A Neural Material Point Method for Particle-based Emulation

Omer Rochman-Sharabi *o.rochman@uliege.be*
University of Liège

Sacha Lewin *sacha.lewin@uliege.be*
University of Liège

Gilles Louppe *g.louppe@uliege.be*
University of Liège

Reviewed on OpenReview: *https://openreview.net/forum?id=zSK81A2hxQ*

Abstract

Mesh-free Lagrangian methods are widely used for simulating fluids, solids, and their complex interactions due to their ability to handle large deformations and topological changes. These physics simulators, however, require substantial computational resources for accurate simulations. To address these issues, deep learning emulators promise faster and scalable simulations, yet they often remain expensive and difficult to train, limiting their practical use. Inspired by the Material Point Method (MPM), we present NeuralMPM, a neural emulation framework for particle-based simulations. NeuralMPM interpolates Lagrangian particles onto a fixed-size grid, computes updates on grid nodes using image-to-image neural networks, and interpolates back to the particles. Similarly to MPM, NeuralMPM benefits from the regular voxelized representation to simplify the computation of the state dynamics, while avoiding the drawbacks of mesh-based Eulerian methods. We demonstrate the advantages of NeuralMPM on 6 datasets, including fluid dynamics and fluid-solid interactions simulated with MPM and Smoothed Particles Hydrodynamics (SPH). Compared to GNS and DMCF, NeuralMPM reduces training time from 10 days to 15 hours, memory consumption by 10x-100x, and increases inference speed by 5x-10x, while achieving comparable or superior long-term accuracy, making it a promising approach for practical forward and inverse problems. A project page is available at `https://neuralmpm.isach.be`.

1 Introduction

The Navier-Stokes equations describe the time evolution of fluids and their interactions with solid materials. As analytical solutions rarely exist, numerical methods are required to approximate the solutions. On the one hand, Eulerian methods discretize the fluid domain on a fixed grid, simplifying the computation of the dynamics, but requiring high-resolution meshes to solve small-scale details in the flow. Lagrangian methods, on the other hand, represent the fluid as virtual moving particles defining the system's state, hence maintaining a high level of detail in regions of high density. While effective at handling deformations and topological changes (Monaghan, 2012), Lagrangian methods struggle with collisions and interactions with rigid objects (Vacondio et al., 2021; Lind et al., 2020).

Regardless of the discretization strategy, large-scale high-resolution numerical simulations are computationally expensive, limiting their practical use in downstream tasks such as forecasting, inverse problems, or computational design. To address these issues, deep learning emulators have shown promise in accelerating

simulations by learning an emulator model that can predict the system's state at a fraction of the cost. Next to their speed, neural emulators also have the strategic advantage of being differentiable, enabling their use in inverse problems and optimization tasks (Allen et al., 2022; Zhao et al., 2022; Forte et al., 2022). Moreover, they have the potential to be learned directly from real data, bypassing the costly and resource-intensive process of building a simulator (Lam et al., 2023; Pfaff et al., 2021; Lemos et al., 2023; Jumper et al., 2021; He et al., 2019). In this direction, particle-based neural emulators (Sanchez-Gonzalez et al., 2020; Ummenhofer et al., 2020; Prantl et al., 2022) have seen success in accurately simulating fluids and generalizing to unseen environments. These emulators, however, suffer from the same issues as traditional Lagrangian methods, with collisions and interactions with rigid objects being particularly challenging. These emulators can also require long training and inference times, limiting their practical use.

Taking inspiration from the hybrid Material Point Method (MPM) (Sulsky et al., 1993; Nguyen et al., 2023) that combines the strengths of both Eulerian and Lagrangian methods, we introduce NeuralMPM, a neural emulation framework for particle-based simulations.

- As in MPM, NeuralMPM uses Lagrangian particles to represent the system's state but models system dynamics on voxelized grids. This approach benefits from a regular grid structure for simplified state dynamics computation while avoiding the drawbacks of mesh-based Eulerian methods.
- It interpolates particles onto a fixed-size grid, bypassing the need for expensive neighbor searches at each timestep and replacing them with two efficient voxelization-based interpolation steps (Xu et al., 2021)
- By defining dynamics on a grid, NeuralMPM leverages well-established grid-to-grid neural architectures. This introduces an inductive bias that enables easier processing of global and local point cloud structures, freeing capacity to learn the system's dynamics.

Compared to previous data-driven approaches (Sanchez-Gonzalez et al., 2020; Ummenhofer et al., 2020; Prantl et al., 2022), these improvements reduce the training time from days to hours, while achieving higher or comparable accuracy.

2 Computational Fluid Dynamics

Computational fluid dynamics simulations can be classified into two broad categories, Eulerian and Lagrangian, depending on the discretization of the fluid (Rakhsha et al., 2021). In Eulerian simulations, the domain is discretized with a mesh, with state variables u_i^t (such as mass or momentum) maintained at each mesh point i. Well-known examples of Eulerian simulations are the finite difference method, where the domain is divided into a uniform regular grid (also called an Eulerian grid), and the finite element method, where the domain is divided into regions, or elements, that may have different shapes and density, allowing to increase the resolution in only some areas of the domain (Iserles, 2008; Morton & Mayers, 2005). Another widely used family of Eulerian methods is the spectral and pseudo-spectral family of methods, where the equation is solved in Fourier space or alternating between Fourier and real space, respectively, leveraging the fact that differentiation in real space is multiplication in Fourier space Pope (2000). Lagrangian simulations, on the other hand, discretize the fluid as a set of virtual moving particles $\{p_i^t, u_i^t\}_{i=1}^N$, each described by its position p_i^t and state variables u_i^t that include the particle velocity v_i^t. To simulate the fluid, the particles move according to the dynamics of the system, producing a new set of particles $\{p_i^{t+1}, u_i^{t+1}\}_{i=1}^N$ at each timestep. Simulations in Lagrangian coordinates are particularly useful when the fluid is highly deformable, as the particles can move freely and adapt to the fluid's shape. Among Lagrangian methods, Smoothed Particle Hydrodynamics (SPH) (Gingold & Monaghan, 1977; Price, 2012) is one of the most popular, where the fluid is represented by a set of particles that interact with each other through a kernel function that smooths the interactions.

Hybrid Eulerian-Lagrangian methods combine the strengths of both frameworks. Like Lagrangian methods, they carry the system state information via particles, thereby automatically adjusting the resolution to the local density of the system. By using a regular grid, however, they simplify gradient computation, make entity

contact detection easier, and prevent cracks from propagating only along the mesh. Among hybrid methods, the Material Point Method has gained popularity for its ability to handle large deformations and topological changes. MPM combines a regular Eulerian grid with moving Lagrangian particles. It does so in four main steps: (1) the quantities carried by the particles are interpolated onto a regular grid $G^t = \mathbf{p2g}(\{p_i^t, u_i^t\})$ using a particle-to-grid ($\mathbf{p2g}$) function, (2) the equations of motion are solved on the grid, where derivatives and other quantities are easier to compute, resulting in a new grid state $G^{t+1} = f(G^t)$, (3) the resulting dynamics are interpolated back onto the particles as $\{u_i^{t+1}\} = \mathbf{g2p}(G^{t+1}, \{p_i^t\})$, using a grid-to-particle ($\mathbf{g2p}$) function, (4) the positions of the particles are updated by computing particle-wise velocities and using an appropriate integrator, such as Euler, i.e., $p_i^{t+1} = p_i^t + \Delta t v_i^{t+1}$. The grid values are then reset for the next step. MPM has been used in soft tissue simulations (Ionescu et al., 2005), in molecular dynamics (Lu et al., 2006), in astrophysics (Li & Liu, 2002), in fluid-membrane interactions (York II et al., 2000), and in simulating cracks (Daphalapurkar et al., 2007) and landslides (Llano Serna et al., 2015). MPM is also widely used in the animation industry, perhaps most notably in Disney's 2013 film Frozen (Stomakhin et al., 2013), where it was used to simulate snow.

Notwithstanding the success of numerical simulators, they remain expensive, slow, and, most of the time, non-differentiable. In recent years, differentiable neural emulators have shown great promise in accelerating fluid simulations, most notably in a series of works to emulate SPH simulations in a fully data-driven manner. Graph network-based simulators (GNS) (Sanchez-Gonzalez et al., 2020) use a graph neural network (GNN) and a graph built from the local neighborhood of the particles to predict the acceleration of the system. The approach requires building a graph out of the point cloud at every timestep to obtain structural information about the cloud, which is an expensive operation. In addition, the GNN needs to extract global information from its nodes, which is only possible with a high number of message-passing steps, resulting in a large computational graph and long training and inference times. This large computational graph, along with repeated construction, makes fully autoregressive training over long rollouts impractical, as the gradients need to backpropagate all the way back to the first step. Cheaper strategies exist, like the push-forward trick (Brandstetter et al., 2022b), but they have been shown to be inferior to fully backpropagating through trajectories (List et al., 2024; Sharabi & Louppe, 2023). As autoregressive training is not available, the stability of the learned dynamics can be compromised, making the model prone to diverging or oscillating. Noise injection training strategies can be used to increase the stability of the rollouts, but the magnitude of the noise becomes a critical parameter. Related approaches to GNS include Han et al. (2022), who introduce improvements to GNS to make them subequivariant to certain transformations and show increased accuracy on simulations involving solid objects, and Viswanath et al. (2024), who apply GNS on a reduced representation of the particle system obtained by farthest point sampling.

An alternative approach is the continuous convolution (CConv) (Ummenhofer et al., 2020; Winchenbach & Thuerey, 2024), an extension of convolutional networks to point clouds. In this method, a convolutional kernel is applied to each particle by interpolating the values of the kernel at the positions of its neighbors, which are found via spatial hashing on GPU, a cheaper alternative to tree-based searches that allows for autoregressive training. In (Prantl et al., 2022), Deep Momentum Conserving Fluids (DMCF) build upon CConv to design a momentum-conserving architecture. Nevertheless, to account for long-range interactions, the authors introduce different branches, with different receptive fields, into their network. The number of branches, and their hyperparameters, need to be tuned to capture global dependencies, leading to long training times even with optimized CUDA kernels. Finally, Zhang et al. (2020) propose an approach that uses nearest neighbors to construct the local features of each particle. Those local features are then averaged onto a regular grid. Like GNS, this method suffers from the need to repeat the neighbor search at every simulation timestep. Ultimately, the performance of point cloud-based simulators is tightly linked to the method used to process the spatial structure of the cloud. Brute force neighbour search is $\mathcal{O}(N^2)$, K-d trees are $\mathcal{O}(N \log N)$, and voxelization and hashing are $\mathcal{O}(N)$ (Xu et al., 2021; Hastings & Mesit, 2005). Recently, Alkin et al. (2024), attempt to bridge the gap between Lagrangian and Eulerian representations by avoiding explicit grid- or particle-based latent structures, but necessitating extra networks to account for particle movement or empty volumes.

An approach different from data-driven modeling is hybrid models, where parts of a classical solver are replaced with learned components. For instance, Yin et al. (2021) employ a neural network to learn unknown

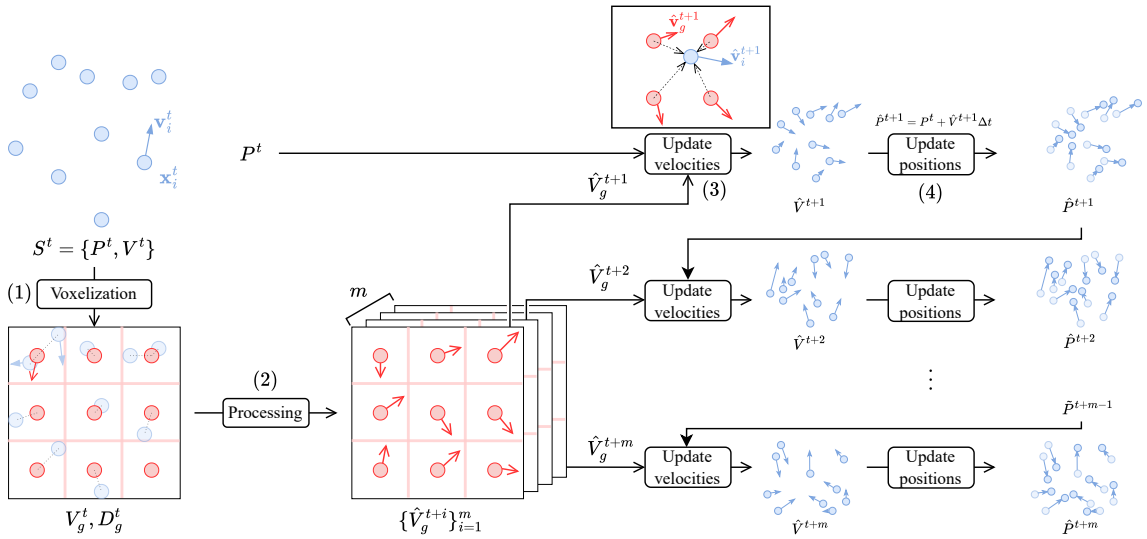

Figure 1: NeuralMPM works in 4 steps. (1) The positions P^t and velocities V^t of the particles are used to compute the velocity V_g^t and density D_g^t of each grid node through voxelization. (2) From this grid, the processor neural network predicts the grid velocities at the next m timesteps. The next m positions are computed iteratively by (3) performing bilinear interpolation of the predicted velocities onto the previous positions and (4) updating the positions using the predicted velocities.

physics, which is then integrated into a simulator. Similarly, Li et al. (2024a) use a neural network to bypass computational bottlenecks in MPM simulators, while Ma et al. (2023) learn general constitutive laws, allowing for one-shot trajectory learning. These approaches achieve impressive results by leveraging extensive physics knowledge, but this reliance also limits their applicability. Hybrid models may inherit both the strengths and weaknesses of classical and ML methods. Our approach falls within the data-driven domain, as the prior constraints on the learnable dynamics induced by the MPM-like algorithm are minimal and can be overcome with data (Table 1).

3 NeuralMPM

We consider a Lagrangian system evolving in time and defined by the positions p_i^t and velocities v_i^t of a set of N particles $i = 1, ..., N$. We denote with P^t and V^t the set of positions and velocities of all particles at time t and with $S^t = (P^t, V^t)$ the full state of the system. In a more complex setting, the state of the system can include other local properties, such as pressure or elastic stiffness of materials, and global properties, such as an external force. In this work, for simplicity, we let the network learn the relevant simulation parameters implicitly. The evolution of the particles is described by a function f mapping the current state of the system to its next state $S^{t+1} = f(S^t)$. Given a starting system $S^0 = (P^0, V^0)$, its full trajectory, or rollout, is denoted by $S^{1:T}$. Our goal is to build an emulator $\hat{f}_\theta(\cdot)$ capable of predicting a full rollout $\hat{f}_\theta^{1:T}(S^0)$ of T timesteps from the initial state S^0. Following MPM, NeuralMPM operates in four steps, described below, as illustrated in Figure 1 and in Alg 1.

Step 1: Voxelization. Using the particle positions P^t, the velocities V^t are interpolated onto a regular fixed-size grid. This interpolation is performed through *voxelization*, which divides the domain into regular volumes (voxels). Each grid node is identified as the center of a voxel (e.g., square in 2D) in the domain, and the velocities of the particles in the voxel are averaged to give the node's velocity. Similarly, the density is computed as the normalized number of particles in the voxel. This results in the grid tensor G^t that contains the grid velocities V_g^t and density D_g^t.

Step 2: Processing. Taking advantage of the regular grid representation of the cloud, the grid velocities $\{\hat{V}_g^i\}_{i=t+1}^{t+m}$ of the next m timesteps are predicted using a neural network. We chose a U-Net (Ronneberger et al., 2015) as it is a well-established image-to-image model, known to perform well in various tasks, including physical applications. The combination of kernels applied with different receptive fields (from smaller to larger) can allow the U-Net to efficiently extract both local and global information. Nonetheless, any grid-to-grid architecture could be used. We experiment with the FNO (Li et al., 2021) architecture in Appendix D and find it to underperform, leading us to keep the U-Net. A fully convolutional U-Net and an FNO have the additional advantage of being able to generalize to different domain shapes, a desirable property (Section 4.3).

Step 3: Update of particle velocities. The predicted velocities $\hat{V}^{t+1}$ at the next timestep are then interpolated back to the particle level onto the positions P^t using bilinear interpolation. The velocity of each particle is computed as a weighted average of the four surrounding grid velocities, based on its Euclidean distance to each of them.

Step 4: Update of particle positions. Finally, the positions of the particles are updated with Euler integration using the next velocities and known current positions of the particles, that is $\hat{P}^{t+1} = P^t + \Delta t \hat{V}^{t+1}$. Steps 3 and 4 are performed m times to compute the next m positions from the set of grid velocities computed at step 2.

Additional features of the individual particles can be included in the grid tensor G^t by interpolating them in the same way as the velocities. Local, such as boundary conditions, or global, such as gravity or external forces, features are represented as grid channels. For simulations with multiple types of particles, the features of each material are interpolated independently and stacked as channels in G_t.

NeuralMPM is trained end-to-end on a set of trajectories $S^{0:T}$ to minimize the mean squared error $||P^{t+1} - \hat{P}_\theta^{t+1}(S^t)||_2^2$ between the ground-truth and predicted next positions of the particles. At inference time, the model is exposed to much longer sequences, which requires carefully stabilizing the rollout procedure to prevent the accumulation of large errors over time. To address this, we first make use of autoregressive training (Prantl et al., 2022; Ummenhofer et al., 2020), where the model is unrolled K times on its own predictions, producing a sequence of $\hat{S}^k = \hat{f}_\theta(\hat{S}^{k-1})$ for $k = 1, ..., K$ and initial input $\hat{S}^0 = S^0$, before backpropagating the error through the entire rollout. Unlike more costly methods that require alternative stabilization strategies, such as noise injection (Sanchez-Gonzalez et al., 2020), NeuralMPM's efficiency makes autoregressive training possible. Nevertheless, to further stabilize the training, we couple autoregressive training with time bundling (Brandstetter et al., 2022b), resulting in a training strategy where the model predicts m steps $\hat{S}^{1:m}$ at once from a single initial state, inside an outer autoregressive loop of K steps of length m. We show in Section 4 that this training strategy leads to more accurate rollouts.

Algorithm 1 NeuralMPM

Require: P^0, V^0, grid g, t, neural backbone h, functions **p2g** and **g2p**

$\hat{P}^t \leftarrow P^t$, $\hat{V}^t \leftarrow V^t$
while $t < T$ **do**
  Voxelization: Interpolate density and velocities onto the grid using $G^t = (D_g^t, V_g^t) = \mathbf{p2g}(\hat{P}^t, \hat{V}^t)$
  Processing: $\{\hat{V}_g^i\}_{i=t+1}^{t+m} = h(G^t)$
  for i in range$(1, m)$ **do**
    Update velocities: $\hat{V}^{t+i} = \mathbf{g2p}(\hat{V}_g^{t+i}, \hat{P}^{t+i-1})$
    Update positions: $\hat{P}^{t+i} = P^{t+i-1} + \Delta t \hat{V}^{t+i}$
  end for
  $t \leftarrow t + m$
end while
return Full trajectory $\{\hat{P}, \hat{V}\}_t^T$

4 Experiments

We conduct a series of experiments to demonstrate the accuracy, speed, and generalization capabilities of NeuralMPM. Specifically, we examine its robustness to hyperparameter and architectural choices through an ablation study (4.1). We compare NeuralMPM to GNS and DMCF in terms of accuracy, training time, convergence, and inference speed (4.2). We also evaluate the generalization capabilities of NeuralMPM (4.3) and illustrate how its differentiability can be leveraged to solve an inverse design problem (4.4). Through these experiments, we demonstrate that NeuralMPM is a flexible, accurate, and fast method for emulating complex particle-based simulations. The baselines established by Winchenbach & Thuerey (2024) and hybrid simulators (Li et al., 2024a; Ma et al., 2023) have promising results. However, we do not compare against them as they either use different benchmarks or are specifically tailored for certain physical domains, requiring material-specific knowledge. In contrast, NeuralMPM, like GNS and DMCF, requires only particle positions without being restricted to any particular domain. Rollouts for all experiments, models, and datasets are available at drive and in the supplementary material.

Data. We consider 6 datasets with variable sequence lengths, numbers of particles, and materials. The first three datasets, WaterRamps, SandRamps, and Goop, contain a single material, water, sand, and goop, respectively, with different material properties. The first two datasets contain random ramp obstacles to challenge the model's generalization capacity. The fourth dataset, MultiMaterial, mixes the three materials together in the same simulations. These four datasets are taken from Sanchez-Gonzalez et al. (2020) and were simulated using the Taichi-MPM simulator (Hu et al., 2018b). They each contain 1000 trajectories for training and 30 (Goop) or 100 (WaterRamps, SandRamps, MultiMaterial) for validation and testing. The fifth dataset, Dam Break 2D, was generated using SPH and contains 50 trajectories for learning, and 25 for validation and testing. The last dataset, VariableGravity, was also generated using Taichi-MPM. It consists of simulations with variable gravity of a water-like material, and contains 1000 trajectories for training and 100 for validation and testing.

Protocol. NeuralMPM is trained on trajectories with varying initial conditions and number of particles. The training batches are sampled randomly in time and across sequences. We use Adam (Kingma & Ba, 2014) with the following learning rate schedule: a linear warm-up over 100 steps from 10^{-5} to 10^{-3}, 900 steps at 10^{-3}, then a cosine annealing (Loshchilov & Hutter, 2017) for $100,000$ iterations. We use a batch size of 128, $K = 4$ autoregressive steps per iteration, bundle $m = 8$ timesteps per model call (resulting in 24 predicted states), and a grid size of 64×64. For most of our experiments, we use a U-Net (Ronneberger et al., 2015) with three downsampling blocks with a factor of 2, 64 hidden channels, a kernel size of 3, and MLPs with three hidden layers of size 64 for pixel-wise encoding and decoding into a latent space. For a fair comparison, we ran training and inference for NeuralMPM, DMCF, and GNS on the exact same hardware. GNS and DMCF were trained until convergence (a maximum of 120 and 240 hours, respectively), while NeuralMPM required 20 hours or less to converge. For WaterRamps, Sandramps, Goop, and MultiMaterial, we use the same parameters as those reported by authors. We hyperparameter search DMCF for Dam Break 2D and both GNS and DMCF for VariableGravity and report the best performance obtained for a budget of 60 GPU-days per dataset. Further details on training can be found in Appendix B.

4.1 Ablation study

To study the robustness of NeuralMPM to hyperparameter and architectural choices, we start with the default architecture and hyperparameters and ablate its components individually to examine their impact on performance. We vary the number K of autoregressive steps with and without grid and particle noise in the input state the number of bundled timesteps m predicted by a single model call, and the depth and number of hidden channels of the network. We also investigate adding noise to stabilize rollouts, either directly to the particles' positions or to the grid-level representation after voxelization.

Figure 3 summarizes the ablation results. A larger number K of autoregressive steps yields more accurate rollouts without the need to add noise. Indeed, injecting noise does not improve accuracy and is even detrimental for $K = 4$. Individually tuning the noise levels for grids and particles can modestly lower error rates, but is either very sensitive or negligible. The model performs better when bundling more timesteps,

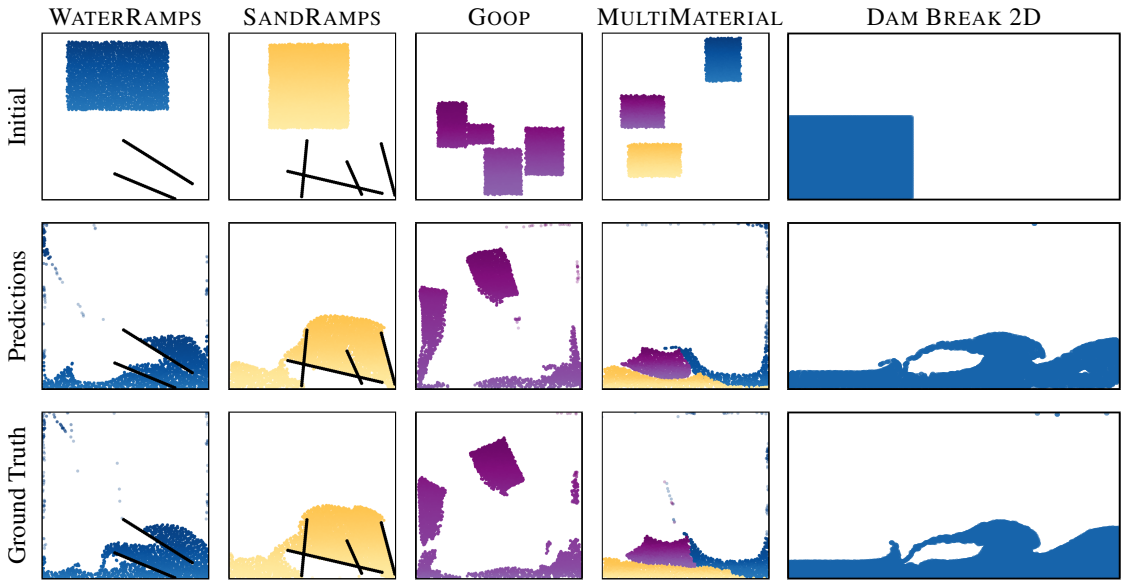

Figure 2: **Example snapshots.** We train and evaluate NeuralMPM on WaterRamps, SandRamps and Goop, each consisting of a single material, on MultiMaterial that mixes water, sand and goop, and on Dam Break 2D, a rectangular-shaped SPH dataset. NeuralMPM is able to learn various kinds of materials, their interactions, and their interactions with solid obstacles. Despite being inspired by MPM, it is not limited to data showing MPM-like behaviour.

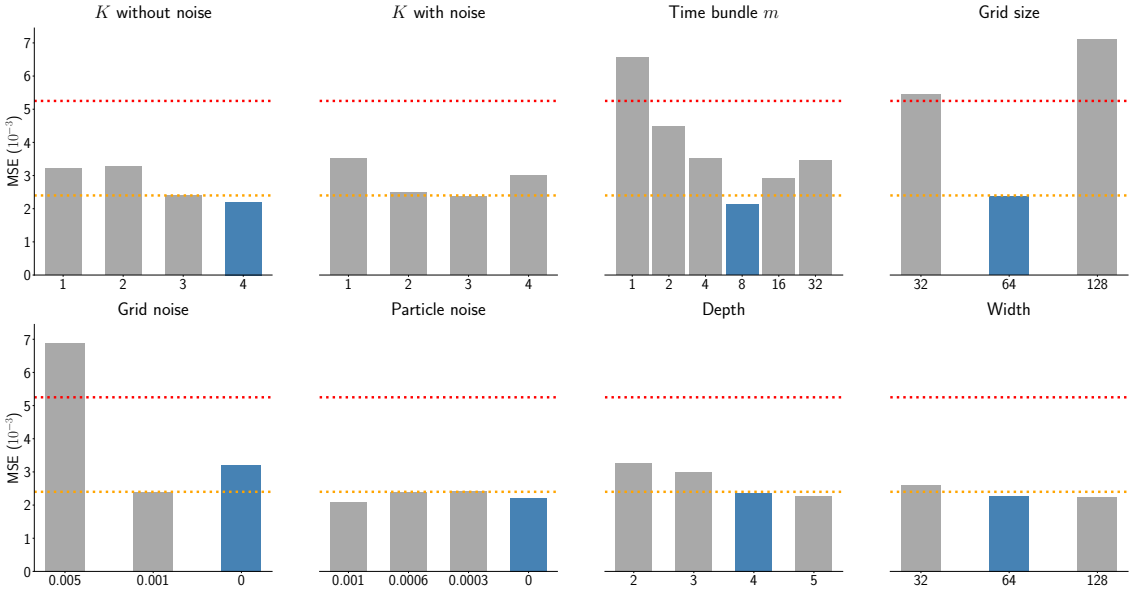

Figure 3: **Ablation results.** Mean squared error of full rollouts on unseen test data for Goop. The default parameters are in blue. The dotted orange line (2.4×10^{-3}) indicates the MSE we obtained for GNS after 240 hours (20M training steps). The dotted red line is the MSE for DMCF after the same amount of time (5.25×10^{-3}). NeuralMPM is robust to hyperparameter changes, with the biggest effects coming from the number of timesteps bundled together (m) and grid noise. For a rollout of length T, the model is called T/m times, meaning lower values of m require maintaining stability for longer. Autoregressive training coupled with time bundling suffices to stabilize the model, eliminating the need for noise injection. Although GNS reportedly slightly outperforms NeuralMPM, these results could not be reproduced in our experiments.

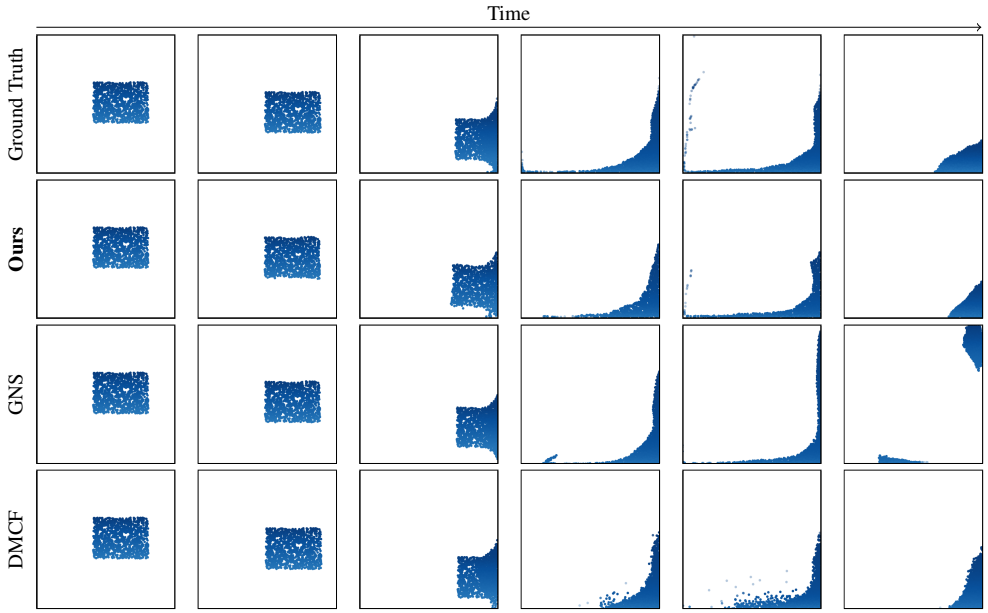

Figure 4: **Example VariableGravity trajectory against baselines.** Each method is unrolled starting from the initial conditions of a random test trajectory not seen during training.

enabling faster rollouts as a single forward pass predicts more steps. We found $m = 8$ to be optimal with the other default hyperparameters, outperforming larger bundling. This is because more network capacity is needed to extract information for the next 16 or 32 timesteps from a single state. Instead, we opted for a shallower and narrower network to balance speed and memory footprint with performance gains. In terms of network architecture, we chose a U-Net. We experiment with an FNO (Li et al., 2021) in Appendix D and find it to underperform, leading us to keep the U-Net architecture. We find the U-Net's width and depth to have a minor impact on performance, confirming that a larger network is not needed. The grid size, however, is critical. A low resolution loses fine details, while a high resolution turns meaningful structures, such as liquid blobs or walls, into isolated voxels.

4.2 Comparison with previous work

We compare NeuralMPM against GNS and DMCF. We use the official implementations and training instructions to assess training times, inference times, as well as accuracy. We compare against both GNS and DMCF on WaterRamps, Sandramps, Goop, Dam Break 2D, and VariabelGravity. We also compare against GNS on MultiMaterial, but not against DMCF since it does not support multiple materials.

Accuracy. We report quantitative results comparing the long-term accuracy in Table 1 and show trajectories of NeuralMPM in Figure 2, as well as comparisons against baselines on WaterRamps in Figure 4. On the mono-material datasets WaterRamps, SandRamps, and Goop, NeuralMPM performs competitively with GNS and better than DMCF in terms of mean squared error (MSE). For MultiMaterial, NeuralMPM reduces the MSE by almost half, which we attribute to it being a hybrid method, known to better handle interactions, mixing, and collisions between different materials. In Dam Break 2D, NeuralMPM outperforms both baselines, despite the data being simulated using SPH. Finally, NeuralMPM surpasses the performance of DMCF in VariableGravity, even though the latter accounts for gravity explicitly. In terms of Earth Mover's Distance (EMD), NeuralMPM outperforms both baselines across all benchmarks, suggesting that NeuralMPM is better at capturing the spatial distribution of the particles.

Training. In Figure 5, we report the evolution of the mean squared error of full emulated rollouts on the held-out test set during training, for each method, along with predicted snapshots at increasing training

Data (Simulator)	N	T	NeuralMPM		GNS		DMCF	
			MSE↓	EMD↓	MSE↓	EMD↓	MSE↓	EMD↓
WaterRamps (MPM)	2.3k	600	13.92	**68**	**11.75**	90	20.45	105
SandRamps (MPM)	3.3k	400	3.12	**61**	**3.11**	84	6.22	91
Goop (MPM)	1.9k	400	**2.18**	**55**	2.4	73	5.25	85
MultiMaterial (MPM)	2k	1000	**9.6**	**66**	14.79	105	-	-
Dam Break 2D (SPH)	5k	401	**29.07**	**348**	87.04	384	74.77	381
VariableGravity (MPM)	600	1000	**14.48**	**92**	134	350	28.77	97

Table 1: Full rollout MSE & EMD (both $\times 10^{-3}$) for NeuralMPM and the baselines on each dataset, with the maximum number of particles N and sequence length T. Each method was trained until full convergence (NeuralMPM: 15h, GNS: 240h, DMCF: 120h), and the best model was used.

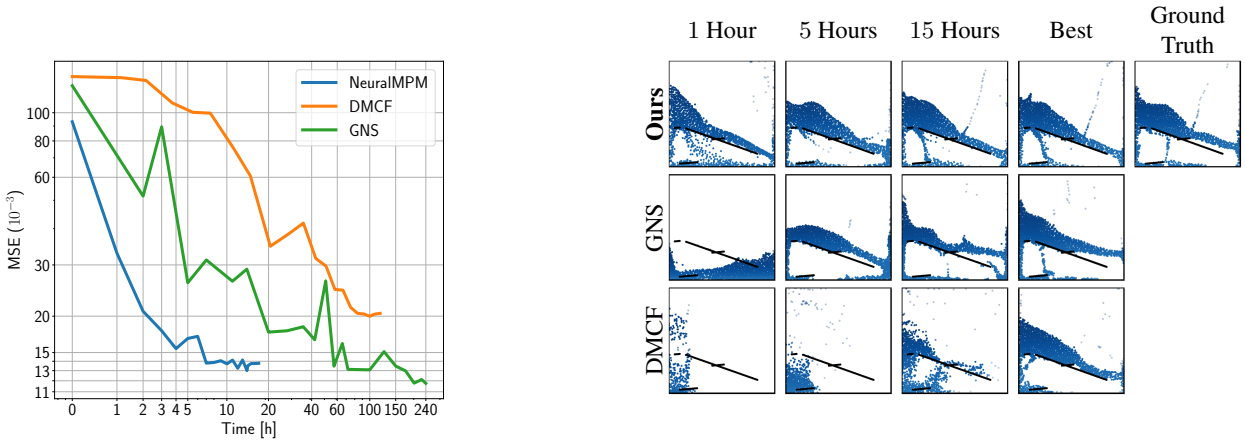

Figure 5: **Training convergence.** (Left) NeuralMPM trains and converges much faster than GNS and DMCF. Note the log scale on both axes. (Right) Snapshots of models trained for increasing durations then unrolled until the same timestep on a held-out simulation. For a fair comparison, out-of-bounds particles in GNS and DMCF were clamped.

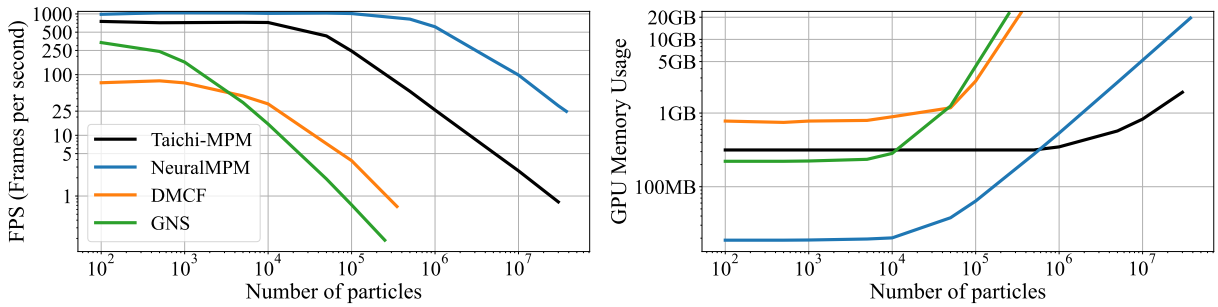

Figure 6: **Time and memory performance.** Average FPS (left) and GPU VRAM usage (right) for increasing numbers of particles for a traditional solver (Taichi-MPM (Hu et al., 2018a)), NeuralMPM, and the two baselines. The two baselines quickly require very large amounts of memory and become very slow. Although Taichi-MPM is more memory efficient for high numbers of particles, NeuralMPM remains much faster, emulating 30 million particles at 25FPS. For the low particle count regime ($<$ 10K) we used the NeuralMPM and baselines WaterRamps models. For the high particle count regime we used untrained models and measured the throughput. The figures measures just FPS, and not the real simulation time. Taichi-MPM needs a much smaller step size than the three neural emulators (0.2ms vs 2.5ms), and is therefore likely slower than all of them

durations. NeuralMPM converges significantly faster than both baselines while reaching lower error rates. Furthermore, the convergence of the training procedure and quality of the architecture can be assessed much earlier during training, effectively saving compute and enabling the development of more refined final models. Moreover, NeuralMPM is also more memory-efficient, which enables the use of higher batch sizes of 128, as opposed to only 2 in GNS and DMCF.

Inference time and memory. In Figure 6, we display the time and memory performance of NeuralMPM, the two baselines GNS and DMCF, and the reference solver Taichi-MPM. In terms of speed, NeuralMPM strongly outperforms all three methods, partly thanks to time bundling, which considerably reduces the number of model calls required for a given number of frames to emulate. In terms of memory, although NeuralMPM remains inferior to Taichi-MPM, which is highly optimized, it can emulate tens of millions of particles on a single GPU, while GNS and DMCF struggle to reach half a million.

4.3 Generalization

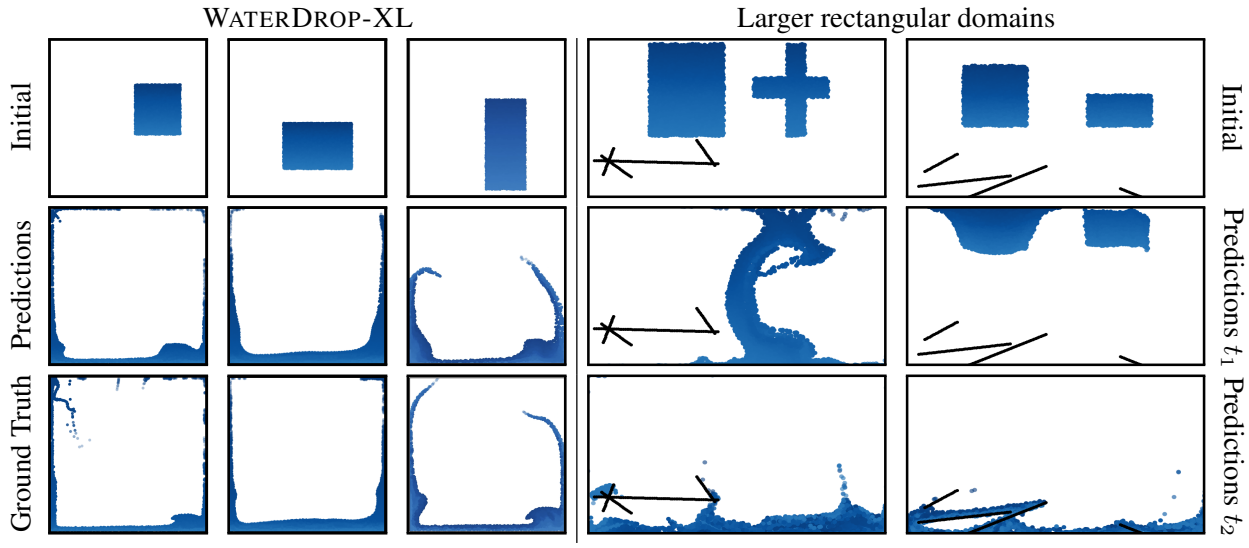

Figure 7: **Generalization.** (Left) NeuralMPM generalizes to domains with more particles ($\sim 4\times$ here) with minimal inference time overhead due to the processing of the voxelized representation. (Right) A NeuralMPM model trained on a square domain can naturally generalize to larger rectangular domains (twice as wide here) when using a fully convolutional U-Net.

One notable advantage of NeuralMPM is that the processor is invariant to the number of particles, as the transition model only processes the voxelized representation, while both **p2g** and **g2p** scale linearly. To demonstrate this, we train a model on WaterRamps, which contains about 2.3k particles and 600 timesteps, and evaluate it on WaterDrop-XL, which features about four times more particles, 1000 timesteps, and no obstacles. An example snapshot is displayed in Figure 7. The larger number of particles only affects interpolation steps between the grid and particles, resulting in a negligible impact on total inference time, making the model nearly as fast despite 4 times more particles. We also validate generalization quantitatively by comparing the error rates on WaterDrop-XL of a model trained directly on it and the model trained solely on WaterRamps. With the same training budget, the latter achieves a lower MSE at 20.92×10^{-3} against 28.09×10^{-3}. More trajectories are displayed in Figure 23.

If a domain-agnostic processor architecture is used, such as a fully convolutional U-Net or an FNO, then NeuralMPM can generalize to different domain shapes without retraining, as shown in Fig 7. We demonstrate this ability by considering a model solely trained on WaterRamps, a square domain of size 0.84×0.84 mapped to 64×64 grids. Without retraining, we perform inference with this model on larger unseen environments of size 1.68×0.84, and change the grid size to 128×64. The unseen environments were built

by merging and modifying initial conditions of held-out test trajectories from WaterRamps. NeuralMPM emulates particles in this larger and rectangular domain despite being trained on a smaller square domain with a smaller grid, showing that a U-Net can generalize to other domains. No ground truth is displayed as Sanchez-Gonzalez et al. (2020) provide no information about the data generation. More trajectories are shown in Figure 24.

4.4 Inverse design problem

Finally, we demonstrate the application of NeuralMPM for inverse problems on a toy inverse design task that consists in optimizing the direction of a ramp to make the particles reach a target location, similar to (Allen et al., 2022). We place a blob of water at different starting locations, and we then place a ramp at some location, with a random initial angle α. The goal is to spin the ramp by tuning α in order to make the water end up at a desired location. The main challenges of this task are the long-range time horizon of the goal and the presence of nonlinear physical dynamics. We proceed by selecting the point where we want the water to end up and compute the average distance between the point and particles at the last simulation frame. We then minimize the distance via gradient descent, leveraging the differentiability of NeuralMPM to solve this inverse design problem. We show an example optimization in Figure 8, and additional examples in Appendix D.

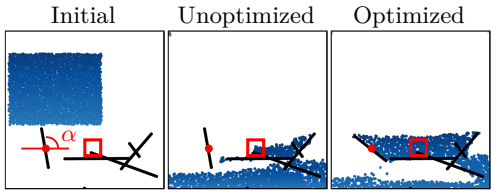

Figure 8: **Inverse design problem.** We exploit NeuralMPM's differentiability to optimize the angle α of a ramp, anchored at the red dot, in order to get the water close to the red square region.

5 Conclusion

Summary. We presented NeuralMPM, a neural emulation framework for particle-based simulations inspired by the hybrid Eulerian-Lagrangian Material Point Method. We have shown its effectiveness in simulating a variety of materials and interactions, its ability to generalize to larger systems and its use in inverse problems. Crucially, NeuralMPM trains in 6% of the time it takes to train GNS and DMCF to comparable accuracy, and is 5x-10x faster at inference time. By interpolating particles onto a fixed-size grid, global information is distilled into a voxelized representation that is easier to learn and process with powerful image-to-image models. The use of voxelization allows NeuralMPM to bypass expensive graph constructions, and the interpolation leads to easier generalization to a larger number of particles and constant runtime. The lack of expensive graph construction and message passing also allows for more autoregressive steps and parallel rollouts.

Limitations. Like other approaches, NeuralMPM is limited by the computation used to process the structure of the point cloud. In our case, voxelization means we cannot deal with particles that lie outside of the domain and are limited to regular grids. Additionally, the size of the voxels is directly related to the number of particles within a given volume. If the voxels are too large, the model will fail to capture finer details. Conversely, if they are too small, the model may struggle due to insufficient local structure. Similarly, performance can degrade in very sparse domains. Compressible fluids might also present challenges, though this requires further verification.

Future work. Our work is only a first step towards hybrid Eulerian-Lagrangian neural emulators, leaving many avenues for future research. Extending NeuralMPM to 3D systems is a natural continuation of this work. Future studies could also explore alternative particle-to-grid and grid-to-particle functions, like the non-uniform Fourier transform (Fessler & Sutton, 2003), or more sophisticated interpolation methods

from classical MPM literature (Nguyen et al., 2023). A less traditional direction is to make NeuralMPM probabilistic and encode richer distributional information about the particles in the grid nodes, instead of maintaining only a mean value. This could potentially improve NeuralMPM's ability to resolve subgrid phenomena. Finally, advances in Lagrangian Particle Tracking (Schröder & Schanz, 2023) will eventually make it possible to create datasets from real-world data, enabling the training of NeuralMPM directly from data without the need for the costly design process of a numerical simulator.

References

Benedikt Alkin, Andreas Fürst, Simon Schmid, Lukas Gruber, Markus Holzleitner, and Johannes Brandstetter. Universal physics transformers: A framework for efficiently scaling neural operators, 2024. URL `https://arxiv.org/abs/2402.12365`.

Kelsey Allen, Tatiana Lopez-Guevara, Kimberly L Stachenfeld, Alvaro Sanchez Gonzalez, Peter Battaglia, Jessica B Hamrick, and Tobias Pfaff. Inverse design for fluid-structure interactions using graph network simulators. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 13759–13774. Curran Associates, Inc., 2022.

Johannes Brandstetter, Rob Hesselink, Elise van der Pol, Erik J Bekkers, and Max Welling. Geometric and physical quantities improve e(3) equivariant message passing. In *International Conference on Learning Representations*, 2022a.

Johannes Brandstetter, Daniel E. Worrall, and Max Welling. Message passing neural PDE solvers. In *International Conference on Learning Representations*, 2022b.

Marco Cuturi, Laetitia Meng-Papaxanthos, Yingtao Tian, Charlotte Bunne, Geoff Davis, and Olivier Teboul. Optimal transport tools (ott): A jax toolbox for all things wasserstein. *arXiv preprint arXiv:2201.12324*, 2022.

Nitin P. Daphalapurkar, Hongbing Lu, Demir Coker, and Ranga Komanduri. Simulation of dynamic crack growth using the generalized interpolation material point (gimp) method. *International Journal of Fracture*, 143(1):79–102, 01 2007.

J.A. Fessler and B.P. Sutton. Nonuniform fast fourier transforms using min-max interpolation. *IEEE Transactions on Signal Processing*, 51(2):560–574, 2003.

Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

Antonio Elia Forte, Paul Z. Hanakata, Lishuai Jin, Emilia Zari, Ahmad Zareei, Matheus C. Fernandes, Laura Sumner, Jonathan Alvarez, and Katia Bertoldi. Inverse design of inflatable soft membranes through machine learning. *Advanced Functional Materials*, 32(16):2111610, 2022.

R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181(3):375–389, 12 1977.

Jiaqi Han, Wenbing Huang, Hengbo Ma, Jiachen Li, Josh Tenenbaum, and Chuang Gan. Learning physical dynamics with subequivariant graph neural networks. *Advances in Neural Information Processing Systems*, 35:26256–26268, 2022.

Erin Hastings and Jaruwan Mesit. Optimization of large-scale, real-time simulations by spatial hashing. January 2005.

Siyu He, Yin Li, Yu Feng, Shirley Ho, Siamak Ravanbakhsh, Wei Chen, and Barnabás Póczos. Learning to predict the cosmological structure formation. *Proceedings of the National Academy of Sciences*, 116(28): 13825–13832, June 2019.

Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Transactions on Graphics (TOG)*, 37(4):150, 2018a.

Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Trans. Graph.*, 37(4), 07 2018b.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. pmlr, 2015.

Irina Ionescu, James Guilkey, Martin Berzins, Robert M Kirby, and Jeffrey Weiss. Computational simulation of penetrating trauma in biological soft tissues using the material point method. *Stud Health Technol Inform*, 111:213–218, 2005.

Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2 edition, 2008.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 8 2021.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Remi Lam, Alvaro Sanchez-Gonzalez, Matthew Willson, Peter Wirnsberger, Meire Fortunato, Ferran Alet, Suman Ravuri, Timo Ewalds, Zach Eaton-Rosen, Weihua Hu, Alexander Merose, Stephan Hoyer, George Holland, Oriol Vinyals, Jacklynn Stott, Alexander Pritzel, Shakir Mohamed, and Peter Battaglia. Learning skillful medium-range global weather forecasting. *Science*, 382(6677):1416–1421, 2023.

Pablo Lemos, Niall Jeffrey, Miles Cranmer, Shirley Ho, and Peter Battaglia. Rediscovering orbital mechanics with machine learning. *Machine Learning: Science and Technology*, 4(4):045002, 10 2023.

Jin Li, Yang Gao, Ju Dai, Shuai Li, Aimin Hao, and Hong Qin. Mpmnet: A data-driven mpm framework for dynamic fluid-solid interaction. *IEEE Transactions on Visualization and Computer Graphics*, 30(8): 4694–4708, August 2024a. ISSN 2160-9306. doi: 10.1109/tvcg.2023.3272156. URL `http://dx.doi.org/10.1109/TVCG.2023.3272156`.

Shaofan Li and Wing Kam Liu. Meshfree and particle methods and their applications. *Applied Mechanics Reviews*, 55(1):1–34, 01 2002.

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.

Zongyi Li, Nikola Kovachki, Chris Choy, Boyi Li, Jean Kossaifi, Shourya Otta, Mohammad Amin Nabian, Maximilian Stadler, Christian Hundt, Kamyar Azizzadenesheli, et al. Geometry-informed neural operator for large-scale 3d pdes. *Advances in Neural Information Processing Systems*, 36, 2024b.

Steven J. Lind, Benedict D. Rogers, and Peter K. Stansby. Review of smoothed particle hydrodynamics: towards converged lagrangian flow modelling. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 476(2241):20190801, 2020.

Bjoern List, Li-Wei Chen, Kartik Bali, and Nils Thuerey. How temporal unrolling supports neural physics simulators, 2024. URL `https://arxiv.org/abs/2402.12971`.

Bjoern List, Li-Wei Chen, Kartik Bali, and Nils Thuerey. Differentiability in unrolled training of neural physics simulators on transient dynamics. *Computer Methods in Applied Mechanics and Engineering*, 433: 117441, 2025. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2024.117441. URL `https://www.sciencedirect.com/science/article/pii/S0045782524006960`.

Marcelo Alejandro Llano Serna, Márcio Muniz-de Farias, and Hernán Eduardo Martínez-Carvajal. Numerical modelling of alto verde landslide using the material point method. *DYNA*, 82(194):150–159, 11 2015.

Ilya Loshchilov and Frank Hutter. SGDR: Stochastic gradient descent with warm restarts. In *International Conference on Learning Representations*, 2017.

Hongbing Lu, Nitin Daphalapurkar, B. Wang, Samit Roy, and Ranga Komanduri. Multiscale simulation from atomistic to continuum – coupling molecular dynamics (md) with the material point method (mpm). *Philosophical Magazine*, 86:2971–2994, 2006.

Pingchuan Ma, Peter Yichen Chen, Bolei Deng, Joshua B. Tenenbaum, Tao Du, Chuang Gan, and Wojciech Matusik. Learning neural constitutive laws from motion observations for generalizable pde dynamics, 2023. URL `https://arxiv.org/abs/2304.14369`.

J.J. Monaghan. Smoothed particle hydrodynamics and its diverse applications. *Annual Review of Fluid Mechanics*, 44(Volume 44, 2012):323–346, 2012.

K. W. Morton and D. F. Mayers. *Numerical Solution of Partial Differential Equations: An Introduction*. Cambridge University Press, 2 edition, 2005.

Vinh Phu Nguyen, Alban de Vaucorbeil, and Stéphane Bordas. *The Material Point Method: Theory, Implementations and Applications (Scientific Computation) 1st ed. 2023 Edition*. 02 2023. ISBN 3031240693.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

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

Stephen B. Pope. *Turbulent Flows*. Cambridge University Press, 2000.

Lukas Prantl, Benjamin Ummenhofer, Vladlen Koltun, and Nils Thuerey. Guaranteed conservation of momentum for learning particle-based fluid dynamics. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 6901–6913. Curran Associates, Inc., 2022.

Daniel J. Price. Smoothed particle hydrodynamics and magnetohydrodynamics. *Journal of Computational Physics*, 231(3):759–794, 2012. Special Issue: Computational Plasma Physics.

Milad Rakhsha, Christopher E. Kees, and Dan Negrut. Lagrangian vs. eulerian: An analysis of two solution methods for free-surface flows and fluid solid interaction problems. *Fluids*, 6(12), 2021.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pp. 234–241. Springer, 2015.

Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. Learning to simulate complex physics with graph networks. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 8459–8468. PMLR, 7 2020.

Patrick Schnell and Nils Thuerey. Stabilizing backpropagation through time to learn complex physics, 2024. URL `https://arxiv.org/abs/2405.02041`.

Andreas Schröder and Daniel Schanz. 3d lagrangian particle tracking in fluid mechanics. *Annual Review of Fluid Mechanics*, 55(Volume 55, 2023):511–540, 2023. ISSN 1545-4479.

Omer Rochman Sharabi and Gilles Louppe. Trick or treat? evaluating stability strategies in graph network-based simulators. 2023. URL https://api.semanticscholar.org/CorpusID:268033249.

Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. A material point method for snow simulation. *ACM Trans. Graph.*, 32(4), 07 2013.

Deborah Sulsky, Zhen Chen, and Howard L. Schreyer. A particle method for history-dependent materials. *Computer Methods in Applied Mechanics and Engineering*, 118:179–196, 1993.

Artur Toshev, Gianluca Galletti, Fabian Fritz, Stefan Adami, and Nikolaus Adams. Lagrangebench: A lagrangian fluid mechanics benchmarking suite. *Advances in Neural Information Processing Systems*, 36, 2024.

Benjamin Ummenhofer, Lukas Prantl, Nils Thuerey, and Vladlen Koltun. Lagrangian fluid simulation with continuous convolutions. In *International Conference on Learning Representations*, 2020.

Renato Vacondio, Corrado Altomare, Matthieu De Leffe, Xiangyu Hu, David Le Touzé, Steven Lind, Jean-Christophe Marongiu, Salvatore Marrone, Benedict D. Rogers, and Antonio Souto-Iglesias. Grand challenges for smoothed particle hydrodynamics numerical schemes. *Computational Particle Mechanics*, 8(3): 575–588, 5 2021.

Hrishikesh Viswanath, Yue Chang, Julius Berner, Peter Yichen Chen, and Aniket Bera. Reduced-order neural operators: Learning lagrangian dynamics on highly sparse graphs, 2024. URL https://arxiv.org/abs/2407.03925.

Alistair White, Niki Kilbertus, Maximilian Gelbrecht, and Niklas Boers. Stabilized neural differential equations for learning dynamics with explicit constraints, 2024. URL https://arxiv.org/abs/2306.09739.

Rene Winchenbach and Nils Thuerey. Symmetric basis convolutions for learning lagrangian fluid mechanics, 2024. URL https://arxiv.org/abs/2403.16680.

Yusheng Xu, Xiaohua Tong, and Uwe Stilla. Voxel-based representation of 3d point clouds: Methods, applications, and its potential use in the construction industry. *Automation in Construction*, 126:103675, 2021.

Yuan Yin, Vincent Le Guen, Jérémie Dona, Emmanuel de Bézenac, Ibrahim Ayed, Nicolas Thome, and Patrick Gallinari. Augmenting physical models with deep networks for complex dynamics forecasting*. *Journal of Statistical Mechanics: Theory and Experiment*, 2021(12):124012, December 2021. ISSN 1742-5468. doi: 10.1088/1742-5468/ac3ae5. URL http://dx.doi.org/10.1088/1742-5468/ac3ae5.

Allen R. York II, Deborah Sulsky, and Howard L. Schreyer. Fluid–membrane interaction based on the material point method. *International Journal for Numerical Methods in Engineering*, 48(6):901–924, 2000.

Yalan Zhang, Xiaojuan Ban, Feilong Du, and Wu Di. Fluidsnet: End-to-end learning for lagrangian fluid simulation. *Expert Systems with Applications*, 152:113410, 2020. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2020.113410. URL https://www.sciencedirect.com/science/article/pii/S0957417420302347.

Qingqing Zhao, David B Lindell, and Gordon Wetzstein. Learning to solve PDE-constrained inverse problems with graph networks. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 26895–26910. PMLR, 7 2022.

A Supplementary Material

In addition to the supplementary material on the project page (`https://neuralmpm.isach.be`), we have uploaded videos, minimal code for reproduction, and baseline comparisons to an online cloud storage that can be found at `https://bit.ly/neuralmpm_suppmaterial`.

B Training details

Hardware. We run all our experiments using the same hardware: 4 CPUs, 24GB of RAM, and an NVIDIA RTX A5000 GPU with 24GB of VRAM. For reproducing the results of DMCF, we kept the A5000 GPU but it required up to 96GB of RAM for training.

Data Preprocessing. Similar to Prantl et al. (2022), we slightly alter the original MPM datasets to add boundary particles, as the original data from Sanchez-Gonzalez et al. (2020) does not have them. We define the velocity at a timestep to be $\mathbf{v}_t = \mathbf{v}_t - \mathbf{v}_{t-1}$. We therefore skip the first step during training for which no velocity is available.

Implementation. Our implementation, training scripts, experiment configurations, and instructions for reproducing results are publicly available at [URL]. We implement NeuralMPM using PyTorch (Paszke et al., 2019), and use PyTorch Geometric (Fey & Lenssen, 2019) for implementing efficient particle-to-grid functions, more specifically from the Scatter and Cluster modules. For memory efficiency, we do not store all (up to) 1,000 training trajectories in memory, and rather use a buffer of about 16 trajectories over which several epochs are performed before loading a new buffer of random trajectories.

Baselines. We use the official implementations and training instructions of GNS (Sanchez-Gonzalez et al., 2020) and DMCF (Prantl et al., 2022) to reproduce their results and conduct new experiments. More specifically, we train GNS as instructed for 20M steps on all four datasets, using their provided configuration. For DMCF, we follow their default configurations and train for 400K iterations for each dataset. In datasets that were not used by the original authors, VariableGravity and Dam break 2D, we performed hyperparameter search. GNS and DMCF both were trained for a total budget of 60 GPU-days per dataset, and the best performance was reported.

Normalization. We normalize the input of the model over each channel individually. We investigated computing the statistics across a buffer, resembling (Ioffe & Szegedy, 2015), and precomputing them on the whole training set and found no difference in performance. During inference, we use the precomputed statistics.

Code. The code, together with additional videos, is available at the project's website [URL].

C Additional Discussion

Boundary conditions. Boundary conditions are represented as stationary particles. Like moving particles, their density and zero velocity are interpolated into channels, but their positions is enforced to remain fixed during rollout. This approach allows for flexible boundary shapes, such as the ramps in WaterRamps.

Voxelization and interpolation. The voxelization procedure used is implemented using CUDA by PyTorchCluster Fey & Lenssen (2019), **p2g** is implemented by us based on the voxelized representation, and **g2p** uses PyTorch's bilinear interpolation (`grid_sample`) Paszke et al. (2019) based on the voxelized representation of the data. The cost of those operations is $\mathcal{O}(N)$, e.g., a naive implementation of voxelization assigns each point $\mathbf{p} = (x, y, z)$ to a voxel index computed as:

$$\mathbf{v} = \left(\left\lfloor \frac{x}{s} \right\rfloor, \left\lfloor \frac{y}{s} \right\rfloor, \left\lfloor \frac{z}{s} \right\rfloor \right),$$

where s is the voxel size and $\lfloor \cdot \rfloor$ denotes the floor function. Since this computation is performed in constant time for each of the N points, the overall complexity is $\mathcal{O}(N)$. Alternative interpolation methods could be considered, although quick experimentation with bicubic interpolation for grid-to-particles or different weighting schemes for voxelization did not yield much difference in our pipeline. Some methods, like GIOROM Viswanath et al. (2024) and GINO Li et al. (2024b), learn the interpolation through the use of graph-based neural networks, but this strongly increases computational costs due to the construction of a graph and the forward pass in the neural network.

Limitations of voxelization. One of the main issues of voxelization onto a fixed regular grid means the domain is limited and it must be rectangular, like for the material point method. While a rectangular domain can be defined with local boundary conditions over the irregular domain of interest, this will be a disadvantage for sparse systems, for which Lagrangian methods might be a better fit. Likewise, out-of-bounds particles can be handled either with a larger domain, which might be more computationally expensive, by clamping the positions, or simply dropping the out-of-bounds particles. These particular conditions are known disadvantages of MPM, and of some implementations of SPH, and are therefore inherited by NeuralMPM.

Rollout stability. Long-term stability during rollout of neural emulators is an open problem. As such, we decouple this problem from the design of the method or the architecture. There are commonly used tricks in the literature, such as noise injection, backpropagating through the autoregressive rollout, temporal bundling, having two models work in tandem (one to produce large timesteps and one to interpolate between them), using explicit constraints White et al. (2024), or modifying the gradients Schnell & Thuerey (2024). (List et al., 2025) perform a comparative study of some of these methods.

D Supplementary results

Additional results on DamBreak2D We have also compared the accuracy and inference speed of NeuralMPM against a different implementation of GNS, and one of SEGNN, both provided by (Toshev et al., 2024), in Tables 2 and 3. As in Table 1, NeuralMPM outperforms both by a margin baselines.

	MSE↓	EMD↓
Ours	**20.76**	**2.88**
GNS	114.40	224.1
SEGNN	124.39	268.4

Table 2: Full-trajectory MSE ($\times 10^{-3}$) and Sinkhorn distance (EMD) ($\times 10^{-4}$) for NeuralMPM, GNS, and SEGNN Brandstetter et al. (2022a) on the Dam Break 2D dataset from LagrangeBench. The two latter models are baselines provided by LagrangeBench.

	Single call ($T = 1$)	Rollout ($T = 401$)
Ours	**7.41**	**193.50**
GNS	20.46	8,170.47
SEGNN	46.04	18,194.10

Table 3: Inference time (in ms) of NeuralMPM, GNS, and SEGNN Brandstetter et al. (2022a) on the Dam Break 2D dataset from LagrangeBench. Times were averaged over all test trajectories. NeuralMPM predicts 16 frames in a single model call and still outperforms the two baselines per call, which further widens the gap for the total rollout time.

Evaluation. In Table 4, we report the numerical MSE rollout values that were reported in the bar plots depicted in Figure 3 for Goop. Also, Figures 12 and 10 displays the error when rolling out a model for each

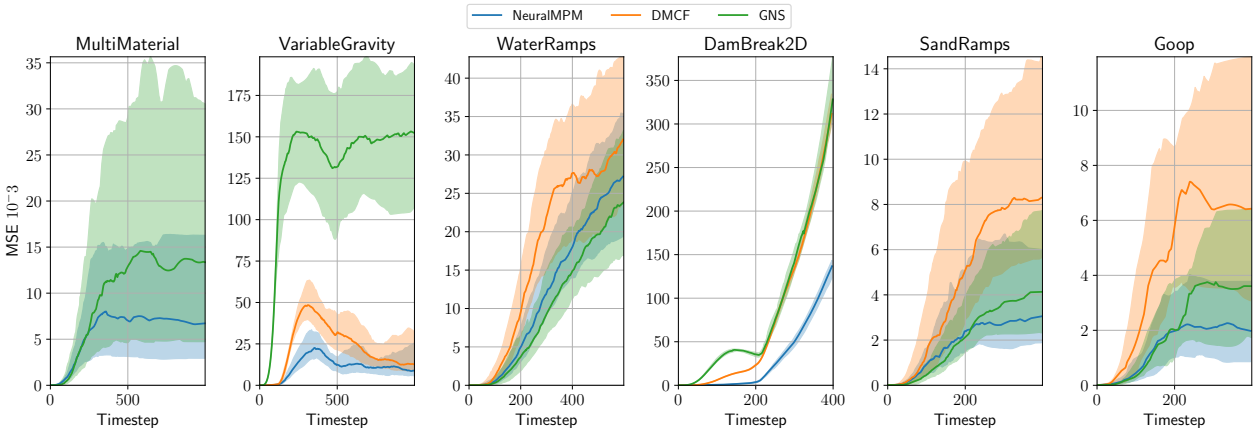

Figure 9: **MSE propagation during rollout.** We show the 25th, 50th and 75th percentile of the MSE, computed over particles and simulations, at each timestep during the rollout for each model and dataset. The accuracy decreases as errors accumulate. While training in 5% of the time, NeuralMPM outperform the baselines on all datasets, except WaterRamps, where it is slightly worse than GNS.

dataset, both in terms of MSE and EMD. For both metrics, the error starts low and slowly accumulates over time. For the EMD, we use the Sinkhorn algorithm provided by (Cuturi et al., 2022).

Parameter	**Value**	**MSE** ($\times 10^{-3}$)
K (No noise)	1	3.2
	2	3.3
	3	2.4
	4	2.2
K (With noise)	1	3.5
	2	2.5
	3	2.4
	4	3.0
Time bundling m	1	6.6
	2	4.5
	4	3.5
	8	2.1
	16	2.9
	32	3.5
Grid size	32	5.5
	64	2.4
	128	7.1

Parameter	**Value**	**MSE** ($\times 10^{-3}$)
Grid noise	0	3.2
	0.001	2.4
	0.005	6.9
Particle noise	0	2.2
	0.0003	2.4
	0.0006	2.4
	0.001	2.1
U-Net Depth	2	3.3
	3	3.0
	4	2.4
	5	2.3
U-Net Width	32	2.6
	64	2.3
	128	2.2

Table 4: Ablation results for Goop.

Grid-to-grid network. Although we have used a U-Net architecture for the grid-to-grid processor, NeuralMPM can be used with any grid-to-grid processor and is not limited to that network. For example, in Figure 13 and Table 5 we present qualitative and quantitative ablation results, respectively, for NeuralMPM using an FNO network (Li et al., 2021) as the grid-to-grid processor. Results show that the FNO processor is slightly worse than the U-Net processor.

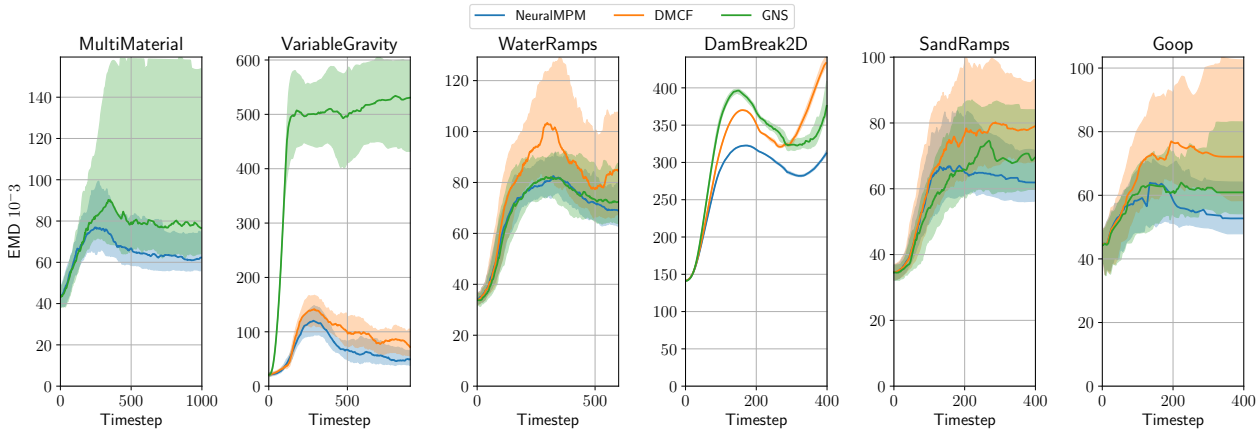

Figure 10: **EMD propagation during rollout.** We show the 25th, 50th and 75th percentile of the EMD, computed over particles and simulations, at each timestep during the rollout for each model dataset daset. The accuracy decreases as errors accumulate. While training in 5% of the time, NeuralMPM outperform the baselines on all datasets.

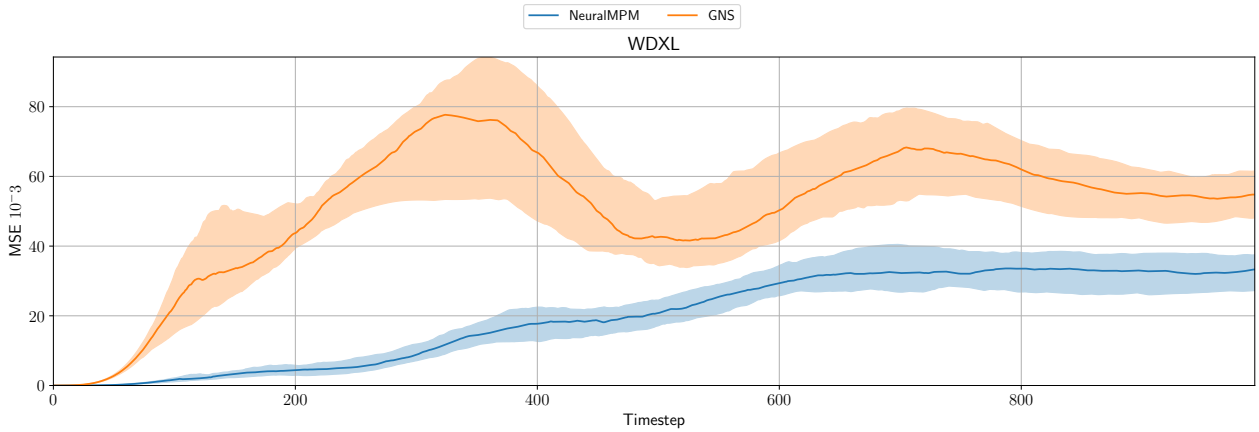

Figure 11: **MSE propagation during rollout for the generalization task WaterDrop-XL.** We show the 25th, 50th and 75th percentile of the MSE, computed over particles and simulations, at each timestep during the rollout for each model and daset. The accuracy decreases as errors accumulate. The models used were trained on WaterRamps and tested on WaterDrop-XL to evaluate their generalization. NeuralMPM performs better despite having a slightly worse MSE on WaterRamps than GNS.

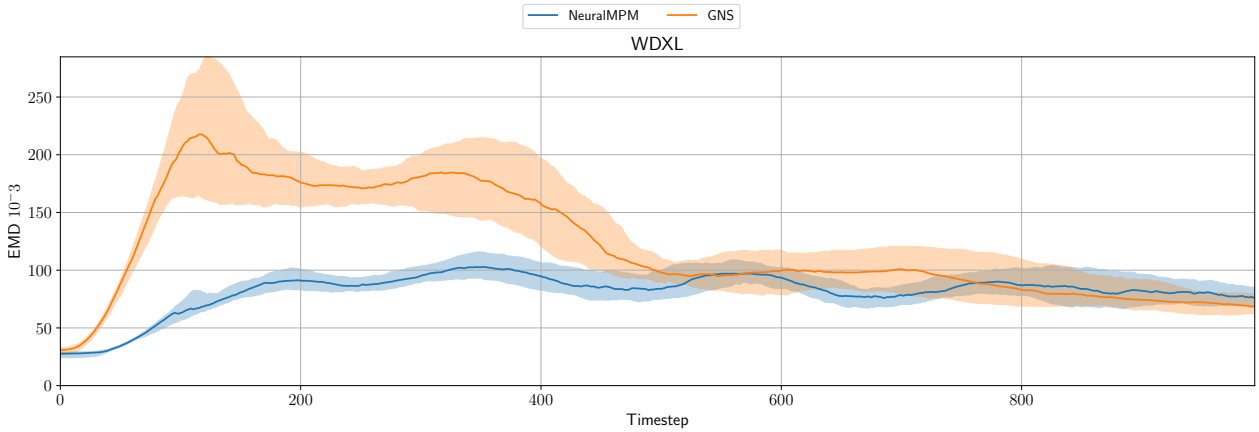

Figure 12: **EMD propagation during rollout for the generalization task WaterDrop-XL.** We show the 25th, 50th and 75th percentile of the MSE, computed over particles and simulations, at each timestep during the rollout for each model and daset. The accuracy decreases as errors accumulate. The models used were trained on WaterRamps and tested on WaterDrop-XL to evaluate their generalization. NeuralMPM performs better despite having a slightly worse MSE on WaterRamps than GNS.

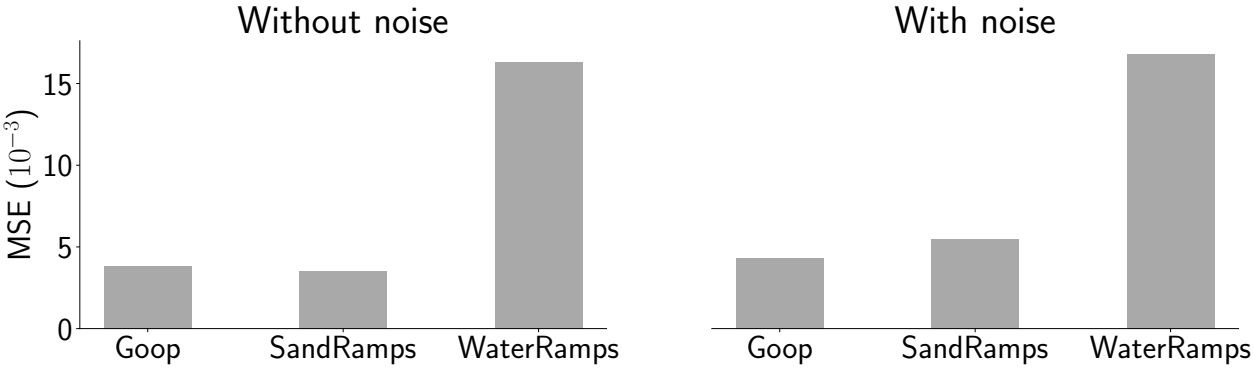

Figure 13: **FNO processor.** NeuralMPM with an FNO processor and default architecture. Rollout MSE ($\times 10^{-3}$) for different datasets.

Data	FNO with noise	FNO without noise
WaterRamps	16.8	16.3
SandRamps	5.5	3.5
Goop	4.3	3.8

Table 5: Rollout MSE ($\times 10^{-3}$) for NeuralMPM with an FNO processor and default architecture, with and without noise.

Additional inverse problem examples. We show two additional optimization examples in Figure 14.

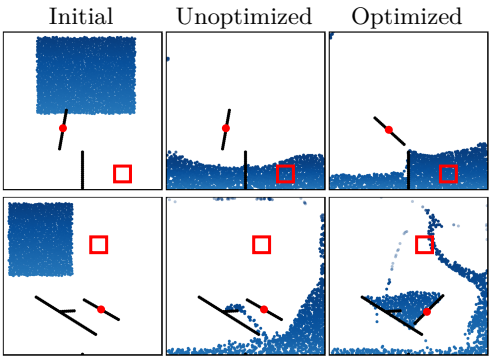

Figure 14: **Inverse design problem.** Additional optimization examples. We exploit NeuralMPM's differentiability to optimize the angle α of a ramp, anchored at the red dot, in order to get the water close to the red square region.

E Gallery of predicted trajectories

We present additional rollout comparisons in Figures 15 and 16. Further, in addition to the trajectories in Figures 2 and 4, we show additional trajectories emulated with NeuralMPM for all datasets in Figures 17, 18, 19, 20, 21, 22, 23, and 24. We also release *videos* in the supplementary material, which we recommend watching to better see the details and limitations of NeuralMPM. This includes 10 videos per dataset of emulated trajectories on held-out test simulations.

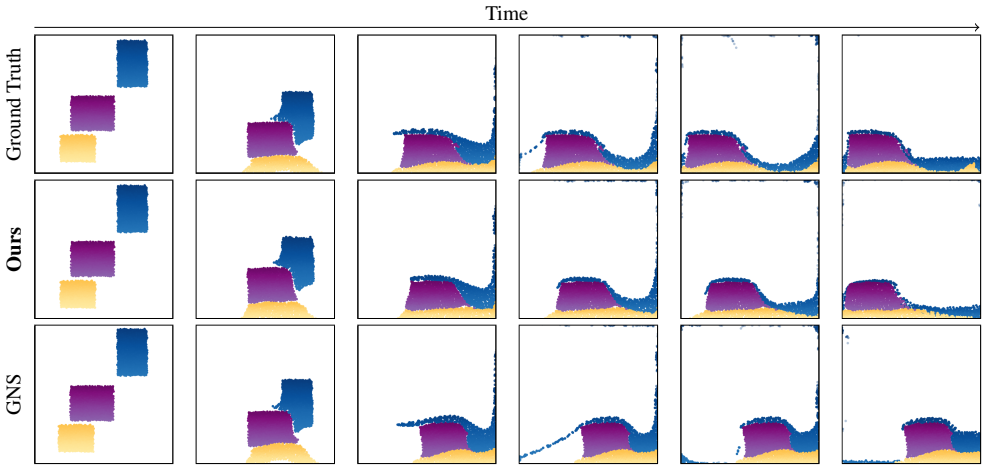

Figure 15: **Example MultiMaterial trajectory against baselines. Each method is unrolled starting from the initial conditions of a random test trajectory not seen during training.**

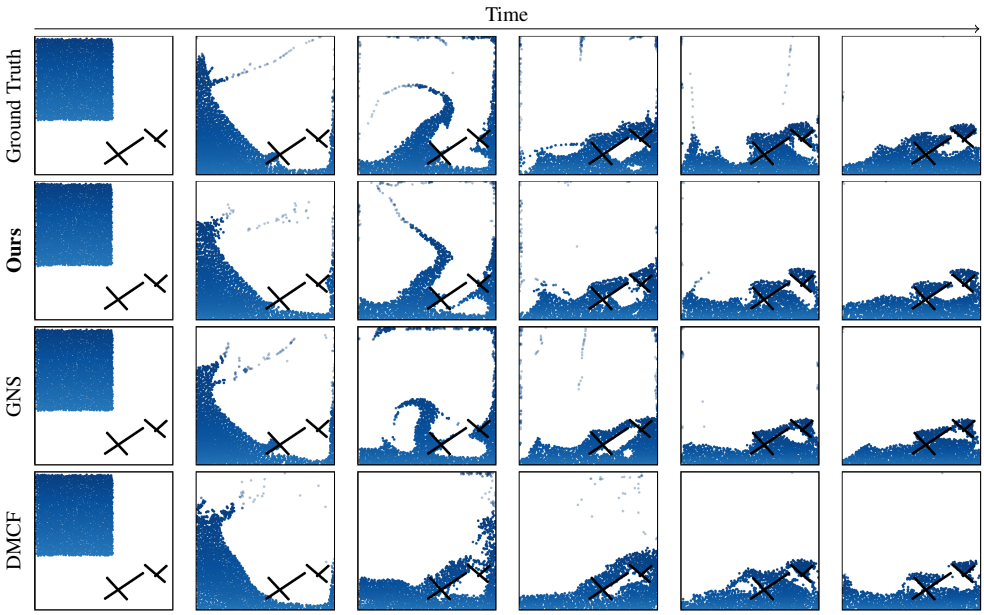

Figure 16: **Example WaterRamps trajectory against baselines. Each method is unrolled starting from the initial conditions of a random test trajectory not seen during training.**

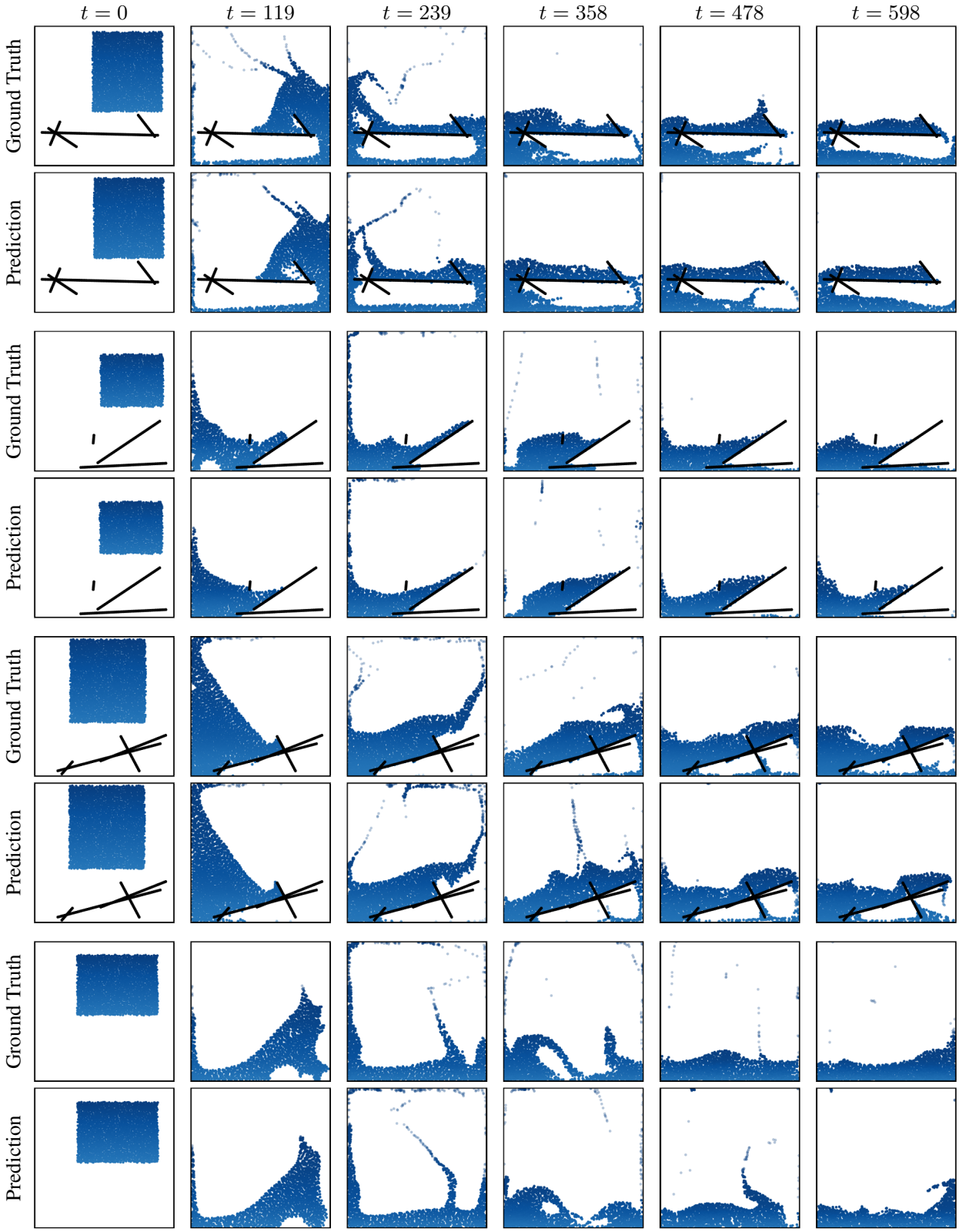

Figure 17: **Additional WaterRamps predicted trajectories.** Evenly spaced in time snapshots of predicted unrolled trajectories against ground truth. All trajectories are from the held-out test set.

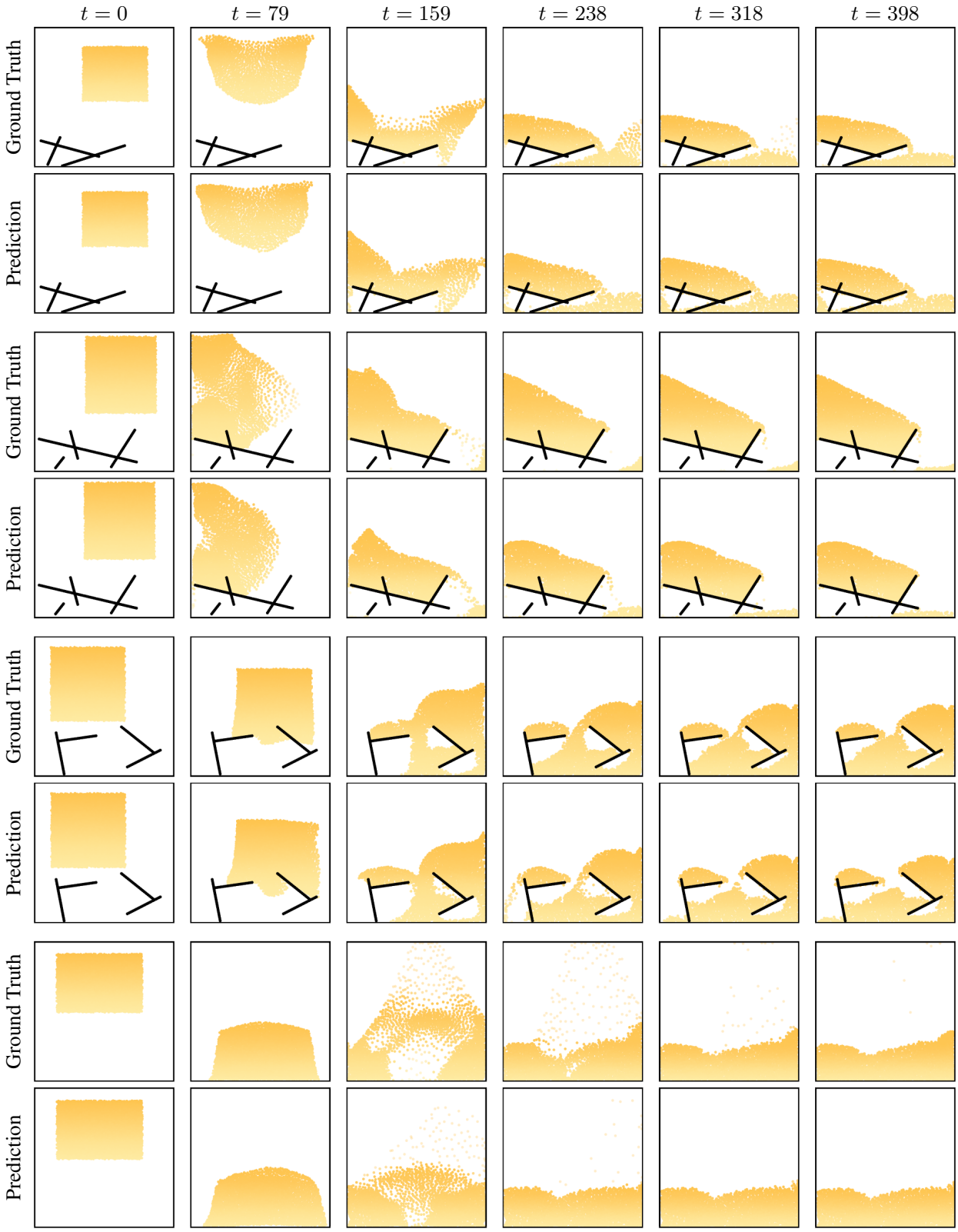

Figure 18: **Additional SandRamps predicted trajectories.** Evenly spaced in time snapshots of predicted unrolled trajectories against ground truth. All trajectories are from the held-out test set.

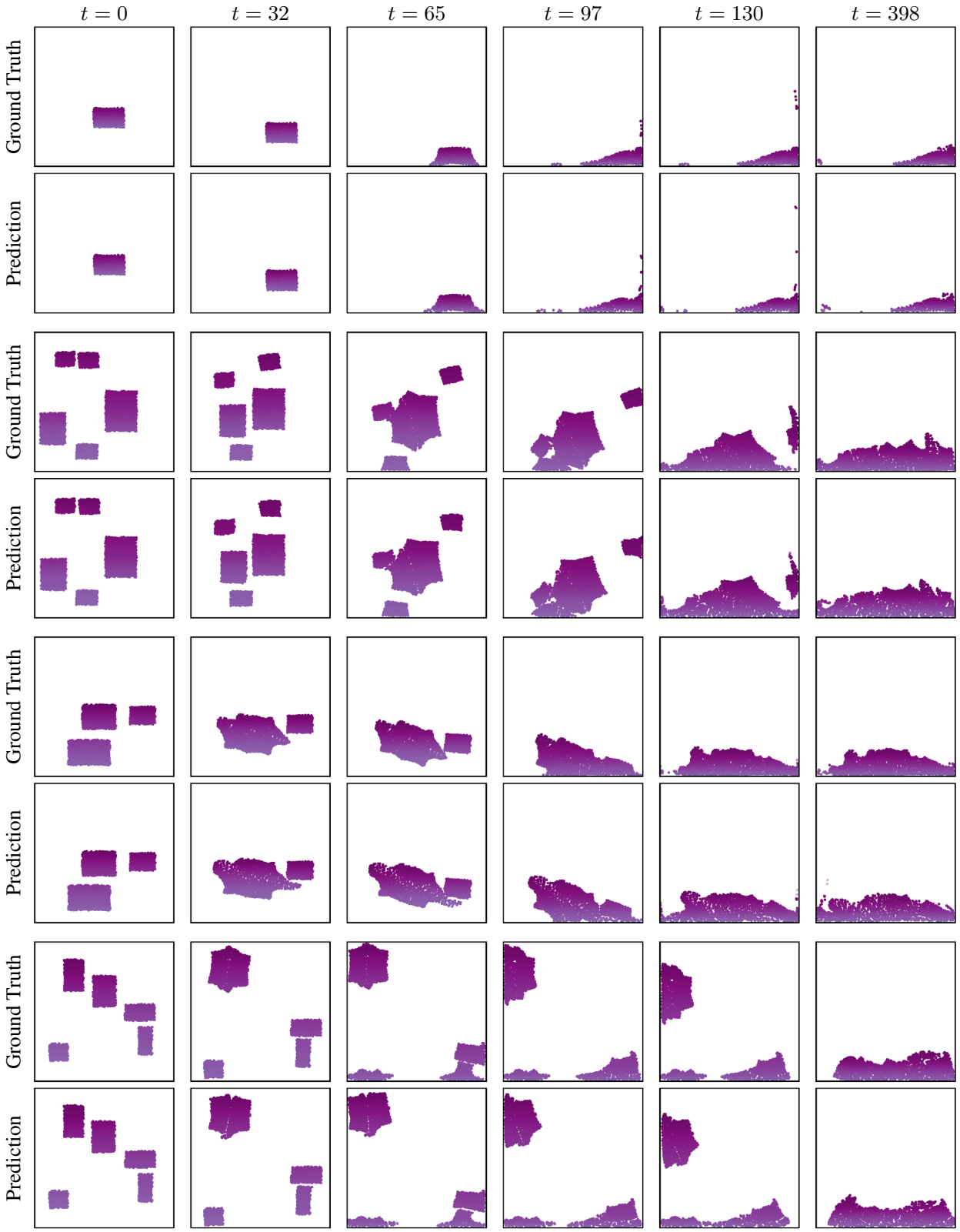

Figure 19: **Additional Goop predicted trajectories.** Snapshots of predicted unrolled trajectories against ground truth. All trajectories are from the held-out test set. Due to Goop quickly reaching equilibrium, more snapshots are taken in the first half of the trajectory.

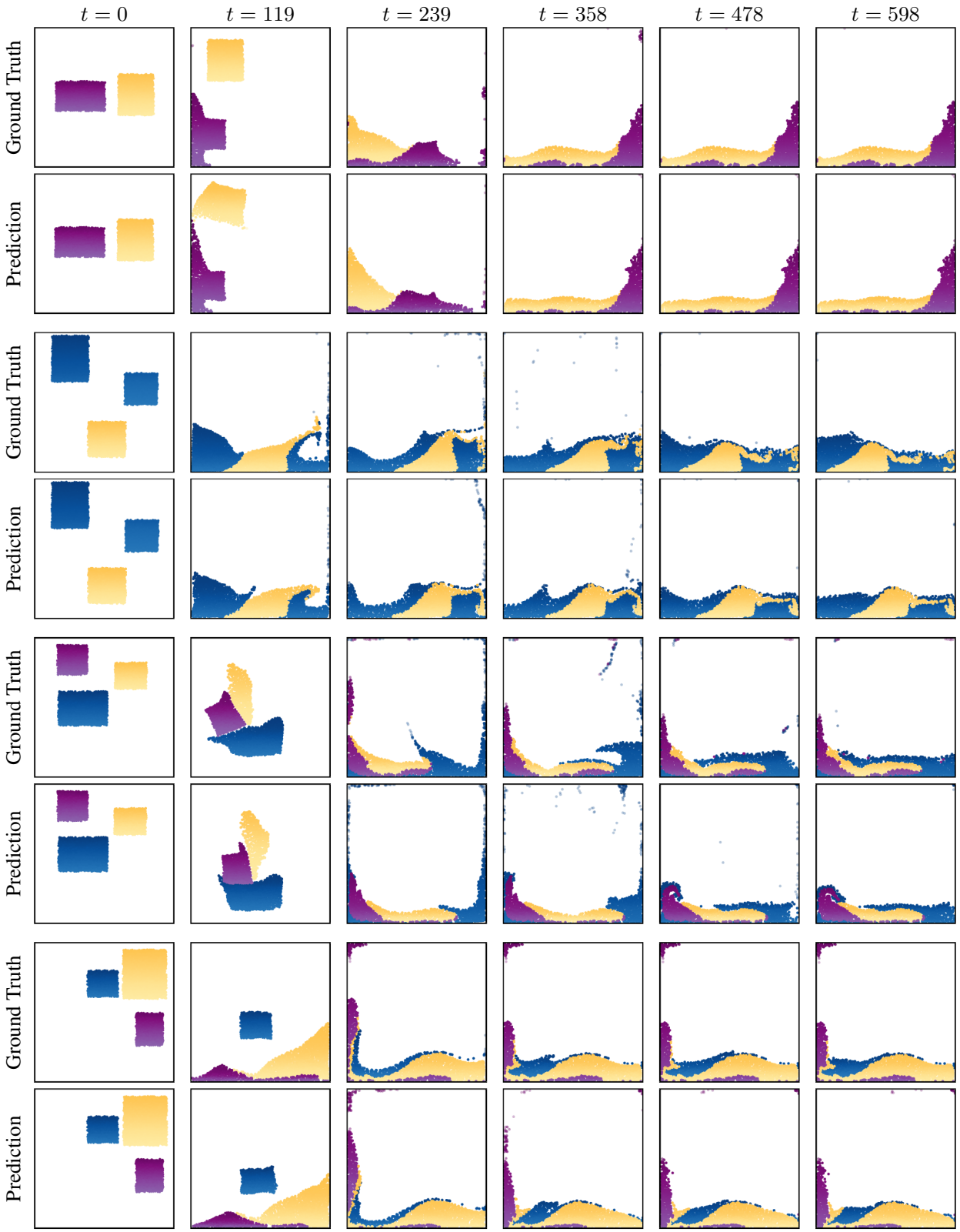

Figure 20: **Additional MultiMaterial predicted trajectories.** Evenly spaced in time snapshots of predicted unrolled trajectories against ground truth. All trajectories are from the held-out test set. The first trajectory illustrates a rare failure where the shape of sand particles is not retained, even though all particles are supposed to maintain the same velocity while airborne, as they are thrown against the wall.

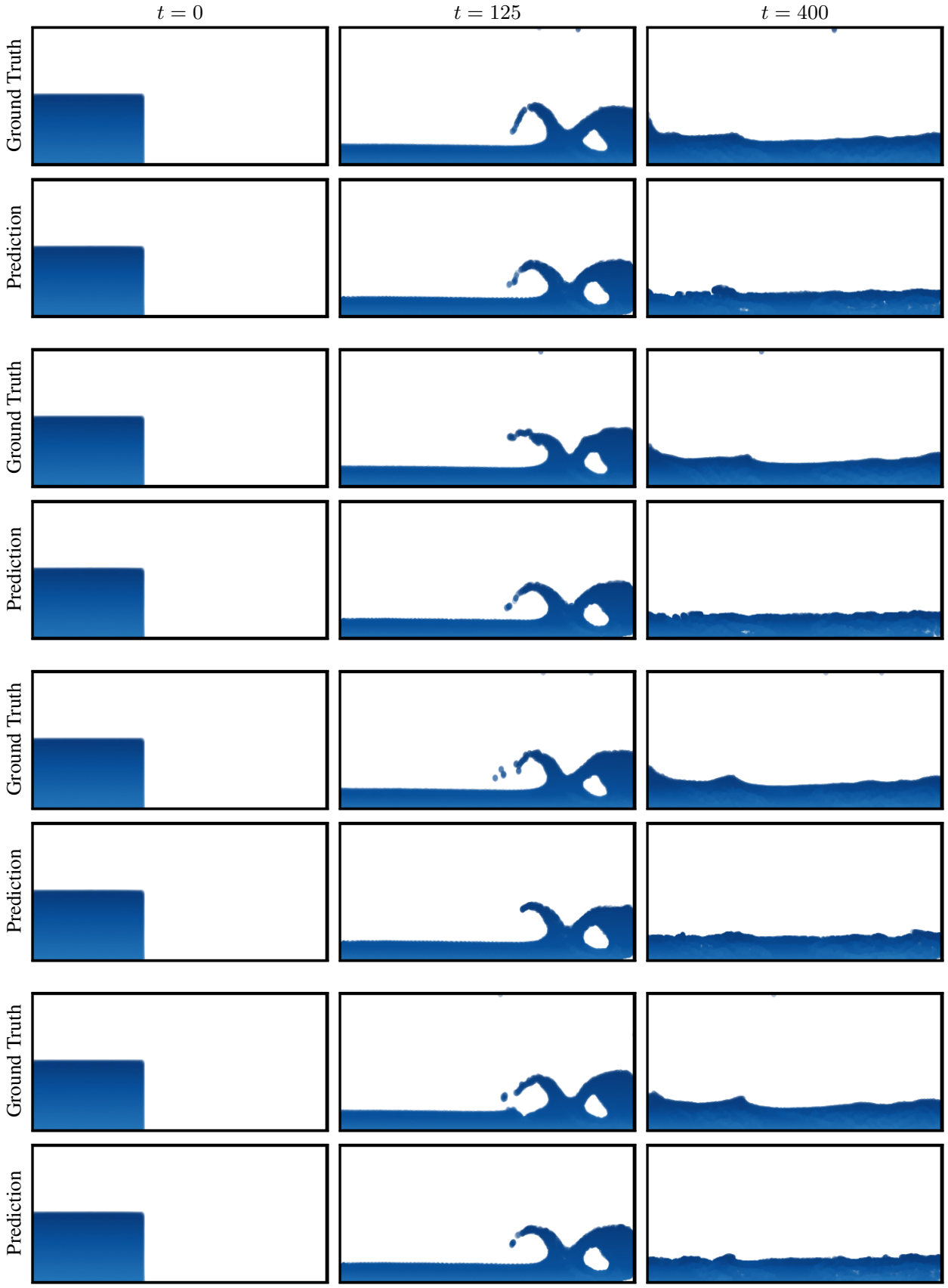

Figure 21: **Additional Dam Break 2D predicted trajectories.** Snapshots of predicted trajectories against ground truth. All trajectories come from the held-out test set. To better show the differences of these longer sequences, we select the following timesteps not even in time: $t \in \{0, 125, 400\}$. In the last trajectory, NeuralMPM struggles to follow the gravity direction and breaks down over time.

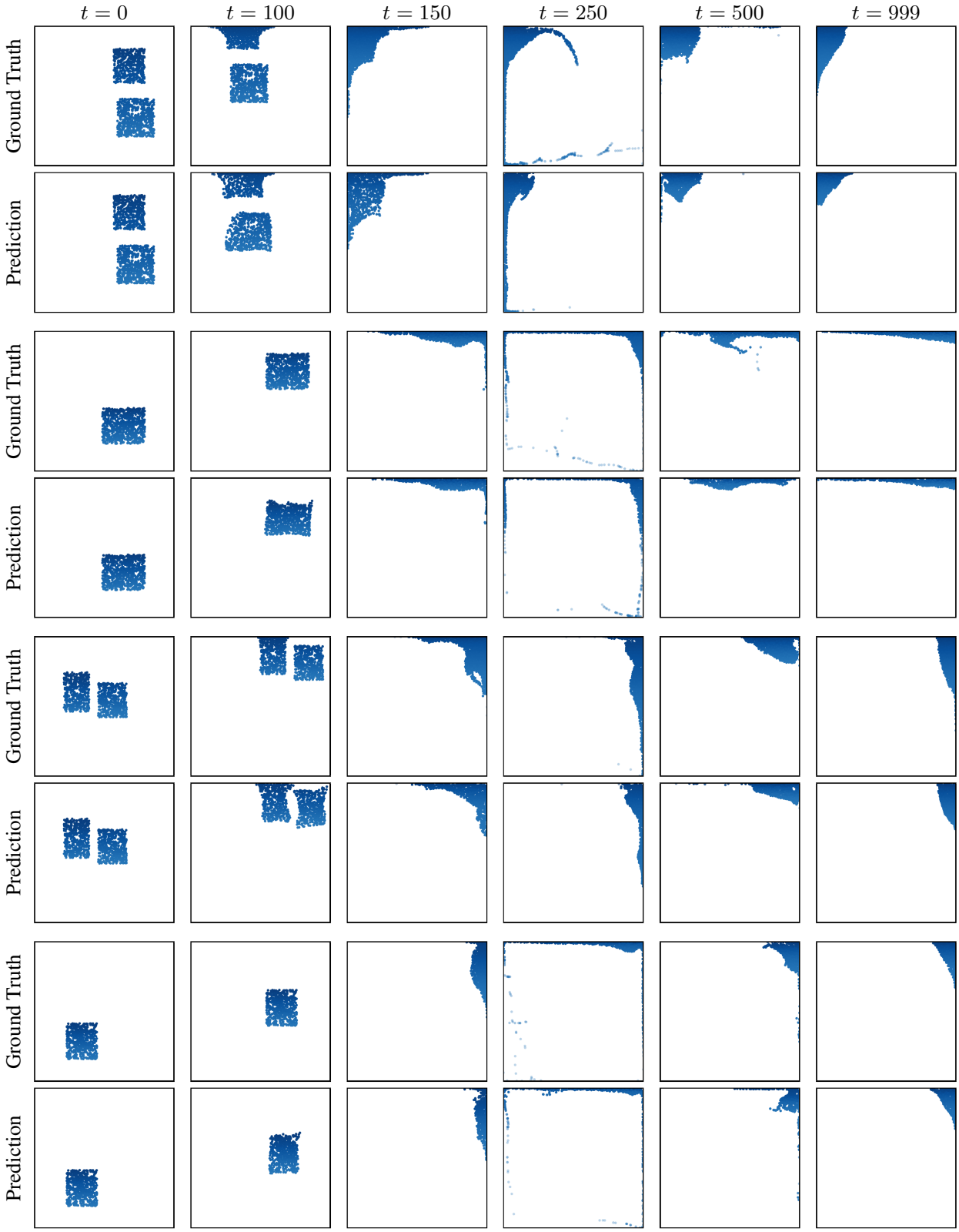

Figure 22: **Additional VariablyGravity predicted trajectories.** Snapshots of predicted trajectories against ground truth. All trajectories come from the held-out test set.

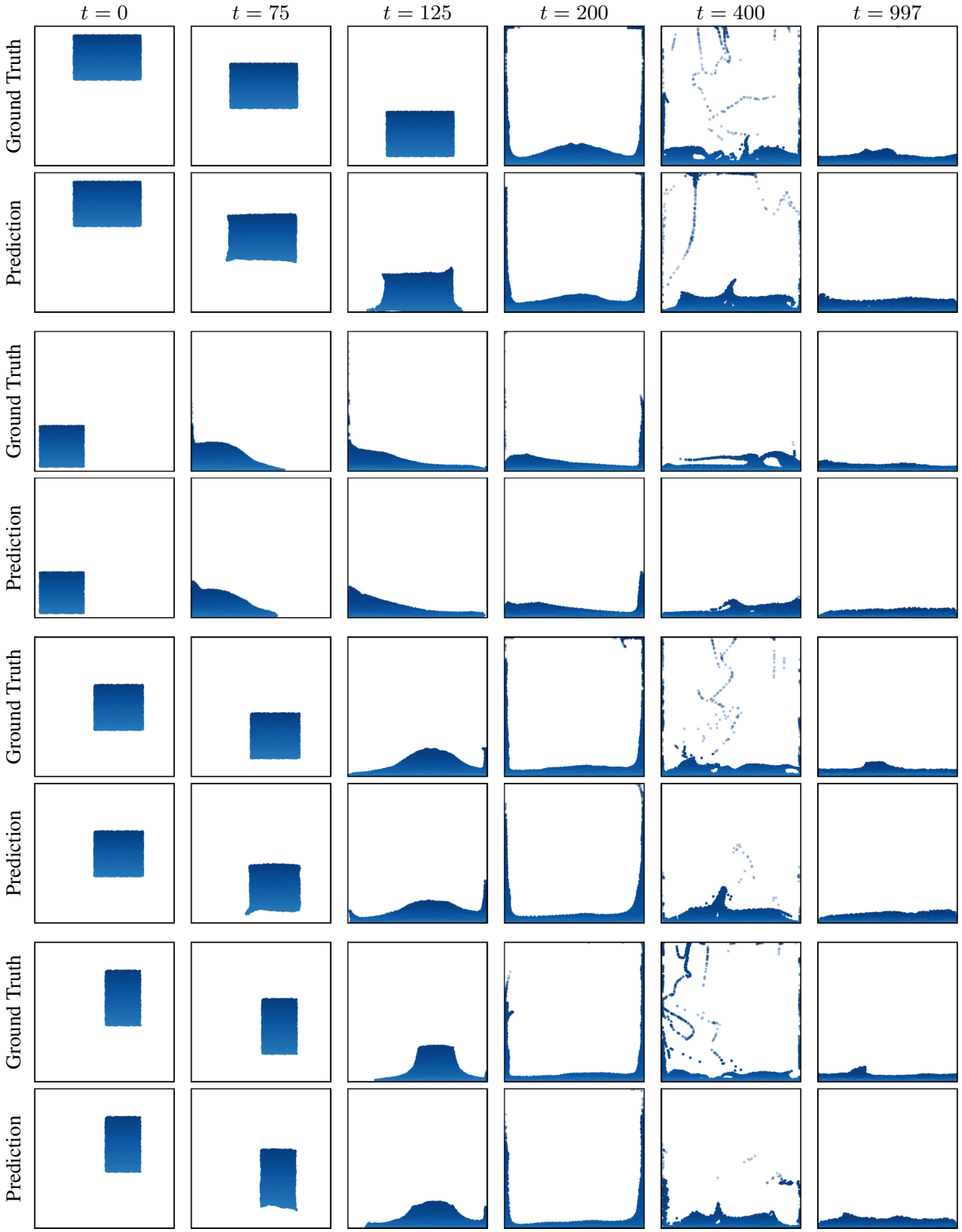

Figure 23: **Generalization to more particles on WaterDrop-XL.** Snapshots of predicted trajectories emulated using a model trained solely on WaterRamps, against ground truth. All trajectories come from the held-out test set of WaterDrop-XL. To better show the differences of these longer sequences, we select the following timesteps not even in time: $t \in \{0, 75, 125, 200, 400, 999\}$. We can observe that the generalizing model struggles to retain the shape of water while it's falling.

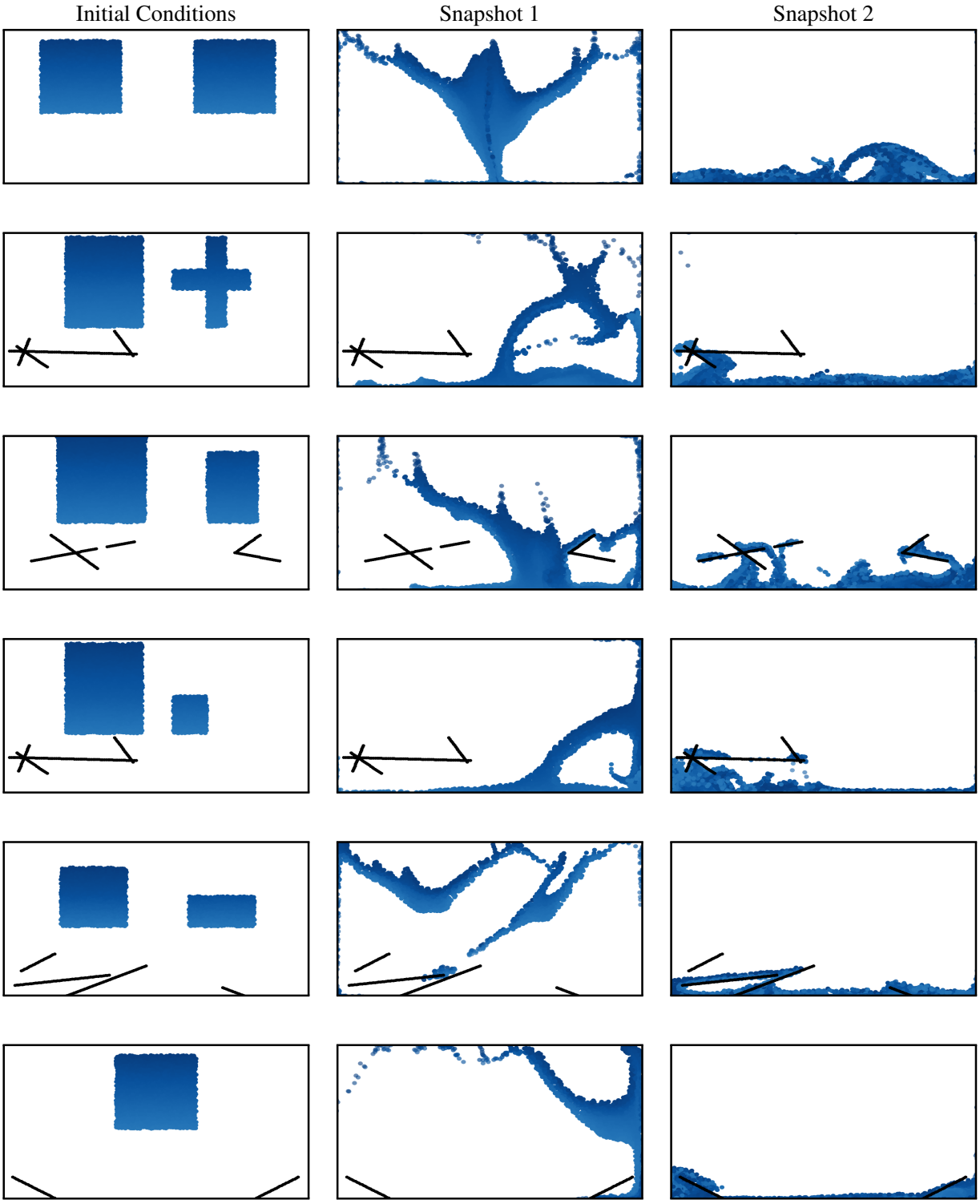

Figure 24: **Generalization to larger and non-square domains.** We train a model on the square domains in WaterRamps using 64×64 input grids to the U-Net, and then perform inference on manually generated non-square environments that are twice as wide and use a 128×64 input grid to the same U-Net. NeuralMPM flawlessly generalizes and emulates particles in these new environments. Note: no ground truth is available because the authors of GNS did not provide the physical parameters for simulating WaterRamps using Taichi. Chosen time steps are $0, 150, 575$. We recommend watching the videos in the supplementary material for more detailed evaluation.