16-833A Robot Localisation and Mapping, Spring 2023
Homework 4: Dense SLAM with Point-based Fusion

Sushanth Jayanth

Andrew ID: sushantj

April 17, 2023

Contents

float

1 Overview

In this assignment, we're working on implementing dense SLAM. Dense SLAM aims to create a dense 3D map of an environment in real-time while simultaneously tracking the camera or sensor's pose within that environment.

We implement SLAM in three stages:

- Localization: projective ICP, a common implementation of visual odometry that is similar to (2).
- Mapping: point-based fusion, a simplified map fusion algorithm (similar to (3)).
- SLAM: glue the components and build the entire system.

We are also given a dataset to test out the performance of our dense SLAM method. This dataset (1) has RGB-D images (multiple smooth frames) of synthetically generated environmnets. Additionally, the ground truth poses for each of these frames are also known.

Note. the dataset is used to compare our accuracy in the last step of dense slam. In all intermediate steps, such as in ICP, we only use the vertex map of the dataset, but the loss which we report is just the b-vector in $Ax = b$ after the linearization process.

In Dense SLAM we also get our vertex map, normal map (a.k.a. source map and source normals) from the dataset.

2 Iterative Closest Point (ICP)

2.1 Projective Data Association

In Kinect fusion, we find correspondence points by finding the projective nearest neighbors. This is more effective than searching in a KDTree used in conventional pointcloud ICP.

Projective nearest neighbors is similar to finding correspondences in computer vision's triangulation (see code to understand the steps better).

2.1.1 Question 1

Q. Suppose you have projected a point p to a vertex map and obtained the u, v coordinate with a depth d. The vertex map's height and width are H and W . Write down the conditions u, v, d must satisfy to setup a valid correspondence.

Ans. Here we take a target point (3D point) and project it onto the target's image plane.

We use the simple pinhole projection equations of:

$$x = \frac{fX}{Z}$$
$$y = \frac{fY}{Z}$$

The above is slightly different in code since they also include the intrinsics in finding the projection onto image plane.

Now, since we have source points (3D), target points (3D) and we've projected target points onto an image plane (**i.e. vertex map**), we can then define the criteria which each such projected point should satisfy:

$$0 \leq u < W \quad (1)$$
$$0 \leq v < H \quad (2)$$
$$0 \leq d \quad (3)$$

This first_filter is also implemented in code.

2.1.2 Question 2

Q. After obtaining the correspondences q from the vertex map and the correspond- ing normal n q in the normal map, you will need to additionally filter them by distance thresholds so that $\|p - q\| < d_{thr}$. Why is this step necessary?

Ans. This check is necessary to ensure that the projective nearest neighbors are actually close in 3D.

One might come across the case where a very far away 3D point projects onto the vertex map (very near to the source point's projection). In such cases, even though the projection looks similar, the actual euclidean distance between two points in 3D space would be huge. Therefore, such two points should not be considered correspondences.

This second_filter is also implemented in code in the 'find_projective_correspondences' function

2.2 Linearization

KinectFusion seeks to minimize the point-to-plane error between associated points (p, q) where p is from the source and q is from the target.

The Error function is originally defined as:

$$\sum_{i\in\Omega} r_i^2(R,t) = \left\| n_{q_i}^\top \left(Rp_i + t - q_i\right)\right\|^2 \tag{4}$$

However, we linearize the error function to account for small displacements (small angle approximation). Additionally, we parameterize the small rotation as:

$$\delta R = \begin{bmatrix} 1 & -\gamma & \beta \\ \gamma & 1 & -\alpha \\ -\beta & & 1 \end{bmatrix} \tag{5}$$

Based on the above changes, the error function then becomes:

$$\sum_{i\in\Omega} r_i^2(\delta R,\delta t) = \left\| n_{q_i}^\top \left((\delta R)p'_i + \delta t - q_i\right)\right\|^2 \tag{6}$$

Where $p'_i = R^0 p_i + t^0$. We then solve for $\alpha, \beta, \gamma, t_x, t_y, t_z$ using the initial R^0 and t^0 estimates.

2.2.1 Question 1

Q. Now reorganize the parameters and rewrite $r_i(\delta R, \delta t)$ in the form of:

$$r_i(\alpha,\beta,\gamma,t_x,t_y,t_z) = A \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ t_x \\ t_y \\ t_z \end{bmatrix} + b_i \tag{7}$$

Where A_i is a 1 x 6 matrix and b is a scalar.

Ans. We expand $p'_i, \delta t,$ and q_i to be:

$$p'_i = \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix}, \delta t = \begin{bmatrix} \delta t_x \\ \delta t_y \\ \delta t_z \end{bmatrix}, q_i = \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix} \tag{8}$$

We start by solving the right hand side of the equation:

$$n_{q_i}^\top \left((\delta R)p'_i + \delta t - q_i\right) \tag{9}$$

$$\begin{bmatrix} n_1 & n_2 & n_3 \end{bmatrix} \begin{bmatrix} \beta p'_z - \gamma p'_y + t_x - q_x + p_x \\ \gamma_{p'_x} - \alpha p'_z + t_y - q_y + p_y \\ \alpha p'_y - \beta p'_x + t_z - q_z + p_z \end{bmatrix}$$

$$(n_1p'_z - n_3p'_x) + \gamma(-n_1p'_y + n_2p'_x) + \alpha(n_3p'_y - n_2p'_z) + n_1t_x + n_2t_y + n_3t_z + n_1(p'_x - q_x) + n_2(p'_y - q_y) + n_3(p'_2 - q_z) \tag{10}$$

Now, writing the equation 10 in linear matrix form of $Ax = b$ we get:

$$\begin{bmatrix} -n_2p'_z + n_3p'_y \\ n_1p'_z - n_3p'_x \\ -n_1p'_y + n_2p'_z \\ n_1 \\ n_2 \\ n_3 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ t_x \\ t_y \\ t_z \end{bmatrix} = n'_1\left(p'_x - q_n\right) + n'_2\left(p'_y - q_y\right) + n'_3\left(p'_z - q_z\right) \tag{11}$$

Where:

$$A = \begin{bmatrix} -n_2p'_z + n_3p'_y \\ n_1p'_z - n_3p'_x \\ -n_1p'_y + n_2p'_z \\ n_1 \\ n_2 \\ n_3 \end{bmatrix}$$

$$b = n'_1\left(p'_x - q_n\right) + n'_2\left(p'_y - q_y\right) + n'_3\left(p'_z - q_z\right)$$

Now, the first three rows of the A matrix can actually be written as a matrix multiplication between a skew symmetric matrix $[p']_x$ and n_{q_i}. This is also used in code.

$$\begin{bmatrix} -n_2p'_z + n_3p'_y \\ n_1p'_z - n_3p'_x \\ -n_1p'_y + n_2p'_z \end{bmatrix} = [p']_\times n_{q_i} \tag{12}$$

Where,

$$[p']_\times = \begin{bmatrix} 0 & -p'_z & p'_y \\ p'_z & 0 & -p'_x \\ -p'_y & p'_x & 0 \end{bmatrix}, p' = \left[p'_x, p'_y, p'_z\right]^\top \in \mathbb{R}^3$$

2.3 Optimization

Now, if we have n correspondences between two frames and we want to optimze the δR and δt according to our linearization step above. The function to minize is then:

$$\sum_{i=1^n} r_i^2(\alpha, \beta, \gamma, t_x, t_y, t_z) = \sum_{i=1}^{n} \left\| A_i \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ t_x \\ t_y \\ t_z \end{bmatrix} + b_i \right\|^2 \tag{13}$$

2.3.1 Question 1

Q. Write down the linear system that provides a closed form solution of , , , t x , t y , t z in terms of A i and b i . You may either choose a QR formulation by expanding a matrix and filling in rows (resulting in a n × 6 linear system), or a LU formulation by summing up n matrices (resulting in a 6 × 6 system). Implement build linear system and the corresponding solve with your derivation.

Ans. The derivation for the linearization is shown in the previous question.

$$\begin{bmatrix} -n_2 p'_z + n_3 p'_y \\ n_1 p'_z - n_3 p'_x \\ -n_1 p'_y + n_2 p'_z \\ n_1 \\ n_2 \\ n_3 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ t_x \\ t_y \\ t_z \end{bmatrix} = n'_1 (p'_x - q_n) + n'_2 (p'_y - q_y) + n'_3 (p'_z - q_z) \tag{14}$$

Where,

$$A = \begin{bmatrix} -n_2 p'_z + n_3 p'_y \\ n_1 p'_z - n_3 p'_x \\ -n_1 p'_y + n_2 p'_z \\ n_1 \\ n_2 \\ n_3 \end{bmatrix}$$

$$b = n'_1 (p'_x - q_n) + n'_2 (p'_y - q_y) + n'_3 (p'_z - q_z)$$

For the solver, QR decomposition was used (similar to what was implemented in the previous assignment).

Note. In the previous assignment we solved for $Ax - b$, but here we solve for $Ax = b$. I presume this is why my code only worked when I multiplied the last three elements of A vector by -1

```python
def solve(A, b):
    '''
    \param A (6, 6) matrix in the LU formulation, or (N, 6) in the QR formulation
    \param b (6, 1) vector in the LU formulation, or (N, 1) in the QR formulation
    \return delta (6, ) vector by solving the linear system. You may directly use dense solvers
        from numpy.
    '''
    # TODO: write your relavant solver
    # Using the QR solver from the previous assignment
    N = A.shape[1]
    x = np.zeros((N, ))
    R = np.eye(N)

    # convert A to a sparse matrix in COO format
    A_coo = scipy.sparse.coo_matrix(A)

    # rz gives the upper triangular part
    Z, R ,_ ,_ = rz(A_coo, b, permc_spec='NATURAL')
    x = spsolve_triangular(R,Z,lower=False)
    return x
```

2.3.2 Question 2

Q. Report your visualization before and after ICP with the default source and target (frame 10 and 50). Then, choose another more challenging source and target by yourself (e.g., frame 10 and 100) and report the visualization. Analyze the reason for its failure or success.

Ans. Frame 10 and 50:

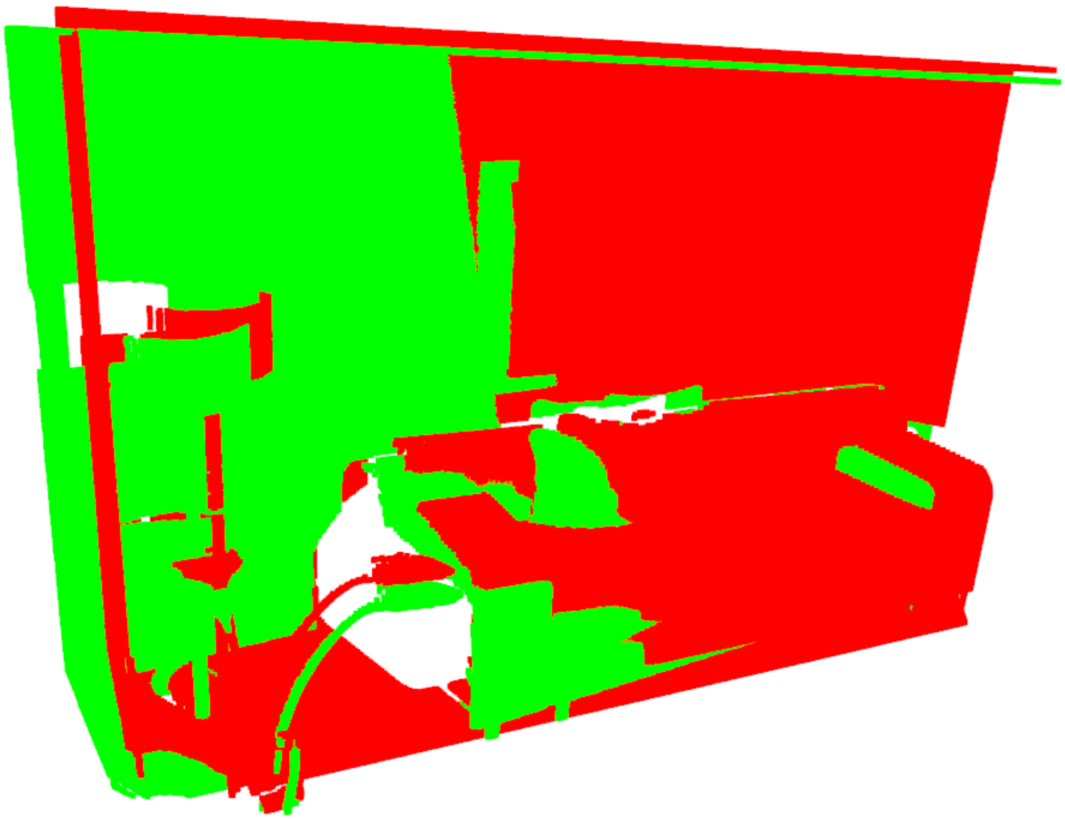

Figure 1: Frame 10 and 50 **before** ICP

Frame 10 and 100:

The algorithm fails to converge for frames 10 and 100. This is because the frames are too far apart and the small angle approximation may not hold well.

However, increasing the number of iterations seems to make it better.

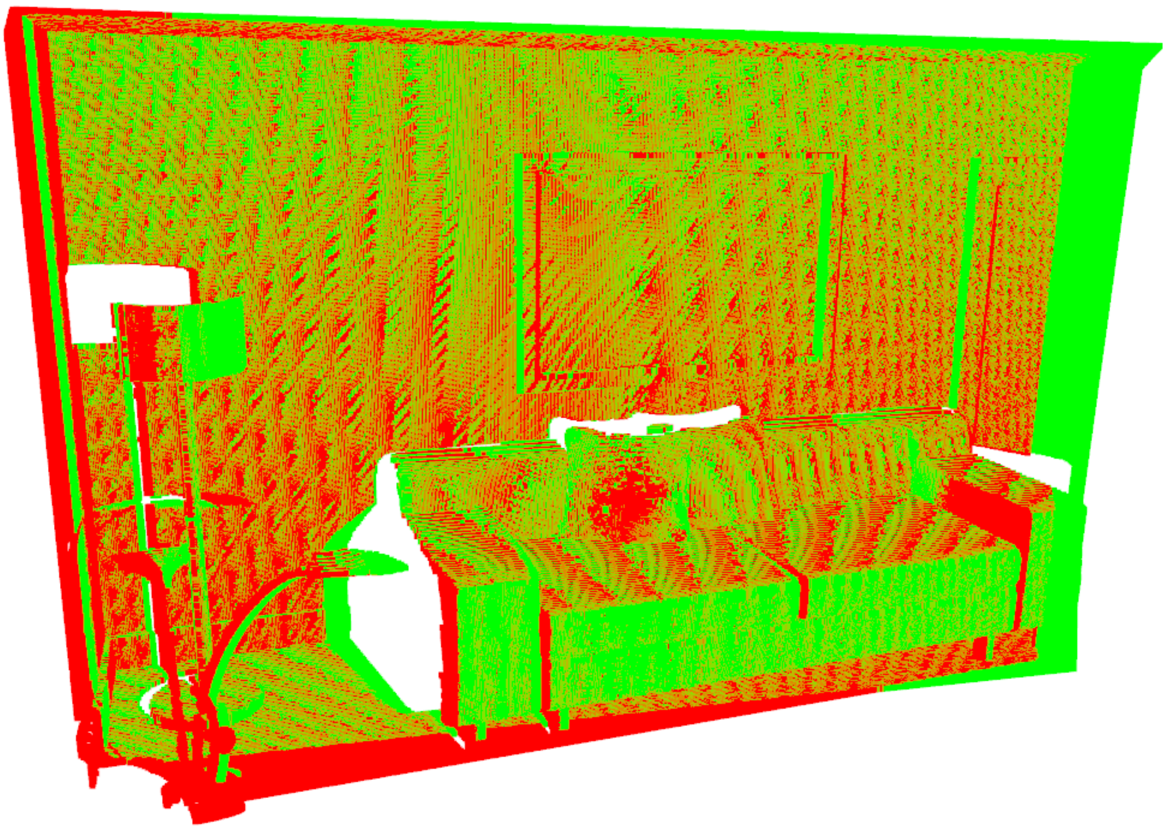

Figure 2: Frame 10 and 50 **after** ICP

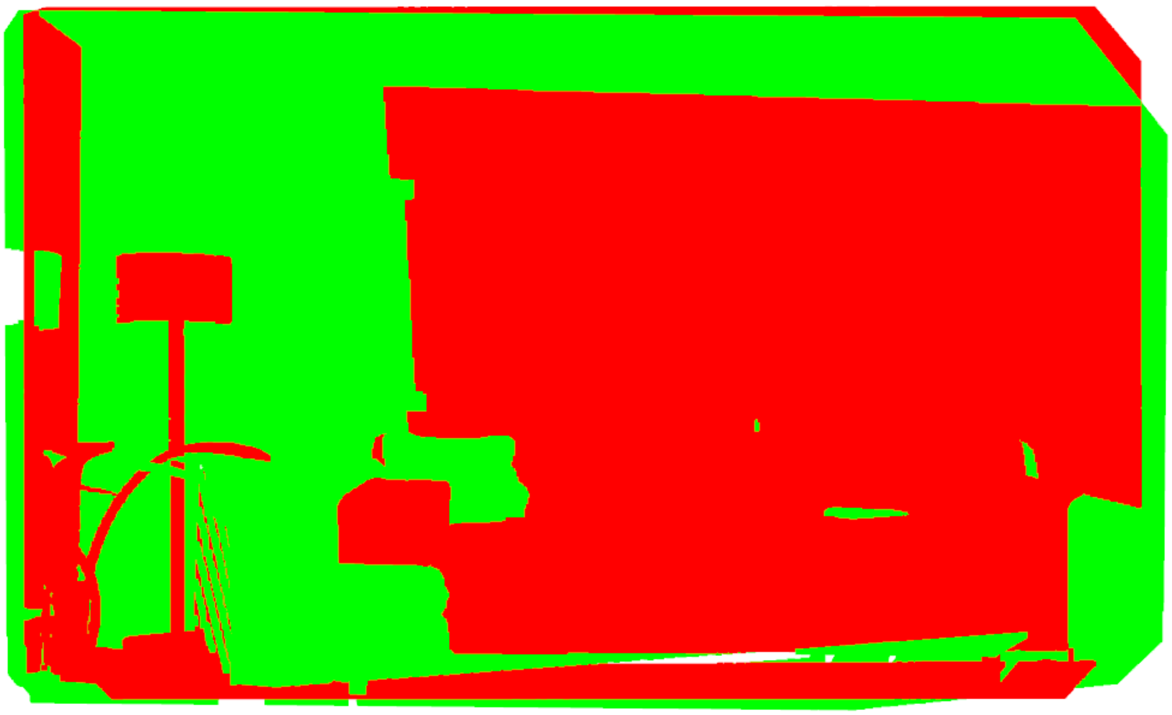

Figure 3: Frame 10 and 100 **before** ICP

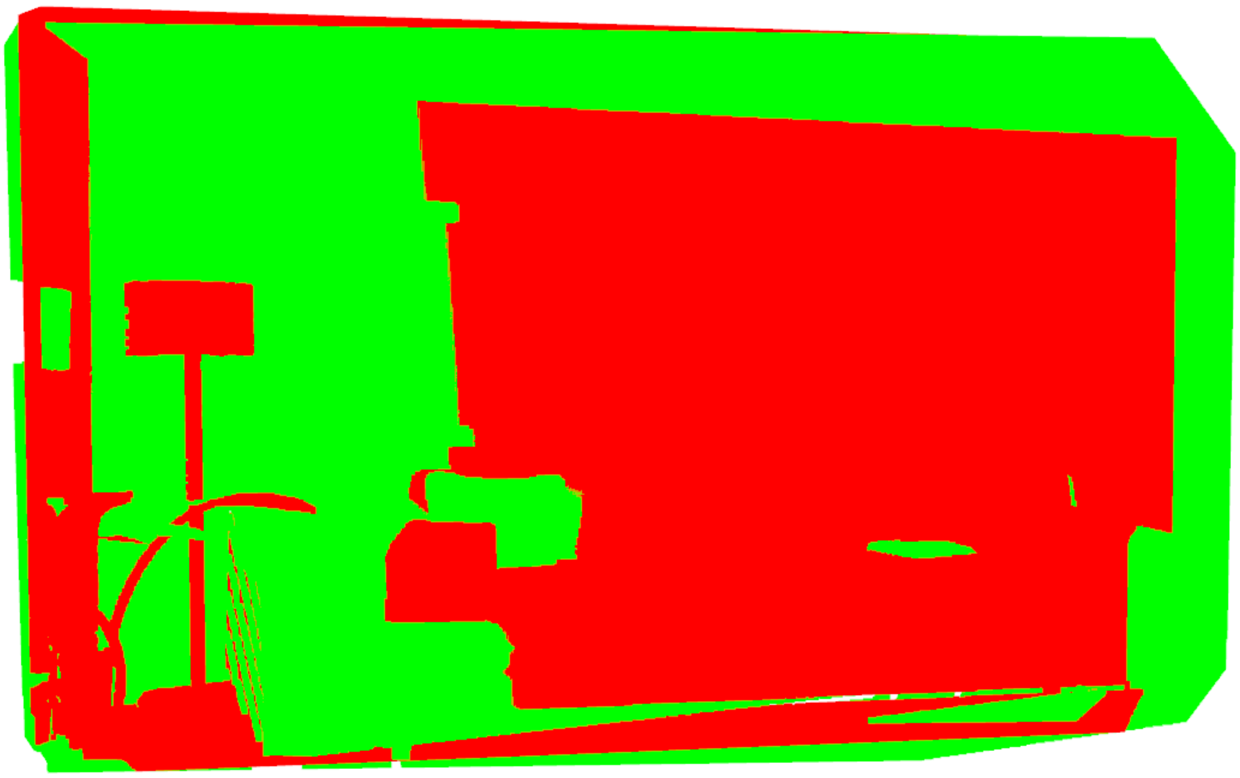

Figure 4: Frame 10 and 100 **after** ICP

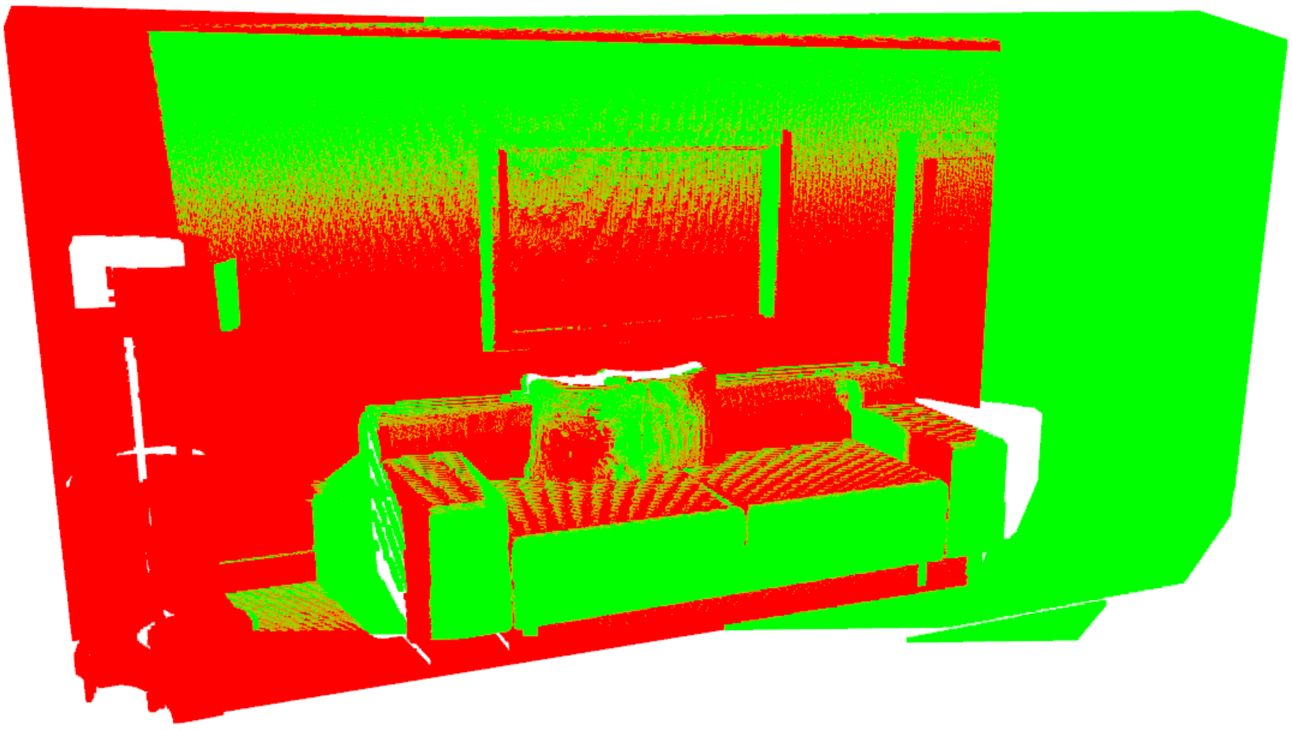

Figure 5: Frame 10 and 100 **after ICP with 80 iterations**

3 Point Based Fusion

Point-based fusion maintains a weighted point cloud and actively merges incoming points.

3.1 Filter

The filtering here is similar to the projective data association seen in ICP. Only an additional normal angle constraint will be added for stricter filtering.

3.1.1 Question 1

Q. Implement filter pass1, filter pass2 to obtain mask arrays before merging and adding input points.

Ans. Filter is implemented as shown below:

```
def filter_pass1(self, us, vs, ds, h, w):
    # NOTE: Refer the find_projective_correspondence in icp.py
    mask = np.zeros_like(us).astype(bool)
    mask = ((us >= 0) & (vs >= 0) & (ds >= 0) & (us < w) & (vs < h))
    return mask

def filter_pass2(self, points, normals, input_points, input_normals,
                 dist_diff, angle_diff):
    # mask1 = np.zeros((len(points))).astype(bool) # checks for distance threshold
    # mask2 = np.zeros((len(points))).astype(bool) # checks for normal threshold

    euclidean_dists = np.linalg.norm(points - input_points, axis=1)
    mask1 = np.where(euclidean_dists < dist_diff, True, False)

    angluar_dists = np.sum((normals * input_normals), axis=1)
    magnitude_n = np.linalg.norm(normals, axis=1)
    magnitude_inp_n = np.linalg.norm(input_normals, axis=1)
    cos_theta = angluar_dists / (magnitude_n * magnitude_inp_n)
    theta = np.abs(np.arccos(cos_theta))
    mask2 = np.where(theta < angle_diff, True, False)

    final_mask = np.logical_and(mask1, mask2)
    return final_mask
```

3.2 Merge

The merge operation updates existing points in the map by calculating a weighted average on the desired properties.

3.2.1 Question 1

Q. Given $p \in \mathbb{R}^3$ in the map coordinate system with a weight w and its corresponding point $q \in \mathbb{R}^3$ (read from the vertex map) in the frame's coordinate system with a weight 1, write down the weighted average of the positions in terms of p, q, R_c^w, t_c^w, w. Similarly, write down the weighted average of normals in terms of n_p, n_q, R_c^w, w. Implement the corresponding part in merge.

Ans. The weighted average of the position is shown below:

$$p = \frac{w \cdot p + q}{w + 1} = \frac{w \cdot p + R_c^w \cdot p + t}{w + 1}$$

The weighted average of the normals is:

$$n_p = \frac{w \cdot n_p + n_q}{w + 1} = \frac{w \cdot n_p + R_c^w \cdot n_p + t}{w + 1}$$

This new n_p vector is then normalized as well by doing $n_p = \frac{n_p}{\|n_p\|_2}$

3.3 Addition

Here we add new points (which are not yet part of the map) to the map.

3.3.1 Question 1

Q. Implement the corresponding part in add. You will need to select the unassociated points and concatenate the properties to the existing map.

Ans. This was implemented in code and the same is shown below:

```
def add(self, points, normals, colors, R, t):
    # first we'll find the new point poses using the rotation and translation given
    # we convert points to column vector to left mulitply with rotation matrix
    new_pts = (R @ points.T + t).T
    new_normals = (R @ normals.T).T

    # make the updates
    self.points = np.vstack((self.points, new_pts))
    self.normals = np.vstack((self.normals, new_normals))
    self.weights = np.concatenate((self.weights, np.ones((len(points), 1))))
    self.colors = np.concatenate((self.colors, colors))
```

3.4 Results

3.4.1 Question 1

Q. Report your visualization with a normal map, and the final number of points in the map. Estimate the compression ratio by comparing the number of points you obtain and the number of points if you naively concatenate all the input. Note to speed up we use a downsample factor 2 for all the input.

Ans. The final number of points after point-fusion was 1362143 after 200 steps.

The compression ratio can be estimated by noting the total number of pixels in every frame multiplied by the number of frames. This can then be compared to our original number.

$$CompressionRatio = \frac{1362143}{640 * 480 * 200} = 0.022 \tag{15}$$

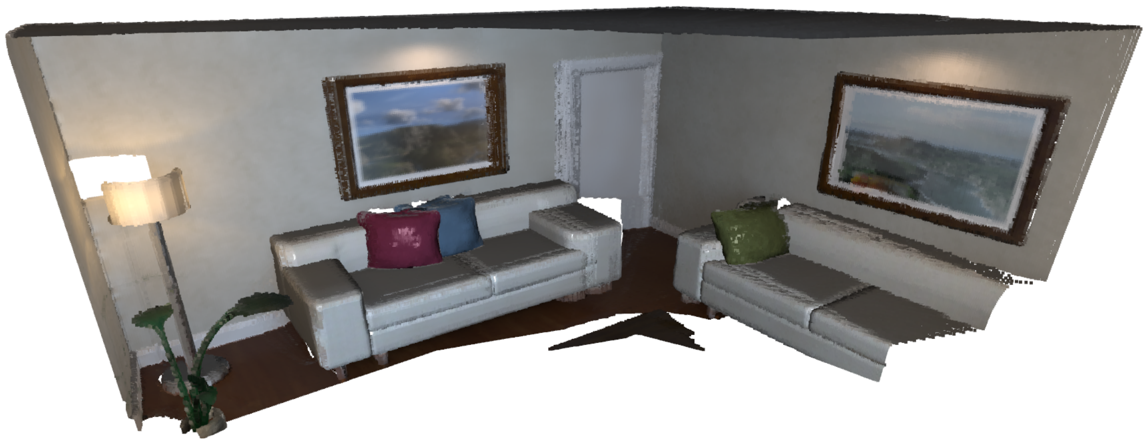

Figure 6: Point-Fusion output PCD

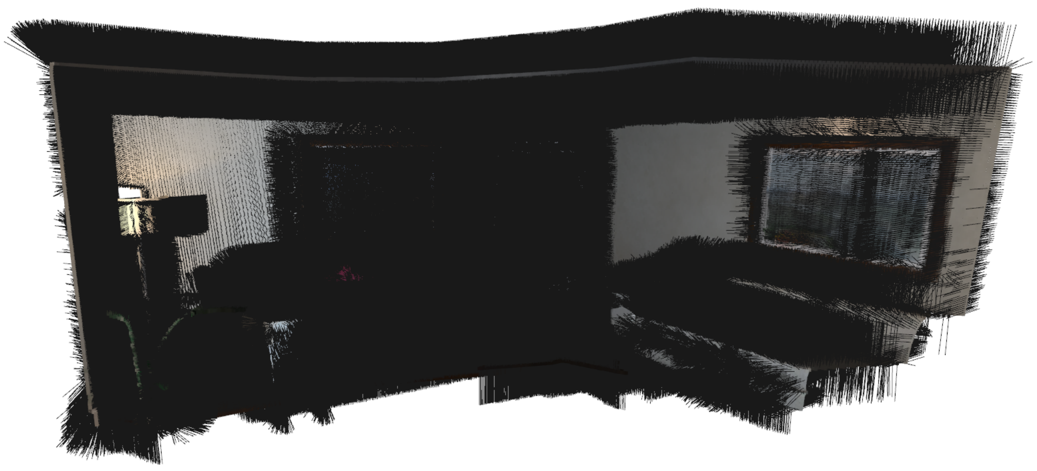

Figure 7: Point-Fusion output Normal Map

4 The Dense SLAM System

Now we put together both ICP and point-fusion implemented in the main code.

4.0.1 Question 1

Q. Which is the source and which is the target, for ICP between the map and the input RGBD-frame? Can we swap their roles, why or why not?

Ans. The source frame is RGB-D and the target is the map (vertex_map and normal_map). This was inferred by looking at the code in main.py as shown below. Additionally, we cannot swap their roles, since the source frames undergoes certain rotations and translations to give the target at each sequence. However, the same will not be true if we reverse their roles.

```python
intrinsic_struct = o3d.io.read_pinhole_camera_intrinsic('intrinsics.json')
    intrinsic = np.array(intrinsic_struct.intrinsic_matrix)
    indices, gt_poses = load_gt_poses(
        os.path.join(args.path, 'livingRoom2.gt.freiburg'))

    rgb_path = os.path.join(args.path, 'rgb')
    depth_path = os.path.join(args.path, 'depth')
    normal_path = os.path.join(args.path, 'normal')

    # TUM convention
    depth_scale = 5000.0

    m = Map()

    down_factor = args.downsample_factor
    intrinsic /= down_factor
    intrinsic[2, 2] = 1

    # Only use pose 0 for 1-th frame for alignment.
    # DO NOT use other gt poses here
    T_cam_to_world = gt_poses[0]

    T_gt = []
    T_est = []
    for i in range(args.start_idx, args.end_idx + 1):
        print('loading frame {}'.format(i))
        depth = o3d.io.read_image('{}/{}.png'.format(depth_path, i))
        depth = np.asarray(depth) / depth_scale
        depth = depth[::down_factor, ::down_factor]
        vertex_map = transforms.unproject(depth, intrinsic)

        color_map = np.asarray(
            o3d.io.read_image('{}/{}.png'.format(rgb_path,
                                                 i))).astype(float) / 255.0
        color_map = color_map[::down_factor, ::down_factor]

        normal_map = np.load('{}/{}.npy'.format(normal_path, i))
        normal_map = normal_map[::down_factor, ::down_factor]

        if i > 1:
            print('Frame-to-model icp')
            T_world_to_cam = np.linalg.inv(T_cam_to_world)
            # T_world_to_cam is the transformation which maps world_frame coords to cam_frame
            """
            In the ICP function the 1st and 2nd arguments = source points, source normals
                                 3rd and 4th arguments = target vertex map, target normals

            the m.points and m.normals below are RGB-D frame points and normals (viz our SOURCE)
            the vertex_map and normal map are from dataset (viz TARGET here).

            Note: looks like vertex_map is a function of depth_data + intrinsics
            looks like normal_map is purely obtained from the dataset
        """
        T_world_to_cam = icp(m.points[::down_factor],
                             m.normals[::down_factor],
                             vertex_map,
                             normal_map,
                             intrinsic,
                             T_world_to_cam,
                             debug_association=False)
        T_cam_to_world = np.linalg.inv(T_world_to_cam)
    print('Point-based fusion')
    m.fuse(vertex_map, normal_map, color_map, intrinsic, T_cam_to_world)

    # A shift is required as gt starts from 1
    T_gt.append(gt_poses[i - 1])
    T_est.append(T_cam_to_world)
```

4.0.2 Question 2

Q. Report your visualization. Also, record the estimated trajectory and compare against the ground truth.

Ans. Vizualization is shown below:

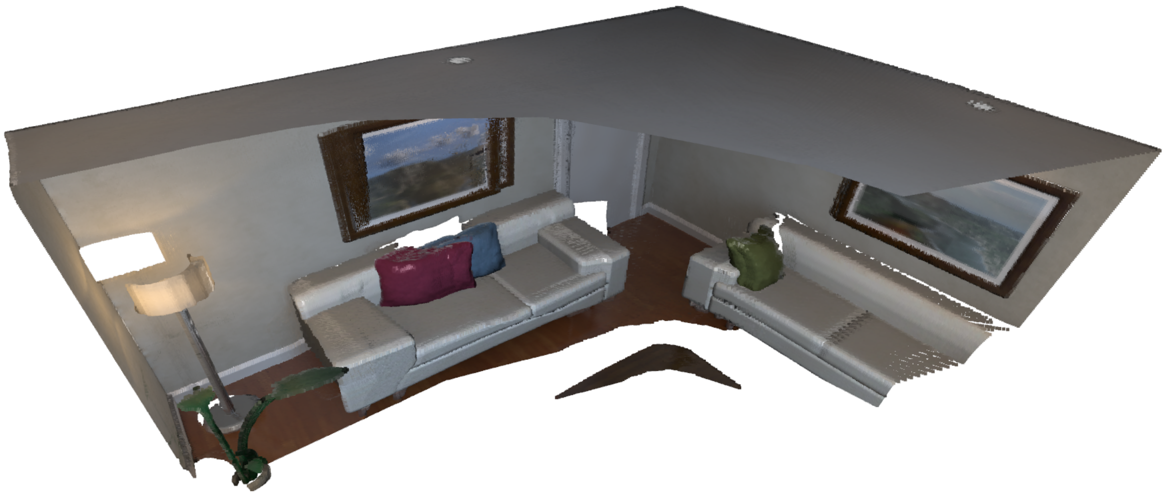

Figure 8: Dense SLAM Output View 1

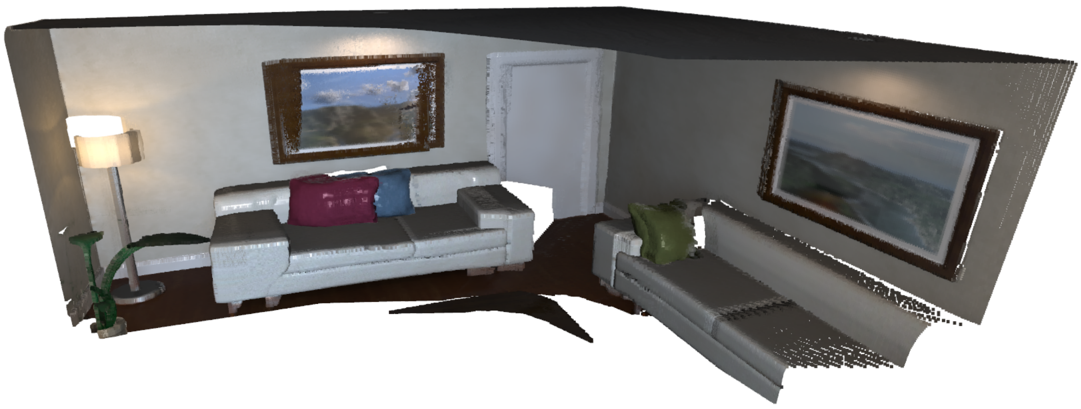

Figure 9: Dense SLAM Output View 2

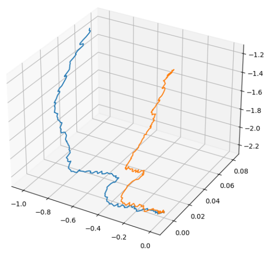

Figure 10: Estimated Trajectory vs Ground Truth

References

[1] Ankur Handa, Thomas Whelan, John McDonald, and Andrew J. Davison. A benchmark for rgb-d visual odometry, 3d reconstruction and slam. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1524–1531, 2014.

[2] Maik Keller, Damien Lefloch, Martin Lambers, Shahram Izadi, Tim Weyrich, and Andreas Kolb. Real-time 3d reconstruction in dynamic scenes using point-based fusion. In *2013 International Conference on 3D Vision - 3DV 2013*, pages 1–8, 2013.

[3] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136, 2011.