LEARNING CONTROLLABLE ADAPTIVE SIMULATION FOR MULTI-SCALE PHYSICS

Anonymous authors
Paper under double-blind review

ABSTRACT

Simulating the time evolution of physical systems is pivotal in many scientific and engineering problems. An open challenge in simulating such systems is their multi-scale dynamics: a small fraction of the system is extremely dynamic, and requires very fine-grained resolution, while a majority of the system is changing slowly and can be modeled by coarser spatial scales. Typical learning-based surrogate models use a uniform spatial scale, which needs to resolve to the finest required scale and can waste a huge compute to achieve required accuracy. In this work, we introduce Learning controllable Adaptive simulation for Multi-scale Physics (LAMP) as the first full deep learning-based surrogate model that jointly learns the evolution model and optimizes appropriate spatial resolutions that devote more compute to the highly dynamic regions. LAMP consists of a Graph Neural Network (GNN) for learning the forward evolution, and a GNN-based actor-critic for learning the policy of spatial refinement and coarsening. We introduce learning techniques that optimizes LAMP with weighted sum of error and computational cost as objective, which allows LAMP to adapt to varying relative importance of error vs. computation tradeoff at inference time. We test our method in a 1D benchmark of nonlinear PDEs and a challenging 2D mesh-based simulation. We demonstrate that our LAMP outperforms state-of-the-art deep learning surrogate models with up to 60.5% error reduction, and is able to adaptively tradeoff computation to improve long-term prediction error.

1 INTRODUCTION

Simulating the time evolution of a physical system is of vital importance in science and engineering (Lynch, 2008; Carpanese, 2021; Sircombe et al., 2006; Courant et al., 1967; Lelievre & Stoltz, 2016). Usually, the physical system has a multi-scale nature: a small fraction of the system is highly dynamic, and requires very fine-grained resolution to simulate accurately, while a majority of the system is changing slowly. Examples include hazard prediction in weather forecasting (Majumdar et al., 2021), disruptive instabilities in the plasma fluid in nuclear fusion (Kates-Harbeck et al., 2019), air dynamics near the boundary for jet engine design (Athanasopoulos et al., 2009), and more familiar examples such as wrinkles in a cloth (Pfaff et al., 2021) and fluid near the boundary for flow through the cylinder (Vlachas et al., 2022) Due to the typical huge size of such systems, it is pivotal that those systems are simulated not only *accurately*, but also with as small of a *computational cost* as possible. A uniform spatial resolution that pays similar attention to regions with vastly different dynamics, will waste significant compute on slow-changing regions while may be insufficient for highly dynamic regions.

To accelerate physical simulations, deep learning (DL)-based surrogate models have recently emerged as a promising alternative to complement (Um et al., 2020) or replace (Li et al., 2021) classical solvers. They reduce computation and accelerate the simulation with larger spatial (Um et al., 2020; Kochkov et al., 2021) or temporal resolution (Li et al., 2021), or via latent representations (Sanchez-Gonzalez et al., 2020; Wu et al., 2022). However, current deep learning-based surrogate models typically assume a uniform or fixed spatial resolution, without *learning* how to best assign computation to the most needed spatial region. Thus, they may be insufficient to address the aforementioned multi-scale challenge. Although adaptive methods, such as Adaptive Mesh Refinement (AMR) (Soner et al., 2003; Cerveny et al., 2019) exist for classical solvers, they share

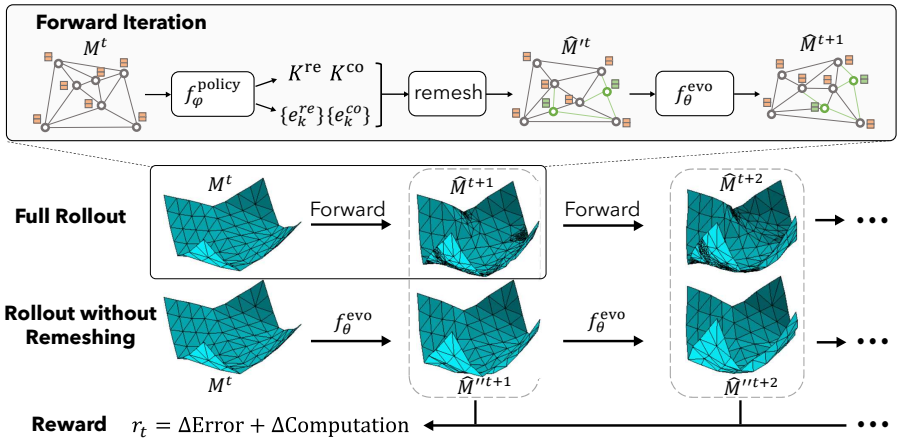

Figure 1: LAMP schematic. The forward iteration (upper box) first uses the policy $f_\varphi^{\text{policy}}$ to decide the number K^{re} and K^{co} of edges as well as which edges among the full mesh to be refined or coarsened, and then executes remeshing and interpolation. The evolution model f_θ^{evo} is applied to the updated mesh $\hat{M}'^t$ to predict the state $\hat{M}^{t+1}$ at the next time step. We use the *reduction* of both *Error* and *Computation* (mesh size), compared to a multi-step rollout without remeshing, as reward to learn the policy.

similar challenge (e.g. slow) as classical solvers. A deep learning-based surrogate models, that is able to learn both the evolution and learn to assign computation to the needed region, is needed.

In this work, we introduce Learning controllable Adaptive simulation for Multi-scale Physics (LAMP) as the first fully DL-based surrogate model that jointly learns the evolution model and optimizes appropriate spatial resolutions that devote more compute to the highly dynamic regions. Our key insight is that by explicitly setting the error and computation as the combined objective to optimize, the model can learn to adaptively decide the best local spatial resolution to evolve the system. To achieve this goal, LAMP consists of a Graph Neural Network (GNN)-based evolution model for learning the forward evolution, and a GNN-based actor-critic for learning the policy of discrete actions of local refinement and coarsening of the spatial mesh, conditioned on the local state and a coefficient β that weights the relative importance of error vs. computation. The policy (actor) outputs both the *number* of refinement and coarsening actions, and *which* edges to refine or coarsen, while the critic evaluates the return of the current policy. The full system is trained with an alternating fashion, iterating between training the evolution model with supervised loss, and training the actor-critic via reinforcement learning (RL). Taken together, a single instance of evolution model and actor-critic jointly optimizes reduction of error and computation for the physical simulation, and can operate across the relative importance of the two metrics at inference time.

We evaluate our model on a 1D benchmark of nonlinear PDEs (which tests generalization across PDEs of the same family), and a challenging 2D mesh-based simulation of paper folding. In 1D, we show that our model outperforms state-of-the-art deep learning-based surrogate models in terms of long-term evolution error by up to 60.5%, and can adaptively tradeoff computation to improve long-term prediction error. On a 2D mesh-based simulation, our model can strategically choose appropriate edges to refine or coarsen, and outperforms a baseline without remeshing.

2 Problem Setting and Related Work

We consider the numerical simulation of a physical system, following the notation introduced in (Pfaff et al., 2021). The system's state at time t is discretized into the mesh-based state $M^t = (V^t, E^t), t = 0, 1, 2, ...$, where E^t is the mesh edges and V^t is the states at the nodes at time t. Each node $i \in V$ contains the mesh-space coordinate u_i and dynamic features q_i. Note that this representation of the physical system is very general. It includes Eulerian systems (Wu et al., 2022) where the mesh is fixed and the field q_i on the nodes are changing, and Lagrangian systems (Sanchez-Gonzalez et al., 2020; Pfaff et al., 2021) where the mesh coordinate in physical space is also dynamically moving (for this case, an additional world coordinate x_i is accompanying the

mesh coordinate u_i). During prediction, a simulator f (classical or learned) autoregressively predicts system's state $\hat{M}^{t+1}$ at the next time step based on its previous prediction $\hat{M}^t$ at previous time step:

$$\hat{M}^{t+1} = f(\hat{M}^t), t = 0, 1, 2, ... \tag{1}$$

where $\hat{M}^0 = M^0$ is the initial state. During the prediction, both the dynamic features V^t at the mesh nodes, and the mesh topology E^t can be changing. The error is typically computed by comparing the prediction and ground-truth after long-term prediction: error $:= \ell(\hat{M}^t, M^t)$ for a metric ℓ (e.g. MSE, RMSE), and the computation cost (in terms of floating point operations, or FLOPs) typically scales with the size of the mesh (e.g. the number of nodes). The task is to evolve the system long-term into the future, with a low error and a constraint on the computational cost.

Most classical solvers use a fixed mesh E^t whose topology does not vary with time. For example, the mesh $E^t \equiv E^0$ can be a 2D or 3D regular grid, or an irregular mesh that is pre-generated at the beginning of the simulation (Geuzaine & Remacle, 2009). Classical Adaptive Mesh Refinement (AMR) (Narain et al., 2012) addresses the multi-scale challenge by adaptively refine or coarsen the mesh resolution, with heuristics based on local state variation. Since they are based on classical solvers, they may not benefit from the many advantages that deep learning brings (GPU acceleration, less stringent spatial and temporal resolution requirement, explicit forward, etc.). In contrast, our LAMP is a deep-learning based surrogate model, and can benefit from the many advantages (e.g. speedup) offered by the deep learning framework. Furthermore, since it directly optimize for a linear combination of error and computation, it has the potential to directly optimize to a better error vs. computation tradeoff, nearer to the true Pareto frontier.

Deep-learning based surrogate models, although having achieved speedup compared to classical solvers, still typically operate on a fixed grid or mesh (Li et al., 2021; Sanchez et al., 2020; Wu et al., 2022; Zhao et al., 2022), and have yet to exploit the multi-scale nature typical in physical simulations. One important exception is MeshGraphNets (Pfaff et al., 2021), which both learns how to evolve the state V^t, and uses supervised learning to learn the spatial adaptation that changes E^t. However, since it uses supervised learning where the ground-truth mesh is provided from the classical solver with AMR, it cannot exceed the performance of AMR in terms of error vs. computation tradeoff, and has to interact with the classical solver in inference time to perform the adaptive mesh refinement. In contrast, our LAMP directly optimizes for the objective, which uses reinforcement learning for learning the the policy of refinement and coarsening, and has the potential to surpass classical AMR and achieve a better error vs. computation tradeoff. Moreover, a single trained LAMP has the ability to adapt to the full range of relative importance β of error vs. computation at inference time, thus can be more versatile than MeshGraphNets with a fixed strategy.

3 Method

In this section, we detail our LAMP method. We first introduce its architecture in Sec. 3.1. Then we introduce its learning method (Sec. 3.2), including learning objective and training, and technique to let it learn to adapt to varying importance of error and computation.

3.1 Model architecture

The model architecture of LAMP consists of two components. An actor-critic which updates the mesh topology, and an evolution model which evolves the states defined on the mesh. We will detail them one by one.

Actor-critic. The actor-critic consists of a policy network $f_\varphi^{\text{policy}}$ (with parameter φ) which predicts the probability of performing the spatial coarsening or refining actions, and a value network f_φ^{value} which evaluates the long-term expected reward of the policy network:

$$P(A = a^t | M = M^t) = p_\varphi(a^t | M^t, \beta) = f_\varphi^{\text{policy}}(M^t, \beta) \tag{2}$$

$$\hat{v}^t = f_\varphi^{\text{value}}(M^t, \beta) \tag{3}$$

where a^t is the (refining and coarsening) action performed on the edges E^t so that it will become $\hat{E}^{t+1}$. The policy network $f_\varphi^{\text{policy}}$ outputs the probability of performing such action and can sample

from this probability. v^t estimates the "value" of the current policy starting from current state M^t (for more information, see Sec. 3.2 below). The explicit dependence on β (as the β in Eq. 8) allows the policy and value network to condition on the varying importance of error and computation. Given the predicted mesh $\hat{E}^{t+1}$ and the current node features V^t on the current mesh E^t, an interpolation is performed which obtains the node features on the new mesh:

$$\hat{V}'^t = g^{\text{interp}}(V^t, \hat{E}^{t+1}, E^t) \tag{4}$$

Now the new intermediate state $\hat{M}'^t = (\hat{V}^t, \hat{E}^{t+1})$ is defined on the new mesh $\hat{E}^{t+1}$.

Evolution model. The second component is an evolution model f_θ^{evo} which takes as input the intermediate state M'^t defined on $\hat{E}^{t+1}$, and outputs the prediction of node features $\hat{V}^{t+1}$ for time $t+1$:

$$\hat{V}^{t+1} = f_\theta^{\text{evo}}(M'^t) \tag{5}$$

Note that in this stage, the mesh topology $\hat{E}^{t+1}$ is kept constant, and the evolution model f_θ^{evo} (with parameter θ) learns to predict the state based on the current mesh.

Taken together, Eqs. (2)(4)(5) evolve the system state from M^t at time t to state $\hat{M}^{t+1} = (\hat{V}^{t+1}, \hat{E}^{t+1})$ at $t+1$. During inference, they are executed autoregressively following Eq. (1), to predict the system's future states $\hat{M}^t, t = 1, 2, 3, ...$, given an initial state M^0.

GNN architecture. One requirement for the evolution model f_θ^{evo}, policy network $f_\varphi^{\text{policy}}$ and value network f_φ^{value} is that they can operate on changing mesh topology E^t. Graph Neural Networks (GNNs) are an ideal choice that meets this requirement. Since we represent the system's state as mesh, we adopt MeshGraphNets (Pfaff et al., 2021) as the base architecture for the above three models. Specifically, we encode V^t as node features for the graph, and encode the mesh topology E^t as edges and world edges as two types of edges, and the edge features depend on the relative positions in the mesh coordinates and world coordinates. Based on the graph, N layers of message passing are performed to locally exchange and aggregate the information:

$$Z_{ij}^{(e)n+1} = \text{MLP}_\theta^{(v)}(E_{ij}^n, Z_i^{(v)n}, Z_j^{(v)n}) \tag{6}$$

$$Z_i^{(v)n+1} = \text{MLP}_\theta^{(v)}(Z_i^{(v)n}, \sum_j Z_{ij}^{(e)n+1}). \tag{7}$$

where $Z_i^{(v)n}$ is the latent node vector on node i at layer n, and $Z_{ij}^{(e)n}$ is the latent edge vectors on the nth layer on the edge from node i to node j. We have $Z_i^{(v)0} = \hat{V}_i'^t$ and $Z_i^{(v)N} = \hat{V}^{t+1}$ are input and predicted node features at time t and $t+1$, respectively, in Eq. (5). Figure 1 provides an illustration of the architecture. We use an independent processor for the evolution model, and share the processor for the policy and value networks. After the processor, the latent vectors are concatenated with β to feed into downstream decoders. For the evolution model f_θ^{evo}, a decoder is operated on the latent state and outputs the prediction $\hat{V}'^{t+1}$ on the nodes. For the value network, an value MLP is operated on all nodes, and a global pooling is performed to compute the overall estimated value. For the policy network, we design the action decoder as follows.

Action representation. To predict the action for policy network and its probability, we first need to design the action space. Note that compared to typical reinforcement learning problems, here the action space is extremely high-dimensional and complicated: (1) each edge of the mesh may have the option of choosing refinement or coarsening. If there are thousands of edge, then the possible actions will be on the order of $2^{N_{\text{edge}}}$. (2) Not all actions are valid, and many combinations of actions are invalid. For example, two edges on the same face of the mesh cannot be refined at the same time, nor can they be both coarsened. To address this high-dimensionality action problem, we introduce the following design of action space, where for both refinement and coarsening, the policy network first samples integers $K^{\text{re}}, K^{\text{co}} \in \{0, 1, 2, ...K^{\text{max}}\}$, and then independently samples K^{re} edges to perform refinement and K^{co} edges to perform coarsening. The combined action is $a^t = (K^{\text{re}}, e_1^{\text{re}}, e_2^{\text{re}}, ...e_{K^{\text{refine}}}^{\text{re}}, K^{\text{co}}, a_1^{\text{co}}, a_2^{\text{co}}, ...a_{K^{\text{co}}}^{\text{co}})$, where $K^{\text{re}}, K^{\text{co}} \in \{0, 1, ...K^{\text{max}}\}$, and $e_k^{\text{re}}, e_k^{\text{co}} \in E^t, k = 1, 2, ...$ The combined log-probability of choosing the action a^t is given by:

$$\log p_\varphi(a^t|M^t) = \log p_\varphi(K^{\text{re}}|M^t) + \sum_{k=1}^{K^{\text{re}}} \log p_\varphi(e_k^{\text{re}}|M^t) + \log p_\varphi(K^{\text{co}}|M^t) + \sum_{k=1}^{K^{\text{co}}} \log p_\varphi(e_k^{\text{co}}|M^t)$$

Compared to an independent sampling on each edge, the above design has the following benefits:

- The action space reduces from $2^{N_{\text{edge}}}$ to ${N_{\text{edge}}}^K$, where N_{edge} is the number of edges. In the case of $N_{\text{edge}} \sim 1000$, 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 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.

3.2 LEARNING

The ultimate goal of the learning for LAMP is to optimize the objective Eq. (8) as follows:

$$L = (1 - \beta) \cdot \text{Error} + \beta \cdot \text{Computation} \tag{8}$$

for a wide range of β. To achieve this, we break down the above objective into an alternative learning of two phases: learning the evolution model with objective L^{evo} that minimizes long-term evolution error, and learning the policy with objective L^{policy} that optimizes both the long-term evolution error and computational cost.

Learning evolution. In this phase, the evolution model f_θ^{evo} is optimized to reduce the multi-step evolution *error* for the evolution model. As before, we denote $M^{t+s}, t = 0, 1, 2, ..., s = 0, 1, ...S$ as the state of the system at time $t + s$ simulated by the ground-truth solver with very fine-grained mesh, and denote $\hat{M}^{t+s}, t = 0, 1, 2, ..., s = 0, 1, 2, ..S$ as the prediction by the current LAMP following the current policy, up to a horizon of S steps into the future. We further denote $\hat{M}''^{t+s}, t = 0, 1, 2, ..., s = 0, 1, 2, ..S$ as the prediction by the current evolution model on the fine-grained mesh, where its mesh is provided as ground-truth mesh E^{t+s} at each time step. Then the loss is given by:

$$L^{\text{evo}} = L_S^{\text{evo}}[f^{\text{policy}}, f^{\text{evo}}; \hat{M}^t] + L_S^{\text{evo}}[\mathbb{I}, f^{\text{evo}}; \hat{M}''^t] \tag{9}$$

$$= \sum_{s=1}^{S} \alpha_s^{\text{policy}} \ell(\hat{M}^{l+s}, M^{l+s}) + \sum_{s=1}^{S} \alpha_s^{\mathbb{I}} \ell(\hat{M}''^{l+s}, M^{l+s}) \tag{10}$$

Essentially, we optimize two parts of the evolution loss: (1) $L_s^{\text{evo}}[f^{\text{policy}}, f^{\text{evo}}; \hat{M}^t]$ which is the evolution loss by following policy network $f_\varphi^{\text{policy}}$ and evolution model f_θ^{evo}, starting at initial state of $\hat{M}^t$ for S steps. (here α_s^{policy} is the coefficient for the s-step loss with loss function ℓ). This makes sure that evolution model f_θ^{evo} adapts to the current policy $f_\varphi^{\text{policy}}$ that designates proper computation. (2) The second part of the loss, $L_s^{\text{evo}}[\mathbb{I}, f^{\text{evo}}; \hat{M}''^t]$, is the evolution loss by following an identity policy network $\mathbb{I}$ that does not change the mesh, and evolution model f^{evo}, starting at initial state of fine-grained mesh $\hat{M}''^t$ and evolve for s steps. This encourages the evolution model to learn to utilize more computation to achieve a better prediction error.

Learning the policy. In this phase, the policy network $f_\varphi^{\text{policy}}$ is learned to select update the spatial resolution (refinement or coarsening of mesh) at each location, to improve both the *computation* and the prediction *error*. Since the spatial refinement and coarsening are both discrete action, and the

metric of computation is typically non-differentiable, we use Reinforcement Learning (RL) to learn the policy. Specifically, we model it as a Markov Decision Process (MDP), where the environment state is the system's state M^t, the actions are the local refinement or coarsening at each edge of E^t, and we design the reward as the *improvement* on both the error and computation, between following the current policy's action, and an *counterfactual* scenario where the agent follows an identity policy $\mathbb{I}$ that does not update the mesh topology, starting on the initial state $\hat{M}^t$. Concretely, the reward is given by

$$r^t = (1-\beta) \cdot \Delta\text{Error} + \beta \cdot \Delta\text{Computation} \tag{11}$$

where

$$\Delta\text{Error} = L_S^{\text{evo}}[\mathbb{I}, f^{\text{evo}}; \hat{M}^t] - L_S^{\text{evo}}[f^{\text{policy}}, f^{\text{evo}}; \hat{M}^t] \tag{12}$$

$$\Delta\text{Computation} = \mathcal{C}_S[\mathbb{I}, f^{\text{evo}}; \hat{M}^t] - \mathcal{C}_S[f^{\text{policy}}, f^{\text{evo}}; \hat{M}^t] \tag{13}$$

Here $\mathcal{C}_S[\cdot]$ is a surrogate metric that quantifies "Computation" based on the predicted mesh topology $\hat{E}^{t+1}, \hat{E}^{t+2}, ...\hat{E}^{t+S}$ up to S steps into the future. In this paper we use the number of nodes as the surrogate metric for measuring the computation, since typically for the GNNs, the computation (in terms of FLOPs) scales linearly with the number of nodes (since each node has a bounded number of edges on the mesh, the number of edges thus scales linearly with number of nodes, so will message passing and node updates).

To optimize the reward r^t, we employ the standard REINFORCE as used in (Sutton et al., 1999; Hafner et al., 2021) to update the policy network $f_\varphi^{\text{policy}}$, with the following objective:

$$L_\beta^{\text{actor}} = \mathbb{E}_t \left[-\log p_\varphi(a^t|M^t, \beta) \text{sg}(r^t - f_\varphi^{\text{value}}(M^t, \beta)) - \eta \cdot \text{H}[p_\varphi(a^t|M^t, \beta)] \right] \tag{14}$$

Here $\text{H}[\cdot]$ is the entropy, which encourages the action to have higher entropy to increase exploration, where η here is a hyperparameter. The $\text{sg}(\cdot)$ is stop-gradient. Essentially, the first term in loss L^{policy} encourages to increase the log-probability of actions that have a higher "advantage", where the advantage is defined as the difference between the current reward r^t that follows the current action a^t taken, and the expected reward (value) $f_\varphi^{\text{value}}(M^t)$ that follows the current policy starting from the current state M^t. We can also think of it as an actor-critic where the critic tries to evaluate accurately the expected reward of the current policy, and an actor (policy) is trying to exceed that expectation. To train the value network, we use MSE loss:

$$L_\beta^{\text{value}} = \mathbb{E}_t \left[(f_\varphi^{\text{value}}(M^t, \beta) - r^t)^2 \right] \tag{15}$$

Learning to adapt to varying β. In the overall objective (Eq. 8), the β stipulates the relative importance between Error and Computation. $\beta = 0$ means we only focus on minimizing Error, without constraint on Computation. $\beta = 1$ means we only focus on minimizing computation, without considering the evolution error. In practice, we typically wish to improve both, with a $\beta \in (0, 1)$ that puts more emphasis on one metric but still considers the other metric. To allow LAMP to be able to operate at varying β at inference time, during the learning of policy, we sample β uniformly within a range $\mathcal{B} \subseteq [0, 1]$ (e.g. $\mathcal{B}$ can be $[0, 1]$ or $[0, 0.5]$), for different examples within a minibatch, and also train the policy and value network jointly, where the total loss L^{policy} is the weighted sum of the two policy and value losses:

$$L^{\text{policy}} = \mathbb{E}_{\beta \sim \mathcal{B}}[L_\beta^{\text{policy}} + \alpha^{\text{value}} \cdot L_\beta^{\text{value}}] \tag{16}$$

where α^{value} is a hyperparameter, which we set as 0.5. In this way, the policy can learn a generic way of spatial coarsening and refinement, conditioned on β. For example, for smaller β that focuses more on improving error, the policy network may learn to refine more on dynamic regions and coarsen less, sacrificing computation to improve prediction error.

4 EXPERIMENTS

In the experiments, we set out to answer the following questions on our proposed LAMP:

- Can LAMP learn to coarsen and refine the mesh, focusing more computation on the more dynamic regions to improve prediction accuracy?
- Can LAMP improve the Pareto frontier of Error vs. Computation, compared to state-of-the-art deep learning surrogate models?
- Can LAMP learn to condition on the β to change is behavior, and perform varying amount of refinement and coarsening depending on the β?

We evaluate our LAMP on two challenging datasets: (1) a 1D benchmark nonlinear PDEs , which tests generalization of PDEs in the same family (Brandstetter et al., 2022); (2) a mesh-based paper simulation generated by the ArcSim solver (Narain et al., 2012). Both datasets possess multi-scale characteristics where some parts of the system is highly dynamic, while other parts are changing more slowly.

4.1 1D NONLINEAR FAMILY OF PDES

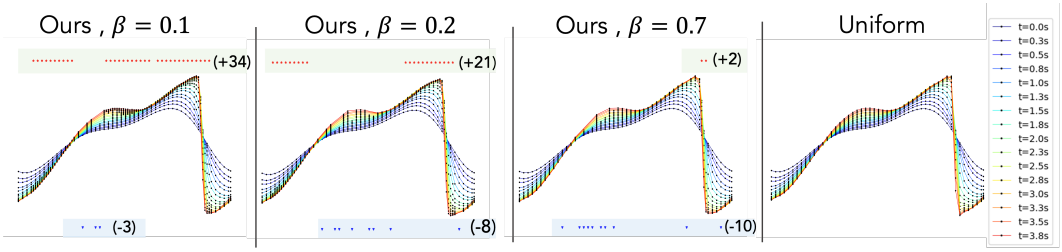

Figure 2: Example rollout result of our LAMP on 1D nonlinear PDEs. The rollout is performed over 200 time steps, where different color denotes the system's state at different time. On each state, we also plot the corresponding mesh as black dots. The upper green and lower blue band shows the added and removed nodes of the mesh, compared with the end state and initial state. We see that with a smaller β (e.g. $\beta = 0.1$) that emphasizes more on "Error", it refines more on highly-dynamic regions and coarsen less. With a larger β (e.g. $\beta = 0.7$) that focuses more on reducing computation, it almost doesn't refine, and choose to coarsen on more static regions.

Data and Experiments. In this section, we test LAMP's ability to balance error vs. computation tested on unseen equations with different parameters in a given family. We use the 1D benchmark in Brandstetter et al. (2022), whose PDEs are given by

$$\left[\partial_t u + \partial_x(\alpha u^2 - \beta \partial_x u + \gamma \partial_{xx} u)\right](t,x) = \delta(t,x) \tag{17}$$

$$u(0,x) = \delta(0,x), \quad \delta(t,x) = \sum_{j=1}^{J} A_j \sin(\omega_j t + 2\pi \ell_j x/L + \phi_j) \tag{18}$$

The parameter for the PDE is given by $p = (\alpha, \beta, \gamma)$. The term δ is a forcing term (Bar-Sinai et al., 2019) with $J = 5, L = 16$ and coefficients A_j and ω_j sampled uniformly from $A_j \sim U[-0.5, 0.5]$, $\omega_j \sim U[-0.4, 0.4]$, $\ell_j \in \{1, 2, 3\}$, $\phi_j \sim U[0, 2\pi)$. We uniformly discretize the space to $n_x = 200$ in $[0, 16)$ and uniformly discretize time to $n_t = 250$ points in $[0, 4]$. Space and time are further downsampled to resolutions of $(n_t, n_x) \in \{(250, 100), (250, 50), (250, 40), (250, 20)\}$ as initial resolution. We use the **E2** scenario in the benchmark, which tests the model's ability to generalize to novel parameters of PDE with the same family. Specifically, we have that the parameter $p = (1, \eta, 0)$ where $\eta \in [0, 0.2]$.

As our LAMP autoregressively simulate the system, it can refine or coarsen the mesh at appropriate locations by the policy network, before evolving to the next state with the evolution model. We evaluate the models with the metric of Computation and long-term evolution Error. For the computation, we use the average number of nodes throughout the full trajectory as a surrogate metric, since the number of floating point operations typically scales linearly with the number of nodes in the mesh. For long-term evolution error, we use the cumulative MSE over 200 steps of rollout, starting with initial state from time steps 25 to 49. We compare LAMP with strong baselines of deep learning-based surrogate models, including CNNs, Fourier Neural Operators (FNO) (Li et al., 2021) which is a state-of-the-art deep-learning based surrogate model. Our base neural architecture is based on MeshGraphNet (Pfaff et al., 2021) which is one state-of-the-art GNN-based model.

Table 1: Computation vs. Error for 1D family of nonlinear PDEs, for a total trajectory of 200 time steps, where we provide the first 25 steps as input, and predict the rest steps.

Model	Initial # vertices	Average # vertices	Error (MSE)
CNN	33	33.0	2.66
FNO	33	33.0	2.83
LAMP (no remeshing)	33	33.0	2.17
LAMP (ours)	33	37.6	**1.05**
CNN	50	50.0	1.10
FNO	50	50.0	1.79
LAMP (no remeshing)	50	50.0	1.07
LAMP (ours)	50	53.2	**0.75**
CNN	100	100.0	0.81
FNO	100	100.0	1.39
LAMP (no remeshing)	100	100.0	**0.75**
LAMP (ours)	100	100.0	0.76

We compare an ablation of our model that does not perform remeshing (LAMP no remeshing), and a full version of our model. For all models, we autoregressively roll out to predict the states for a full trajectory length of 200, using the first 25 steps as initial steps. We perform three groups of experiments, starting at initial number vertices of 33, 50 and 100 that is downsampled from the 100-vertice mesh, respectively. The three baselines all do not perform remeshing, and our full model has the ability to perform remeshing that coarsen or refine the edges at each time step. We record the accumulated MSE as measure for error, and average number of vertices over the full rollout trajectory as metric for computational cost. Note that for models, the MSE is computed on the full ground-truth mesh with 100 vertices, where the values of prediction are linearly interpolated onto the location of the ground-truth. This prevents the model to "cheat" by reducing the number of vertices and only predict well on those vertices. Additional details of the experiments are given in Appendix A.

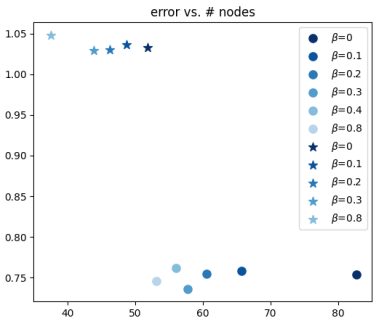

Figure 3: Pareto curve for 1D PDE experiments with varying β, for error (y-axis) vs. average number of vertices (x-axis).

Results. Table 1 shows the results. We see that our LAMP outperforms all baselines by a large margin, achieving an error reduction of 60.5%, 31.8% and 6.1% compared with the best performing baseline of CNN or FNO. In addition, we see that compared with an ablation with no remeshing, our full LAMP is able to significantly reduce error, with only slight increase of average number of vertices. This shows the ability of LAMP to adaptively tradeoff computation to improve long-term prediction error.

To investigate whether LAMP is able to focus computation on the most dynamic region, we visualize example trajectory of LAMP, as shown in Fig. 2. Starting with 50 vertices, we test our model on different β, where smaller β focuses more on improving error. We see that with smaller β (e.g. $\beta = 0.1$), LAMP is able to add more nodes (34 nodes) on the most dynamic region, and only coarsen few (removing 3 nodes in total). With a larger β that focuses more on reducing computation, we see that LAMP refines less and coarsen more, and only coarsen on the more slowly changing region. Additionally, we visualize the error (y-axis) vs. number of vertices for varying β for two different models, over all test trajectories. We see that with increasing β, LAMP is able to reduce the vertices more, with only slight increase of error. In summary, the above results show that LAMP is able to focus computation on dynamic regions, and able to adapt to different β at inference time.

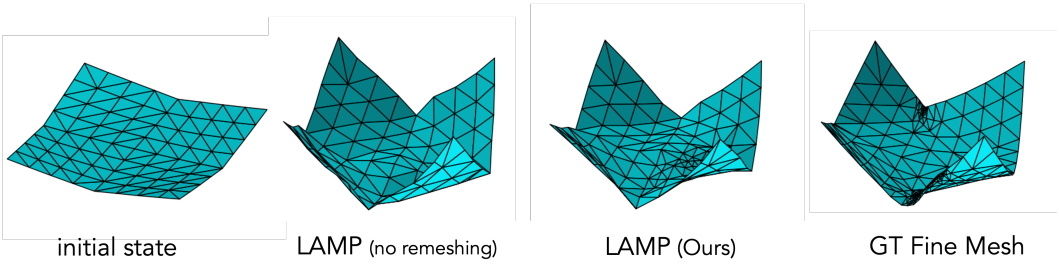

Figure 4: Example result of 2D mesh-based paper simulation. We observed that more resolution has been added to the high curvature region to resolve the details.

4.2 2D MESH-BASED SIMULATION

Here we evaluate our LAMP's ability on a more challenging setting with paper folding simulation. The paper in this simulation is square-shaped and its boundary edge length is 1. During the simulation, 4 corners of the paper will receive different magnitude of force. When generating ground-truth trajectories by the solver Narain et al. (2012), we set the minimum and maximal length of edges to be 0.01 and 0.2. We evaluate the models with the metric of Computation and long-term evolution error. Similar to the Section 4.1, we use the average number of nodes throughout the full trajectory as a surrogate metric for complexity. As shown in Table 2, our model is able to add more resolution to the high curvature region, and achieve better roll-out accuracy than the ablation without remeshing. Additional details of the experiments and 2D triangular mesh remeshing actions are given in Appendix A.2. In Fig. 4, we see that our LAMP learns to add more mesh onto the more dynamic regions near to the folding part (with high curvature), showing LAMP's ability to assign computation to the most needed region.

Model	Initial # vertices	Average # vertices	Error (MSE)
LAMP (no remeshing)	81.0	81.0	1.9e-4
LAMP (ours)	81.0	87.9	1.6e-4

Table 2: Computation vs. Error for 2D mesh-based paper simulation between LAMP (with remeshing) and LAMP (without remeshing). With the proposed learned remeshing framework, we are able to achieve better roll-out accuracy. Reported number is the MSE over 10 learned-simulator-steps roll-out averaged over 50 test trajectories.

5 CONCLUSION

In this work, we have introduced LAMP, the first fully deep learning-based surrogate model that jointly learns the evolution of physical system and optimizes assigning the computation to the most dynamic regions. In 1D and 2D datasets, we show that our method is able to adaptively perform refinement or coarsening actions, which improves long-term prediction error than strong baselines of deep learning-based surrogate models. We hope our method is provide a useful tool that can pave the way for more efficient simulation of physical systems.

REFERENCES

Michael Athanasopoulos, Hassan Ugail, and Gabriela González Castro. Parametric design of aircraft geometry using partial differential equations. *Advances in Engineering Software*, 40(7):479–486, 2009.

Yohai Bar-Sinai, Stephan Hoyer, Jason Hickey, and Michael P Brenner. Learning data-driven discretizations for partial differential equations. *Proceedings of the National Academy of Sciences*, 116(31):15344–15349, 2019.

Johannes Brandstetter, Daniel E. Worrall, and Max Welling. Message passing neural PDE solvers. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=vSix3HPYKSU.

Francesco Carpanese. Development of free-boundary equilibrium and transport solvers for simulation and real-time interpretation of tokamak experiments. Technical report, EPFL, 2021.

Jakub Cerveny, Veselin Dobrev, and Tzanio Kolev. Nonconforming mesh refinement for high-order finite elements. *SIAM Journal on Scientific Computing*, 41(4):C367–C392, 2019.

Richard Courant, Kurt Friedrichs, and Hans Lewy. On the partial difference equations of mathematical physics. *IBM journal of Research and Development*, 11(2):215–234, 1967.

Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

Danijar Hafner, Timothy P Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=0oabwyZbOu.

Julian Kates-Harbeck, Alexey Svyatkovskiy, and William Tang. Predicting disruptive instabilities in controlled fusion plasmas through deep learning. *Nature*, 568(7753):526–531, 2019.

Dmitrii Kochkov, Jamie A Smith, Ayya Alieva, Qing Wang, Michael P Brenner, and Stephan Hoyer. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118(21), 2021.

Tony Lelievre and Gabriel Stoltz. Partial differential equations and stochastic methods in molecular dynamics. *Acta Numerica*, 25:681–880, 2016.

Zongyi Li, Nikola Borislavov Kovachki, Kamyar Azizzadenesheli, Burigede liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=c8P9NQVtmnO.

Peter Lynch. The origins of computer weather prediction and climate modeling. *Journal of computational physics*, 227(7):3431–3444, 2008.

Sharanya J Majumdar, Juanzhen Sun, Brian Golding, Paul Joe, Jimy Dudhia, Olivier Caumont, Krushna Chandra Gouda, Peter Steinle, Béatrice Vincendon, Jianjie Wang, et al. Multiscale forecasting of high-impact weather: current status and future challenges. *Bulletin of the American Meteorological Society*, 102(3):E635–E659, 2021.

Rahul Narain, Armin Samii, and James F O'brien. Adaptive anisotropic remeshing for cloth simulation. *ACM transactions on graphics (TOG)*, 31(6):1–10, 2012.

Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W. Battaglia. Learning mesh-based simulation with graph networks. In *International Conference on Learning Representations*, 2021.

Alvaro Sanchez, Dmitrii Kochkov, Jamie Alexander Smith, Michael Brenner, Peter Battaglia, and Tobias Joachim Pfaff. Learning latent field dynamics of PDEs. *Advances in Neural Information Processing Systems*, 2020.

Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*, pp. 8459–8468. PMLR, 2020.

NJ Sircombe, TD Arber, and RO Dendy. Kinetic effects in laser-plasma coupling: Vlasov theory and computations. In *Journal de Physique IV (Proceedings)*, volume 133, pp. 277–281. EDP sciences, 2006.

Halil Mete Soner, Wolfgang Bangerth, Rolf Rannacher, Hans Foellmer, and LCG Rogers. *Adaptive finite element methods for differential equations*. Springer Science & Business Media, 2003.

Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

Kiwon Um, Robert Brand, Yun Raymond Fei, Philipp Holl, and Nils Thuerey. Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers. *Advances in Neural Information Processing Systems*, 33:6111–6122, 2020.

Pantelis R Vlachas, Georgios Arampatzis, Caroline Uhler, and Petros Koumoutsakos. Multiscale simulations of complex systems by learning their effective dynamics. *Nature Machine Intelligence*, 4(4):359–366, 2022.

Tailin Wu, Takashi Maruyama, and Jure Leskovec. Learning to accelerate partial differential equations via latent global evolution. *Advances in neural information processing systems*, 36, 2022.

Qingqing Zhao, David B. Lindell, and Gordon Wetzstein. Learning to solve pde-constrained inverse problems with graph networks. *International Conference on Machine Learning*, 2022.