Perception and Decision Making in Intelligent Systems

Homework 1: BEV projection and 3D Scene Reconstruction

109350008　張詠哲

1. Implementation

Task1

a. code

To project the pixel from one camera to another camera, we need to know the intrinsic matrix and the extrinsic matrix of both cameras. Here is how I get these 2 matrixes by given horizontal of view, Euler angle and position.

Intrinsic matrix

```python
def cal_intrinsic_matrix(self, hov, h, w) -> np.ndarray:
    """
    Calculate the intrinsic matrix by given horizontal Of View in degrees and resolution
    """

    vertical_fov = (h / w * hov) * np.pi / 180
    hov *= np.pi / 180

    f_x = (w / 2.0) / np.tan(hov / 2.0)
    f_y = (h / 2.0) / np.tan(vertical_fov / 2.0)

    K = np.array([[f_x, 0.0, w / 2.0], [0.0, f_y, h / 2.0], [0.0, 0.0, 1.0]])
    return K
```

By the basic geometry, we can know $\tan\left(\frac{\text{horizontal HOV}}{2}\right) = \frac{\text{Image width} / 2}{\text{focal len}_x}$. So on for y focal length. Then, we can calculate the intrinsic matrix of both camera.

Extrinsic matrix

```python
def axis_rotation_matrix(self, axis: str, angle: np.ndarray) -> np.ndarray:
    """
    Return the rotation matrix by given axis and eular angles.

    Args:
        axis: Axis label "X", "Y", or "Z".
        angle: numpy array of Euler angles in radians

    Returns:
        Rotation matrix (3 * 3).
    """

    cos = np.cos(angle)
    sin = np.sin(angle)
    one = np.ones_like(angle)
    zero = np.zeros_like(angle)

    if axis == "X":
        R_flat = (one, zero, zero, zero, cos, -sin, zero, sin, cos)
    elif axis == "Y":
        R_flat = (cos, zero, sin, zero, one, zero, -sin, zero, cos)
    elif axis == "Z":
        R_flat = (cos, -sin, zero, sin, cos, zero, zero, zero, one)

    return np.stack(R_flat, -1).reshape(angle.shape + (3, 3))
```

```python
def euler2matrix(self, euler_angles: np.ndarray, convention: str) -> np.ndarray:
    """
    Convert rotations given as Euler angles in radians to rotation matrix.

    Args:
        euler_angles: Euler angles (in radians)
        convention: Convention string of three uppercase letters from
            {"X", "Y", and "Z"}.

    Returns:
        Rotation matrix (3 * 3).
    """

    if euler_angles.ndim == 0 or euler_angles.shape[-1] != 3:
        raise ValueError("Invalid euler angles.")
    for letter in convention:
        if letter not in ("X", "Y", "Z"):
            raise ValueError(f"Only XYZ")

    matrices = [
        self.axis_rotation_matrix(c, e)
        for c, e in zip(convention, np.moveaxis(euler_angles, -1, 0))
    ]

    return np.matmul(np.matmul(matrices[0], matrices[1]), matrices[2])
```

```python
def cal_transformation_matrix(self, p:list, euler_angle:list):
    '''
    Calculate the transformation matrix by given camera position and rotation (in radians)

    Return:
        transformation matrix (4*4)
    '''
    H = np.eye(4)
    euler_angle = np.array(euler_angle)
    p = np.array(p)

    H[:3, :3] = self.euler2matrix(euler_angle, "XYZ")
    H[:3, -1] = p

    return H
```

By the following equation, we can get the rotation matrix from given 3 axes Euler angle. And the translation just uses the position given by SPEC.

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

Calculate the projection

After we get the intrinsic matrix and extrinsic matrix, we can calculate the point position in world coordinate from pixel coordinate. Then, project the point position to front camera pixel coordinate by related transformation matrix. And the related rotation matrix can be calculated by $\mathrm{R_1 R_{related}} = \mathrm{R_2} \Rightarrow \mathrm{R_{related}} = \mathrm{R_1^{-1} R_2}$.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \underbrace{\mathbf{K}}_{\text{intrinsics}} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\text{projection}} \left[\begin{array}{c|c} I & t \\ \hline \mathbf{0} & 1 \end{array}\right] \times \left[\begin{array}{c|c} R & \mathbf{0} \\ \hline \mathbf{0} & 1 \end{array}\right] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

```python
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 ###
    cam1_H = self.cal_transformation_matrix(p = [0, 1, 0], euler_angle = [0, 0, 0])
    cam2_H = self.cal_transformation_matrix(p = [0, 2.5, 0], euler_angle = [-np.pi/2, 0, 0])

    cam1_intrin = self.cal_intrinsic_matrix(fov, 512, 512)
    cam2_intrin = self.cal_intrinsic_matrix(fov, 512, 512)

    related_H = np.linalg.inv(cam1_H) @ cam2_H # R_1 * R_related = R_2 => R_related = inverse(R_1) * R_2

    bev_points = np.array(self.points)
    bev_points_h = np.hstack((bev_points, np.ones((len(bev_points), 1))))
    bev_points_in_cam = np.linalg.inv(cam2_intrin) @ (bev_points_h.T * -cam2_H[1, -1]) # depth is negative to camera
    bev_points_in_cam = np.vstack((bev_points_in_cam, np.ones((1, len(bev_points)))))

    front_points_in_cam = related_H[:3, :4] @ bev_points_in_cam
    front_points_in_pixel = cam1_intrin @ front_points_in_cam
    front_points_in_pixel = np.round(front_points_in_pixel[:2] / front_points_in_pixel[2]).astype(int)

    new_pixels = [[front_points_in_pixel[0, i], front_points_in_pixel[1, i]] for i in range(front_points_in_pixel.shape[1])]
    print("new_pixels: ", new_pixels)

    return new_pixels
```

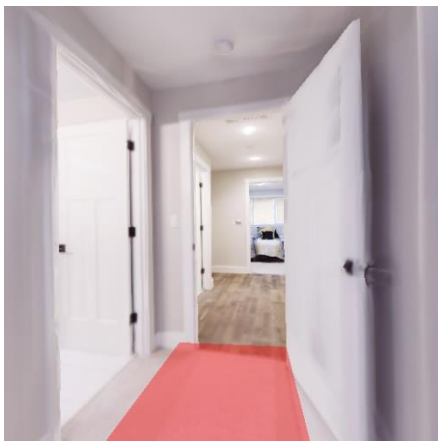
b. Result and Discussion

i. Result of your projection

Bev1.png

Front1.png

Bev1.png



Front.png

ii. Anything you want to share

I found that if I choose a too large space that the image can't contain the projection area, it will project to the wrong place. Like following figure.

iii. Any reference you take

https://photo.stackexchange.com/questions/97213/finding-focal-length-from-image-size-and-fov

https://pytorch3d.org/

Task2

a. Code

To complete this part, we need to reconstruct each frame. Then, align every point cloud into one scene.

Reconstruct

Use the same method as the bev projection to get the position in world coordinate. Then, turn the points into open3d library point cloud. I also set a depth threshold to filter the points that it's too far.

```
def depth_image_to_point_cloud(rgb, depth, z_threshold) -> o3d.geometry.PointCloud:
    '''
    Reconstruct the point cloud by given rgb and depth image
    '''
    # TODO: Get point cloud from rgb and depth image

    # common setup
    K = cal_intrinsic_matrix(90, 512, 512)
    pcd = o3d.geometry.PointCloud()

    # point cloud
    ymap = np.array([[j for i in range(depth.shape[0])] for j in range(depth.shape[1])])
    xmap = np.array([[i for i in range(depth.shape[0])] for j in range(depth.shape[1])])

    if len(depth.shape) > 2:
        depth = depth[:, :, 0]
    mask_depth = depth <= z_threshold
    choose = mask_depth.flatten().nonzero()[0].astype(np.uint32)
    if len(choose) < 1:
        assert "out of z theshold"

    depth_masked = depth.flatten()[choose][:, np.newaxis].astype(np.float32)
    ymap_masked = ymap.flatten()[choose][:, np.newaxis].astype(np.float32)
    xmap_masked = xmap.flatten()[choose][:, np.newaxis].astype(np.float32)

    pt2 = -depth_masked / 255 * 10 # turn to m
    cam_cx, cam_cy = K[0][2], K[1][2]
    cam_fx, cam_fy = K[0][0], K[1][1]
    pt0 = (xmap_masked - cam_cx) * pt2 / cam_fx
    pt1 = (ymap_masked - cam_cy) * pt2 / cam_fy

    points = np.concatenate((pt0, pt1, pt2), axis=1)
    pcd.points = o3d.utility.Vector3dVector(points)

    # color
    rgbs = (rgb[:, :, [2, 1, 0]].astype(np.float32) / 255).reshape(-1, 3)
    rgbs = rgbs[choose, :]
    pcd.colors = o3d.utility.Vector3dVector(rgbs)

    return pcd
```

ICP

This step we need to calculate the point normal and FPFH feature for global registration to get the initial transformation. Then, use the local ICP to get the refinement result.

- **Point cloud preprocessing and global registration**

I use the open3d library to complete this part.

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

    radius_normal = voxel_size * 2
    pcd_down.estimate_normals(o3d.geometry.KDTreeSearchParamHybrid(radius=radius_normal, max_nn=30))

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

```
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.transformation
```

- **Local ICP**

This part has 2 implementations. One use the open3d, another use the self-implemented ICP. Both implementations are point-to-plane ICP. Then, we can align the previous point cloud frame and the next point cloud frame.

```
def local_icp_algorithm(source_down, target_down, trans_init, voxel_size):
    # TODO: Use Open3D ICP function to implement
    threshold = voxel_size * 0.3
    result = o3d.pipelines.registration.registration_icp(
        source_down, target_down, threshold, trans_init,
        o3d.pipelines.registration.TransformationEstimationPointToPlane()
    )

    return result.transformation
```

Open3d library implementation

For self-implemented ICP, I take the point-to-plane method. Here is the step by step explanation.

First, I transform the source points by given initial transformation which is calculated by the global registration.

```
source_points = np.asarray(source_down.points)
target_points = np.asarray(target_down.points)
target_normals = np.asarray(target_down.normals)

trans = trans_init.copy()
prev_error = float('inf')
current_max_iter = 0
for iteration in range(max_iters):
    current_max_iter = iteration
    source_transformed = (trans[:3, :3] @ source_points.T).T + trans[:3, 3]
```

Then, for each iteration, we need to calculate and filter the correspondence points for each source or target point. Here I use the cKDTree to choose the nearest neighbor for correspondence point.

```
def nearest_neighbor(source, target):
    '''
    Finding the nearest points from given target points
    '''
    target_kdtree = cKDTree(target)
    distances, indices = target_kdtree.query(source, k=1)
    return distances, indices
```

```
# finding the nearest neighbor of each points
distances, indices = nearest_neighbor(source_transformed, target_points)
valid_mask = distances < voxel_size * 0.3
valid_source = source_transformed[valid_mask]
valid_target = target_points[indices[valid_mask]]
valid_normals = target_normals[indices[valid_mask]]
```

After that, we need to minimize the point-to-plane error (1) and (2). We can take it as the least square problem to get the $\triangle\theta$ and the $\triangle t$ we want. Note that for each iteration, we can assume the updating rotation $\triangle R$ is very small. So, we can take $\triangle R \approx I + S$, the S here is a skew-symmetric matrix to approximate the small delta rotation Euler angles θ on each rotation axes.

(1) $E = \sum \left(n_i^T(Rp_i + T - q_i)\right)^2$

(2)
$$
\begin{aligned}
E_i &= \boldsymbol{n}_i^T(\boldsymbol{I} + \boldsymbol{S})\boldsymbol{p}_i + \boldsymbol{n}_i^T\boldsymbol{T} - \boldsymbol{n}_i^T\boldsymbol{q}_i \\
&= \boldsymbol{n}_i^T\boldsymbol{p}_i + \boldsymbol{n}_i^T(-\boldsymbol{p}_i \times \boldsymbol{\theta}) + \boldsymbol{n}_i^T\boldsymbol{T} - \boldsymbol{n}_i^T\boldsymbol{q}_i \\
&= (\boldsymbol{p}_i \times \boldsymbol{n}_i)^T\boldsymbol{\theta} + \boldsymbol{n}_i^T\boldsymbol{T} - \boldsymbol{n}_i^T(\boldsymbol{q}_i - \boldsymbol{p}_i)
\end{aligned}
$$

```
# calculate point2plane
A = np.zeros((len(valid_source), 6))
b = np.zeros((len(valid_source), 1))

for i in range(len(valid_source)):
    source_point = valid_source[i]
    target_point = valid_target[i]
    normal = valid_normals[i]

    # normal * (R * source_point + t - target_point)
    cross_prod = np.cross(source_point, normal)
    A[i, :3] = cross_prod
    A[i, 3:] = normal
    b[i] = normal.dot(target_point - source_point)

# solve the LS
delta = np.linalg.lstsq(A, b, rcond=None)[0].flatten()
```

Finally, we update the previous transformation and calculate the error. Also check if the transformation converge.

```
# update
trans = delta_transformation @ trans

# calculate error
mean_error = np.mean(distances[valid_mask]**2)
# print("current error: ", mean_error)

if np.abs(prev_error - mean_error) < convergence_threshold:
    break
prev_error = mean_error
```

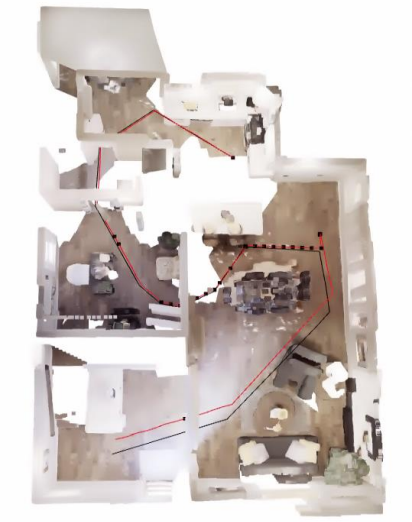
- **visualization**

I use the open3d library to visualize the predicted and ground-truth trajectory.

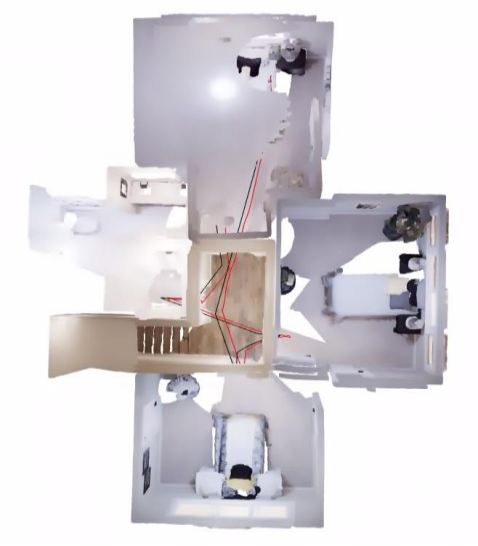
```
def draw_camera_trajectory(result_pcd, ground_truth_path, estimate_pose, num = None):
    if num is None:
        ground_truth = np.load(ground_truth_path)
    else:
        ground_truth = np.load(ground_truth_path)[:num, ...]

    ground_truth_pose = np.tile(np.eye(4), (ground_truth.shape[0], 1, 1))
    ground_truth_pose[:, 0:3, 0:3] = R.from_quat(ground_truth[:, 3:7]).as_matrix() # quaternion to rotation martix
    ground_truth_pose[:, 0:3, 3] = ground_truth[:, 0:3]

    ground_truth_pose = np.tile(np.linalg.inv(ground_truth_pose[0, ...]), (ground_truth_pose.shape[0], 1, 1)) @ ground_truth_pose
    ground_truth_pose[:, 0, 2] = -ground_truth_pose[:, 0, 2]
    ground_truth_pose[:, 2, 0] = -ground_truth_pose[:, 2, 0]

    ground_truth_position = ground_truth_pose[:, 0:3, 3]
    ground_truth_position[:, 0] = -ground_truth_position[:, 0]
    ground_truth_position[:, 2] = -ground_truth_position[:, 2]
    ground_truth_position = ground_truth_position

    estimate_position = estimate_pose[:, 0:3, 3]

    # draw path
    line = [[i, i+1] for i in range(ground_truth_position.shape[0] - 1)]
    ground_truth_color = [[0, 0, 0] for _ in range(len(line))]
    estimate_color = [[1, 0, 0] for _ in range(len(line))]

    ground_truth_line = o3d.geometry.LineSet(
        points = o3d.utility.Vector3dVector(ground_truth_position),
        lines = o3d.utility.Vector2iVector(line)
    )
    ground_truth_line.colors = o3d.utility.Vector3dVector(ground_truth_color)

    estimate_line = o3d.geometry.LineSet(
        points = o3d.utility.Vector3dVector(estimate_position),
        lines = o3d.utility.Vector2iVector(line)
    )
    estimate_line.colors = o3d.utility.Vector3dVector(estimate_color)

    o3d.visualization.draw_geometries(result_pcd+[ground_truth_line, estimate_line])
```

b. Result and Discussion

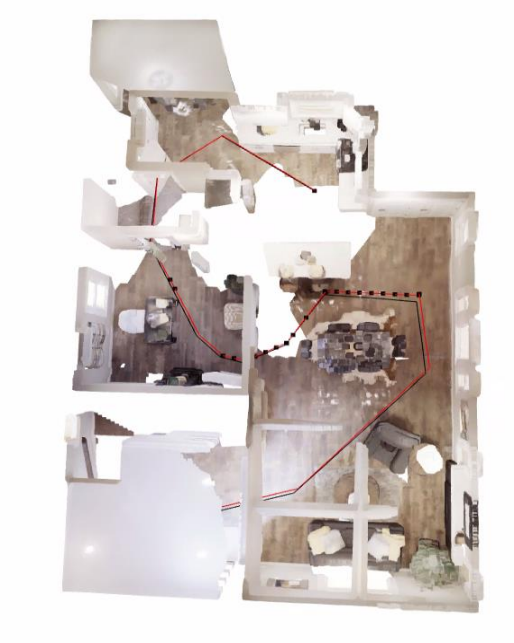
i. Result of your reconstruction



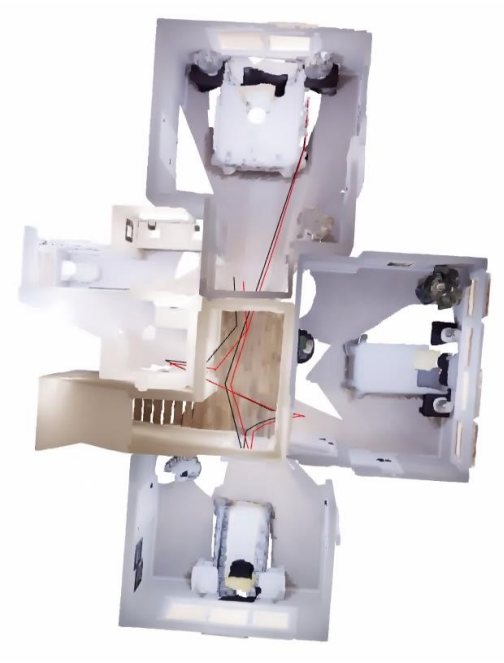
Open3d, time: 59.9925 (s)



Open3d, time: 59.6441 (s)



My_icp, time: 141.3234 (s)



My_icp, time: 123.8883 (s)

ii. Mean L2 distance

Open3d, first floor	3.8022
Open3d, second floor	3.1889
My_icp, first floor	1.8849
My_icp, second floor	2.2747

iii. Anything you want to share

I originally implement the point-to-point ICP, then I found that the result was quite terrible and slowly generated. So, I implement the point-to-plane ICP instead. Thought the generated time is not as fast as the open3d, the mean L2 distance is better than it. I think the threshold has the deterministic role, cuz I set voxel_size * 0.3 for the open3d, while voxel_size * 0.5 for my icp instead. Even it only has 0.2 difference, it can cause a huge performance gap.

iv. Any reference

https://zhuanlan.zhihu.com/p/385414929
https://blog.csdn.net/keineahnung2345/article/details/120880598
https://blog.csdn.net/taifyang/article/details/129774493
https://www.youtube.com/watch?v=2hC9IG6MFD0
https://www.open3d.org/docs/release/tutorial/pipelines/icp_registration.html

2. Question

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

Intrinsic matrix (3*3 matrix) is the internal parameters of a camera to map the 3D coordinates of a point in the camera coordinate system to its 2D image coordinates. While extrinsic matrix (4*4 matrix) is the external parameters of a camera, it describes the position and orientation of the camera in the world coordinate system.

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. Though ICP without global registration can still work, the generally results in worse performance, especially when the initial alignment between the source and target point clouds is not close enough. That's because ICP is a local optimization method, if the initial guess is far off, ICP will often converge to a local minimum solution.

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

- I implement the point-to-plane ICP, instead of point-to-point ICP to make the converge speed faster and more accurate.
- I set the threshold larger to make the result more accurate.