16-833: Robot Localization and Mapping, Spring 2021

Homework #4
Dense SLAM with Point-based Fusion

Due: Wednesday April 21, 11:59pm, 2021

Your homework should be submitted as **a typeset PDF file** with **a folder of Python code (and/or output results)**. Please fill in the 'TODO' sections in the provided python skeleton. If you have questions, please post them on Piazza or come to office hours. Do not post solutions or codes on Piazza. Discussion is allowed, but each student must write and submit his or her own solution. Note that you should list the name and Andrew ID of each student you have discussed with in the first page of your PDF file. You have a total of 3 late days, use them wisely. Good luck and have fun!

1 Overview

In this homework you will be implementing a 3D dense SLAM system using point-based fusion, which can be regarded as a simplified version of [1]. There are mainly three steps to setup the system:

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

Generally SLAM is a tightly coupled system and hard to debug. We simplify the problem by giving you the ground truth poses for testing. In the ICP step, you will be working on two RGB-D frames (without a map). You may verify your implementation with the loss and the visualization. In the map-based fusion step, you will be working on a sequence of RGB-D frames with ground truth, and generate a fused map. Finally, you may test your algorithm by putting everything together.

You will be testing the system on the synthethic ICL-NUIM [3] dataset. Unlike real-world data, you don't have to care about holes and noise, which can be tedious to deal with. **Please download the dataset from this link**. Prior to running the system, use `python preprocess.py /path/to/dataset` to generate normal maps.

2 Iterative Closest Point (ICP) (50 points)

Please read Section 5 in [1] before working on the following problems. Note, many equations in [1] are using unconventional parameterizations, and the notations are not easy to follow. We provide our derivation below as a reference.

In general, as its name indicates, ICP runs iteratively. Here is the general flow:

1. Setup data correspondences between source and target point clouds given the current pose $[R^t \mid t^t]$ (usually initialized with identity).
2. Linearize the loss function at $[R^t \mid t^t]$ and solve for incremental update $[\delta R \mid \delta t]$.
3. Update by left multiplying the incremental transformation $[R^{t+1} \mid t^{t+1}] = [\delta R \mid \delta t][R^t \mid t^t]$.
4. Go back to step 1 until convergence.

You will be working on `icp.py` in this section.

2.1 Projective data association

Data association is critical for point cloud registration, since the overall spirit is to push together correspondent points. In the conventional point cloud ICP, this is achieved by nearest neighbor search, usually depending on a KDTree. Although there are various fast implementations, building and searching in a KDTree can be relatively slow.

In KinectFusion, a more efficient variation, projective data association is used. Suppose the target point cloud is in the form of a `vertex_map` (where each pixel correpsonds to a point in 3D), then naturally there is an indexable geometry layout. Now given a point p, instead of searching for q in the 3D space, after initial transformation, we can project p to the 2D image and obtain the pixel coordinate (u, v), and directly read the corresponding point q from the `vertex_map`. In this case, p and q are not strictly nearest neighbors in 3D, but projective nearest neighbors. In practice, this is very efficient and works pretty well.

In case you are unfamiliar with projective pinhole camera model, we have provided you with the `project` function.

Question (5 points): 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. Also implement the `TODO : first filter` section in function `find_projective_correspondence`.

After obtaining the correspondences q from the vertex map and the corresponding normal n_q in the normal map, you will need to additionally filter them by a distance threshold, so that $|p - q| < d_{thr}$.

Question (5 points): Why is this step necessary? Implement this filter in `TODO : second filter` section in function `find_projective_correspondence`.

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 can be written by:

$$\sum_{i \in \Omega} r_i^2(R, t) = \left\| n_{q_i}^\top (Rp_i + t - q_i) \right\|^2, \tag{1}$$

where (p_i, q_i) is the i-th associated point pair in the assocation set $\Omega = \{(p_n, q_n)\}$, and n_{q_i} is the estimated normal at q_i (we have covered the data association step, i.e., how to associate p_i and q_i).

It is hard to directly solve $R \in SO(3)$. So we rely on the small-angle assumption by parameterizing δR with

$$\delta R = \begin{bmatrix} 1 & -\gamma & \beta \\ \gamma & 1 & -\alpha \\ -\beta & \alpha & 1 \end{bmatrix}. \tag{2}$$

and also introduce $\delta t = [t_x, t_y, t_z]$. Note this parameterization is unlike [2] and is closer to the conventional Lie Algebra's formulation [4].

With δR and δt, we can write down the incremental update version

$$\sum_{i \in \Omega} r_i^2(\delta R, \delta t) = \left\| n_{q_i}^\top \Big((\delta R) p_i' + \delta t - q_i \Big) \right\|^2, \tag{3}$$

where $p_i' = R^0 p_i + t^0$ aiming at solving $\alpha, \beta, \gamma, t_x, t_y, t_z$ with the given initial R^0, t^0.

Question (15 points): Now reorganize the parameters and rewrite $r_i(\delta R, \delta t)$ in the form of

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

where A_i is a 1×6 matrix and b_i is a scalar.

Note: You don't have to explicitly write down all the entries, and feel free to introduce intermediate variables. A cross-product operator may be useful here:

$$[w]_\times = \begin{bmatrix} 0 & -w_2 & w_1 \\ w_2 & 0 & -w_0 \\ -w_1 & w_0 & 0 \end{bmatrix}, w = [w_0, w_1, w_2]^\top \in \mathbb{R}^3.$$

2.3 Optimization

The aforementioned step in fact does the linearization. Now suppose we have collected n associated pairs, we move on to optimize

$$\sum_{i=1^n} r_i^2(\alpha, \beta, \gamma, t_x, t_y, t_z) = \sum_{i=1}^{n} \left\| A_i \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ t_x \\ t_y \\ t_z \end{bmatrix} + b_i \right\|^2. \tag{4}$$

Question (15 points): Write down the linear system that provides a closed form solution of $\alpha, \beta, \gamma, t_x, t_y, t_z$ 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 $n \times 6$ 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.

By solving the linear system you derived above, you can obtain the incremental transformation update in the tangent space. This 6D vector can be converted to a 4×4 matrix by `pose2transformation` provided by us.

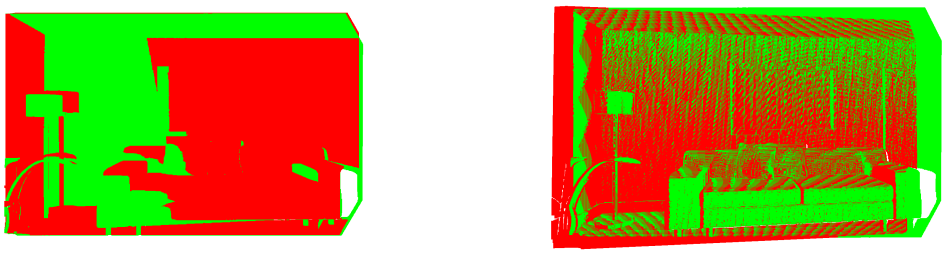

Figure 1: Point clouds before and after registration.

Now, run `python icp.py /path/to/dataset`. If everything works fine, you will find ICP's loss converges in less than 10 iterations and the inlier count increases correspondingly. The two point clouds fit better, resulting in an interleaved pattern shown in Fig. 1.

Question (10 points): 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.

Note: Press P to take a screenshot.

3 Point-based Fusion (40 points)

Please read Section 4 in [1] before working on the following problems.

KinectFusion uses volumetric TSDF as the map representation, which is non-trivial to implement. Point-based fusion, on the otherhand, maintains a weighted point cloud and actively merges incoming points, hence is easy to implement.

The essential operations of point-based fusion are very similar to projective data association. Given an estimated pose, we first project the point p in the map to the image and obtain its corresponding (u, v) coordinates. Then, after proper filtering, we compute a weighted average of the properties (positions, normals, colors).

You will be working on `fusion.py` in this section.

3.1 Filter

Similar to projective data association, we need a transformation. In a typical SLAM system, this is obtained from accumulative ICP. In this section, however, we assume ground truth transformations are known. The transformation $T_c^w = [R_c^w \mid t_c^w]$ is from the input **c**amera frame's coordinate system to the **w**orlds's coordinate system.

With the available transformation, you need to perform filtering similar to projective data association. The only difference is that you will need to add a normal angle constraint to be more strict with filtering.

Question (5 points): Implement `filter_pass1`, `filter_pass2` to obtain mask arrays before merging and adding input points.

3.2 Merge

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

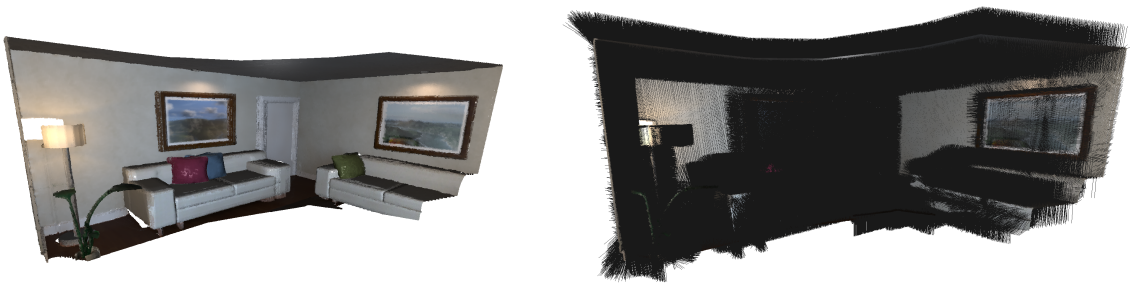

Figure 2: Fusion with ground truth poses with a normal map.

Question (15 points): 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 weight average of the positions in terms of p, q, R_c^w, t_c^w, w. Similarly, write down the weight average of normals in terms of n_p, n_q, R_c^w, w. Implement the corresponding part in `merge`. Note a normalization of normals is required after weight average.

3.3 Addition

The projective data association may cover a big portion of the input vertex map, but still there are uncovered regions. These points have to be added to the map.

Question (10 points): Implement the corresponding part in `add`. You will need to select the unassociated points and concatenate the properties to the existing map.

3.4 Results

Now run `python fusion.py /path/to/dataset`. The system will take the ground truth poses and produce the resulting image after fusing 200 frames, see Fig. 2.

Question (10 points): 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.

Note: Press N to visualize the normal map.

4 The dense SLAM system (20 points + 10 points)

Now we put together both ICP and fusion code in `main.py`. We have called the relevant functions implemented by you for your convenience.

Question (5 points): 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?

Run `python main.py /path/to/dataset`. By default, 200 frames will be tracked and mapped.

Question (15 points): Report your visualization. Also, record the estimated trajectory and compare against the ground truth. Drift is acceptable in the case, to the extend shown in Fig. 3.

Bonus (10 points): Can you try to reduce the drift? There could be several methods, from improving the RGBD odometry to improving the fusion by rejecting

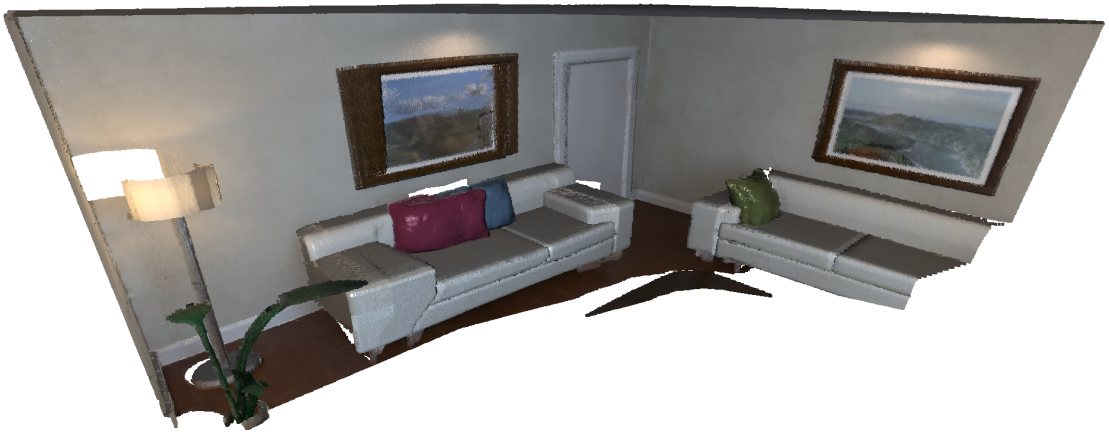

Figure 3: Fusion with poses estimated from frame-to-model ICP.

outliers and eliminating dormant points (i.e., not associated to any points for a certain period). Report your updated visualization and trajectories.

5 Code Submission Rules

Upload all of your python code in a folder. Include all your visualizations in your PDF file. Do not upload the dataset when you submit.

References

[1] Keller, Maik, et al. "Real-time 3D reconstruction in dynamic scenes using point-based fusion." *International Conference on 3D vision (3DV)*, 2013. Link: *http://ieeexplore.ieee.org/document/6599048/*.

[2] Newcombe, Richard A., et al. "KinectFusion: Real-time dense surface mapping and tracking." *10th IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, 2011. Link: *http://ieeexplore.ieee.org/document/6162880/*.

[3] Handa, Ankur, et al. "A benchmark for RGB-D visual odometry, 3D reconstruction and SLAM." *IEEE International Conference on Robotics and Automation (ICRA)*, 2014. Website: *https://www.doc.ic.ac.uk/~ahanda/VaFRIC/iclnuim.html*.

[4] Sola, Joan and Deray, Jeremie and Atchuthan, Dinesh. "A micro Lie theory for state estimation in robotics," *arXiv 2018*.