Perception and Decision Making in Intelligent Systems Homework 1

313554024 林慧旻

1. Implementation

Task1:

a. **Code**

Detailed explanation of your implementation. For example, how you do the projection.

```
class Projection(object):

    def __init__(self, image_path, points):
        """
            :param points: Selected pixels on top view(BEV) image
        """

        if type(image_path) != str:
            self.image = image_path
        else:
            self.image = cv2.imread(image_path)
        self.height, self.width, self.channels = self.image.shape
        self.points = points
        self.focal = self.width / (2*np.tan(np.pi/4))

    def ex_mat(self, alpha, beta, gamma, tx, ty, tz):
        r11 = np.cos(alpha)*np.cos(beta)
        r12 = np.cos(alpha)*np.sin(beta)*np.sin(gamma) - np.sin(alpha)*np.cos(gamma)
        r13 = np.cos(alpha)*np.sin(beta)*np.cos(gamma) + np.sin(alpha)*np.sin(gamma)
        r21 = np.sin(alpha)*np.cos(beta)
        r22 = np.sin(alpha)*np.sin(beta)*np.sin(gamma) + np.cos(alpha)*np.cos(gamma)
        r23 = np.sin(alpha)*np.sin(beta)*np.cos(gamma) - np.cos(alpha)*np.sin(gamma)
        r31 = -np.sin(beta)
        r32 = np.cos(beta)*np.sin(gamma)
        r33 = np.cos(beta)*np.cos(gamma)
        ex = np.array([[r11, r12, r13, tx],
                       [r21, r22, r23, ty],
                       [r31, r32, r33, tz],
                       [0, 0, 0, 1]])
        return ex
```

1. `__init__(self, image_path, points)`:
   - The constructor takes the image path (or image itself) and a list of pixel points (`points`) from the BEV image.
   - It reads the image from `image_path`.
   - It also calculates the **focal length** using the formula:

$$focal = \frac{\text{width}}{2 \cdot \tan(\frac{\pi}{4})}$$

   This corresponds to a horizontal field of view (FoV) of 90 degrees.

2. `ex_mat(self, alpha, beta, gamma, tx, ty, tz)`:
   - This function generates the **extrinsic matrix** which represents a transformation from the world coordinate system (3D space) to the camera coordinate system.
   - It combines both rotation (angles `alpha`, `beta`, `gamma` for the x, y, z axes) and translation (shifts `tx`, `ty`, `tz` along x, y, and z).
   - The matrix structure:

$$ex = \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix}$$

   where `R` is the rotation matrix and `T` is the translation vector.

   The whole pattern is as follows:

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

Though, in this task, it only rotate by x-axis.

- This matrix will be used later to transform points from the BEV image.

```
    def top_to_front(self, theta=0, phi=0, gamma=0, dx=0, dy=0, dz=0, fov=90):
        """
            Project the top view pixels to the front view pixels.
            :return: New pixels on perspective(front) view image
        """

        ### TODO ###
        in_mat = np.array([[self.focal, 0, 256],
                           [0, self.focal, 256],
                           [0, 0, 1]])

        new_pixels = []
        for i in self.points:
            i.append(1)
            XYZ = np.dot(np.linalg.inv(in_mat), np.reshape((np.array(i) * sensor_height_bev), (3, 1)))
            #print('XYZ: ', XYZ)
            ex_mat = self.ex_mat(theta, phi, gamma, dx, dy, dz) #dy = sensor_height - sensor_height_bev
            #print("transform: ", ex_mat)
            tmp = np.array([XYZ[0][0], XYZ[1][0], sensor_height_bev, 1])
            trans = np.dot(ex_mat, np.reshape(tmp, (4, 1)))

            convert_back = np.reshape(np.dot(in_mat, trans[:3]), (3))
            convert_back = convert_back / convert_back[2]
            convert_back = np.around(convert_back, decimals=0).astype(int)
            #print(convert_back)
            new_pixels.append([convert_back[0], convert_back[1]])

        return new_pixels
```

3. `top_to_front(self, theta=0, phi=0, gamma=0, dx=0, dy=0, dz=0, fov=90)`:

- The core function of the code is to project the **BEV** image points into a **front-facing perspective view**.

Process:

1. **Intrinsic matrix**:
   - It first creates an **intrinsic matrix** `in_mat`, which is defined as:

$$in\_mat = \begin{bmatrix} \text{focal} & 0 & 256 \\ 0 & \text{focal} & 256 \\ 0 & 0 & 1 \end{bmatrix}$$

     I set the center of the image as (256, 256).This matrix is used to convert from 3D points to 2D image points using the camera's internal parameters.

2. **Transformation loop**:
   - For each pixel point in `self.points`, the following transformations are applied:

   1. **Convert to 3D**:
      - The point is extended with an additional homogeneous coordinate (a `1` is appended).
      - The point is transformed into 3D coordinates using the **inverse of the intrinsic matrix** and the sensor height (`sensor_height_bev`).
   2. **Apply extrinsic transformation**:
      - The transformation matrix `ex_mat` (obtained from the `ex_mat()` function) is applied to move the point from BEV to the front-facing camera view. The `sensor_height_bev` is also considered in this step.
   3. **Reproject to 2D**:
      - The transformed 3D coordinates are converted back into 2D pixel coordinates using the **intrinsic matrix**.
      - The final pixel values are adjusted by dividing by the third coordinate (to account for the perspective divide), rounded, and added to the list `new_pixels`.

   Each new pixel is calculated and appended to `new_pixels`.

b. **Result and Discussion**

i. Result of your projection (2 different pairs).

Floor 1:

Floor 2:

ii. Anything you want to discuss

One thing I want to mention is that in `load.py`, the bev images is added as follows:

```
#rgb bev
rgb_sensor_spec_bev = habitat_sim.CameraSensorSpec()
rgb_sensor_spec_bev.uuid = "color_sensor_bev"
rgb_sensor_spec_bev.sensor_type = habitat_sim.SensorType.COLOR
rgb_sensor_spec_bev.resolution = [settings["height"], settings["width"]]
rgb_sensor_spec_bev.position = [0.0, 2.5, 0.0]
rgb_sensor_spec_bev.orientation = [
    -(np.pi/2),
    0.0,
    0.0,
]
```

where y of rgb_sensor_spec_bev.position is not `settings["sensor_height"]`. Since a higher bev hight offers a larger sight to do projection, and it will be easier to observe the results.

iii. Any reference you take

https://en.wikipedia.org/wiki/Rotation_matrix

https://blog.csdn.net/MengYa_Dream/article/details/120043541

Task2:

a. **Code**

Detailed explanation of your implementation. For example, how you implement ICP, depth_projection and other functions.

```
def reconstruct(args):
    # TODO: Return results
    os.makedirs(args.data_root + 'pcd/', exist_ok=True)
    # 1. Unproject depth images ti and ti+1 to reconstruct two point clouds.
    depth_image_to_point_cloud(512, args.data_root)
```

```
    n = len(os.listdir(args.data_root + 'rgb/'))
    # 2. Do voxelization to your point cloud for reducing the number of points for less memory usage and spe
    #o3d_img_to_3d(start=0, total_img=n)
    target, target_down, target_fpfh = prepare_dataset(args.voxel_size, args.data_root + 'pcd/', 1)

    vis = o3d.visualization.Visualizer()
    add_points(vis, target, 0.00005)

    if args.version == 'open3d':
        print('Use Open3D version.')
    elif args.version == 'my_icp':
        print('Use my ICP version.')
    else:
        raise Exception(f"Invalid version. Expected 'open3d' or 'my_icp'.")

    trans = []
    lines = []

    # Dealing with trajectory
    ex_matrix = ex_mat(0, 0, np.pi, 0, 0, 0)[:3, :3]
    track_point = np.zeros((1, 3))
    track_arr = np.zeros((1, 3))
    track_gt = pd.read_csv(args.data_root+'record.txt', header=None, sep=' ')
    #print(track_gt)
    track_gt_arr = np.array(track_gt.drop(track_gt.columns[3:], axis=1))
    #print(track_gt_arr[0])
    #print(track_gt_arr)
    track_gt_arr = np.subtract(track_gt_arr, track_gt_arr[0])
    track_gt_arr[:,2] *= -1.0
    #print(track_gt_arr)
    #track_gt_arr[1:] += 1.0
    track_point_pc = o3d.geometry.PointCloud()
    track_point = np.zeros((1, 3))
    track_point_pc.points = o3d.utility.Vector3dVector(track_point)

    prog = tqdm(total=n)
    for i in range(1, n+1):
        source, source_down, source_fpfh = prepare_dataset(args.voxel_size, args.data_root + 'pcd/', i-1)
        if args.wo_RANSAC:
            result_ransac = np.eye(4)
        else:
            result_ransac = execute_global_registration(source_down, target_down, source_fpfh, target_fpfh,
            result_ransac = result_ransac.transformation


        if args.version == 'open3d':
            # 3. Apply global registration first which is used as the initialization of the local methods.
            result = local_icp_algorithm(
                source_down,
                target_down,
                source_fpfh,
                target_fpfh,
                args.voxel_size,
                result_ransac
            )
            trans.append(result.transformation)
        else:
            # 4. Apply local registration, using ICP to obtain the transformation matrix.
            transformation_matrix, aligned_points = my_local_icp_algorithm(
                source_down,
                target_down,
                args.voxel_size,
                result_ransac,
                max_iterations=60,
                tolerance=1e-7,
                point_pairs_threshold=10,
                verbose=False
            )
            trans.append(transformation_matrix)

        source_temp = copy.deepcopy(source)
        track_pcd = o3d.geometry.PointCloud()
        track_pcd.points = o3d.utility.Vector3dVector(track_point)
        #print('track_gt_arr: ', track_gt_arr.shape)
        track_gt_arr[i-1] = np.dot(ex_matrix, track_gt_arr[i-1].T).T
```

```
        tmp = []
        for j in range(0, len(trans)):

            source_temp.transform(trans[-j])
            track_pcd.transform(trans[-j])
        add_points(vis, source_temp, 0.000005)
        track_point_T = np.array(track_pcd.points)
        track_arr = np.concatenate((track_arr, track_point_T))
        lines.append([i - 1, i])
        target, target_down, target_fpfh = copy.deepcopy(source), copy.deepcopy(source_down), copy.deepcopy(
        prog.update(1)

    # Estinated trajectory
    #print('track_arr: ', track_arr)
    track = o3d.geometry.PointCloud()
    track.points = o3d.utility.Vector3dVector(track_arr[1:])
    track.paint_uniform_color([1, 0, 0])# red
    add_points(vis, track, 0.00005)
    track_line = o3d.geometry.LineSet()
    track_line.points = o3d.utility.Vector3dVector(track.points)
    track_line.lines = o3d.utility.Vector2iVector(lines[:n-1])
    track_line.paint_uniform_color([1, 0, 0])
    vis.add_geometry(track_line)

    # GT trajectory
    track_gt = o3d.geometry.PointCloud()
    track_gt.points = o3d.utility.Vector3dVector(track_gt_arr)
    track_gt.paint_uniform_color([0, 0, 0])# red
    add_points(vis, track_gt, 0.00005)
    track_gt_line = o3d.geometry.LineSet()
    track_gt_line.points = o3d.utility.Vector3dVector(track_gt.points)
    track_gt_line.lines = o3d.utility.Vector2iVector(lines[:n-1])
    track_gt_line.paint_uniform_color([0, 0, 0])
    vis.add_geometry(track_gt_line)
    error = track_gt_arr[:track_arr.shape[0]]- track_arr[1:]
    print("Trajectory Error: " , np.linalg.norm(error))
    tb = time.time()
    print('Time: ', tb-ta)
    vis.run()
```

The simple breakdown of the codes above:

1. Unproject depth images to point clouds
2. Do Point Cloud Voxelization for the first target point cloud
3. Open3D Visualization
4. Load Ground Truth Trajectory
5. Main Loop for Point Cloud Registration

   ```
   for i in range(1, n+1):
   ```

   A loop starts to process each pair of consecutive depth images.

   - **Prepare Source Point Cloud for each Loop**:

     ```
     source, source_down, source_fpfh = prepare_dataset(args.voxel_size, args.data_root + 'pcd/', i-1)
     ```

   - **Perform Global Registration**:

     ```
     if args.wo_RANSAC:
         result_ransac = np.eye(4)
     else:
         result_ransac = execute_global_registration(source_down, target_down, source_fpfh, target_fpfh,
     args.voxel_size)
         result_ransac = result_ransac.transformation
     ```

     If the `args.wo_RANSAC` flag is set, no global registration (RANSAC) is performed, and an identity matrix is used. Otherwise, global registration is computed as an initialization for local registration.

   - **Perform Local Registration (ICP)**:
     Depending on the version, either Open3D's or the custom ICP algorithm is used to refine the alignment:

     ```
     if args.version == 'open3d':
         result = local_icp_algorithm(...)
         trans.append(result.transformation)
     else:
     ```

```
    transformation_matrix, aligned_points = my_local_icp_algorithm(...)
    trans.append(transformation_matrix)
```

- **Update Point Cloud**:
  The source point cloud is transformed and visualized after each registration step:

```
source_temp.transform(trans[-j])
add_points(vis, source_temp, 0.000005)
```

In the following, I will explain each functions used in `def reconstruction`

`make_pcd() & depth_image_to_point_cloud`

```
def make_pcd(rgb, depth):

    pcd = o3d.geometry.PointCloud()
    hight, weight, focal = 512, 512, 256
    v, u = np.mgrid[0:hight, 0:weight]


    z = depth[:, :, 0].astype(np.float32) / 255 * (-10)  #convert depth map to meters
    x = -(u - weight*.5) * z / focal
    y = (v - hight*.5) * z / focal

    points = np.stack((x, y, z), axis=-1).reshape(-1, 3)
    colors = (rgb[:, :, [2, 1, 0]].astype(np.float32) / 255).reshape(-1, 3)

    pcd.points = o3d.utility.Vector3dVector(points)
    pcd.colors = o3d.utility.Vector3dVector(colors)
    return pcd


def depth_image_to_point_cloud(width, path):
    # TODO: Get point cloud from rgb and depth image
    focal = width / (2 * np.tan(np.pi / 4))
    focal = float(focal)

    in_matrix = o3d.camera.PinholeCameraIntrinsic(width, width, focal, focal, width / 2.0, width / 2.0)
    n_img = len(os.listdir(path + "rgb/"))
    prog_bar = tqdm(desc='making point cloud...', total=n_img)
    for i in range(n_img):
        depth = cv2.imread(f"{path}/depth/{i}.png")
        rgb = cv2.imread(f"{path}/rgb/{i}.png")
        pcd = make_pcd(rgb, depth)
        o3d.io.write_point_cloud(path + 'pcd/{}.xyzrgb'.format(i), pcd, True)
        prog_bar.update(1)
    return pcd
```

1. `make_pcd()`
   1. Initializes an empty point cloud using Open3D.
   2. The depth image values are converted to real-world distances (in meters).
   3. Pixel coordinates (x, y) are transformed into 3D world coordinates using the depth information.
   4. The RGB values are extracted and normalized to fit the range `[0, 1]`.
   5. The points and their corresponding colors are added to the point cloud.
   6. Returns the generated point cloud.
2. `depth_image_to_point_cloud()`

This function creates point clouds for multiple pairs of RGB and depth images and saves them.

1. Calculates the focal using the image width.
2. For each RGB and depth image:
   - Reads the images from disk.
   - Calls `make_pcd()` to generate a point cloud.
   - Saves the point cloud in `.xyzrgb` format.

`preprocess_point_cloud` & `prepare_dataset`

```
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

    # raise NotImplementedError
    return pcd_down, pcd_fpfh


def prepare_dataset(voxel_size, path, i):
    pcd = o3d.io.read_point_cloud(path + '{}.xyzrgb'.format(i))
    pcd_down, pcd_fpfh = preprocess_point_cloud(pcd, voxel_size)
    return pcd, pcd_down, pcd_fpfh
```

1. `preprocess_point_cloud(pcd, voxel_size)`

- **Voxelization**: Downsamples the point cloud using a voxel grid with a given `voxel_size`, reducing the number of points to save memory and improve speed.
- **Estimate Normals**: Estimates the normals for each downsampled point using a search radius based on `voxel_size * 2`.
- **FPFH Features**: Computes Fast Point Feature Histograms (FPFH) for the downsampled point cloud to capture the local geometric properties. The search radius is set to `voxel_size * 5`.

2. `prepare_dataset(voxel_size, path, i)`

Reads a point cloud file and use `preprocess_point_cloud` to processes it.

`execute_global_registration`

```
def execute_global_registration(source_down, target_down, source_fpfh, target_fpfh, voxel_size):
    distance_threshold = voxel_size * 2.0
    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(0.95),
            o3d.pipelines.registration.CorrespondenceCheckerBasedOnDistance(distance_threshold)
        ], o3d.pipelines.registration.RANSACConvergenceCriteria(500000, 0.99))
    return result
```

Performs global point cloud registration using the RANSAC (Random Sample Consensus) algorithm to align two point clouds.

- **Distance Threshold**:

  ```
  distance_threshold = voxel_size * 2.0
  ```

  Sets the threshold for feature matching between source and target points. It's based on the voxel size.

- **RANSAC Registration**:

  ```
  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(0.95),
          o3d.pipelines.registration.CorrespondenceCheckerBasedOnDistance(distance_threshold)
      ], o3d.pipelines.registration.RANSACConvergenceCriteria(500000, 0.99))
  ```

  This line performs global registration using feature matching based on RANSAC. The algorithm:

  - Matches FPFH features between the source and target.
  - Estimates the transformation between the point clouds.
  - Uses a distance threshold and checks the correspondence based on:
    1. **Edge Length**: Ensures that the relative lengths between points in both point clouds are similar.
    2. **Distance**: Ensures the distance between corresponding points is within the distance threshold.
  - The **RANSAC convergence criteria** are set to 500,000 iterations and a confidence level of 99%.

`local_icp_algorithm`

```
def local_icp_algorithm(source, target, source_fpfh, target_fpfh, voxel_size, transformation):
    # TODO: Use Open3D ICP function to implement
    distance_threshold = voxel_size * 0.4
    result = o3d.pipelines.registration.registration_icp(
        source, target, distance_threshold, transformation,
        o3d.pipelines.registration.TransformationEstimationPointToPlane(),
```

```
        o3d.pipelines.registration.ICPConvergenceCriteria(max_iteration=50))
    return result
```

Performs local point cloud registration using the ICP (Iterative Closest Point) algorithm with open3d.

- **Distance Threshold**:

  ```
  distance_threshold = voxel_size * 0.4
  ```

  This defines the maximum allowable distance between corresponding points in the source and target point clouds for registration. A lower threshold focuses on closer point correspondences.

- **ICP Registration**:

  ```
  result = o3d.pipelines.registration.registration_icp(
      source, target, distance_threshold, transformation,
      o3d.pipelines.registration.TransformationEstimationPointToPlane(),
      o3d.pipelines.registration.ICPConvergenceCriteria(max_iteration=50))
  ```

  This performs the local ICP registration, which refines the alignment between the source and target point clouds by minimizing the difference between corresponding points:

  - **ICP**: Iteratively matches points between the source and target point clouds based on their current alignment, and adjust the transformation matrix at each step.
  - **Distance Threshold**: Only points within the threshold are considered as valid correspondences.
  - **TransformationEstimationPointToPlane**: A method that minimizes the point-to-plane distance, which is more accurate for surfaces than point-to-point minimization.
  - **ICP Convergence Criteria**: The registration stops after 50 iterations or when it converges to a solution.

`my_local_icp_algorithm`

```
def my_local_icp_algorithm(points, reference_points, voxel_size, mtx_init, max_iterations=10, tolerance=1e
    points = np.array(points.points)
    reference_points = np.array(reference_points.points)
    n = points.shape[0]
    m = points.shape[1]
    src = np.ones((m + 1, n))
    trg = np.ones((m + 1, reference_points.shape[0]))

    src[:m, :] = np.copy(points.T)
    trg[:m, :] = np.copy(reference_points.T)

    transformation_matrix = mtx_init

    prev_error = float('inf')
    nbrs = NearestNeighbors(n_neighbors=1).fit(trg[:m, :].T)
    for iteration in range(max_iterations):
        src_transformed = transformation_matrix @ src

        distances, indices = nbrs.kneighbors(src_transformed[:m, :].T)

        valid_pairs = distances < voxel_size
        valid_source_points = src_transformed[:m, valid_pairs.flatten()]
        valid_target_points = trg[:m, indices.flatten()[valid_pairs.flatten()]]


        centroid_source = np.mean(valid_source_points, axis=1)
        centroid_target = np.mean(valid_target_points, axis=1)

        source_centered = valid_source_points - centroid_source[:, np.newaxis]
        target_centered = valid_target_points - centroid_target[:, np.newaxis]

        H = source_centered @ target_centered.T

        U, S, Vt = np.linalg.svd(H)
        R = Vt.T @ U.T

        if np.linalg.det(R) < 0:
            Vt[-1, :] *= -1
            R = Vt.T @ U.T

        t = centroid_target - R @ centroid_source

        new_transform = np.identity(4)
        new_transform[:3, :3] = R
        new_transform[:3, 3] = t
```

```
        transformation_matrix = new_transform @ transformation_matrix

        mean_error = np.mean(distances)


        prev_error = mean_error

    aligned_points = (transformation_matrix @ src)[:3, :].T

    return transformation_matrix, aligned_points
```

It is a custom implementation of the Iterative Closest Point (ICP) algorithm, used to align two point clouds (3D datasets of points) by iteratively minimizing the distance between them.

- **Initialization**:
  - The source (`points`) and target (`reference_points`) point clouds are converted to NumPy arrays.
  - `transformation_matrix` is initialized either as an identity matrix or the provided `mtx_init` matrix.
- **Nearest Neighbor Search Setup**:
  - The `NearestNeighbors` class from `sklearn` is used to find the closest point in the target point cloud for each point in the source point cloud.
- **ICP Loop**:
  - The algorithm runs for a maximum number of iterations.
  - **Transformation Application**:
    - In each iteration, the current transformation matrix is applied to the source points to align them with the target.
  - **Find Nearest Neighbors**:
    - After transforming the source points, the nearest points in the target are found using the `kneighbors` method.
  - **Filter Out Distant Points**:
    - `voxel_size` decides the number of total units (that is, how many points are seen as a unit and computed togerther).
    - If the number of valid point pairs is smaller than `point_pairs_threshold`, the iteration is stopped.
  - **Compute Centroids**:
    - The centroids (mean position) of the valid source and target points are calculated.
  - **Cross-Covariance Matrix**:
    - A cross-covariance matrix (`H`) is computed, which captures the relationships between the centered source and target points.
  - **SVD (Singular Value Decomposition)**:
    - The cross-covariance matrix is decomposed using SVD to calculate the optimal rotation matrix `R`.
    - If `R` leads to reflection (negative determinant), the sign of one of the axes is flipped to correct the rotation matrix.
  - **Translation Vector**:
    - The translation `t` is computed as the difference between the centroids of the target and rotated source points.
  - **Update Transformation Matrix**:
    - The transformation matrix is updated by combining the rotation `R` and translation `t`. This updated transformation matrix is applied to the source point cloud in the next iteration.
    - **Mean Error Calculation**:
      - The mean error (average distance between matched points) is computed.
    - **Convergence Check**:
      - If the change in error between iterations is smaller than the `tolerance`, the algorithm stops early, indicating convergence.
- **Final Transformation**:
  - Once the algorithm converges or reaches the maximum number of iterations, the final transformation matrix and the aligned points are returned.

`add_points`: Used to visualize the point clouds.

b. **Result and Discussion**

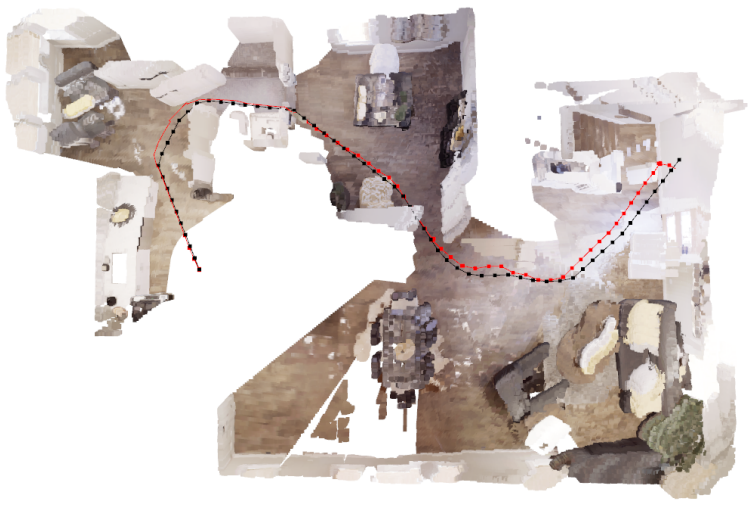
i. Result of your reconstruction (Floor 1 and Floor2, Both open3d implementation and your own implementation)

Make sure your execution time for each reconstruction less than 5 mins, or you will get 0 point in this part.

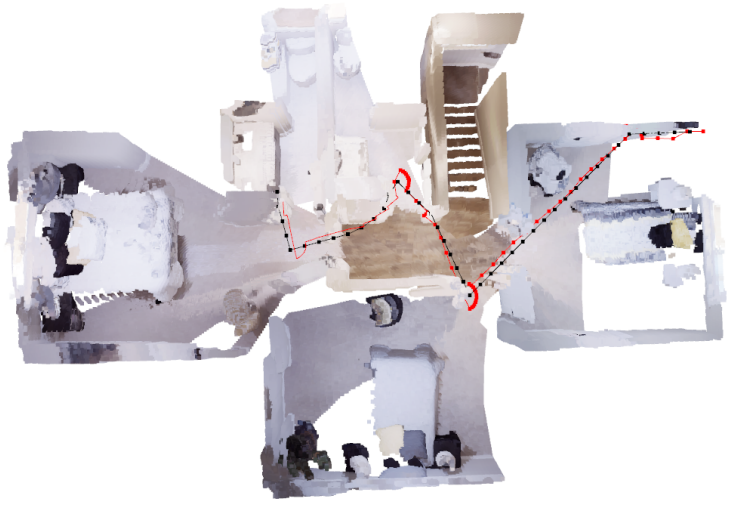
Data Size of F1=119/F2=96

My ICP (voxel_size=0.07)

Floor 1:

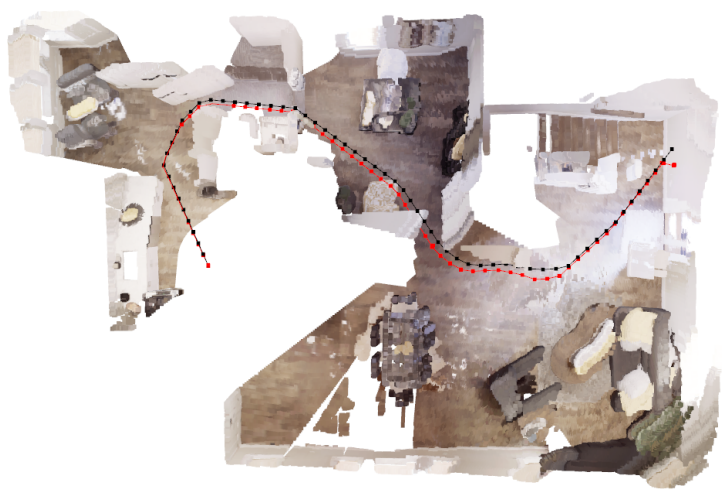


Floor 2:

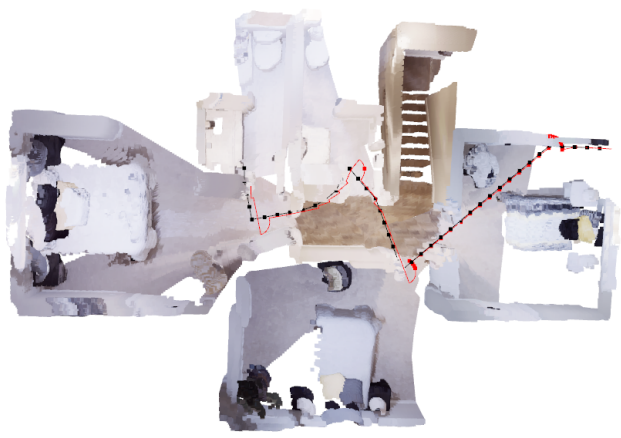


Open3d ICP (voxel_size=0.07)

Floor 1:



Floor 2:



ii. Mean L2 distance between ground truth and estimated trajectory.

My ICP (voxel_size=0.07)

Floor 1 Trajectory Error: 0.10963136632230293

Floor 2 Trajectory Error: 0.098673048813785

Open3d ICP (voxel_size=0.07)

Floor 1 Trajectory Error: 0.0681455961929277

Floor 2 Trajectory Error: 0.1237533220359232

iii. Anything you want to discuss, such as comparing the performance of two implementations.

Open3d undoubtedly offers a faster ICP registration, which usually 100 seconds earlier to finish the computation than mine. However, as the trajectory error shown above, mine ICP sometimes performs slightly better.

iv. Any reference you take

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

2. Questions (30%)

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

1. **Intrinsic Matrix**

Intrinsic matrix represents the internal parameters of the camera. It defines how 3D points in the camera's coordinate system are projected onto the 2D image plane. In simpler terms, it encodes the characteristics of the camera itself, such as the focal length and the optical center (the point where the camera is focusing).

Represented as:

$$\text{Intrinsic Matrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Where:

- fx and fy: The focal lengths of the camera in the x and y directions. Typically, fx=fy, but sometimes they can be different.
- cx and cy: The coordinates, which is the point on the image where the optical axis intersects the image plane.
- The bottom row is always [0,0,1] to allow for homogeneous coordinates

This matrix converts 3D points in the camera's coordinate space into 2D pixel coordinates on the image plane.

2. **Extrinsic Matrix**

Extrinsic matrix (a 4×4 matrix in my code) defines the position and orientation of the camera in the world. It describes the transformation of points from the world coordinate system into the camera's coordinate system (how the camera views the world).

$$\text{Extrinsic Matrix} = \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix}$$

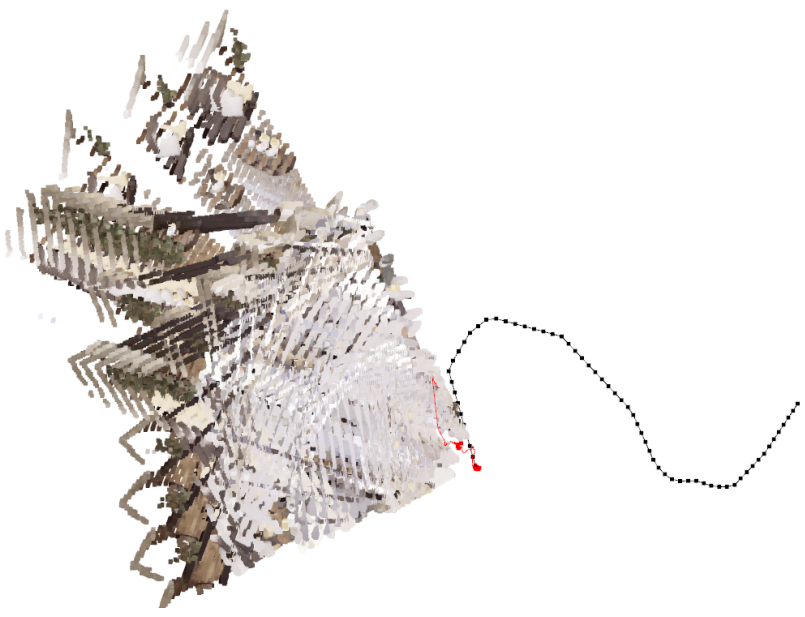
Where:

- R is the **rotation matrix** (a 3×3 matrix) that rotates points from the world coordinate system to align with the camera's orientation.
- T is the **translation vector** (a 3×1 column vector) that shifts the origin of the world coordinate system to the camera's position.
- The last row is [0,0,0,1] to allow for homogeneous coordinates and matrix multiplication.

The extrinsic matrix is used to describe where the camera is located and how it is oriented relative to the objects in the scene, determining how points in the world will appear to the camera.

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)**

My ICP/voxel_size=0.07/Floor 1



Here are the reasons lead to the result above:

1. **Poor Initialization**:

   Global registration provides an initial transformation that roughly aligns the source and target point clouds. Without global registration, **ICP starts from an identity transformation**, meaning it assumes the two point clouds are already aligned. If the point clouds have significant initial misalignment (in terms of rotation, translation, or scaling), ICP may fail to converge to the correct solution. Moreover, since ICP is a **local optimization algorithm**, meaning it will align the point clouds based on their nearest neighbors. If the initial alignment is poor (due to missing global registration), ICP can get stuck in a **local minimum**. It might find an alignment that minimizes the distance between some local sections of the point clouds but fails to find the correct global alignment.

2. **Convergence to Incorrect Alignment**:

   In the end, without a good initialization, ICP may converge to the wrong alignment, especially when the point clouds are in very different positions or orientations. ICP assumes that the source and target clouds are already close, so without an initial transformation, it may incorrectly align nearby points, leading to a bad result.

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

   1. **Threshold on Nearest Neighbors (`voxel_size`)**:

      When preparing the point clouds, it uses a **voxel size** to speed up the computation in ICP. However, instead of using the point clouds down by voxel size, in the visualization, the original point clouds are added.

   2. **`NearestNeighbors` in sklearn**

      `NearestNeighbors` is used to efficiently find the **closest point** in the target (reference) point cloud for each point in the source (moving) point cloud. ICP relies on aligning corresponding points, and this trick ensures that we can quickly and accurately find these correspondences.