Homework #4: Mesh processing; 3D vision; primitive fitting; programming with deep learning framework [100 points]
Due Date: Friday, 19 March 2021 – No late days are available for this assignment

Theory and implementation

This is a mixed theory/implementation problem set related to point cloud segmentation [1, 3] and building on the tools you have developed in homework 3. Problem 1 provides some practice on primitive fitting over meshes, using geometry processing. Problem 2 gives you a chance to segment a point cloud into primitives and estimate parameters in an alternate way, using deep learning

You may hand in a joint write-up for these implementation problems. Recall that the maximum group size is three. You can download all helper code for this problem from the Handouts class web page.

Problem 1. [50 points]

In this problem, we segment a triangle mesh into surface patches corresponding to different geometric primitives, using the theory derived in problem 4 of homework 3. Examples (CA's implementation) are shown in figure 1.

Note that in the last assignment we are able to analyze the slippable motions for a point cloud belonging to a single primitive. In this assignment, we will implement a region merging method to automatically segment the primitive surfaces and estimate their parameters [1], according to their slippability. We first segment the original mesh into small regions assuming that each belongs to a single primitive. We can assign a similarity score between two neighboring regions according to the error when merging them as a single primitive. We greedily merge regions in decreasing order of the similarity score, until the score of the most similar neighboring regions is below a certain threshold.

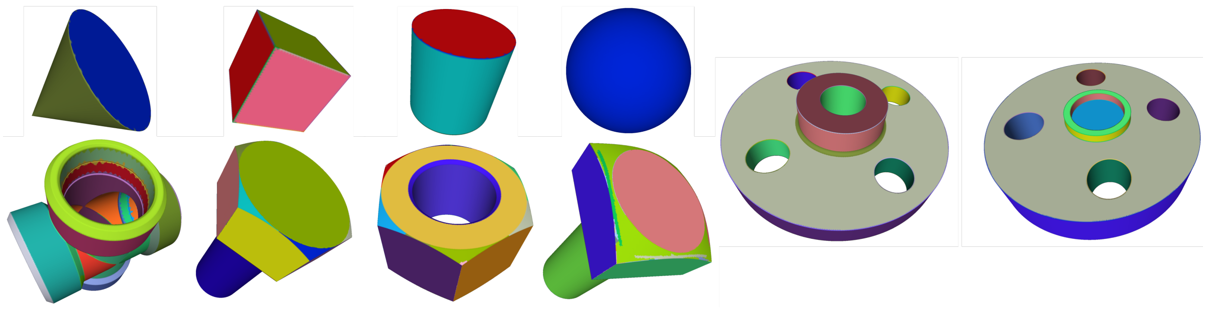
Figure 1: Primitive segmentation results from CA's implementation.

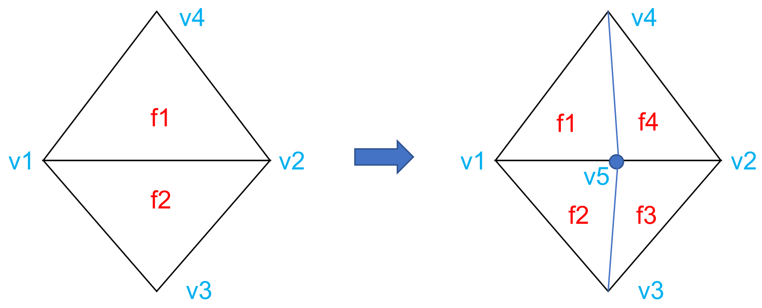

Figure 2: Subdivision Operator.

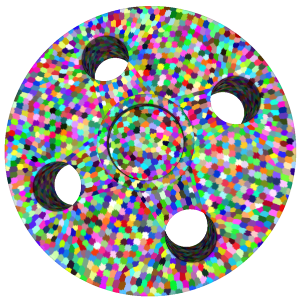

Figure 3: Visualization of the small region extraction.

1(a). [Data structure, 5 points] Our input mesh is guaranteed to be a watertight triangle manifold. The mesh is stored as an n-by-3 vertex matrix V and an m-by-3 face matrix F (indexing starts from zero). We use the Eigen library to store and manipulate the matrices. Since the algorithm is strongly related to surface normals, we need to compute the vertex normals and store them as an n-by-3 normal matrix N.

There are many operations related to searching for the vertex neighbors, where a *half-edge data structure* helps[1]. Implement *ComputeVertexNormals* and *BuildHalfEdges* in `mesh.cpp` to derive the normals and the data structure containing an array of vertices, edges and faces. You are free to add utility functions to help with your mesh processing in the following steps. The starter code serves as a guide in your implementation; feel free to add and modify it.

1(b). [Subdivision, 5 points] The input mesh is usually coarse and without uniformly distributed vertices. This prevents us from extracting small regions with enough vertices. We therefore subdivide the mesh so that the length of any edge is smaller than a threshold (set as 1e-2 in CA's implementation). One way to subdivide is shown in figure 2: we add the midpoint for an edge and split the two adjacent triangles by the medians corresponding to the midpoint. We keep subdividing the longest edge in the mesh until all edges are short enough. With heap (std::priority_queue in CPP), this algorithm can be performed in $O(e \log e)$ where e is the number of edges.

1(c). [Region Extraction, 5 points] The CA proposes to extract small regions with a radius R ranging from 0.015 to 0.03. A unit operation is to find all vertices whose shortest path to the source vertex is smaller than the radius threshold, and this set of vertices roughly form the region specified by the radius. This can be implemented with Dijkstra's shortest path algorithm:

We have a candidate vertices pool where paths are found (but not necessarily shortest). Initially the pool contains only the source vertex whose distance is zero. At every step, we pick the vertex from the pool with the shortest distance, remove it from the pool, put all its

[1]The half-edge data structure is a simpler version of the quad edge discussed in class – see, for example, `https://www3.cs.stonybrook.edu/~gu/lectures/lecture_8_halfedge_data_structure.pdf`.

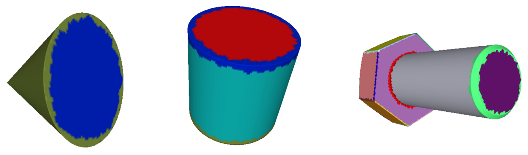

Figure 4: Example result after region merging. Notice that visual artifacts are apparent

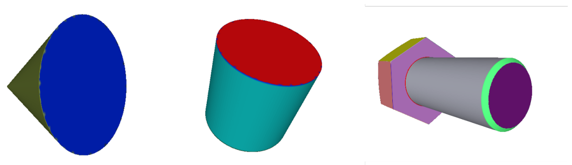

Figure 5: Corresponding example result after region refinement.

neighboring vertices into the pool and update their distance. The CA manages the pool with a priority queue to help with the efficient query of the vertices with the shortest distance. More details can be found in any standard algorithms text, such as *Introduction to Algorithms, 3rd Edition (The MIT Press), by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein*.

With this unit operation, we can use a flood-fill algorithm to extract source vertices with small regions that cover the whole mesh. We loop over all vertices in the sequential order. If we find a vertex as not visited, we put it into a source vertices list, extract the region whose radius is 0.03, and mark all vertices in the region as visited.

Finally, we apply the Dijkstra algorithm with multiple sources from our source vertices list, so that we know for each vertex in the mesh which is its nearest source vertex. Each source vertex and all vertices closest to it form a small neighborhood. Visualization of the CA's implementation of the small regions is shown in Figure 3.

1(d). [Region Merging, 15 points] For each region, you should compute a slippage signature including the number of points n, the matrix $\mathbf{C}$ you derived from problem 4(c) of Homework 3 (consider dividing by n for numerical stability), and the sub-spaces of acceptable motion vectors. In the original paper, the points are normalized before computing $\mathbf{C}$, which is ignored in the CA's implementation. When merging two regions, $\mathbf{A}$ of the combined region can be directly computed from $\mathbf{C}$ of the two separated regions in $O(1)$ (constant time).

The region merging algorithm is shown as lines 6-24 of algorithm 1 in [1]. Basically, we merge pairs of adjacent regions in the decreasing order of the number of dimensions for the slippable motions. Within the same number of dimensions, we merge regions with decreasing order of similarity score (eq. 9 in [1]). In the CA's implementation, the only difference is line 15: The CA rejects the pair if two regions have different dimensions or the potential merged region has reduced dimensions of slippable motion. You are encouraged to experiment with different choices for similarity score or threshold, to get your best result. Figure 4 shows example results after region merging from the CA's implementation.

1(e). [Region Refinement, 10 points] The CA maintains a priority queue of boundary edges whose two vertices belong to different regions after the merging step. The score of the edge is the smaller fitting error of changing the region label for the two vertices. We keep selecting the best edge and changing the region label if the fitting error is smaller than a threshold. Once a vertex label is changed, edges connecting to the vertices need to be updated to the queue, since they may become the boundary edges. Figure 5 shows example results after

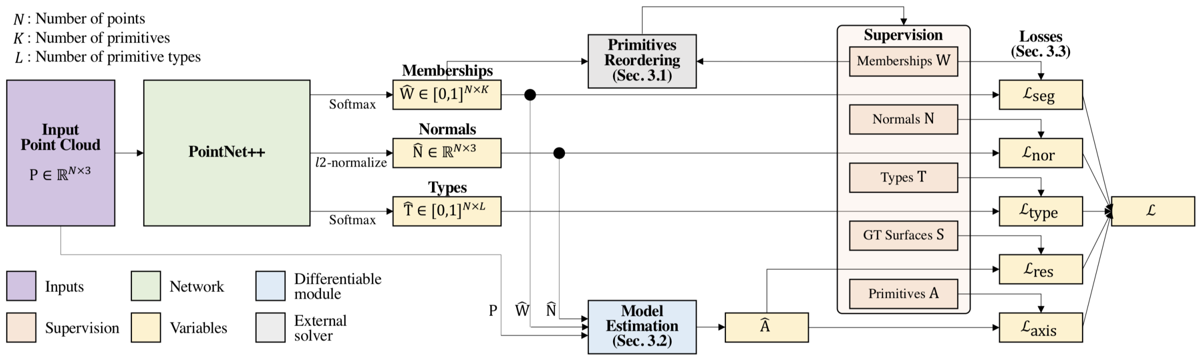

Figure 6: The architecture for the primitive fitting network.

region refinement from the CA's implementation.

1(f). [Quality and Efficiency, 10 points] By running `run.sh`, you should be able to produce the results for the given data. The score will be given based on the quality of the segmentation and the efficiency of the algorithm. You are encouraged to try your best in implementing the best results, which would require algorithm correctness, parameter tuning and efficient implementations. Figure 1 shows the output quality from the CA's implementation, which can process all given meshes in around 1 minute.

Problem 2. Implement primitive fitting via a supervised deep network [50 points]

In this problem, we aim at jointly segmenting point clouds into primitive instances and estimating the primitive parameters, following the theory of problem 3 in homework 3.

The key idea is to train a neural network that associates each point with a primitive type and instance ID, and estimates the parameters for the primitives. Figure 6 shows the architecture for the primitive fitting network. The input to the network is a point cloud with n points. Using *PointNet++* [4] as the backbone network, we extract a high-level feature for each point. This feature is translated by three different fully connected layers respectively into primitive instance ID, point normal, and primitive type.

Instance ID Assuming the maximum number of primitive instances one object has is K, the instance information can be represented as a K-dimensional vector $\mathbf{w}$ where $\mathbf{w}_i$ represents the probability that the point belongs to the i-th instance. Therefore, a softmax layer is followed by the first fully connected layer to extract such a vector, and instance information for all points form a $n \times K$ membership matrix $\hat{W}$, as shown in figure 6. This membership matrix can be viewed as a classification for each point into a primitve instance.

Point Normal The point normal is simply a 3-dimensional unit vector for each point. Therefore, a L2-normalization is followed by the second fully connected layer to extract the $n \times 3$ normal matrix.

Primitive Type We estimate the primitive type for each point. Similar to the representation of instance ID, we use softmax to extract a $n \times L$ matrix $\hat{\mathbf{T}}$ where $L = 4$ is the number of primitive types we study.

2(a). [Loss functions, 10 points] We have a dataset with ground truth instance, primitive type and parameter, and use losses to supervise the network to make reasonable estimations. Specifically, for each point cloud the ground truth data contain the membership matrix $\mathbf{W}$, $\mathbf{N}$, primitive types $\mathbf{T}$, and for each instance with ID $i \le K$ the corresponding parameter $\mathbf{A}_i$. [3] proposes several losses and use the sum of them for training.

Instance Segmentation Loss As shown in figure 6, our network predicts a membership matrix $\hat{\mathbf{W}}_{ij}$ indicating possibility each point i is associated with primitive instance j. This segmentation should be consistent with the ground truth $\mathbf{W}$. One challenge is that the order of predicted instances can be different from the ground truth, and the loss should not penalize the reordering.

For each instance, the segmentation mask is indicated by the columns of the membership matrix as a n-dimensional vector. For each pair of segmented instances in the prediction and ground truth as $\hat{\mathbf{w}}$ and $\mathbf{w}$, we can compute the intersection of union (IoU) score as

$$IoU(\hat{\mathbf{w}}, \mathbf{w}) = \frac{\hat{\mathbf{w}}^T \mathbf{w}}{||\hat{\mathbf{w}}||_1 + ||\mathbf{w}||_1 - \hat{\mathbf{w}}^T \mathbf{w}}\,. \quad (1)$$

We aim to find the best one-to-one matching between $\hat{\mathbf{w}}$ and $\mathbf{w}$ for columns of membership matrix so that the sum of IoU scores is maximized, which can be solved by a classical hungarian matching [2] algorithm.

Assuming the best ordering pairs each instance k in the prediction with e_k in the ground truth, the instance segmentation loss can be defined as

$$\mathscr{L}_{seg} = \frac{1}{K} \sum_{k=1}^{K} (1 - IoU(\hat{\mathbf{W}}_{:,k}, \mathbf{W}_{:,e_k}))\,. \quad (2)$$

Point Normal Loss The ground truth normal $\mathbf{N}$ is used to supervise the predicted normal $\hat{\mathbf{N}}$ in figure 6, where the loss is implemented as

$$\mathscr{L}_{norm} = \frac{1}{N} \sum_{i=1}^{N} (1 - |\hat{\mathbf{N}}_i^T \mathbf{N}_i|)\,. \quad (3)$$

Primitive Type Loss We penalize the errors of primitive type estimation $\hat{\mathbf{T}}$ for each point i with the cross entropy H according to the ground truth $\mathbf{T}$:

$$\mathscr{L}_{type} = \sum_{i=1}^{N} H(\hat{\mathbf{T}}_i, \mathbf{T}_i)\,. \quad (4)$$

Note that for points that are not associated with any primitive in the ground truth, we simply ignore them.

Primitive Fitting Loss The primitive loss is computed as

$$\mathscr{L}_{res} = \frac{1}{K}\sum_{k=1}^{K} \mathscr{E}(\mathbf{A}_k; \mathbf{P}_k)\,. \tag{5}$$

$\mathscr{E}(\mathbf{A}_k; \mathbf{P}_k)$ is the mean squared distance of the k-th primitive's ground truth point cloud $\mathbf{P}_k$ under the estimated parameter $\mathbf{A}_k$.

Note that the primitive type for each instance is directly decided from ground truth during training time. In test time the primitive type is inferred by

$$\hat{\mathbf{t}}_k = \arg\max_{l} \sum_{i=1}^{N} \hat{\mathbf{T}}_{i,l}\hat{\mathbf{W}}_{i,k}\,. \tag{6}$$

Axis Angle Loss When estimating the primitive axes, the SVD decomposition becomes numerically unstable when the number of points is small. Therefore, we additionally add regularization terms for axes estimation as

$$\mathscr{L}_{axis} = \frac{1}{K}\sum_{k=1}^{K} (1 - |\hat{\mathbf{a}}^T \mathbf{a}|)\,. \tag{7}$$

where $\mathbf{a}$ is the normal of the plane or axis of the cylinder or the cone. Spheres are ignored in this loss function.

Implement the following losses in `src/evaluation.py` in function `evaluate` (The training loss should be sum of them). We provide `test_loss.py` for basic verification.

- $\mathscr{L}_{seg}$, instance segmentation loss.
- $\mathscr{L}_{norm}$, point normal loss.
- $\mathscr{L}_{type}$, primitive type loss.
- $\mathscr{L}_{res}$, primitive fitting loss.
- $\mathscr{L}_{axis}$, axis angle loss.

2(b). [Network implementation, 30 points] Implement the data loader and the network following the *README* instructions. Train the network to get decent network predictions. For the data loader, run `test_dataloader.py` for verification.

2(c). [Colorize the result for the segmented point cloud, 10 points] Once you obtain the segmented point cloud with primitive information, visualize them as colored pointclouds for the different primitive types and instances

What to hand in

You are required to submit your "assignment4.zip" archive for this assignment by email to the CA with the same file distribution as you downloaded. In it, please provide a "./report.pdf" including names of group members, the discussion of your code/method for both problems, and visualization of the results.

No late days are available for this assignment. All class assignments are due by Friday, March 19, 2021, 11:59 pm PDT.

On Friday morning, March 19, 2021, we will schedule brief demos by each team on Homework 4.

References

[1] Natasha Gelfand and Leonidas J Guibas. Shape segmentation using local slippage analysis. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pages 214–223. ACM, 2004.

[2] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[3] Lingxiao Li, Minhyuk Sung, Anastasia Dubrovina, Li Yi, and Leonidas Guibas. Supervised fitting of geometric primitives to 3D point clouds. In *Proceedings of the IEEE Conferencec on Computer Vision and Image Processing*. 2019.

[4] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *Advances in neural information processing systems*, pages 5099–5108, 2017.