase the probability of the error happening during the process of ICP. Then I tried not to record the scene with so many photos, but still have enough detail (I think). I found the result a little bit better.