Perception and Decision Making in Intelligent Systems

Homework 1 Report

312553038 楊奕儒

1. Implementation

Task1 :

A. First, based on the homework document, I set the parameter of the two matrices. Next, I converted each point from image coordinates to camera coordinates by applying the inverse of the intrinsic matrix and the 'y' value of the BEV camera location. Following this step, I altered the coordinates from BEV to a top view by using matrix multiplication with the extrinsic matrix. Finally, I transferred the points from camera coordinates back to image coordinates by multiplying them with the intrinsic matrix.

```
for i in range(4):
    current_point = [points[i][0], points[i][1], 1]
    #image cor to camera cor
    c2_camera_point = 2.5 * np. linalg. inv(intrinsic_matrix) @ current_point
    print(c2_camera_point)
    c2_camera_point = np.append(c2_camera_point, [1])

    #change c2 cor to c1 cor
    c1_camera_point = extrinsic_matrix@c2_camera_point
    #camera cor to image cor
    new_point = homo_intrinsic_matrix @ c1_camera_point
    u = int(new_point[0]/new_point[2])
    v = int(new_point[1]/ new_point[2])
    new_pixels.append([u, v])
return new_pixels
```

B.

Result :

I discovered that the order in which points are selected can potentially impact the results. If I don't choose the corresponding points in a circular manner, I won't achieve the correct projection.

Task2 :

A. To generate a point cloud from RGB and depth images, I utilize functions provided by the Open3D library. Initially, I obtain an RGBD image using Open3D's dedicated function. Then, I leverage this RGBD image and the intrinsic matrix as parameters for another Open3D function, which allows me to transform this data into a point cloud. Finally, I adjust this point cloud to a front-view perspective and restrict its y-values to remove ceiling.

```python
def depth_image_to_point_cloud(rgb, depth):
    # TODO: Get point cloud from rgb and depth image

    rgbd_image = o3d.geometry.RGBDImage.create_from_color_and_depth(
        rgb, depth, depth_scale=1000, convert_rgb_to_intensity = False
    )

    intrinsic = o3d.camera.PinholeCameraIntrinsic(
        width=512,
        height=512,
        fx=256.0,
        fy=256.0,
        cx=512 / 2,
        cy=512 / 2
    )

    pcd = o3d.geometry.PointCloud.create_from_rgbd_image(rgbd_image, intrinsic)
    pcd.transform([[1, 0, 0, 0], [0, -1, 0, 0], [0, 0, -1, 0], [0, 0, 0, 1]])
    pcd = no_ceiling(pcd)
    return pcd
```

After I transfer all image to point cloud, I did voxelization to reduce the number of points by applying voxel_down_sample function to get pcd_down, compute the normals for each point by KDTreeSearchParamHybrid, finally use it as a parameter to get the point cloud feature pcd_fpfh.

```python
def preprocess_point_cloud(pcd, voxel_size ):
    # TODO: Do voxelization to reduce the number of points for less memory usage and speedup
    pcd_down = pcd.voxel_down_sample(voxel_size)

    radius_normal = voxel_size * 2
    pcd_down.estimate_normals(o3d.geometry.KDTreeSearchParamHybrid(radius=radius_normal, max_nn=30))
    radius_feature = voxel_size * 5
    pcd_fpfh = o3d.pipelines.registration.compute_fpfh_feature(
                                                pcd_down,
                                                o3d.geometry.KDTreeSearchParamHybrid(radius=radius_feature, max_nn=100)
                                                )
    return pcd_down, pcd_fpfh
```

In the process of aligning point clouds, I employ a two-step approach to enhance precision. First, I execute the "execute_global_registration" to establish an initial alignment, and then I apply the "local_icp_algorithm" for fine-tuning.

In the global registration step, I utilize the "registration_ransac_based_on_feature_matching" function from the Open3D library to compute a global transformation. This transformation helps bring the point clouds into an initial alignment.

For local registration, I utilize the "PointToPlane" method to derive a local transformation. This step refines the alignment by considering the point-to-plane correspondence, which is especially effective in achieving fine adjustments.

After obtaining these two transformation matrices, I can align each subsequent point cloud to the original one. I set the initial transformation matrix as an identity matrix. In each iteration, I update this matrix using both the global and local transformations to align each new point cloud with the initial reference point cloud. This iterative refinement process ensures that the point clouds are accurately aligned. Additionally, I use the final transformation matrix to estimate the camera pose in each point cloud, providing valuable information for camera localization.

```python
def execute_global_registration(source_down, target_down, source_fpfh,
                                target_fpfh, voxel_size):
    distance_threshold = voxel_size * 1.5
    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))
    global_trans = result.transformation
    return global_trans
```

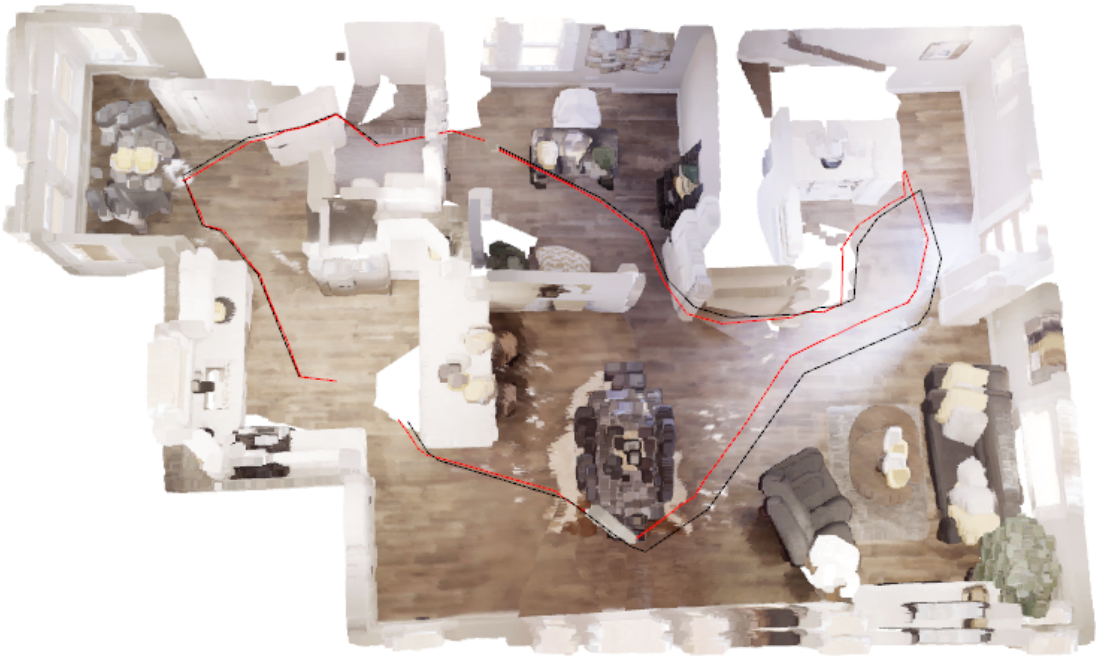
```python
def local_icp_algorithm(source_down, target_down, global_trans , voxel_size ):
    # TODO: Use Open3D ICP function to implement
    distance_threshold = voxel_size * 0.4

    result = o3d.pipelines.registration.registration_icp(
        source_down, target_down, distance_threshold, global_trans,
        o3d.pipelines.registration.TransformationEstimationPointToPlane(),
    )
    trans = result.transformation
    return trans
```

```python
trans = trans@local_trans
target.transform(trans)
aligned_pcd_list.append(target)

#predict the camera pose
pre_cam_pos = np.array(trans[:3, 3])
pred_cam_poses.append(pre_cam_pos)
```

B. Result:
used open3D library for local_registration:

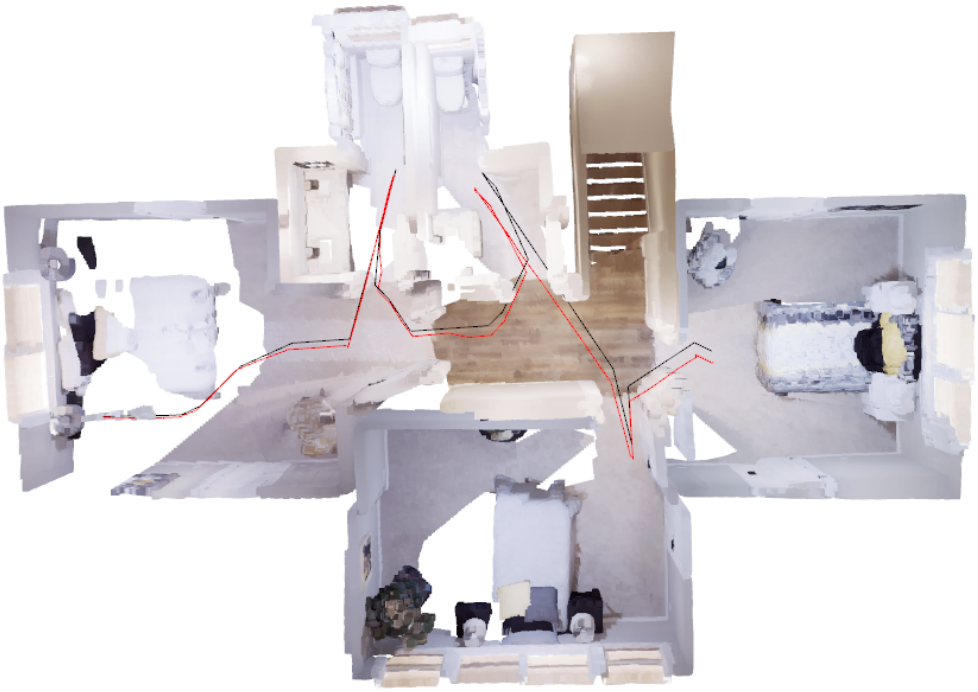


```
Mean L2 distance: 1.6082992217181 93e-05
```

used my_local function for local_registration:

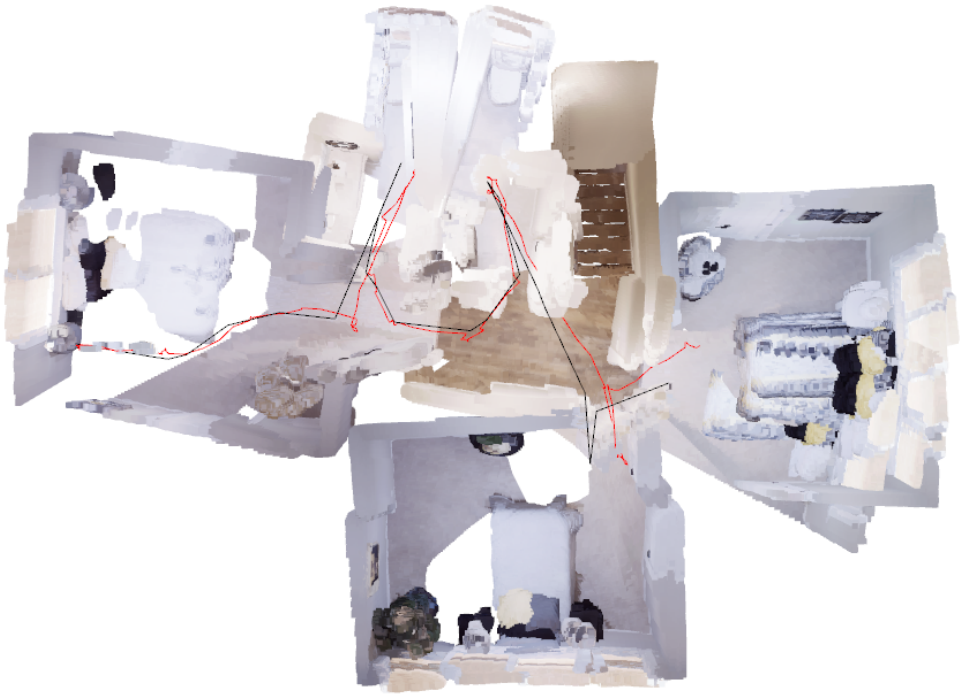
```
Mean L2 distance: 6.292020995914264e-05
```

used open3D library for local_registration:



```
Mean L2 distance: 1.3437926813475835e-05
```

used my_icp for local_registration:



```
Mean L2 distance: 7.216358717484213e-05
```

C . From the graph, we can observe that the performance of using my_icp as the local registration function is worse compared to the results obtained using open3D. Additionally, I also compared the performance of doing only global alignment and incorporating my_icp. Surprisingly, the performance was worse when my_icp was added. I believe this is related to how I select points to align. If I choose too many points that do not simultaneously exist in both point clouds for alignment, it may lead to poorer alignment results.

D. Reference:

https://blog.csdn.net/weixin_36219957/category_10038692.html

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

2. Questions

a. The extrinsic matrix is a transformation matrix from the world coordinate system to the camera coordinate system, and the parameters of it depend on location and orientation of the camera.
The intrinsic matrix is a transformation matrix that converts points from the camera coordinate system to the pixel coordinate system. Parameters such as focal length, aperture, field-of-view, resolution, govern the intrinsic matrix of a camera model.

b. If I do ICP alignment without global registration, the alignment result becomes worse. In my opinion, maybe it is because ICP is highly sensitive to the initial transformation, if skip global registration, ICP relies solely on an initial guess for the alignment, which may not be accurate.

c. I created KD-tree for the target point cloud to find nearest neighbors and distances from source to target. When the distances between corresponding points exceed the half of mean_distance across all point pairs, I employ a filtering process to exclude them. This step is essential as it helps identify potential disparities between the two point clouds. Following this filtering, I proceed to calculate the transformation matrix, which deriving both the rotation matrix and the translation matrix. These two components are subsequently combined to yield the final transformation.

```
mean_distance = np.mean(distances)

# Filter correspondences based on the mean distance
cut_correspondences = [correspondences[i] for i in range(len(correspondences)) if distances[i] <= mean_distance/2]
filtered_source = [source[i] for i in range(len(correspondences)) if distances[i] <= mean_distance/2]
filtered_source = np.array(filtered_source)
```