BEV projection and 3D Scene Reconstruction

I. Introduction

In this work, leverage the Habitat simulation environment in combination with the Replica dataset to execute image projection and 3D scene reconstruction. In the bird-eye-view projection task, involves the selection of points within the Bird's Eye View (BEV) image, typically a top-down view. These selected points define an enclosed area, which we then project from the BEV image onto the perspective (front view) image. This projection process facilitates a transition from a top-down view to a more conventional front view, providing valuable insights into the scene. In the 3D scene reconstruction task, control an agent walking through the whole scene, meanwhile saving observation data of the agent, then transform the observations into point cloud data, align their coordinate system by using the ICP algorithm. In the end, Visualize the trajectory and show the difference between the trajectory estimate by the ICP algorithm and the ground truth trajectory.

II. BEV projection

Code:

```
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
```

```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 ###
        np.set_printoptions(precision=3, suppress=True)

        BEV_pixels = []
        BEV_points = []
        FV_points  = []
        FV_pixels  = []
        new_pixels = []

        # camera instric
        principal_point = [int(self.height/2),int(self.height/2)]
        print("-----------")
        print("principal point:")
        print(principal_point)
        focal_length = np.tan(np.deg2rad(fov/2))*256
        print("-----------")
        print("focal length:")
        print(focal_length)

        #Convert to principal points coordinate
        for point in self.points:
            pixel_x,pixel_y = principal_point[0]-
point[0],principal_point[1]-point[1]
            BEV_pixels.append([pixel_x,pixel_y,1])
        print("-----------")
        print("BEV_pixels:")
        print(BEV_pixels)

        #2D to 3D
        #2.5
        for BEV_pixel in BEV_pixels:
            BEV_point =
[2.5*BEV_pixel[0]/focal_length,2.5*BEV_pixel[1]/focal_length,1*2.5]
            BEV_points.append(BEV_point)
        BEV_points= np.array(BEV_points)
        print("-----------")
        print("BEV_points:")
        print(BEV_points)
        row_of_ones = np.ones((BEV_points.shape[0],1))
        BEV_points = np.column_stack((BEV_points,row_of_ones))

        # Computing Transform Matrix
        rotation_matrix, _ = cv2.Rodrigues(np.array([np.deg2rad(theta),
np.deg2rad(phi), np.deg2rad(gamma)], dtype=np.float32))
        transformation_matrix = np.eye(4)
        transformation_matrix[:3,:3] = rotation_matrix
        transformation_matrix[:3, 3] = [dx,dy,dz]
        print("-----------")
        print("Transform Matrix:")
        print(transformation_matrix)

        #Transform to another camera
        FV_points = np.dot(transformation_matrix, BEV_points.T).T
        FV_points = np.delete(FV_points, -1, axis=1)
        print("-----------")
        print("FV_points:")
        print(FV_points)

        #3D to 2D
        for FV_point in FV_points:
            FV_point[0],FV_point[1],FV_point[2] =
FV_point[0]/FV_point[2]*focal_length,FV_point[1]/FV_point[2]*focal_leng
th,1
            FV_pixels.append([FV_point[0],FV_point[1],FV_point[2]])
        print("-----------")
        print("FV_pixels")
        print(FV_pixels)

        for FV_pixel in FV_pixels:
            new_pixel_x,new_pixel_y = principal_point[0]-
int(FV_pixel[0]),principal_point[1]-int(FV_pixel[1])
            new_pixels.append([new_pixel_x,new_pixel_y])
        print("-----------")
        print("new pixel")
        for new_pixel in new_pixels:
            print(new_pixel)

        return new_pixels

    def show_image(self, new_pixels, img_name='projection_1.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
#end of class

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__":

    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)
    projection = Projection(front_rgb, points)
    new_pixels = projection.top_to_front(theta=90,dy=1.5)
    projection.show_image(new_pixels)
    cv2.destroyAllWindows()
```

To begin, an empty list called 'points' is created to store the points selected in the bird's-eye-view picture. Following that, a class named 'Projection' is constructed to manage various functions, including 'top_to_front()', 'show_image()', and an initializer designed to set the class's variable values:

(1) __init__(): This initializer is responsible for setting the 'image' path using the 'image_path' variable by assigning 'image_path' to 'self.image' and extracting the 'height,' 'width,' and 'channels' values from the image's shape. Additionally, it assigns the points stored in the 'points' list to 'self.points.

(2) top_to_front(): This function project the top view pixels to the front view pixels, and return new pixels on perspective(front) view image. This function

contain the following processes:

(a) Transfer the bird eye view pixel to 2D points relatively to the camera's principal points.
(b) Transfer the 2D bird eye view points into 3D bird eye view points in 3D space.
(c) Compute the transformation matrix by the poses of two camera.
(d) Transforming the 3D bird eye view points to 3D front viewpoints by multiplying the transformation matrix.
(e) Transfer the 3D front view points to the 2D front viewpoints relatively to the camera's principal points.
(f) Transfer the 2D front viewpoints to 2D front view image pixel that is " new_pixels " in the code.

Declare several lists representing different elements, including the bird's eye view image pixels, bird's eye view 2D points, front view 3D points, front view 2D points, and the final pixels that will be returned from the function. Proceed by configuring the camera intrinsic parameters. The principal point is positioned at the image's center: (256, 256). The focal length is determined using the following formula:

$$tan\left(\frac{fov}{2}\right) \cdot 0.5 \cdot 512$$

Converting the points to the coordinates relative to the camera's principal points by the formula:

$$(u', v') = (256 - u, 256 - v)$$

Converting the 2D points to 3D points by:

$$x = u \cdot \frac{depthoftheimage}{focallength}, y = u \cdot \frac{depthoftheimage}{focallength}$$

$$z = 1 \cdot depthoftheimage$$

And note that in this work, the depth of the image is set at 2.5 .

Proceed to compute the transformation matrix. For the rotation matrix, we employ the cv2 function to calculate the rotation matrix. The function takes inputs in radians for the roll, pitch, and yaw angles and returns an array with a data type of numpy.float32. In the case of the translation matrix, it is represented as [dx, dy, dz]. The transformation matrix is constructed by combining the rotation matrix and the translation matrix.

Transforming the point from the bird eye view camera to front view camera by multiplying the transformation matrix. Then transform the 3D front view point to 2D front points by:

$$(u, v) = \left(\frac{x}{z \cdot focallength}, \frac{y}{z \cdot focallength}\right)$$

In the end transform the 2D front points to front view image pixel by:

$$(pixelx, pixely) = (256 - u, 256 - v)$$

(3) show_image(): This function displays the image, setting the window name to 'img_name.' The function concludes and closes the window when the user presses any key, utilizing cv2.waitKey(0).

Finally, we implement a function called 'click_event.' This function enables users to click on the image, resulting in red dots being drawn at the clicked points

Result and Discussion:

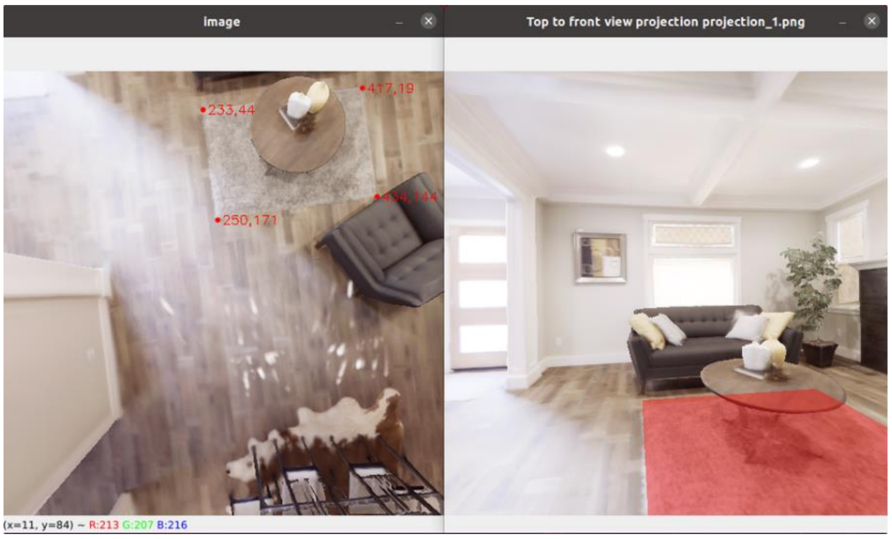


image
417,19
233,44
434,144
250,171
(x=11, y=84) ~ R:213 G:207 B:216
Top to front view projection projection_1.png
Fig(1). The result of the first pair of images

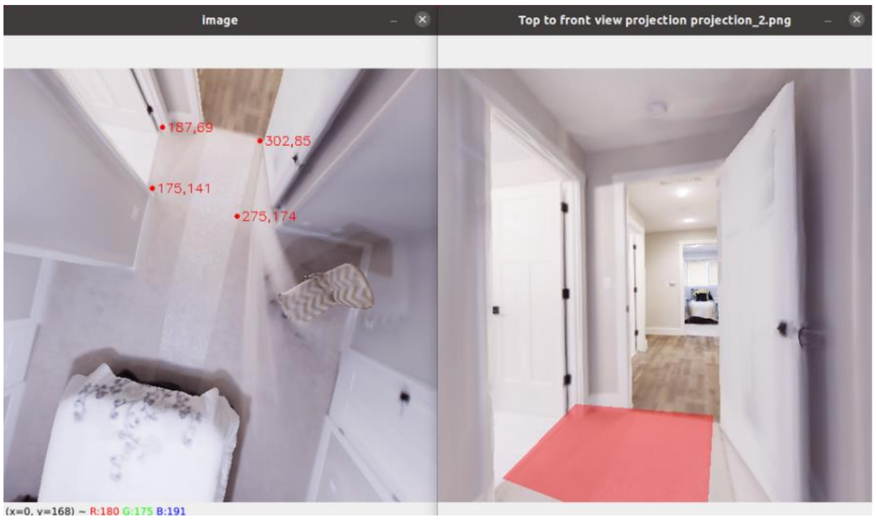


image
Top to front view projection projection_2.png
187,69
302,85
175,141
275,174
(x=0, y=168) ~ R:180 G:175 B:191

Fig(2). The result of the second pair of images

This task involves the translation of a user-selected region in a bird's-eye-view image to the corresponding front view image using Python libraries OpenCV and NumPy. We begin by addressing camera intrinsic parameters, with a specific focus on the focal length and depth, as they have a significant impact on the projected region. The transformation of pixels into camera points involves a series of operations, each resulting in a different projection image.

To perform this transformation, we first utilize the pinhole model to convert 2D points into a 3D space. Then, we transform these 3D points into another view's coordinate system using the calculated transformation. Afterward, we convert these 3D points back into 2D points, allowing us to visualize the projection of the bird's-eye-view points. This comprehensive process is a crucial step before constructing a 3D point cloud scene map. It's essential to understand how to transfer points from one viewpoint to another effectively.

III. 3D Reconstruction

Code:

```
import numpy as np
import open3d as o3d
import argparse
import os
import copy
import time
import math
```

```python
from sklearn.neighbors import NearestNeighbors

def depth_image_to_point_cloud(rgb, depth):

    # Camera instrinsics
    principal_point = [256, 256]
    focal_length = np.tan(np.deg2rad(90/2)) * 256

    # Read the rgb images and the depth images
    image = np.asarray(o3d.io.read_image(rgb))
    depth = np.asarray(o3d.io.read_image(depth))
    rgb_colors = (image / 255).reshape(-1, 3)
    pixel_coords = np.indices((512, 512)).reshape(2, -1)
    d = depth.reshape(-1) / 1000 # real depth unit is m
    f = focal_length

    x = (principal_point[0] - pixel_coords[1]) * d / f
    y = (principal_point[1] - pixel_coords[0]) * d / f
    z = d

    xyz_points = np.vstack([x, y, z]).T

    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(xyz_points)
    pcd.colors = o3d.utility.Vector3dVector(rgb_colors)

    #calculate camera point
    camera_pcd = o3d.geometry.PointCloud()
    camera_pcd.points.append([0,0,0])
    camera_pcd.colors.append([1,0,0]) # set the color to red

    return pcd, camera_pcd

def preprocess_point_cloud(pcd, voxel_size):

    pcd_down = pcd.voxel_down_sample(voxel_size=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 draw_registration_result(source, target, transformation):
    source_temp = copy.deepcopy(source)
    target_temp = copy.deepcopy(target)
    source_temp.paint_uniform_color([1, 0.706, 0])
    target_temp.paint_uniform_color([0, 0.651, 0.929])
    source_temp.transform(transformation)
    o3d.visualization.draw_geometries([source_temp, target_temp])

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_matchin
g(
        source_down, target_down, source_fpfh, target_fpfh, True,
        distance_threshold,
        o3d.pipelines.registration.TransformationEstimationPointToPoint
(False),
        3, [
            o3d.pipelines.registration.CorrespondenceCheckerBasedOnEdge
Length(
                0.9),
            o3d.pipelines.registration.CorrespondenceCheckerBasedOnDist
ance(
                distance_threshold)
        ], o3d.pipelines.registration.RANSACConvergenceCriteria(100000,
0.999))
    return result

def execute_fast_global_registration(source_down, target_down,
source_fpfh,
                                     target_fpfh, voxel_size):
    distance_threshold = voxel_size * 0.5
    result =
o3d.pipelines.registration.registration_fgr_based_on_feature_matching(
        source_down, target_down, source_fpfh, target_fpfh,
        o3d.pipelines.registration.FastGlobalRegistrationOption(
            maximum_correspondence_distance=distance_threshold))
    return result

def local_icp_algorithm(source_down, target_down, trans_init,
voxel_size):
    distance_threshold = voxel_size * 0.4
    result = o3d.pipelines.registration.registration_icp(
        source_down, target_down, distance_threshold, trans_init,
        o3d.pipelines.registration.TransformationEstimationPointToPlane
())
    return result

'''
Start Implementing tht local_icp_algorithm
'''


def best_fit_transform(source_points, target_points):

    # Calculates the least-squares best-fit transform between
corresponding 3D points source -> target
    assert np.shape(source_points) == np.shape(target_points)

    # number of dimensions
    m = np.shape(source_points)[1]

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

    # rotation matrix
    W = np.dot(BB.T, AA)
    U, S, VT = np.linalg.svd(W)
    rotation = np.dot(U, VT)

    # special reflection case
    if np.linalg.det(rotation) < 0:
        VT[m-1,:] *= -1
        rotation = np.dot(U, VT)

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

    # homogeneous transformation
    transformation = np.identity(m+1)
    transformation[:m, :m],transformation[:m, m] = rotation,translation

    return transformation, rotation, translation

def nearest_neighbor(source_points, target_points):

    # Find the nearest (Euclidean) neighbor in target points for each
point in source points
    neigh = NearestNeighbors(n_neighbors=1)
    neigh.fit(target_points)

    # Euclidean distance (errors) of the nearest neighbor and the
nearest neighbor
    distances, indices = neigh.kneighbors(source_points,
return_distance=True)
    valid = distances < np.median(distances)*0.8
    return distances[valid].ravel(),
indices[valid].ravel(),valid.ravel()
```

```
def implemented_local_icp_algorithm(source_down, target_down,
trans_init=None,  max_iterations=100000 ,tolerance=0.000005):
    # the user may tuns the parameter

    source_points = np.asarray(source_down.points)
    target_points = np.asarray(target_down.points)
    m = np.shape(source_points)[1]

    # make points homogeneous, copy them so as to maintain the
originals
    src = np.ones((m+1,source_points.shape[0]))
    dst = np.ones((m+1,target_points.shape[0]))
    src[:m,:] = np.copy(source_points.T)
    dst[:m,:] = np.copy(target_points.T)

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

    # main part
    for i in range(max_iterations):
        # find the nearest neighbours between the current source and
destination points
        distances, indices, valid = nearest_neighbor(src[0:m,:].T,
dst[0:m,:].T)

        # compute the transformation between the current source and
nearest destination points
        transformation,_,_ = best_fit_transform(src[0:m,valid].T,
dst[0:m,indices].T)

        # update the current source
        src = np.dot(transformation, src)

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

    # calculcate final tranformation
    transformation,_,_ = best_fit_transform(source_points,
src[0:m,:].T)

    return transformation

def read_pose(args):

    # setting the file path
    file_path = args.data_root + "/GT_pose.npy"
    poses = np.load(file_path)

    # transforming to real points with unit:m
    if args.floor == 1 :
        # mm to m / rw => 10 / 0.25
        x = -poses[:, 0]/ 40
        y = poses[:, 1]/ 40
        z = -poses[:, 2]/ 40
    elif args.floor == 2:
        # 10 / rw => 10 / 0.25
        x = -poses[:, 0] / 40 -0.00582
        y = ( poses[:, 1] / 40 ) - 0.07313
        z = -poses[:, 2]/ 40 -0.03

    xyz_points = np.vstack([x,y,z]).T

    gt_pose_pcd = o3d.geometry.PointCloud()
    gt_pose_pcd.points = o3d.utility.Vector3dVector(xyz_points)
    gt_pose_pcd.paint_uniform_color([0,0,0]) #set color to black

    gt_lines = []
    for i in range(len(xyz_points) - 1):
        gt_lines.append([i, i + 1])
```

```python
    gt_line_set = o3d.geometry.LineSet()
    gt_line_set.points = o3d.utility.Vector3dVector(xyz_points)
    gt_line_set.lines = o3d.utility.Vector2iVector(gt_lines)

    return gt_pose_pcd, gt_line_set

def reconstruct(args):

    # config
    voxel_size = 0.00225
    point_cloud = o3d.geometry.PointCloud()
    estimate_camera_cloud = o3d.geometry.PointCloud()
    data_folder_path = args.data_root
    rgb_images = os.listdir(os.path.join(data_folder_path, "rgb/"))
    depth_images = os.listdir(os.path.join(data_folder_path, "depth/"))
    if args.floor == 1:
        print("Start reconstructing the first floor...")
    if args.floor == 2:
        print("Start reconstructing the second floor...")
    reconstruct_start = time.time()
    print("Numbers of images is %d" % len(rgb_images))

    # temps
    pcd = []
    camera_pcd = []
    pcd_down = []
    pcd_transformed = [] # contain the pcd transformed to the main axis
    fpfh = []

    for i in range(1,len(rgb_images)):
    # for i in range(1,10):
        if i == 1:
            # target point cloud
            rgb_principal = data_folder_path + "/rgb/%d.png" % i
            depth_principal = data_folder_path + "/depth/%d.png" % i
            print("Principal picture is set as picture %d." % i)
            pcd.append(depth_image_to_point_cloud(rgb_principal,
depth_principal)[0]) # pcd[i-1]
```

```
            camera_pcd.append(depth_image_to_point_cloud(rgb_principal,
depth_principal)[1])
            pcd_down.append(preprocess_point_cloud(pcd[i-
1],voxel_size)[0]) # pcd_down[i-1]
            fpfh.append(preprocess_point_cloud(pcd[i-1],voxel_size)[1])
# fpfh[i-1]
            pcd_transformed.append(pcd_down[i-1])
        else:
            # get source point cloud
            rgb= data_folder_path + "/rgb/%d.png" % i
            depth = data_folder_path + "/depth/%d.png" % i
            print("------------------------------------")
            print("dealing with picture %d..." % i)
            pcd.append(depth_image_to_point_cloud(rgb, depth)[0]) #
pcd[i-1]
            camera_pcd.append(depth_image_to_point_cloud(rgb,
depth)[1])
            pcd_down.append(preprocess_point_cloud(pcd[i-
1],voxel_size)[0]) # pcd_down[i-1]
            fpfh.append(preprocess_point_cloud(pcd[i-1],voxel_size)[1])
# target_fpfh[i-1]

            # Global registration
            global_start = time.time()
            result_ransac =
execute_fast_global_registration(pcd_down[i-1], pcd_transformed[i-2],
                                                              fpfh[i-1],
fpfh[i-2], voxel_size)
            print("Global registeration took %.3f sec." % (time.time()
- global_start))

            # ICP
            icp_start = time.time()
            if args.version == 'open3d':
                print("Using open3d's icp...")
                result_icp = local_icp_algorithm(pcd_down[i-1],
pcd_transformed[i-2],
                                                         result_ransac.transfor
mation, voxel_size)
                transformation = result_icp.transformation #
transformation of i to i-1
            elif args.version == 'my_icp':
                print("Using the implemented icp...")
                result_icp =
implemented_local_icp_algorithm(pcd_down[i-1], pcd_transformed[i-2],
                                                         result_ransac.transfor
mation)
                transformation = result_icp # transformation of i to i-
1
                # draw_registration_result(pcd_down[i-
1],pcd_transformed[i-2],transformation)
            print("ICP took %.3f sec.\n" % (time.time() - icp_start))

            # transformation to 1st camera axis
            # transformation = result_icp.transformation #
transformation of i to i-1
            pcd_transformed.append(pcd_down[i-
1].transform(transformation))
            camera_pcd[i-1] = camera_pcd[i-1].transform(transformation)

    for pcd in pcd_transformed:
        point_cloud += pcd

    for pcd in camera_pcd:
        estimate_camera_cloud += pcd

    estimate_lines = []
    for i in range(len(estimate_camera_cloud.points) - 1):
        estimate_lines.append([i, i + 1])
    estimate_line_set = o3d.geometry.LineSet()
    estimate_line_set.points =
o3d.utility.Vector3dVector(estimate_camera_cloud.points)
    estimate_line_set.lines =
o3d.utility.Vector2iVector(estimate_lines)
    estimate_line_set.paint_uniform_color([1, 0, 0])
    # filter the ceiling
    xyz_points = np.asarray(point_cloud.points)
    colors = np.asarray(point_cloud.colors)
    if args.floor == 1:
        threshold_y = 0.0135
    elif args.floor == 2:
        if args.version == 'open3d':
            threshold_y = 0.0115
        elif args.version == 'my_icp':
            threshold_y = 0.009
    filtered_xyz_points = xyz_points[xyz_points[:, 1] <= threshold_y]
    filtered_colors = colors[xyz_points[:, 1] <= threshold_y]
    point_cloud.points =
o3d.utility.Vector3dVector(filtered_xyz_points)
    point_cloud.colors = o3d.utility.Vector3dVector(filtered_colors)

    gt_pose_cloud, gt_line_set = read_pose(args)
    print("------------------------------------")
    print("3D reconstruction took %.3f sec." % (time.time() -
reconstruct_start))
    return point_cloud, gt_pose_cloud, gt_line_set,
estimate_camera_cloud, estimate_line_set

def calculate_mean_l2_distance(gt_pos_pcd, estimate_camera_cloud):
    sum = 0
    for i in range(len(estimate_camera_cloud.points)):
        x = gt_pos_pcd.points[i][0]-estimate_camera_cloud.points[i][0]
        y = gt_pos_pcd.points[i][1]-estimate_camera_cloud.points[i][1]
        z = gt_pos_pcd.points[i][2]-estimate_camera_cloud.points[i][2]
        sum += math.sqrt(x**2 + y**2 + z**2)
    return sum/len(estimate_camera_cloud.points)

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')
```

```
    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/"

    # Output result point cloud and estimated camera pose
    result_pcd, gt_pos_pcd, line_set, estimate_camera_cloud,
estimate_line_set= reconstruct(args)

    # Calculate and print L2 distance
    print("Mean L2 distance:", calculate_mean_l2_distance(gt_pos_pcd,
estimate_camera_cloud))

    # Visualize result
    o3d.visualization.draw_geometries([result_pcd,gt_pos_pcd,line_set,e
stimate_camera_cloud, estimate_line_set])
    print("3D reconstruction finished.")
```

For task 2's implementation, define the following functions to finish the task.

(1) depth_image_to_point_cloud(rgb,depth): This function converts a pair of RGB and depth images into a 3D point cloud object. It takes two arguments: an RGB image and a depth image. The camera intrinsic parameters are defined as follows: the principal point, representing the principal camera pixel coordinates, is set to [256, 256], and the focal length is calculated using the formula provided in the code, assuming a 90-degree field of view (FOV).

The function reads the RGB and depth images, and similar to Task 1, it projects 2D points into 3D points in space using the pinhole camera model. The x-coordinate is transformed to points relative to the camera's principal point, multiplied by the depth, and divided by the focal length. The same transformation is applied to the y-coordinate. The z-coordinate is set to the depth value.

This function also returns a camera point point cloud for pose estimation. According to the pinhole model, the principal point of the camera in 3D space is located at (0, 0, 0). As per the specifications, the color of the camera point cloud is set to red, which is represented as [1, 0, 0] in the code.

(2) preprocess_point_cloud(pcd, voxel_size): This function preprocesses the point cloud by performing a down sampling process to reduce the number of points, resulting in decreased memory usage and runtime costs. It also calculates the 3D Local Feature Descriptor, Fast Point Feature Histograms (FPFH), for use in the alignment process.

The function takes two arguments: a point cloud object and a parameter 'voxel_size.' Using the Open3D library, the point cloud is down sampled. The radius is set to twice the voxel size to estimate the normal vectors of the point cloud, and the radius feature is set to five times the voxel size. The function then computes the FPFH of the point clouds.

Finally, it returns the downsampled point cloud and the FPFH, which are used in subsequent processes.

(3) draw_registration_result(source, target, transformation): This function is used to visualize the alignment results between the source and target point clouds. During development, it assists in assessing the quality of an individual alignment process. This function is reference at the website:
http://www.open3d.org/docs/release/tutorial/pipelines/global_registration.html

The function takes three arguments: the source point cloud, the target point cloud, and the transformation, which is a 4x4 homogeneous matrix representing the transformation between the two point clouds in 3D space. It sets the color of one point cloud to blue and the color of the other point cloud to yellow. Then, it uses the Open3D library to visualize the alignment result.

(4) execute_global_registration(source_down, target_down, source_fpfh, target_fpfh, voxel_size): This function performs the global registration of two point clouds as first rough alignment for the initialization. The argument are the source point cloud, the target point cloud, the source point cloud’s FPFH and the target point cloud’s FPFH and the voxel size as the function’s parameter. Define the threshold distance as 1.5 times of the voxel_size. This function is reference at the website:
http://www.open3d.org/docs/release/tutorial/pipelines/global_registration.html

Returns the result for the following process such as ICP for local registration enhancement.

(5) execute_fast_global_registration(source_down, target_down, source_fpfh, target_fpfh, voxel_size): This function performs the same function as the

above one, but this function is the faster version of the above, which is also offered by the open3d official at the following website:
http://www.open3d.org/docs/release/tutorial/pipelines/global_registration.html

(6) local_icp_algorithm(source_down, target_down, trans_init, voxel_size): This function performs the local enhancement of the alignment by using the ICP(Iterative Closest Point) algorithm offered by the open3d official document at
http://www.open3d.org/docs/release/tutorial/pipelines/global_registration.html
The argument of this function is the down-sampled source point cloud and the target point cloud, an initial rough alignment transformation of the two point clouds and the parameter voxel size for the implementation. This function returns the result of the ICP process, the result would be used at the following reconstruction method.

(7) best_fit_transform(source_points, target_points): This function calculates the least-squares best-fit transform between corresponding 3D points source points and the target points for implementing the ICP algorithm. Configuring the dimension of the point clouds by knowing the shape by np.shape() numpy function.
This section of the code calculates the rotation matrix to align two sets of points. It follows these steps:
(a) Calculate the centroids of the source and target point clouds.
(b) Translate both point clouds so that their centroids coincide. This ensures that both point clouds are centered around the origin.
(c) Calculate the covariance matrix by computing the product of the centered coordinates of points in the source and target point clouds.
(d) Perform a Singular Value Decomposition (SVD) on the covariance matrix (W). SVD results in three matrices: the left singular vector matrix (U), the singular value vector (S), and the right singular vector matrix (VT).
(e) Calculate the rotation matrix (R) by multiplying the left singular vector matrix (U) and the right singular vector matrix (VT) obtained from the SVD. This rotation matrix represents the optimal rotation between the two sets of points to align them properly.
This method is commonly used in point cloud registration and is essential for aligning the source and target point clouds accurately.
Calculate the translation matrix by subtracting the "**dot of the rotation matrix and the transpose of the centroid of the source points matrix** " to

"**the transpose of the centroid of target points matrix**".

The transformation matrix could be defined by the rotation matrix and the translation matrix. Return the transformation matrix, and the rotation matrix, translation matrix for the following process.

(8) nearest_neighbor(source_points, target_points): This function find the nearest Euclidean neighbor in the target points for each point in the source points using the toolkit from sklearn. The function finds a single nearest point for each point, prepares the data structure that allows efficient nearest neighbor searches for the points in target_points.

By using the function keighbors(), save the distance and indices of the process: Distance, an array containing the Euclidean distances from each point in source_points to its nearest neighbor in target_points. Indices, an array containing the indices of the nearest neighbors in the target_points array for each point in source_points

Setting a booling variable to define whether a valid nearest neighbor is found base on the distance threshold. The distance threshold is set as 80% of the median distance between the source and target points. This threshold helps identify valid correspondences and ignore potential outliers.

(9) implemented_local_icp_algorithm(source_down, target_down, trans_init=None, max_iterations=100000 ,tolerance=0.000005): This function is the implemented local icp algorithm required by the specification, The arguments of the function is the down-sampled source point cloud and the target point cloud, the initial transformation between the two point clouds, the max iteration of the process, and the error tolerance for the operation. (The parameters could be tuned). Converting the points in the point clouds to numpy arrays for the calculation. Setting the dimension by using the numpy shape() function, in the task dimension m would be 3 in 3D space. Applying the initial pose estimation of two point.

Within a single loop (at most operation max_iterations times), the nearest neighbors between the current source point cloud (src) and the destination point cloud (dst) are first found. The nearest_neighbor function is used to perform this step, and it returns the distances for each source point, the indices of the nearest neighbors, and a flag indicating which points are valid.

Subsequently, the transformation matrix transformation that aligns the current source point cloud to its nearest neighbors in the destination point cloud is computed. This is achieved through the best_fit_transform function, which attempts to find the best rigid transformation (rotation and translation) to align the source and target points.

The current source point cloud src is transformed using this transformation matrix in order to prepare it for the next iteration. This is done by applying the transformation to all points in the current source point cloud.

After each iteration, an error check is performed. mean_error represents the average matching error (the mean of distances from points to their nearest neighbors) for the point clouds in the current iteration. If the change in error between two consecutive iterations is smaller than a specified threshold tolerance, the iterations stop.

Finally, upon exiting the loop, the final transformation matrix transformation is computed. It aligns the entire source point cloud (source_points) with the destination point cloud.

(10) read_pose(args): This function reads ground truth camera pose data from the dataset file "GT_pose.npy." It extracts this information to construct a point cloud and create a 3D line set representing the camera's actual path. The function allows you to specify a particular floor or elevation level, enabling you to isolate and align the camera data accurately. It first performs a rough alignment to establish a basic reference point between the camera's movements and the floor. Subsequently, a scale and coordinate transformation ensures that the camera data and point cloud match the scene's dimensions. The camera pose and collected points are converted into point cloud objects, facilitating the visualization of the camera's path and its relationship to the 3D scene. This process provides valuable insights into how the camera navigated the environment in the context of the scene's scale and structure.

(11) reconstruct(args): This function serves as the core of the entire reconstruction process, managing various key components. It primarily operates within the reference coordinate system established by the first image. The configuration of the process is as follows:

(a) The voxel size is set as 0.00225

(b) Two main point cloud objects are created - "point_cloud" to store the final scene point cloud and "estimate_camera_cloud" to hold points of the estimated camera positions.

(c) The function receives arguments, including the data path and file paths for RGB and depth images.

(d) The function starts by recording the time at the beginning of the reconstruction.

(e) Setting the starting point of the reconstruction, to record the whole running time of the code.

Create some templates of the point clouds and FPFH using list structure:

(a) pcd: Contains point clouds from each image in their respective camera coordinate systems.

(b) camera_pcd: contains the point clouds of each camera point in each coordinate relative to the main coordinate system (which is the first image's system).

(c) pcd_down: contains the down-sampled point clouds of each image in their corresponding camera coordinate system.

(d) pcd_transformed: contains the down-sampled pointclouds that after processing the global registeration and ICP registeration, all the down-sampled point clouds are transformed into the main coordinate system.

(e) fpfh: contains the FPFH of each down-sampled pointcloud.

The function then utilizes a "for" loop to process the data collection folder. During the first iteration:

(a) get the correct file path of the rgb and depth images.

(b) append the point cloud to pcd list.

(c) append the camera point point cloud to camera_pcd list.

(d) append the down-sampled point cloud to pcd_down list.

(e) append the FPFH of the down-sampled point cloud to the fpfh list.

(f) since the first point cloud is already in the main coordinate system, directly append the first point cloud to the list pcd_transformed

In subsequent iterations of the loop:

(a) Get the correct file path of the rgb and depth images.

(b) append the pointcloud to pcd list.

(c) append the camera point pointcloud to camera_pcd list.

(d) append the down-sampled pointcloud to pcd_down list.

(e) append the FPFH of the down-sampled point cloud to the fpfh list.

(f) Initiating the global registration process begins with marking the starting point to calculate the execution time. To obtain the result of fast global registration, the function "execute_fast_global_registration()" is employed. This process is executed iteratively, where for each operation, the source down-sampled point cloud is the current one. The target down-sampled point cloud is set to the one contained within the "pcd_transformed" list. Additionally, the Fast Point Feature Histograms (FPFH) used in the operation correspond to the current and previous iterations. The voxel size parameter is set according to the predefined configuration. The time taken for the fast global registration to execute is also printed for reference. This step ensures that the point clouds from different perspectives are cohesively aligned in the reconstruction process.

(g) Starting the ICP process, Setting the starting point of ICP registration to calculate the execution time. If the args.version is 'open3d' use the open3d's official method, otherwise use the implemented one. Similar with the global registeration process, save the result from the function whether by the local_icp_alogorithm() or the implementd_local_icp_algorithm, the source down-sampled point cloud is the current one, and the target down-sampled point cloud need to be set as the one that contains in pcd_transformed, and the initial alignment transformation is compute from the global registration function. Print the time that the ICP registration computes.

Upon completion of the loop, the down-sampled point cloud after alignment is appended to the "pcd_transformed" list, and the camera point cloud after alignment is added to the "camera_pcd" list. These lists now hold the transformed point clouds, and they are the key components in the subsequent reconstruction.

The final output is created by adding the point cloud from the "pcd_transformed" list to the overall scene point cloud. Additionally, the point cloud from the "camera_pcd" list is used to generate the final output for the estimated camera points. These final point clouds represent the reconstructed 3D scene and the estimated camera positions.

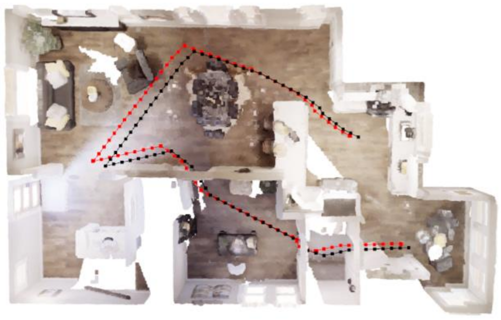
To ensure the accuracy of the reconstruction and to remove unwanted elements, a ceiling filtering step is performed. This step involves setting a threshold for the y-value of the points. The specific threshold used depends on the floor of the scene. If the scene is on the first floor, the threshold is set to 0.0135; for the second floor, the threshold is adjusted to 0.0115. Finally, the code prints the time it took for the entire reconstruction process to execute. This allows for monitoring and optimizing the runtime of the code, ensuring efficient reconstruction.

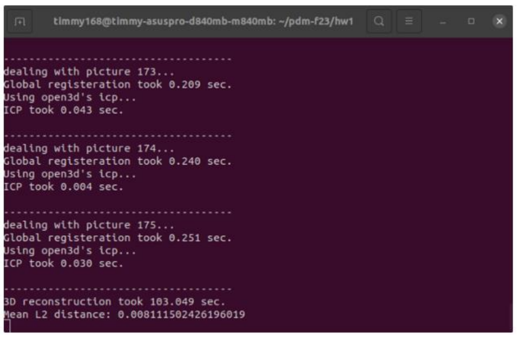
(12) calculate_mean_l2_distance(gt_pos_pcd, estimate_camera_cloud): This function calculates the Euclidean distance of each points in the ground truth

point cloud and the estimate point cloud and print the mean value of the error.

(13) main function: Setting the parsers as “f” for floor, “v” for the icp method version and data_root for the file path of the datacollection. Setting the data_root depends on which floor. Executing the reconstruction result, and output result point cloud and estimated camera pose. Calculate and print L2 distance with the function calculate_mean_l2_distance().Visualize the result with open3d function.

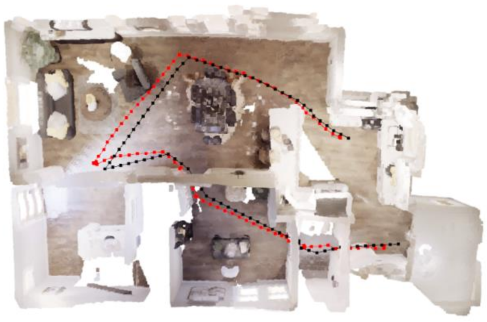
Results and Discussion:

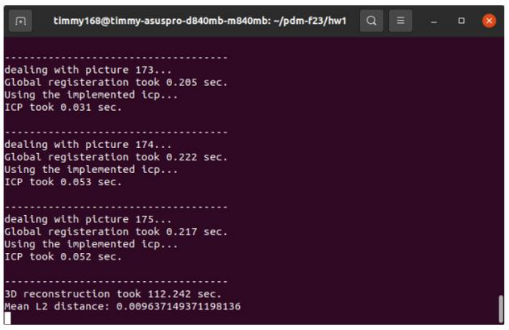




```
-----------------------------------
dealing with picture 173...
Global registeration took 0.209 sec.
Using open3d's icp...
ICP took 0.043 sec.

-----------------------------------
dealing with picture 174...
Global registeration took 0.240 sec.
Using open3d's icp...
ICP took 0.004 sec.

-----------------------------------
dealing with picture 175...
Global registeration took 0.251 sec.
Using open3d's icp...
ICP took 0.030 sec.

-----------------------------------
3D reconstruction took 103.049 sec.
Mean L2 distance: 0.008111502426196019
```

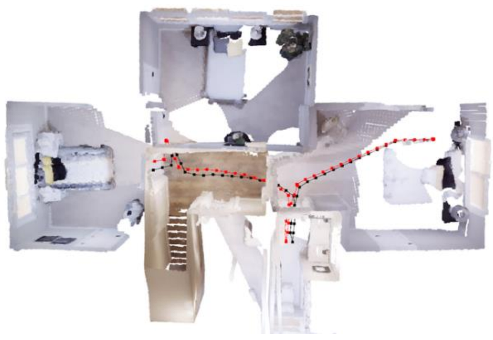
Fig(3). First Floor using Open3d’s ICP

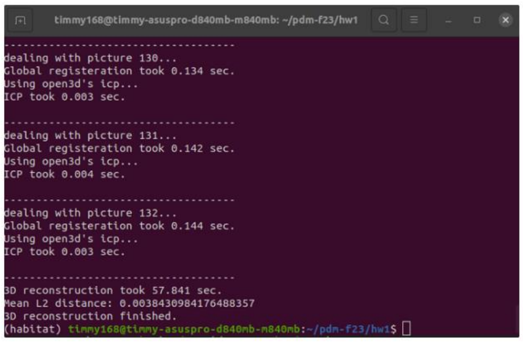




```
-----------------------------------
dealing with picture 173...
Global registeration took 0.205 sec.
Using the implemented icp...
ICP took 0.031 sec.

-----------------------------------
dealing with picture 174...
Global registeration took 0.222 sec.
Using the implemented icp...
ICP took 0.053 sec.

-----------------------------------
dealing with picture 175...
Global registeration took 0.217 sec.
Using the implemented icp...
ICP took 0.052 sec.

-----------------------------------
3D reconstruction took 112.242 sec.
Mean L2 distance: 0.009637149371198136
```

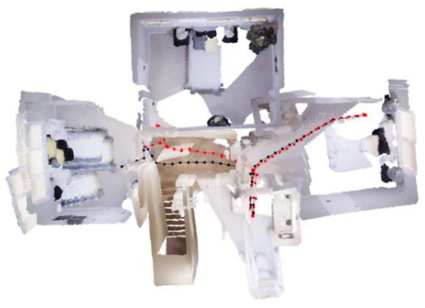
Fig(4). First Floor using implemented ICP

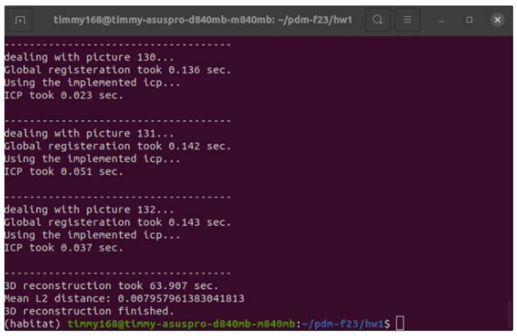




```
-----------------------------------
dealing with picture 130...
Global registeration took 0.134 sec.
Using open3d's icp...
ICP took 0.003 sec.

-----------------------------------
dealing with picture 131...
Global registeration took 0.142 sec.
Using open3d's icp...
ICP took 0.004 sec.

-----------------------------------
dealing with picture 132...
Global registeration took 0.144 sec.
Using open3d's icp...
ICP took 0.003 sec.

-----------------------------------
3D reconstruction took 57.841 sec.
Mean L2 distance: 0.0038430984176488357
3D reconstruction finished.
(habitat) timmy168@timmy-asuspro-d840mb-m840mb:~/pdm-f23/hw1$
```

Fig(5). Second Floor using Open3d’s ICP





```
-----------------------------------
dealing with picture 130...
Global registeration took 0.136 sec.
Using the implemented icp...
ICP took 0.023 sec.

-----------------------------------
dealing with picture 131...
Global registeration took 0.142 sec.
Using the implemented icp...
ICP took 0.051 sec.

-----------------------------------
dealing with picture 132...
Global registeration took 0.143 sec.
Using the implemented icp...
ICP took 0.037 sec.

-----------------------------------
3D reconstruction took 63.907 sec.
Mean L2 distance: 0.007957961383041813
3D reconstruction finished.
(habitat) timmy168@timmy-asuspro-d840mb-m840mb:~/pdm-f23/hw1$
```

Fig(6). Second Floor using Implemented ICP

The results are displayed in the figures above. For the first floor, when using Open3D's ICP, the process took 103 seconds, and the Mean L2 distance is 0.008. When using the implemented ICP for the first floor, the process took 112 seconds, and the Mean L2 distance is 0.0096.

In the case of the second floor, using Open3D's ICP took 57 seconds, and the Mean L2 distance is 0.0038. For the first floor with Open3D's ICP, the process took 63 seconds, and the Mean L2 distance is 0.0079.

The results indicate that the performance of Open3D's ICP and the implemented ICP is comparable. Both methods provide similar accuracy and efficiency in the reconstruction process.

In Task 2, implement a 3D reconstruction code that consists of the following components:

(a) Converting RGBD images into point clouds

(b) Preprocessing of point clouds

(c) Global registration

(d) Local Enhancement with ICP(Iterative Closest Point) algorithm

(e) Reconstruction of the 3D scene

(f) Estimation of camera poses

(g) Calculate the L2 distance

As part of the data collection process, convert the collected RGB images and depth images into point clouds for use in the reconstruction process. Voxelize and down-sample the point clouds to reduce the number of points, resulting in lower memory usage and faster execution times.

To reconstruct the 3D scene, it is crucial to perform a rough alignment of the point cloud using global registration. Without this initial alignment, ICP registration can often fail. Global registration provides an essential initialization step. Open3D offers two global registration functions, and for efficiency, I opt for the faster option - the global fast registration. Surprisingly, this method delivers accuracy comparable to that of ICP.

After initializing the alignment using the global registration method, I further refine the alignment with the ICP (Iterative Closest Point) algorithm. Open3D offers a user-friendly ICP function that runs at a remarkable speed. In our implementation, both methods performed admirably, with each execution times consistently falling within the range of 0.002 to 0.03 seconds. The execution time of the ICP method is significantly influenced by the preceding global registration step, whereas the number of iterations and the tolerance parameters do not have as significant an impact as the previously mentioned alignment process.

While implementing the ground truth pose and camera pose functions, it's crucial to establish the correct coordinate systems and scale factors that relate the data to the actual coordinate system. Based on the results, the average L2 distance is notably small, indicating the effective performance of both global registration and the ICP implementation.

IV. Reference resources

(1) Bird eye view transformation - https://nikolasent.github.io/opencv/2017/05/07/Bird's-Eye-View-Transformation.html

(2) Global registration - http://www.open3d.org/docs/release/tutorial/pipelines/global_registration.html

(3) Point Cloud Alignment - https://github.com/rpadmanabhan/point-cloud-alignment

(4) Iterative closest points algorithm - https://github.com/ClayFlannigan/icp