Physical AI Homework 1:

BEV projection and 3D Scene Reconstruction

111550132 張家睿

1. Implementation

Task1

a. Code

```python
import cv2
import numpy as np

points = []

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 if points else []

    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 ###
        def rotation_matrix(theta, phi, gamma):
            theta, phi, gamma = np.deg2rad(theta), np.deg2rad(phi), np.deg2rad(gamma)

            cx, sx = np.cos(theta), np.sin(theta)
            Rx = np.array([[1, 0,    0],
                           [0, cx, -sx],
                           [0, sx,  cx]])

            cy, sy = np.cos(phi),   np.sin(phi)
            Ry = np.array([[ cy, 0, sy],
                           [  0, 1,  0],
                           [-sy, 0, cy]])

            cz, sz = np.cos(gamma), np.sin(gamma)
            Rz = np.array([[cz, -sz, 0],
                           [sz,  cz, 0],
                           [ 0,   0, 1]])

            return Rz @ Ry @ Rx

        # 1. intrinsics
        cx = self.width / 2.0
        cy = self.height / 2.0
        f  = (self.width / 2.0) / np.tan(np.deg2rad(fov) / 2.0)

        # 2. extrinsics
        Center_front = np.array([0.0, 1.0, 0.0])
        Rotation_front = np.array([[1.0, 0.0, 0.0],
                                   [0.0, 1.0, 0.0],
                                   [0.0, 0.0, 1.0]])

        Center_BEV = np.array([0.0, 2.5, 0.0])
        Rotation_BEV = rotation_matrix(theta, phi, gamma) # BEV coordinates -> World coordinates

        # Transfer each point
        new_pixels = []

        for (u, v) in self.points:
            # world direction
            x_cam, y_cam = (u - cx) / f, -(v - cy) / f
            d_world = (Rotation_BEV @ [x_cam, y_cam, 1.0])

            # 3. BEV coordinates -> World coordinates
            if abs(d_world[1]) < 1e-6: # Remove ray parallel to ground(y=0)
                continue
            t = (0.0 - Center_BEV[1]) / d_world[1]
            if t < 0:  # intersect behind cam
                continue
            P_world = Center_BEV + t * d_world
```

```python
            # 4. World coordinates -> Front coordinates
            Pc = (Rotation_front.T @ (P_world - Center_front))
            if Pc[2] < 0: # intersect behind cam
                continue

            u_ = (f * (Pc[0] / Pc[2])) + cx
            v_ = cy - (f * (Pc[1] / Pc[2]))
            u_ = int(np.clip(u_, 0, self.width - 1))
            v_ = int(np.clip(v_, 0, self.height - 1))

            new_pixels.append([u_, v_])

        return new_pixels

    def show_image(self, new_pixels, img_name='projection.png', color=(0, 0, 255), alpha=0.4):
        """
            Show the projection result and fill the selected area on perspective(front) view image.
        """

        new_image = cv2.fillPoly(
            self.image.copy(), [np.array(new_pixels)], color)
        new_image = cv2.addWeighted(
            new_image, alpha, self.image, (1 - alpha), 0)

        cv2.imshow(
            f'Top to front view projection {img_name}', new_image)
        cv2.imwrite(img_name, new_image)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

        return new_image


def click_event(event, x, y, flags, params):
    # checking for left mouse clicks
    if event == cv2.EVENT_LBUTTONDOWN:

        print(x, ' ', y)
        points.append([x, y])
        font = cv2.FONT_HERSHEY_SIMPLEX
        # cv2.putText(img, str(x) + ',' + str(y), (x+5, y+5), font, 0.5, (0, 0, 255), 1)
        cv2.circle(img, (x, y), 3, (0, 0, 255), -1)
        cv2.imshow('image', img)

    # checking for right mouse clicks
    if event == cv2.EVENT_RBUTTONDOWN:

        print(x, ' ', y)
        font = cv2.FONT_HERSHEY_SIMPLEX
        b = img[y, x, 0]
        g = img[y, x, 1]
        r = img[y, x, 2]
        # cv2.putText(img, str(b) + ',' + str(g) + ',' + str(r), (x, y), font, 1, (255, 255, 0), 2)
        cv2.imshow('image', img)


if __name__ == "__main__":

    pitch_ang = 90

    front_rgb = "bev_data/front1.png"
    top_rgb = "bev_data/bev1.png"

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

    projection = Projection(front_rgb, points)
    new_pixels = projection.top_to_front(theta=pitch_ang)
    projection.show_image(new_pixels)
```

__init__(): Save the points to self.points.

top_to_front(): Map user-clicked BEV pixels to their corresponding pixels in the front-view image. First, derive intrinsics from image size and FOV. Second, define camera poses with rotation_matrix(), a function that build rotation from Euler angles. Third, for each point, transfer the BEV pixels to BEV coordinates with (u - cx) / f and -(v - cy) / f. Transfer BEV coordinates to world coordinates with a ray Center_BEV + t * d_world with ground (y = 0), Remove the point parallel to ground and behind camera. Forth, transfer world point to front coordinates with Rotation_front.T @ (P_world - Center_front). Transfer front coordinates to front pixels f * (Pc.x / Pc.z) + cx and cy - f * (Pc.y / Pc.z), then clip the result and save the point to array that will return.

__name__ == "__main__": Change pitch_ang to 90.

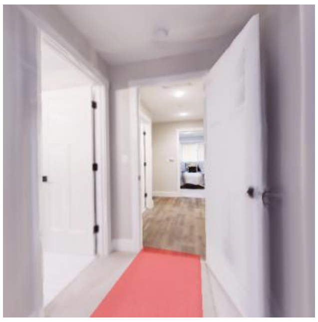
b. Result and Discussion

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


bev1.png


front1.png


b ev2.png


front2.png

ii. Anything you want to discuss

Will there be any deviation when approaching the edge? Why?

The projection of the table is offset.

Yes, when the deviation is amplified near the image edges because the viewing angle becomes steep there; the denominator dy becomes small, leading to a magnification effect.

iii. Any reference you take

None

Task2

a. Code

```python
import numpy as np
import open3d as o3d
import argparse
from pathlib import Path
import os
from tqdm import tqdm
import copy


def map_png(root, subdir):
    return {int(p.stem): str(p)
            for p in (Path(root)/subdir).glob("*.png")
            if p.stem.isdigit()}


def depth_image_to_point_cloud(rgb, depth):
    # TODO: Get point cloud from rgb and depth image
    h, w, _ = rgb.shape
    f = w / (2 * np.tan(np.pi * 0.5 / 2))
    cx, cy = w / 2, h / 2

    camera_intrinsic = o3d.camera.PinholeCameraIntrinsic()
    camera_intrinsic.set_intrinsics(w, h, f, f, cx, cy)

    color_image = o3d.geometry.Image(rgb)
    depth_meters = depth.astype(np.float32) / 255.0 * 10.0 # load.py save like this
    depth_image = o3d.geometry.Image(depth_meters)

    rgbd = o3d.geometry.RGBDImage.create_from_color_and_depth(
        color_image, depth_image,
        depth_scale=1.0,  # Scaled to m
        depth_trunc=10.0,  # Truncate at 10m
        convert_rgb_to_intensity=False
    )

    pcd = o3d.geometry.PointCloud.create_from_rgbd_image(
        rgbd, camera_intrinsic
    )

    return pcd


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)
    return pcd_down


def get_FPFH(pcd, voxel_size):
    # Compute FPFH feature
    pcd.estimate_normals(o3d.geometry.KDTreeSearchParamHybrid(radius=voxel_size*2, max_nn=30))
    fpfh = o3d.pipelines.registration.compute_fpfh_feature(
        pcd,
        o3d.geometry.KDTreeSearchParamHybrid(radius=voxel_size*5, max_nn=100)
    )
    return 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),
        4, [
            o3d.pipelines.registration.CorrespondenceCheckerBasedOnEdgeLength(0.9),
            o3d.pipelines.registration.CorrespondenceCheckerBasedOnDistance(distance_threshold)
        ], o3d.pipelines.registration.RANSACConvergenceCriteria(100000, 0.999))
    return result


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.transformation,
        o3d.pipelines.registration.TransformationEstimationPointToPoint()
    )
    return result


def my_local_icp_algorithm(source_down, target_down, trans_init, threshold):
    # TODO: Write your own ICP function
    MAX_ITERS = 30
    TOLERANCE = 1e-6

    # Convert to NumPy
    T = np.array(trans_init.transformation, dtype=np.float64)
    source = np.asarray(source_down.points, dtype=np.float64)
    target = np.asarray(target_down.points, dtype=np.float64)

    result = o3d.pipelines.registration.RegistrationResult()

    # Build KD-Tree on target for fast 1-NN search
    target_pcd = o3d.geometry.PointCloud()
    target_pcd.points = o3d.utility.Vector3dVector(target)
    kdtree = o3d.geometry.KDTreeFlann(target_pcd)

    # Homogeneous source for fast transforms
    source_h = np.c_[source, np.ones(len(source))]
    prev_rmse = np.inf
    thr2 = threshold * threshold

    for i in range(MAX_ITERS):
        # 1. Transform source by current T
        source_trans = (T @ source_h.T).T[:, :3]

        # 2. Build correspondences via 1-NN within threshold
        source_corr = []
        target_corr = []
        for p in source_trans:
            _, idx, d2 = kdtree.search_knn_vector_3d(p, 1)
            if len(idx) == 1 and d2[0] <= thr2:
                source_corr.append(p)
                target_corr.append(target[idx[0]])

        # Not enough pairs to solve a stable rigid transform
        if len(source_corr) < 6:
            break

        A = np.asarray(source_corr)  # transformed source
        B = np.asarray(target_corr)  # matched target

        # 3. Solve rigid transform using SVD
        muA, muB = A.mean(axis=0), B.mean(axis=0)
        AA, BB = A - muA, B - muB
        U, S, Vt = np.linalg.svd(AA.T @ BB)
        R = Vt.T @ U.T
        # Reflection fix to ensure det(R)=+1
        if np.linalg.det(R) < 0:
            Vt[-1, :] *= -1
            R = Vt.T @ U.T
        t = muB - R @ muA

        # 4. Update transform
        T_update = np.eye(4)
        T_update[:3, :3] = R
        T_update[:3, 3] = t
        T = T_update @ T

        # 5. Compute RMSE and check convergence
        A_aligned = (A @ R.T) + t
        rmse = float(np.sqrt(np.mean(np.sum((A_aligned - B) ** 2, axis=1))))
        if abs(prev_rmse - rmse) < TOLERANCE:
            prev_rmse = rmse
            break
        prev_rmse = rmse

    result.transformation = T
    result.fitness = float((len(source_corr) / max(1, len(source))))
    result.inlier_rmse = float(prev_rmse)
    return result


def reconstruct(args):
# TODO: Return results
    """
    For example:
        ...
        args.version == 'open3d':
            trans = local_icp_algorithm()
        args.version == 'my_icp':
            trans = my_local_icp_algorithm()
        ...
    """
    VOXEL_SIZE = 0.1

    # Prepare img
    rgb_map = map_png(args.data_root, "rgb")
    depth_map = map_png(args.data_root, "depth")
    ids = sorted(rgb_map.keys() & depth_map.keys())
    rgb_sequence = [rgb_map[i] for i in ids]
    depth_sequence = [depth_map[i] for i in ids]

    result_pcd = o3d.geometry.PointCloud()
    pred_cam_pos = [np.eye(4)]
    accumulated_pose = np.eye(4)

    for i in tqdm(range(len(rgb_sequence)-1)):
        # RGB-D -> Point Clouds
        source_rgb = np.asarray(o3d.io.read_image(rgb_sequence[i+1]))
        source_depth = np.asarray(o3d.io.read_image(depth_sequence[i+1]))
        source_pcd = depth_image_to_point_cloud(source_rgb, source_depth)

        target_rgb = np.asarray(o3d.io.read_image(rgb_sequence[i]))
        target_depth = np.asarray(o3d.io.read_image(depth_sequence[i]))
        target_pcd = depth_image_to_point_cloud(target_rgb, target_depth)

        # Downsample and FPFH
        source_pcd = preprocess_point_cloud(source_pcd, VOXEL_SIZE)
        features_source = get_FPFH(source_pcd, VOXEL_SIZE)

        target_pcd = preprocess_point_cloud(target_pcd, VOXEL_SIZE)
        features_target = get_FPFH(target_pcd, VOXEL_SIZE)
```

```python
        # RANSAC
        result_ransac = execute_global_registration(
            source_pcd, target_pcd, features_source,
            features_target, VOXEL_SIZE
        )

        # ICP
        if args.version == 'open3d':
            trans = local_icp_algorithm(source_pcd, target_pcd, result_ransac, VOXEL_SIZE)
        else:
            trans = my_local_icp_algorithm(source_pcd, target_pcd, result_ransac, VOXEL_SIZE)

        # Accumulate transformation in global coordinate frame
        accumulated_pose = accumulated_pose @ trans.transformation

        # Transform source point cloud to global frame and merge
        pcd_source_global = copy.deepcopy(source_pcd)
        pcd_source_global.transform(accumulated_pose) # Cam coordinate -> World coordinate
        result_pcd += pcd_source_global
        # preprocess_point_cloud(result_pcd, VOXEL_SIZE)

        # Store camera pose
        pred_cam_pos.append(accumulated_pose.copy())

    return result_pcd, np.array(pred_cam_pos)


def remove_ceiling(pcd, starting_height, offset=0.2):
    # Remove ceiling points from pcd, so that we can see the room
    points = np.asarray(pcd.points)
    colors = np.asarray(pcd.colors)

    ceiling_threshold = starting_height - offset
    mask = points[:, 1] > ceiling_threshold

    filtered_pcd = o3d.geometry.PointCloud()
    filtered_pcd.points = o3d.utility.Vector3dVector(points[mask])
    filtered_pcd.colors = o3d.utility.Vector3dVector(colors[mask])

    return filtered_pcd

if __name__ == '__main__':
    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/"

    # TODO: Output result point cloud and estimated camera pose
    '''
    Hint: Follow the steps on the spec
    '''
    result_pcd, pred_cam_pos = reconstruct(args)
    gt_pos = np.load(os.path.join(args.data_root, 'GT_pose.npy'))
    gt_pos[:, 2] *= -1 # Reflect to align reconstruct

    # TODO: Calculate and print L2 distance
    '''
    Hint: Mean L2 distance = mean(norm(ground truth - estimated camera trajectory))
    '''
    pred_positions = np.array([pose[:3, 3] for pose in pred_cam_pos])
    gt_positions = gt_pos[:, :3]

    # Align prediction to GT and Compute L2 distanc
    pred_positions_align = pred_positions + gt_positions[0] - pred_positions[0]
    L2 = np.mean(np.linalg.norm(pred_positions_align - gt_positions, axis=1))
    print("Mean L2 distance: ", L2)

    # TODO: Visualize result
    '''
    Hint: Sould visualize
    1. Reconstructed point cloud
    2. Red line: estimated camera pose
    3. Black line: ground truth camera pose
    '''
    # 1. Reconstructed point cloud
    alignment_transform = np.eye(4)
    alignment_transform[:3, 3] = gt_positions[0] - pred_positions[0]
    result_pcd.transform(alignment_transform)

    # Remove ceiling points
    starting_height = gt_positions[0, 1] # Y-up
    result_pcd = remove_ceiling(result_pcd, starting_height)

    # 2. Red line: estimated camera pose
    est_traj_lines = o3d.geometry.LineSet()
    est_traj_lines.points = o3d.utility.Vector3dVector(pred_positions_align)
    est_traj_lines.lines = o3d.utility.Vector2iVector([[i, i + 1] for i in range(len(pred_positions)-1)])
    est_traj_lines.colors = o3d.utility.Vector3dVector([[1, 0, 0] for i in range(len(pred_positions)-1)])

    # 3. Black line: ground truth camera pose
    gt_traj_lines = o3d.geometry.LineSet()
    gt_traj_lines.points = o3d.utility.Vector3dVector(gt_positions)
    gt_traj_lines.lines = o3d.utility.Vector2iVector([[i, i + 1] for i in range(len(pred_positions)-1)])
    gt_traj_lines.colors = o3d.utility.Vector3dVector([[0, 0, 0] for i in range(len(pred_positions)-1)])

    o3d.visualization.draw_geometries(
        [result_pcd, est_traj_lines, gt_traj_lines],
        lookat=[0, 0, 0],
        up=[0, -1, 0],
        front=[-1, 0, 0],
        zoom=1
    )
```

map_png(): Scans a folder contains *.png files. Returns a dict mapping the integer frame id to its filepath. This lets you align RGB and depth by frame number.

depth_image_to_point_cloud(): Turns one RGB–Depth pair into a 3D point cloud using Open3D's RGBD pipeline. Builds pinhole intrinsic from image size and a 90° FOV; converts depth (0–255) linearly to meters (0–10 m), then creates an RGBD Image and a PointCloud. Assumes the /255*10 scaling puts depth in meters (depth_scale=1.0 m).

preprocess_point_cloud(): Voxel-downsamples the input point cloud with the given voxel_size to reduce point count and speed up registration steps.

get_FPFH(): Estimates normals (radius = 2×voxel_size, max_nn=30) and computes FPFH descriptors (radius = 5×voxel_size, max_nn=100) on the point cloud. Returns an Open3D Feature used for global matching, so that we can use RANSAC in execute_global_registration().

execute_global_registration(): Performs RANSAC global registration using the source & target FPFH features. Uses edge-length and distance checkers, ransac_n=4, and a correspondence threshold=1.5×voxel_size; returns a coarse RegistrationResult transform.

local_icp_algorithm(): Refines alignment with Open3D ICP (point-to-point) starting from the RANSAC initial transform and a given distance threshold. Returns an Open3D RegistrationResult with the refined transform.

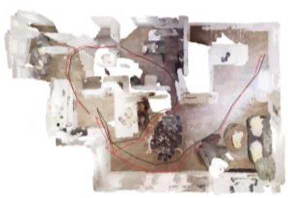
my_local_icp_algorithm(): Custom point-to-point ICP. Takes downsampled source_down & target_down, an initial transform trans_init, and a max correspondence distance threshold. Internally it: (1) Resolves the initial 4×4 T. (2) Builds a KD-Tree on the target to speed up.(Complexity per iteration is O(NlogM)) (3) Iterates up to 30 times: transform source by T, for each transformed source point, find its 1-NN in target and keep pairs within threshold. If fewer than 6 pairs, stop. (4) Solve the rigid motion with SVD (Umeyama, no scale): R = Vt^T @ U^T and t = μ_B - R @ μ_A. (5) Form an incremental transform T_{update}=[[R, t], [0, 1]] and left-multiply T = T_update @ T. (6) Compute RMSE = $\sqrt{mean\|RA+t-B\|^2}$ and stop when improvement < 1e-6. Returns an Open3D-style RegistrationResult with transformation = T, fitness = (# inlier matches / # source points), and inlier_rmse = final RMSE. Complexity per iteration is O(NlogM) due to KD-Tree.

reconstruct(): Full incremental reconstruction over the RGB-D sequence. Set VOXEL_SIZE = 0.1. For each consecutive pair: makes point clouds, downsamples, computes FPFH, runs RANSAC for a coarse pose, refines with ICP (Open3D or custom), chains the relative pose into a global accumulated_pose, transforms the source cloud into a global point cloud, and records the camera poses. Returns result_pcd & pred_cam_pos.

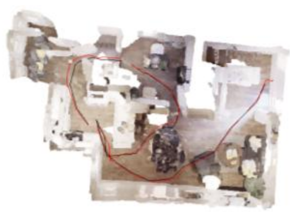
remove_ceiling(): Filters the point cloud along the Y axis to remove ceiling points for clearer room visualization. Uses a threshold set as starting_height - offset and keeps points with y > threshold, returning the filtered cloud.

b. Result and Discussion

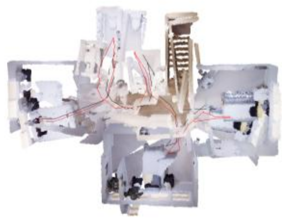
i. Result of your reconstruction (Floor 1 and Floor2, Both open3d implementation and your own implementation) Make sure your execution time for each reconstruction less than 5 mins, or you will get 0 point in this part.



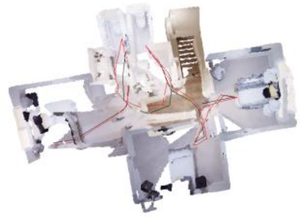
1F/open3d



1F/my_own



2F/open3d

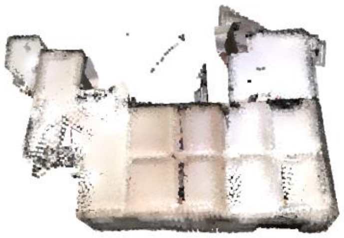


2F/my_own

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

	open3d	my_local_icp_algorithm
1F Mean L2 distance	0.3563992774349988	0.3428335922227827
2F Mean L2 distance	0.24607247247257658	0.37385491764591783

iii. Anything you want to discuss, such as comparing the performance of two implementations. (Discovered in the process) Why does the result look weird in visualization, but it has normal L2 distance?



Because the point cloud includes the ceiling, I need to remove the points that are too high.

iv. Any reference you take
https://www.open3d.org/docs/release/tutorial/pipelines/global_registration.html

2. Questions

a. What's the meaning of extrinsic matrix and intrinsic matrix?
Extrinsic matrix: camera pose, rotation R and translation t, that maps world coordinates to the camera frame.

$$\begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = R * \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} + t$$

Intrinsic matrix: camera's internal parameters that map camera-frame 3D points to pixels—mainly focal length f_x, f_y. and principal point (c_x, c_y).

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 0 \end{bmatrix}$$

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 result is horrible. If you remove RANSAC and only use ICP, it is easy to get stuck in local extreme values or diverge directly, causing the entire map to be messed up.

c. Describe the tricks you apply to improve your ICP alignment.
(1) Strong initialization from FPFH+RANSAC, which puts ICP inside its small basin of convergence.
(2) Distance-gated 1-NN correspondences via KD-Tree, which both speed up matching and reject outliers beyond a threshold.
(3) Stable SVD rigid solve plus RMSE early stopping, ensuring numerically sound updates and halting once improvements are negligible to avoid overfitting.