Perception and Decision Making in Intelligent Systems
Homework 1

310551003 陳嘉慶

This homework requires to reconstruct an environment through Habitat and Replica simulator captured images from apartment_0.

Task 1: BEV projection

The first task is BEV projection, utilize the simulator and add new code to load.py as below.

Part1: Data Collection (Save image from simulator)

1. We need to add new sensor for capture BEV view, below RGB sensor is necessary to add in the simulator and we need to adjust orientation and position configure parameter as we need (at least both BEV and front view have common part), you can refer to below code remark for explanation.

Below code was added under `def make_simple_cfg(settings):`

```
# RGB sensor BEV view
rgb_bev_sensor_spec = habitat_sim.CameraSensorSpec()  # initial a new
sensor
rgb_bev_sensor_spec.uuid = "color_bev_sensor"  # new sensor ID
rgb_bev_sensor_spec.sensor_type = habitat_sim.SensorType.COLOR
# new sensor type (COLOR=RGB)
rgb_bev_sensor_spec.resolution = [settings["height"],
settings["width"]]
# resolution of sensor view
rgb_bev_sensor_spec.position = [0.0, 2.0, 0.0]
# sensor coordinate, height=2.0 for BEV
rgb_bev_sensor_spec.orientation = [settings["sensor_pitch"], 0.0,
0.0,]
# orientation, x=-90deg for BEV
rgb_bev_sensor_spec.sensor_subtype = habitat_sim.SensorSubType.PINHOLE
# sensor subtype
```

2. The picture can be captured by edited navigateAndSee function as below, capture image from sensor observations and convert data to RGB image, finally save it as png.

Below code was added under `def navigateAndSee(action="", num=0):`

```
img_rgb_bev = transform_rgb_bgr(observations["color_bev_sensor"])  #
get observation from sensor and transform to image

cv2.imwrite("./save/RGB/{}.png".format(num), img_rgb)  # save image
```

Part 2: BEV Projection

Mark click point on BEV view and project the mark area on front view.

310551003 陳嘉慶

1. From bev.py we have click point function, modify bev.py as below to read image and send to click_event for recording the click point.
2. Input the recorded points and build camera projection matrix (intrinsic and extrinsic matrix) under projection.top_to_front function, transform input point to front view.
3. Finally, show new points on front image.

Below code is the main code, the rest code will be shown later:

```
if __name__ == "__main__":
    pitch_ang = -np.pi / 2  # BEV rotation angle
    image_name = 69  # image to be convert marked point
    front_rgb = "./save/RGB/{}.png".format(image_name)  # path for read front view image
    top_rgb = "./save/RGB_bev/{}.png".format(image_name)  # path for read BEV image

    # click the pixels on window
    img = cv2.imread(front_rgb, 1)  # read BEV RGB
    cv2.imshow('image', img)  # show image
    cv2.setMouseCallback('image', click_event)  # show click point on image
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    print("out: ", points)
    projection = Projection(front_rgb, points)  # initial projection
    new_pixels = projection.top_to_front(gamma=pitch_ang, dy=sensor_height - sensor_height_bev)
    # input point from BEF and get the new points (at front view) from intrinsics and extrinsic matrix
    projection.show_image(new_pixels)  # project BEV point to front RGB image
```

Initial projection function, initial all points, height, width information and calculate the focal for intrinsic matrix.

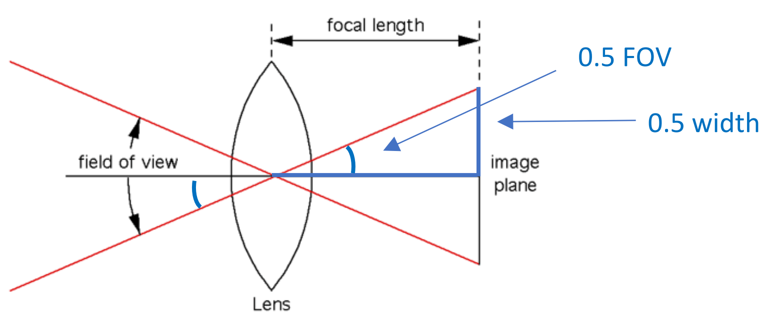

Figure 1. Focal calculation

Perception and Decision Making in Intelligent Systems
Homework 1

310551003 陳嘉慶

We can see Figure 1 explain the focal length calculation (geometry calculation):

horizontal field of view = 2 atan(0.5 width / focal length)

According to this equation, we can have focal length by focal = width / (tan (pi / (2x2)) x 2)

Utilize focal to build intrinsic matrix as below:

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & u_0 \\ 0 & f & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Figure 2. Intrinsic matrix

Where u_0, v_0 is 256, 256 (the center of image counting from top left), f is focal length, u, v is the click point to be converted, w=Z is homogenous element, XYZ is corresponding camera coordinate, and the axis is in the same direction of original one.

Below code is the implementation from python with explanation, we also calculate inverse intrinsic matrix for later use.

```
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.points = points
        self.height, self.width, self.channels = self.image.shape
        self.focal = self.width / (np.tan(np.pi / 4) * 2)
        # calculate focal from dimension of image and FOV(90deg):
horizontal field of view = 2 atan(0.5 width / focallength)
        self.in_matrix = np.array([[self.focal, 0, 256], [0,
self.focal, 256], [0, 0, 1]])
        # intrinsic matrix, center is offset to (256, 256) from top
left corner
        self.in_matrix_inv = np.linalg.inv(self.in_matrix) # calculate
inverse matrix for later use
```

Perception and Decision Making in Intelligent Systems
Homework 1

310551003 陳嘉慶

We first load a pixel from image and convert it to homogeneous coordinate, utilize the known height which we set on the simulate to convert the pixel as [u, v, w] matrix and multiple the inverse intrinsic matrix for converting pixel/image coordinate to camera coordinate.

Second, we build extrinsic matrix for convert image coordinate to camera coordinate, input all the parameter we set on simulator to build the matrix as shown on Figure 3.

Where Gamma = -90 degree, ty = 1500-2000 = -500mm (1500mm is height of front view camera, 2000mm is height of BEV camera, we convert BEV to front view, so that front view minus BEV, the rest of parameter is 0.

$$\alpha = 0, \beta = 0, \gamma = -90, t_x = 0, t_y = -500, t_z = 0,\ {}^1x,\ {}^1y,\ {}^1z\ are\ the\ point\ from\ BEV$$

$$R = \begin{bmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma \end{bmatrix}$$

$$\begin{bmatrix} {}^0x \\ {}^0y \\ {}^0z \\ 1 \end{bmatrix} = \begin{bmatrix} & {}^0R_1 & & \begin{matrix} t_x \\ t_y \\ t_z \end{matrix} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^1x \\ {}^1y \\ {}^1z \\ 1 \end{bmatrix}$$

Figure 3 Extrinsic matrix transformation

Thus, we can have front camera coordinate of BEV pixels by multiple extrinsic matrix and BEV points on BEV camera coordinate. Finally, we convert front camera coordinate back to pixel/image coordinate by using the same intrinsic matrix.

Below code was added under `class Projection(object):`

```
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
    """
    new_pixels = []  # create new pixels
    for i in self.points:
        i.append(1)  # make uv to homogeneous coordinate
        uv = np.array(i) * sensor_height_bev  # convert uv to matrix
        XY = np.dot(self.in_matrix_inv, np.reshape(uv, (3, 1)))
```

```
        # using inverse intrinsic matrix to find out the camera coordinate from pixel/image coordinate
        ex_matrix = T3_matrix(theta, phi, gamma, dx, sensor_height - sensor_height_bev, dz)
        # create extrinsic matrix for convert BEV camera coordinate to front camera coordinate
        # bev[x,y,z] = bev[z,y,x, tx,ty,tz]: rotate x axis -90deg, ty=-500
        new_point = np.dot(ex_matrix, np.reshape([[XY[0][0], XY[1][0], sensor_height_bev, 1]], (4, 1)))
        # transform points from BEV camera coordinate to front camera coordinate
        point = new_point[:3]
        uv = np.dot(self.in_matrix, point)  # convert front camera coordinate to pixel/image coordinate
        uv = np.around(uv / uv[2][0], decimals=0).astype(int)  # round the num, as pixel can only be int
        new_pixels.append([uv[0][0], uv[1][0]])  # save every point which already converted
    print(new_pixels)
    return new_pixels
```

Result:

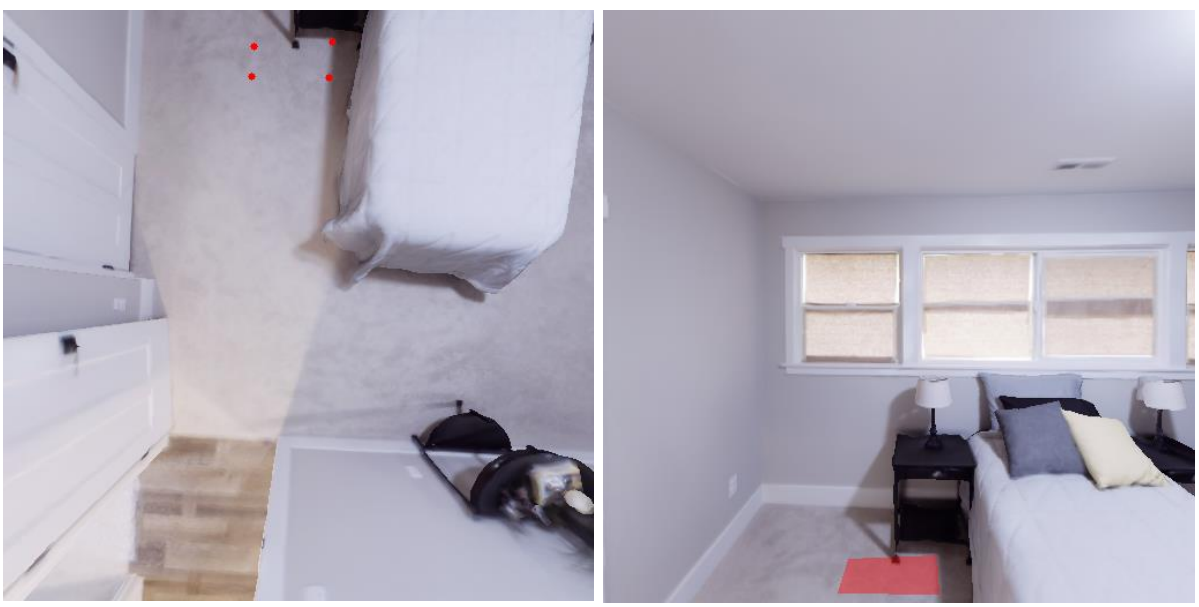

Figure 4 Result of BEV Projection

Figure 5 Result of BEV Projection

I selected two figures from 1st floor and 2nd floor to convert the points from BEV to front view and mark the points at front view, finally draw the fill the polygon in sequence.

This task allows us to understand the intrinsic and extrinsic matrix from a simple example. However, I work on this task before lecture and spend a lot of time by studying both matrix from opensource reference and course assistance. Finally, I figure it out and confirm what I learn from the class. The transformation is not difficult, but it takes time to understand coordinate relationship on it and it need to practice more than once for understanding.

Task 2: ICP alignment

Part 1: Data Collection

Utilize load.py to go through simulator and save all RGB and depth image for later use.

Part2: Point Cloud Alignment and Reconstruction

1. Initial the intrinsic matrix as mentioned on Part 1, read RGB and depth images for coordinate transformation.
2. For an image, we repeat the process we mentioned on Part 2 of Task 1, but using depth instead of known Z (2000mm) from BEV. Convert pixels to camera coordinate by dot produce of inverse intrinsic matrix and homogeneous pixel array.
3. Save every pixel as X Y Z R G B format where R, G, B need to be normalized.

```
def img_to_3d(start=0, total_img=11, width=512):
    focal = float(width / (np.tan(np.pi / 4) * 2))
# for virtual camera, 1 pixel = 1 mm. width=512mm, focal = 256
    in_matrix = np.array([[focal, 0, 256], [0, focal, 256], [0, 0,
1]])  # intrinsic matrix as explain
    in_matrix_inv = np.linalg.inv(in_matrix)  # inverse intrinsic
matrix
    progress = tqdm(total=total_img-start, desc="img_to_3d_progress")
# progress bar
    for image_name in range(start, total_img):
        front_rgb = "./save/RGB/{}.png".format(image_name)  # image
name format for load image
        front_depth = "./save/depth/{}.png".format(image_name)
        output_name="./pcd/result_output_{}.xyzrgb".format(image_name)
# save image to pointcloud as xyzrgb format
        outputFile = open(output_name, "w")  # create point cloud name
        img = cv2.imread(front_rgb, 1)  # read image as BGR sequence
        norm_img = cv2.normalize(img, None, alpha=0, beta=1,
norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F).astype(str)
        # normalize image color as xyzrgb require
        img_depth = cv2.imread(front_depth, cv2.IMREAD_GRAYSCALE)  #
read depth image
        for v, u in itertools.product(range(img.shape[0]),
range(img.shape[1])):  # v and u inverse
            uv = np.array([u,v,1]) * img_depth[u][v]  # homogeneous
array of uv using depth of image
            xyz = np.around(np.dot(in_matrix_inv, uv),
decimals=5).astype(str)  # around coordinate before save file
            outputFile.write("{} {} {} {} {} {}".format(  # save point
cloud data as xyzrgb format
                xyz[0],  # x-value
                xyz[1],  # y-value
                xyz[2],  # z-value
                norm_img[u][v][2],  # r = cv2[2]
                norm_img[u][v][1],  # g = cv2[1]
                norm_img[u][v][0]  # b  = cv2[0]
                # normalize rgb value
            ) + "\n")
        progress.update(1)  # update progress bar
```

As we have point cloud of image, we can now do reconstruction. We first do the sampling on point cloud to reduce the calculation of iterative closest point (ICP), and we can convert source camera coordinate to target camera coordinate utilized the transformation matrix from ICP. The detail will be discussing in the following.

310551003 陳嘉慶

```
def reconstruction(voxel_size=1.0, total_img=10):
    track_arr = np.zeros((1, 3))  # initial track arr for record the
camera movement
    lines = []  # connect track arr points by lines, record index
later on
    # trans_total = []
    trans_init = np.identity(4)  # initial transformation matrix
    # trans_mtx_init = np.identity(4)
    # trans_init[3,:3] = np.array([[0],[0.12523484],[-0.25]])
    progress = tqdm(total=total_img, desc="reconstruction")  #
progress bar
    target_name = "./pcd/result_output_{}.xyzrgb".format(0)  # target
camera point cloud name
    target, target_down, target_fpfh = prepare_dataset(voxel_size,
target_name)
    # sample point cloud and find relationship between them
    vis = o3d.visualization.Visualizer()  # initial o3d visualizer
    target_temp = copy.deepcopy(target_down)  # copy target point
cloud for later use

    track_gt_name = "./save/record.txt"  # name of track ground truth
from record
    track_gt = pd.read_csv(track_gt_name, header=None, sep=' ')  # get
the track ground truth from record
    track_gt_arr = np.array(track_gt.drop(track_gt.columns[3:],
axis=1))  # take xyz only
    track_gt_arr = np.subtract(track_gt_arr, track_gt_arr[0]) * 20  #
translate gt to global coordinate
    ex_matrix = T3_matrix(np.pi / 2, -np.pi / 3, 0, 0, 0, 0)[:3, :3]
# rotate z 90deg, y 60 deg to global coordinate
    # vis.run()

    for image_name in range(1, total_img):
        source_name =
"./pcd/result_output_{}.xyzrgb".format(image_name)  # name of image to
be project
        source, source_down, source_fpfh = prepare_dataset(voxel_size,
source_name)
        # sample point cloud data and get their relationship
        result_ransac = execute_global_registration(source_down,
target_down, source_fpfh, target_fpfh, voxel_size)
        # execute global icp and get initial matrix
        result_icp = refine_registration(source, target, voxel_size,
result_ransac.transformation)
        # execute local icp
```

```
        trans_init = np.dot(trans_init, result_icp.transformation)

        # add camera point at global coordinate
        track_point = np.zeros((1, 4))  # initial xyz track point
        track_point[0][3] = 1  # homogeneous track point
        track_point = np.dot(trans_init, track_point.T).T  # Transform
current camera (0,0,0) to global camera
        track_arr = np.concatenate((track_arr, track_point[:, :3]))  #
save converted track point

        track_gt_arr[image_name] = np.dot(ex_matrix,
track_gt_arr[image_name].T).T  # rotate ground truth track point
        source_temp = copy.deepcopy(source)  #
        source_temp.transform(trans_init)  # transform point cloud
from current camera (0,0,0) to global camera
        rmv_ceiling(source_temp, vis)  # remove points at ceiling and
add point cloud to visualizer
        target, target_down, target_fpfh = source, source_down,
source_fpfh
        # save source infomation to target for next loop
        progress.set_description("%s" % result_ransac)
        progress.update(1)
        lines.append([image_name - 1, image_name])
    track_arr = track_arr.reshape(-1, 3)

    track = o3d.PointCloud()
    track.points = o3d.Vector3dVector(track_arr)
    track.paint_uniform_color([1, 0.706, 0])  # yellow
    track_line = o3d.geometry.LineSet()
    track_line.points = o3d.utility.Vector3dVector(track.points)
    track_line.lines = o3d.utility.Vector2iVector(lines)
    vis.add_geometry(track_line)
    rmv_ceiling(track, vis)
    rmv_ceiling(target_temp, vis)

    track_gt = o3d.PointCloud()
    track_gt.points = o3d.Vector3dVector(track_gt_arr)
    track_gt.paint_uniform_color([0, 0.651, 0.929])  # blue
    track_gt_line = o3d.geometry.LineSet()
    track_gt_line.points = o3d.utility.Vector3dVector(track_gt.points)
    track_gt_line.lines = o3d.utility.Vector2iVector(lines)
    vis.add_geometry(track_gt_line)
    rmv_ceiling(track_gt, vis)
    opt = vis.get_render_option()
    opt.show_coordinate_frame = True
```

```
    opt.background_color = np.asarray([0.5, 0.5, 0.5])
    vis.run()
```

4. Prepare data from save point cloud data, we first sample the point cloud for reducing calculation, using voxel_down_sample with voxel size and filter out redundancy point according voxel size. Then estimate normal and FPFH feature by KDTreeSearch, return point cloud and it's FPFH feature.

```
def prepare_dataset(voxel_size, source_name):
    source = o3d.read_point_cloud(source_name)  # source
    source_down, source_fpfh = preprocess_point_cloud(source,
voxel_size)  # sample points and get points relationship
    return source, source_down, source_fpfh

def preprocess_point_cloud(pcd, voxel_size):
    pcd_down = pcd.voxel_down_sample(voxel_size)  # Downsample with a
voxel size
    radius_normal = voxel_size * 2  # Estimate normal with search
radius
pcd_down.estimate_normals(o3d.geometry.KDTreeSearchParamHybrid(radius=
radius_normal, max_nn=30))
    radius_feature = voxel_size * 5  # Compute FPFH feature with
search radius
    pcd_fpfh = o3d.registration.compute_fpfh_feature(
        pcd_down,
o3d.geometry.KDTreeSearchParamHybrid(radius=radius_feature,
max_nn=100))
    return pcd_down, pcd_fpfh
```

5. Execute global ICP for getting initial matrix for local ICP use, the main idea of ICP is to calculate the distance between a pair of point clouds and using SVD to decomposition of a matrix H which form as below formula, where centroid is mean of each point cloud.

$$H = \sum_{i=1}^{N}(P_A^i - centroid_A)(P_B^i - centroid_B)^T$$

As R = V * U.T and t = $centroid_B$ – R x $centroid_A$, where U, V we can get from SVD as below, and finally we can combine extrinsic matrix as Figure 3 shown.

$$[U, S, V] = SVD(H)$$

310551003 陳嘉慶

```
# RANSAC global_reg
def execute_global_registration(source_down, target_down, source_fpfh,
                                target_fpfh, voxel_size):
    distance_threshold = voxel_size * 1.5  # distance threshold of
corresponding points
    result =
o3d.registration.registration_ransac_based_on_feature_matching(
        source_down, target_down, source_fpfh, target_fpfh,
distance_threshold,
        o3d.registration.TransformationEstimationPointToPoint(False),
        4, [
            o3d.registration.CorrespondenceCheckerBasedOnEdgeLength(
                0.9),
            o3d.registration.CorrespondenceCheckerBasedOnDistance(
                distance_threshold)
        ], o3d.registration.RANSACConvergenceCriteria(4000000, 500))
    # get the Iterative Closest Point from open3d
    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_fast_based_on_feature_matching
(         source_down, target_down, source_fpfh, target_fpfh,
        o3d.pipelines.registration.FastGlobalRegistrationOption(
            maximum_correspondence_distance=distance_threshold))
    return result
```

<u>Self ICP implementation:</u>

Point base matching for ICP:

Given source points and target points, the H matrix can be calculated as above-mentioned formula, where we calculate the mean of both point clouds and subtract it one by one, finally do the dot produce of both point clouds build the H matrix, where H matrix is familiar covariance matrix.

After we have H matrix we can decomposition this matrix by SVD to U∑V and U, V is rotation and reflection matrix, it can be used to calculate rotation and translation matrix as below:

$$R = UV^T$$
$$t = \mu_x - R\mu_p$$

310551003 陳嘉慶

```
def point_based_matching(points_trans, points_ref):
    assert points_trans.shape == points_ref.shape  # check the pairs
of points num equal or not
    n = points_trans.shape[0]  # n= num of pair points
    m = points_trans.shape[1]  # m= dimension of matrix
    if n == 0:  # if no pair point return
        return None, None
    points_trans_mean = np.mean(points_trans, axis=0)  # calculate the
center of source point cloud
    points_ref_mean = np.mean(points_ref, axis=0)  # calculate the
center of target point cloud
    points_trans = np.subtract(points_trans, points_trans_mean)  # pre
process for build H matrix
    points_ref = np.subtract(points_ref, points_ref_mean)
    points_mtx = np.dot(points_trans.T, points_ref)  # build H matrix
    # x=u*s*v;  u: rotation , s: scaler, vT: rotation
    u, s, vT = np.linalg.svd(points_mtx)  # decomposition SVD matrix
to U, S and VT
    rot_mtx = np.dot(vT.T, u.T)  # calculate the rotation matrix
    if np.linalg.det(rot_mtx) < 0:  # check if rot_mtx is reflection
matrix or not
        print ("sign")
        Vt[m-1, :] *= -1
        rot_mtx = Vt.T * U.T
        print('new R', rot_mtx)
    t_mtx = points_ref_mean.T - np.dot(rot_mtx, points_trans_mean.T)
# calculate the translation matrix
    T_mtx = np.identity(m + 1)  # initial transformation matrix
    T_mtx[:m, :m] = rot_mtx  # corresponding transformation matrix
element replaced by rotation matrix element
    T_mtx[:m, m] = t_mtx  # corresponding transformation matrix
element replaced by translation matrix element
    return rot_mtx, t_mtx, T_mtx
```

ICP:

Fit the nearest neighbor from target point cloud, find the nearest neighbor of source point cloud which matching to the target point cloud, compute the transformation matrix from point_base_matching function mentioned above. Then, check the distance between target and source after transform source point, break the loop if error convergence or error under limit.

```
def icp(points, reference_points, voxel_size, mtx_init=np.identity(4),
max_iterations=50, tolerance=1e-3, point_pairs_threshold=10,
verbose=False):
    points = np.array(points.points)  # get points from point cloud
```

310551003 陳嘉慶

```
    reference_points = np.array(reference_points.points)
    n = points.shape[0]  # total number of points
    m = points.shape[1]  # matrix size
    src = np.ones((m + 1, n))  #
    trg = np.ones((m + 1, reference_points.shape[0]))
    src[:m,:] = copy.deepcopy(points.T)  # build source matrix
    trg[:m,:] = copy.deepcopy(reference_points.T)  # build target
matrix
    nbrs = NearestNeighbors(n_neighbors=1).fit(trg[:m,:].T)  # initial
target in Nearest Neighobrs
    prev_error = 0
    min_error = 1
    distance_threshold = voxel_size * 0.4  # set distance threshold
    T_mtx_final = mtx_init  # initialize Transformation matrix
    progress = tqdm(total=max_iterations, desc="icp_implement",
position=0, leave=True)
    for iter_num in range(max_iterations):
        found = 1
        points_trans = np.array([])
        points_ref = np.array([])
        distances, indices = nbrs.kneighbors(src[:m,:].T)
        for nn_index in range(len(distances)):
            if distances[nn_index][0] < distance_threshold:  # filter
points if pair points < threshold
                points_trans = np.concatenate((points_trans, src[:,
nn_index]))
                points_ref = np.concatenate((points_ref, trg[:,
indices[nn_index][0]]))
        points_trans = points_trans.reshape(m+1, -1)  # reshape new
pair points
        points_ref = points_ref.reshape(m+1, -1)
        if points_trans.shape[1] < point_pairs_threshold:    # if too
few points break
            break

        # compute translation and rotation using point correspondences
        r_mtx, t_mtx, T_mtx =
point_based_matching(points_trans[:m, :].T, points_ref[:m, :].T)  #
compute T_matrix
        if r_mtx is None or t_mtx is None:
            break
        src = np.dot(T_mtx, src)  # transform source matrix for next
loop
        T_mtx_final = np.dot(T_mtx_final, T_mtx)  # accumulate
transformation matrix
```

```
        # mean_error = np.mean(np.square(distances))  # compute the
error using two point distance
        error = np.mean(np.abs(np.subtract(points_ref, points_trans)))
        # error = np.abs(prev_error - mean_error)
        if error < tolerance:  # check convergence
            break
        # prev_error = mean_error  # save error for next loop
        if error < min_error:  # save minimum error transform matrix
            T_mtx_min = copy.deepcopy(T_mtx_final)
            min_error = error
        if iter_num == max_iterations-1:  # mark not found if reach
max iteration
            found = 0
        progress.set_description("icp_implement: error: %s, pairs: %s"
% (error, points_trans.shape[1]))
        progress.update(1)
    if found == 0:  # take minimum error transform matrix
        T_mtx_final = T_mtx_min
        print("not found tolerance matrix", end=' ')
    # print("T_mtx_final: ", T_mtx_final)
    return T_mtx_final, src
```

6. L2 distance:
   As we saved the global coordinate (which also noted as ground truth trajectory) when we observe environment from simulator, we only need to read the record and transform it back to global camera coordinate. For estimate trajectory, we will get it from every camera coordinate (0,0,0) and dot product with extrinsic matrix from ICP so that every other camera coordinate can transform to global camera coordinate. Finally, calculate every point L2 distance by subtract it one by one.

```
def reconstruction(voxel_size=1.0, total_img=10):
    track_arr = np.zeros((1, 3))  # initial track arr for record the
camera movement
    lines = []  # connect track arr points by lines, record index
    track_gt_name = "./save/record.txt"  # name of track ground truth
from record
    track_gt = pd.read_csv(track_gt_name, header=None, sep=' ')  # get
the track ground truth from record
    track_gt_arr = np.array(track_gt.drop(track_gt.columns[3:],
axis=1))  # take xyz only
    track_gt_arr = np.subtract(track_gt_arr, track_gt_arr[0]) * 20  #
translate gt to global coordinate
    ex_matrix = T3_matrix(np.pi/2, -np.pi/3, 0, 0, 0, 0)[:3, :3]  #
rotate z 90deg, y 60 deg to global coordinate
```

```
    for image_name in range(1, total_img):
        source_name =
"./pcd/result_output_{}.xyzrgb".format(image_name)  # name of image to
be project
        source, source_down, source_fpfh = prepare_dataset(voxel_size,
source_name)
        # sample point cloud data and get their relationship
        result_global = execute_global_registration(source_down,
target_down, source_fpfh, target_fpfh, voxel_size)
        # execute global icp and get initial matrix
        # trans_init = np.dot(trans_init,
result_ransac.transformation)
        result_icp = refine_registration(source, target, voxel_size,
result_global.transformation)
        # execute local icp
        trans_init = np.dot(trans_init, result_icp.transformation)

        # add camera point at global coordinate
        track_point = np.zeros((1, 4))  # initial xyz track point
        track_point[0][3] = 1  # homogeneous track point
        track_point = np.dot(trans_init, track_point.T).T  # Transform
current camera (0,0,0) to global camera
        track_arr = np.concatenate((track_arr, track_point[:, :3]))  #
save converted track point

        track_gt_arr[image_name] = np.dot(trans_init[:3,:3],
track_gt_arr[image_name].T).T  # rotate ground truth track point
        source_temp = copy.deepcopy(source)  #
        source_temp.transform(trans_init)  # transform point cloud
from current camera (0,0,0) to global camera
        rmv_ceiling(source_temp, vis)  # remove points at ceiling and
add point cloud to visualizer
        target, target_down, target_fpfh = source, source_down,
source_fpfh
        # save source infomation to target for next loop
        progress.set_description("result_global: %s" % result_global)
        progress.update(1)
        lines.append([image_name-1, image_name])
    track_arr = track_arr.reshape(-1, 3)
    # print(track_arr)
    track = o3d.PointCloud()
    track.points = o3d.Vector3dVector(track_arr)
    track.paint_uniform_color([1, 0.706, 0])  # yellow
    track_line = o3d.geometry.LineSet()
    track_line.points = o3d.utility.Vector3dVector(track.points)
```

```
    track_line.lines = o3d.utility.Vector2iVector(lines)
    vis.add_geometry(track_line)
    rmv_ceiling(track, vis)
    rmv_ceiling(target_temp, vis)

    track_gt = o3d.PointCloud()
    track_gt.points = o3d.Vector3dVector(track_gt_arr)
    track_gt.paint_uniform_color([0, 0.651, 0.929])  # blue
    track_gt_line = o3d.geometry.LineSet()
    track_gt_line.points = o3d.utility.Vector3dVector(track_gt.points)
    track_gt_line.lines = o3d.utility.Vector2iVector(lines)
    vis.add_geometry(track_gt_line)
    rmv_ceiling(track_gt, vis)
    opt = vis.get_render_option()
    opt.show_coordinate_frame = True
    opt.background_color = np.asarray([0.5, 0.5, 0.5])
    distance = np.subtract(track_gt_arr, track_arr)
    for i in range(track_gt_arr.shape[0]):  # L2 distance
        print(track_gt_arr[i], track_arr[i], distance[i])
    vis.run()
```

Result:

The result was shown as Figure 6, we use these parameters for reconstruction:

Voxel_size=1e-5,

Global ICP: TransformationEstimation: RANSAC, PointToPoint, distance_threshold = 2e-5

RANSACConvergenceCriteria: iteration=4e6, max_validation=500

Local ICP: PointToPlane, distance_threshold=4e-6

radius_outlier_removal: nb_points=30, radius=5e-5

KDTreeSearchParamHybrid: radius=2e-5, max_nn=30

KDTreeSearchParamHybrid: radius=5e-5, max_nn=100

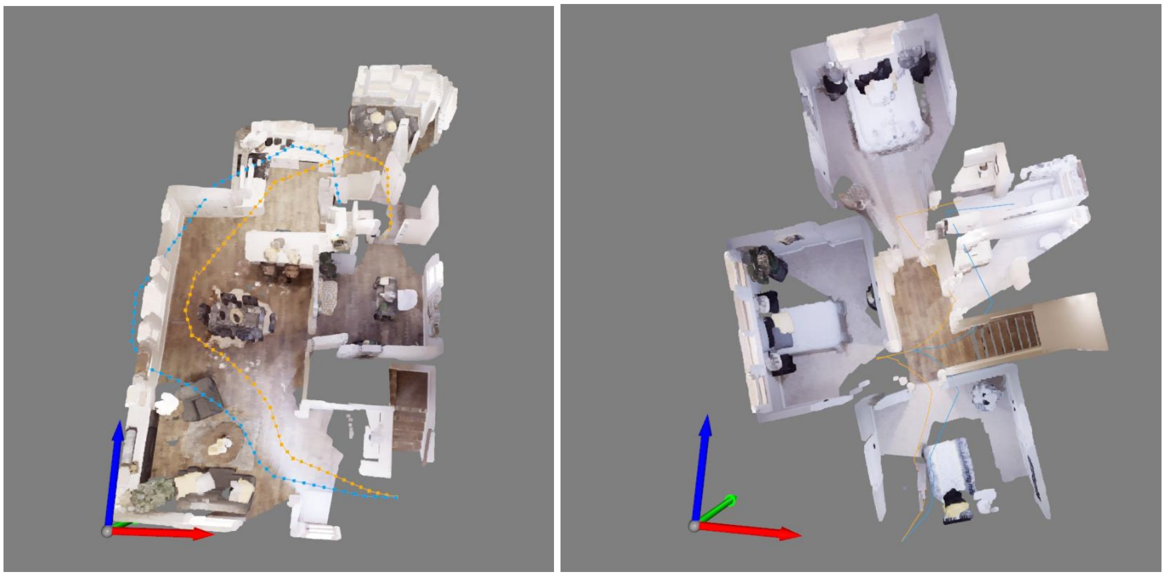

Figure 6 Result of Open3d reconstruction

The result shows the reconstruct overall is acceptable, but some of location reconstruct not correctly and it is most likely related to the input image not good enough or the trajectory path no enough to provide good quality of reconstruction. However, I believe if I can filter out less related corresponding point pairs, the result should become better. And I found out that there is a method to make a combine point cloud, in this case I believe it will improve because we can combine it plane corresponding to plane or surface to surface rather than point to point.

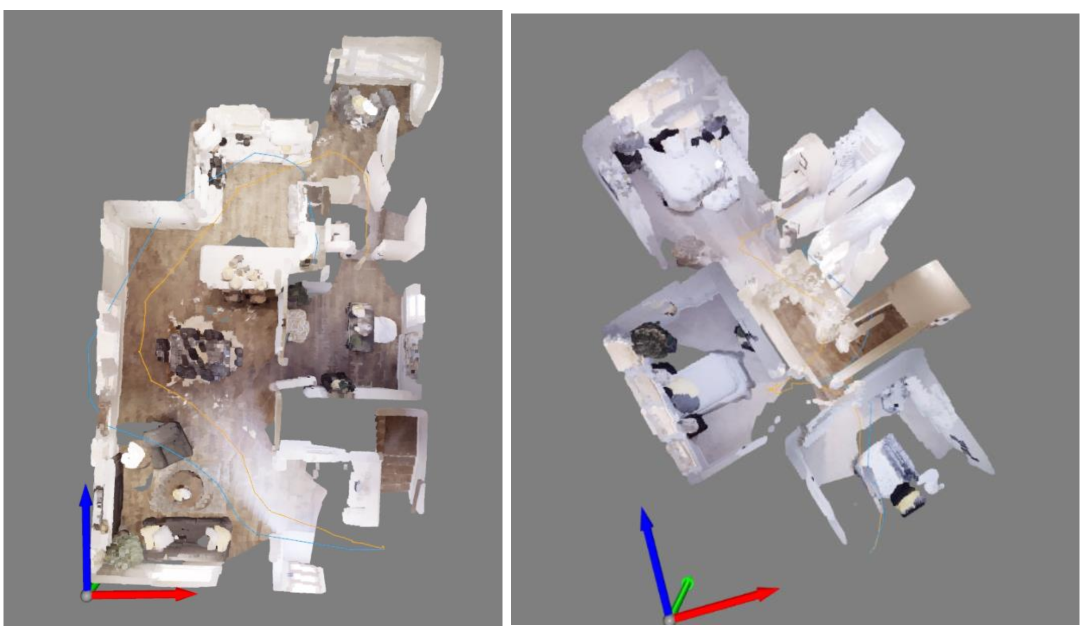

Figure 7 Result of implement reconstruction

310551003 陳嘉慶

Figure 7 is my own implement ICP, it seems not reconstruct as well as open3d library, however, it should be adjusted if I can have time to modify my loss function and adjust parameters to tune it right. There are many reasons that the result become worst:

1. The transformation process may cause error estimation problem (it called gradient vanishing/explosion in deep learning), it means too many cumulative products of estimate transformation will cause overestimate or under-estimate when reconstruction the scene, that is, the reconstruction become worst when images become more as shown on Figure 8.
2. As the problem mentioned on 2nd point, I try to control the step of capturing image to be as less as possible and try to adjust the parameter to crop the ceiling of small rest room at 1st floor, to minimize the error transformation problem, that is, result of Figure 7 perform.
3. Different ICP estimation also will cause different result for reconstruction.
4. Adjust ICP variants/parameter such as number of sampling points, data association, weight of correspondences should also improve the result. And I applied outlier points removal on this assignment to have Figure 9 result, but seems it is not good, maybe need to tune the parameter properly.

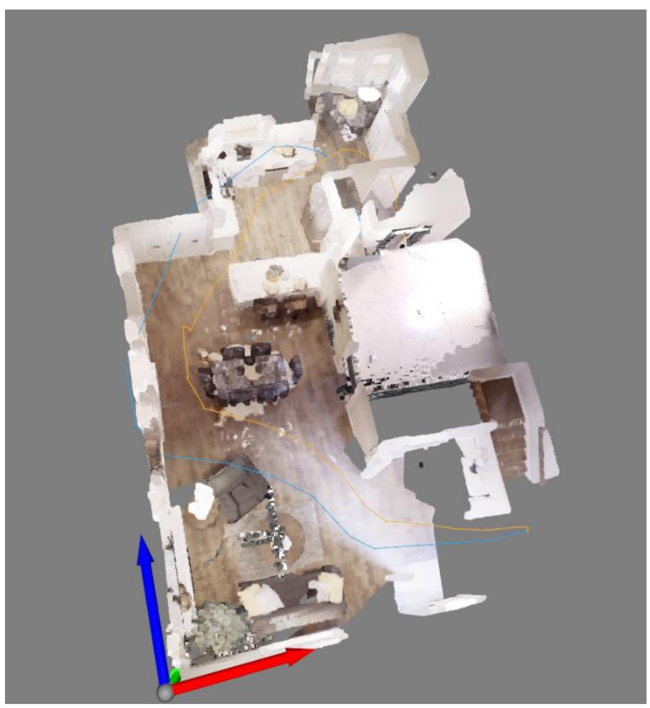

Figure 8 different parameter for own ICP implementation

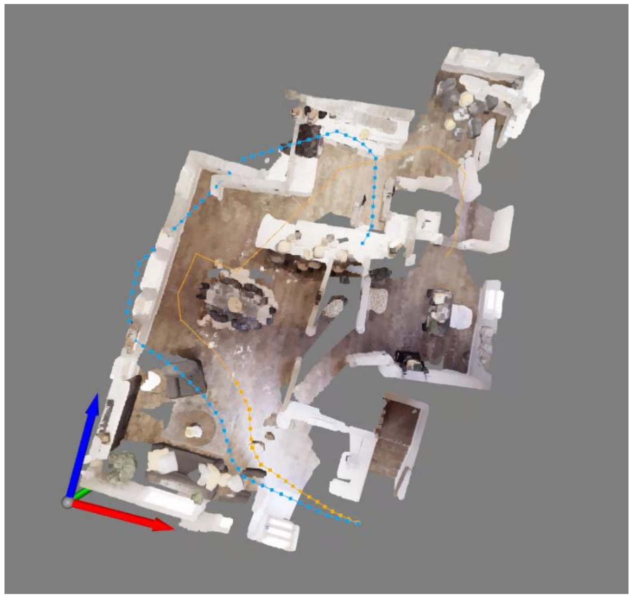

Figure 9 Removal outlier applied

```
0.0012886713370566317: 100%|██████████| 100/100 [05:58<00:00,  2.77s/it]Trajectory Error: 0.00011594319222332845
0.0012886713370566317: 100%|██████████| 100/100 [06:59<00:00,  4.19s/it]

Process finished with exit code 0
```

```
icp_implement:   0%|          | 0/1000 [00:00<?, ?it/s]
error:  0 1.5537771174205638e-09
error below tolerance!
0.001416354068393428: 100%|██████████| 100/100 [08:49<00:00,  3.35s/it]Trajectory Error: 0.00017593338723000934
```

In the graph we can see the ground truth trajectory and estimate trajectory, I found out that the error between it is quite a lot. I believe this ground truth trajectory should have much more meaning on it, for example, it can be used as a reference to adjust our trajectory and also to scale the reconstruction correctly. However, I don’t have enough time for prove my though.

Discussion:

This homework can familiar with coordinate transformation, from image to camera and from camera to global coordinate. I believe that there are many ways to improve or optimize the result, e.g. filter out irrelative points, limit simulator every step for capture a close relative image and using others new ICP method, etc.

open3d tool is not stable, it wastes a lot of time on solving function name change or bugs.

Please note that the code on the report may not be the latest code, the latest python code should reference on reconstruct.py file.

Perception and Decision Making in Intelligent Systems
Homework 1

310551003 陳嘉慶

Reference:

https://github.com/facebookresearch/habitat-lab

https://askubuntu.com/questions/1408367/installing-docker-getting-predepends-init-system-helpers-1-54-but-1-51

https://stackoverflow.com/questions/72299444/docker-desktop-doesnt-install-saying-docker-ce-cli-not-installable

https://askubuntu.com/questions/422950/root-folder-access-via-gui

https://aihabitat.org/docs/habitat-lab/

https://aihabitat.org/docs/habitat-sim/

https://colab.research.google.com/github/facebookresearch/habitat-sim/blob/main/examples/tutorials/colabs/ECCV_2020_Navigation.ipynb#scrollTo=gAjK9DIT1pMK

http://paulbourke.net/miscellaneous/lens/

chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.cse.psu.edu/~rtc12/CSE486/lecture12.pdf

chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.cse.psu.edu/~rtc12/CSE486/lecture13.pdf

https://github.com/DanielsKraus/SLAM-python/blob/master/3.%20Landmark%20Detection%20and%20Tracking.ipynb

https://blog.csdn.net/weixin_44345862/article/details/114298124

https://github.com/DanielsKraus/SLAM-python

https://github.com/DLR-RM/SingleViewReconstruction

https://zhuanlan.zhihu.com/p/37023649

https://stackoverflow.com/questions/63882698/how-to-estimate-intrinsic-properties-of-a-camera-from-data

https://github.com/facebookresearch/habitat-sim/issues/402

https://medium.com/mlearning-ai/a-single-camera-3d-functions-fdec7ffa9a83

https://towardsdatascience.com/camera-intrinsic-matrix-with-example-in-python-d79bf2478c12

Perception and Decision Making in Intelligent Systems
Homework 1

310551003 陳嘉慶

https://blog.csdn.net/weixin_36219957/article/details/106346469

http://www.open3d.org/docs/release/tutorial/geometry/pointcloud.html

https://blog.csdn.net/weixin_42488182/article/details/105196148

http://www.open3d.org/docs/release/tutorial/geometry/file_io.html

https://gist.github.com/Shreeyak/9a4948891541cb32b501d058db227fff

https://www.796t.com/content/1545305955.html

http://www.open3d.org/docs/release/python_api/open3d.geometry.PointCloud.html

https://my-it-notes.com/2014/09/how-to-convert-png-pair-of-rgb-and-depth-frames-into-pointcloud-library-pcd-format/

https://gist.github.com/danielkochdakitec/a7f68de0aff07ebf85732f4830e7a229

http://pointclouds.org/documentation/tutorials/pcd_file_format.html

https://blog.csdn.net/qq_44116315/article/details/126343848

https://github.com/isl-org/Open3D/issues/2901

https://blog.csdn.net/io569417668/article/details/112859104

http://www.open3d.org/docs/0.8.0/python_api/open3d.registration.registration_icp.html

https://blog.csdn.net/u014072827/article/details/113788879

https://www.twblogs.net/a/5b80b23a2b71772165a8ca6a

https://stackoverflow.com/questions/20120384/iterative-closest-point-icp-implementation-on-python

https://blog.csdn.net/u011736771/article/details/85103293

https://blog.51cto.com/domi/2988173

https://github.com/richardos/icp/blob/master/icp.py

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NearestNeighbors.html

http://nghiaho.com/?page_id=671

https://medium.com/machine-learning-world/linear-algebra-points-matching-with-svd-in-3d-space-2553173e8fed

https://www.youtube.com/watch?v=MJAvyt9v0g4&ab_channel=SteveBrunton

https://colab.research.google.com/drive/1pZPV5GuR7ZPW6vyoG8DnawJx13k6zHmD

Perception and Decision Making in Intelligent Systems
Homework 1

310551003 陳嘉慶

https://blog.csdn.net/baidu_40840693/article/details/103084646

http://www.open3d.org/docs/release/tutorial/visualization/non_blocking_visualization.html

https://github.com/isl-org/Open3D/issues/2291

https://github.com/isl-org/Open3D/issues/483

http://www.open3d.org/docs/0.7.0/tutorial/Basic/visualization.html