Perception and Decision Making

in Intelligent Systems Homework 1 Report

Student ID:312512061

Name: 劉郁昇

1. Implementation

Task 1:

a. Code

The essence of task 1 is to take two-dimensional points and expand them to three dimensions, and then project them back to two dimensions through a series of matrix operations. This process enables us to map the Bird's Eye View (BEV) image to the front view image. There are two places where we need to write or make modifications, namely “main” and function “top to front”.

In function “top to front”, I will explain this process through five matrix operations.

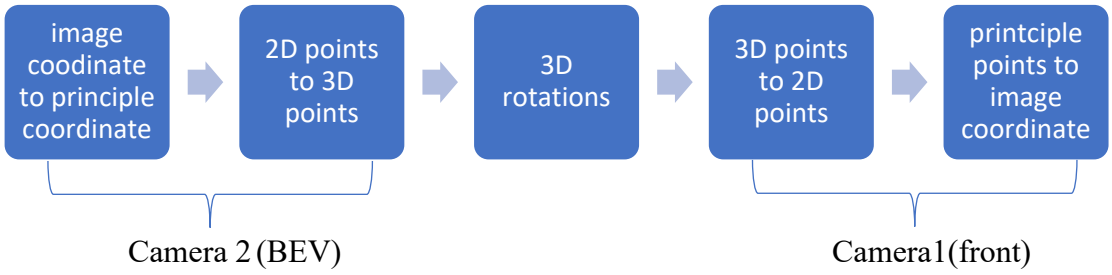
I. Main:

```python
if __name__ == "__main__":

    parser = argparse.ArgumentParser()
    parser.add_argument('-f', '--floor', type=int, default=1)
    args = parser.parse_args()
    if args.floor == 1:
        floor =1
    elif args.floor == 2:
        floor =2

    pitch_ang = 90
    dy=1.5

    front_rgb = "bev_data/front"+str(floor)+".png"
    top_rgb = "bev_data/bev"+str(floor)+".png"

    # click the pixels on window
    img = cv2.imread(top_rgb, 1)
    cv2.imshow('image', img)
    cv2.setMouseCallback('image', click_event)
    cv2.waitKey(0)


    projection = Projection(front_rgb, points)
    new_pixels = projection.top_to_front(theta=pitch_ang,dy=1.5)
    projection.show_image(new_pixels)
    cv2.destroyAllWindows()
```

Fig1.main code

Using the ArgumentParser() to let user choose the picture they want to use. Pitch_ang is 90 degree and dy is 15 due to the different between camera1 and camera 2

II. def(top_to_front)



1. Image coordinate to principle coordinate

```
'''image Coordinate'''
principle_point = [self.height/2,self.width/2]
convention_matrix = np.eye(3)
focal = -1
convention_matrix = np.array([[focal ,0 ,principle_point[0]],
                              [0 ,focal ,principle_point[1]],
                              [0 ,0 , 1]])
print("Convention Matrix")
print(convention_matrix)
```

Fig2. Image coordinate to principle coordinate code

In this matrix, we create a 3x3 matrix based on the concept shown in Figure 3. For the images we used, the principal point (256, 256) represents half the width and height, and the focal length is set to -1. During the matrix operations, I initially set z to 1, which is shown in Figure 4, since we are working with a three-dimensional matrix.

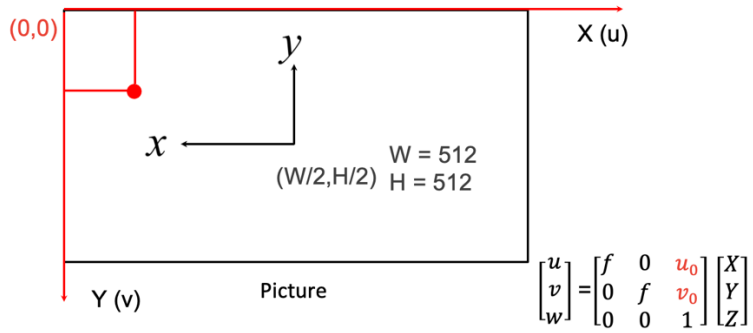


Fig3. image coordinate concept

```
'''Convert to principle coordinate '''
pointsz1 = points
for point in pointsz1:
    point.append(1) #from (x,y) to (x,y,1)
    bev = np.dot(convention_matrix,np.array(point))
    bev_pixel.append(bev)
print("bev_pixel")
print(np.array(bev_pixel))
```

Fig4. implementation code

2. 2D points to 3D points

Based on the concept shown in Figure 6, we can determine the relationship between 2D coordinates and 3D coordinates as long as we have the 'Z' value. Given that the Bird's Eye View (BEV) position is (0, 2.5, 0), we can deduce that the 'Z' value is 2.5.

The following is the derivation of the formula (based on Figure 6)

1. 2D to 3 D matrix

$$\begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

$$\rightarrow w' \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = w' \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & f \end{bmatrix}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

2. Focal length

Focal length = half height of image / tan(FOV/2)

```
'''2d to 3d'''
#here is the hardest part
w_p = 2.5
focal_length = (self.height/2)/np.tan(np.deg2rad(fov/2))
k_matrix = np.eye(3)
k_matrix = np.array([focal_length ,0 ,0],
                    [0 ,focal_length , 0],
                    [0 ,0 , 1])
k_matrix_inverse = np.linalg.inv(k_matrix)

for bev in bev_pixel:
    bev_point.append(w_p*np.dot(k_matrix_inverse,bev))
print("bev_point")
print(np.array(bev_point))
```

Fig5. 2D points to 3D points code

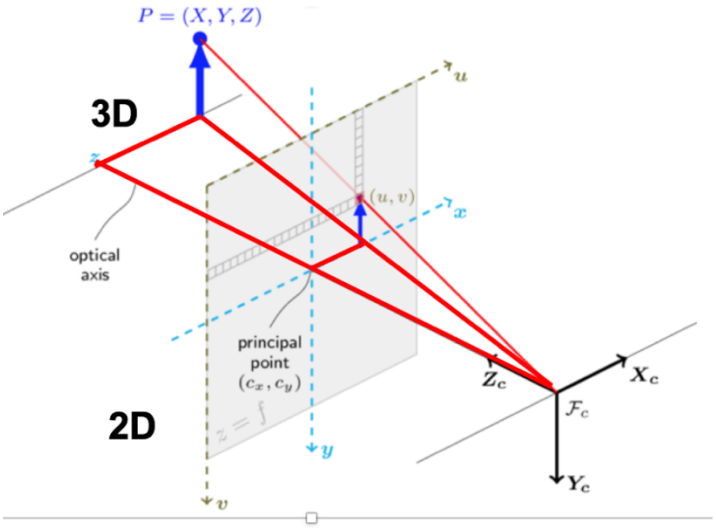


Fig6. Perspective projection

3. 3D rotations

In this section, our objective is to transform the camera2 (BEV) to camera1 (front). A 3D rotation occurs around an axis, and we have chosen to rotate along the XYZ axis.

Below is the derivation of formula

$$R = R_z(\alpha)\, R_y(\beta)\, R_x(\gamma) = \overset{\text{yaw}}{\begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}} \overset{\text{pitch}}{\begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}} \overset{\text{roll}}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix}}$$

$$\begin{bmatrix} {}^0x \\ {}^0y \\ {}^0z \\ 1 \end{bmatrix} = \begin{bmatrix} & {}^0R_1 & & t_x \\ & & & t_y \\ & & & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^1x \\ {}^1y \\ {}^1z \\ 1 \end{bmatrix}$$

Shown in Figure7, the code just implements the function above.

When implementing the matrix transformation, we first need to convert (x, y, z) into (x, y, z, 1). After applying the rotation, we then revert (x, y, z, 1) back to (x, y, z).

```
'''3D rotation'''
# base on the lecture3 p21
yaw = np.pi*((gamma)/180) # around z axis
pitch = np.pi*((phi)/180) # around y axis
roll = np.pi*(theta/180)  # around x axis

rotation_yaw = np.array([[np.cos(yaw),-np.sin(yaw),0],[np.sin(yaw),np.cos(yaw),0],[0,0,1]])
rotation_pitch = np.array([[np.cos(pitch),0,np.sin(pitch)],[0,1,0],[-np.sin(pitch),0,np.cos(pitch)]])
rotation_roll = np.array([[1,0,0],[0,np.cos(roll),-np.sin(roll)],[0,np.sin(roll),np.cos(roll)]])
rotation = np.dot((np.dot(rotation_yaw,rotation_pitch)),rotation_roll)

transformation_matrix = np.eye(4)
transformation_matrix[:3,:3] = rotation
transformation_matrix[:3,3] = [dx,dy,dz]
print("transformation")
print(transformation_matrix)
```

Fig7. Rotation Matrix

```
'''Do rotate'''
# we first augment a column of 1
column_of_ones = np.ones((np.shape(bev_point)[0],1))
bev_point = np.column_stack((bev_point,column_of_ones))

for bev in bev_point:
    front_point.append(np.dot(transformation_matrix,bev))

front_point = np.delete(front_point,-1, axis=-1)
print("front_point")
print(np.array(front_point))
```

Fig8. Do rotation

4. 3D point to 2D points

   As we discussed in part 2 (2D points to 3D points), we already have the 3x3 matrix relating 2D and 3D coordinates.

   As illustrated below, (X, Y, Z) represents our 3D coordinates, and (u, v) denotes our 2D coordinates.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} 1/w'$$

```
'''3d to 2d '''
for front in front_point:
    front_pixel.append(np.dot(k_matrix,front)/front[2])

print("front_pixel")
print(np.array(front_pixel))
```

Fig9. 3D to 2D

5. Principle points to image coordinate

   As we discussed in part 1, we already have the 3x3 matrix relating to principle coordinate and pixel coordinate.

   The implementation is shown in Figure 10. We need to exclude the Z-axis from our data since Z is irrelevant and not required for our purposes.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} -1 & 0 & 256 \\ 0 & -1 & 256 \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

```
'''from principle coordinate to pixel cooridate'''
convention_matrix_inverse = np.linalg.inv(convention_matrix)
for front in  front_pixel:
    new_pixel_x,new_pixel_y ,new_pixel_z= np.dot(convention_matrix_inverse,front)
    new_pixels.append([int(new_pixel_x),int(new_pixel_y),int(new_pixel_z)]) # the data type of new

new_pixels = np.delete(new_pixels,-1, axis=-1)
print("new_pixels")
print(new_pixels)
```

Fig10. Principle points to image coordinate

b. Result and Discussion

I. Result of projection

There are two pairs of projection. The figure11,12,13,14 show the result.

1. floor 1

Fig11. BEV select points

Fig12. Front Projection

2. floor 2

Fig13. BEV select points

Fig14. Front Projection

II. Discussion

As I approached the completion of task 1, I encountered an error message consistently. Despite a series of checks, I couldn't identify the cause. In the end, I found that the issue was related to the requirement for pixel values to be positive. In other words, the return value of the "top_to_front" function needs to be integers since pixels must have integer values.

III. Reference

https://medium.com/maochinn/筆記-camera-dee562610e71
https://flyhead.medium.com/為什麼需要齊次座標-homogeneous-coordinate-bd86356f67b1
https://medium.com/maochinn/筆記-camera-dee562610e71

Task 2:

a. Code

This code performs the processing and reconstruction of point cloud data and visualizes the processed point cloud, real trajectory, and camera trajectory. I will explain the functionality of each function and their interrelationships.

- 'depth_image_to_point_cloud' function:

Shown in figure15. This function computes a 3D point cloud from a depth image (depth) and a color image (rgb). It first defines the camera's intrinsic parameters such as field of view (fov), focal length, and image center (cx, cy). The parameters is same as we have mentioned in Task1. We merge the function "Image coordinate to principle coordinate" and "2D points to 3D points " as one 3X3 matrix. Then, it transforms 2D image coordinates into 3D space coordinates based on the camera parameters. Finally, it calculates the 3D coordinates and colors of each point based on the depth information and returns a point cloud object in Open3D.

```python
def depth_image_to_point_cloud(rgb, depth,count):
    # camera intrinsic parameters
    fov = 90
    height, width = depth.shape
    focal_length = -(height/2)/np.tan(np.deg2rad(fov/2))
    cx ,cy = 256,256
    #matrix
    k_matrix = np.array([[focal_length, 0, cx],
                         [0, focal_length, cy],
                         [0, 0, 1]])
    #create coordinate point
    i, j = np.indices(depth.shape)
    coordinates_point = np.column_stack((j.flatten(), i.flatten(), np.ones_like(i).flatten()))

    # 2d to 3d
    colors = (rgb /255).reshape(-1,3) #shape is (512*512,3)
    pcd  =  np.dot(np.linalg.inv(k_matrix), coordinates_point.T).T #shape is (512*512,3)
    pcd *= depth.flatten()[:, np.newaxis] / 1000.0

    # creat point cloud
    pcd_o3d = o3d.geometry.PointCloud()  # create point cloud object
    pcd_o3d.points = o3d.utility.Vector3dVector(pcd)  # set pcd_np as the point cloud points
    pcd_o3d.colors = o3d.utility.Vector3dVector(colors)

    print(f"Point cloud {count+1} done")
    return pcd_o3d
```

Fig15. Depth image to point cloud

- ‘preprocess_point_cloud’ function:
  The function is from open3D ICP registration.
  http://www.open3d.org/docs/release/tutorial/pipelines/icp_registration.html
  Shown in figure16. This function preprocesses the point cloud, including voxel downsampling, normal estimation, and feature computation. It uses the ‘voxel_down_sample’ method to down sample the point cloud to reduce computational complexity.
  It estimates the normal vectors for each point. It also computes feature descriptors (FPFH) for each point.

```python
def preprocess_point_cloud(pcd, voxel_size, counter):
    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))

    print(f"Processed Point{counter+1} Done")
    return pcd_down, pcd_fpfh
```

Fig16. preprocess_point_cloud

- ‘execute_global_registration’ function:
  The function is from open3D ICP registration.
  http://www.open3d.org/docs/release/tutorial/pipelines/icp_registration.html
  Shown in figure17. This function performs global point cloud registration to find the best initial transformation. It uses RANSAC-based feature matching to calculate the transformation matrix between two point clouds.

```
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))
    return result
```

Fig17. execute_global_registration function

- ‘local_icp_algorithm’ function:
  The function is from open3D ICP registration.
  http://www.open3d.org/docs/release/tutorial/pipelines/icp_registration.html
  Shown in figure18. This function performs local point cloud registration to refine the results of global registration.It uses the ICP (Iterative Closest Point) method to calculate the transformation matrix between two point clouds.

```
def local_icp_algorithm(source, target, transformation, voxel_size):
    distance_threshold = voxel_size * 0.4 #0.4 original

    result = o3d.pipelines.registration.registration_icp(
        source, target, distance_threshold, transformation,
        o3d.pipelines.registration.TransformationEstimationPointToPlane())
    return result
```

Fig18. local_icp_algorithm

- My ICP approach including three function:
  These functions are used for point cloud registration, where ‘**best_fit_transform**’ finds the optimal transformation between two point clouds, ‘**nearest_neighbor**’ identifies the nearest neighbors, and

‘**my_local_icp_algorithm**’ performs local point cloud registration using the ICP algorithm. These are essential steps in aligning and registering 3D data for applications such as 3D reconstruction and mapping.

- ‘best_fit_transform’ function:

Shown in figure18. This function calculates the least-squares best-fit transform between two sets of corresponding 3D points A and B.
It first translates both sets of points to their respective centroids (centers of mass). Then, it computes the rotation matrix R by performing Singular Value Decomposition (SVD) on a specific matrix W constructed from the points. A special reflection case is handled to ensure proper orientation (avoid reflection) of the transformation. Finally, it calculates the translation vector t, and the homogeneous transformation matrix T that combines the rotation and translation.

```
def best_fit_transform(A, B):
    '''
      A: Nx3  B: Nx3
      T: 4x4 homogeneous transformation matrix
      R: 3x3 rotation matrix
      t: 3x1 column vector
    '''
    assert len(A) == len(B)

    # translate points to their centroids
    centroid_A = np.mean(A, axis=0)
    centroid_B = np.mean(B, axis=0)
    AA = A - centroid_A
    BB = B - centroid_B

    # rotation matrix
    W = np.dot(BB.T, AA)
    U, s, VT = np.linalg.svd(W)
    R = np.dot(U, VT)
    # special reflection case
    if np.linalg.det(R) < 0:
        VT[2,:] *= -1
        R = np.dot(U, VT)

    # translation
    t = centroid_B.T - np.dot(R, centroid_A.T)

    # homogeneous transformation
    T = np.identity(4)
    T[0:3, 0:3] = R
    T[0:3, 3] = t

    return T, R, t
```

Fig19. best_fit_transform function

- ‘nearest_neighbor’ function:

Shown in figure 20. This function finds the nearest neighbor in the destination point cloud **dst** for each point in the source point cloud **src**.
It uses the **NearestNeighbors** algorithm to efficiently locate the nearest

neighbors.
It returns the Euclidean distances (errors) of the nearest neighbors, the indices of the nearest neighbors in the destination point cloud, and a validity flag indicating whether a neighbor is found.
The implementation of this function is based on the **NearestNeighbors**, which is a Scikit-Learn tool for nearest neighbor search. The function first constructs a **NearestNeighbors** model and uses the destination point cloud **dst** to train the model. It then uses the source point cloud **src** to query for the nearest neighbors, returning the distances and indices for each source point. Finally, it filters out invalid distances and returns the results to the user. This function is commonly used in applications like point cloud registration to find the nearest neighbors for each point in a source point cloud within a target point cloud.

```
def nearest_neighbor(src, dst):
    '''
    Find the nearest (Euclidean) neighbor in dst for each point in src
    Input:
        src: Nx3 array of points
        dst: Nx3 array of points
    Output:
        distances: Euclidean distances (errors) of the nearest neighbor
        indecies: dst indecies of the nearest neighbor
    '''

    neigh = NearestNeighbors(n_neighbors=1)
    neigh.fit(dst)

    distances, indices = neigh.kneighbors(src, return_distance=True)
    valid=distances < np.median(distances)*0.8
    return distances[valid].ravel(), indices[valid].ravel(),valid.ravel()
```

Fig20. nearest_neighbor function

- ‘my_local_icp_algorithm’ function:

Shown in figure21. This function implements the Iterative Closest Point (ICP) algorithm for local point cloud registration.
It takes source point cloud A and destination point cloud B, an initial transformation init_pose, maximum iterations max_iterations, and a convergence tolerance tolerance as input.
The function first converts the point clouds to homogeneous coordinates. It applies the initial pose estimation if provided.

Then, it iteratively updates the source point cloud **src** based on the nearest neighbors in the destination point cloud and calculates the transformation T. The iterations continue until the change in mean error (distance) between the source and destination points falls below the specified tolerance or the maximum iterations are reached.
The final transformation T is calculated, which represents the optimized alignment between the two point clouds.
For each iteration:
Find the nearest neighbors in the destination point cloud (B) for each point in the current source point cloud (A). This is done by calling the nearest_neighbor function.Compute the transformation T (comprising both rotation and translation) that best aligns the source (A) with the nearest points in the destination (B). The **best_fit_transform** function is called for this purpose. Update the current source point cloud (A) by applying the transformation T. This is a standard step in the ICP algorithm, where the source point cloud is iteratively refined. Calculate the mean error by computing the average distance between corresponding points in the updated source point cloud and the destination point cloud.
Check if the change in mean error between iterations is smaller than the specified tolerance. If it is, the algorithm converges, and the loop breaks.

```python
def my_local_icp_algorithm(A, B, init_pose=None,  max_iterations=1000000 ,tolerance=0.000002):
    # floor 1 max_iterations=1000000 ,tolerance=0.000002

    A = np.asarray(A.points)
    B = np.asarray(B.points)
    # make points homogeneous, copy them so as to maintain the originals
    src = np.ones((4,A.shape[0]))
    dst = np.ones((4,B.shape[0]))
    src[0:3,:] = np.copy(A.T)
    dst[0:3,:] = np.copy(B.T)

    # apply the initial pose estimation
    if init_pose is not None:
        src = np.dot(init_pose, src)
    prev_error = 0

    # main part
    for i in range(max_iterations):
        # find the nearest neighbours between the current source and destination points
        distances, indices,valid = nearest_neighbor(src[0:3,:].T, dst[0:3,:].T)
        # compute the transformation between the current source and nearest destination points
        T,_,_ = best_fit_transform(src[0:3,valid].T, dst[0:3,indices].T)
        # update the current source
    # refer to "Introduction to Robotics" Chapter2 P28. Spatial description and transformations
        src = np.dot(T, src)

        # check error
        mean_error = np.sum(distances) / distances.size
        if abs(prev_error-mean_error) < tolerance:
            break
        prev_error = mean_error

    # calculcate final tranformation
    T,_,_ = best_fit_transform(A, src[0:3,:].T)

    return T
```

Fig21. my_local_icp_algorithm

- ‘reconstruct’ function:
  Shown in figure 22 and figure 23. This function executes the entire reconstruction process, including global registration, local registration, and generating the reconstructed point cloud.
  It returns the generated 3D point cloud and camera trajectory.
  I will detailed explain every part below.

1. **Determine Reconstruction Mode**:
   - The function first checks the reconstruction mode specified in the **args** variable, which could be either 'open3d' or 'my_icp'. The mode determines which reconstruction algorithm to use.
2. **Initialize Point Cloud and Camera Cloud**:
   - A new **point_cloud** variable is created as an empty Open3D point cloud, which will store the final reconstructed scene.
   - An array **point_final** is initialized with the first point cloud (indexed by **pcd_down[0]**), and it will be gradually updated with each iteration.
   - A **pred_cam_pos** variable is initialized to store the camera positions as a LineSet, and a list **camera_cloud_list** is created to store individual camera cloud point clouds.
3. **Iterative Reconstruction**:
   - The function enters an iterative loop to reconstruct the scene point cloud. It iterates over all point clouds in **pcd_down**.
   - For each iteration, it performs the following steps:
     - It calculates the transformation between the current point cloud and the previous one using global registration (**execute_global_registration**). This transformation is essential for aligning point clouds.
     - Depending on the chosen mode ('open3d' or 'my_icp'), it either performs local ICP (Iterative Closest Point) registration using Open3D's **local_icp_algorithm** or a custom **my_local_icp_algorithm**. This step fine-tunes the alignment.
     - The transformed current point cloud is added to **point_final**, and the corresponding camera cloud is updated with the transformation.
     - A message is printed indicating the completion of each reconstruction step.
4. **Create Camera Line Set**:
   - Camera positions from **camera_cloud_list** are concatenated into a single point cloud (**pred_cam_pos**) for visualization as a LineSet. It represents the estimated camera trajectories.
5. **Final Point Cloud Processing**:
   - All point clouds in **point_final** are combined into the **point_cloud** to

represent the final reconstructed scene.

- A thresholding operation is applied to hide the ceiling by removing points with y-coordinates above a certain threshold (**threshold_y**).
- The filtered point cloud is returned, along with the camera positions (**pred_cam_pos**) and the camera point clouds.

In summary, this function iteratively reconstructs a 3D scene by aligning a series of input point clouds using the specified reconstruction mode (either Open3D's methods or a custom ICP algorithm). It also visualizes the estimated camera trajectories. The result is a final 3D point cloud representing the reconstructed scene with the ceiling removed.

```python
def reconstruct(args,pcd_down,pcd_fpfh):
    # TODO: Return results
    if args.version == 'open3d':
        mode = "open3d"
    elif args.version == 'my_icp':
        mode = "my_icp"

    #recreate
    point_cloud = o3d.geometry.PointCloud()  # create point cloud object
    point_final=[]
    point_final.append(pcd_down[0])

    #create camera point cloud

    pred_cam_pos = o3d.geometry.LineSet()
    camera_cloud_list = []
    for i in range(len(pcd_down)):
        camera_cloud = o3d.geometry.PointCloud()
        camera_cloud.points.append([0 ,0, 0])
        camera_cloud.colors.append([1 ,0, 0])
        camera_cloud_list.append(camera_cloud)

    # algorimn for recreate
    for i in range(1,len(pcd_down)):
        result_ransac = execute_global_registration(pcd_down[i], point_final[i-1], pcd_fpfh[i], pcd_fpfh[i-1], voxel_size)
        transformation = result_ransac.transformation
        if mode == 'open3d':
            result_icp = local_icp_algorithm(pcd_down[i], point_final[i-1], transformation,voxel_size)
            transformation = result_icp.transformation
        elif mode == 'my_icp':
            result_icp = my_local_icp_algorithm(pcd_down[i], point_final[i-1], transformation)
            transformation = result_icp

        point_final.append(pcd_down[i].transform(transformation))
        camera_cloud_list[i]=camera_cloud_list[i].transform(transformation)
        print(f'recreate {i} done')
```

Fig22. reconstruct function part1

```
#camera point to line
camera_points = []
for point in camera_cloud_list:
    point_coordinates = np.asarray(point.points)
    camera_points.append(point_coordinates)
points_o3d = o3d.utility.Vector3dVector(np.concatenate(camera_points, axis=0))

pred_cam_pos.points = points_o3d
lines = []
for i in range(len(camera_points) - 1):
    lines.append([i, i+1])

pred_cam_pos.lines = o3d.utility.Vector2iVector(lines)
line_color = [1, 0, 0]
line_colors = [line_color] * len(lines)
pred_cam_pos.colors = o3d.utility.Vector3dVector(line_colors)

# final point cloud
for final in point_final:
    point_cloud += final

#hidden ceiling
xyz_points = np.asarray(point_cloud.points)
colors = np.asarray(point_cloud.colors)
threshold_y = 0.0128
filtered_xyz_points = xyz_points[xyz_points[:, 1] <= threshold_y]
filtered_colors = colors[xyz_points[:, 1] <= threshold_y]
point_cloud.points = o3d.utility.Vector3dVector(filtered_xyz_points)
point_cloud.colors = o3d.utility.Vector3dVector(filtered_colors)

return point_cloud  , pred_cam_pos , camera_points
```

Fig23. reconstruct function part2

- ‘goundtruth’ function:
  Shown I figure24. This function is used to read ground truth trajectory data. It converts the read ground truth trajectory data into a 3D point cloud and a LineSet for visualizing the real trajectory.
  In the data on the second floor, fine-tuning is performed to align it with the initial position of the regression and the camera.

```
def goundtruth(args):
    #ground truth
    data = np.load('./'+str(args.data_root)+'GT_pose.npy')
    if args.floor == 1:#args.floor == 1:
        x = -data[:, 0]/40
        y = data[:, 1]/40
        z = -data[:, 2]/40
        rotation = data[:, 3:]  # rw, rx, ry, rz
    elif args.floor == 2: #
        x = -data[:, 0]/40-0.00582038
        y = data[:, 1]/40-0.07313087
        z = -data[:, 2]/40-0.03052245
        rotation = data[:, 3:]  # rw, rx, ry, rz

    # 創建真實軌跡的3D模型
    real_trajectory_points = np.column_stack((x, y, z))
    real_trajectory_line = o3d.geometry.LineSet()
    real_trajectory_line.points = o3d.utility.Vector3dVector(real_trajectory_points)

    # 創建線的連接
    lines = []
    for i in range(len(real_trajectory_points) - 1):
        lines.append([i, i+1])
    real_trajectory_line.lines = o3d.utility.Vector2iVector(lines)

    return real_trajectory_line ,real_trajectory_points
```

Fig24. Goundtruth function

- ‘computeL2 function’

  Shown in figure25. This function calculates the L2 distance between the ground truth trajectory and the camera trajectory.

```python
def computeL2(goundtruthpoints,camera_points):

    l2 = []
    for i in range(len(goundtruthpoints)):#len(goundtruthpoints)):
        data = abs (goundtruthpoints[i]-camera_points[i])
        distance = np.sqrt(data[0][1]**2 + data[0][1]**2 + data[0][2]**2)
        l2.append(distance )
    l2_average = np.mean(l2)

    return l2_average
```

  Fig25. computeL2 function

- **if __name__ == '__main__'**

In the **if __name__ == '__main__':** block, the code starts executing and selects the floor and processing mode based on command-line parameters. Image data is read and converted into 3D point clouds. The point clouds are preprocessed. 3D point clouds are reconstructed through global and local registration. Ground truth trajectory data is read. The L2 distance is calculated. 3D point clouds, real trajectories, and camera trajectories are visualized.

In summary, this code performs a 3D reconstruction pipeline based on depth and color images, including point cloud processing, registration, and visualization. It can be used to transform real-world scenes into 3D models for further analysis and visualization.

```python
if __name__ == '__main__':
    start = time.time()
    parser = argparse.ArgumentParser()
    parser.add_argument('-f', '--floor', type=int, default=1)
    parser.add_argument('-v', '--version', type=str, default='my_icp', help='open3d or my_icp')
    parser.add_argument('--data_root', type=str, default='data_collection/first_floor/')
    args = parser.parse_args()

    if args.floor == 1:
        args.data_root = "data_collection/first_floor/"
    elif args.floor == 2:
        args.data_root = "data_collection/second_floor/"

    #data list
    rgb_file_list = glob.glob(r'./'+str(args.data_root)+'rgb/*')
    depth_file_list = glob.glob(r'./'+str(args.data_root)+'depth/*')

    rgb_file_list = ["./"+str(args.data_root)+"rgb/{}.png".format(i+1) for i in range(len(rgb_file_list))]
    depth_file_list = ["./"+str(args.data_root)+"depth/{}.png".format(i+1) for i in range(len(depth_file_list))]

    #read image
    rgb_image = np.array([np.array(Image.open(fname)) for fname in rgb_file_list]) # data shape is (number of pic
    depth_image = np.array([np.array(Image.open(fname)) for fname in depth_file_list]) # data shape is (number of
```

Fig26. Main function part 1

```python
#original point cloud
pcd_list = []
camera_list = []
pcd_list = [ depth_image_to_point_cloud(rgb_image[i],depth_image[i],i)  for i in range(len(rgb_image))] #le

#processed poind cloud
voxel_size = 0.0028
pcd_down = []
pcd_fpfh = []
for i in range(len(rgb_image)):
    pcd ,fpfh= preprocess_point_cloud(pcd_list[i], voxel_size, i)
    pcd_down.append(pcd)
    pcd_fpfh.append(fpfh)

#reconstruct
#o3d.visualization.draw_geometries([reconstruct(args,pcd_down,pcd_fpfh)])
hiddenceiling_pointcloud , camera_pose ,camera_points = reconstruct(args,pcd_down,pcd_fpfh)
real_trajectory_line , goundtruthpoints = goundtruth(args)
print(f'The L2 is {computeL2(goundtruthpoints,camera_points)}')
end = time.time()
o3d.visualization.draw_geometries([hiddenceiling_pointcloud,real_trajectory_line,camera_pose])


print(f'The construction takes {end-start} second')
```

Fig27. Main function part 2

b. Result and Discussion

I. Result of Construction

Data size :188 Execution time:49.83sec

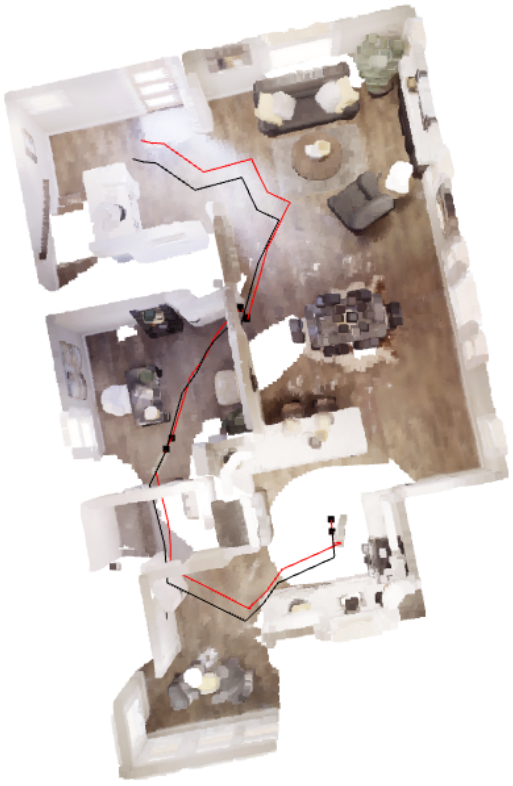


Fig28. open3d floor1

Data size :148 Execution time: 34.75sec

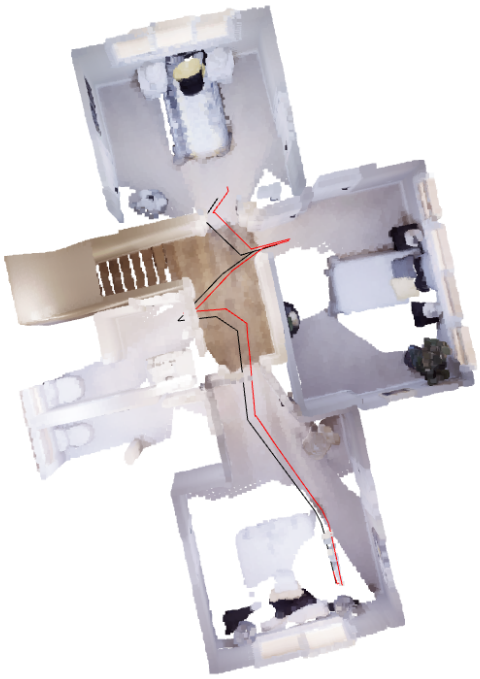


Fig29. open3d floor2

Data size :188 Execution time:58.16sec

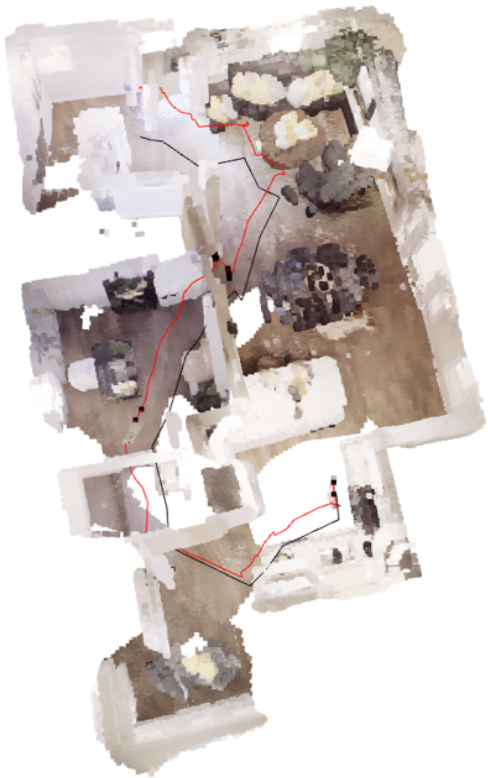


Fig30. My ICP floor1

Data size :148 Execution time: 36.05sec

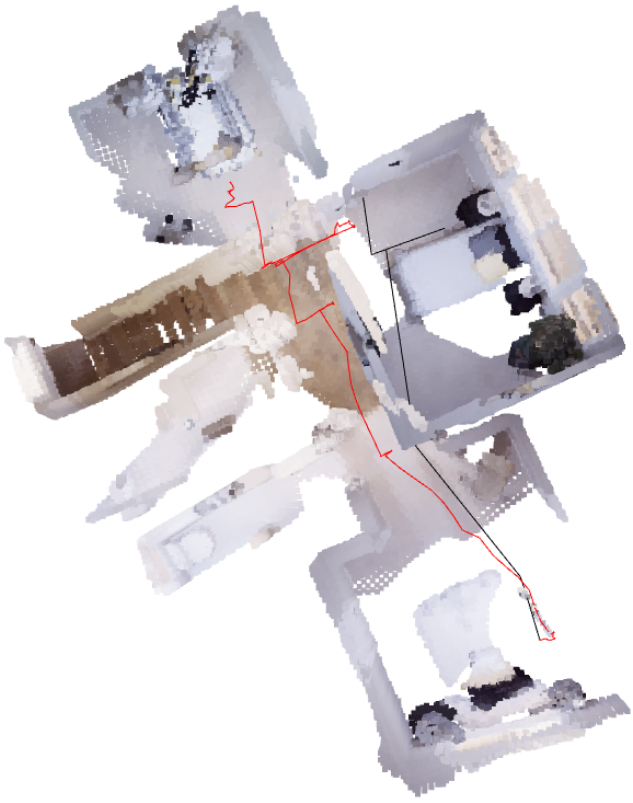


Fig31. My ICP floor2

II. Mean L2 distance

The function to compute the L2 have already mentioned above (Task 2 Code)

Table. L2 distance

Mean L2	Open3D	My ICP
Floor1	0.0079	0.01919
Florr2	0.00209	0.0135

III. Discussion

I found that the most time-consuming part during the reconstruction process is function **'depth_image_to_point_cloud'**. Thus , I was looking for a way that can speed it up.

So, I would like to discuss the different between using 'numpy' or not in function **'depth_image_to_point_cloud'**. Figure32 is normal version without Numpy. Figure33 is Numpy version.

Shown in table2, numpy version can save much time.

Table2. Normal v.s Numpy

	Normal	Numpy
Floor1 with data 188	186.21sec	28.746 sec
Floor2 with data 148	133 sec	15 sec

```python
def depth_image_to_point_cloud2(rgb, depth):
    # TODO: Get point cloud from rgb and depth image
    FX_DEPTH = -256
    FY_DEPTH = -256
    CX_DEPTH = 256
    CY_DEPTH = 256
    pcd = []
    colors = []
    height, width = depth.shape

    for i in range(height):
        for j in range(width):
            z = depth[i][j]/50.0 # normalize (not sure neccesse
            x = (j - CX_DEPTH) * z / FX_DEPTH
            y = (i - CY_DEPTH) * z / FY_DEPTH

            color = rgb[i, j] / 255.0
            pcd.append([x, y, z])
            colors.append(color)
    pcd_o3d = o3d.geometry.PointCloud()  # create point cloud o
    pcd_o3d.points = o3d.utility.Vector3dVector(pcd)  # set pcd
    pcd_o3d.colors = o3d.utility.Vector3dVector(colors)
    return pcd_o3d
```

Fig32. Normal version

```python
def depth_image_to_point_cloud(rgb, depth):
    # camera intrinsic parameters
    fov = 90
    height, width = depth.shape
    focal_length = -(height/2)/np.tan(np.deg2rad(fov/2))
    cx ,cy = 256,256
    #matrix
    k_matrix = np.array([[focal_length, 0, cx],
                         [0, focal_length, cy],
                         [0, 0, 1]])
    #create coordinate point
    i, j = np.indices(depth.shape)
    coordinates_point = np.column_stack((j.flatten(), i.flatten(), np.ones_like(i).flatten()))

    # 2d to 3d
    colors = (rgb /255).reshape(-1,3) #shape is (512*512,3)
    pcd  =  np.dot(np.linalg.inv(k_matrix), coordinates_point.T).T #shape is (512*512,3)
    pcd *= depth.flatten()[:, np.newaxis] / 1000.0

    # creat point cloud
    pcd_o3d = o3d.geometry.PointCloud()  # create point cloud object
    pcd_o3d.points = o3d.utility.Vector3dVector(pcd)  # set pcd_np as the point cloud points
    pcd_o3d.colors = o3d.utility.Vector3dVector(colors)


    return pcd_o3d
```

Fig 33 Numpy version

IV. Reference

https://zhuanlan.zhihu.com/p/107218828
https://github.com/richardos/icp/blob/master/icp.py
https://github.com/alvinwan/pcmatch/blob/master/src/icp.py

2. Questions

a. What's the meaning of extrinsic matrix and intrinsic matrix

Shown in figure34. The picture coordinate vector is [u, v, 1], and the real-world coordinate vector is [X, Y, Z, 1].

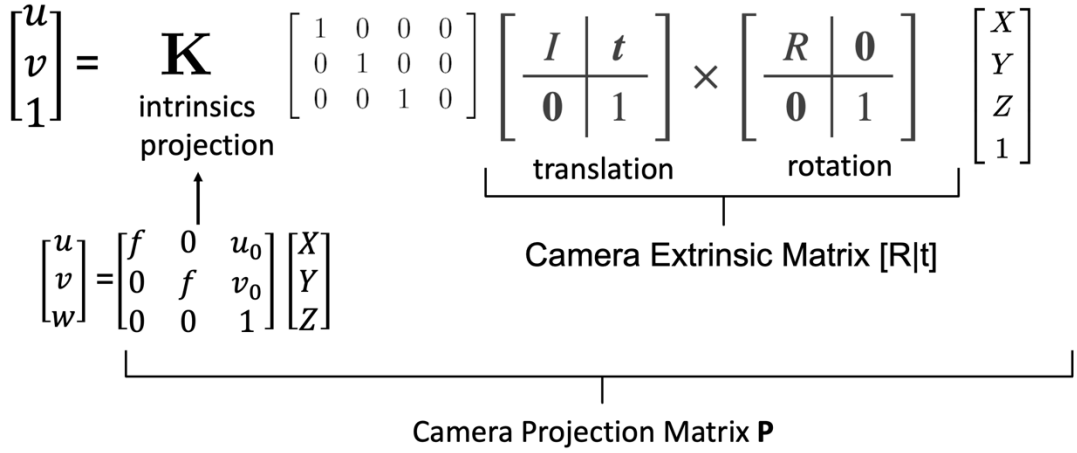


Fig34. Camera parameter

1. Extrinsic Matrix:
   Shown below.
   Left 3X3 matrix is rotation matrix and the right 3X1 matrix is translate vector.

$$[R\,|\,t] = \left[\begin{array}{ccc|c} r_{1,1} & r_{1,2} & r_{1,3} & t_1 \\ r_{2,1} & r_{2,2} & r_{2,3} & t_2 \\ r_{3,1} & r_{3,2} & r_{3,3} & t_3 \end{array}\right]$$

   - The extrinsic matrix (also known as the "pose matrix" or "view matrix") represents the position and orientation of a camera or imaging device in the 3D world.
   - It describes the camera's location and orientation in a global coordinate system, such as the world coordinate system.

- The extrinsic matrix typically includes a 3x3 rotation matrix that describes how the camera is oriented in space and a 3x1 translation vector that represents the camera's position.
- It allows transforming 3D points from the global coordinate system to the camera's coordinate system, enabling the projection of 3D points into 2D image coordinates.

2. Intrinsic Matrix:
   Shown below.
   Fx and fy are focal length, which is the distance from the center of the lens to the focal point where light converges.
   x0 and y0 are principle points offset.

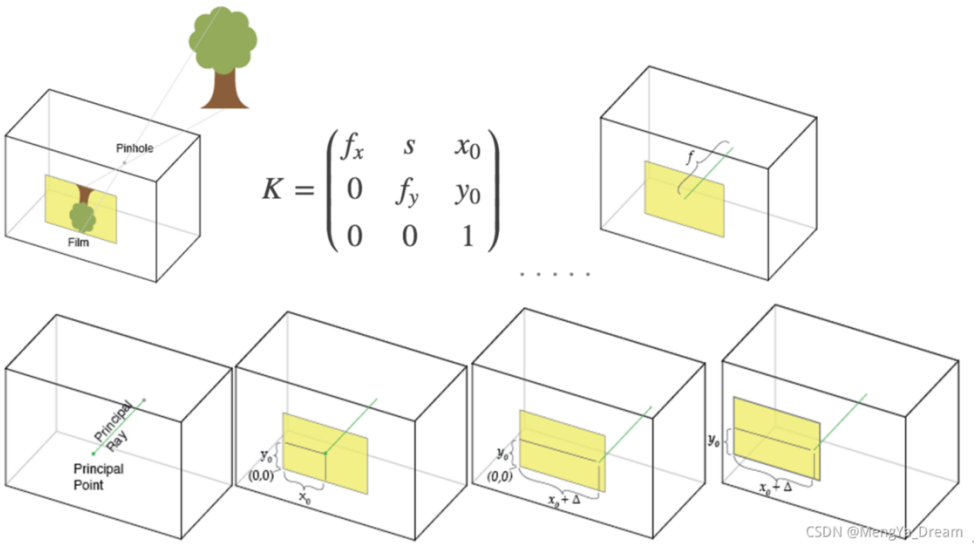


Fig35 concept of Intrinsic Matrix

- The intrinsic matrix (also known as the "camera matrix" or "calibration matrix") represents the internal parameters of a camera.
- It describes how the 3D world is projected onto a 2D image plane within the camera.
- The intrinsic matrix typically includes parameters like the focal length, the principal point (the image center), and the skew factor.
- It allows mapping 3D points from the camera's coordinate system to 2D image coordinates.
- The intrinsic matrix is essential for the camera calibration process, as it helps correct for distortions and allows accurate mapping between the 3D world and 2D images.

In summary, the extrinsic matrix deals with the camera's position and orientation in the global 3D world, while the intrinsic matrix deals with the camera's internal parameters that affect how 3D objects are projected onto a 2D image. Both matrices are crucial for tasks like 3D reconstruction, camera calibration, and pose estimation.

b. Have you ever tried to do ICP alignment without global registration, i.e. RANSAC? How's the performance? Explain the reason. (Hint: The limitation of ICP alignment)

As shown in figure35, the performance without global registration is much worse than the version with global registration.

The main reason is **the importance of initial values lies in that an inappropriate initial value could lead ICP algorithm to trap in a local minimum.**

What's more, Iterative Closest Point (ICP) is a widely used algorithm for point cloud registration and alignment. However, ICP has certain limitations:

1. **Local Minimums:** ICP is sensitive to the initial alignment, and it may converge to a local minimum instead of the global minimum. To mitigate this, one can use techniques like random restarts or alternative algorithms that rely on feature correspondences rather than point-to-point distances.
2. **Limited to Rigid Transformations:** ICP assumes that the transformation between the source and target point clouds is a rigid transformation, meaning it consists of rotations and translations only. It cannot handle non-rigid deformations, such as bending or stretching.
3. **Convergence:** While ICP is often fast, it may converge slowly or not at all in some cases, especially if the initial alignment is far from the correct alignment. This is especially problematic in large-scale or noisy datasets.
4. **Computational Complexity:** The time complexity of ICP is O(n^2), where n is the number of points in the point clouds. This can make ICP impractical for very large datasets.
5. **Sensitivity to Outliers:** ICP is sensitive to outliers, which can significantly affect the quality of the alignment. Outliers can distort the transformation and lead to incorrect results. Outlier rejection techniques, such as trimming or robust cost functions, are often used to mitigate this issue.

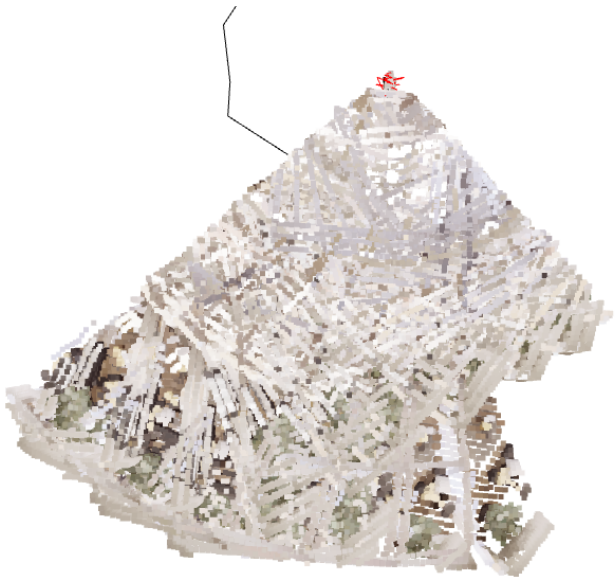


Fig35. Floor1 without global registration

In response to the limitations of the standard ICP algorithm, many researchers have proposed various improved versions of the ICP algorithm, primarily involving the following six aspects.

1. Selection source points
2. Matching to points in other mesh
3. Weighting the correspondences
4. Rejecting certain point pairs
5. Assigning an error metric to the current transform
6. Minimizing the error metric w.r.t transform

c. Describe the tricks you apply to improve your ICP alignment.

We have already detailed explained our ICP on above(Task2 Code).

However, there are two tricks I would like to mentioned.

1. As shown in figure19, our best fit approach had a limit to rotation matrix. It checks for a special reflection case by examining the determinant of the rotation matrix R. If the determinant is negative, it corrects it by inverting the third row of the VT matrix. In 3D, a proper rotation matrix should have a determinant of 1 because it represents a transformation that preserves volume (i.e., it is a rigid-body transformation). However, in some cases, the computed rotation matrix may have a determinant of -1, indicating a reflection (improper rotation) rather than a proper rotation. A reflection is a transformation that flips the object over a plane and changes its orientation.

In the context of point cloud registration and alignment, it's important that the calculated rotation represents a proper rotation rather than a reflection. A reflection can significantly affect the alignment of two point sets and can lead to incorrect results in registration. That's why the correction is applied.
The correction involves **inverting the third row of the VT matrix, which effectively corrects the sign of the determinant to make it positive (or 1)**. This ensures that the resulting rotation matrix R represents a proper rotation, allowing it to accurately align the point sets.

2. As shown on figure20, in the final step of our code, it filters the valid nearest neighbors based on the distance information in distances. The filtering condition is to keep only those points whose distance is less than 80% of the median of all distances. This condition aims to remove outliers or anomalies to obtain a more stable and reliable nearest neighbor relationship. The function returns the distances of the valid nearest neighbors, their indices, and a boolean array indicating which points are valid nearest neighbors.