Perception and Decision Making in Intelligent Systems
Homework 1:
BEV projection and 3D Scene Reconstruction
Announce: 9/19, Deadline: 10/17 23:59

Report

1. Implementation (40%)

Task1:

a. Code

There are 3 steps of my BEV projection. We want to project the point we picked from top view to front view.

First, we simply transform from top image coordinates to top camera coordinates. The below code is the same as multiplying the inverse of an intrinsic matrix with homogeneous uv. And the focal is computed by FOV:

$$f = \frac{512}{2 \cdot \tan\left(\frac{90}{2}\right)}$$

```
uv = np.array(points)
# top_image_coord to top_camera_coord
x = (uv[:, 0:1] - W*.5) * depth / focal
y = (uv[:, 1:] - H*.5) * depth / focal
z = np.full(x.shape, depth)
```

Second, we want to transform the above x, y, z from the top camera coordinates to world coordinates to front camera coordinates. And this is just a chain of matrix multiplication. So we can use 2 extrinsic matrices: the camera2world matrix at top view, and world2camera at front view to

get the results.

```
roll = -(np.pi / 2)
cos_val = np.cos(roll)
if np.isclose(cos_val, 0, atol=1e-10):
    cos_val = 0.0
sin_val = np.sin(roll)
if np.isclose(sin_val, 0, atol=1e-10):
    sin_val = 0.0

top_homo_c2w = np.eye(4)
top_homo_c2w[:3, :3] = np.array([[1, 0, 0], [0, cos_val, -sin_val], [0, sin_val, cos_val]]) # rotation
top_homo_c2w[:3, 3] = top_T = np.array([0, -2.5, 0]) # translation

front_homo_c2w = np.eye(4)
front_homo_c2w[:3, 3] = np.array([0, -1, 0])
front_homo_w2c = np.linalg.inv(front_homo_c2w)

c2c = np.matmul(front_homo_w2c, top_homo_c2w)
```

```
# top_camera_coord to front_camera_coord
c2c = np.tile(c2c, (uv.shape[0], 1, 1))
xyz = np.matmul(c2c, xyz)
```

Finally, transform from front camera coordinates to front image coordinate. The below code is the same as multiplying an intrinsic matrix with the above xyz.

```
# front_camera_coord to front_image_coord
u = xyz[:, 0] / xyz[:, 2] * focal + W*.5
v = xyz[:, 1] / xyz[:, 2] * focal + H*.5
new_pixels = np.round(np.concatenate([u, v], -1)).astype(int).tolist()

print("new_pixels", new_pixels)
return new_pixels
```

b. Result and Discussion

   i. Result of your projection (2 different pairs). Like the example result above.

   ii. Anything you want to discuss

   iii. Any reference you take

https://github.com/facebookresearch/consistent_depth (warping)

Task2:

a. Code

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

depth projection

First of all, I project each image to point clouds with their depth, applying the same strategy as BEV projection part to get the results. After get the position and rgb value for each 3d point, I turn them into open3d library point cloud objects for further applications. Note that I set a z threshold for each point. Because I thought that the depths farther to the camera are not that accurate.

```
def depth_image_to_point_cloud(rgb, depth, z_threshold):
    # TODO: Get point cloud from rgb and depth image
    H, W, focal, depth_scale = 512, 512, 256, 1000
    v, u = np.mgrid[0:H, 0:W]
    pcd = o3d.geometry.PointCloud()

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

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


    valid = points[:, 2] <= z_threshold
    points = points[valid, :]
    rgbs = rgbs[valid, :]

    pcd.points = o3d.utility.Vector3dVector(points)
    pcd.colors = o3d.utility.Vector3dVector(rgbs)
    # raise NotImplementedError
    return pcd
```

Voxelization & Global Registration

Do voxelization using open3d library functions, and get the fpfh feature for global registrations. And I set the voxel_size to be 0.07. As for global registration, I use the function from open3 library to done it.

```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))

    # raise NotImplementedError
    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
    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)
    )

    # raise NotImplementedError
    return result.transformation
```

Local ICP

For local icp from open3d library, I use the point2plane metric for error functions. And try to minimize the error metrics functions. Therefore, given source(frame i pcd) and target(frame i-1 pcd), we can get the transformation matrix from source to target.

```python
def local_icp_algorithm(source_down, target_down, trans_init, threshold):
    # TODO: Use Open3D ICP function to implement
    result = o3d.pipelines.registration.registration_icp(
        source_down, target_down, threshold, trans_init,
        o3d.pipelines.registration.TransformationEstimationPointToPlane(),
        # o3d.pipelines.registration.ICPConvergenceCriteria(max_iteration=10000)
    )

    # raise NotImplementedError
    return result.transformation
```

For local icp by my implementation, first, given an initial transformation matrix from global registration and a source(frame i pcd) and a

target(frame i-1 pcd, transform from source to target by the transformation matrix.

```
# Transform the source with current transformation matrix
source_points_homo = np.hstack((source_points, np.ones((source_points.shape[0], 1))))
source_points = np.dot(source_points_homo, trans_update.T)[:, :3]
```

Then, find the nearest neighbor for each point between source and target. Because the number of points in source and target may be different. So I sample the points in source and target by normal-space sampling. And we set the number of the sample points with the minimum number between source points and target points.

```
# sample from normal space
source_normal = np.asarray(source_down.normals)
target_noraml = np.asarray(target_down.normals)
num_sample_points = min(len(source_normal), len(target_noraml))
source_sampled_indices = sample(range(len(source_normal)), num_sample_points)
target_sampled_indices = sample(range(len(target_noraml)), num_sample_points)
# sampled_points
source_points = np.asarray(source_down.points)[source_sampled_indices]
target_points = np.asarray(target_down.points)[target_sampled_indices]
source_normal = source_normal[source_sampled_indices]
```

Find the nearest corresponding points between source and target by cKDTree from scipy library.

```
def nearest_neighbor(source, target):
    tree = cKDTree(target)
    distances, indices = tree.query(source, k=1)

    return distances, indices
```

After getting the nearest corresponding points between source and target, we fix the corresponding, and use this to find the final transformation for source and target. And we can use the function from the class slide to find the optimal transformation from source to target by minimize the error (point2point distance) because we have estimated data association.

```
# compute the optimal solution of the transformation from estimated data association
source_center = np.mean(source_points, axis=0)
target_center = np.mean(target_points[indices], axis=0)
source_center_diff = source_points - source_center # p' = {p - mean}
target_center_diff = target_points[indices] - target_center # x' = {x - x_mean}
W = np.matmul(target_center_diff.T, source_center_diff)
U, _, VT = np.linalg.svd(W)
optimal_rotation = U @ VT
if np.linalg.det(optimal_rotation) < 0: # reflection case
    VT[:, :] *= -1
    optimal_rotation = U @ VT

optimal_translation = target_center - np.dot(source_center, optimal_rotation.T)

trans_update = np.eye(4)
trans_update[:3, :3] = optimal_rotation
trans_update[:3, 3] = optimal_translation
# Update the current transformation
trans = trans_update @ trans
```

Iteratively update the transformation, if the iteration greater than max_iterations or error smaller than threshold, just break.

```
for iter in range(max_iters):
    error = np.mean((source_points - target_points[indices])**2) # want the error to be minimized
    print("error: ", error)

    # compute the optimal solution of the transformation from estimated data association
    source_center = np.mean(source_points, axis=0)
    target_center = np.mean(target_points[indices], axis=0)
    source_center_diff = source_points - source_center # p' = {p - mean}
    target_center_diff = target_points[indices] - target_center # x' = {x - x_mean}
    W = np.matmul(target_center_diff.T, source_center_diff)
    U, _, VT = np.linalg.svd(W)
    optimal_rotation = U @ VT
    # if np.linalg.det(optimal_rotation) < 0: # reflection case
    #     VT[:, :] *= -1
    #     optimal_rotation = U @ VT

    optimal_translation = target_center - np.dot(source_center, optimal_rotation.T)

    trans_update = np.eye(4)
    trans_update[:3, :3] = optimal_rotation
    trans_update[:3, 3] = optimal_translation
    # Update the current transformation
    trans = trans_update @ trans

    # Check for convergence
    if np.linalg.norm(trans_update - np.identity(4)) < convergence_threshold:
        break

    # Transform the source with current transformation matrix
    source_points_homo = np.hstack((source_points, np.ones((source_points.shape[0], 1))))
    source_points = np.dot(source_points_homo, trans_update.T)[:, :3]
```

Overall reconstruction

For each i, set frame i pcd as source, frame i-1 pcd as target (after voxelization), and get the transformation by local icap algorithm.

```
# init pred_cam_pos_list
pred_cam_pose = [np.identity(4)]
# get point cloud list
z_threshold = 80 / 255 * 10
pcd_list = get_point_cloud_list(args, z_threshold=z_threshold)
# preprocess
voxel_size = 0.07
pcd_down_list, pcd_fpfh_list = preprocess(pcd_list, voxel_size=voxel_size)
#
# breakpoint()
for i in range(1, len(pcd_list)):
    print(i)
    pcd_down_source, pcd_down_target = pcd_down_list[i], pcd_down_list[i-1]
    pcd_fpfh_source, pcd_fpfh_target = pcd_fpfh_list[i], pcd_fpfh_list[i-1]
    # global registration
    init_trans = execute_global_registration(pcd_down_source, pcd_down_target,
                                             pcd_fpfh_source, pcd_fpfh_target, voxel_size=voxel_size)
    # init_trans = execute_fast_global_registration(pcd_down_source, pcd_down_target,
    #                                               pcd_fpfh_source, pcd_fpfh_target, voxel_size=voxel_size)
    # local registration
    if args.version == 'open3d':
        trans = local_icp_algorithm(pcd_down_source, pcd_down_target, init_trans, threshold=voxel_size*0.4)
    elif args.version == 'my_icp':
        trans = my_local_icp_algorithm(pcd_down_source, pcd_down_target, init_trans,
                                       voxel_size=voxel_size, max_iters=100, convergence_threshold=1e-6)

    pred_cam_pose.append(pred_cam_pose[i-1] @ trans)

for i in range(len(pcd_list)):
    pcd_list[i].transform(pred_cam_pose[i])

return pcd_list, np.array(pred_cam_pose)
```

So we can get the transformation matrix for each pose to frame 0, by doing this.

```
pred_cam_pose.append(pred_cam_pose[i-1] @ trans)
```

Then, transform the pcd at each frame to align the pose with frame 0 pcd.

```
for i in range(len(pcd_list)):
    pcd_list[i].transform(pred_cam_pose[i])
```

Visualization

I visualize the results by applying functions from the open3d library.

```
# TODO: Visualize result
'''
Hint: Sould visualize
1. Reconstructed point cloud
2. Red line: estimated camera pose
3. Black line: ground truth camera pose
'''
# breakpoint()
edges = [[i, i+1] for i in range(gt_cam_pose.shape[0] - 1)]
gt_color = [[0, 0, 0] for i in range(len(edges))]
pred_color = [[1, 0, 0] for i in range(len(edges))]

gt_line_set = o3d.geometry.LineSet(
    points = o3d.utility.Vector3dVector(gt_cam_position),
    lines = o3d.utility.Vector2iVector(edges)
)
gt_line_set.colors = o3d.utility.Vector3dVector(gt_color)

pred_line_set = o3d.geometry.LineSet(
    points = o3d.utility.Vector3dVector(pred_cam_position),
    lines = o3d.utility.Vector2iVector(edges)
)
pred_line_set.colors = o3d.utility.Vector3dVector(pred_color)


o3d.visualization.draw_geometries(result_pcd+[gt_line_set, pred_line_set])
```

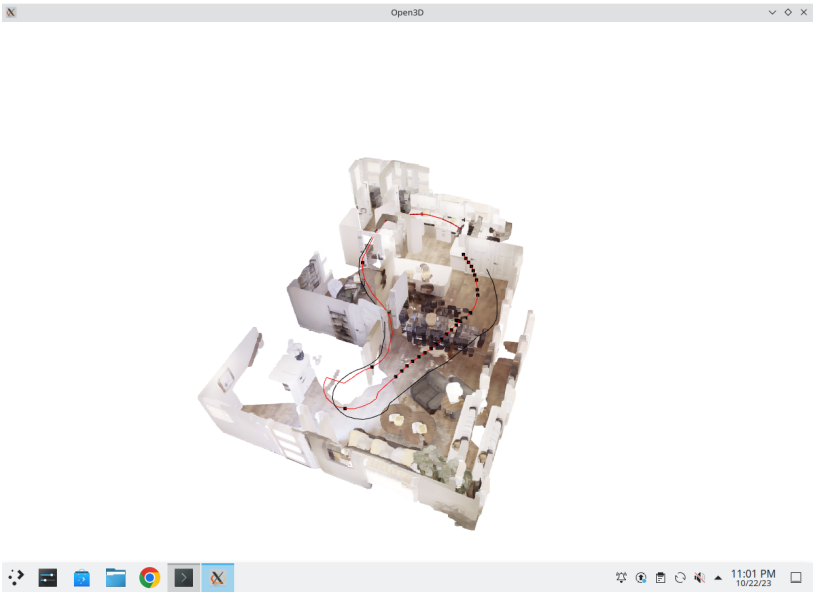
b. Result and Discussion

i. Result of your reconstruction (Floor 1 and Floor2, Both open3d implementation and your own implementation)**Make sure your execution time for each reconstruction is less than 5 mins, or you will get 0 point in this part.**

open3d floor1

Execution time: 77.428565 seconds;

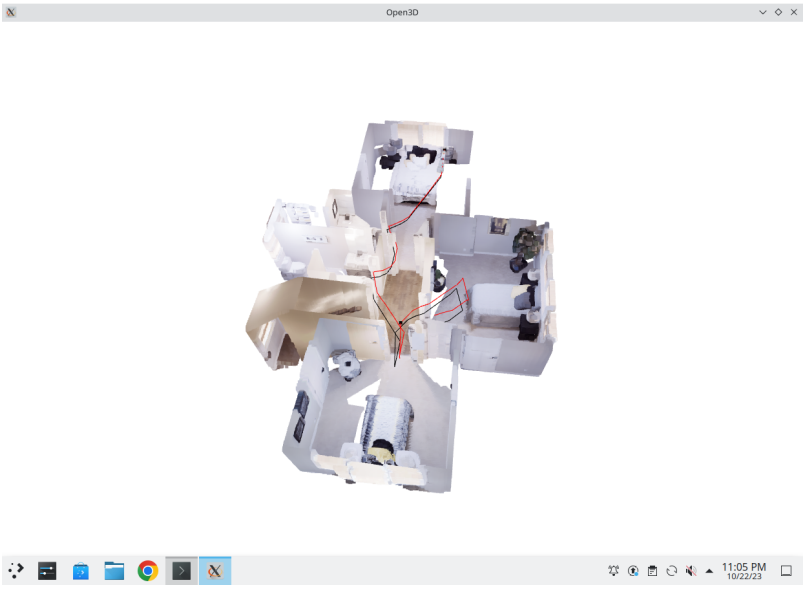
Mean L2 distance: 22.22302226092445



open3d floor 2

Execution time: 66.053789 seconds

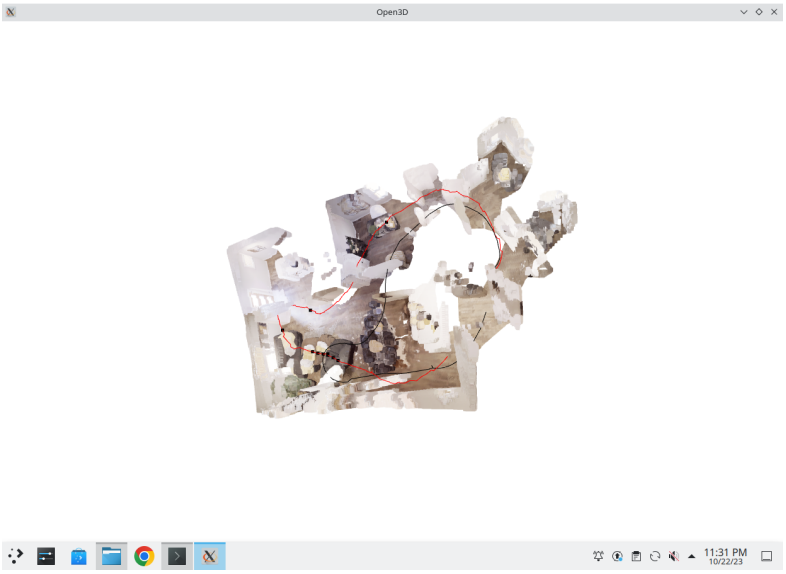
Mean L2 distance: 24.3769827845101



my_icp floor 1

Execution time: 56.082489 seconds

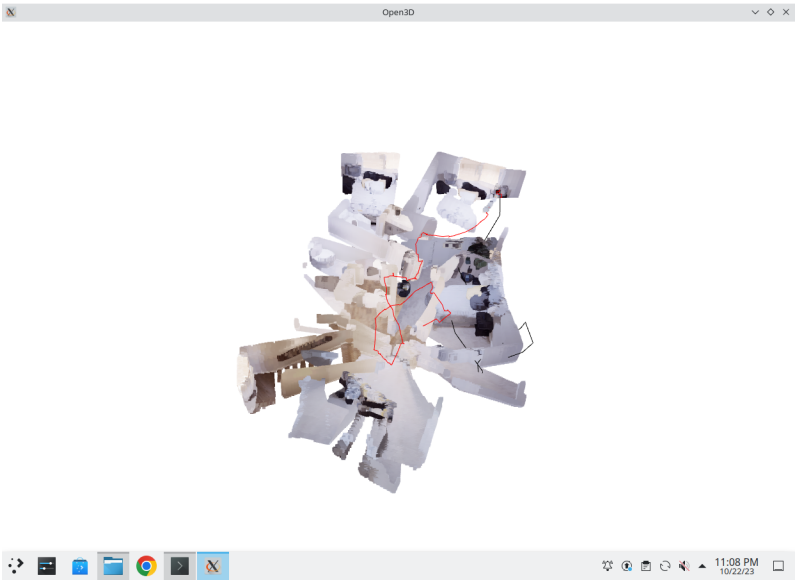
Mean L2 distance: 36.77614250156448



my_icp floor 2

Execution time: 42.591926 seconds

Mean L2 distance: 41.60129662859416



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

I have shown the results above.

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

Because I implement the local icp with point2point error metric instead of point2plane, the convergence should be faster. However, the result seems to be not as I imagined. I thought that the reason may be the error threshold or the max iterations, which open3d library adopts.

And I found that the longer trajectory I set, the error L2 distance will be higher than the shorter trajectory, because the error by local icp will gradually increase.

And for the second floor, because there are more rotations in the trajectory. So the reconstruction will be worse than floor 1.

iv. Any reference you take
http://www.open3d.org/docs/release/tutorial/pipelines/icp_registration.html
https://youtu.be/2hC9IG6MFD0

2. Questions (30%)

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

Extrinsic matrix is a 3*4 matrix: the left 3*3 part is the rotation matrix of camera pose. the left 3*1 part is the translation matrix. It can also be called camera to world matrix, which can be used to transform from camera to world. Intrinsic matrix is a 3*3 matrix, which saves focal length and the coordinates of principal points and skew factor. it can be used to transform from camera coordinates to image coordinates.

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)

Yes, the performance will be bad. Because in my implementation, the corresponding points are guessed by the global transformation. So the local icp will depend on the guess significantly.

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

After getting the optimal rotation by SVD, I do some tricks for the reflection case.

```
if np.linalg.det(optimal_rotation) < 0: # reflection case
    VT[:, :] *= -1
    optimal_rotation = U @ VT
```