

Appendix

A MODEL ARCHITECTURE

Here we detail the architecture of LAMP, complementary to Sec. 3.1. This architecture is used throughout all experiment, with just a few hyperparameter (*e.g.*, latent dimension, message passing steps) depending on the dimension (1D, 2D) of the problem.

A.1 ARCHITECTURE

In this subsection, we first detail the base architecture that is common among the evolution model f_{θ}^{evo} , the policy network $f_{\varphi}^{\text{policy}}$, and value network $f_{\varphi}^{\text{value}}$. Then describe their respective aspects, and explain the benefits of our action representation.

Base architecture. We use MeshGraphNets (Pfaff et al., 2021) as our base architecture, for the evolution model f_{θ}^{evo} , the policy network $f_{\varphi}^{\text{policy}}$, and value network $f_{\varphi}^{\text{value}}$. All three models have an encoder and a processor. For the encoder, we encode both node features and edge features, uplift to latent dimension using MLPs, to Z_{ij}^e, Z_i^v . For the processor, it consists of N message passing layers. For each message-passing layer, we first update the edge features using current edge features, and connected node features, $Z_{ij}^{(e)n+1} = \text{MLP}_{\theta}^{(v)}(E_{ij}^n, Z_i^{(v)n}, Z_j^{(v)n})$, then we update the node features using current node feature and connected edge features, $Z_i^{(v)n+1} = \text{MLP}_{\theta}^{(v)}(Z_i^{(v)n}, \sum_j Z_{ij}^{(e)n+1})$. Depending on the models and their intended functions, different models have different modules for predicting the output, described as follows.

Evolution model. For the evolution model, the output is at each node, and uses the last latent vector on the node, feeds into a decoder, to predict the output $\hat{V}_i^{t+1} = Z_i^{(v)N}$ (See Eq. 6) at the next time step $t + 1$.

Policy network and value network. Both $f_{\varphi}^{\text{policy}}$ and $f_{\varphi}^{\text{value}}$ shares the same processor $f_{\varphi}^{\text{processor}}$, which has N message-passing layers. The final node feature is appended with β for the controllability. The $f_{\varphi}^{\text{policy}}$ takes the encoded graph as input and consist of two parts - a global mean pooling is used to obtain the global latent representation of the whole graph, which is then feed into a $\text{MLP}^{(k)}$ followed by a action specific linear layer to predict the probability distribution for different number of actions should be performed; an action specific $\text{MLP}^{(a)}$ is applied on each edge to predict the probability of a certain action (*i.e.*, split, coarse) to be applied on this edge. The $f_{\varphi}^{\text{value}}$ takes the same encoded graph as input, and perform a global mean pooling to obtain the global latent features for the graph, which is then feed through a linear layer to predict the accuracy reward, and another linear layer to predict the computation reward, the final predicted value is, $v = \text{loss} + \beta \cdot \text{compute}$.

A.2 ARCHITECTURAL HYPERPARAMETERS USED IN 1D AND 2D EXPERIMENTS

Here we detail the architectures used in the 1D and 2D experiment. A summary of the hyperparameters is also provided in Table 3.

A.2.1 1D NONLINEAR PDES

For 1D experiment, our evolution model f_{θ}^{evo} has $N = 3$ message-passing layers in the processor and latent dimension of 64. It uses the SiLU activation (Elfwing et al., 2018). The shared processor for $f_{\varphi}^{\text{policy}}$ and $f_{\varphi}^{\text{value}}$ has $N = 3$ message-passing layers and latent dimension of 64.

A.2.2 2D MESH-BASED SIMULATION

For f_{θ}^{evo} , we set message passing layers for MeshGraphNets (Pfaff et al., 2021) to be 8. We also model our message passing function with MLP having 56 units followed by SiLU activation (Elfwing et al., 2018). Here, we share the same message passing function across all the 8 layers because we found in our experiments that sharing the same message passing function achieved better

performance than having independent MLP for each of the layers. The shared processor for $f_{\varphi}^{\text{policy}}$ and $f_{\varphi}^{\text{value}}$ has $N = 3$ message-passing layers and latent dimension of 64.

B EXPERIMENT DETAILS

In this section, we provide experiment details for 1D and 2D datasets. Firstly, we explain the reasoning behind several design choices of LAMP and their benefits. Then we provide the training procedure summary and hyperparameter table in Appendix B.1, followed by specific training details for 1D (Appendix B.2) and 2D (Appendix B.3). Finally, we detail the baselines in Appendix B.4.

Use of r^t as value target. Different from typical RL learning scenario (computer games or robotics) where the episode always has an end and the reward is bounded, here in physical simulations, there are two distinct characteristics as follows. (1) The rollout can be performed infinite time steps into the future, (2) as the rollout continues, at some point the error between predicted state and ground-truth state will diverge, so the error is not bounded. Based on these two characteristics, a value target based on the error for infinite horizon (*e.g.*, the one used in Dreamer v2 (Hafner et al., 2021)) does not make sense, since the error will not be bounded. In our experiment, we also observe similar phenomena, in which both the value target and value prediction continue to increase indefinitely. Thus, we use the average reward in the S step rollout as the value target, which measures the error and computation improvement within the rollout window we care about and proves to be much more stable.

Benefit of our action space definition. we have provided the description of action representation in Section our definition of action space in Section 3.1. Compared to an independent sampling on each edge, the above design of action space has the following benefits:

- The action space reduces from $2^{N_{\text{edge}}}$ to $N_{\text{edge}}^{K^{\max}}$, where N_{edge} is the number of edges. In the case of $N_{\text{edge}} \sim 1000$ and $K^{\max} \sim 10 \ll N_{\text{edge}}$, the difference in action space dimensionality is significant, *e.g.*, $2^{1000} = 10^{300}$ vs. $1000^{10} = 10^{30}$. Therefore, it is easier to credit assign the reward to appropriate action, with a smaller space of action.
- Compared with each edge performing action independently, now only $K \leq K^{\max}$ actions of refinement or coarsening can be performed. Therefore, it will need to focus on a few actions that can best improve the objective.
- The sampling of K will also make the policy more “controllable”, since now the K is explicitly dependent on the β , and learning how many refinement or coarsening action to take depending on β is much easier than figuring out which concrete independent actions to take.

B.1 TRAINING PROCEDURE SUMMARY

Here we provide the training procedure for learning the evolution model and the policy. There are two stages of training. The first stage is pre-training the evolution model alone without remeshing, and the second stage is alternatively learning the policy with RL and finetuning the evolution model. The detailed hyperparameter table for 1D and 2D experiments is provided in Table 3. We train all our models on an NVIDIA A100 80GB GPU.

Pre-training. In the pre-training stage, the evolution model is trained without remeshing, and the loss is computed by rolling out the evolution model for S steps, and the loss is given by $\text{loss} = (\text{1-step loss}) \times 1 + (\text{2-step loss}) \times 0.1 + \dots + (\text{S-step loss}) \times 0.1$ (the number S is provided in Table 3). We use a smaller weight for later steps, so that the training is more stable (since at the beginning of training, the evolution model can have large error, having too much weight on later steps could result in large error and make the training less stable). The pre-training lasts for certain number of epochs (see Table 3), before proceeding to the next stage.

Joint training of policy and evolution model. In this stage, the actor-critic and the evolution model are trained in an alternative fashion. Specifically, in the policy-learning phase, the evolution model is frozen, and the actor-critic is learned via reinforcement learning (see “learning the policy” part of Sec. 3.2) for J^{policy} steps, and in the evolution-learning phase, the actor-critic is frozen, and the

evolution model is learned according to the “learning evolution” part of Sec. 3.2 for J^{evo} steps. These two phases proceed alternatively.

The reasoning behind using alternating training strategy for actor-critic and evolution is as follows. When learning the actor-critic, the *return* for the RL is the expected reward based on the *current* policy and evolution model. If at the same time the evolution model is also optimized together, then the reward function will always be changing, which will likely make learning the policy harder. Therefore, we adopt the alternating training strategy, which is also widely used in many other applications, such as in GAN training.

In the following two subsections, we detail the action space and specific settings for 1D and 2D.

Table 3: Hyperparameters used for model architecture and training.

Hyperparameter name	1D dataset	2D dataset
<i>Hyperparameters for model architecture:</i>		
Temporal bundling steps	25	1
f_{θ}^{evo} : Latent size	64	56
f_{θ}^{evo} : Activation function	SiLU	SiLU
f_{θ}^{evo} : Encoder MLP number of layers	3	4
f_{θ}^{evo} : Processor number of message-passing layers	3	8
$f_{\varphi}^{\text{policy}}$: Latent size	64	56
$f_{\varphi}^{\text{policy}}$: Activation function	ELU	ELU
$f_{\varphi}^{\text{policy}}$: Encoder MLP number of layers	2	2
$f_{\varphi}^{\text{policy}}$: Processor number of message-passing layers	3	3
$f_{\varphi}^{\text{policy}}$: MLP ^(k) : MLP number of layers	2	2
$f_{\varphi}^{\text{policy}}$: MLP ^(k) : Activation function	ELU	ELU
$f_{\varphi}^{\text{policy}}$: MLP ^(a) : MLP number of layers	3	3
$f_{\varphi}^{\text{policy}}$: MLP ^(a) : Activation function	ELU	ELU
<i>Hyperparameters for training:</i>		
β sampling range \mathcal{B}	[0, 0.5]	{0}
Loss function	MSE	L2
α_s^{policy} , for $s = 1, 2, \dots$ (Eq. 10)	{1, 1, 1, ...}	{1, 1, 1, ...}
α_s^{evo} , for $s = 1, 2, \dots$ (Eq. 10)	{1, 1, 1, ...}	{1, 1, 1, ...}
Number of epochs for pre-training evolution model	50	100
Number of rollout steps S to for multi-step loss during pre-training	4	1
Number of epochs for joint training of actor-critic and evolution model	30	30
J^{policy} : # of steps for updating the actor-critic during joint training	200	200
J^{evo} : # of steps for updating the evolution model during joint training	100	100
Batch size	128	64
Evolution model learning rate for pre-training	10^{-3}	10^{-3}
Evolution model learning rate during policy learning	10^{-4}	10^{-4}
Value network learning rate	10^{-4}	10^{-4}
Policy network learning rate	5×10^{-4}	5×10^{-4}
Optimizer	Adam	Adam
Coefficient for value loss	0.5	0.5
K^{max} : Maximum number of actions for coarsen or refine	20	20
Maximum gradient norm	2	20
Optimizer scheduler	cosine	cosine
Input noise amplitude	0	10^{-2}
S : Horizon	4	6
Weight decay	0	0
η : Entropy coefficient	10^{-2}	2×10^{-2}

B.2 1D NONLINEAR PDES

The action space for 1D problem is composed of split and coarsen actions. The coarse action is initially defined on all edges, which is then sampled based on the predicted number of coarsening actions, and probability of each edge to be coarsened. Among those sampled edges, if two edges share a common vertex, only the rightmost one will be coarsened.

During pre-training, to let the evolution model adapt to the varying size of the mesh, we perform random vertex dropout, where we randomly sample 10% of the minibatch to perform dropout, and if a minibatch is selected for node dropout, for each example, randomly drop 0-30% of the nodes in the mesh.

Here the interpolation g^{interp} (in Eq. 4) uses barycentric interpolation (Berrut & Trefethen, 2004), where during refinement, the value of the node feature for the newly added node is a linear combination of its two neighbors' node features, depending on the coordinates of the newly-added node and the neighboring nodes.

B.3 2D MESH-BASED SIMULATION

In this subsection, we provide details on definition of remeshing actions, invalid remeshing action, how to generate 2d mesh data based on Narain et al. (2012) as well as how to pre-train the evolution model on the generated data, and the interpolation method to remedy inconsistency between vertex configurations of different meshes.

Action space. There are three kinds of remeshing operations defined in the 2D mesh simulation: split, flip, and coarsen. The action space for RL is defined as the product of sets of splitting and coarsening operations (the flipping is automatically performed). Elements of each set indicate edges in a mesh and the policy function chooses up to K^{max} of the elements as edges to split or coarsen. The reason why the flip action is not an action of RL is because we flip all edges satisfying some condition (will be explained in the following). The rest of this paragraph provides details on respective remeshing actions. Split action in 2D case can be performed on any edges that are shared with two triangular faces in a mesh and also on the boundary edges of the mesh. The split action results in 4 triangles (Fig. 5a). Flip operation is also performed on edges shared with two faces, but it also requires some additional condition, that is when sum of angles at vertices located at opposite side of edges is greater than π : see also Fig 5b. In our remeshing function, all edges satisfying the condition are flipped, which is the reason why flip operation is not a component of the action space for RL. Finally, coarsening action has relatively strong conditions. One condition is that one of the source or target nodes of a coarsened edge needs to be of degree 4 and another condition is that all the faces connected to the node of degree 4 needs to have acute angles except angles around the node. See also Fig. 5c. We filter the sampled actions from f_{ϕ}^{policy} based on aforementioned conditions to get sets of valid edges to split and coarse.

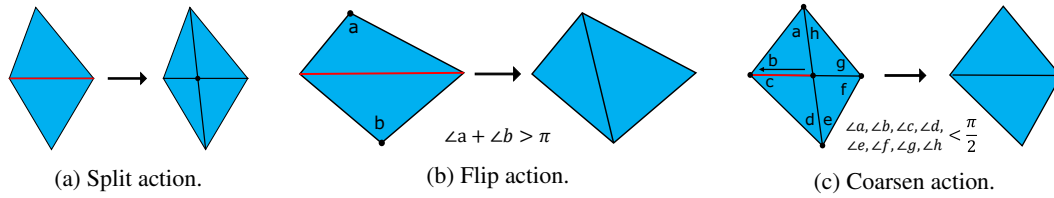


Figure 5: Illustration of split, flip and coarsen actions. The actions are performed on edges.

Invalid remeshing. As described in Section 3.1, two edges on the same face of a mesh cannot be refined at the same time, nor can they be both coarsened. This is because by doing so we may have an invalid mesh with some non-triangular faces such as quadrilaterals. We can give such an example as follows; see also Fig. 6 for reference. Suppose that we have a triangular face ABC and we split edges AB and AC by D and E, we have new edges DC and EB. If we denote the intersection of DC and EB by F, we will have a quadrilateral ADFE, which violates the requirement that all faces must be triangles. In order to avoid this situation, remeshing and coarsening action can only be performed on up to one edge of every face.

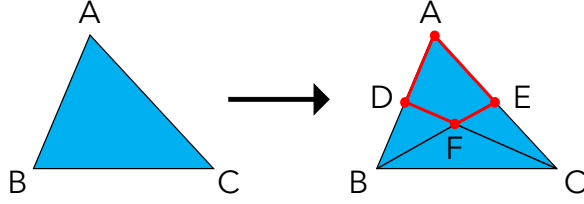


Figure 6: Invalid split action on edges. Splitting two edges in a same triangular face at the same time gives a quadrilateral.

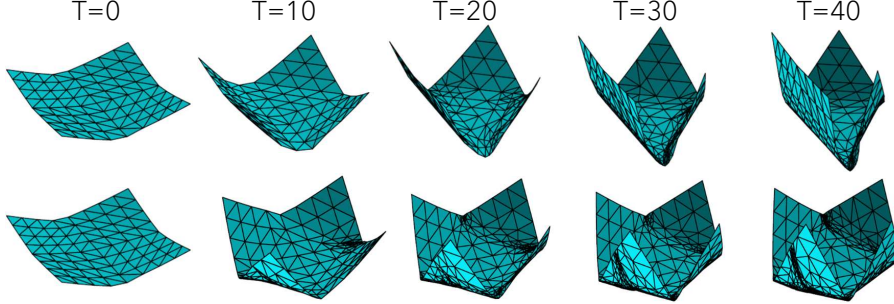


Figure 7: Part of trajectories generated with different configurations. As the time goes, finer triangular faces are added to high-curvature regions in meshes.

Generating data. To generate ground-truth data, we use a 2D triangular mesh-based AMR simulator [Narain et al. \(2012\)](#). This simulator adopts adaptive anisotropic kernel method to automatically conform to the geometric and dynamic detail of the simulated cloth: see also Fig. 7. In our experiment, each trajectory consists of 325 meshes (excluding initial mesh.) Each mesh consists of vertices and faces and every vertex is equipped with its 2D coordinates as well as 3D world coordinates. Time frame between two consecutive meshes are set to be 0.04. The edge length was set to range between 0.01 and 0.2. Note that decision on respective remeshing actions is made based on the 2D coordinates.

For training and test dataset, we generate 1050 configurations specifying direction and magnitude of forces applied to 4 corners of meshes, and generate 1000 trajectories for training data and use 50 trajectories as test data based on the generated configurations. When training our evolution model, we downsample data by half of the original data; we use every other meshes starting from an initial mesh in each trajectory. The reason of downsampling is that we observed that cumulative RMSE over rollout with model trained with full trajectories blew up after iteration exceeded 160 steps.

Noted that all dataset are pre-generated and will be loaded during the training, so that no ground-truth solver is needed during the training of LAMP.

Pre-training of evolution model. We model our evolution function f_{θ}^{evo} with MeshGraphNets [\(Pfaff et al., 2021\)](#). The input of the model adopts a graph representation where a vertex is equipped with 3D velocity computed from mesh information at both current and past time steps, and feature for an edge consists of relative distance and its magnitude of boundary vertices of the edge. Since mesh topology varies during the forward iteration, we perform barycentric interpolation on the mesh at past time step to get interpolated vertex coordinates corresponding to vertices at current time step.

When evaluating our model, we use rollout RMSE, taking the mean for all spatial coordinates, all mesh nodes, all steps in each trajectory, and all 50 trajectories in the test dataset. The rollout RMSE achieved 4.45×10^{-2} with our best parameter setting. We found that adding noise helped to improve its performance; we added the noise with scale 0.01 to vertex coordinates.

Interpolation. Since we may have different mesh topology at each time step in a trajectory, it is not possible to simply compare node features at one time step with those at another time steps or even at the same time step if the mesh to compare is in a different trajectory. When we compare node features on different meshes, we perform barycentric interpolation [\(Berrut & Trefethen, 2004\)](#). For velocity in 2D simulation used as the input of the evolution model, we interpolate mesh at time step

$t - 1$ into mesh at t and take the difference between them. To compute the rollout error metrics, we always interpolate the predicted mesh to the ground-truth mesh at each step, and compute the metrics in the ground-truth mesh.

B.4 BASELINES

Here we provide additional details on the baselines used in 2D mesh-based simulation. All the baselines use pre-trained evolution function f_{θ}^{evo} as part of the respective forward models. In the following, we mainly give details on functions responsible for choosing edges to split and coarsen which correspond to the policy function $f_{\varphi}^{\text{policy}}$ in LAMP’s forward model.

MeshGraphNets + heuristic remeshing. At each step, instead of having $f_{\varphi}^{\text{policy}}$ to infer probability on edges to split and coarsen, we compute local curvature on edges of the mesh and use the curvature to filter out edges to split. Here, the local curvature on an edge is defined as the angle made by two unweighted normal vectors on boundary nodes of the edge (the normal vector on nodes is defined as the mean of normal vectors on faces surrounding the nodes). When we filter out edges in the mesh, we first choose edges with curvature exceeding pre-defined threshold, which is set to be 0.1 in the experiment. We next check that after splitting, the edges are not shorter than a pre-defined minimum length, which is set to be 0.04, to avoid having exceedingly small faces. Edges violating either of these criteria are not chosen as edges to split.

MeshGraphNets + GT remeshing. In this baseline, we use the mesh configuration of ground-truth meshes provided by the classical solver as the mesh configuration of predicted meshes. Specifically, after having f_{θ}^{evo} predict a mesh at the next time step, we interpolate the predicted mesh into the ground-truth mesh. We generate all the ground-truth meshes with the same initial condition as LAMP’s rollout experiment.

LAMP (no remeshing). This baseline does not involve any functions responsible for remeshing and we just recursively apply f_{θ}^{evo} to evolve meshes. Therefore, mesh topology of all meshes in a trajectory is isomorphic.

C ADDITIONAL RESULT VISUALIZATION

C.1 1D NONLINEAR PDES

In this section, we provide additional randomly sampled results for the 1D nonlinear PDE experiment, as shown in Fig. 8. We see that similar to Fig. 2, our LAMP is able to add more vertices to the locations with more dynamicity, while remove more vertices at more static regions. Moreover, with increasing β that emphasizes more on reducing computational cost, LAMP splits less and coarsens more.

C.2 2D MESH-BASED SIMULATION

In this section, we provide additional randomly sampled results for the 2D mesh-based simulation, as shown in Fig. 9. We see that our LAMP (fifth column) learns to add more edges to locations with higher curvature, resulting in better rollout performance than with no remeshing.

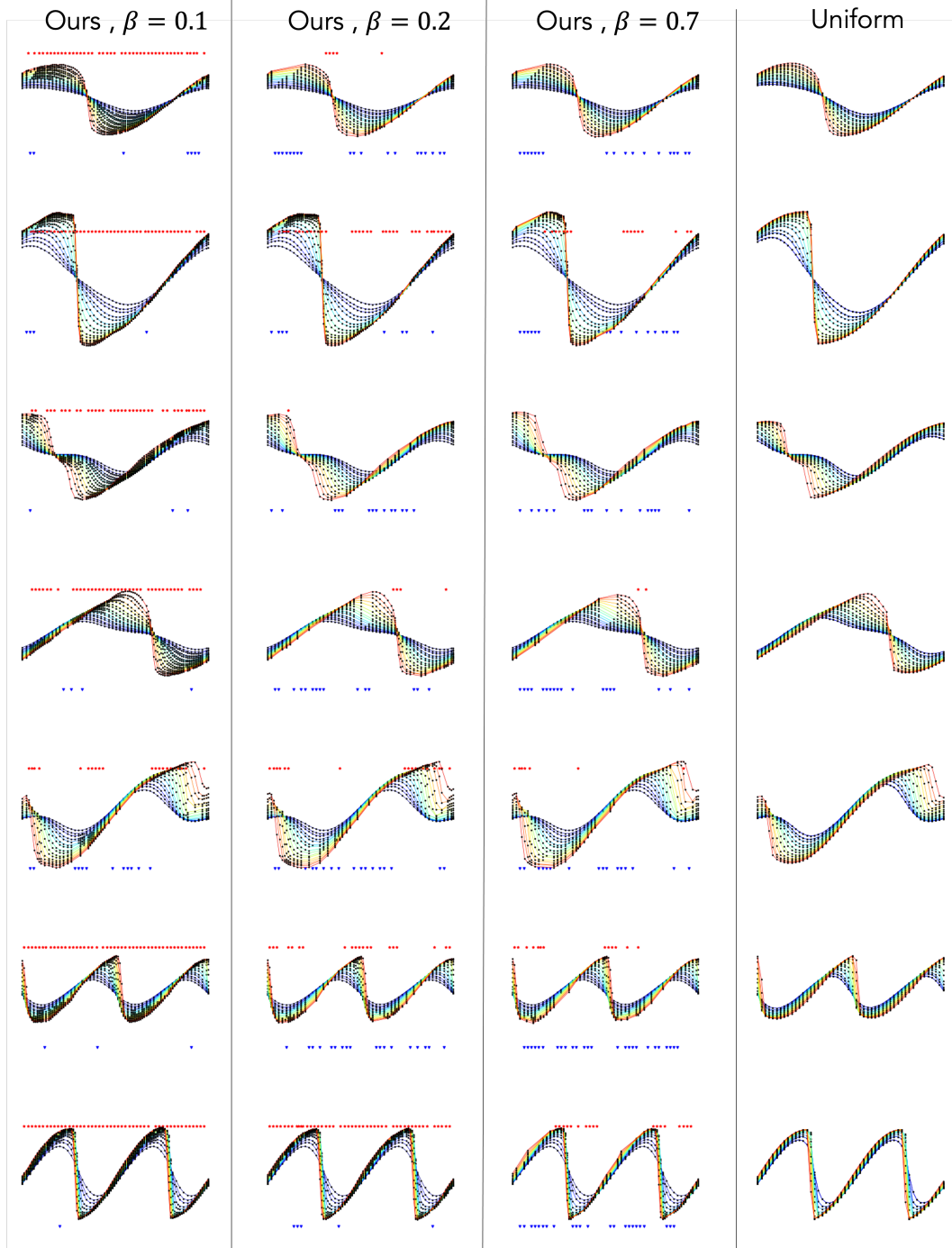


Figure 8: Additional example rollout results with different β of our LAMP on 1D nonlinear PDEs with initial state on 50 vertices which is uniformly downsampled from the 100 vertices. The rollout is performed over 200 time steps, where different color denotes the system’s state at different time. The upper red dots and lower blue dots shows the added and removed nodes of the mesh, comparing the end and the initial mesh.

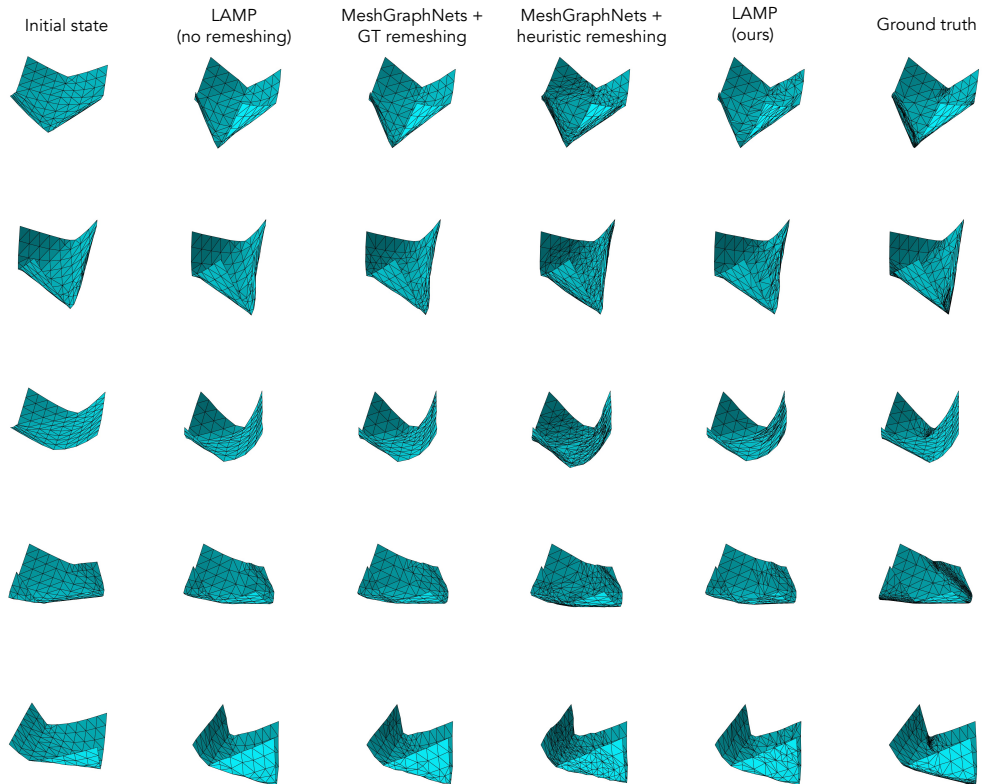


Figure 9: Additional example rollout results for 2D paper folding, at $t = 20$. We see that our LAMP (fifth column) learns to add more edges to locations with higher curvature, and learns to coarsen on the flat region (third and fourth row of “LAMP (ours)”, where we see coarsening in the middle of our meshes), resulting in better rollout performance than with no remeshing and other baselines.