End-to-End Rotation Averaging with Multi-Source Propagation
Supplementary Material

Luwei Yang\textsuperscript{1}  Heng Li\textsuperscript{1}  Jamal Ahmed Rahim\textsuperscript{1}  Zhaopeng Cui\textsuperscript{2}  Ping Tan\textsuperscript{1}\ast
\textsuperscript{1} Simon Fraser University  \textsuperscript{2} State Key Lab of CAD & CG, Zhejiang University
\{luweiy, lihengl, jrahim, pingtan\}@sfu.ca, zhpcui@zju.edu.cn

To make our submission self-contained, the supplementary material provides additional details about: 1) The Network and Modules; 2) Training; 3) Test & Iterative Edge Weight Refinement, and lastly 4) additional experiments on execution time and outlier impact. The code demonstration and dataset (YFCC100) are available at https://github.com/sfu-gruvi-3dv/msp_rot_avg.

1. Network Details

In this section, we provide additional details of the network modules, as we mentioned in the main paper.

\textbf{Feature Extraction Module}: The Feature Extraction Module extracts useful information for the graph nodes and edges from images with matched correspondences. Recall that it employs cluster-based aggregation for obtaining fixed-size node features and edge features regardless of arbitrary input dimensions and the number of feature matches.

For constructing the node feature at a node $i$, we first extract feature maps $F_4^i$ (512 channels) from VGG16\textsuperscript{[14]}’s layer group 4. The VLAD pooling operation applies $k$-means to cluster all the pixels from the feature maps $F_4^i$ into $P = 64$ groups, resulting in a pooled VLAD feature $g^i$. We then finally reduce the channel size of $g^i$ with a $1 \times 1$ convolution, producing the node feature, $\hat{g}^i$, with a size of $32 \times 64$. For constructing the edge feature for an edge $i,j$, we first reduce the channels of the feature maps $F_4^i$ to 64 channels with another $1 \times 1$ convolution, resulting in $\hat{F}_4^i$. Given that the edge connects node $i$ and $j$ with $n$ pairs of matches, we gather the features from each pair of correspondence:

$$\{f_{ij}\}_n = \{[x_a^i, x_b^j, f_a^i, f_b^j]\}_n$$

where $f_a^i = \hat{F}_4^i(x_a^i)$ and $f_b^j = \hat{F}_4^j(x_b^j)$ are the features of points $a, b$ in the feature map $\hat{F}_4^i, \hat{F}_4^j$. The $\hat{x}_a^i, \hat{x}_b^j$ are 2D positions of keypoints $a, b$ on frames $i, j$, and normalized to [-1, 1]; Next, a two-layer MLPs $\phi(\cdot)$ with $\{128, 64\}$ hidden units is used to generate a vector encoding the features, and yield a $n \times 64$ feature: $\{\hat{f}_{ij}\}_n = \{\phi(f_{ij})\}_n$; Lastly, to remove any dependency on the number of matches, we apply $k$-means clustering to the positions of keypoint $\{\hat{x}_i\}_n$, resulting in $K = 10$ different groups. Then, for all features that belongs to a cluster group $m$, $\{\hat{f}_{ij}\}_n^m$, we use Maxpool to aggregate the feature set to a single vector feature and get

\begin{verbatim}
Figure 1. Structure of Appearance-Geometry Fusion Network. The size of input and output feature is listed in each Edge Convolution layer.
\end{verbatim}
our edge features $e_{ij} = \{\text{maxpool}(\{\hat{f}_{ij}\}_m^m) | m < K\}$. The aggregate function Maxpool is chosen as it is symmetric for all items regardless of ordering dependency [11]. This results in edge features with a fixed dimension of $10 \times 64$.

AGF Network Structure: The Appearance-Geometry Network takes the node and edge feature, along with the measured relative geometry as input, and predicts a weight for each edge, which indicates whether the edge is reliable or not; The Figure 1 shows the structure of AGF Network. The AGF consists of 6 Edge Convolution layers[5], each with 128 hidden units. Note that from layer #3 to #6, the input node and edge features are concatenated output node and edge features from the previous two Edge Convolution layers, respectively. For example, the output node features from Edge Conv.#1 and Edge Conv.#2 are concatenated as the input node features for Edge Conv.#3. Similarly, the input edge features for Edge Conv.#3 is the concatenation of the edge features Edge Conv.#1 and Edge Conv.#2. Lastly, a MLP layer followed by a sigmoid activation is appended to the output edge features from Edge Conv.#6, and generates the normalized weight on each edge \( w_{ij} (i,j) \in E, 0 < w_{ij} < 1 \).

Self-loops in MSP: In MSP, we add a virtual self-loop for each node in the view-graph with a preset weight 1.0. This will fix the rotation at the node when it reaches a high confidence score and will not propagate much information from neighbors in the subsequent iterations.

FineNet Structure: With the initialized orientation as the node input and relative rotation as the edge input, the FineNet [10] is used to produce the refined orientation for all nodes in a view-graph. FineNet consists of 4 layers of Edge Convolution. The input node features are the initial camera orientations $q_i$, and edge features are the residual of the estimated and measured relative rotations: $f_{ij} = q_i^{-1}\hat{q}_{ij}q_i$. With the input node and edge features, the FineNet produces the residual orientation $\hat{q}_i$ over the initial orientation $q_i$ for each node $i \in V$. Later, its final refined orientation is obtained by $q_i' = \hat{q}_i + q_i$, and further normalized to unit with $\frac{q_i'}{|q_i'|}$.

2. Training Details

In this section, we introduce more details of training data generation and training process.

Constructing View-Graph for YFCC100: We use the YFCC100’s reconstruction as the ground-truth camera orientation for training. To build the missing view-graph for every YFCC100 scene, the COLMAP [13] is used to extract 2D key-point features in every image at beginning, and the vocabulary tree is built to match each image against its visual 20 nearest neighbours. Then, any two nodes are connected by an edge if the five-points algorithm [8] yields a valid relative orientation. Lastly, for all YFCC100 and 1DSfM scenes, the view-graph is transformed from directed to un-directed by adding reverse edges with the inverse relative orientation \( \{q_{ij}^{-1} (i,j) \in E\} \).

Sampling Training Sub-graphs: Recall that we sample multiple sub-graphs during training due to the limited GPU memory. Given the full view-graph $G = \{V, E\}$, we aim to find a sub-graph $G_s = \{V_s, E_s\}$ with maximum $N_s$ nodes. Firstly, a node $v_r \in V$ is randomly chosen as the root node, then we start Breadth-first Search (BFS) from node $v_r$, and add all visited nodes into sampled node set $\{V_s\}$ until $|V_s| = N_s$. Later, the sampled edge set $E_s$ is formed by adding all edges $(u, v)$ that connects any two nodes $u \in V_s$ and $v \in V_s$ in sampled node set $\{V_s\}$. We randomly sample 10-200 sub-graphs from each view-graph of YFCC100 scene, depending on its total number of nodes. Typically, this generates 3,000 – 4,000 sub-graphs for training. As for the 1DSfM dataset, we randomly sample 250-300 sub-graphs from each scene, and form a training set with a total of 3000-3500 sub-graphs.

Implementation and Parameter Initialization: Our model is implemented with the PyTorch [9] framework. We use the pytorch-geometry [4]’s Edge Convolution implementation for constructing the Appearance-Fusion Network while the Multi-Source Propagator is implemented using the DGL library [15].

For training on the YFCC100 dataset, the parameters of VGG16 [14] and VLAD layer in Feature Extractor are copied from the trained NetVLAD[1] model, and the FineNet is initialized with the model trained by NeuRoRA’s synthetic data [10]. Edge Convolution layers in AGF and other MLPs are randomly filled by PyTorch’s default initialization generator. Later, the model trained on YFCC100 is used to initialize the parameters when fine-tuning the 1DSfM dataset.

Training: Recall that due to the memory limitation, our model is trained in three stages, with each stage containing 30, 80 or 180 nodes in sampled sub-graphs. We first use sub-graphs with 30 nodes to train our full model till it converges. Then, we fix the parameters of Feature Extractor for memory efficiency and train subsequent modules with 80 nodes. When training on sub-graphs with 180 nodes, in order to get better generalization, a pair of FineNet that shared the parameters are chained sequentially such that the output of the first FineNet is fed into the second FineNet. Thus, the total loss for the sub-graph with 180 nodes is:

$$L_{180} = L_e + L_{\text{init}} + L_{\text{opt}}^1 + \omega_{\text{opt}}L_{\text{opt}}^2,$$

where the $L_{\text{opt}}^1$ and $L_{\text{opt}}^2$ are the same loss function (Equation 13 in main paper) applied to the output of the first and the second FineNet, while weight $\omega_{\text{opt}} = 2.0$ balances the two term.
Adam optimizer [7] is used to train our pipeline with different learning rates in various modules, as shown in Table 1. It takes 36 hours (for 15 epochs) to train the first stage with 30 nodes in each sub-graph. For the following two stages, the training can be accelerated by caching the node and edge features to disk during the first epoch. We then train the model with another 39 epochs, with the two stages costing 24 hours and 36 hours, respectively.

<table>
<thead>
<tr>
<th>Module</th>
<th>$\lambda$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node and Edge Feature Extraction</td>
<td>$1.0 \times 10^{-4}$</td>
</tr>
<tr>
<td>AGF Network</td>
<td>$1.0 \times 10^{-4}$</td>
</tr>
<tr>
<td>FineNet</td>
<td>$1.0 \times 10^{-4}$</td>
</tr>
</tbody>
</table>

Table 1. The learning rates used in training various modules.

3. Test Details
In this section, we provide additional testing details. The Fig. 2 shows the overview of the test procedure, including 1) sampling multiple sub-graphs from a full view-graph that covers all edges; 2) generating edge weights for each sub-graph using AGF and later aggregate into full view-graph; 3) iterative edge weight refinement with MSP and FineNet in testing time. The following sections will detail the test pipeline.

3.1. Sub-graph Sampling During Testing
Recall that limited GPU memory prevents us from loading neither all images and correspondences nor all node and edge features at once. Therefore, we also sample sub-graphs from a view-graph to circumvent the limitations. However, compared to the training, the sampled sub-graphs set must cover all edges in the graph $G$ during testing. Therefore, we aim to sample a sub-graph set $S_G = \{S_1, \ldots, S_k\}$ from the original view-graph $G = \{V, E\}$, where each sub-graph has $N_s = 80$ nodes and all edges in the original graph $G$ are covered by any sub-graph in $S_G$.

The sub-graph set is constructed iteratively, we build a residual graph $G^R = \{V^R, E^R\}$ by duplicating $G$ at beginning; Next, we sample a sub-graph $S_k = \{V_k, E_k\}$ from $G^R$ using BFS, and then delete the $S_k$ from residual graph $G^R$. Specifically, we delete edges $e \in E^R$ that were previously sampled in $S_k$, and a node $v \in V^R$ is removed if no adjacent edge exists. Later, the remaining residual graph $G^R$ is used for sampling new sub-graphs in the following iterations. The sampling stops when $|E^R| = 0$, where all edges in $E$ are covered in the sub-graph set $S_G$. Note that the number of nodes $|V_k|$ of the sampled sub-graph $S_k$ might be less than $N_s$ after several iterations. Therefore, another function ExpandSubGraph is called to apply BFS on the original $G$ to extend $V_k$ until $|V_k| = N_s$. The steps are summarized in Algorithm 1.

Algorithm 1: Sub-graph Sampling During Testing

```plaintext
Input : $G = \{V, E\}$
Output: Sub-graph sets $S_G = \{S_1, \ldots, S_k\}, S_i = \{V_i, E_i\}$
Initialization:
1. $S_G \leftarrow \emptyset, E^R \leftarrow E, V^R \leftarrow V$  // copy original view-graph $G$
2. $k \leftarrow 1$

while ($|E^R| \neq 0$) do
3. $v_r \leftarrow \text{random}(V^R)$  // select a random node to BFS
4. $S_k \leftarrow \text{BFS}(V^R, E^R, v_r)$  // find sub-graph nodes
5. \quad if $|V_k| < N_s$ then
6. \quad \quad $S_k \leftarrow \text{ExpandSubGraph}(V, E, S_k)$
7. \quad end
8. $S_G \leftarrow S_G \cup \{S_k\}$
9. $E^R \leftarrow E^R - E_k$  // update residual graph $G^R$
10. $V^R \leftarrow \{u, v\mid e = (u, v), e \in E^R\}$
11. $k \leftarrow k + 1$
end
```

3.2. Building Complete Graph from Sub-graphs
Given a sampled sub-graphs set $S_G$ as input, we first construct the complete view-graph with weights on all edges. Specifically, this is done by first feeding the sub-graphs into the Feature Extractor and AGF modules, and caching the output edge weights. Note that the input node and edge features can be pre-computed and cached onto the disk at the beginning. Computation of relative poses and construction of the view-graph can similarly be done before the feed-forward for efficiency. Next, a complete weighted view-graph is aggregated where the edge’s weight is computed by the average of its weights in all sub-graphs containing it, resulting in $w^{(9)}$. Later, the full view-graph with
initial edge-weights is then fed together into the MSP followed by Finenet, and finally refined using Iterative Edge Weight Refinement.

3.3. Edge Weight Refinement

Recall that the edge weight \( w^0 \) might be imprecise. With the fully-differentiable MSP and FineNet modules, we can refine the edge weights by minimizing the discrepancy error towards relative orientation measurement. In practice, to speed up the overall refining process, we first obtain an approximate initial solution without involving the FineNet and minimize the following energy term w.r.t initial candidates \((M = 15)\) produced by MSP

\[
E' = \frac{1}{|M||E|} \sum_{m \in M} \sum_{(i,j) \in E} w^0_{ij} \rho(d_q(q_{ij}, q^m_{ij})).
\]  

(2)

Here, the \( d_q(q_a, q_b) = \min(||q_a - q_b||, ||q_a + q_b||) \) defines the distance of two quaternions and \( \rho \) denotes smooth \( L_1 \) costs:

\[
\rho(x) = \begin{cases} 
|x| - 0.5\alpha & \text{if } |x| > \alpha \\
0.5\alpha^2/x & \text{if } |x| \leq \alpha,
\end{cases}
\]

(3)

where \( \alpha = 5 \times 10^{-2} \). The rotation \( q^m_{ij} \) is computed using absolute orientations generated by MSP. We use Adam optimizer and set the learning rate \( lr = 1.0 \) to minimize the \( E' \) with 20 iterations, resulting in the approximate refined weight \( w' \). Note that the weight \( w \) is normalized at any iteration by a sigmoid function to clamp the range from 0 to 1 before feeding into the MSP.

Later, the weight \( w' \) is further refined by fixing the parameters of FineNet module, and use the same Adam optimizer (with \( lr = 1.0 \)) to minimize the following energy until convergence:

\[
E = \frac{1}{|M||E|} \sum_{m \in M} \sum_{(i,j) \in E} w'_{ij} \rho(d_q(q_{ij}, q^m_{ij})) + \frac{1}{|E|} \sum_{(i,j) \in E} w'_{ij} \rho(d_q(q_{ij}, q^f_{ij})).
\]

(4)

Similar to [6, 10], given the outputs from MSP, we empirically select the initial candidate that has the highest sum of adjacent weights of the reference node, and then feed it into FineNet to get refined absolute orientations \( \{q^f_i, i \in V\} \). Later, the relative rotation \( q^f_{ij} \) is computed using refined absolute orientations.

4. Additional Experiments

Execution Time: Table 2 reports the execution times of our method using the 1DSfM dataset. It includes the time costs of both, AGF and Iterative Refinement steps. Note that the features for nodes and edges can be extracted and pre-computed when computing the relative poses between any two images at initialization. The running time of NeuRoRA[10] and Chatterjee[2] (IRLS) are also shown for comparison. NeuRoRA is regression-based and is the fastest since it produces results with a single forward pass, while, as discussed in [12], pose regression is generally less accurate. Our method is slower because of the iterative edge re-weighting, which produces more accurate results. Compared to IRLS[2], our method runs faster in larger scenes (e.g. Piccadilly, 2152 nodes) while slower in smaller scenes, as large scenes could benefit from massive GPU parallelization. Our overall running time is acceptable for the current SFM pipeline, e.g. Global SFM [3] takes 578s for Alamo and 1480s for Piccadilly.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Alamo</td>
<td>16s</td>
<td>65s</td>
<td>81s</td>
<td>20.5s</td>
<td>2.2s</td>
</tr>
<tr>
<td>Ellis Island</td>
<td>6s</td>
<td>38s</td>
<td>44s</td>
<td>2.5s</td>
<td>0.4s</td>
</tr>
<tr>
<td>GendarmMarket</td>
<td>4s</td>
<td>49s</td>
<td>53s</td>
<td>11.1s</td>
<td>0.5s</td>
</tr>
<tr>
<td>Madrid</td>
<td>6s</td>
<td>36s</td>
<td>42s</td>
<td>3.2s</td>
<td>0.2s</td>
</tr>
<tr>
<td>Montreal N. D.</td>
<td>13s</td>
<td>40s</td>
<td>53s</td>
<td>9.1s</td>
<td>1.0s</td>
</tr>
<tr>
<td>Piazza D. P.</td>
<td>8s</td>
<td>31s</td>
<td>39s</td>
<td>3.3s</td>
<td>0.4s</td>
</tr>
<tr>
<td>Roman F.</td>
<td>18s</td>
<td>97s</td>
<td>115s</td>
<td>20.2s</td>
<td>1.3s</td>
</tr>
<tr>
<td>T. London</td>
<td>5s</td>
<td>42s</td>
<td>47s</td>
<td>1.9s</td>
<td>0.3s</td>
</tr>
<tr>
<td>U. Square</td>
<td>7s</td>
<td>48s</td>
<td>55s</td>
<td>6.8s</td>
<td>0.6s</td>
</tr>
<tr>
<td>V. Cathedra</td>
<td>33s</td>
<td>114s</td>
<td>147s</td>
<td>48.1s</td>
<td>2.1s</td>
</tr>
<tr>
<td>Yorkminster</td>
<td>6s</td>
<td>75s</td>
<td>83s</td>
<td>4.0s</td>
<td>0.4s</td>
</tr>
<tr>
<td>NYC Lib.</td>
<td>6s</td>
<td>43s</td>
<td>49s</td>
<td>4.8s</td>
<td>0.2s</td>
</tr>
<tr>
<td>Piccadilly</td>
<td>98s</td>
<td>262s</td>
<td>360s</td>
<td>449s</td>
<td>6s</td>
</tr>
</tbody>
</table>

Table 2. Time costs (sec.) of each step (AGF, Iterative Refinement) and the comparison with Chatterjee[2] (IRLS) and NeuRoRA[10] on 1DSfM.

Outlier Impact: We follow NeuRoRA[10] and test our pipeline w.r.t different levels of outliers in the louvre scene (624 nodes, YFCC100). In order to construct different outlier levels, similar to [10], for all edges in viewgraph, we first corrupt their ground-truth relative rotations by a Gaussian noise with std \( \sigma = 30° \). Then, we randomly select edges by the percentage of \( \gamma \) as outliers and replace their orientations with random orientations. From Table 3, when \( \gamma = 40\% \) of edges are outliers, the performance drops considerably.

<table>
<thead>
<tr>
<th></th>
<th>( \gamma = 5% )</th>
<th>( \gamma = 10% )</th>
<th>( \gamma = 40% )</th>
<th>( \gamma = 50% )</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean err.</td>
<td>2.1°</td>
<td>2.2°</td>
<td>3.3°</td>
<td>4.9°</td>
</tr>
<tr>
<td>median err.</td>
<td>3.13°</td>
<td>3.44°</td>
<td>4.83°</td>
<td>6.64°</td>
</tr>
</tbody>
</table>

Table 3. The rotation error in deg. w.r.t ratio of outliers (\( \gamma \)).
References


