NICE-SLAM with Adaptive Feature Grids

Ganlin Zhang
ETH Zürich
zhangga@student.ethz.ch

Deheng Zhang
ETH Zürich
dezhang@student.ethz.ch

Anqi Li
ETH Zürich
lianqi@student.ethz.ch

Feichi Lu
ETH Zürich
feiclu@student.ethz.ch

Abstract

NICE-SLAM is a dense visual SLAM system that combines the advantages of neural implicit representations and hierarchical grid-based scene representation. However, the hierarchical grid features are densely stored, leading to memory explosion problems when adapting the framework to large scenes. In our project, we present sparse NICE-SLAM, a sparse SLAM system incorporating the idea of Voxel Hashing into NICE-SLAM framework. Instead of initializing feature grids in the whole space, voxel features near the surface are adaptively added and optimized. Experiments demonstrated that compared to NICE-SLAM algorithm, our approach takes much less memory and achieves comparable reconstruction quality on the same datasets. Our implementation is available at https://github.com/zhangganlin/NICE-SLAM-with-Adaptive-Feature-Grids.

1. Introduction

Simultaneous Localization and Mapping (SLAM) is a fundamental problem in 3D computer vision. It has extensive applications in domains including autonomous driving, indoor robotics, and mixed reality. Previous work introduced NICE-SLAM [8], a dense visual SLAM approach that combines the advantages of neural implicit representations and the geometry preservability of hierarchical grid-based scene representation. NICE-SLAM is proved to be real-time capable, scalable, predictive, and robust to various challenging scenarios. However, due to the dense nature of this algorithm, its depth fusion system needs large memory storage space and is difficult to scale up and to be used in outdoor settings.

In our project, we manage to tackle the issue of memory explosion in NICE-SLAM by implementing a sparse representation based on the sparse method of Voxel Hashing [3]. Our key idea is to only assign features to voxels near the surface, and only optimize the voxel features already assigned in previous keyframes. We test our framework on existing datasets ScanNet [1] and Replica [5]. The results proved that our sparse NICE-SLAM algorithm significantly saves the memory space required and at the same time achieves a comparable or even better reconstruction quality as the original NICE-SLAM system. Overall, we make the following contributions:

- We present sparse NICE-SLAM, a sparse RGB-D SLAM system that is scalable to large scenes.
- The core idea is to inherit the hierarchical, grid-based neural implicit encoding in NICE-SLAM, and integrate Voxel Hashing method to make the voxel features sparse in space.
- We conduct evaluations on the ScanNet and Replica datasets which demonstrate comparable performance and less memory requirement compared to the original NICE-SLAM algorithm.

2. Related Work

Our project is mainly based on the developed framework of NICE-SLAM [8]. NICE-SLAM is a dense RGB-D SLAM system combining hierarchical grid-based features with neural implicit representations. Neural implicit representations demonstrated promising results for geometry representation and scene completion, and display smoother surface than traditional explicit methods. Moreover, NICE-SLAM allows for local updates compared to other global neural scene encoding. The hierarchical structure can help to preserve geometric details and enable reconstructing complex scene. It is also proved to be scalable, predictive and robust according to experiments on various datasets.

Voxel Hashing [3] is a system using a simple spatial hashing scheme that compresses the space, and allows for

real-time access and updates of implicit features. Data can be streamed efficiently in or out of the hash table, allowing for further scalability. This idea is adopted by DI-Fusion [2], which is based on Probabilistic Local Implicit Voxels (PLIVoxs). The approach incorporates scene priors of geometry and uncertainty, achieving more accurate camera pose estimation and higher-quality 3D reconstruction. In our project, we use the same voxel feature storing and updating method as that in PLIVoxs (detail in Section 3.2).

3. Method

We provide an overview of the original NICE-SLAM framework, Voxel Hashing flow diagram, and schematics of dense and sparse trilinear interpolation in Figure 1, Figure 2, and Figure 3 respectively. Our sparse method inherits the NICE-SLAM pipeline and adds the adaptive sparse representation of the grid features.

3.1. Original NICE-SLAM Pipeline

This project is based on the structure of NICE-SLAM algorithm [8], which is a dense visual SLAM system combining the advantages of neural implicit representations with the scalability of a hierarchical grid-based scene representation. The pipeline of NICE-SLAM is displayed in Figure 1. The input is a video containing several keyframes of a scene. At first, a hierarchical Feature Grid is built to store the features for each grid point in the whole 3D space. There are three hierarchies of feature grid: coarse, middle and fine levels. All of these three levels encode geometric information, while the fine layer also encode color information. Starting from a camera pose, we can sample 3D points along a ray and calculate the trilinear interpolation features for each point using the neighboring gird point features. After that, these features go through a pre-trained decoder to get the occupancy of each point. For the color information, the decoder is trained alongside the network. Then we can run the color rendering process to rebuild the 2D scene at the current camera pose. Lastly, we minimize the reconstruction loss between rendered and ground truth keyframes to update the grid features.

As mentioned previously, the depth fusion system of NICE-SLAM needs a huge amount of memory which prevents it from scaling up in 3D scenes. We want to solve the memory exhaustion problem by integrating the idea of Voxel Hashing [3] in NICE-SLAM.

3.2. Voxel Hashing

We follow the idea of Voxel Hashing [3] and use the implementation method in DI-Fusion [2] to realize sparse storage. As is shown in Figure 2, we first allocate a feature position tensor, a voxel feature tensor, and a voxel position tensor. Assuming we divide the whole 3D space: $x \in [x_0, x_1], y \in [y_0, y_1], z \in [z_0, z_1]$ with grid length $l_i, i \in \{c, m, f\}$ for coarse, middle and fine levels. Let $n_x^i = \lfloor \frac{x_1 - x_0}{l_i} \rfloor, n_y^i = \lfloor \frac{y_1 - y_0}{l_i} \rfloor, n_z^i = \lfloor \frac{z_1 - z_0}{l_i} \rfloor$. The feature position tensor is initialized with size $n_x^i \times n_y^i \times n_z^i$ and values -1. The voxel feature tensor is initialized with size $d \times M$ and values -1, where d is the dimension of features for each point, and M is a relatively small number randomly chosen. The voxel position tensor is initialized with size M and values -1. Then we begin the process of adaptively assigning voxel features. If a sampled point is near the surface, we would assign features to its neighbor voxels and update the voxel feature tensor. The feature position tensor would then be updated to store the feature index of the assigned voxel, and the voxel position tensor would be updated to store the assigned voxel index. In the case that all the M columns are already assigned, we update $M \leftarrow 2M$ to continue the process.

3.3. Sparse Trilinear Interpolation

One important step is how to adapt dense trilinear interpolation [7] to a sparse one when some of the neighbor voxels of a point are not assigned. As is shown in Figure 3, for dense trilinear interpolation, we can easily find $C_{ijk}, i, j, k \in \{0, 1\}$. But in the sparse case, if $\exists i, j, k \in \{0, 1\}$ and C_{ijk} is not assigned with features, we can only use the points with features for interpolation. We create mask $M_{ijk} \in \{0, 1\}, i, j, k \in \{0, 1\}$ on each C_{ijk} with $M_{ijk} = 0$ meaning C_{ijk} is not assigned. Then we can also calculate the masks for $C_{jk}, j, k \in \{0, 1\}$ as $M_{jk} = M_{0jk} \vee M_{1jk}$, masks for $C_k, k \in \{0, 1\}$ as $M_k = M_{0k} \vee M_{1k}$, and mask for C as $M = M_0 \vee M_1$. Similar to dense trilinear interpolation, we first calculate the relative coordinates: $x_d = \frac{x - x_0}{x_1 - x_0}, y_d = \frac{y - y_0}{y_1 - y_0}, z_d = \frac{z - z_0}{z_1 - z_0}$. Then we calculate the features for each C_{jk}.

$$f_{jk} = M_{jk} \cdot \frac{f_{0jk} M_{0jk} (1 - x_d) + f_{1jk} M_{1jk} x_d}{M_{0jk}(1 - x_d) + M_{1jk} x_d} \tag{1}$$

Here f denote the value of features at each voxel C. Then, we can also calculate the features for each C_k and finally C.

$$f_k = M_k \cdot \frac{f_{0k} M_{0k} (1 - y_d) + f_{1k} M_{1k} y_d}{M_{0k}(1 - y_d) + M_{1k} y_d} \tag{2}$$

$$f = M \cdot \frac{f_0 M_0 (1 - z_d) + f_1 M_1 z_d}{M_0(1 - z_d) + M_1 z_d} \tag{3}$$

3.4. Our Pipeline

Our pipeline is macroscopically the same as the pipeline of NICE-SLAM, but we add the procedure of assigning voxel features before optimization and classifying different points while passing through the decoder. We also add a different sampling method. Apart from uniformly sampling points along the ray, we also randomly sample more points near the surface.

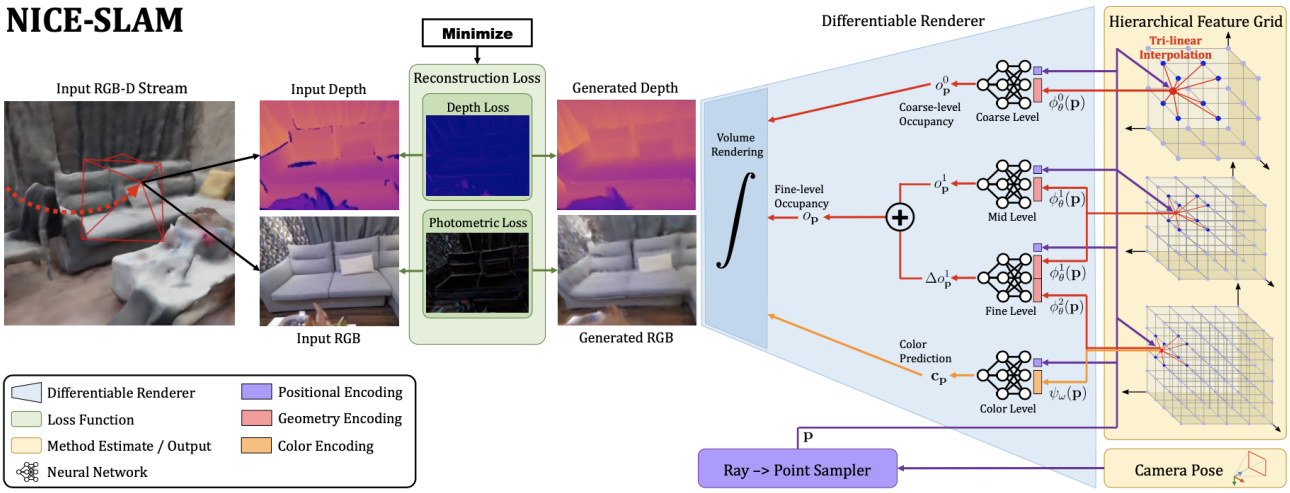

Figure 1. NICE-SLAM Pipeline [8]

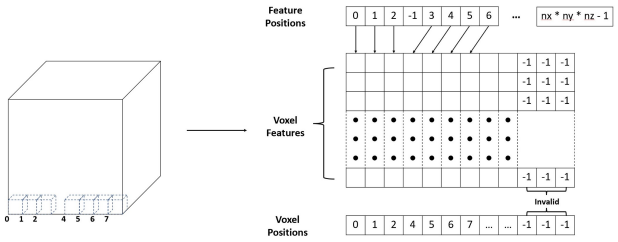

Figure 2. Voxel Hashing Sketch Map

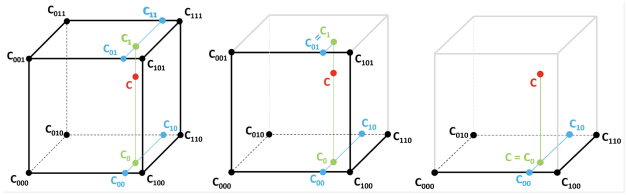

Figure 3. Dense and Sparse Trilinear Interpolation

After sampling points uniformly along the back-projected ray, we only assign features to the neighbor voxels of points near the surface. We do not store features of voxels far from the surface. We compare the distance between a sampled point and the camera with the input depth to decide whether a point is near the surface. We back-project each pixel using the intrinsic camera matrix, and get a back-projected ray for every pixel. Since we have the input depth value of every pixel of the camera in keyframes, this combined with back-projected ray and depth value we can roughly know where the surface of the scene is.

We use sparse trilinear interpolation (Section 3.3) for each selected point to get its corresponding latent representation and send it to decoder to get occupancy value and color.

For those points not interpolated, all their neighbors are not assigned. Instead of passing it through the decoder, we directly assign the occupancy value 0 (-10000 before sigmoid function) and color value 0 to it (details will be explained in Section 4.1).

4. Implementation Details

We have come across some implementation challenges. For example, how to deal with points with no assigned neighbor voxels When adaptively assigning and updating voxel features (Section 4.1). The efficiency of calculating the sparse trilinear interpolation is also a problem during implementation (Section 4.2).

4.1. Feature and Occupancy Assignment

We implement our sparse NICE-SLAM based on the original NICE-SLAM framework. We only need to change the mapping phase for this sparse system. We divide the render ray sample function in renderer into two different parts. If the sampled point is near the surface, we assign features to the 8 nearest neighbor voxels. If the point has no neighbors with assigned features (denoted as 'not assigned points'), we directly put the occupancy to 0. In detail, we set the decoder output to -10000 before passing it through the sigmoid function. Otherwise, we do sparse trilinear interpolation about the point and pass it through the decoder to get the occupancy.

Here, we do not assign all 0 features to the 'not assigned points', because the output of the decoder may not be 0. Also, after performing grid search test, we find that the 3D position of the point affect its occupancy badly, so we can not find an exact feature representation for a point with occupancy 0. Based on this finding, we divide the points into two groups and let the 'not assigned points' skip the decoder.

Another detail is why we still sample points far from the surface in this sparse algorithm. This is because the depth is not always well detected by the RGB-D camera. For some pixels, the depth may be 0 but the actual surface is actually far from the camera. Under the assumption that surface are mostly continuous, we still uniformly sample points along the ray. This way there would be points near the true surface with already assigned neighbor voxels. This point would then be interpolated and will have a relatively precise occupancy.

4.2. einops einsum

When implementing sparse trilinear interpolation, we use $einsum$ to deal with matrix calculation. $einsum$ is a function in the $einops$ package [4]. The notation for $einsum$ is elegant and clean which can help us quickly understand and implement high dimensional tensor calculations. For example:

$$C_{N\times M\times D} = A_{N\times M\times K\times D} B_{N\times M\times K} \tag{4}$$

Here, A, B and C are all high dimensional tensors. The string "nmkd, nmk $\rightarrow$ nmd" will help us to directly calculate each element in C:

$$C_{nmd} = \sum_{k} A_{nmkd} B_{nmk} \tag{5}$$

This notation would highly simplify the tensor calculation during interpolation.

5. Experiments and Results

We evaluate our sparse NICE-SLAM framework using the same datasets as used in NICE-SLAM paper [8]. Since our main contribution is on memory saving, we mainly compare the memory usage and some result evaluation metrics of our sparse algorithm with NICE-SLAM.

5.1. Experiment Setup

Datasets. We use the same dataset ScanNet [1] and Replica [5] as are used in NICE-SLAM paper. ScanNet has 500 keyframes. Replica contains highly photo-realistic 3D indoor scene reconstruction at both room and flat scale. It has 8 sub-scenes, each one with 2000 keyframes.

Metrics. Following NICE-SLAM, which uses the metrics *Accuracy, Completion and Completion Ratio* from paper [6] for scene geometry evaluation. Accuracy (cm) is the average distance between sampled points from the reconstructed mesh and the nearest ground-truth point. Completion (cm) is the average distance between sampled points from the ground-truth mesh and the nearest reconstructed. Completion Ratio (< 5 cm %) is the percentage of points in the reconstructed mesh with Completion under 5 cm.

Implementation Details. For the experiment, we use the AutoDL server: NVIDIA GeForce RTX 3090, 24GB of GPU memory; 5 core CPU (Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz), 64GB of memory. Because of the limitation of memory, we use a relatively large grid length with lower resolution. We choose $l_c = 2, l_m = 0.64, l_f = 0.32$.

5.2. Reconstruction Results

We can see in Figure 4 a reconstructed keyframe from the ScanNet Dataset. This is the 50th. keyframe in this dataset. We can see that our result cannot construct the floor very well. This is because the depth of the floor might not be precise and the points near the true depth have not been assigned yet at the 50th. keyframe. So the occupancies are 0 and appear as black in the generated RGB image. But if we increase the number of iterations, more points would be sampled and the black area would gradually disappear. This problem would be solved when more keyframes are processed and more points are assigned. Notice that for the points already assigned, the generated RGB image has a good result.

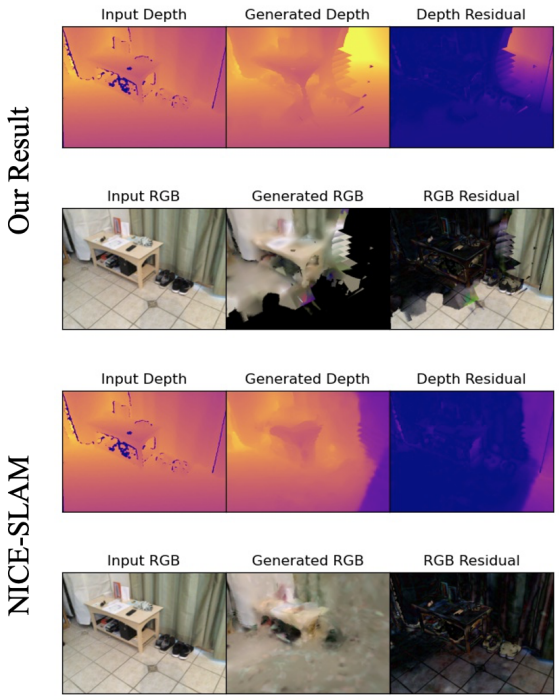

Figure 4. Visualization of the Reconstructed ScanNet Dataset

Figure 5 shows the 3D meshes of the reconstructed Replica Dataset. Because Replica is a simulated dataset, we have the ground truth mesh. We compare our sparse NICE-SLAM result with NICE-SLAM and ground truth. We can see that the surface produced in our framework is smoother than that of NICE-SLAM. More reconstruction metrics are shown in Section 5.3.

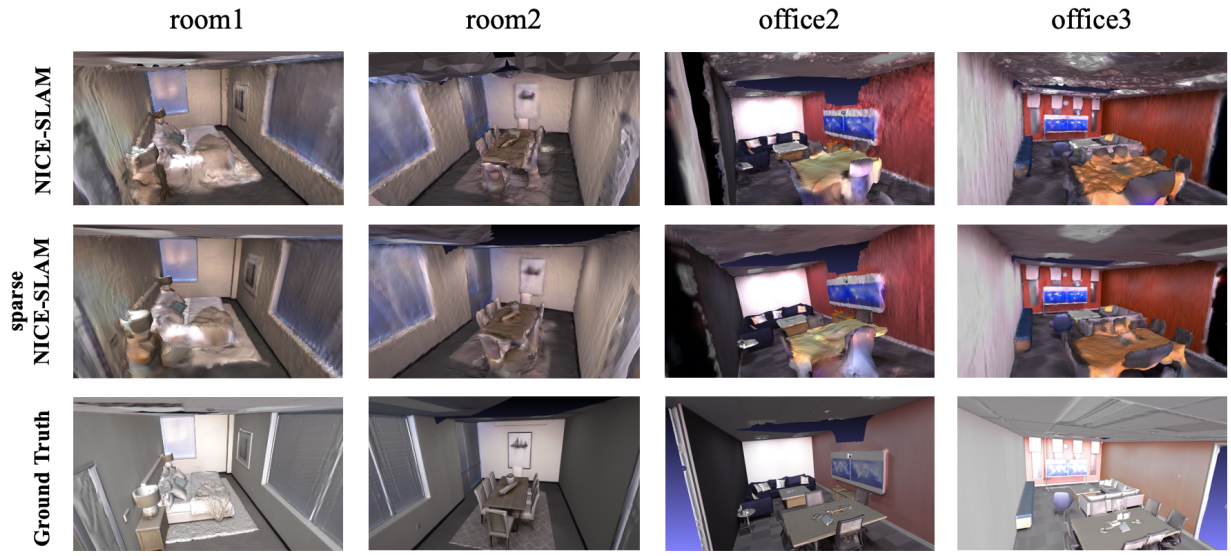

Figure 5. Meshes of the Reconstructed Replica Dataset

5.3. Memory and Loss Comparison

In Figure 6, we calculate the number of occupied voxels for both middle level and fine level with respect to the number of keyframes processed. We can find that for the original dense representation NICE-SLAM, the number of occupied voxels is a constant and very large. In our improved sparse NICE-SLAM system, the middle level memory is on average 5 times less than the dense situation, and fine level memory is on average 9 times less than the dense situation. Specifically, in office1 dataset, the middle level memory is 11 times less than the dense situation, and fine level memory is 22 times less than the dense situation. It means that we can use the same amount of memory to reconstruct space at least 11 times larger than the current dataset.

In Table 1, we compare the reconstruction metrics for NICE-SLAM and our sparse NICE-SLAM on dataset Replica. We have the average Accuracy of 3.15, Completion of 2.85 and Completion Ratio of 88.55. Two of the three metrics are better than that of NICE-SLAM. For the Accuracy, our framework outperforms NICE-SLAM in one-half of the datasets. For Completion, our framework performs better in 7 datasets. Lastly, for Completion Ratio, our framework performs better in 6 datasets.

6. Conclusion and Limitation

As we have shown, our developed sparse NICE-SLAM algorithm is capable of achieving a comparable scene reconstruction quality while using significantly less storage memory. However, limitations still exist in our algorithm and future work would be beneficial in improving our sparse NICE-SLAM algorithm. One major limitation is the running time of our algorithm, as it takes longer time to run compared with the original algorithm. One possible reason is that we use more mask operation on the related tensor to filter out unrelated voxels for rendering depth and color image. Thus, we need more memory access with stride which may take longer time. Moreover, because this pipeline is implemented in multi-process way, both mapper and tracker have an individual process. So we need to communicate with different process to synchronize the scene data. But since we also need to increase the size of voxel hashing map because of the increasing size of map during camera moving, the tensor itself will change address in memory and is hard to share across processes. To solve that, we allocate a fixed size of voxel hasing map with estimated size, which means it may still has some redundant amount of memory.

In this project, we present sparse NICE-SLAM, a sparse SLAM approach combining the advantage of hierarchical grid-based neural implicit representation and Voxel Hashing. We sample points and adaptively add neighboring voxel features into the optimization process. Experiments on Replica and ScanNet datasets demonstrate that, compared to the original NICE-SLAM, our algorithm significantly decreases memory usage as well as comparable reconstruction metrics. Our framework is scalable and can be further adapted to outdoor scene reconstruction.

7. Division of Work

Ganlin Zhang. Literature review. Implement Voxel Hashing class, integrate it into NICE-SLAM pipeline.

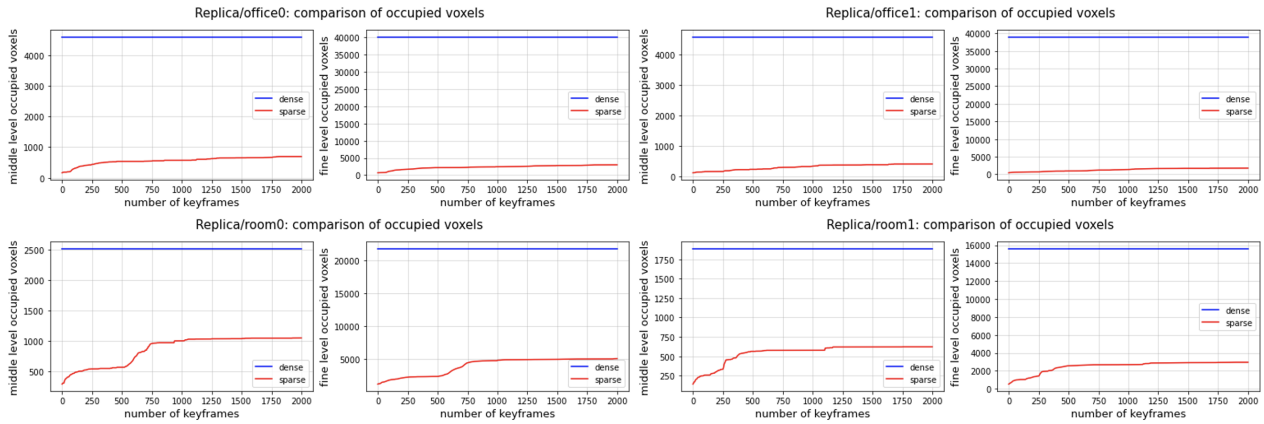

Figure 6. Memory Metrics on Reconstruction of the Replica Dataset

		room-0	room-1	room-2	office-0	office-1	office-2	office-3	office-4	Avg.
NICE-SLAM	**Acc.[cm]↓**	2.95	2.35	2.47	2.29	3.32	3.26	3.53	2.67	2.86
	Comp.[cm]↓	3.39	2.54	2.84	2.17	2.49	3.30	4.05	4.42	3.15
	Comp.Ratio[<5cm %]↑	87.29	90.95	88.50	93.21	90.97	84.85	80.46	84.52	87.59
sparse NICE-SLAM	**Acc.[cm]↓**	2.99	4.50	2.25	3.34	3.11	3.08	3.05	2.84	3.15
	Comp.[cm]↓	3.03	2.44	2.45	2.94	2.47	2.96	3.30	3.24	2.85
	Comp.Ratio[<5cm %]↑	89.25	91.20	90.31	88.98	87.65	87.72	85.72	87.51	88.55

Table 1. Reconstruction Metrics for the Replica Dataset

Write the overall structure of the codes. Implement sampling the reprojected points which near scene surface to assign map voxels.

Deheng Zhang. Literature review. Implement Voxel Hashing class, integrate it into NICE-SLAM pipeline. Design, implement and test the sparse trilinear interpolation function. Adapt the sparse representation to the tracker. Test the used voxel size.

Feichi Lu. Literature review. Design the Voxel Hashing class. Test occupancy oscillation with features and positions on decoder. Run experiments for sparse NICE-SLAM on Replica Dataset and visualize results. Write poster and final report.

Anqi Li. Literature review. Design the Voxel Hashing class. Help with testing and experiment. Proposal and Mid-term presentations. Write poster and final report, proof reading. Documentation and keeping track of project progress, requirement, and deadlines.

References

[1] Angela Dai, Angel X Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Nießner. Scannet: Richly-annotated 3d reconstructions of indoor scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5828–5839, 2017.

[2] Jiahui Huang, Shi-Sheng Huang, Haoxuan Song, and Shi-Min Hu. Di-fusion: Online implicit 3d reconstruction with deep priors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8932–8941, 2021.

[3] Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Marc Stamminger. Real-time 3d reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (ToG)*, 32(6):1–11, 2013.

[4] Alex Rogozhnikov. Einops: Clear and reliable tensor manipulations with einstein-like notation. In *International Conference on Learning Representations*, 2022.

[5] Julian Straub, Thomas Whelan, Lingni Ma, Yufan Chen, Erik Wijmans, Simon Green, Jakob J Engel, Raul Mur-Artal, Carl Ren, Shobhit Verma, et al. The replica dataset: A digital replica of indoor spaces. *arXiv preprint arXiv:1906.05797*, 2019.

[6] Edgar Sucar, Shikun Liu, Joseph Ortiz, and Andrew J. Davison. imap: Implicit mapping and positioning in real-time, 2021.

[7] Wikipedia contributors. Trilinear interpolation — Wikipedia, the free encyclopedia, 2021. [Online; accessed 11-June-2022].

[8] Zihan Zhu, Songyou Peng, Viktor Larsson, Weiwei Xu, Hujun Bao, Zhaopeng Cui, Martin R Oswald, and Marc Pollefeys. Nice-slam: Neural implicit scalable encoding for slam. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12786–12796, 2022.